PROGRESS®

LANGUAGE TUTORIAL

617-275-4500

Printed in U.S.A.

December 1990

CONTENTS

Figures

Tables

Preface

This book introduces you to programming in PROGRESS. It presents the basic features of PROGRESS by walking you through the development and use of a PROGRESS application.

In this book, the term PROGRESS refers to the PROGRESS 4GL/RDBMS (formerly Full PROGRESS). The PROGRESS Application Development System consists of the PROGRESS 4GL/RDBMS and the PROGRESS FAST TRACK Application Builder. For information about PROGRESS FAST TRACK, see the *PROGRESS FAST TRACK Tutorial* and the *PROGRESS FAST TRACK User's Guide.*

THE ORGANIZATION OF THIS BOOK

This book is organized as follows:

Chapter 1 — Starting PROGRESS

Describes the elements of a filing system, the structure of the demo database, and how to start PROGRESS.

Chapter 2 — The PROGRESS Editor

Introduces the capabilities and keys you can use to create and edit procedures in the PROGRESS editor.

Chapter 3 — The PROGRESS Data Dictionary

Explains the options available in the PROGRESS Data Dictionary.

Chapter 4 — PROGRESS Procedures

Describes how to use sample PROGRESS procedures and how to write your own simple procedures.

Chapter 5 — Defining An Application Database

Explains how to use the Data Dictionary to define a database.

Chapter 6 — Working with Related Files.

Explains how to write procedures that use your database definitions.

Chapter 7 — Using PROGRESS to Develop Applications

If you are developing applications for use by others, there are issues to consider before you begin your work. This chapter explains those issues and how to deal with them.

Chapter 8 — A Closer Look At PROGRESS Procedures

Chapters 2 and 3 explained the basics of data definition and procedures. This chapter builds on those basics, explaining in detail how PROGRESS statements work and how to use them to do more sophisticated processing.

Chapter 9 — Record Reading

Earlier chapters showed you how to write procedures that read database records. This chapter explains, in detail, how PROGRESS statements read records and how you can read records in several different ways depending on the kind of information you want.

Chapter 10 — Using Variables, Expressions, Functions, And Arrays

The PROGRESS language provides a rich variety of application development tools. This chapter provides an overview of those tools and explains how to use the most commonly used ones.

Chapter 11 — Understanding and Modifying Screen Designs

Procedures you wrote and used in earlier chapters mostly used PROGRESS defaults for screen formatting. This chapter explores how to use the PROGRESS language to design your own screens.

Chapter 12 — Writing Reports

Once you have an understanding of how to design screens in PROGRESS, writing reports is simply the next step. This chapter explains how to write reports, from the simple to the sophisticated.

Chapter 13 — Compiling and Precompiling Procedures

This chapter explains how PROGRESS compiles procedures and how you can use compilation to make your application run faster.

Chapter 14 — Using Procedures As Building Blocks

Previous chapters showed how to use individual procedures to do the work of your application. This chapter shows how to combine procedures and use them together to do more structured application development.

Chapter 15 — Providing Help Information for an Application

This chapter explains how to build an on-line help facility for your application.

Chapter 16 — Preparing Your Application for Use

This chapter describes the options you have for packaging and distributing an application to end users.

Appendix A — Demo Database Files, Fields, and Indexes

Lists information about the fields in the customer, order, order-line, and item files of the demo database.

TYPOGRAPHICAL CONVENTIONS

This document uses the following typographical conventions:

- **Bold typeface** indicates commands and characters you type. It also emphasizes important points.
- *Italic typeface* indicates a parameter or argument you supply. It also introduces new terms and manual titles.
- `Typewriter typeface` indicates system output and PROGRESS procedures. It also highlights file names, field names, command names, and menu options in text.

THE SAMPLE PROCEDURES

This book uses many sample procedures to show you how to use PROGRESS. All procedure examples appear in a shaded box with the name of the procedure in the upper right corner. For example:

t-demo1.p

```
FOR EACH customer:
   DISPLAY Cust-num name max-credit.
END.
```

The procedure name is the name of the file in which the procedure is stored. All procedure names start with a "t" for "tutorial" and end with a ".p" for procedure.

In the sample procedures, words that are part of the PROGRESS language are in uppercase letters and words you supply are in lowercase letters. When you write procedures you can mix uppercase and lowercase in whatever way you want.

You can find all of the sample procedures for the Programming Handbook in the proguide subdirectory of the directory where you installed the PROGRESS software (by default, `\DLC\PROGUIDE` under DOS and OS/2, `[sys] <dlc>` under BTOS/CTOS, `/usr/dlc/proguide` under UNIX, and `[DLC.PROGUIDE]` under VMS). These procedures are stored in a "packed" format in a file named `PHPROC`. This packed format ensures that the many small procedures take up as little disk space as possible. You can "unpack" all of the Programming Handbook procedures into individual procedure files, or you can extract specific procedures one at a time.

To unpack these *procedures*, access the PROGRESS Procedure Library from the PROGRESS Help menu, as follows:

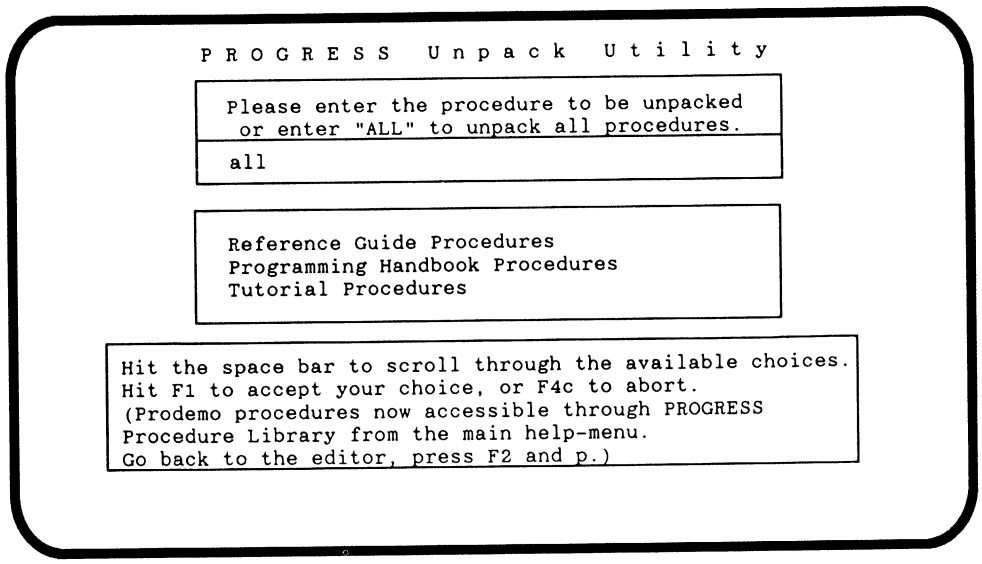
1. Press the HELP (F2) key to display the PROGRESS Help menu.

2. From the Help menu, select option **p**, Access the Procedure Library. The Main Menu of the PROGRESS Procedure Library appears on the screen.

3. From the Library Main Menu, select option **h**, Help, for an overview of the Library contents and instructions on how to use the Library.

From the Main Menu of the PROGRESS Procedure Library, you can also access the procedures used in the *Programming Handbook*, and the *PROGRESS Language Reference* by choosing option **u**, Unpack Utility.

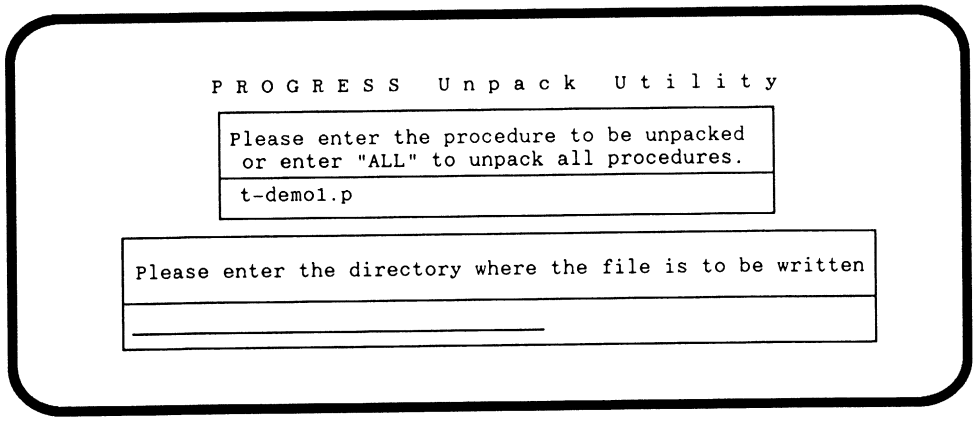
4. When you are prompted for the name of the procedure to unpack, enter **all** and press RETURN . The procedure prompts you to choose the book whose procedures you want to unpack. Choose "Tutorial Procedures."



You are then asked to enter the name of the directory where you want the files to be written. If you enter blanks, they will be written to your working directory. If you unpack the procedures into the proguide directory, all users can easily access them.

- To unpack a *specific procedure*, when you are prompted for the name of the procedure to unpack, type the name of the procedure you want to use (e.g. `t-demo1.p`).

  To put the procedure in your working directory, enter blanks when prompted for the name of the directory where you wish to place the procedure.



- All of the examples are based on the demo database. You make a working copy of this database by using the following commands:

Operating System	To Copy the demo Database
UNIX	prodb *database-name* demo
DOS and OS/2	prodb *database-name* demo
VMS	PROGRESS/CREATE *database-name* demo
BTOS/CTOS	PROGRESS Create Database New Database Name *database-name* Copy From Database Name demo

OTHER USEFUL PUBLICATIONS

The following is a list of other publications written by Progress Software Corporation which you may find useful:

PROGRESS Installation Notes

Contains step-by-step instructions for installing PROGRESS. Describes the prerequisites and procedures to get PROGRESS up and running on your machine.

PROGRESS Test Drive

Introduces new users to PROGRESS through a Sporting Goods distributor's inventory and order processing application.

PROGRESS Programming Handbook

Details advanced PROGRESS programming techniques. Provides more detailed information about application development with PROGRESS.

PROGRESS Language Reference

A detailed library of information on a number of PROGRESS topics. Provides descriptions and examples for each statement, function, phrase, and operator in the PROGRESS language.

PROGRESS System Administration Guide I: Environments

Explains the DOS, OS/2, UNIX, VMS, and BTOS/CTOS concepts required to run PROGRESS, and provides information about running PROGRESS on networks.

PROGRESS System Administration Guide II: General

Describes PROGRESS limits, disk and memory requirements, startup and shutdown procedures, backing up and restoring databases, and PROGRESS utilities. It also provides information about security administration, using multi-volume databases, and Roll Forward Recovery.

Pocket PROGRESS

Lets you quickly look up information about the PROGRESS language or programming environment. There is also a master index for the documentation in this quick reference.

Developer's Toolkit Manual

Explains how to use the PROGRESS Developer's Toolkit, a set of tools used to prepare PROGRESS applications for distribution.

Database Gateway Guide

Provides information about how to use the PROGRESS 4th generation programming language on different relational database management systems other than PROGRESS RDBMS.

3GL Interface Guide

Supplies information about the PROGRESS Host Language Call (HLC), embedded SQL, and the Host Language Reference (HLI). This manual also contains information on how to use the PROBUILD utility.

Chapter 1
Starting PROGRESS

The wait is over. You finally have software with which you can build powerful and complete database applications without time-consuming programming. You're in partnership with PROGRESS! In the first chapter we go over the following topics:

- Why use PROGRESS?
- How PROGRESS helps you get the job done.
- The elements of a filing system.
- How to start PROGRESS.
- How to leave PROGRESS.
- What the demo database is all about.

1.1 WHY USE PROGRESS?

You probably became interested in PROGRESS because you have a lot of information you want to organize. You also want to be able to get at your information, add to it, and change it without a lot of work and aggravation. You made the right choice with PROGRESS.

You use PROGRESS to create **applications**. An application is a computerized solution to any kind of business problem. Take All Around Sports, for example. All Around Sports supplies sporting goods to various retail outlets. One of these retailers, Second Skin Scuba, might call All Around Sports to order some scuba equipment. Before the sales rep, Spike, can fill the scuba order, she has to check in the overstuffed filing cabinet to see if Second Skin Scuba is a current All Around Sports account. If so, Spike pulls out the customer file to note the billing and delivery address.

Spike also fills out an order form. This form lists each item ordered, its price, and the grand total for the order. Second Skin Scuba will get a copy of this form when billing time rolls around. After she gets off the phone, Spike makes notes on an inventory sheet, indicating there are now fewer wet suits and buoyancy vests in inventory.

This procedure worked reasonably well while All Around Sports was a small company. But now they have grown large and all that paperwork just doesn't seem to make sense anymore. In fact, Spike has threatened to quit unless All Around Sports becomes computerized. Finally, All Around Sports gets a computer and hires a programmer to write a sales and inventory control application. The programmer knows just what to do - use PROGRESS to automate All Around Sports.

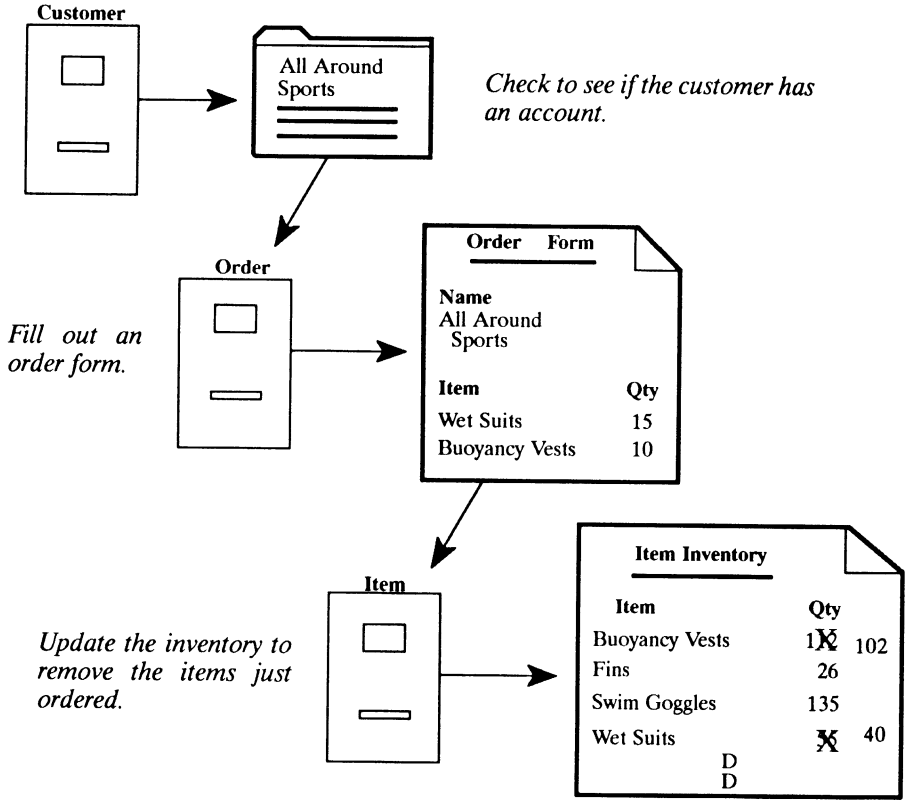


Figure 1–1: Processing an Order

Normally, applications that can handle even the simplest business problems may be difficult to develop. You usually have to know a lot about computers; you may have to know about different **file systems** or **databases**. PROGRESS makes developing applications easy!

Every application has three basic parts:

- A place to store the information.
- A way to get at the information.
- The information itself.

PROGRESS handles two of these three parts for you. That is, PROGRESS stores the information and makes it easy for you to get at it. All you have to do is supply the information.

1.2 HOW PROGRESS GETS THE JOB DONE

PROGRESS has five components that handle your information:

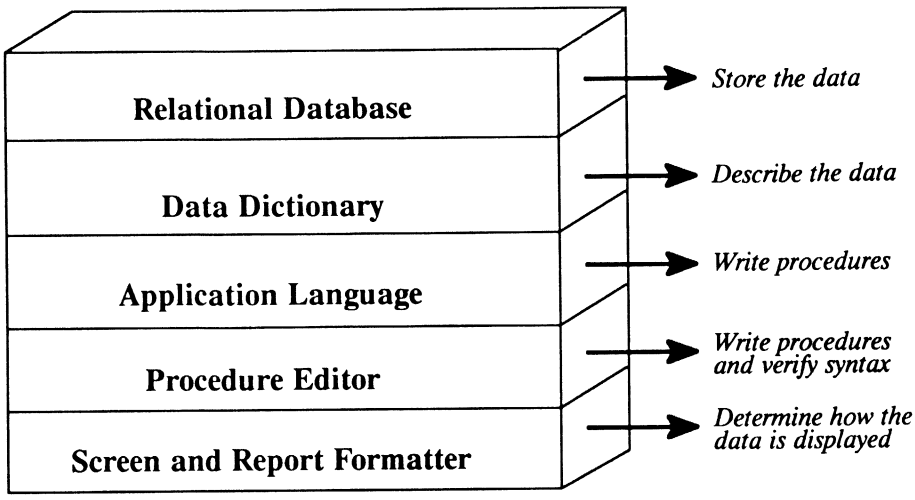


Figure 1–2: PROGRESS Components

- A **relational database** is a place to store information (like the information about the different customers, orders, and inventory items at All Around Sports).

- You use the **Data Dictionary** to describe the information you are going to store in the database. For example, All Around Sports uses the dictionary to describe customers; each has a name, address, city, state, zip code, and maximum credit limit.

- With the **procedure editor** as your writing tool, you use the PROGRESS **language** to write procedures to do the work of your application.

- PROGRESS has a built-in **screen and report formatter** so that you can create reports and order forms that look just like the paper forms you're currently using.

PROGRESS has made Spike's (and the other sales representative) job a lot easier. Now she can find the right customer record, put up a computerized order form, and update the item inventory without fiddling with paper files. All Around Sports is running more efficiently than ever before. PROGRESS can make your work a lot easier, too.

If you are already familiar with database concepts and terminology, you may want to just skim the next section. But don't forget — we're going to start using PROGRESS later on in this chapter. If you are new to the use of a computer as a filing system, take time to read the next sections carefully.

1.3 ELEMENTS OF A FILING SYSTEM

PROGRESS organizes your information, or data, in a logical way similar to the way you would organize paper files. Suppose you have many rooms filled with filing cabinets. To find a piece of data, you would have to know:

- What room to go to.
- What filing cabinet to look in.
- What folder or card to look at.
- Where to look in the folder or on the card.

When you use PROGRESS, the logic is the same but the terms are a little different, and you won't waste time rummaging around in filing cabinets.

1.3.1 A Database

When setting up a filing system, you typically place all the files that are used together in one room. You do the same thing in PROGRESS by placing all those files in a **database**. For example, if you are running a business like All Around Sports, you might have a sales order database that contains information about your customers, the number and kinds of products they've ordered, and your overall product inventory.

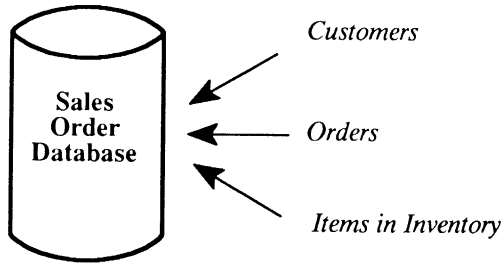


Figure 1–3: Components of Sales Order Database

1.3.2 A File

A database is made up of **files**. A file is simply a collection of information about one kind of thing. PROGRESS collects the information on a computer in much the same way as you would organize it in a filing cabinet. (Other terms for a file are "table" or "relation".)

Your sales order database might have a customer file to store each customer's name, address, city, state, zip code and maximum credit limit. It may also have a file to keep track of the orders each customer places, payment terms for those orders, and the date each order is shipped.

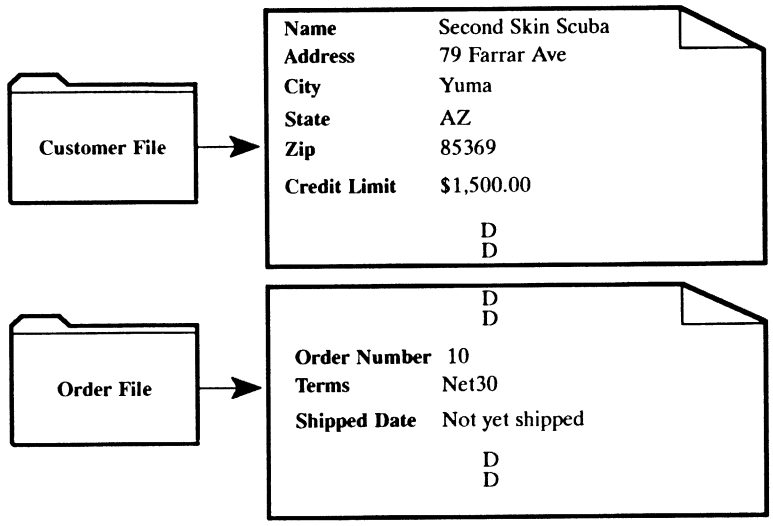


Figure 1–4: Files in the Sales Order Database

1.3.3 A Record

A file is made up of **records**. A customer file may have hundreds of records; one for every customer who orders products from you. (Other terms for a record are "row" or "tuple".)

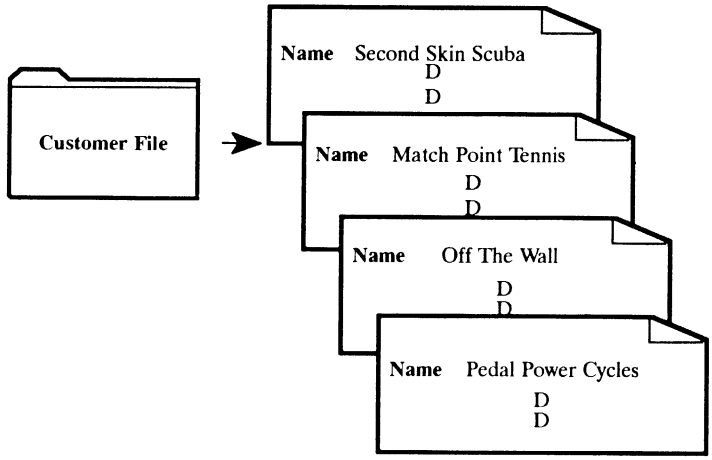


Figure 1–5: Records in the Customer File

1.3.4 A Field

A record is made up of **fields.** A field holds a value for the record. For example, the customer record might contain fields to store a customer's account number, name, address, city, state, zip, and maximum credit limit. All records in a file have the same set of fields. (Other industry-standard terms for a field are "column" or "attribute".)

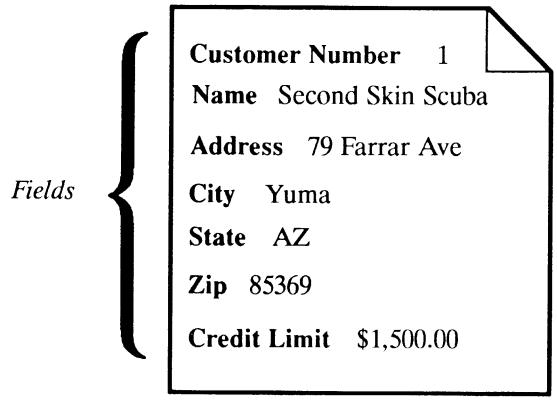


Figure 1–6: Fields in a Customer Record

1.3.5 An Index

To find a folder in your filing cabinet containing a particular customer record, you could search for it in a couple of ways. You could rummage through all of the folders (possibly hundreds or thousands) until you come across the right one. This could take a very long time.

Usually, folders in manual filing systems have an **index** tab that sticks up for easy viewing. In most cases, some identifying field, such as a customer's name, is copied onto a tab and the records are sorted by this tab. This allows you to find a particular record quickly.

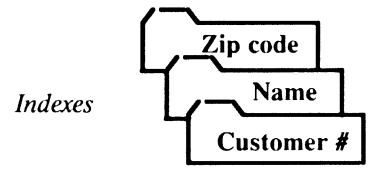


Figure 1–7: Index Tabs

PROGRESS also lets you use fields as indexes to more efficiently retrieve records. You can use a single field or a combination of fields to create an index. To decide which fields to use as indexes, you figure out how a file is usually going to be accessed. For example, you might often find a customer record by using the last name of the customer. Therefore, you can define the name field as an index of the customer file.

In addition, PROGRESS can use indexes to efficiently sort records. Let's assume that when you generate reports about your customers you usually want to list the customers in order by customer number. To do this, you make the customer number the **primary index** in your customer file. You'll learn in later chapters how to tell PROGRESS to use other indexed fields, or even nonindexed fields, to sort records any way you want.

1.3.6 Relationships Between Files

Files in a database may share information. This common information **relates** files to one another. Suppose you have a file in your database that contains information about orders placed by your customers. The records in this order file have fields containing the customer number, the number of the order, the promised delivery date, the actual shipping date, the payment terms, and whether or not the order has been shipped.

Rather than duplicate all the customer information on each order, it is common to store the customer number as a field in the order file. Because the customer number is a unique index in the customer file, you can quickly find customer information using an order's customer number. If you also define the customer number as an index for the order file, you can quickly find all orders for a given customer. In this way, the customer record is related to all the order records placed by that customer.

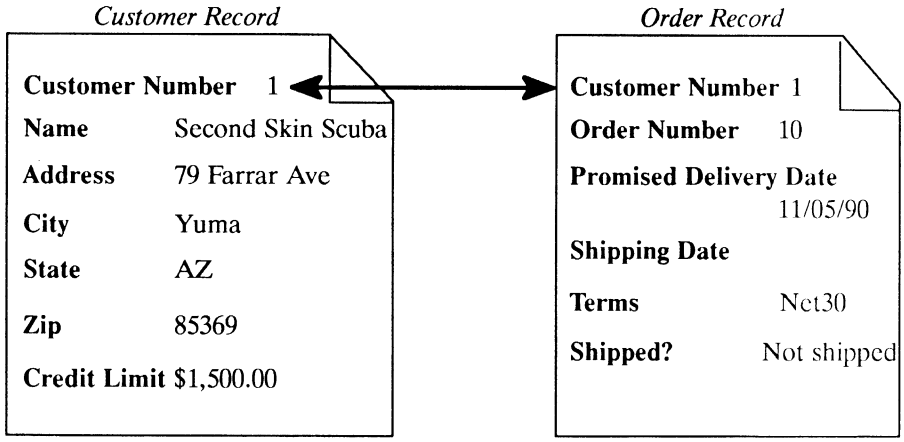


Figure 1-8: Relation Between Customer Record and Order Record

OK, let's start PROGRESS!

1.4 STARTING PROGRESS

Before you go any further, be sure you have:

- An IBM PC or compatible microcomputer running under PC-DOS or OS/2, **OR** a terminal connected to a UNIX, XENIX, BTOS/CTOS, or VMS system.
- PROGRESS installed on your system (refer to the *PROGRESS Installation Notes* instructions).

1.4.1 Starting Your DOS or OS/2 System

If you are using a microcomputer running under DOS, you'll see the hard disk system prompt on your system when you turn on your computer:

```
C>
```

If you are under OS/2, you'll see this hard disk system prompt:

```
[C:]
```

Your hard disk system prompt might be different; perhaps it is `C:` or `E>` or even `Z>`. We'll depict the hard disk prompt as `C>` in this book.

Type the following command to get into your working directory:

```
CD working-directory-name
```

If you are running under DOS, your AUTOEXEC.BAT file may automatically set your default to your working directory on your hard disk. Or if you are running under OS/2, your CONFIG.SYS file may automatically set your default to your working directory on your hard disk. In either case, if it is automatically set you do not need to type this command.

1.4.2 Starting Your UNIX System

The startup, or login procedure for UNIX varies from one system to another. The most complex login procedure may require you to enter three or four items. The simplest requires that you only enter your name.

Here is an example of a simple login procedure on a UNIX system:

```
login: your-userid

Fri Aug 31 13:49:09 EDT 1990
$
```

The dollar sign ($) is the UNIX prompt in this example. Your system may use a different prompt character or it may require that you "escape" to UNIX from a command menu after the login procedure.

1.4.3 Starting Your VMS System

The startup, or login procedure for VMS varies from one system to another. The most complex login procedure may require you to enter three or four items. The simplest requires that you only enter your name.

Here is an example of a simple login procedure on a VMS system:

```
MICROVAX VMS 4.5
USERNAME: your-userid

Today is Friday, August 31, 1990
The time is 1:49 PM
$
```

The dollar sign ($) is the VMS prompt in this example.

1.4.4 Starting Your BTOS/CTOS System

At some point you may find it necessary or desirable to start the PROGRESS server when you first turn on your system. In order to do this, you must create a small JCL (Job Control Language) file or modify an existing JCL file to tell BTOS/CTOS how to initialize PROGRESS.

In the directory in which you installed PROGRESS, there is a JCL file named "SysInit.Jcl". That file contains the following example:

```
$JOB SysInit
$RUN [sys]<dlc>PROSERV.RUN,[sys]<demo>demo,5
$END
```

In the preceding example:

- The first line, $JOB SysInit, tells the operating system that a batch job is being started.
- The second line tells the system to execute the run file named "PROSERV.RUN" on the [sys] volume in the < dlc > directory. The run file name is followed by the parameters passed to the run file. In this case, the full database name and the maximum number of users to have access to the database at one time. (The number of users is an optional parameter; it does not need to be specified in the JCL file.)
- The last line, $END, ends the JCL job.

You can modify the volume, directory, and database names (the items in lowercase) in the "SysInit.Jcl" file to fit your system. The file must be located on the [SYS] volume in the < SYS > directory to be executed at system startup.

If a system initialization file already exists on your system, you can modify that file to include the entry

```
$RUN [sys]<dlc>PROSERV.RUN,[sys]<demo>demo,5
```

For a complete explanation of how JCL works, refer to your BTOS/CTOS operating system documentation.

1.4.5 Creating a PROGRESS Database on DOS, OS/2 or UNIX

When you use PROGRESS, you work with one database at a time. If you do not have a database available to you, you must create one. You use the PROGRESS **prodb** command to create a database.

```
→ prodb
```


A Note About Using Uppercase and Lowercase Letters. DOS, OS/2, VMS, and BTOS/CTOS commands are "case insensitive." In other words, you can type a command in either uppercase letters (PRODB) or lowercase letters (prodb) and the system recognizes the command.

UNIX commands are "case sensitive." UNIX does not view PRODB and prodb as identical. While working with this manual, you should use lowercase letters when typing UNIX commands.

The prodb command asks you for the name of the database you want to create:

```
   prodb
→  Please enter the name of the new database:
```

If you are using DOS or OS/2, database names may be up to 8 characters long. UNIX database names may include up to 11 characters. Go ahead and type **mywork** and press RETURN. On a PC, the RETURN key is indicated by the keyboard symbol ↵.

```
prodb

Please enter the name of the new database:
mywork
Please enter:

demo    to get the system demonstration database, or
empty   to get the system empty database
anyname to get a copy of that database
```

You always create a PROGRESS database by copying an existing database. PROGRESS provides an empty database, which has no predefined files in it, and a demo database to acquaint you with the way PROGRESS operates. The demo database already contains files, fields, and indexes, as well as some data. You will be using a copy of this "demo" database as you work through this Tutorial.

To make a copy of the system demonstration database, type **demo** and press [RETURN]:

```
prodb

Please enter the name of the new database:
mywork

Please enter:

demo    to get the system demonstration database, or
empty   to get the system empty database
anyname to get a copy of that database.
demo
```

1.4.6 Creating a PROGRESS Database on VMS

When you use PROGRESS, you work within one database at a time. If you do not have a database available to you, you must create one. You use the PROGRESS/CREATE command to create a database:

```
PROGRESS/CREATE
```

The PROGRESS/CREATE command asks you for the name of the database you want to create:

```
PROGRESS/CREATE
Please enter the name of the new database:
```

If you are using VMS, database names may be up to 39 characters long. We'll create a database in your working directory that you can use with this Tutorial. Type **mywork** and press RETURN.

```
PROGRESS/CREATE
_New database name: mywork
_Existing database name: empty

New Progress database mywork created from empty
```


You always create a PROGRESS database by copying an existing database. PROGRESS provides an empty database, which contains no predefined files, and a demo database to acquaint you with the way PROGRESS operates. The demo database already contains files, fields, and indexes, as well as some data. You will be using a copy of this "demo" database as you work through this Tutorial.

To make a copy of the system demonstration database, type **demo** and press RETURN:

```
PROGRESS/CREATE
_New database name: mywork
_Existing database name: demo

New Progress database mywork created from demo
```

You have now copied the PROGRESS database (demo) into a new database (mywork) in your working directory.

1.4.7 Creating a PROGRESS Database on BTOS/CTOS

The PROGRESS Create Database command creates a single-volume database. To create a single-volume database, type the command **PROGRESS Create Database** at the BTOS/CTOS command line and press [GO] (F1 or CODE X). The following form appears on your screen:

```
PROGRESS Create Database
  New Database Name
  Copy From Database Name
```

The form prompts for the following values:

New Database Name Enter the name of the database file you want to create.

Copy From Database Name
Enter the name of the database whose schema and data you want to copy. If you want a clean schema to work with, type EMPTY as this parameter's value; if you want to use PROGRESS's demonstration database, type DEMO for this value.

1.4.8 Starting PROGRESS

To start PROGRESS, you use the **PROGRESS** command, followed by the name of the database you want to connect:

UNIX, DOS and OS/2:

```
→ pro mywork
```

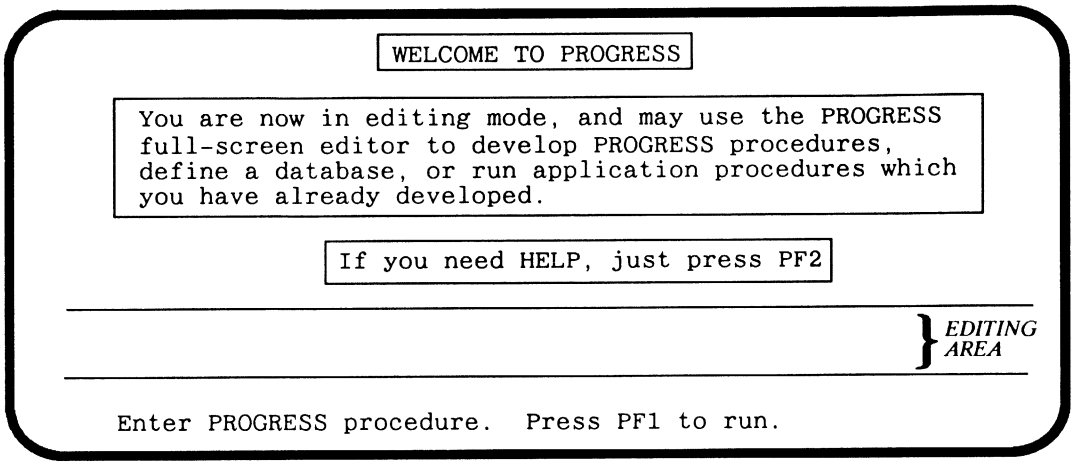
VMS:

```
→ PROGRESS mywork
```

BTOS/CTOS (single user):

```
➔ PROGRESS 4GL
     [Options] -1 mywork
```

PROGRESS displays the PROGRESS welcome banner and then displays the following screen:



You are now in the PROGRESS editor. The horizontal lines enclose the PROGRESS editing area. The editing area can expand to fill the entire screen. The PROGRESS editor is explained in greater detail in the following chapter.

For more information on starting PROGRESS and connecting databases, see Chapters 2 and 3 in *System Administration II: General*.

1.5 LEAVING PROGRESS

Anytime you are in the PROGRESS editor and want to leave PROGRESS:

1. If you were editing a procedure, you must save it before exiting or you will lose your most recent changes. You can save a procedure by pressing the F6 or PF6 key. If your terminal does not have this key, then use CTRL-P by holding down the CTRL key and pressing P. F6, PF6, and CTRL-P are all called the [PUT] key. (On BTOS/CTOS systems, use either F6 or CODE P.)

2. Clear the editor. Do this by pressing the F8 or PF8 key. If your terminal does not have this key, then use CTRL-Z by holding down the CTRL key and pressing Z. F8, PF8, and CTRL-Z are all called the [CLEAR] key. (On BTOS/CTOS systems, use either F8 or CODE Z.)

3. Type QUIT and press the F1 or PF1 key. If your terminal does not have an F1 or PF1 key, then use CTRL-X. These keys are all called the [GO] key. PROGRESS returns you to your operating system. (On BTOS/CTOS systems, use [GO], F1 or CODE X.)

Alternatively, you can leave PROGRESS from the PROGRESS help menu:

1. Press the F2 or PF2 key to display the help menu. If your terminal does not have an F2 or PF2 key, then press CTRL-W. These keys are called the [HELP] key.

2. Choose **q** from the menu. PROGRESS returns you to your operating system.

1.6 WHAT THE DEMO DATABASE IS ALL ABOUT

The demo database that you copied into your mywork database is made up of several files which comprise a sales order administration application (if this sounds familiar to you, that's because it's the same database that makes up the Test Drive application). For the purpose of this Tutorial, we'll concern ourselves with the following files and their fields and indexes:

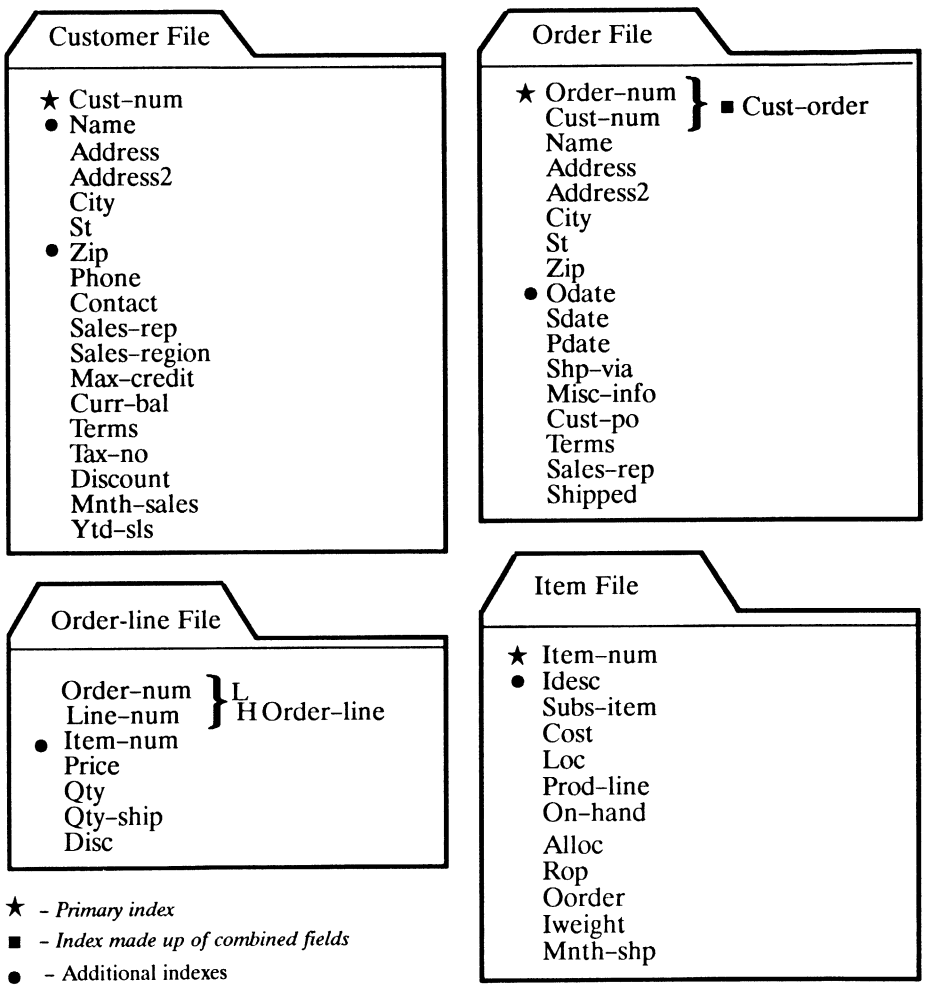


Figure 1-9: Files and Fields in the Demo Database

You'll learn more about these files and their fields and indexes as we go along. (If you'd like more information about the demo database, see Appendix A at the back of this book.)

1.7 SUMMARY

In this chapter you've been introduced to the five PROGRESS components:

- The PROGRESS Relational Database.
- The PROGRESS Data Dictionary.
- The PROGRESS Application Language.
- The PROGRESS Procedure Editor.
- The PROGRESS Screen and Report Formatter.

You have become acquainted with the following database concepts and terminology:

- A **database** is a collection of files that relate to a broad subject area.
- A **file** is a collection of records that relate to a specific subject.
- A **record** is a collection of items of information (called fields).
- A **field** is a specific item of information.
- An **index** is a field or combination of fields used to rapidly retrieve a particular record in a file.
- Files can be **related** by using an indexed field to join one file with another.

Finally, you have seen how to create a PROGRESS database and how to start and stop PROGRESS.

Lets move on to Chapter 2, where you'll learn the fundamentals of **procedures**, the foundation of all applications.

Chapter 2
The PROGRESS Editor

This chapter introduces the capabilities and keystrokes of the PROGRESS editor. PROGRESS has a full-screen editor that you can use to create and edit PROGRESS procedures. When you enter PROGRESS, you are automatically placed in the PROGRESS editor. The PROGRESS editor is fully integrated with the PROGRESS language compiler and Data Dictionary. It checks the syntax of your procedures and verifies the existence of files and fields named in your PROGRESS procedures. This chapter covers the following topics:

- Keystrokes in the PROGRESS editor.
- Getting Help.
- Entering text.
- Manipulating text blocks.
- Search and replace operations.
- Manipulating procedures.
- Leaving the PROGRESS editor.

2.1 KEYSTROKES IN THE PROGRESS EDITOR

Simple keystrokes allow you to move, delete, and copy text within a single procedure or from one procedure to another. PROGRESS assigns certain functions to various keys. If you are using UNIX, you can also reassign key functions in your protermcap file. For information about defining keys in the protermcap file, see Chapter 2 in the *PROGRESS Programming Handbook*.

Pocket PROGRESS contains a table depicting the special function keys and their standard operating system keyboard equivalents. There are special PROGRESS keys you can use when you are in the PROGRESS editor writing procedures and when you are running a PROGRESS procedure. This chapter describes only the keys used in the PROGRESS editor.

There are four ways you can use special PROGRESS keys:

- Press a single key on the terminal keyboard. For example, most terminals have [TAB].
- Press a series of keys. For example, the [ESC] [P] key sequence corresponds to the [REPAINT] key cap and refreshes the edit area of the PROGRESS editor.
- Press [CTRL], [CODE], or [ALT] and another key at the same time. [CTRL]-[X], for example, usually performs the same function as [GO].
- Press a function key. Some terminals have numbered F or PF keys. Some have both F and PF keys, and some have neither.

The following sections introduce the PROGRESS editor keystrokes. There are keystrokes for cursor movement and editing purposes.

2.1.1 Cursor Movement Keystrokes

There are several keystrokes in the PROGRESS editor that allow you to position your cursor at specific locations within the editor. These keystrokes are called cursor movement keystrokes. The table below contains all of the cursor movement keystrokes in the PROGRESS editor.

Table 2-1: Cursor Movement Keystrokes

Key	Standard Keyboard Key	Standard Control Key			Standard Function Key			Movement
		DOS OS/2	UNIX VMS	BTOS/ CTOS	DOS OS/2	UNIX VMS	BTOS/ CTOS	
BACK-TAB	SHIFT-TAB (DOS & OS/2)	CTRL-U	CTRL-U	CODE-U			CODE-TAB	Tab backwards. In the editor, tab stops are every four columns: 1, 5, 9, 13, etc.
CURSOR-UP	↑	CTRL-K	CTRL-K					Move the cursor up one line.
CURSOR-DOWN	↓	CTRL-J	CTRL-J					Move the cursor down one line.
CURSOR-LEFT	←							Move the cursor to the right one space.
CURSOR-RIGHT	→	CTRL-L	CTRL-L					Move the cursor up one line.
GOTO-LINE				CODE-G	ALT-G	** ESC, G		Move the cursor to a specified line.
HOME	HOME					ESC, H		Move to the first or last lines of the text in the editor.
LEFT-END		CTRL-←				ESC, ←	CODE-←	Move cursor to the start of the current line on DOS.
PAGE-DOWN	NEXT PAGE or PG DOWN				ALT-F6	F16		Move down one page (approximately 20 lines) in the file you are editing.
PAGE-UP	PREV PAGE or PG UP				ALT-F5	F15		Move up one page in the file you are editing.
RETURN	RETURN or NEXT (BTOS/CTOS)	CTRL-M	CTRL-M					Move the cursor to the next line.
RIGHT-END		CTRL-→ CTRL-E	CTRL-→ CTRL-E		F4	F4 ESC, →	CODE-→	Move the cursor to the right end of the current line in the PROGRESS editor.
TAB	TAB	CTRL-I	CTRL-I					Tab to different locations. In the editor, tab stops are every four columns: 1, 5, 9, 13, etc. During execution, each field on a frame is a tab position.

**Must be defined in your PROTERMCAP file.

2.1.2 Editing and Procedure Manipulation Keystrokes

The PROGRESS editor uses keystrokes to perform editing functions and manipulate procedures. These keystrokes are called editing keystrokes and they allow you to do everything from manipulating text with procedures to saving procedures, and clearing the PROGRESS editor. The following table contains all of the editing keystrokes available in the PROGRESS editor.

Table 2-2: Editing and Procedure Manipulation Keystrokes

Key	Standard Keyboard Key	Standard Control Key			Standard Function Key			Function
		DOS OS/2	UNIX VMS	BTOS CTOS	DOS OS/2	UNIX VMS	BTOS CTOS	
ABORT		CTRL -ALT -DEL	**	ACTION/ FINISH			ACTION/ FINISH	Stop the current session. On DOS, reboots the computer. On UNIX or VMS, returns you to the prompt.
APPEND-LINE		CTRL-A	CTRL-A	CODE-A	ALT-F2	F12		Join the current and next lines.
BACK SPACE	BACK SPACE							Delete the character to the left of the cursor. On some terminals (e.g., WYSE 50, WYSE 350), the backspace key sends the same code as the left arrow key. On such terminals the backspace key will not do a destructive backspace, but will move the cursor left one position
BLOCK		CTRL-V	CTRL-V	CODE-V	ALT-F4	F14	MARK or SHIFT-F4	Mark the beginning of a block of text for a move or copy operation.
BREAK-LINE		CTRL-B	CTRL-B	CODE-B	ALT-F1	F11	SHIFT-F1	Split the current line into two at the cursor.
CLEAR		CTRL-Z	CTRL-Z	CODE-Z	F8	F8	F8	Clear the editing area. If you have not saved your work, PROGRESS displays: "Discard your changes? (y/n)". If you enter no, you can save save your work. If you enter yes, the editing area is cleared and your work is not saved.
DELETE-CHARACTER	DEL DELETE (BTOS/CTOS)							Delete character at cursor.
DELETE-LINE		CTRL-D	CTRL-D	CODE-D	F10	F10	F10	Delete line the cursor is on.

** UNIX: \ (Depends on UNIX stty setting for quit)
VMS: Ctrl-Y, STOP

Table 2-2: Editing and Procedure Manipulation Keystrokes (Continued)

Key	Standard Keyboard Key	Standard Control Key			Standard Function Key			Function
		DOS OS/2	UNIX VMS	BTOS CTOS	DOS OS/2	UNIX VMS	BTOS CTOS	
END ERROR	CANCEL or ERROR (BTOS/CTOS)	CTRL-E	CTRL-E	CODE-E	F4	F4 PF4	F4	Cancel the current operation.
FIND		CTRL-F	CTRL-F	CODE-F	ALT-F3	F13	SHIFT-F3	Find a string of characters.
GET		CTRL-G	CTRL-G	CODE-G	F5	F5	F5	Retrieve a procedure and put it into the edit buffer.
GO		CTRL-X	CTRL-X	CODE-X	F1	F1 PF1	F1	Compile and run the statements in the editing area.
HELP	HELP	CTRL-W	CTRL-W	CODE-W	F2	F2 PF2	F2	Receive on-line help.
INSERT-MODE	INSERT OVERTYPE (BTOS/CTOS)	CTRL-T	CTRL-T	CODE-T	F3	F3 PF3	F3	Change from insert mode to overstrike mode or vice versa. When you are in insert mode, everything you type is inserted into the line at the cursor position. When you are in overstrike mode, the character at the cursor is replaced by the one you type. If you are in insert mode in the editor and you fill up the current line, you must press [BREAKLINE] before you can add any more characters.
NEW LINE		CTRL-N	CTRL-N	CODE-N	F9	F9	F9	Insert a blank line below the line the cursor is on and position the cursor on that line.
PUT		CTRL-P	CTRL-P	CODE-P	F6	F6	F6	Store the contents of the editing area in a file. When you press [PUT] PROGRESS prompts you for the name of the file where you want to store the edit buffer text. If you do not name a file, PROGRESS stores the contents of the edit buffer in the unnamed buffer. You can retrieve the contents of that buffer by pressing [GET] and entering a blank file name when PROGRESS prompts you for one.

Table 2-2: Editing and Procedure Manipulation Keystrokes (Continued)

Key	Standard Keyboard Key	Standard Control Key			Standard Function Key			Function
		DOS OS/2	UNIX VMS	BTOS CTOS	DOS OS/2	UNIX VMS	BTOS CTOS	
RECALL		CTRL-R	CTRL-R	CODE-R	F7	F7	F7	Recall into the edit area the last PROGRESS procedure you ran, omitting any blank lines that were at the end of that procedure when it was in the edit buffer.
REPAINT					ALT-P	ESC, P		Refresh the edit area.
RESUME DISPLAY		CTRL-Q	CTRL-Q					Restart a display that was stopped by pressing STOP DISPLAY .
SEARCH AND REPLACE				CODE-F	ALT-F	ESC, F		Search for a specified string and replace it with a specified string.
STOP-DISPLAY		CTRL-S	CTRL-S	ACTION-CANCEL				Temporarily stop the display of information on the screen.

2.2 GETTING HELP

While you're using the editor, you have easy access to HELP, an on-line reference guide which gives you detailed information about PROGRESS statements, functions, operators, keywords, the keyboard, error messages, and design limits. You can also access the PROGRESS Data Dictionary from the PROGRESS Help system.

Be sure to keep HELP in mind as you use the PROGRESS editor. Simply press HELP (F2) if you need some quick answers. To return to the editor from HELP at any time, press STOP (CTRL-BREAK on DOS systems, ACTION-CANCEL on BTOS/CTOS, usually CTRL-C on UNIX and VMS systems).

2.3 ENTERING TEXT

You enter text into the PROGRESS editor by simply typing text as you would in a word processor or any other text editor. There are two modes of text entry; insert and overstrike. When you are in insert mode, the word "Insert" appears in the lower right corner of the editor and everything you type is inserted into the current line at the cursor position. When you are in overstrike mode, the character at the current cursor position is replaced by the one you type. Overstrike mode is the default text entry mode of the PROGRESS editor. Use the INSERT MODE (F3) or OVERTYPE key to toggle between the two text entry modes.

Some characters cannot be typed directly into the edit buffer. To represent ASCII control characters or other character codes the keyboard cannot generate directly (8-bit codes, for example), type the 3-digit octal code of the character, preceded by a tilde (~). For more information about the PROGRESS character set, see Chapter 2 of the *PROGRESS Programming Handbook*.

When you write procedures with the PROGRESS editor, you may use uppercase letters, lowercase letters, or a combination of the two. PROGRESS recognizes that **find, FIND,** and **Find** are the same statement.

The basic line width of the PROGRESS editor is 80 characters. However, you may occasionally want to edit files with lines longer than 80 characters or create files with long lines. A tilde (~) as the last non-blank character on a line indicates that the line is continued beginning with column 1 of the next line on the screen. For example, these two lines in the editor:

```
DISPLAY "This is a long message ~
continued on the next line".
```

are written out to a file as this single line:

```
DISPLAY "This is a long message continued on the next line".
```

If you get a file into the editor that contains lines that are longer than 80 characters, PROGRESS splits the lines into 79 character segments and puts a tilde in column 80.

2.4 MANIPULATING TEXT BLOCKS

The PROGRESS editor supplies several keystrokes that allow you to mark blocks of text for copy and delete operations. To copy a block of lines from one part of a file to another part of a file, put the cursor at either the beginning or end of those lines and press [BLOCK] (F14). Move the cursor to the end of the block. Press [PUT] (F6) to save the block. PROGRESS prompts you for the name of the file in which you want to store the lines. You can supply a file name or blanks. If you give a file name, the block of text is placed in that file. If you enter a blank name, the text is placed in a special save area called the "temporary file", replacing the previous contents of that file. Press [RETURN] or [GO] (F1). To copy the block in a new location in the file, move the cursor to the line above where you want to insert the text, press [GET] (F5), supply a file name or blanks, and press [RETURN] or [GO] (F1).

To cut a block of lines from one part of a file and paste that block somewhere else in the file, put the cursor at either the beginning or end of those lines and press [BLOCK] (F14) or [MARK] (BTOS/CTOS). Move the cursor to the other end of the block. Press [DELETE LINE] (F10). PROGRESS removes the lines and stores them in the temporary file. Move the cursor to the line above where you want to insert the lines, press [GET] (F5), supply a blank file name, and press [RETURN] or [GO] (F1). PROGRESS copies the block of lines back into the procedure.

2.5 SEARCH AND REPLACE OPERATIONS

To perform a search and replace operation on a procedure in the PROGRESS editor, use the SEARCH key. When you press this key, PROGRESS prompts you to enter the string for which you are searching. The search is case insensitive; the string you supply can be in upper or lowercase or both. You cannot use wildcard characters when specifying the search and replace strings. After you enter the string and press RETURN or GO (F1), PROGRESS prompts you to enter a replacement string. After you finish entering the replacement string and press RETURN or GO (F1), PROGRESS finds the first occurrence of the string in the file below the current cursor location.

When a match is found for the search string, PROGRESS prompts you to enter Y, N, or G to designate the type of replacement operation. If you enter G (Global), PROGRESS replaces all occurrences of the search string with the replacement string in the current procedure. If you enter Y (Yes), the current occurrence of the search string is replaced with the replacement string and PROGRESS finds the next occurrence of the search string. If you enter N (No), the current occurrence of the search string is not replaced and PROGRESS finds the next occurrence of the search string. The search automatically wraps from the end of the file to the beginning of the file. Use the END-ERROR (F4) to cancel the search and replace operation and return to the PROGRESS editor.

2.6 MANIPULATING PROCEDURES

Along with the various editing keys listed in Table 2–2, there are several keys that allow you to manipulate whole procedures. These keys allow you to retrieve an existing procedure into the editor and also save procedures. The following sections introduce these keys and tell you how to use them.

2.6.1 Storing Procedures As ASCII Files

You will often use a particular sequence of statements, or a procedure, over and over again. Rather than write the procedure each time you want to use it, you can store it as a regular ASCII operating system file and retrieve it at any time. Because procedures are stored as ASCII files, you can use other editors to write procedures but you can use only the PROGRESS editor to run them.

Using procedure files when you work with PROGRESS provides many benefits beyond simply minimizing your typing:

- A complete procedure developed with PROGRESS can be used by people who have no knowledge of PROGRESS at all. You can learn PROGRESS, save the necessary statements as procedures, and have other personnel use these procedures with very little instruction.

- Procedure files can call other procedures. This lets you create menus that tie several procedures together into a complete application system that is easy to use.

- You can include procedures in other procedures. An operation such as displaying an order form might be used by six or seven related procedures. If you save as a procedure the statements that display the form, you can simply include that procedure in other procedures.

You store a procedure as an ASCII file by pressing PUT (F6) and supplying the name of the file you want to store the procedure in. The following prompt appears at the bottom of the editor:

```
Type the name of the file:   t-demo1.p
Or blank for temporary file
Press F1 to save file, or F4 to escape.
```

Enter a file name. You should give the procedure file the .p extension. Letter case is important to consider when you store or retrieve a procedure file. Remember, both procedures and databases are stored as operating system files. UNIX file names are case sensitive. That means that UNIX treats t-demo1.p and T-DEMO1.P as separate files. You don't have to worry about letter case with DOS, BTOS/CTOS, and VMS files. DOS, BTOS/CTOS, and VMS treat uppercase and lowercase file names in the same way. Unless you specify a pathname when you name the file, PROGRESS stores the file in your current working directory.

Press the GO (F1) key after you type the name of the procedure to save the procedure. You can also use the END (F4) key to cancel the save operation and return to the PROGRESS editor.

2.6.2 Retrieving Procedures to Edit

To edit or run an existing procedure in the PROGRESS editor, use the GET (F5) key. When you select this key, the following prompt appears at the bottom of the PROGRESS editor:

```
Type the name of the file:   t-demo1.p
Or blank for temporary file
Press F1 to retrieve file, or F4 to escape.
```

Enter the name of the PROGRESS procedure that you want to bring into the edit buffer. If the procedure is no located in the current working directory, you must enter the absolute pathname of the procedure. Again, UNIX file names are case-sensitive.

Press the GO (F1) key after you type the name of the procedure to retrieve the procedure. You can also use the END (F4) key to cancel the retrieval and return to the PROGRESS editor.

PROGRESS also allows you to recall the procedure or statements that were in the edit buffer prior to the current procedure. Press the RECALL (F7) key and the previous procedure automatically appears in the edit buffer.

2.7 LEAVING THE PROGRESS EDITOR

To leave the PROGRESS editor and return to your operating system prompt, enter the following statement into the PROGRESS editor:

```
QUIT


Enter PROGRESS procedure.  Press F1 to run.
```

Press the GO (F1) key to execute the statement and your operating system prompt appears.

Chapter 3
The PROGRESS Data Dictionary

In Version 6, the PROGRESS Data Dictionary is rebuilt with a new user interface and a variety of new features. This chapter describes the Data Dictionary and describes the following features:

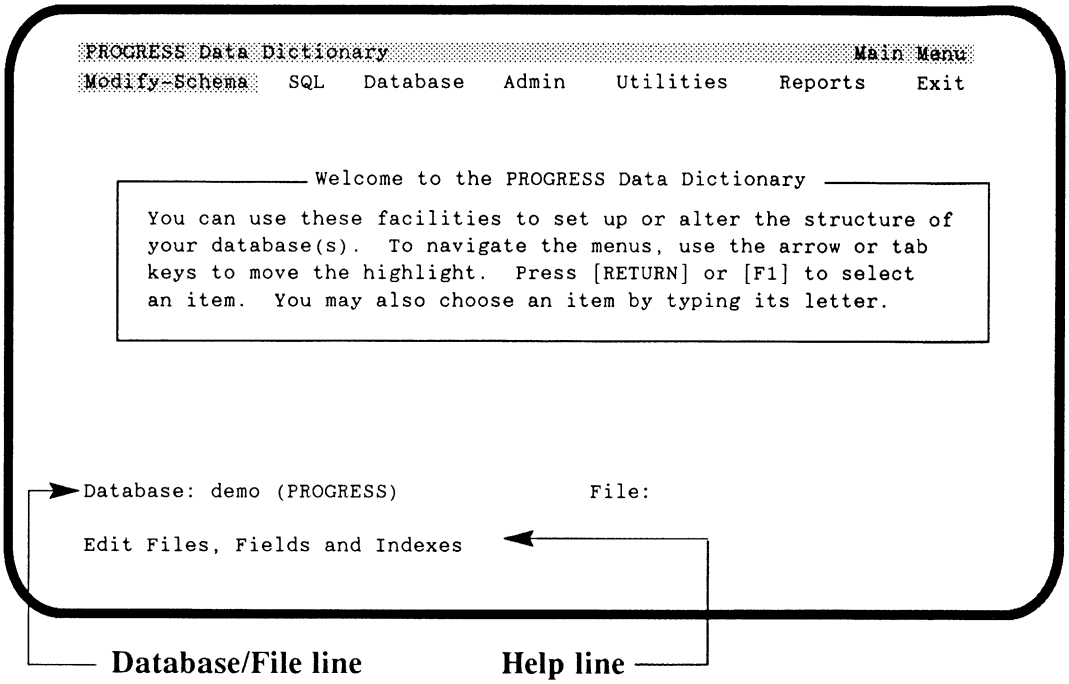
- `Modify-Schema` submenu
- `SQL` submenu
- `Database` submenu
- `Admin` submenu
- `Utilities` submenu
- `Reports` submenu
- `Exit` choice

3.1 INTRODUCTION

The PROGRESS Data Dictionary is an application program written in PROGRESS. Use it to:

- Define and modify your database structure or *schema*--in particular, to add or modify file, field, and index definitions.
- Create and connect databases.
- Dump and load (export and import) database files.
- Implement a database security strategy.
- Perform various system administration and other miscellaneous tasks.

Enter the Data Dictionary by typing `dict` and pressing GO (F1 or CTRL-X) in the procedure editor, or by pressing **D** at the PROGRESS Help main menu. When you enter the Dictionary, the following Main Menu screen appears.



The *database/file line* displays:

- the active database (demo)
- the database type (PROGRESS)
- the active file (none)

The *help line* explains or elaborates the highlighted menu option. During any operation, the help line generally offers basic information on possible actions to take.

At any one time, you have one active database and, during file, record, and index operations, one *active file*. From the Dictionary, you can connect to another database, or you can make another database the active one. Until you reset it, the active file stays active for all field and index operations; this permits you to move laterally from the field editor to the index editor without backing up and specifying a file again.

NOTE: If you start PROGRESS without naming a database (for example, `pro -1 25`) and then enter the Data Dictionary, you see an abbreviated version of the Main Menu. (Only the `Database`, `Utilities`, and `Exit` choices are available.) The `Database` pull-down submenu only lets you do two things: connect to a database or create a new PROGRESS database. The `Utilities` submenu lets you edit parameter files or run the `quoter` program to format input files. Once you connect a database, the full Dictionary Main Menu appears with all the functions available.

3.1.1 Terminology

Several terms that may be unfamiliar to you appear in the Data Dictionary documentation and in the menu choices. Therefore, it is helpful if you learn the descriptions of the following terms.

Active database — The database that you are working on in the Data Dictionary program. If you have more than one database, the Data Dictionary requires that you select an active, working database.

Connected database — A database must be *connected* before you can work on it in the Data Dictionary. Connecting to a database is similar to starting it. If you specify a database in the `pro` command (`pro -1 25 demo`, for example), it connects automatically. The various connection methods and strategies are documented in Chapter 2 of *System Administration II: General* and in Chapter 13 of the *PROGRESS Programming Handbook.*

Data Definitions — The file, field, and index definitions that comprise the database structure. You use the Data Dictionary to add, modify, and delete data definitions. They are described in detail in Chapter 5.

Data Dictionary — This term has two meanings. When capitalized, it refers to the menu-driven utility program documented in this chapter. In lower case, *data dictionary* is a synonym for *schema*; that is, it refers to the actual database structure definitions. Typically in PROGRESS—always in this chapter—Data Dictionary refers to the PROGRESS program, while "schema" and "data definitions" are the terms used to refer to the database structure.

Gateway — A PROGRESS program module that lets you access a non-PROGRESS database from a PROGRESS application, including the Data Dictionary. Gateways currently exist for ORACLE and RMS databases.

Schema — Same as Data Definitions description.

Schema-holder — A PROGRESS database that contains the schema or data definitions for one or more non-PROGRESS databases. Non-PROGRESS databases are documented in the *Database Gateways Guide.*

Table — Database file. SQL statements refer to "tables" rather than "files." For more information, see Chapter 15, "PROGRESS/SQL," in the *PROGRESS Programming Handbook*.

View — A window into one or more tables (files). For more information, see Chapter 15, "PROGRESS/SQL," in the *PROGRESS Programming Handbook*.

3.1.2 Moving Around in the Dictionary

The Data Dictionary has a horizontal Main Menu with "pull-down" submenus. Select a menu item by highlighting it with the arrow or TAB keys and then pressing RETURN, or by typing its first letter.

The Data Dictionary is an application program written in PROGRESS. Therefore, the prompts or forms you fill out as you use the Dictionary are ordinary PROGRESS frames. In PROGRESS, use the GO key (F1 or CTRL-X) to pass the data you enter in a frame to PROGRESS, and press END (F4 or CTRL-E) to cancel a frame or menu.

To help you remember where you are in a menu selection, you can read the pull-down menu choices that appear in the screen's upper right corner until you complete or cancel the operation. This information is helpful if you get interrupted in the middle of a procedure.

You can use GET (F5 or CTRL-G) to change the active file from either the field or index editors in the Dictionary. For example, you can change the active file from the field editor to begin modifying fields in a second file without ever leaving the field editor.

The next section includes a brief tutorial that helps you practice moving around in the Dictionary.

The remaining sections describe the Main Menu and submenu options available in the Data Dictionary. For many tasks, you are referred to other documentation or sections for more complete descriptions. For example, you are referred elsewhere for information about modifying file/field definitions and connecting to a non-PROGRESS database. Use material in this chapter to learn the different kinds of things you can do in the Data Dictionary, and to learn where you can find any detailed information your work requires.

3.2 THE MODIFY-SCHEMA SUBMENU

The Data Dictionary is mainly a tool for modifying the database structure or *schema.* Use the `Modify-Schema` option to add, update, or delete file, field, or index definitions. Detailed information on the file, field, and index definitions that comprise the schema is in Chapter 5.

This section is a brief tutorial. The steps do not accomplish a particular task; they are intended to demonstrate the screens and keystrokes you use to create and modify a database schema.

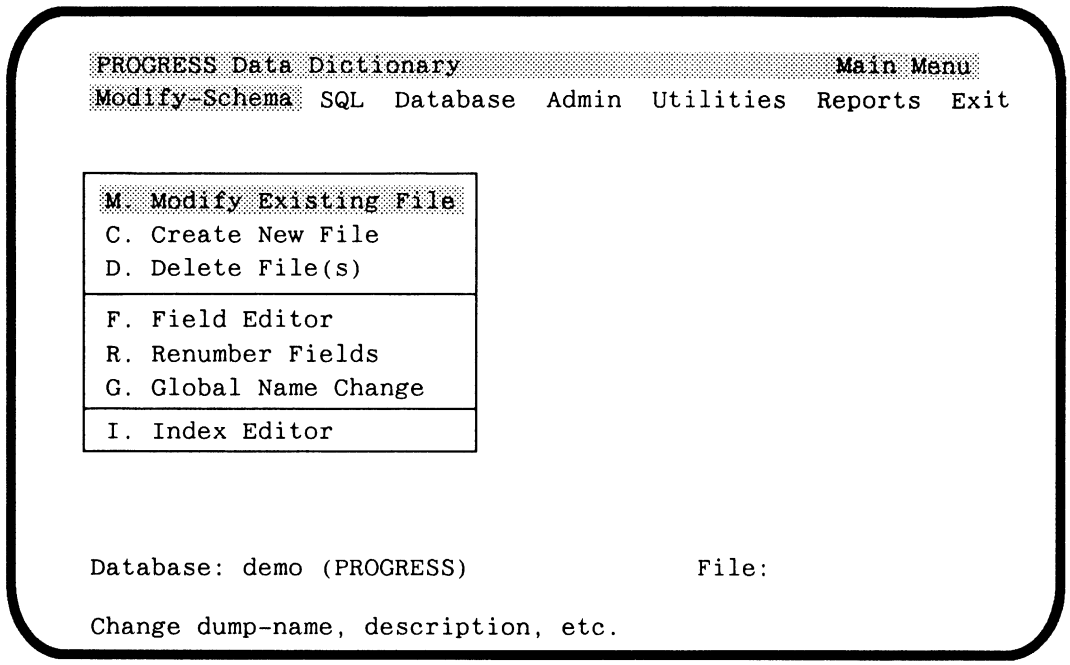
1. Start the PROGRESS demo database in single-user mode with a command like the following:

```
pro  -l 25  demo
```

2. Type "dict" in the procedure editor, and press GO to enter the Data Dictionary. The following screen appears:

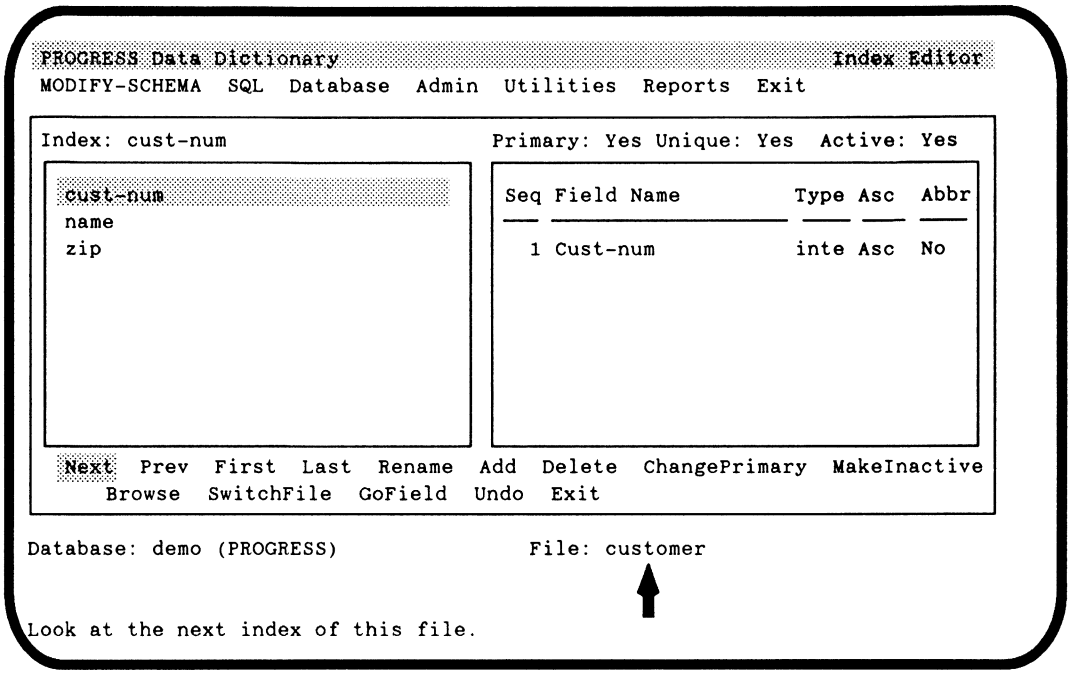
```
PROGRESS Data Dictionary                          Main Menu
Modify-Schema  SQL Database Admin Utilities  Reports  Exit

        Welcome to the PROGRESS Data Dictionary
  You can use these facilities to set up or alter the structure of
  your database(s).  To navigate the menus, use the arrow or tab
  keys to move the highlight.  Press [RETURN] or [F1] to select
  an item.  You may also choose an item by typing its letter.



 Database: demo (PROGRESS)              File:

Edit Files, Fields and Indexes
```

3. Press RETURN and select Modify-Schema from the Data Dictionary Main Menu. The following screen appears:



4. Press **I** to select Index Editor. The screen prompts you to select a single working file. (Note that this version of the file selection list permits you to enter a single file name—no "All" or star names. Also note that although hidden files are not displayed in this list, you can still select one by typing its full name and pressing RETURN.)

5. Type **c** to select the file customer and press [GO] or [RETURN]. The following screen appears:



6. Press [→] several times to move the shaded bar horizontally along the Index Editor menu.

7. Press **n** to select Next. This highlights the name index. Alternatively, use the arrow or [TAB] keys to highlight the Next option, and then press [RETURN] or [GO] .

8. Press **f** (First). The screen displays the first index of the file. Now press **l** (Last). The screen displays the last index of the file. Press **n** (Next). The screen does not change, because it was already at the last index in the file.

9. Press **g** (GoField) to switch to the Field Editor. The following screen appears:

```
PROGRESS Data Dictionary                                    Index Editor
MODIFY-SCHEMA  SQL  Database  Admin  Utilities  Reports  Exit
                          Currently Defined Fields
Address        Address2    City          Contact     Curr-bal     Cust-num
Discount       Max-credit  Mnth-sales    Name        Phone        Sales-region
Sales-rep      St          Tax-no        Terms       Ytd-sls      Zip




NextPage  PrevPage  Add  Modify  Delete  Copy  GoIndex  SwitchFile
   Browse  Order  Undo  Exit
                                                        Total Fields: 18

Database: demo (PROGRESS)                File: customer

See the next page of fields
```

The customer file is still the active file. Therefore, from this screen you can modify field definitions for that file.

10. Press **m** (Modify). Then type **n**, or use the arrow keys, to highlight the Name field. Then press RETURN. The Dictionary displays the following screen, which shows the field definitions for the Name field and which highlights the parts of the name field definition that you can change:

```
PROGRESS Data Dictionary                                    Index Editor
MODIFY-SCHEMA  SQL  Database  Admin  Utilities  Reports  Exit
                          Currently Defined Fields
Address        Address2    City         Contact      Curr-bal      Cust-num
Discount       Max-credit  Mnth-sales   Name         Phone         Sales-region
Sales-rep      St          Tax-no       Terms        Ytd-sls       Zip


  Field-Name: Name                          Data-Type: character
      Format: x(20)                            Extent:
       Label: Name                           Decimals: ?
Column-label: ?                                 Order: 20
     Initial:                                Mandatory: no   (Not Null)
Component of-> View: no    Index: no       Case-sensitive: no
Valexp: ?
      :
      :
      :
Valmsg:
  Help:
  Desc:

  NextPage  PrevPage  Add  Modify  Delete  Copy  GoIndex  SwitchFile
    Browse  Order  Undo  Exit                              Total Fields: 18
Database: demo (PROGRESS)              File: customer

Enter data or press F4 to end.
```

11. Rather than change the definition at this point, press END (F4 or CTRL-E) to cancel.

12. Press **s** (SwitchFile). Type **order** and press RETURN (or GO). You remain in the Field Editor, but order is now the active file.

13. Press **g** (GoIndex). You are now in position to modify any indexes for the file order.

14. Press END (F4 or CTRL-E) to exit the Index Editor. The Data Dictionary displays a menu that allows you to either apply or undo you changes, or to return to the editing files, fields, or indexes.

15. Suppose you want to update the file definitions for `order`, currently the active file. Press **m** (`Modify Existing File`). The order field is already highlighted, so just press RETURN (or GO). The following screen appears:

```
PROGRESS Data Dictionary                                    Index Editor
MODIFY-SCHEMA  SQL  Database  Admin  Utilities  Reports  Exit

  File Name: order                                  Hidden: no
  File Type: PROGRESS                               Frozen: no
 File Owner:                                    File Number: 5
  Dump Name: order     (unique name for data dump into .d file)

Description: Order header information
             ____________________________________________________________
             ____________________________________________________________
             ____________________________________________________________

Enter the message to be displayed for disallowed deletions.
     ValMsg: Cannot delete Order, Order-line records still exist
Enter an expression which must be TRUE to allow record deletions.
     ValExp: NOT (CAN-FIND(FIRST order-line OF order))
             ____________________________________________________________
             ____________________________________________________________
             ____________________________________________________________


Database: demo (PROGRESS)              File: order

Enter data or press F4 to end.
```

16. Press END (F4 or CTRL-E) to exit from this screen. The Data Dictionary again displays the menu that allows you to either apply or undo you changes, or to return to the editing files, fields, or indexes. This time, press **a** (Apply Changes) to apply your changes and exit to the Data Dictionary.

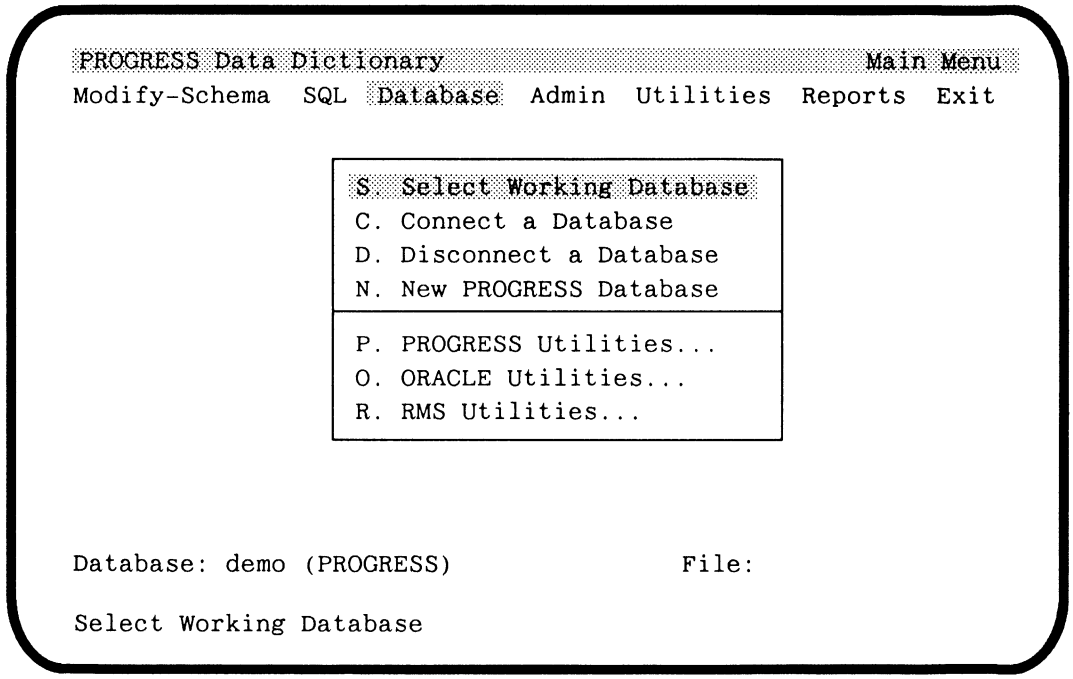
3.3 THE SQL SUBMENU

If you are using an ORACLE database, or if you are using SQL statements to access any database, be familiar with the SQL submenu (shown below). The first two options involve views created with PROGRESS/SQL statements, and the next two provide an easy way to generate SQL language CREATE VIEW and CREATE TABLE statements. For details on these options, see Chapter 15, "PROGRESS/SQL," in the *Programming Handbook*.

```
PROGRESS Data Dictionary                                    Main Menu
Modify-Schema  SQL  Database  Admin  Utilities  Reports  Exit

          B. Browse Views
          S. View Summary
          V. Dump as CREATE VIEW statements
          T. Dump as CREATE TABLE statements




Database: demo (PROGRESS)                    File:

Look at the structure of defined views.
```

3.4 THE DATABASE SUBMENU

Use the Database submenu (shown below) to create, connect, and disconnect databases. If you have more than one connected database, press **S** to select a working database. Note that if you name a database with the pro command, that database is connected and is the working database. Also, if you press **C** and connect a database, the Dictionary forces you to select a working database automatically. Database connection/disconnection is an important issue if you are working with multiple databases. For more information, see Chapter 13, "Multi-Database Programming," in the *Programming Handbook* and Chapter 2, "Startup and Shutdown," in *System Administration II: General.*



Note that PROGRESS appears in upper-case letters in the database/file line. This indicates that the database is connected. The database type (oracle, for example) appears in lower case when the active database is not connected. PROGRESS is always upper case, because PROGRESS databases are always connected.

3.4.1 PROGRESS Database Utilities

Select "P. PROGRESS Utilities" to run any of the three utilities for PROGRESS databases only. (The Utilities submenu contains database utilities common to all database types.) The following menu appears:

```
I. Create Incremental .df File
C. Convert V5 .df into V6 .df
D. Dump Data Definitions in Old V5 Format
E. Edit PROGRESS Auto-Connect List
```

The first option, Incremental Data Definitions, compares two PROGRESS database schemas and creates a .df file that contains any differences. This file can then be used to apply schema changes to an existing database. Incremental data definitions are described in detail in Chapter 9 of the *Developer's Toolkit Manual*, and in Chapter 4 of *System Administration II: General*.

The second option, Convert V5 .df into V6 .df, converts a data definition file created with PROGRESS Version 5 to the new .df file format used in Version 6. See Chapter 4 in *System Administration II: General*. It is not necessary to convert a V5 .df into a V6 .df in order to load it.

To dump data definitions from a PROGRESS Version 6 database so they can be reloaded into a database running with PROGRESS Version 5, use the third option, Dump Data Definitions in Old V5 Format. Database dump and load operations are described in Chapter 4 of *System Administration II: General*.

The fourth option, Edit PROGRESS Auto-Connect List, lets you prepare a list of PROGRESS databases (with connection parameters) that PROGRESS connects automatically as required by program execution on the first database. This list is called an *auto-connect list*, and each PROGRESS database can have one. For more information on the various methods used to connect databases, see Chapter 2 in *System Administration II: General*.

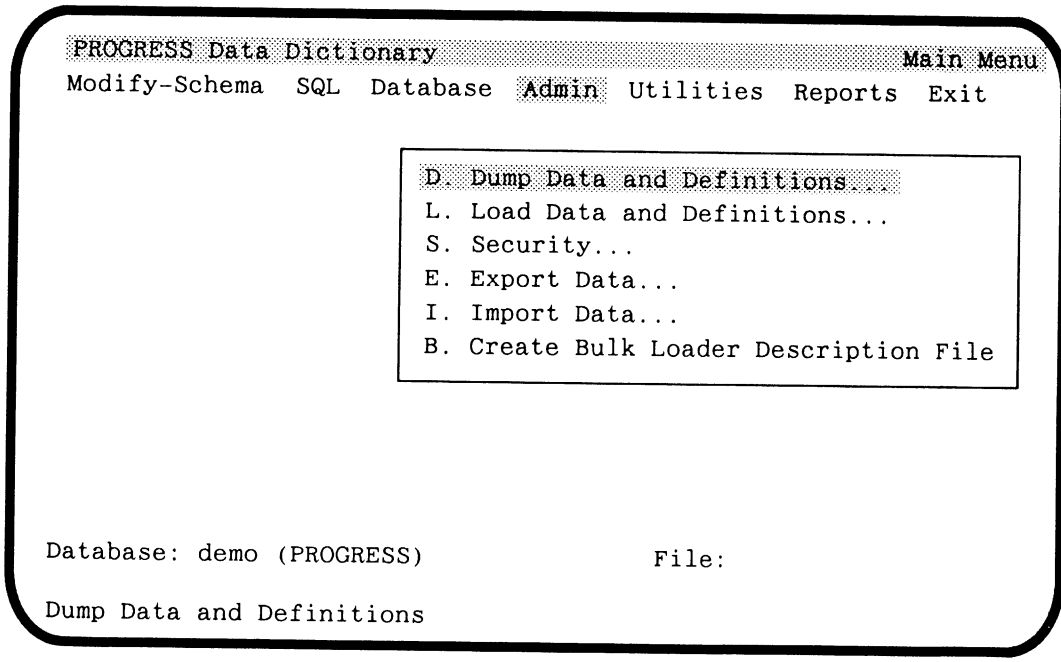
3.4.2 Non-PROGRESS Databases

Select the ORACLE Utilities or RMS Utilities options to see menus relevant to ORACLE and RMS databases running under PROGRESS. The RMS menu, for example, lets you create a PROGRESS image of an existing RMS database structure, add/change/verify RMS file definitions, change the connection parameters, or delete the PROGRESS database image. If you are using a non-PROGRESS database, you **must** use the Data Dictionary to accomplish these tasks. For more information on non-PROGRESS databases see the relevant chapters in the *Database Gateway Guide*.

3.5 THE ADMIN SUBMENU

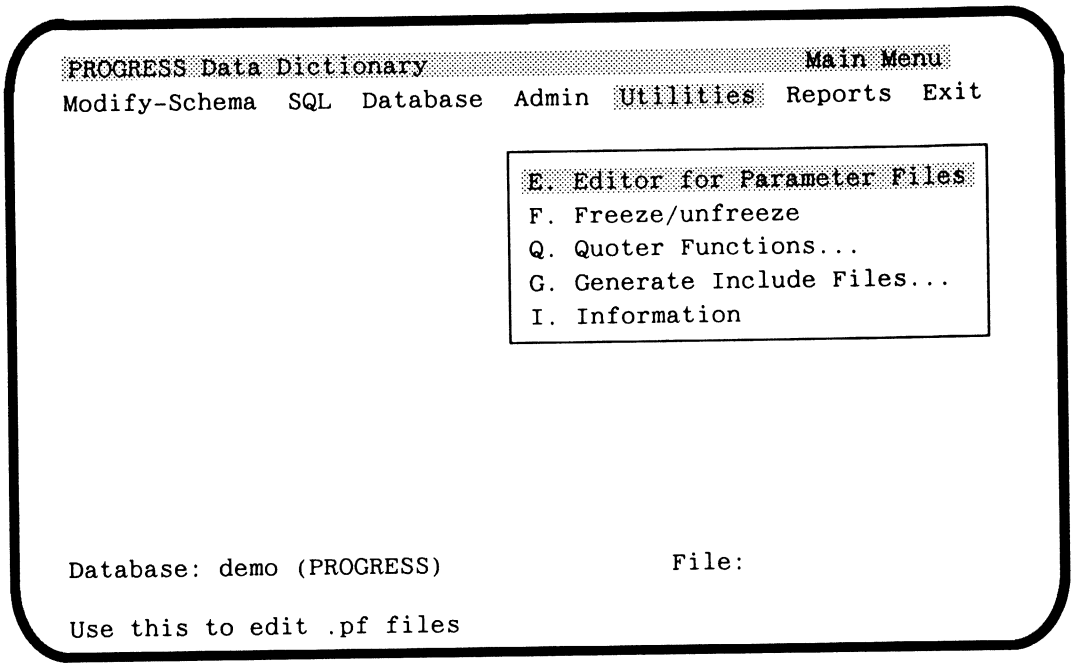
Use the Admin submenu (shown below) to dump and load database data and definitions to export and import database data and to implement a database security scheme. You can export data records in any of the following formats: DIF, SYLK, ASCII, WordStar, Microsoft Word, WordPerfect, and BTOS OfisWriter. You can import records from files having any of the following formats: DIF, SYLK, ASCII, and dBASE II/III/III + /IV.

Database dump/load procedures are documented in Chapter 4 of *System Administration II: General*. Data import/export is documented in Chapter 9 of the *PROGRESS Programming Handbook*, and the IMPORT and EXPORT statements are documented in the *PROGRESS Language Reference Manual*. For detailed information on database security, see Chapter 5 in *System Administration II: General* and Chapter 11 of the *PROGRESS Programming Handbook*.



3.6 THE UTILITIES SUBMENU

The `Utilities` submenu (shown below) provides the interface for four distinct tasks.



3.6.1 Editor for Parameter Files

You can store PROGRESS command line startup parameters together in a *parameter file* (`.pf` suffix). Select `Editor for Parameter Files` to see a complete list of startup parameters, fill in the desired values, and write out a `.pf` file (rather than typing in the parameter file with an ordinary text editor). Parameter files are described in Chapter 3 of *System Administration II: General*.

3.6.2 Freeze/unfreeze

Freezing a database file prevents users from changing the file, field, or index definitions for the file. *Frozen files* are described in Chapter 4 of *System Administration II: General*.

3.6.3 Quoter Functions

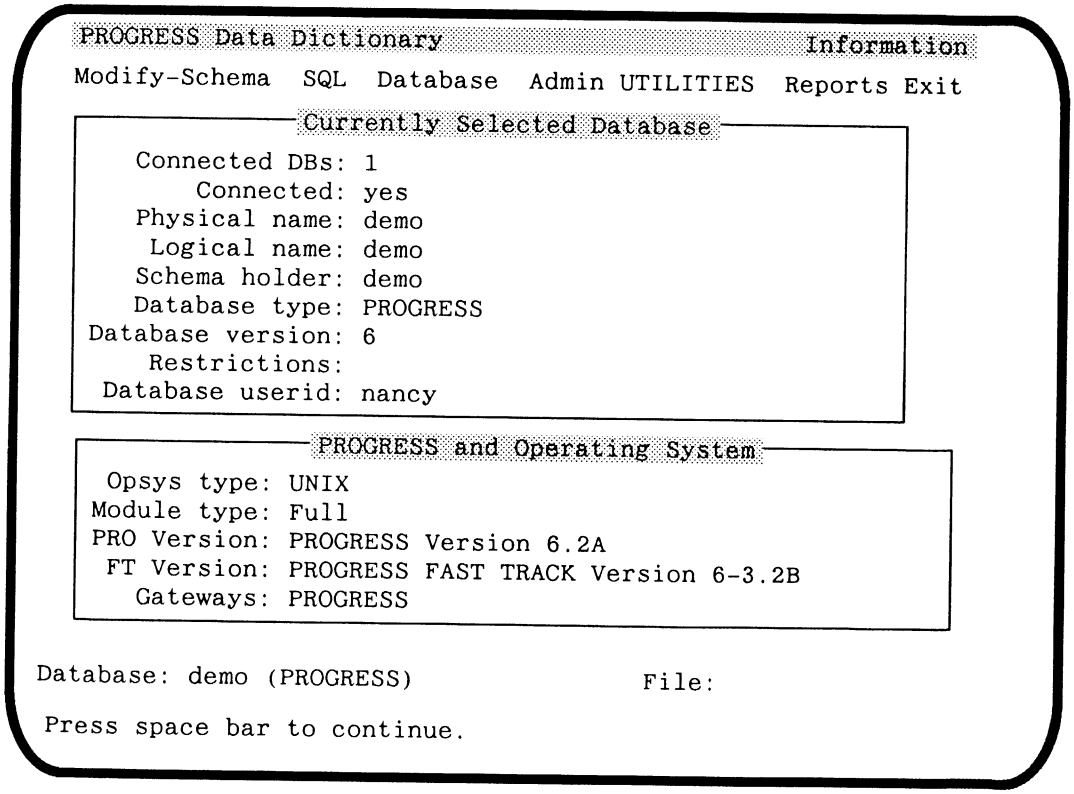
Quoter is a utility that formats a data file so that PROGRESS can read the data into a PROGRESS database. The quoter utility is documented in Chapter 4 of *System Administration II: General* and in Chapter 9 of the *PROGRESS Programming Handbook*.

3.6.4 Generate Include Files

The PROGRESS language statements ASSIGN, FORM, and DEFINE WORKFILE can be long and complicated. Also, particular instances of these statements are often repeated in different procedures. The `Generate Include Files` option lets you quickly prepare an include file to store one of these statements. An include file can be easily inserted into any PROGRESS procedure. See the following entries in the *PROGRESS Language Reference* for more information: { }, ASSIGN, FORM, and DEFINE WORKFILE.

3.6.5 Information

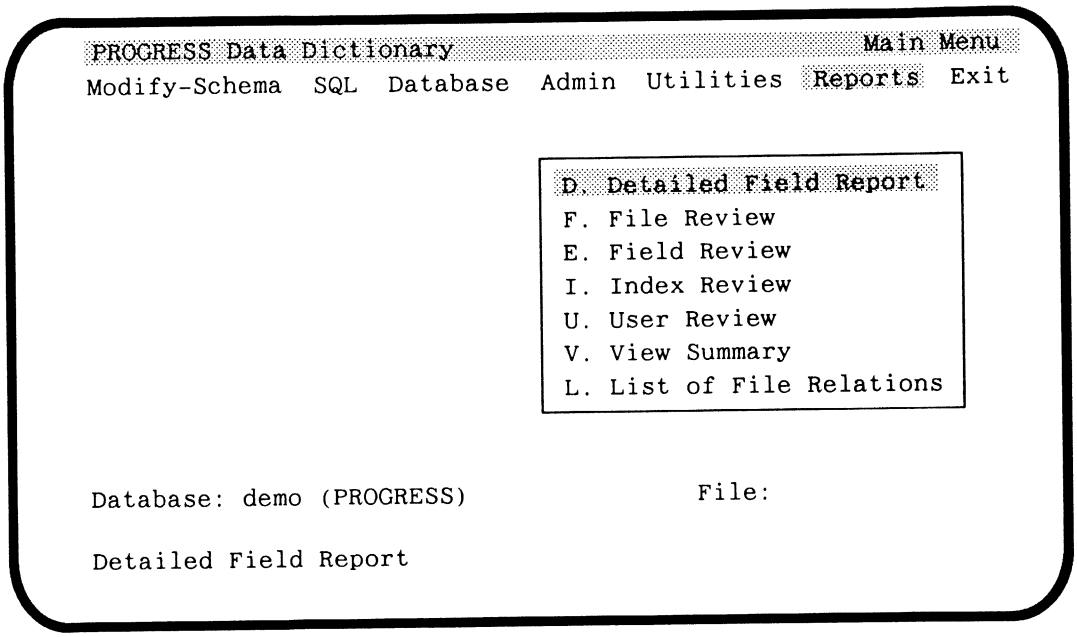
Select `Information` to display general information about the database and the PROGRESS product you are using. For example, you may see information similar to the following:



3.7 THE REPORTS SUBMENU

Use the Reports submenu (shown below) to review your database structure, to list registered database users, or to list all file relations in the database. Chapter 5 describes database design and structure (data definitions), and the file, field, and index report options. Chapter 6, "Working with Related Files," describes the List of File Relations choice. Chapter 5 in *System Administration II: General* describes the User Review choice.

Once you are familiar with PROGRESS data definitions, the output from the various options should be self-explanatory.



Note that the Detailed Field Report option does not display a report to your screen. Because of the size of the report, you must specify a printer or name a file to store the report.

3.8 THE EXIT CHOICE

Press **E** (Exit) to leave the Data Dictionary and return to the PROGRESS Help Main Menu or to the editor, depending on how you entered the Data Dictionary. Exit is equivalent to END (F4 or CTRL-E).

Chapter 4
PROGRESS Procedures

To work with information in your database, PROGRESS needs instructions. The instructions, which you write with the PROGRESS application language, are called **procedures.** Procedures act like conventional programs but are far easier to write and are much more flexible. PROGRESS procedures can run other procedures as well as execute operating system commands.

In Chapter 1, you started PROGRESS and copied the system demo database into your own database. In this chapter you will use the PROGRESS editor and the PROGRESS application language to manipulate the information stored in that database. The chapter covers the following topics:

- Using PROGRESS sample procedures.
- Using the PROGRESS editor.
- Running simple procedures.
- Grouping statements in blocks.
- Writing procedures to perform file maintenance.
- Commenting out procedures.

If you exited from PROGRESS after Chapter 1, you need to get back into your **mywork** database. At the system prompt, start PROGRESS by typing the following commands and pressing RETURN:

Table 4–1: Command to Start PROGRESS

Operating System	Starting mywork Database
UNIX, DOS and OS/2	pro mywork
VMS	PROGRESS mywork
BTOS/CTOS	PROGRESS 4GL [Options] -1 mywork

4.1 THE SAMPLE PROCEDURES

This book uses many sample procedures to show you how to use PROGRESS. All procedure examples appear in a shaded box. The name of the procedure appears in the upper right corner of the box. For example:

t-demo1.p

```
FOR EACH customer:
   DISPLAY cust-num name max-credit.
END.
```

The procedure name is simply the name of the file in which the procedure is stored. All procedure names in this book start with a "t" for "tutorial" and end with a ".p" for procedure.

In the sample procedures, words that are part of the PROGRESS language are in uppercase letters and words you supply are in lowercase letters. This is purely for the sake of teaching you the PROGRESS language; you can mix uppercase and lowercase letters in whatever way you want.

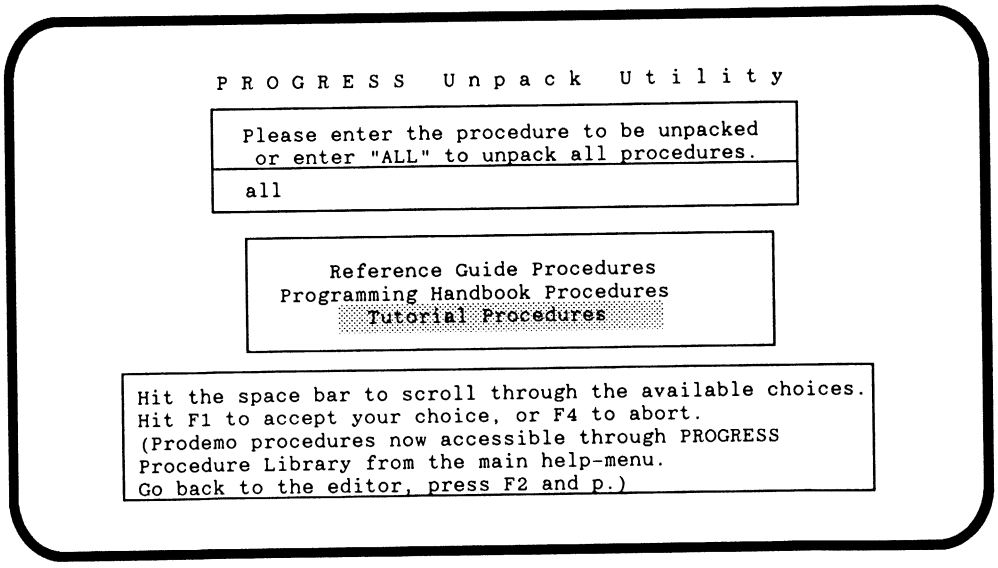
You can find all of the sample procedures for this Tutorial in the proguide subdirectory of the directory where you installed the PROGRESS software (by default, \DLC\PROGUIDE under DOS, /usr/dlc/proguide under UNIX, and [DLC.PROGUIDE] under VMS). Under BTOS/CTOS, use [sys]<dlc>PROGUIDE/, where "[sys]<dlc>" is the directory where PROGRESS is installed and PROGUIDE/ is the file prefix. These procedures are stored in a "packed" format in a file named TUTPROC. This packed format ensures that the many small procedures take up as little disk space as possible. You can "unpack" all of the Tutorial procedures into individual procedure files, or you can extract specific procedures one at a time.

To unpack these *procedures*, access the PROGRESS Procedure Library from the PROGRESS Help menu, as follows:

1. Press the HELP (F2) key to display the PROGRESS Help menu.
2. From the Help menu, select option **p**, `Access the Procedure Library`. The Main Menu of the PROGRESS Procedure Library appears on the screen.
3. From the Library Main Menu, select option **h**, `Help`, for an overview of the Library contents and instructions on how to use the Library.

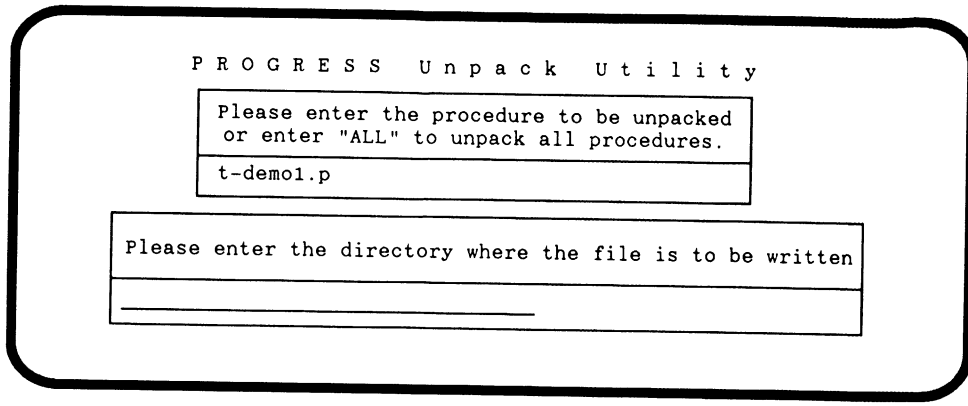
From the Main Menu of the PROGRESS Procedure Library, you can also access the procedures used in the *PROGRESS Tutorial*, the *PROGRESS Programming Handbook*, and the *PROGRESS Language Reference* by choosing option **u**, Unpack Utility.

1. When you are prompted for the name of the procedure to unpack, enter **all** and press RETURN. The procedure prompts you to choose the book whose procedures you want to unpack. Choose "Tutorial Procedures."



2. The unpack.p procedure then asks you to enter the name of the directory where you want the files to be written. If you enter blanks, they will be written to your working directory. If you unpack the procedures into the proguide directory, all users can easily access them.
3. To unpack a *specific procedure*, when you are prompted for the name of the procedure to unpack, type the name of the procedure you want to use (e.g. t-demo1.p).

   To put the procedure in your working directory, enter blanks when prompted for the name of the directory where you wish to place the procedure.



4. All of the examples are based on the demo database. You make a working copy of this database by using the following commands:

Table 4–2: Command to Create Copy of Demo Database

Operating System	To Copy the demo Database
UNIX, DOS and OS/2	prodb *database-name* demo
VMS	PROGRESS/CREATE *database-name* demo
BTOS/CTOS	PROGRESS Create Database New Database Name *database-name* Copy From Database Name demo

4.2 USING THE PROGRESS EDITOR TO WRITE PROCEDURES

You use the PROGRESS full-screen editor to write PROGRESS procedures. The PROGRESS editor has a built-in syntax checker which helps you locate errors in your procedures. The editor is fully integrated with the PROGRESS language compiler and Data Dictionary. It checks the syntax of your procedure and verifies the existence of files and fields you may have named. You can move, delete, and copy text within a single procedure or from one procedure to another. A single keystroke lets you test a PROGRESS procedure or retrieve the last procedure you ran.

4.2.1 The Keys You'll Need

You use special keys to perform different functions when you are editing and running procedures. PROGRESS assigns certain functions to various keys. In your procedures, you can reassign those functions to different keys. If you are using UNIX, you can also reassign key functions in your protermcap file. *Pocket PROGRESS* contains a table depicting the special function keys and their standard operating system keyboard equivalents. Keep this table handy as you continue working through this chapter and the rest of this Tutorial.

Because you may use different keyboard keys to perform the same function with PROGRESS, this manual uses a small box to represent a key. For example:

GO (F1)

represents the GO key. The word inside the box is the general name of the PROGRESS function that the key performs. The characters enclosed in parentheses represent the label that appears on the key on most terminals. If your terminal has PF keys, then F1 is equivalent to PF1. If your terminal has no function keys or its function keys don't seem to work, you can use control keys. For example, pressing

CTRL-X

is the same as pressing

GO (F1)

Under DOS, you use the ALT key for the function keys beyond F10. For example, F12 is the same as ALT-F2.

Under BTOS/CTOS, you use the SHIFT key for the function keys beyond F10. For example, F12 is the same as SHIFT-F2.

The keys you'll use to run the example procedures in this chapter are listed below. Don't worry about remembering all of these keys right now; we'll remind you as you go along.

GO (F1)	Runs the procedure you are currently editing.
HELP (F2)	Lets you access on-line HELP at anytime. HELP gives you information about error messages, the keyboard, statements, functions, operators, keywords, and design limits.
END-ERROR (F4)	Ends the current operation; if it is the last one in a procedure, returns you to the edit area.
GET (F5)	Lets you retrieve a procedure and bring it into the editing area when you press this key, type the name of the procedure you want, and press

RETURN. If you are using a UNIX system, type the procedure name in lowercase letters (under UNIX procedure names are case sensitive).

CLEAR (F8) Erases the current procedure (if there is one) from the edit area.

4.2.2 Storing Procedures as ASCII Files

You will often use a particular sequence of statements, or a procedure, over and over again. Rather than write the procedure each time you want to use it, you can store it as a regular ASCII operating system file and retrieve it at any time. Because procedures are stored as ASCII files, you can use other editors to write procedures but you can use only the PROGRESS editor to run them.

Using procedure files when you work with PROGRESS provides many benefits beyond simply minimizing your typing:

- A complete procedure developed with PROGRESS can be used by people who have no knowledge of PROGRESS at all. You can learn PROGRESS, save the necessary statements as procedures, and have other personnel use these procedures with very little instruction.
- Procedure files can call other procedures. This lets you create menus that tie several procedures together into a complete application system that is easy to use.
- You can include procedures in other procedures. An operation such as displaying an order form might be used by six or seven related procedures. If you save as a procedure the statements that display the form, you can simply include that procedure in other procedures.

You store a procedure as an ASCII file by pressing PUT (F6) and supplying the name of the file you want to store the procedure in. You should give the procedure file the .p extension.

4.2.3 Using Uppercase and Lowercase Letters in Procedures

When you write procedures with the PROGRESS editor, you may use uppercase letters, lowercase letters, or a combination of the two. PROGRESS recognizes that **find, FIND,** and **Find** are the same statement.

Letter case is important to consider when you store or retrieve a procedure file. Remember, both procedures and databases are stored as operating system files. UNIX file names are case sensitive. That means that UNIX treats t-demo1.p and T-DEMO1.P as separate files.

You don't have to worry about letter case with DOS, OS/2, BTOS/CTOS, and VMS files. These operating systems treat uppercase and lowercase file names in the same way.

4.2.4 Getting Help

While you're using the editor, you have easy access to HELP, an on-line reference guide which gives you detailed information about PROGRESS statements, functions, operators, keywords, the keyboard, error messages, and design limits.

Be sure to keep HELP in mind as you use the PROGRESS editor. Simply press HELP (F2) if you need some quick answers. To return to the editor from HELP at any time, press STOP (CTRL-BREAK on DOS and OS/2 systems, Action-Cancel on BTOS/CTOS systems, usually CTRL-C on UNIX and VMS systems).

Let's run some procedures!

4.3 RUNNING SOME SIMPLE PROCEDURES

One of the most common things you want to do with any database is look at the information stored in it. Press GET (F5) and when prompted for the name of the procedure you want, type **t-demo1.p**:

```
Type the name of the file:  t-demo1.p
Or blank for temporary file
Press F1 to retrieve file, or F4 to escape.
```

Then press either GO (F1) or RETURN to retrieve the procedure. PROGRESS displays the procedure t-demo1.p in the edit area.

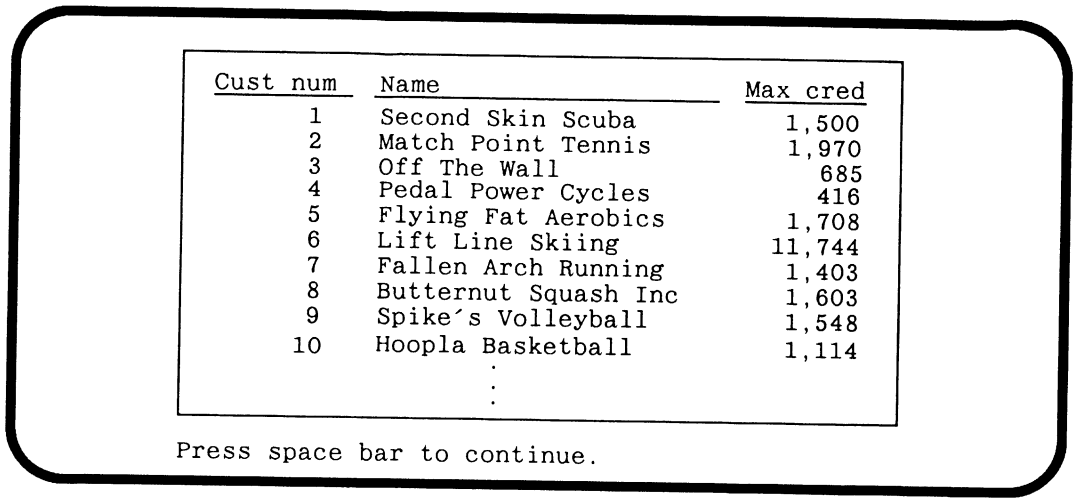
```
                                                    t-demo1.p
FOR EACH customer:
   DISPLAY cust-num name max-credit.
END.
```

If the file was not found in your working directory, then PROGRESS also displays a message telling you the name of the directory from which it retrieved the procedure.

In the first line of the procedure, FOR EACH is a **statement** that tells PROGRESS to read the records from the customer file one by one. The PROGRESS language is made up of many statements that perform different kinds of information processing activities. On the second line, the DISPLAY statement tells PROGRESS to list information on your screen about the customer number, name, and maximum credit allowance.

Because the customer number (cust-num) is the primary index for the customer file, FOR EACH retrieves the records from the customer file in order by customer number. DISPLAY displays the records in the order in which they are read.

Press GO (F1) to run this procedure. PROGRESS displays each customer's number, name, and maximum credit allowance.



Cust num	Name	Max cred
1	Second Skin Scuba	1,500
2	Match Point Tennis	1,970
3	Off The Wall	685
4	Pedal Power Cycles	416
5	Flying Fat Aerobics	1,708
6	Lift Line Skiing	11,744
7	Fallen Arch Running	1,403
8	Butternut Squash Inc	1,603
9	Spike's Volleyball	1,548
10	Hoopla Basketball	1,114
	.	
	.	
	.	

Press space bar to continue.

The ellipses you see in this illustration indicate there are more records after the record for Hoopla Basketball, but we didn't have room to list them all here.

The message at the bottom of your screen

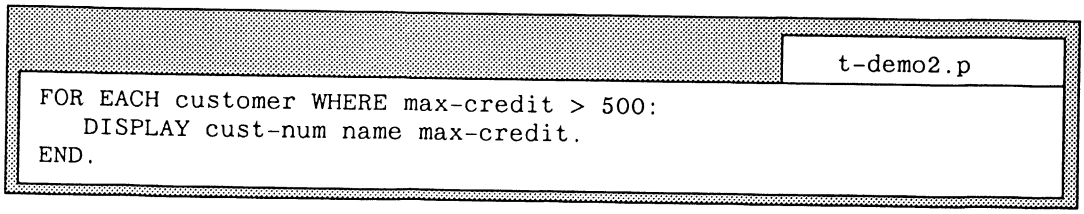
```
Press space bar to continue.
```

means that by pressing the space bar, you can view additional information. When the procedure is complete, PROGRESS returns you to the editor.

4.3.1 Selecting Specific Records to Display

Let's assume that you want to list only those customers whose maximum credit limit (max-credit) is over $500. You use the phrase WHERE to specify selection criteria for the customer records you want. Here, the selection criteria is a credit allowance greater than $500. If the value in the max-credit field is not greater than 500, PROGRESS does not display the record.

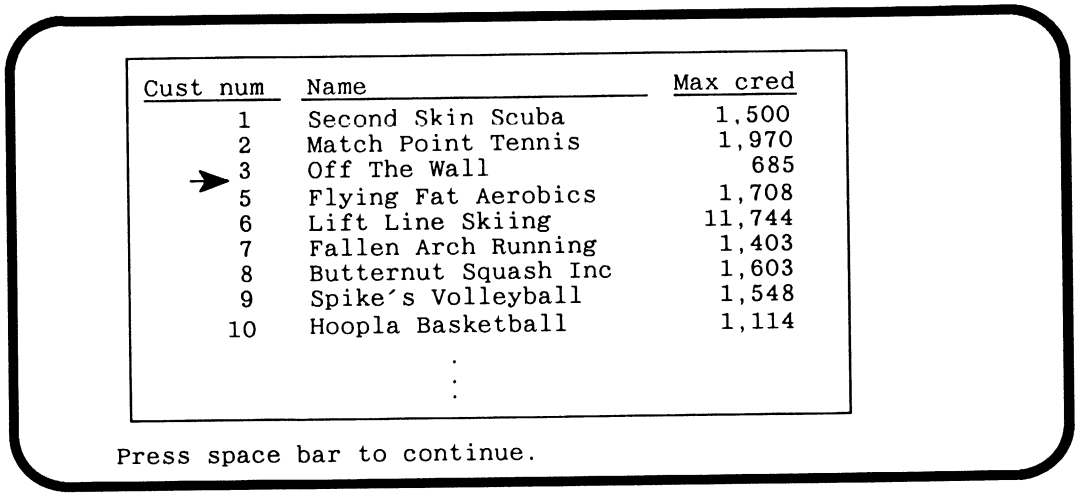
If there is any text in the edit area, press CLEAR (F8) to remove it. Now press GET (F5), type **t-demo2.p**, and press RETURN to retrieve the t-demo2.p procedure.



t-demo2.p

```
FOR EACH customer WHERE max-credit > 500:
   DISPLAY cust-num name max-credit.
END.
```

Press GO (F1) to run this procedure:



Cust num	Name	Max cred
1	Second Skin Scuba	1,500
2	Match Point Tennis	1,970
3	Off The Wall	685
5	Flying Fat Aerobics	1,708
6	Lift Line Skiing	11,744
7	Fallen Arch Running	1,403
8	Butternut Squash Inc	1,603
9	Spike's Volleyball	1,548
10	Hoopla Basketball	1,114

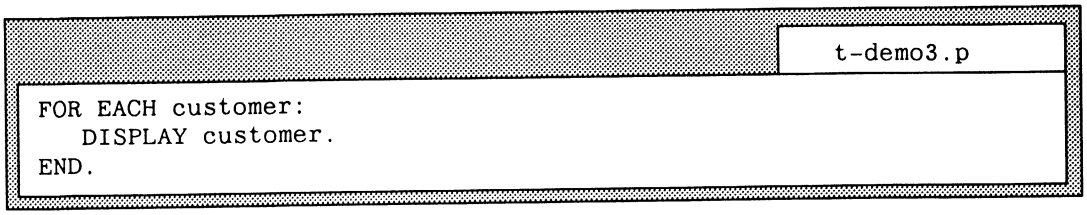
Press space bar to continue.

Because customer number 4, Pedal Power Cycles, has a maximum credit limit of only $416, it does not meet the WHERE selection criteria and is not displayed.

4.3.2 Displaying All the Fields in a Record

The t-demo1.p and t-demo2.p procedures display only a few fields from the customer record. Sometimes you may want to see all the fields in a record. The procedure t-demo3.p displays all the fields in each customer record.

As before, if there is any text in the edit area, press CLEAR (F8) to remove it. Now press GET (F5), type **t-demo3.p**, and press RETURN to retrieve this procedure:



Press GO (F1) to run this procedure. Notice that the field names wrap onto more than one line, making the display harder to read:

```
Cust num     Name                  Addr       Addr2
City     State Zip Tel num Contact      Sls rep  Sls reg
Max cred Unpaid bal  Terms      Tax num     Disc%
Mnth sls[1] Mnth sls[2] Mnth sls[3] Mnth sls[4] Mnth sls[5]
Mnth sls[6] Mnth sls[7] Mnth sls[8] Mnth sls[9] Mnth sls[10]
Mnth sls[11] Mnth sls[12]  Ytd sls
-------------------------------------------------------------
      1 Second Skin Scuba 79 Farrar Ave
Yuma       AZ 85369 (602) 542-0365 Ron Ferrante SLS West
   1,500    937.45 2% 10/Net30
     854.15    74.34  1,462.15    144.49  1,152.23
     248.731,326.05    279.67  1,433.07       0.00
       0.00     0.00  6,974.88
      2 Match Point Tennis  66 Homer Ave
Como       TX 75431 (817) 498-2801 Robert Dorr DKP Central
   1,970    77,674.66 Net30
   2,126.36 3,071.40 3,150.16  2,546.38  2,126.36
   2,415.12 2,336.37 2,625.13  1,492.78       0.00
       0.00     0.00 21,890.06
   Press space bar to continue.
```

If you don't include any **formatting phrases** in your procedure, PROGRESS uses default screen formatting. Let's look at another version of the last procedure. Press GET (F5) and type **t-demo3a.p**:

```
                                                t-demo3a.p
FOR EACH customer:
   DISPLAY customer WITH 2 COLUMNS.
END.
```

Press GO (F1) to run this procedure.

```
   Cust num:  1                      Name:  Second Skin Scuba
       Addr:  79 Farrar Ave         Addr2:
       City:  Yuma                  State:  AZ
        Zip:  85369               Tel num:  (602) 542-0365
    Contact:  Ron Ferrante        Sls rep:  SLS
     Sls reg:  West              Max cred:  1,500
  Unpaid bal:  937.45               Terms:  2% 10/Net30
     Tax num:                       Disc%:
 Mnth sls[1]:    854.15     Mnth sls[2]:    74.34
 Mnth sls[3]:  1,462.15     Mnth sls[4]:   144.49
 Mnth sls[5]:  1,152.23     Mnth sls[6]:   248.73
 Mnth sls[7]:  1,326.05     Mnth sls[8]:   279.67
 Mnth sls[9]:  1,433.07    Mnth sls[10]:     0.00
Mnth sls[11]:      0.00    Mnth sls[12]:     0.00
     Ytd sls:  6,974.88

  Press space bar to continue.
```

You can see that the phrase WITH 2 COLUMNS makes a big difference in the display layout.

4.3.3 Sorting Records for Display

So far, all the displays have been in order by customer number since the cust-num field is the primary index for the customer file. You can use the BY option to sort information by any field. The next example, t-demo4.p, sorts the records by the state field. Use the GET (F5) key to retrieve this procedure.

t-demo4.p

```
FOR EACH customer WHERE max-credit > 500 BY st:
   DISPLAY cust-num name city st zip.
END.
```

Press GO (F1) to run the t-demo4.p procedure. Using a nonindexed field for this kind of sorting may take more time than if you were using an indexed field.

```
Cust num  Name                  City          State  Zip
       1  Second Skin Scuba     Yuma          AZ     85369
      28  Shark Snack Snorkel   San Diego     CA     93441
       8  Butternut Squash Inc  El Centro     CA     92243
      22  Pocket Billiards Co   Phelan        CA     92371
      23  Sub Par Golf          Winter Park   CO     80482
      21  Ship Shape Yachting   Sport Hill    CT     06612
       7  Fallen Arch Running   Codys Corner  FL     32010
      29  Chip's Poker          Dupont        FL     32010
      50  Stay Afloat Swimming  Lula          GA     30554
      12  Batter Up Baseball    Wingo         KY     42088
                    :
Press space bar to continue.
```

4.4 GROUPING STATEMENTS INTO BLOCKS

The basic unit of structure in a PROGRESS procedure is a **block**. A block is a series of statements that PROGRESS treats as a single unit. Each block begins with a block header statement and concludes with an END statement:

```
FOR EACH CUSTOMER:  <-- Block Header Statement
  DISPLAY CUSTOMER.
END.                <-- END Statement
```

In this example, the DISPLAY statement is indented. This formatting is not required, but it makes the procedure easier to read. The DISPLAY statement is part of the FOR EACH block.

The statement that begins a block is a *block header.* It is different from other kinds of statements in two ways:

- It ends with a colon (:). All other statements end with a period.
- It can have a label. The label also ends with a colon.

Here are some examples of block header statements and labels:

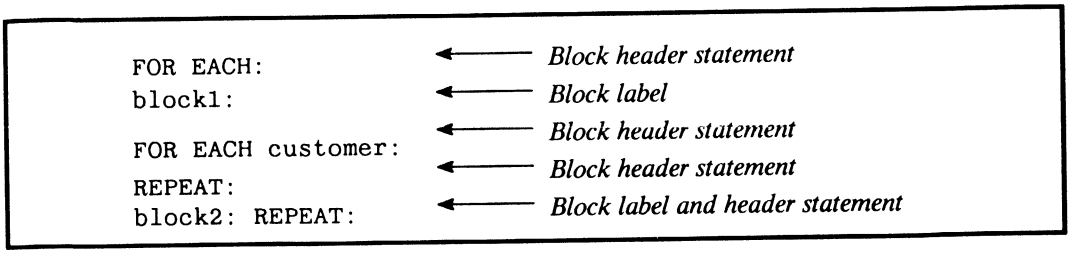


Figure 4–1: Block Header Statements and Labels

Block labels can appear on the same line as a header statement or on the preceding line.

Using blocks in your procedures has many advantages. You can use blocks to:

- Identify points in a procedure where you want PROGRESS to perform special processing services. You'll learn more about these processing services as you work through this Tutorial.
- Group statements for flow control. You may want to tell PROGRESS to process a group of statements under certain circumstances but not under other circumstances.
- Identify a context for the block. You can specify that a group of statements use a particular file by naming that file in the block header.

4.4.1 Properties of Blocks

A block may have one or more of several properties. In this chapter, we'll cover two of them:

- **Looping**

  PROGRESS processes the statements in the block over and over again.

- **Record reading**

  The block automatically reads records.

4.4.2 Types of Blocks

PROGRESS uses four kinds of blocks. Each has special properties:

1. **FOR EACH blocks**

   FOR EACH blocks begin with the FOR EACH statement and have looping and record reading properties. FOR EACH blocks also have special control and display properties which you'll learn more about later on in this Tutorial.

   Each time a FOR EACH block loops, or *iterates*, it reads another record from the file named in the block header.

2. **REPEAT blocks**

   REPEAT blocks begin with a REPEAT statement. Like FOR EACH blocks, they loop, but they do not automatically read records.

3. **DO blocks**

   DO blocks begin with a DO statement and are especially useful for grouping statements. They do not loop or automatically read records unless you use special phrases.

4. **Procedure blocks**

   An entire procedure is a block. Procedure blocks do not begin with a block header or end with an END statement. They have no looping or record reading properties.

Take another look at the procedure t-demo1.p:

t-demo1.p

```
FOR EACH customer:
  DISPLAY cust-num name max-credit.
END.
```

The first time through the block, FOR EACH reads the first customer record from the customer file, using the primary index cust-num to retrieve the first record. After the DISPLAY statement displays data for that record, PROGRESS does another iteration of the FOR EACH block. That is, it reads the next record using the primary index and displays the data for that record. This process of iterating and reading records continues until the end of the customer file.

Keep these block properties in mind when writing procedures. They let you control the program flow and they let you control access to records in the database.

4.5 WRITING PROCEDURES TO PERFORM FILE MAINTENANCE

The most common file maintenance operations are add, update, delete, and list. PROGRESS provides functions for performing these activities. With these four activities, you can begin to build and maintain the data you want to store in the database. In the previous sections, you saw how to use the FOR EACH and DISPLAY statements to list the records in the database. Now let's look at the language statements PROGRESS provides to do the other file maintenance operations.

4.5.1 Adding Records

You can use the INSERT statement to add a record to a file. (If you want to experiment with the editor, you may want to type this statement yourself. Or, you can simply press GET (F5) and retrieve the procedure named **t-adcus1.p**):

t-adcus1.p

```
INSERT customer WITH 2 COLUMNS.
```

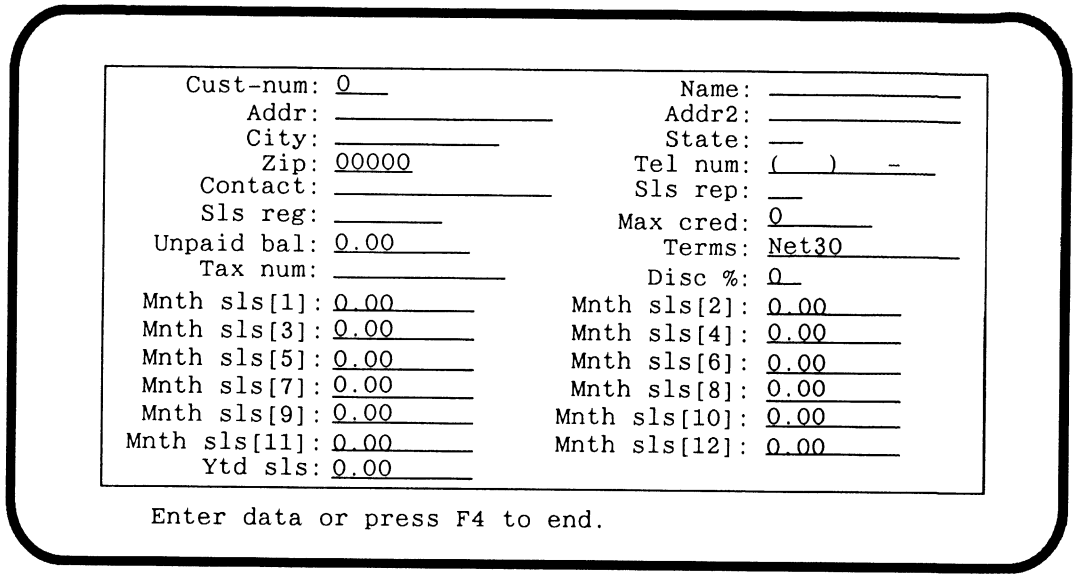
This procedure lets you add one customer record. However, you may want to add *several* records to the customer file. You could use this procedure, but you'd have to press GO (F1) to run it over and over again every time you want to add a new record.

There is an easy solution to this problem. Simply enclose the INSERT statement in a block that has the looping property, such as the REPEAT block.

t-adcust.p

```
REPEAT:
  INSERT customer WITH 2 COLUMNS.
END.
```

Now, press GO (F1) to run this procedure. The following screen appears:

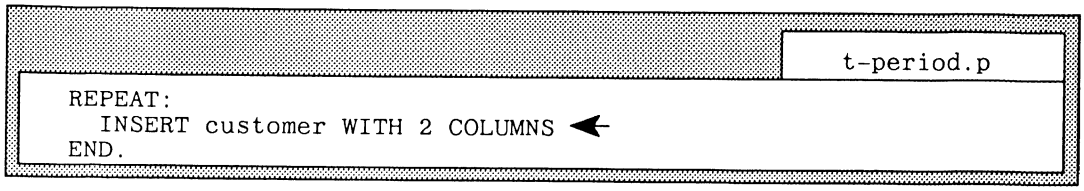


Notice that the fields are underlined to show they are *enabled* for input. Some terminals may use another screen attribute, such as reverse video, but a field that is enabled for input is always shown with a distinctive attribute.

This procedure uses the data you enter to create a new record in the customer file. Remember that the cust-num field is defined as a unique primary index. If the cust-num value you enter has already been assigned to another record in your mywork database, you get an error message.

Add a few records to see how this procedure works (the customer records already in your mywork database have customer numbers under 100). Press END-ERROR (F4) when you are finished. Then, press RECALL (F7) to bring the procedure back into the editor (or press GET (F5) to retrieve the procedure named t-period.p).

If you recalled t-adcust.p, remove the period from the end of the INSERT statement after the word COLUMNS (it's already been done for you in t-period.p).



Press GO (F1) to run t-period.p or your modified t-adcust.p

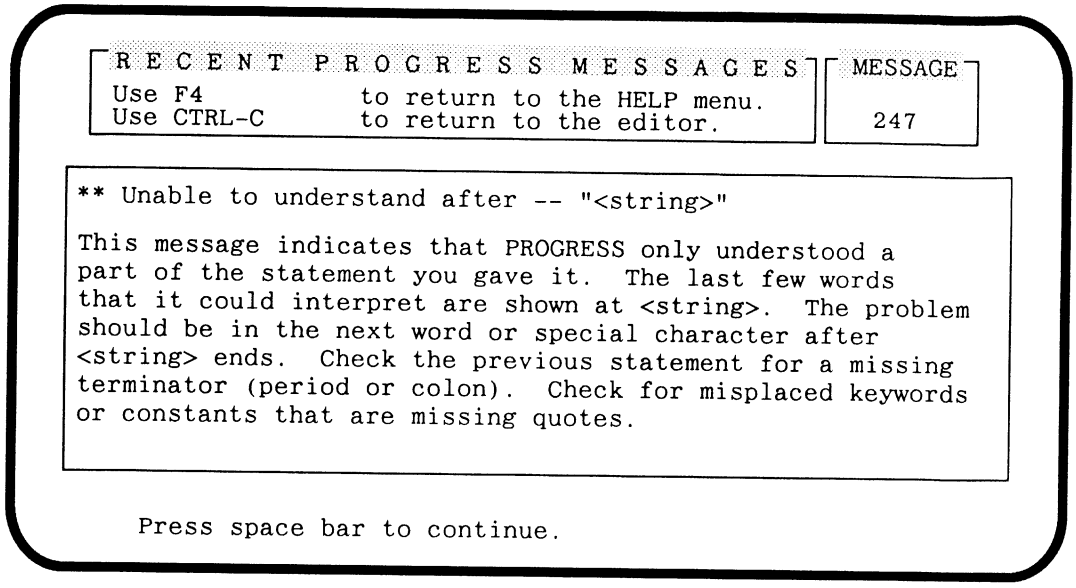
```
REPEAT:
  INSERT customer WITH 2 COLUMNS
END.

** Unable to understand after -- WITH 2 COLUMNS.  <-- Error Message
   Enter PROGRESS procedure.  Press F1 to run.
```

Something went wrong, didn't it? PROGRESS tried to compile the procedure and found an error. Whenever the PROGRESS compiler detects an error in your procedure, you get an error message describing the problem and PROGRESS puts the cursor on or near the statement that caused the error.

Usually you'll be able to figure out, on your own, what the problem is. But, suppose you don't understand the error message. Press HELP (F2) and choose the default option - option 1, Recent Messages - by either pressing RETURN or typing **1**:

```
                P R O G R E S S  H E L P
          1. Recent Messages
          2. Any Messages
          3. Keyboard
          4. Statements
          5. Functions
          6. Operators
          7. Keywords
          8. Design Limits
          9. Ending a PROGRESS Session

          d. Access the DICTIONARY
          e. Escape to the Operating System
          f. List the Filenames in a Directory
          q. Quit to the operating system.
          r. Run a PROGRESS program.

   Use CTRL-C anytime to return to the editor.
Enter selection or press F4 to exit.
```

This is the message you get for the t-period.p procedure:



PROGRESS displays information about error messages you have received, starting with the most recent one. When you are finished using HELP, press END-ERROR (F4) repeatedly to return to the editor. (If you add a period after the word COLUMNS in your t-period.p procedure, it should work exactly like t-adcust.p.)

Look at t-adcust.p once again:

t-adcust.p

```
REPEAT:
  INSERT customer WITH 2 COLUMNS.
END.
```

You may be wondering, "Why wasn't a FOR EACH block used here? It has the looping property, too." The reason is that FOR EACH blocks have a record reading property as well as a looping property. A FOR EACH block reads a record on every iteration of the block. Here, all you want to do is repeatedly add *new* records; you don't want to read existing ones.

4.5.2 Updating Records

Often you want to retrieve a specific record from a file and change some of the information in that record. You can use three simple statements to perform this task. They are PROMPT-FOR, FIND, and UPDATE.

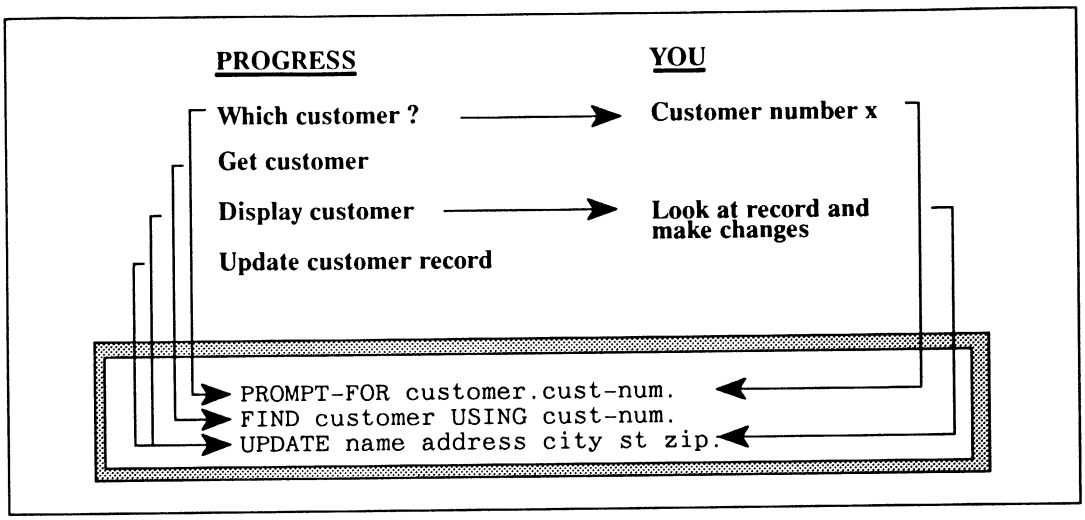
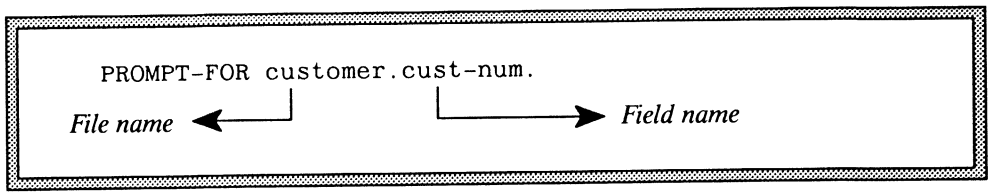


Figure 4–2: Updating Data

The procedure you write to make changes to a customer record has to do three distinct things:

- It must first find out which record you want to change. The PROMPT-FOR statements asks for that data. Since the database has more than one file with a field named cust-num, you must use the field's full name to specify the appropriate file. This full name consists of the file name, a period, and the name of the field:



- Once PROGRESS has the customer number, the FIND statement uses that number to read the corresponding record from the customer file.

- The UPDATE statement then displays the appropriate record and lets you change information in the specified fields.

Type this procedure into the editor:

t-chcust.p

```
PROMPT-FOR customer.cust-num.
FIND customer USING cust-num.
UPDATE name address city st zip.
```

To store this procedure, press PUT (F6) and name it **t-chcust.p**. Then press RETURN. If you store the file without naming a directory, PROGRESS stores that file in your current working directory.

Press GO (F1) to run this procedure. Enter a customer number (for example, 2). Then change the data and press GO (F1), or just press GO (F1) to keep going without changing any data.

4.5.3 Deleting Records

You use the DELETE statement to remove a record from the database. A minor modification turns t-chcust.p into a procedure that deletes records. Use GET (F5) to retrieve t-chcust.p and make this change:

t-delcu.p

```
   PROMPT-FOR customer.cust-num.
   FIND customer USING cust-num.
➔  DELETE customer.
```

The first and second statements are the same, but the third statement deletes rather than updates the selected record.

Press PUT (F6) to save this procedure as **t-delcus.p**. Press GO (F1) to run this procedure if you want to.

4.6 COMMENTING YOUR PROCEDURES

When you write procedures, you can use comments to explain what the procedure is doing. Comments begin with the two-character symbol /* and end it with */.

t-adcust.p

```
/* t-adcust.p */
/* This procedure lets you add customer records */

REPEAT:
  INSERT customer WITH 2 COLUMNS.
END.
```

Comments

4.7 SUMMARY

In this chapter you learned how to:

- Unpack the sample procedures used in this book.
- Write procedures with the PROGRESS Procedure Editor.
- Retrieve and run PROGRESS procedures.
- Work with several PROGRESS language statements and phrases to display records in the database.
- Group statements into FOR EACH and REPEAT blocks.
- Write file maintenance procedures to add, update, and delete records.

Now that you've looked at a few procedures, you probably want to begin writing your own. However, before you do, let's see how to define files, fields, and indexes with the Data Dictionary.

_Chapter 5 Defining an Application Database

In Chapter 1, you looked at the customer, order, order-line, and item files that make up your mywork database. In Chapter 4, you began to write procedures to access the data in those files. The files, fields, and indexes in your mywork database were created with the PROGRESS Data Dictionary. Data was entered into them with PROGRESS procedures. This chapter describes how to use the Data Dictionary to define an application database. In Chapter 5, you will use the database definitions to build an application.

This chapter describes the following topics:

- Displaying and changing data definitions.
- Exhibiting field definitions.
- Adding a new file to the customer file.
- Deleting files
- Defining indexes for a file.
- Adding a new index to the customer file.
- Changing your dictionary definitions
- Saving and printing data definitions
- Using the dictionary in multi-user mode
- Using other dictionary functions
- Using the PROGRESS Query/Run-Time Data Dictionary

Before you define any kind of database (in a filing cabinet or in a computer) you have to determine the following:

- What files will go in the database? Your mywork database contains the customer, order, order-line and item files, and other files.
- What kinds of information (fields) will go into each of the records in the file? For example, each record in the customer file contains a customer number, name, address, city, state, zip code, maximum credit limit, and other fields.
- Which field, or combinations of fields, will be used as the primary and secondary indexes for rapid access to the files and for default sorting? The customer file uses the customer number as its primary index, and the name and zip fields as secondary indexes.

You use the Data Dictionary to define database files, fields and indexes.

Are you still in the PROGRESS editor? If you've taken a break since the last chapter, start PROGRESS with your mywork database connected by typing the appropriate command for your operating system at the system prompt:

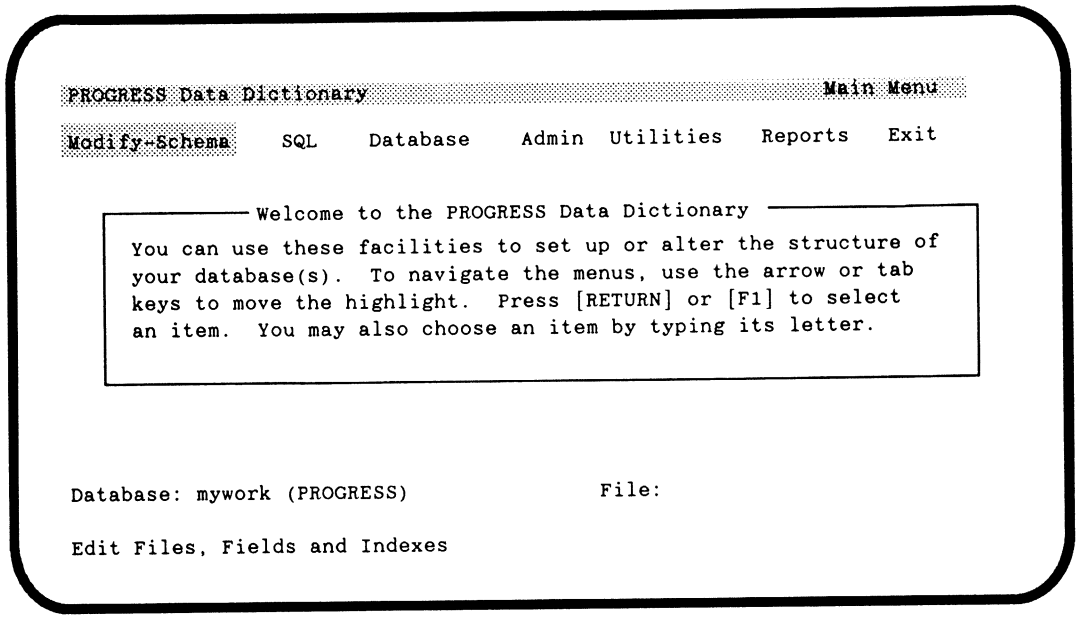
Table 5–1: Command to Start PROGRESS

Operating System	Starting mywork Database
UNIX, DOS, and OS/2	`pro mywork`
VMS	`PROGRESS mywork`
BTOS/CTOS	`PROGRESS 4GL` `[Options] -1 mywork`

When the horizontal lines of the PROGRESS edit area appear, you can access the Data Dictionary in one of two ways. To use the DICTIONARY statement enter the statement as shown in the following screen and press GO (F1). (You can abbreviate this command by typing **dict.**)

```
dictionary

Enter PROGRESS procedure.  Press F1 to run.
```

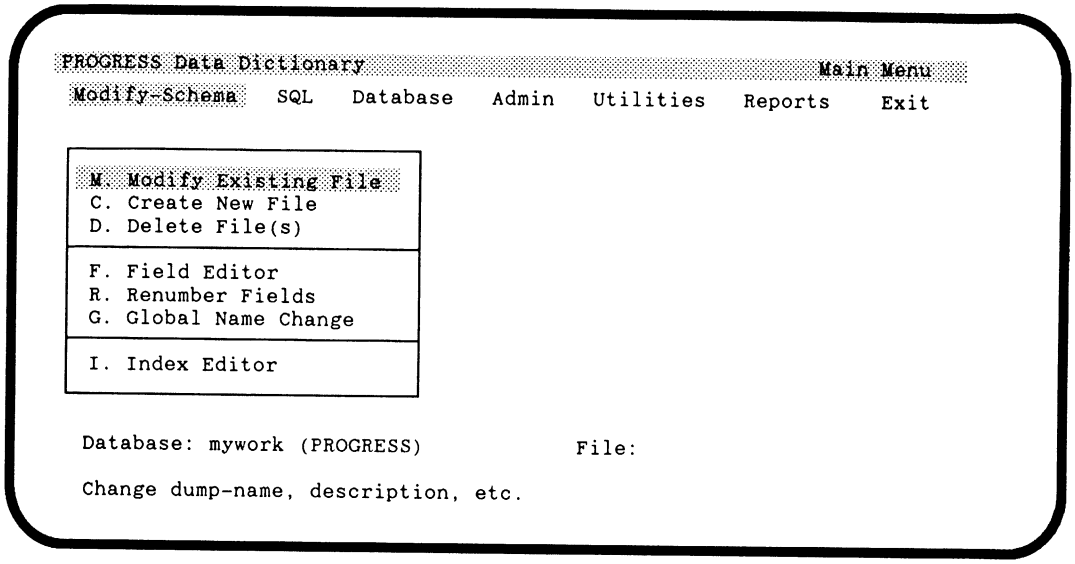
You can also press HELP (F2 or CTRL-W) and then press **d** to access the Dictionary from the PROGRESS Help menu. Either way, PROGRESS then displays the following screen.



For now, we'll look at the first option, `Modify-Schema`.

5.1 MODIFYING SCHEMA

You can display and change data definitions by choosing the Modify Schema option from the Data Dictionary Main Menu. To choose `Modify-Schema` from the Data Dictionary Main Menu, you can type the letter **m** or press RETURN or GO. PROGRESS displays a list of options you can choose.



From this menu you have the following options:

Modify Existing File — Change the file name, dump-name, description, criteria for validation, and deletion validation message for a file, and/or whether the file name is hidden.

Create New File — Create a new file definition.

Delete File(s) — Remove one or more file definitions.

Field Editor — Add, change, or remove fields from a file definition.

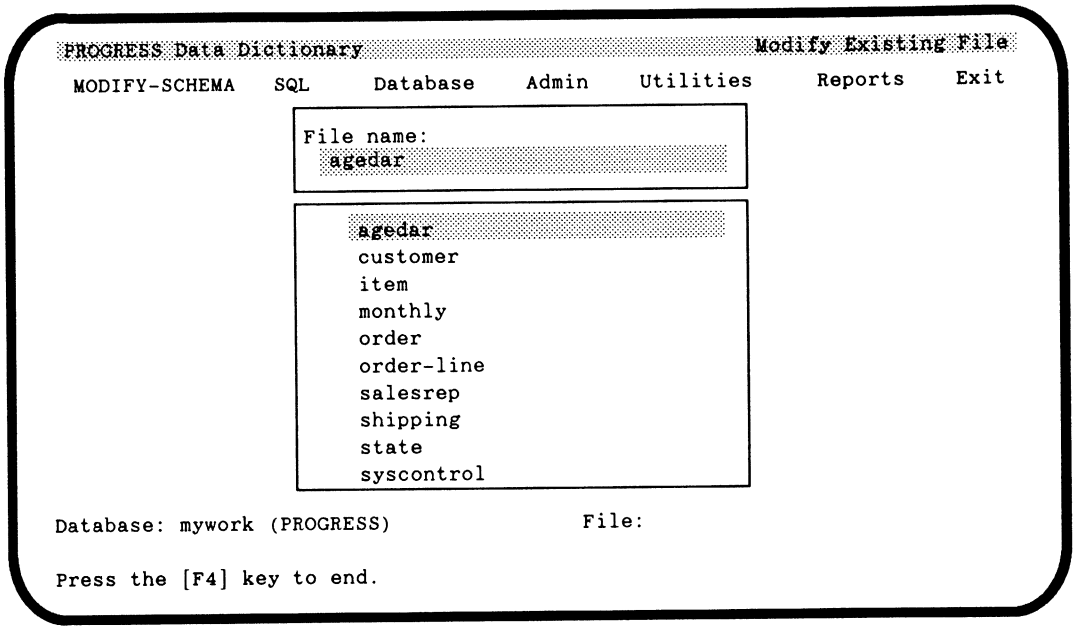
Renumber Fields — Resequence the fields into alphabetical order, or renumber the fields in the order defined by the values in their _Order fields.

Global Name Change — Change a field name throughout the database.

Index Editor — Add, alter, or remove indexes in a file.

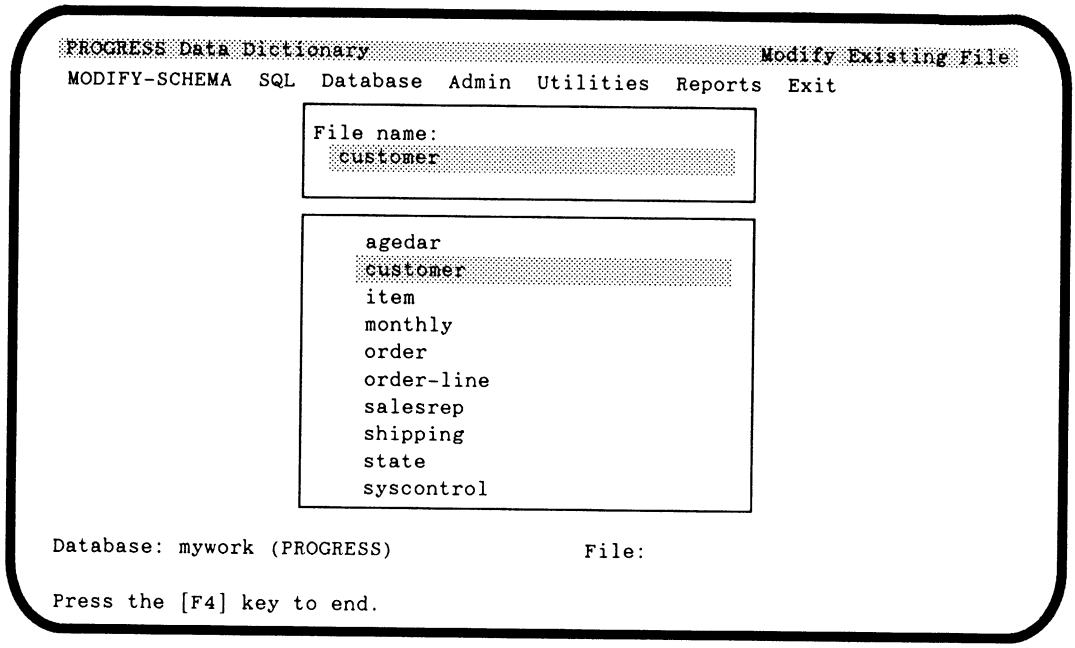
5.2 MODIFYING EXISTING FILES

To choose Modify Existing File from the `Modify-Schema` menu, you can type the letter **m** or press RETURN if the `Modify Existing File` option is highlighted. PROGRESS displays a list of database files and prompts you to choose the file you want to work with, as shown below:



NOTE: Although hidden files are not displayed in this list, you can still select one by typing its full name and pressing RETURN.

We'll look at the customer file now. You can access this file by typing **customer** or by pressing the space bar or the [↓] [↑] keys to move the highlighted box to customer as shown in the following screen. Then press [RETURN] or [GO].



Once you have supplied a file name, PROGRESS displays a screen that lets you modify the existing customer file as shown below:

```
PROGRESS Data Dictionary                                  Modify Existing File
  MODIFY-SCHEMA    SQL    Database    Admin      Utilities      Reports      Exit

  File Name: customer                                  Hidden: no
  File Type: PROGRESS                                  Frozen: no
 File Owner:                                       File Number: 2
  Dump Name: customer (unique name for data dump into .d file)

Description: Customer information

Enter the message to be displayed for disallowed deletions.
     ValMsg: Can not delete customer with outstanding orders
Enter an expression which must be TRUE to allow record deletions.
     ValExp: NOT (CAN-FIND(FIRST order OF customer))


Database: mywork (PROGRESS)                    File: customer

Enter data or press F4 to end.
```

From this screen you can modify the following file description information:

File Name

The name of the file must begin with a character from a–z (or A–Z). The name can consist of alphabetic characters, digits, and the characters #,$,%,&, -, and _. Database file names are not case sensitive. You cannot use PROGRESS keywords as file names. (A list of keywords is included in the Index of the *PROGRESS Language Reference* manual.)

File names must be 32 characters or less. However, under DOS and OS/2 you should restrict file names to 8 characters or fewer or make sure that the first 8 characters of any two database file names are different. Otherwise, you cannot use the Data Dictionary procedure that dumps data into ASCII files.

Hidden

By entering **yes** in the Hidden field, you can prevent a file from appearing in the Data Dictionary File Selection list. A hidden file can still be accessed like any other file by typing its name at the prompt, `File Name:_____`.

You can also prevent a file from appearing in this list by setting the file's _Hidden field with the following PROGRESS procedure.

```
FIND _File "filename".
_Hidden = true.
```

Dump Name This is a unique name for the data dump into the .d file.

Description This field may contain a description of the file.

ValMsg Use this field to define a message you want to display when the expression in VALEXP is false and PROGRESS does not delete the record.

ValMxp Use this field to define a logical expression that must be true before PROGRESS will delete a record from the file. The expression must be a valid PROGRESS expression, it can include constants, field names, or a combination of the two, and it must produce a true or false result. For example, you could define a validation expression to test that no order records exist for a customer before you delete that customer record. The validation expression lets you maintain what is called **referential integrity.**

You can define validation criterion after you create a file by choosing Modify-Schema from the Data Dictionary Main Menu, then choosing Modify Existing File.

If you want to define a validation expression that is too long to enter into the Dictionary, enter the expression into an include file and specify the name of the include file between braces as the valexp entry. Include files are described in Chapter 14 of this book.

Chapter 3 of the *PROGRESS Programming Handbook* describes more about these file characteristics.

Don't change anything just yet. Instead, press END-ERROR (F4 or CTRL-E) to return to the Data Dictionary Main Menu.

5.3 CREATING NEW FILES

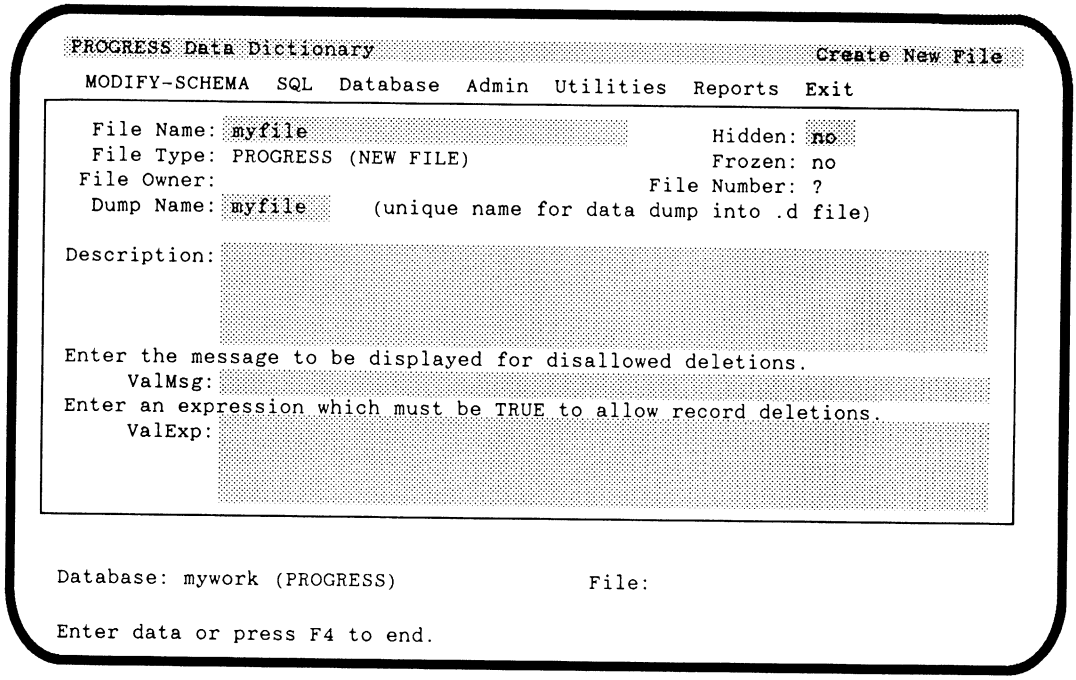
Select Modify-Schema again. PROGRESS displays the Modify-Schema menu. This time, select Create New File. PROGRESS displays the following screen:

```
PROGRESS Data Dictionary                                    Create New File
  MODIFY-SCHEMA  SQL  Database  Admin  Utilities  Reports  Exit

   File Name:                                        Hidden: no
   File Type: PROGRESS (NEW FILE)                    Frozen: no
  File Owner:                                   File Number: 2
   Dump Name: ?          (unique name for data dump into .d file)

 Description:

 Enter the message to be displayed for disallowed deletions.
      ValMsg:
 Enter an expression which must be TRUE to allow record deletions.
      ValExp:


 Database: mywork (PROGRESS)                File:

 Enter data or press F4 to end.
```

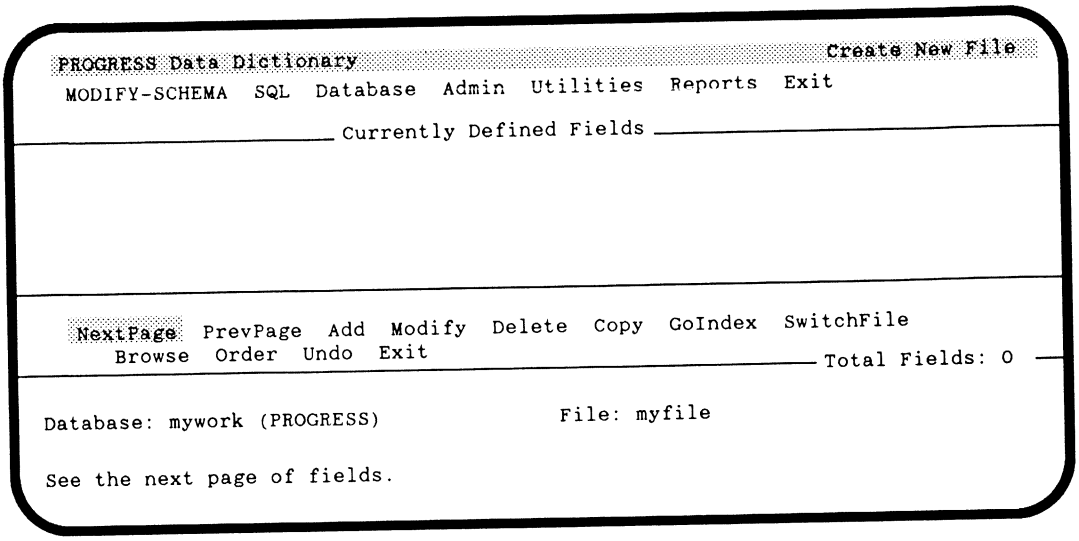
Now you can create a new file called myfile. Enter **myfile** as the File Name and press RETURN. Press RETURN again to accept "no" as the default value for the Hidden field. The Data Dictionary automatically enters "myfile" into the Dump Name field. For now, don't enter any more information on defining this file.

Although "myfile" uses only six characters, you can use file names up to 32 characters long. Since the file name "myfile" is shorter than eight characters, the Data Dictionary automatically enters the name into the Dump Name field. However, if you had used a name longer than 8 characters, the Data Dictionary would truncate the name to eight characters and place the truncated version into the Dump Name field.

Since the Dump Name field is only eight characters long, a problem arises if you use two or more file names that are not unique within the first 8 characters. For example, if you create two files, named `customer1` and `customer2`, the Data Dictionary truncates `customer1` to `customer`, but cannot truncate `customer2` to `customer`, because the Dump Name must be unique. In this example, you would have to supply the Dump Name for `customer2`. Dump Names must be unique and eight characters or less in length.



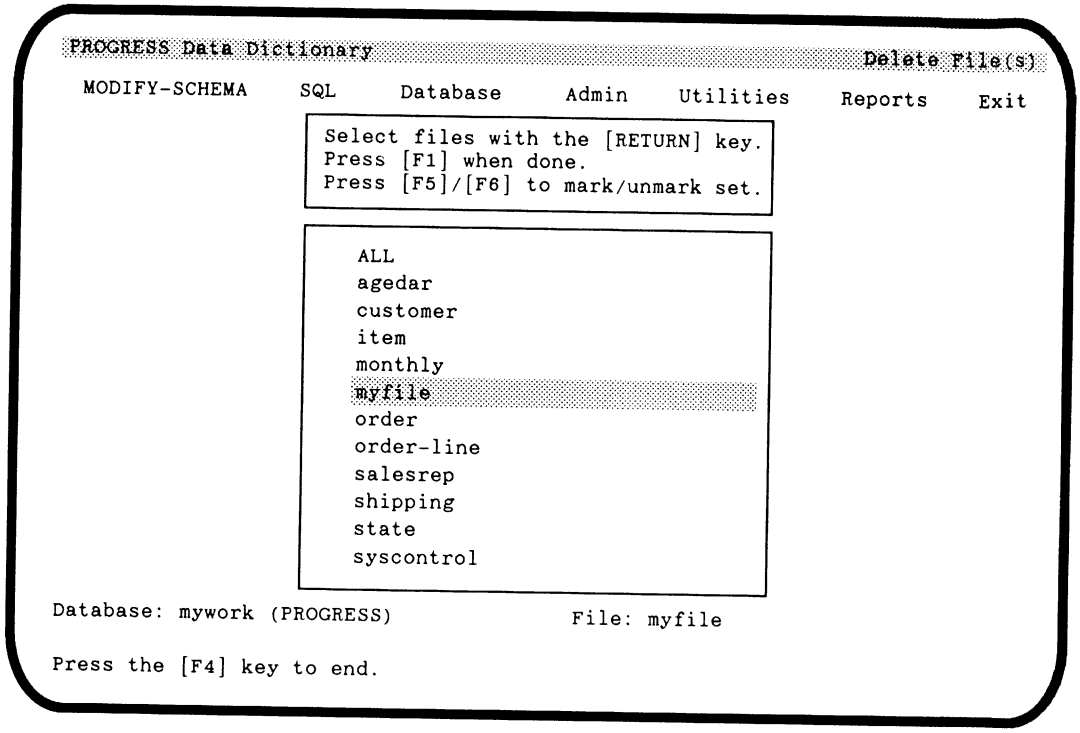
Press GO (F1 or CTRL-X) to create the new file. The following screen appears, so you can add fields for `myfile`:



Don't create any fields for this file. Instead, type **e** for exit to return to the Data Dictionary Main Menu.

5.4 DELETING FILES

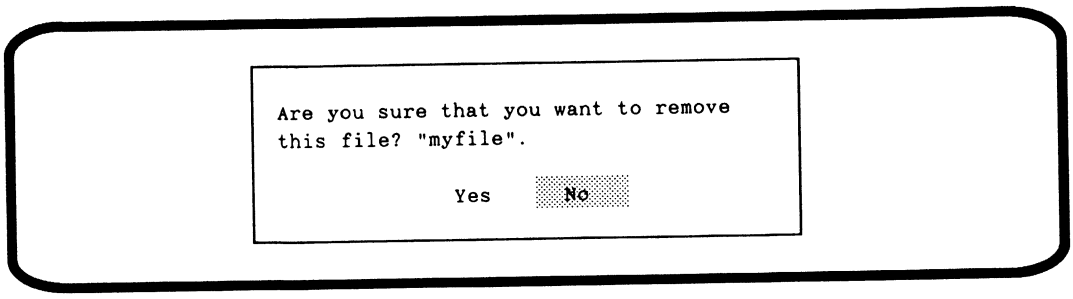
Select `Modify-Schema` again. PROGRESS displays the `Modify-Schema` menu. Now select `Delete File(s)`. PROGRESS displays the following screen.



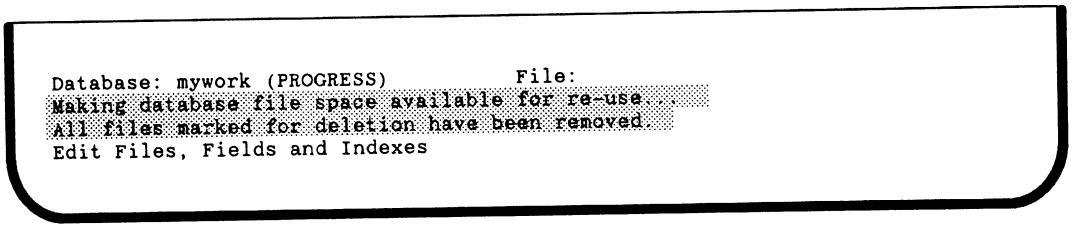
NOTE: If there were SQL tables, Frozen files, or files with no active indexes in this list, the Data Dictionary would display a note to let you know that from this screen you cannot delete SQL tables, Frozen files, or files with no active indexes. Frozen files must be unfrozen, before you can delete them. To unfreeze files, select the `Utilities` option from the Data Dictionary Main menu, then select the `Freeze/Unfreeze` option. To delete SQL files, use the DROP TABLE statement. Refer to the *PROGRESS Language Reference* manual for more information about DROP TABLE.

Delete the file called "myfile" that you created in the last section. Press RETURN to mark the `myfile` file for deletion. Then press GO (F1) to delete the file.

The Data Dictionary displays a window asking you to confirm that you want to delete the file as show below:



Select "`Yes`", by typing **Y** or by using your ← arrow key to highlight it. Then press GO or RETURN. The Data Dictionary displays the following message:



5.5 EDITING FIELDS

Select `Modify-Schema` again. PROGRESS displays the `Modify-Schema` menu. Now select `Field Editor` from the submenu. PROGRESS displays a list of database files and prompts you to choose the file you want to work with.

You now need to access the customer file. You can access it by typing **c** or by pressing the space bar or the [↓] [↑] keys to move the highlighted box to `customer`. Then press [GO] or [RETURN]. PROGRESS displays the following screen:

```
PROGRESS Data Dictionary                                    Field Editor
 MODIFY-SCHEMA  SQL  Database  Admin  Utilities  Reports  Exit

                        Currently Defined Fields

  Address     Address2    City        Contact     Curr-bal    Cust-num
  Discount    Max-credit  Mnth-sales  Name        Phone       Sales-region
  Sales-rep   St          Tax-no      Terms       Ytd-sls     Zip

  NextPage  PrevPage  Add  Modify  Delete  Copy  GoIndex  SwitchFile
    Browse  Order  Undo  Exit
                                                      Total Fields: 18
Database: mywork (PROGRESS)              File: customer

See the next page of fields
```

From this screen you can edit fields for the customer file. You can also add, delete, or simply browse through the fields. Use the spacebar or arrow keys to move through the following choices.

NextPage — See the next page of fields.

PrevPage — See the previous page of fields.

Add — Add a new field.

Modify — Update or rename a field.

Delete — Remove a field.

Copy — Copy fields from another file definition.

GoIndex — Go to the Index Editor.

SwitchFile — Switch to a different file.

Browse — Browse through field definitions for this file.

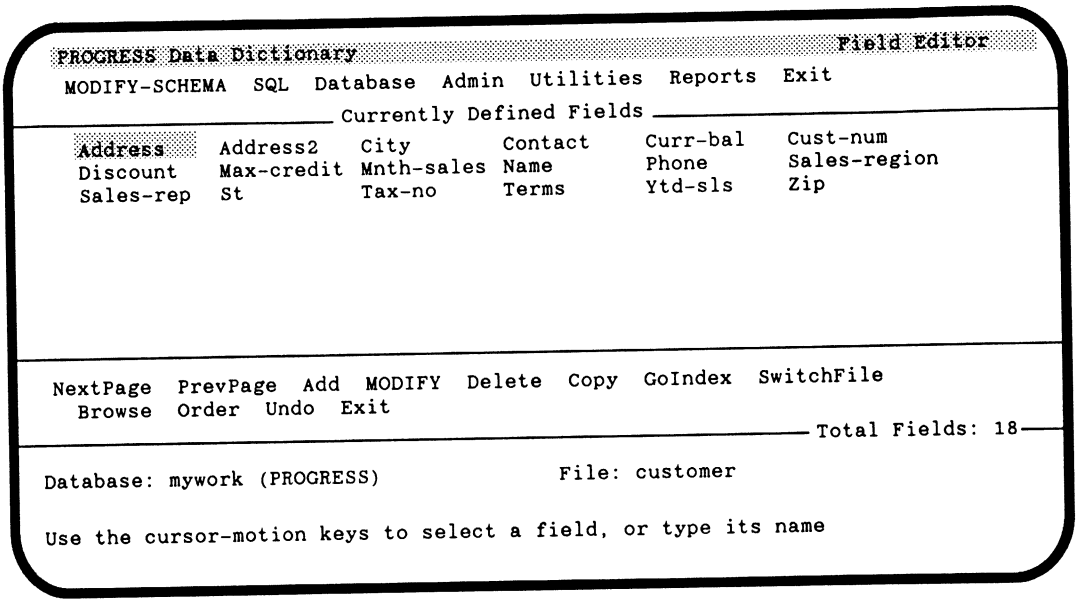
Order — Show fields in alphabetical order, or in the order defined by the values in their _Order fields.

Undo Undo this session's changes to the field structures.

Exit Exit Field Editor, save changes, and return to the Main Menu.

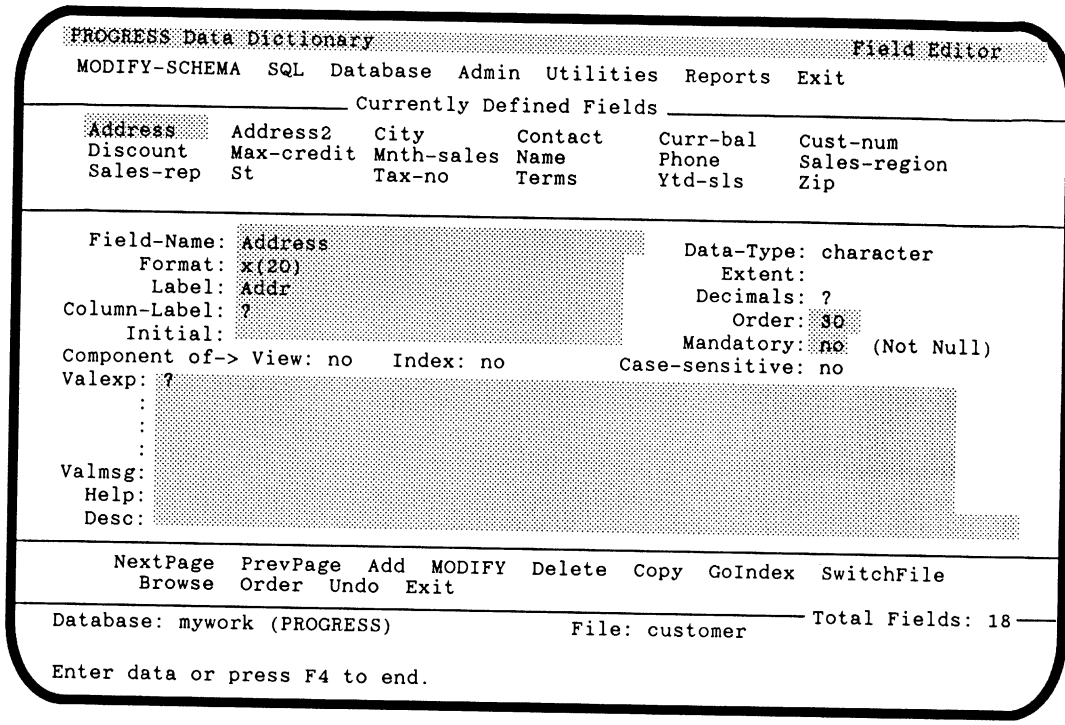
5.5.1 Examining Field Definitions

For now press **m**, for the `Modify` option, to look at the definitions of the fields in the customer file. Press RETURN or GO. PROGRESS displays the following screen:



The `Address` field is highlighted. Press RETURN.

The Dictionary displays the following screen, which shows the field definitions for the Address field and which highlights the parts of the Address field definition that you can change:



This screen shows the definitions for the Address field. Notice that there are 16 characteristics of the field:

- Field name
- Format
- Label
- Column-Label
- Initial
- Component of
- Valexp
- Help
- Data-Type
- Extent
- Decimals
- Order
- Mandatory
- Case-sensitive
- Valmsg
- Desc

NOTE: For non-PROGRESS databases, the list of field characteristics may be different.

When you are through, press END (F4) to return to the Field Editor screen.

5.5.2 Defining the Name of a Field

The Field-Name is the name of the database field. It is a good idea when naming the field to pick a name that describes the data in the field. Field names may be up to 32 characters in length and can consist of alphabetic characters, digits, and the characters $, &, #, % -, and _. In addition, field names must begin with A-Z or a-z. You cannot use PROGRESS keywords as field names.

If a field with the same name already exists in another database file, PROGRESS asks you, `Copy field-data from` *file-name.field-name*`?` `Yes`. By default, all field-data from the existing field is displayed; you can update any value (including `Field-Name`) except `Data-Type`. This dictionary feature makes it easier to join files and to create templates for commonly used field types.

Field names are not case sensitive; they can be upper case, lower case, or a combination of both. If you name a field **Phone** in the Dictionary, you may refer to it as **phone** or **PHONE** in your procedures.

5.5.3 Defining the Data Type of a Field

The Data-Type determines the kind of data values a field can store (letters, numbers, dates, and so on). There are five data types you can use:

- **Character** fields contain data of any kind. Character data may include upper-case and lower-case characters. As you examined the fields in the customer file, you may have noticed that the `Name`, `Address`, `Address2`, `City`, `St`, `Phone`, `Contact`, `Sales-rep`, `Terms` and `Tax-no` are defined as character fields.

  Although PROGRESS allows character field display formats of up to 255 characters, you should restrict the format length of a character field to the input/output line width of your terminal (typically 80 characters) by specifying the appropriate format. (Actually, the field can contain more than 255 characters, but the longest display format is 255 characters.)

- **Integer** fields are whole numbers. They can be positive or negative, ranging in value from -2,147,483,648 through 2,147,483,647. If you enter blanks as the value of an integer field, PROGRESS stores the value of that field as zero. `Cust-num`, `Zip`, and `Discount` are integer fields.

- **Decimal** fields contain decimal numbers up to 50 digits in length. You may use up to 10 digits to the right of the decimal point. If you enter blanks as the value of a decimal field, PROGRESS stores the value of that field as zero. `Max-credit`, `Curr-bal`, `Mnth-sales`, and `Ytd-sls` are decimal fields.

- **Date** fields contain dates from 1/1/32768 B.C. through 12/31/32767 A.D. You can specify dates in this century with a two-digit year, such as 8/9/90, or a four-digit year, 8/9/1990. Dates in other centuries require a four-digit year.

If you enter dates without the year, PROGRESS assumes the current year. If you supply blanks as the value of a date field, PROGRESS stores that field as an *unknown value* and displays it as blanks.

- **Logical** fields contain yes/no or true/false values.

In addition to the range of values listed, you can also use a special value called "unknown value." This value is represented by a question mark (?).

5.5.4 Defining the Display Format of a Field

The format of a field describes the way the data in that field is shown on screens and in printed reports. PROGRESS automatically supplies a default format for each data type, but you can change that default format. You can also override formats you define in the Dictionary by using the FORMAT option with the Format phrase. For more information on formatting, see Chapter 4 of the *PROGRESS Programming Handbook*.

The default display format for a **character** field is **x(8)**. The "x" represents an alphanumeric character position and the "8" represents the number of characters to be displayed. You can override this format in the Dictionary or in your procedures.

You can use several different symbols to define a character format: **X,N,A,!** and **9**. You can also give additional characters to be inserted into the display, such as parentheses around an area code in a phone number.

Table 5–2 contains different display formats.

Table 5–2: Character Field Display Format Examples

Format	Value in Field	Display
xxxxxxxx	These are characters	These ar
x(9)	These are characters	These are
x(20)	These are characters	These are characters
xxx	These are characters	The
AAA-9999	abc1234	abc-1234
(999) 999-9999	6172754500	(617) 275-4500
!!	ma	MA

You can use many different symbols to describe the display format of **decimal** and **integer** fields: **(+ - > , 9 z * . < DR CR** and **DB**. The default display format for a decimal field is **- > >, > >9.99**. The default display format for an integer field is **- >, > > >, > >9**.

- **-** A dash puts a minus sign in front of the number if the number is negative, and puts a blank in front of the number if the number is positive.

- **>** A greater than sign is replaced with a digit if that digit is not a leading zero. If the digit is a leading zero, PROGRESS replaces **>** with a blank and pulls any characters to the left one place to the right.

- **,** A comma is displayed as a comma unless it is preceded by a **>** sign or a **Z**. If the comma is preceded by **>** and the **>** is replaced with a leading zero, the comma is replaced with a null. If the comma is preceded by **Z** and the **Z** is replaced by a blank, the comma is replaced with a blank.

- **9** The number 9 is replaced by a digit.

- **.** A period represents a decimal point.

Use the **<** and **>** symbols together to implement "floating decimal" display. Up to 10 **<** symbols can follow a decimal point. The **<** symbols must be balanced by an equal or greater number of **>** symbols left of the decimal. A **<** is replaced by a digit if its corresponding **>** is collapsed by a leading zero (and the stored value has the required precision).

Table 5–3 on the next page contains some examples of how PROGRESS displays a numeric value using different formats.

Table 5–3: Numeric Display Format Examples

Format	Value	Display
9999	123	0123
9,999	1234	1,234
$zzz9	123	$ 123
$>>>9	123	$123
$->,>>9.99	1234	$1,234.00
$>,>>9.99	1234	$1,234.00
#-zzz9.999	-12.34	#- 12.340
Tot=>>9Units	12	Tot=12Units
$>,>>9.99	-12.34	????????? (1)
$>,>>9.99	1234567	????????? (2)
>>,>99.99<<< (3)	12,345.6789	12,345.68
>>,>99.99<<<	1,234.5678	1,234.568
>>,>99.99<<<	123.45	123.45
>>,>99.99<<<	12.45678	12.45678
$>>9.99 DR	-78.25	$78.25 DR
($>>9.99)	-25.38	($25.38)
($>>9.99)	15.1	$15.10

(1) In this example, there is a negative sign in the value -12.34 but the display format of $>,>>9.99 does not accommodate that sign. The result is a string of question marks.

(2) In this example, the value 1234567 is simply too large to fit in the display format of $>,>>9.99. The result is a string of question marks.

(3) "Floating decimal" display format. The < symbols must follow the decimal point and be balanced by an equal or greater number of > symbols.

Date formats specify a two-digit month and a two-digit day. Date formats can specify a year in this century with two digits or four digits, while years in another century require four digits. You may use a slash (/) or a hyphen (-) as a separator. The default date format is **mm/dd/yy**.

Table 5-4 has some examples of how PROGRESS displays a date value using different formats.

Table 5-4: Date Display Format Examples

Format	Value	Display
99/99/99	3/10/1990	03/10/90
99/99/9999	3/10/2090	03/10/2090
99-99-99	3/10/1990	03-10-90
99-99-99	3/10/2090	???????? (1)

(1) In this example, the value of 3/10/2090 is too large to fit into the display format. The year part of the display format is 99 while the value being displayed is 2090.

Logical fields default to a **yes/no** format. You can change this default to any string value, such as red/black, customer/prospect, or commercial/private. Table 5-5 contains some examples of how PROGRESS displays a logical value with different formats.

Table 5-5: Logical Display Format Examples

Format	True	False
yes/no	yes	no
Yes/no	Yes	no
true/false	true	false
shipped/waiting	shipped	waiting

If you define your own logical values, note that a false value cannot begin with "y" or "t", and a true value cannot begin with a "n" or a "f". For an in-depth description of PROGRESS display formats, see Chapter 4 of the *PROGRESS Programming Handbook*.

5.5.5 Defining the Extent of an Array Field

Most fields represent a single value. **Array** fields contain multiple **elements**. For example, the `Mnth-shp` field of the item file is an array field. It contains 12 elements; one for every month of the year. The extent is the number of elements contained in an array. If you define a field with an extent greater than 0, that field is an array.

5.5.6 Defining the Label of a Field

When PROGRESS displays a field on the screen or in a printed report, it also displays a label for that field. When you are defining a field, there is a question mark ? in the Label area. If you leave that question mark and do not supply a label, PROGRESS uses the field name as the label. If you replace the question mark with a blank, PROGRESS uses no label for the field. Table 5-6 contains some examples of field labels.

Table 5-6: Examples Of Field Labels

Field Name	Label You Define	Label PROGRESS Displays
Name	?	Name
Sales-rep	Salesman	Salesman
Sales-rep	blank	

You can override labels you define in the Dictionary by using LABEL, NO-LABEL, or COLUMN-LABEL in a Format phrase or NO-LABELS in a Frame phrase. PROGRESS uses COLUMN-LABEL only for fields that do not have SIDE-LABELS.

5.5.7 Defining Decimal Places for a Field

When you define a decimal field, you must define the number of digits to the right of the decimal point. For example, `Max-credit` has been defined for two digits to the right of the decimal point, to accommodate dollars and cents. If the field is not a decimal, you simply skip this item.

The number of decimals you specify determines how many places are stored to the right of the decimal point. PROGRESS always rounds a value to this number of decimal places before storing it in the field.

5.5.8 Defining Column Labels for a Field

If you want to specify that PROGRESS use a separate label when the data is listed in columns, i.e. the label is displayed above the data, make it a column label. If you do not enter a column-label, PROGRESS uses the other label. If you do not enter a label, PROGRESS uses the field name. You can override these labels by using LABEL, NO-LABEL, or COLUMN-LABEL in a Format phrase or NO-LABELS in a Frame phrase.

If you want the column label to have more than one line (stacked), separate each line of the label with an exclamation point (!). For example, if you want the label for the `cust-num` field to be `Customer Number`, with the word "`Number`" displayed below the word "`Customer`", put `Customer!Number` on the col-label line in the Data Dictionary. If you want to use the exclamation point (!) literally as one of the characters in a column label, you must use two exclamation points ("!!"). Any space to the right or left of the exclamation point becomes part of the label.

5.5.9 Defining the Default Display Order of a Field

If you scroll through the fields in the `customer` file, you'll notice that the `cust-num`, `name`, and `address` fields have order values of 10, 20, and 30, respectively. These numbers indicate the default order in which PROGRESS displays the fields. Note that this display order doesn't have anything to do with the order that the data is stored in your database. In addition, you can override the Dictionary display order of fields in your procedures by naming the fields in the order you want to display them.

The default order numbering is in increments of 10 to let you insert fields wherever you want. The numbers do not have to be continuous and they do not have to follow in even-numbered increments. For example, if you decide to add a field called category to the customer file and you want it to appear by default on your screen between `cust-num` and `name`, you might assign 15 as its order value.

If you want to change the order number increments, to say 20, you can do so by selecting the `Renumber Fields` option in the Modify-Schema menu. From the same menu option you can also change the the order values to reflect the alphabetical order of the file names.

5.5.10 Defining the Initial Value of a Field

Each data type has an **initial value**. Whenever you create a new record for a file, each field contains this initial value. You can change a field's default initial value when you define the field. Table 5-7 contains the default initial values for fields of each data type.

Table 5-7: Default Initial Values For All Data Types

Data Type	Default Initial Value
Character	Null string (displays as blanks)
Integer	0
Decimal	0
Logical	no (false)
Date	? (unknown value - displays as blanks)

You can use a question mark (?) as a special character to represent an **unknown value**. This allows you to handle data even when some critical item of information is not yet known. If you put a single question mark in any field, PROGRESS treats that data as an unknown value.

You can also use TODAY as the initial value for a date field. Whenever a new record is created, the current date is filled in as the initial value for the field.

5.5.11 Defining a Field as Mandatory

A field that is **mandatory** cannot have an unknown value in it. It can, however, have a blank as its value. The default initial value for this item is **no**, meaning that an unknown value is allowed.

Like initial values, you can use a question mark (?) to represent an unknown value. If you accept **no** for the mandatory option, you indicate that the unknown value is an allowed value for the field.

5.5.12 A Field as a Component of a View

The `Component of -> View:` field is an information field only, it is not updatable. It tells you whether the field is used in an SQL view somewhere. When a field is used as a component of an SQL view somewhere in the database, it cannot be deleted.

5.5.13 A Field as a Component of an Index

The `Component of -> Index:` field is an information field only, it is not updatable. It tells you whether the field is a component of an index defined somewhere in the database.

5.5.14 Defining a Field as Case Sensitive

Ordinarily, PROGRESS character fields are not case-sensitive ("SMITH" = "Smith" = "smith"). However, on rare occasions, you may want to define a field that is case-sensitive. For example, part numbers that contain both upper- and lower-case characters should be stored in a case-sensitive field. Case-sensitive fields are not recommended, because they depart from standard PROGRESS usage. However, if you require strict adherence to the ANSI SQL standard, you may have to define all character fields as case-sensitive. Once a field is defined as case-sensitive, you can change it back and forth, unless it is a component of an index. If a field is a component of an index, its case-sensitivity cannot be changed unless the index is undone.

Case-sensitive fields can be indexed, and they can be grouped with case-**in**sensitive field components in an index. With case-sensitive indexes, `"JOHN"`, `"John"`, and `"john"` are all unique values. However, they do **not** sort next to each other. All upper-case letters sort ahead of all lower-case letters. Note that you can (and should) define case-sensitive variables to hold values moving to and from case-sensitive fields. See also the ANSI SQL (-Q) startup option in Chapter 3 of *System Administration II: General*.

5.5.15 Defining Validation Criteria for a Field

When a user enters a value for a field, you may want to test it to make sure it is a valid entry for the field. The **valexp** option lets you define that test, or **validation expression**. The expression must be logical. That is, it must be a valid PROGRESS expression that produces a true or false result. For example, the validation expression for the `cust-num` field is

```
cust-num > 0
```

When the user enters a customer number, PROGRESS validates that number against `valexp`. If the number is greater than 0, `valexp` is true and the validation succeeds. If the number is less than 0, `valexp` is false, the validation fails. PROGRESS then displays the text in `valmsg` (see the next section for details).

When you write procedures, you can override any validate expressions you define in the Dictionary.

5.5.16 Defining a Validation Message for a Field

If the result of `valexp` is false (the validation of a field fails), PROGRESS displays the text in `valmsg`. For example, the `valexp` for the Cust-num field is "cust-num > 0." If the user enters a customer number less than 0, PROGRESS displays this message:

```
Customer number must be greater than zero.
```

The message you define here is treated as constant (literal) text and may not contain references to the value of fields or variables. If you want to use fields, variables, or expressions in validation messages, use a VALIDATE option in a Frame phrase.

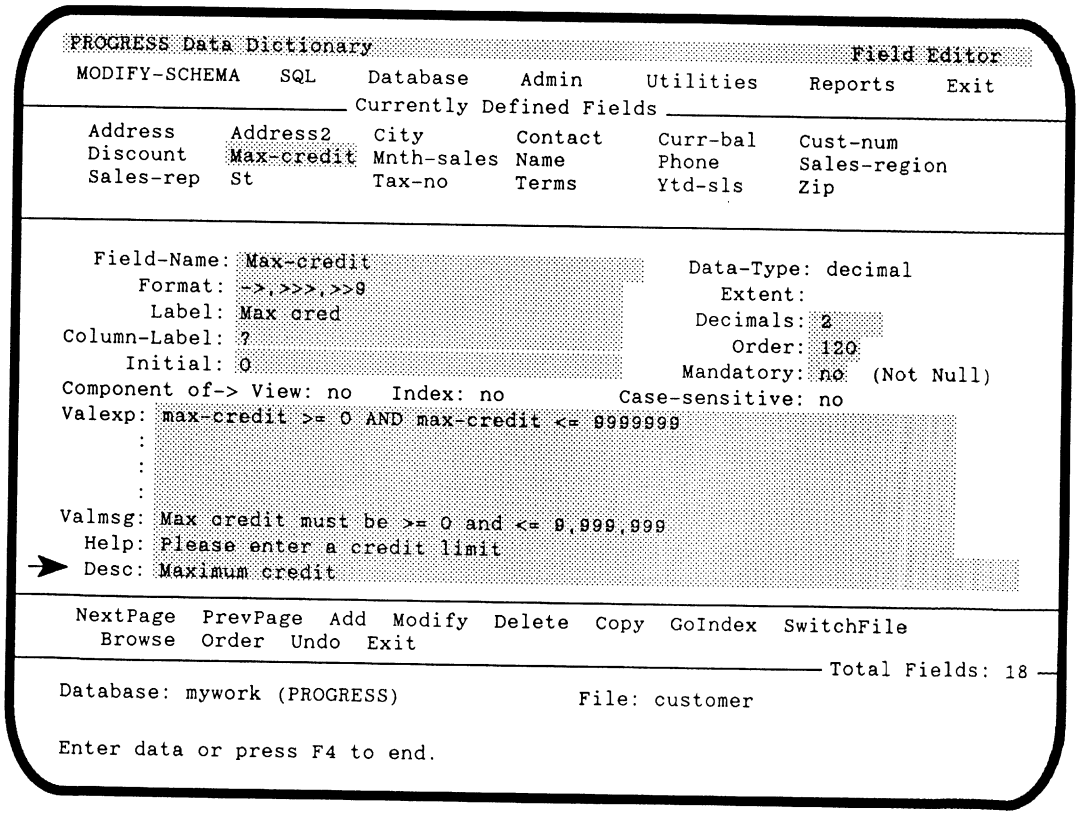
5.5.17 Defining Help Information for a Field

For certain fields, users may not be sure about the kind of data they need to enter. Therefore, you can specify a **help** message so they know what they need to enter for a field. PROGRESS displays this message whenever a user is prompted for input on the field. For example, the state field uses the help message:

```
Enter standard state abbreviation
```

5.5.18 Describing a Field

You may want to document the purpose of a field just as you might supply a description for a file. You use the desc field option to describe a field. PROGRESS does not use this option while running procedures; it is strictly to help you document your application.



5.6 ADDING A NEW FIELD TO THE CUSTOMER FILE

Now that you've looked at field definitions, you can add a new field called `Category` to the `customer` file. Press END-ERROR (F4) to return to the `Field Editor` Screen. Select the Add option by moving the highlight bar to Add and pressing RETURN.

NOTE: The `Add` option is not allowed for frozen files; you cannot change the field and index definitions of *frozen files*. The actual record data is not frozen—just the data definitions. Typically, database files are frozen when application development is completed. Files can be frozen and unfrozen from the `Utilities` menu. See Chapter 11 of the *PROGRESS Programming Handbook* for more information.

```
PROGRESS Data Dictionary                                    Field Editor
MODIFY-SCHEMA    SQL    Database    Admin    Utilities    Reports    Exit

                    Currently Defined Fields
  Address     Address2   City        Contact     Curr-bal    Cust-num
  Discount    Max-credit Mnth-sales  Name        Phone       Sales-region
  Sales-rep   St         Tax-no      Terms       Ytd-sls     Zip


   Field-Name:                                  Data-Type:
       Format:                                     Extent:
        Label:                                  Decimals:
  Column-Label: ?                                  Order:
      Initial:                                  Mandatory:      (Not Null)
  Component of-> View:      Index:          Case-sensitive:
  Valexp: ?
        :
        :
        :
  Valmsg:
    Help:
    Desc:

   NextPage  PrevPage  Add  Modify  Delete  Copy  GoIndex  SwitchFile
      Browse  Order  Undo  Exit
                                                        Total Fields: 18

  Database: mywork (PROGRESS)              File: customer

  Enter data or press F4 to end.
```

This selection lets you add fields in the customer file as shown in the next screen.

Assume that some of the customers in your customer file are businesses and others are private consumers. You might want to place these customers into two different categories: commercial (the businesses) or private (the private consumers). Since it is a good idea to assign a field name according to what kind of information it will contain, type **Category** and press RETURN as shown:

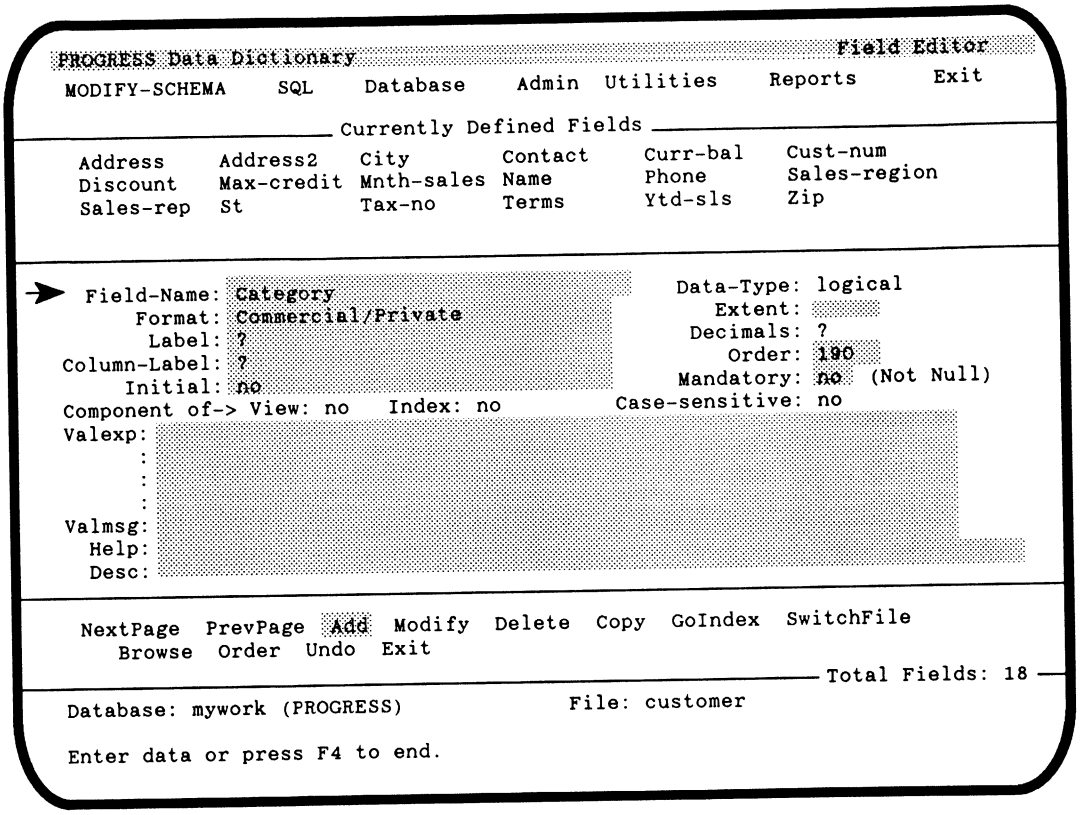
```
PROGRESS Data Dictionary                                    Field Editor
 MODIFY-SCHEMA   SQL     Database    Admin    Utilities    Reports    Exit

                        Currently Defined Fields
  Address     Address2   City        Contact    Curr-bal    Cust-num
  Discount    Max-credit Mnth-sales  Name       Phone       Sales-region
  Sales-rep   St         Tax-no      Terms      Ytd-sls     Zip

→ Field-Name: Category                          Data-Type:
      Format:                                      Extent:
       Label:                                   Decimals:
Column-Label: ?                                    Order:
     Initial:                                   Mandatory:      (Not Null)
Component of-> View:       Index:           Case-sensitive:
Valexp: ?
        :
        :
        :
Valmsg:
  Help:
  Desc:

  NextPage  PrevPage  Add  Modify  Delete  Copy  GoIndex  SwitchFile
     Browse  Order  Undo  Exit

Database: mywork (PROGRESS)              File: customer      Total Fields: 18

Data types: Character DAte DEcimal Integer Logical
```

Now choose a Data-Type. Notice the message at the bottom of your screen:

```
Data types: Character DAte DEcimal Integer Logical
```

Since category can have one of two values (commercial or private), define the field as logical. Try typing **l** (the letter l), and then press RETURN. Similarly, you can type **c** for character, **da** for date, **de** for decimal, or **i** for integer. (You can type them in uppercase too.)

When you choose logical as a data type, the format defaults to yes/no. Because a customer falls into one of the two categories (commercial or private) change the format to read **Commercial/Private** as shown:

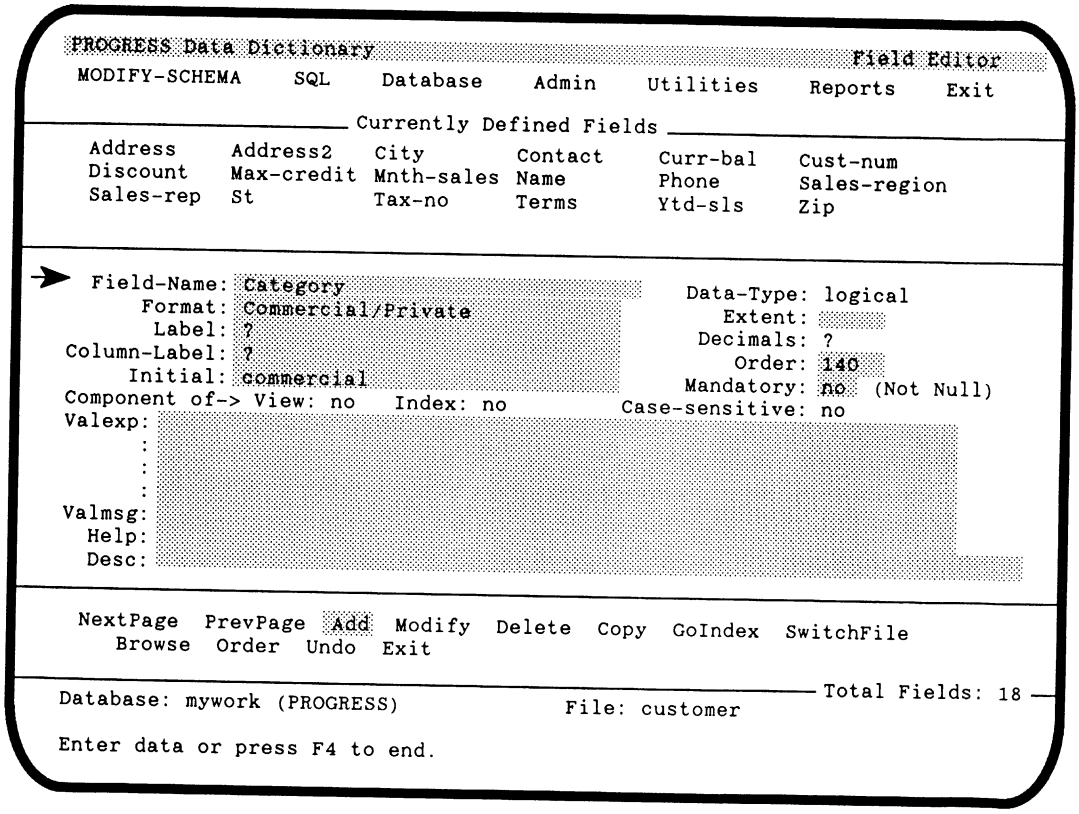


Press RETURN after entering this change.

The cursor is now on the `Label` field. If you don't want to specify a label, PROGRESS uses the field name `Category` as the field label. Just press RETURN.

Now the cursor is on the Column-label option. If you don't want to specify a column label, PROGRESS uses the other label. Just press RETURN.

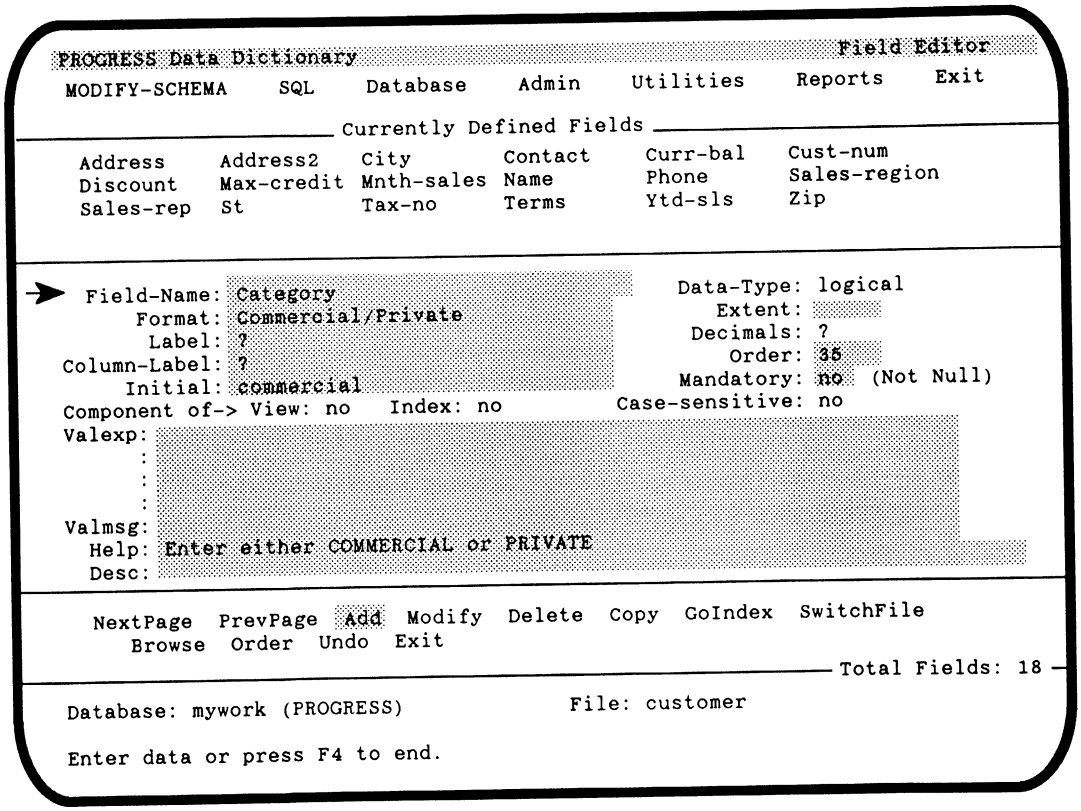
The `Initial` value of a logical field defaults to **no** (false). Since most or all of the customers are commercial (a "yes" or "true" value), change the default value from no to `commercial` by typing **commercial** as shown in the following screen and pressing RETURN.



Since Category is not an array field, you can skip over Extent by pressing RETURN.

You might want Category to appear between City and Contact when you use the PROGRESS statement DISPLAY field. Change the Order to **35** field as shown on the following screen and press RETURN.

Skip over Mandatory, Valexp and Valmsg, but stop when you get to the Help field. You might want to tell the user the valid options for the field. Type **Enter either COMMERCIAL or PRIVATE** as shown on the following screen and press RETURN.



To save this new field definition, press GO (F1). PROGRESS displays a message letting you know that the new field was added successfully, and allows you to add another field. Press END (F4) to returns to the Field Editor screen.

If you made any mistakes while entering the field definitions, you can correct them by selecting Modify, to make changes to the definitions. If you want to look at all of your field definitions, choose Browse.

Now that you've worked with field definitions, you can examine the indexes. If you want to take a break before continuing, you can press END-ERROR (F4).

NOTE: You can use the STOP key (CTRL-BREAK if you are using DOS or OS/2, usually CTRL-C if you are using UNIX or Action-Cancel if you are using BTOS/CTOS) for a quicker return to the editor; however, PROGRESS will now save your additions or changes to the Dictionary. To save your changes, use END-ERROR (F4) to exit from the Dictionary and apply your changes at the menu shown in the following screen.

The Data Dictionary displays the following menu. This menu allows you to either apply or undo you changes, or to return to the editing files, fields, or indexes:

```
PROGRESS Data Dictionary                                    Main Menu
  MODIFY-SCHEMA  SQL  Database   Admin    Utilities    Reports    Exit

                    M. Modify Existing File
                    F. Field Editor
                    I. Index Editor

                    A. Apply Changes
                    U. Undo Changes

        Once you leave this menu, all schema changes to database
        "mywork" will be applied.  This may take some time.  If you
        wish, you can select another change from this menu and include
        it in the transaction.  Selecting "Apply Changes" will apply
        the changes to your database.

Database: mywork (PROGRESS)                  File: customer

Apply Changes to Database
```

Press RETURN to apply your changes to your mywork database. Press END (F4) to return to the PROGRESS Editor.

5.7 DEFINING INDEXES FOR A FILE

In Chapter 1, you learned that an index is a field or combination of fields PROGRESS uses to rapidly retrieve a particular record in a file. A single index can be made up of multiple fields, or components. For example, if you know you are always going to access the order-line file by using a combination of the `order-num` and `line num` fields, you might want to create an index with those two fields as its components.

Your `mywork customer` file uses three fields as indexes:

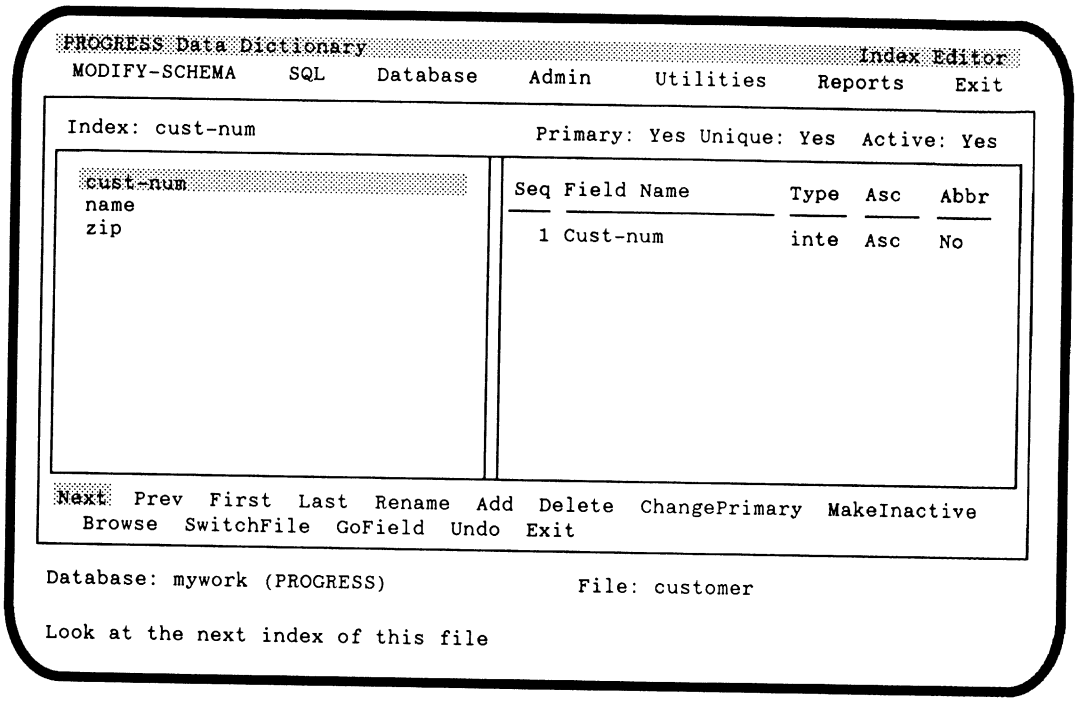
- `name`
- `cust-num`
- `zip`

When you're ready to start the Dictionary again, type **dict** in the edit area and press GO (F1). Or, access the Dictionary through HELP by pressing HELP (F2) and typing **d**. Then press RETURN.

Select **M**, `Modify-Schema` from the Data Dictionary Main Menu, and then choose **I**, `Index Editor` from the submenu as shown below:

```
PROGRESS Data Dictionary                                    Main Menu
 Modify-schema  SQL   Database   Admin   Utilities   Reports   Exit

 M. Modify Existing File
 C. Create New File
 D. Delete File(s)

 F. Field Editor
 R. Renumber Fields
 G. Global Name Change

 I. Index Editor

Database: mywork (PROGRESS)              File:

Add, alter or remove indexes for a file
```

The dictionary displays a list of files for you to choose from. Type **c** to choose `customer` as the file you want to work with. The press RETURN. The dictionary displays a list of indexes for the customer file. The cust-num index is the first index definition you see as shown in the following screen.

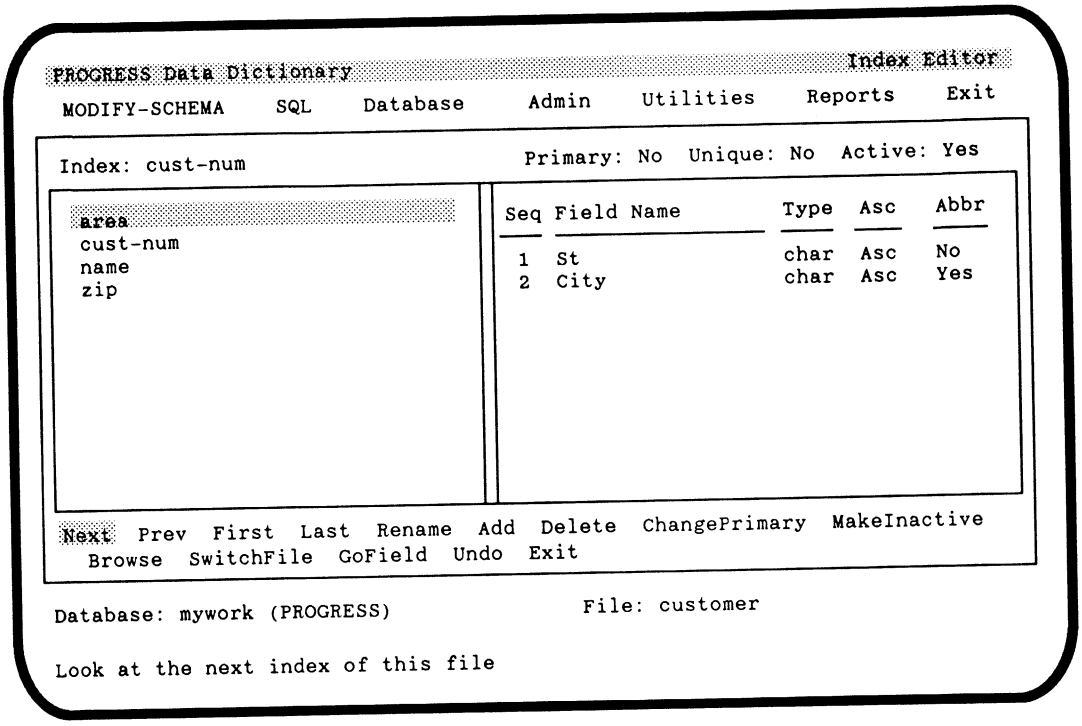


The customer file uses the `cust-num` field to define the cust-num index. The **Index** name doesn't have to be the same as the field name, but it's easier to remember if you give it the same name.

An index is **unique** if one and only one record in a file has a particular index value. For example, the `customer` file cannot contain two records with the same customer number.

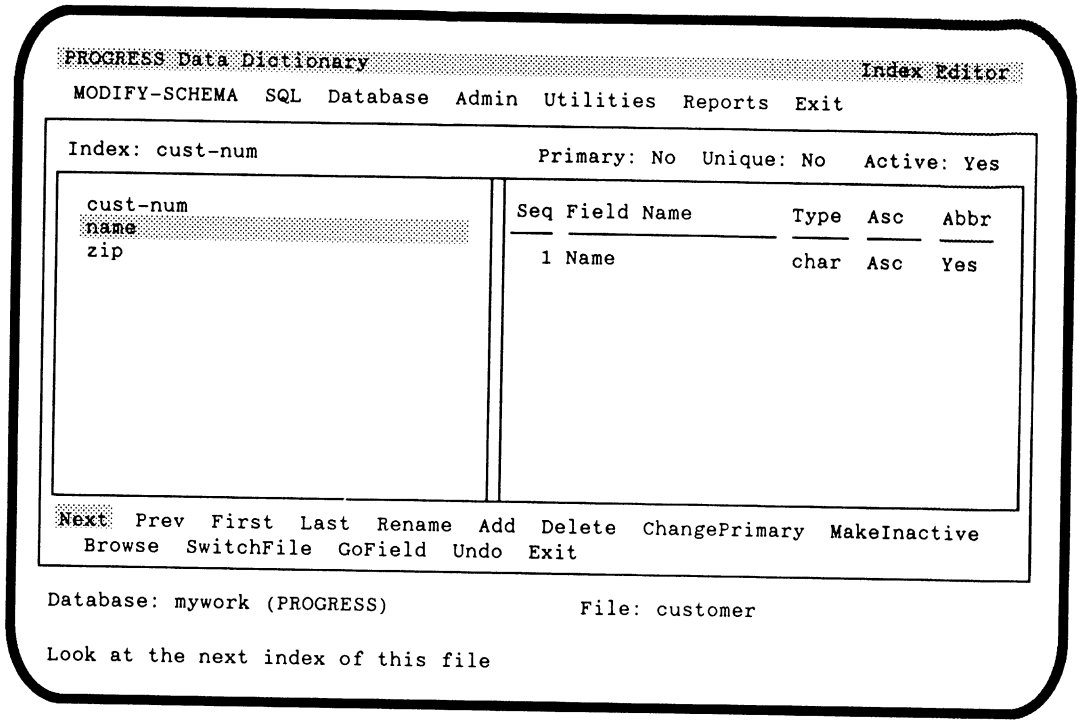
In Chapter 1, you learned that the **primary index** is the index most frequently used to retrieve a record or to order the records for a report. `Cust-num` has been defined as the primary index.

Seq refers to the number of fields used to define an index. For example, suppose you defined an index called "area" made up of the `city` and `st` fields. `Seq` might look like this:



You can define an index to sort records in either **ascending** (yes) or **descending** (no) order. Because `cust-num` has been defined as an ascending order index, PROGRESS sorts customer numbers from lowest to highest.

The next characteristic you'll learn about is **abbreviate.** Make sure that `Next` is highlighted and press RETURN to see the definition for the name index.



Suppose you were looking for the record for Shark Snack Snorkeling, but couldn't remember its full name. You might simply want to enter "Shark" to find it. **Abbreviate** is an index option that lets you conveniently search for a partial match based on the first few characters of a field, if the field is a character data field. As you'll see later, even if you don't indicate Abbreviate in the index definition, you can specify that you want to lookup a record or series of records based on a partial match. Indexes not comprised of character data require an exact match.

Also note that the name index is not a unique index. More than one customer might have the same name. Records are ordered by the name index in ascending alphabetical order.

Press RETURN to view the definition for the zip index, then press END-ERROR (F4) to return to the Data Dictionary Main Menu.

5.8 ADDING A NEW INDEX TO THE CUSTOMER FILE

Suppose you plan to frequently sort your customer information by sales region or you want to frequently access data only for customers in a particular region. You can do this more easily by defining the sales-region field as an index. Start the index definition by typing **I** to select the `Index Editor` option from the `Modify-Schema` menu.

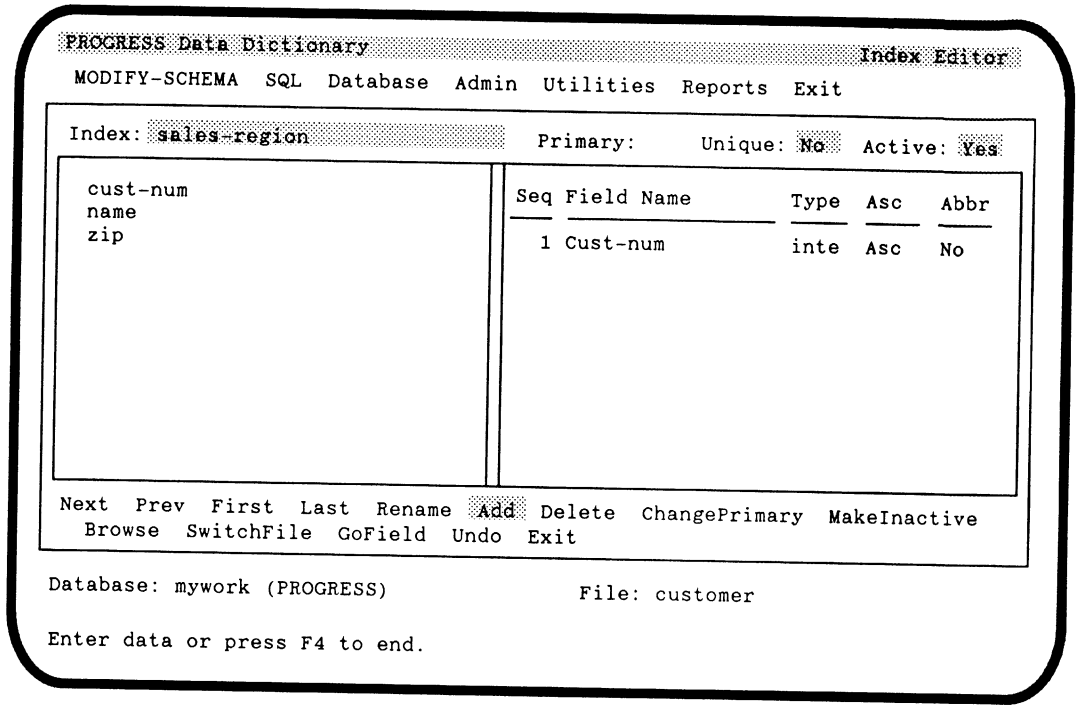
The dictionary displays a list of files for you to choose from as shown below. Type **c** to choose the `customer` file as the file you want to work with. Then press RETURN. The Dictionary then displays a list of indexes for the `customer` file. The `cust-num` index is the first index definition you see.

```
PROGRESS Data Dictionary                                    Index Editor
 MODIFY-SCHEMA  SQL  Database  Admin  Utilities  Reports  Exit

Index: cust-num                    Primary: Yes Unique: Yes  Active: Yes
 cust-num                  Seq Field Name        Type  Asc   Abbr
 name                        1 Cust-num          inte  Asc   No
 zip
 Next  Prev  First  Last  Rename  Add  Delete  ChangePrimary  MakeInactive
   Browse  SwitchFile  GoField  Undo  Exit

Database: mywork (PROGRESS)                File: customer

Look at the next index of this file
```

Type **a**, to select the Add option. Or you can use your space bar or arrow keys to highlight Add option as shown and press RETURN.

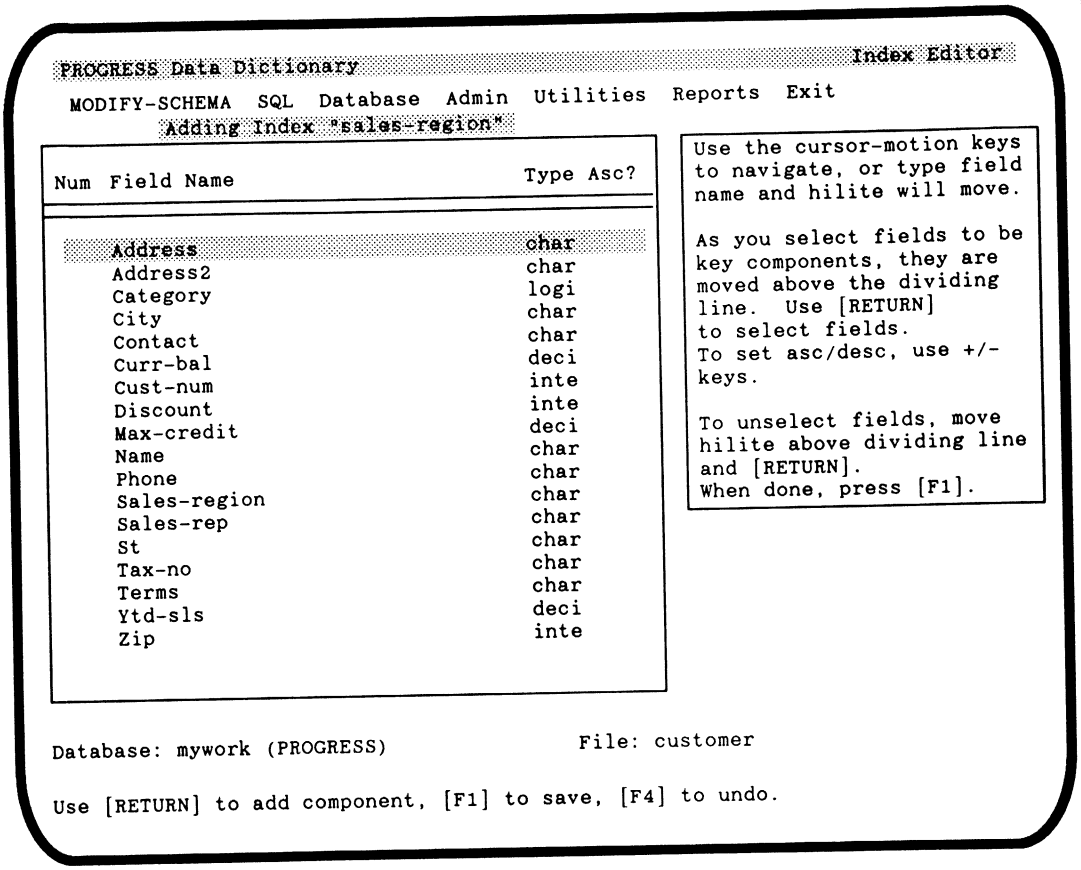
```
Next  Prev  First  Last  Rename  Add  Delete  ChangePrimary  MakeInactive
  Browse  SwitchFile  GoField  Undo  Exit

Database: mywork (PROGRESS)                File: customer

Look at the next index of this file
```

The dictionary prompts you for the Index name. Type **sales-region** and press RETURN.



Since many customers may be in the same sales region, sales-region should not be a unique index. Press RETURN to accept the no default. You want this to be an active index, so press RETURN to accept the yes default and to set the basic definition for the sales-region index.

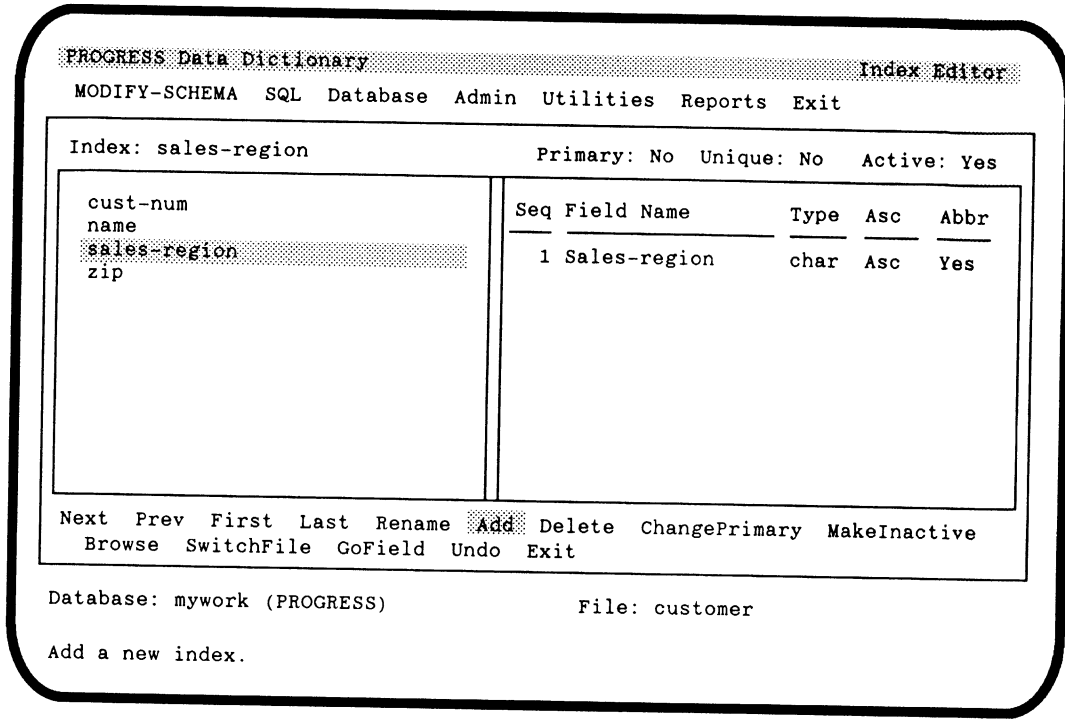
The screen display then asks you for the components of the index:



Because this is a one-field index, you have three remaining steps:

1. Type **s** to highlight the **Sales-region** field and press RETURN .
2. Accept the default ascending order (**yes**) and press GO (F1).
3. Select **Yes** to the prompt asking if you want it to be an abbreviated index.

After you've specified the Abbreviate index component, PROGRESS displays the current indexes for the customer file, including the newly defined sales-region index.



Since you aren't going to define any more indexes, press END-ERROR (F4) and apply your changes by selecting a at the menu. When PROGRESS finishes applying your changes, it returns you to the Data Dictionary Main Menu.

5.9 RENUMBERING FIELDS

You can use the Renumber Fields option in the Modify-Schema menu to resequence the _Order of fields in a file. That is, you can define whether the fields will be sorted alphabetically by field name or by the order defined by the values in their _Order fields.

If you want to change the order number increments, to say 15 or 20, you can do so by selecting the Renumber Fields option in the Modify-Schema menu. From the same menu option you can also change the the order values to reflect the alphabetical order of the file names.

5.10 CHANGING A FIELD NAME THROUGHOUT THE DATABASE

You can use the Global Name Change option in the Modify-Schema menu to change the name of a field throughout a database. If you try to change the name of a field to a name that's already in use somewhere in the database, the Data Dictionary will display an error message and will not accept the new field name. This helps you to maintain the integrity of your database.

5.11 CHANGING YOUR DICTIONARY DEFINITIONS

The following are rules for changing Dictionary definitions:

- When you change the initial value of a field, it is not changed in any of the records containing that field. However, all records you create after the field change use the new initial value.
- Changing the format of a field does not affect the data stored in the database. If you really want to truncate existing data values, you must write a procedure for it and use functions such as SUBSTRING, ROUND, and TRUNCATE to process the data.
- You cannot change the data type or array extent of a field. If you want to change a field's data type or array extent, define a new field with the new data type or array extent information. Write a procedure to copy the data from the existing field into the new field, doing the appropriate conversion. Then delete the old field and rename the new one.
- You can change the name of an index at any time. You can also delete indexes. However, before you delete a primary index, PROGRESS requires that you designate another index as the primary index. You cannot change any of the component definitions of an index. Instead, you must delete the index, recreate it, and specify the new component definitions.
- If you add new fields to an existing file, PROGRESS makes a pass through all the records in the file to initialize the new fields.
- If you add a new index to an existing file, PROGRESS makes a pass through all the records in the file and creates the new index automatically.
- If you add or delete several fields and/or add or delete several indexes for a single file at once, PROGRESS makes all the changes to existing records in a single pass of the file.
- Changing database definitions does not invalidate any precompiled procedures that were compiled against the database. However, adding or deleting files, fields or indexes does invalidate those precompiled procedures that reference the file you changed. In this case, you must erase the object files and use the source procedures, or recompile the procedures. See Chapter 13 in this manual for more information about compiling procedures.

5.12 SAVING AND PRINTING DATA DEFINITIONS

The definitions you create are saved when you apply the changes from the menu you encounter as you leave the Modify-Schema portion of the Data Dictionary. If your printer is on-line and you would like a printed report of the data definitions, select **Reports** from the Data Dictionary Main Menu. Then select option **D**, `Detailed Fields Report`, and select **customer** as the file name. You may send the report to a file or to your printer. When you are finished, choose the `Exit` option.

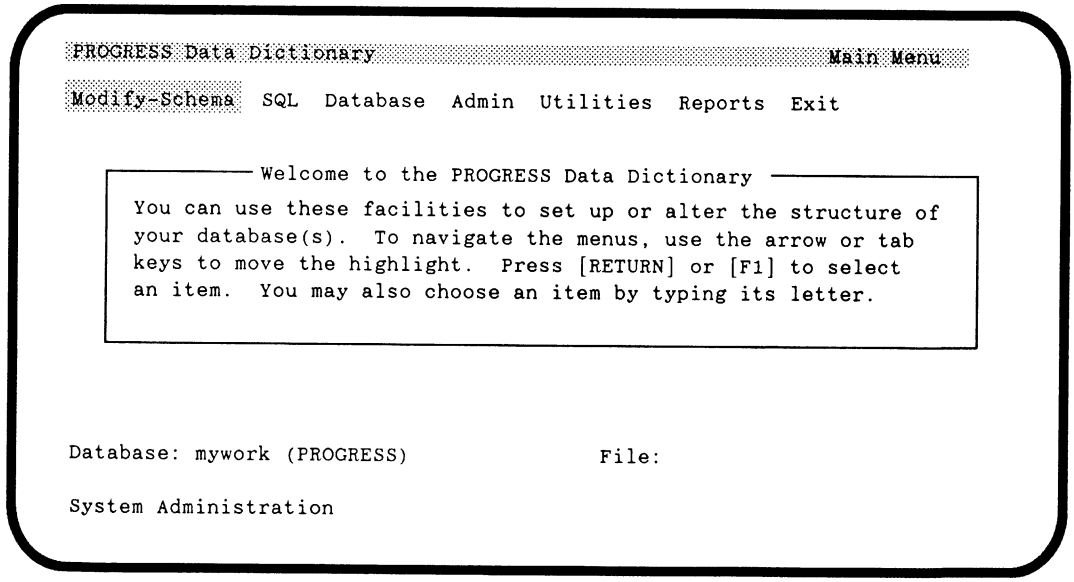
5.13 USING THE DICTIONARY IN MULTI-USER MODE

Only one user at a time can change the file and field definitions of a database and then only if all other users of the database are using the PROGRESS editor or running procedures that do not access the database (for example, menu procedures do not usually access the database). Usually it is best to run the Data Dictionary in single-user mode and to avoid changing the file, field, and index definitions of a database that is actively in multi-user mode.

You can find more information about multi-user PROGRESS in Chapter 12 of the *PROGRESS Programming Handbook.*

5.14 USING OTHER DICTIONARY FUNCTIONS

The last sections have described the first option on the Dictionary Main Menu, `Modify-Schema`. The other options are:



SQL If you are using an ORACLE database, or if you are using SQL statements to access any database, be familiar with the SQL submenu. It has options that involve views created with PROGRESS/SQL statements, as well as options that provide an easy way to generate SQL language `CREATE VIEW` and `CREATE TABLE` statements. For details on these options, see Chapter 15, "PROGRESS/SQL," in the *Programming Handbook.*

Database

Lets you select a working database, connect or disconnect a database, copy a database, or use utilities specific to PROGRESS, ORACLE, or RMS. Use the `Database` submenu to create, connect, and disconnect databases. Database connection/disconnection is important information to know if you are working with multiple databases. For more information, see Chapter 3 of this book, Chapter 13, "Multi-Database Programming," in the *Programming Handbook* and Chapter 2, "Startup and Shutdown," in *System Administration II: General.*

Admin

Use the `Admin` submenu to dump and load database data and definitions, export and import database data, and implement a database security scheme. You can export data records in any of the following formats: DIF, SYLK, ASCII, WordStar, Microsoft Word, WordPerfect, and BTOS OfisWriter. You can import records from files having any of the following formats: DIF, SYLK, ASCII, and dBASE II/III/III + /IV.

Database dump/load procedures are documented in Chapter 4 of *System Administration II: General.* Data import/export is documented in Chapter 9 of the *PROGRESS Programming Handbook*, and the IMPORT and EXPORT statements are documented in the *PROGRESS Language Reference Manual.* The security option lets you define access permissions for the files and fields in your database, set up a security administrator, and create and work with userids and passwords. For detailed information on database security, see Chapter 5 in *System Administration II: General* and Chapter 11 of the *PROGRESS Programming Handbook.*

Use the `Create Bulk Loader Description File` option to automatically write a Bulk Loader description file. You must create a Bulk Loader description file before you can run the Bulk Loader utility. For more information about the `Create Bulk Loader Description File` option, see Chapter 4 in *System Administration Guide II: General.*

Utilities Lets you edit the parameter files, freeze and unfreeze database files, interface to the quoter utility, generate include files for use with the 4GL, and view information about the current status of the system. Parameter files are described in Chapter 3 of *System Administration II: General*. *Frozen files* are described in Chapter 4 of *System Administration II: General*. See the following entries in the *PROGRESS Language Reference* for more information about generating include files: { }, ASSIGN, FORM, and DEFINE WORKFILE.

Reports Gives you reports of file, field, and index definitions, defined users, views defined in the current database, and implied relations between files. Use the `Reports` submenu to review your database structure, to list registered database users, or to list all file relations in the database. Chapter 5, which covers database design and structure (data definitions), discusses the file, field, and index report options. Chapter 6, "Working with Related Files," describes the `List of File Relations` choice. Chapter 5 in *System Administration II: General* describes the `User Review` choice.

Exit Leaves the Dictionary and returns you to the editor. You can also exit by pressing the END-ERROR (F4) key.

5.15 USING THE PROGRESS QUERY/RUN-TIME DATA DICTIONARY

PROGRESS Query/Run-Time provides a restricted version of the Dictionary. When you start this version of the Dictionary, the options that allow you to dump and load data definitions, import and export data, and change schema definitions are not available.

5.16 SUMMARY

This chapter described how to:

- Define files
- Define fields
- Define indexes
- Change database definitions

You should now know how to build an application database and are well on your way to building an application.

Chapter 6
Working with Related Files

Once you have your data definitions in place, you can begin building an application. The first step is to understand and make use of the relationships between the files you have defined. To help you understand these relationships, this chapter covers the following topics:

- File Relationships
- Using Nested Blocks
- Printing a Report
- Putting it All Together with Menus

Your mywork database contains many files. The four we'll be using most of the time are the customer, order, order-line, and item files. You've already seen the definitions for these files in Chapter 1. Let's look more closely at the relationships between them. It is important to understand these relationships before you begin to write procedures so that you can take advantage of PROGRESS' ability to read records within iterating blocks.

If you exited from PROGRESS after the last chapter, you need to get back into your **mywork** database. At the system prompt, start PROGRESS by typing the following command and pressing RETURN:

Table 6–1: Command to Start PROGRESS

Operating System	Starting mywork Database
UNIX, DOS, and OS/2	`pro mywork`
VMS	`PROGRESS mywork`
BTOS/CTOS	`PROGRESS 4GL [Options] -1 mywork`

6.1 FILE RELATIONSHIPS

Related files have at least one field that contains common information. The related field is always an indexed field in at least one of the files. This field often has the same name in both files. For example:

- Cust-num is a field in both the customer and order files.
- Order-num is a field in both the order and order-line files.
- Item-num is a field in both the order-line and item files.

There are four types of relationship that can be established between files; one-to-one, one-to-many, many-to-one, and many-to-many. The type of relationship can determine what type of block statement you use, iterating or non-iterating, record-reading or non-record-reading.

6.1.1 The One-to-One Relationship

There is a one-to-one relationship between the order-line and item file. That is, for any one order-line there is only one item related to that order.

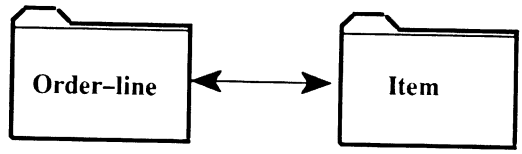


Figure 6-1: The One-to-One Relationship

6.1.2 The One-to-Many Relationship

There is a one-to-many relationship between the customer file and the order file because, for any one customer, there can be many orders.

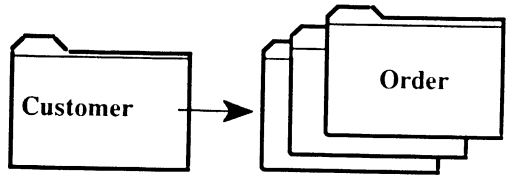


Figure 6-2: The One-to-Many Relationship

6.1.3 The Many-to-One Relationship

The opposite is true for the relationship between the order file and the customer file. That is, many order records can be related to the same customer record. Therefore, the order file has a many-to-one relationship to the customer file.

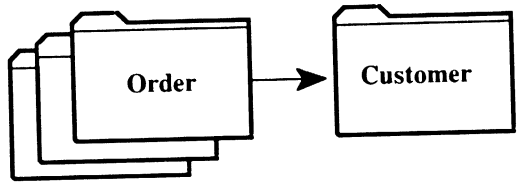


Figure 6–3: The Many-to-One Relationship

6.1.4 The Many-to-Many Relationship

None of the files defined in the demo database have a many-to-many relationship. You could, however, define a warehouse file that describes the location of each item in the warehouse. If a warehouse can store many items and an item can be stored in many warehouses, then these two files would have a many-to-many relationship.

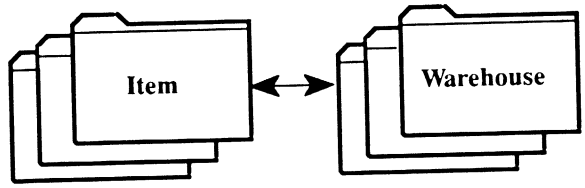


Figure 6–4: The Many-to-Many Relationship

For the sake of good database design when accessing information in files that have the many-to-many relationship, you should create a cross-reference file that contains the related data. For example, for the item and warehouse files, you would create a cross-reference file containing an item number and a warehouse identification. Without this cross-reference file, you would have to store a lot of redundant information or create array fields in both the item and warehouse files. In either case, your database would not be "normalized." Chapter 3 of the *PROGRESS Programming Handbook* contains more information about normalization, the process of making your database design more efficient.

6.2 USING NESTED BLOCKS

Chapter 4 introduced you to the block structure of the PROGRESS application language. Each procedure is a block in itself. Within a procedure, you can have other blocks, which begin with a header statement and end with the END statement. These blocks, in turn, can contain other blocks. In this way, blocks can be **nested** within each other.

Chapter 4 also described some of the characteristics that a block may posses. That is, a block may **loop**, it may read a record from the database, or both; or it may posses neither looping nor record-reading properties. The FOR EACH block, for example, loops and reads another record from the database each time it iterates. A FIND block, on the other hand, reads a single record from the database and stops. By nesting FIND and FOR EACH blocks in different ways, you can look up related records in multiple files having any of the relationships described in the previous section.

6.2.1 Locating Customer's Order Records

Suppose you want to display the name of each customer in your customer file and update information for each of the orders belonging to those customers. Because the customer and order files are related to one another by means of a common field (cust-num), it's easy to write a procedure to handle this task. Remember that the customer file has a one-to-many relationship to the order file because an individual customer may place many orders but the same order cannot be placed by many customers.

The first step is to locate a customer in the database and display the customer name. This can be done with a FOR EACH block, which loops to read customer records from the database, one at a time.

```
FOR EACH customer:
  DISPLAY name.
END.
```

After displaying a customer name, you want to update the orders belonging to that customer. This is done with a second FOR EACH block, which must somehow use the same customer number as is used in the previous FOR EACH block.

```
FOR EACH order OF customer:
    DISPLAY order-num.
    UPDATE sdate pdate.
END.
```

customer" is the key phrase here; it indicates that the order records are related to the customer record that was previously read. In order to "inherit" this customer number, the second FOR EACH block must be nested within the first, as shown below. (If you want to look at this procedure in your edit area, press GET (F5) and type **t-nested.p**.)

t-nested.p

```
FOR EACH customer:
  DISPLAY name.

  FOR EACH order OF customer:
    DISPLAY order-num.
    UPDATE sdate pdate.
  END.
END.
```

The arrows in the listing show how PROGRESS executes this procedure. It first reads a customer record and displays the customer name. When it enters the second FOR EACH block, it uses the customer number to look up orders in the order file. This inner block loops until it has read all orders for that customer. When the inner block ends, the procedure returns to the beginning of the outer FOR EACH block to read the next customer, display the name, and look up the orders for that customer. The procedure continues looping in this way until all customer records have been processed.

If you'd like to see how this procedure works, run it by pressing GO (F1).

6.2.2 Locating Selected Customer Orders

Suppose you want to display additional fields from each customer record and process only those customer records where the maximum credit limit is over $500. This can be done with the WHERE option, which was introduced in Chapter 4:

t-nestd2.p

```
FOR EACH customer WHERE max-credit > 500:
  DISPLAY name address city st zip max-credit.
  FOR EACH order OF customer:
    DISPLAY order-num.
    UPDATE sdate pdate.
  END.
END.
```

Selects only those records meeting the selection criteria.

The WHERE option specifies the criteria used for selecting the records you want to process. By using the expression "WHERE max-credit > 500," PROGRESS skips any record that does not have the specified characteristics.

Press GET (F5) and type **t-nestd2.p** if you want to retrieve this procedure. To run it, press GO (F1). When you have finished working with this procedure, press END (F4).

6.2.3 Sorting Customer Orders

Now, suppose you want to sort the records by zip code. The procedure t-nestd3.p does this kind of sorting. In t-nestd3.p, the BY option names a field, or series of fields, for PROGRESS to use when sorting records. Here, BY uses the zip field to order the customer records for display.

t-nestd3.p

```
FOR EACH customer WHERE max-credit > 500 BY zip:
  DISPLAY name address city st zip max-credit.
  FOR EACH order OF customer:
    DISPLAY order-num.
    UPDATE sdate pdate.
  END.
END.
```

Sorts records in a specific order.

6.2.4 Locating an Order's Customer Record

The last sections showed you how to use FOR EACH blocks to find records having a one-to-many relationship. You used a FOR EACH block to read individual customer records and then, nested within that FOR EACH block, you used another FOR EACH block to read each order record associated with a particular customer.

Suppose that now you want to relate order records to customer records. The order file has a many-to-one relationship to the customer file. That is, instead of reading a customer record and finding all the related order records, you want to read an order record and find the single customer record associated with that order.

To find order records, you must still use a FOR EACH block because you want the procedure to process all order records, one at a time. However, to locate the customer record associated with that order, you can use a FIND statement. The FIND statement reads a single record; it does not loop. Here is the procedure; you can find it in the t-nestd4.p procedure file:

t-nestd4.p

```
FOR EACH order WHERE pdate < TODAY:
  FIND customer OF order.
  DISPLAY order-num odate pdate customer.name customer.sales-rep
        WITH TITLE "Past Due Orders".

END.
```


The outer FOR EACH block reads an order record in the order file. Then the FIND statement uses the OF option to tell PROGRESS to find the customer record associated with that order record. The procedure displays certain fields from the records and then locates the next order in the database.

You should note that the FIND statement fails if it cannot find a single, unique record that satisfies the selection criteria. That is, if more than one customer record has the same order number, the FIND statement would fail. In contrast, a FOR EACH statement will succeed no matter how many records it can find.

6.2.5 Finding Relationships across Three Files

Why not write your own procedure using nested blocks? Try writing a procedure to do the following:

1. Display the name, address, city, state, zip code, and maximum credit limit of each customer with a credit limit greater than $500.

2. Display the order-num, pdate, and sdate fields of each order belonging to each customer.

3. Display the line number, item number, item description, quantity ordered, and price of each item on each order.

When you finish writing your procedure, press PUT (F6) and and save it with the name **t-myproc.p**. Press GO (F1) to run and test the procedure. If you encounter error messages, don't hesitate to use HELP by pressing HELP (F2) to find out how to correct any mistakes you may have made.

How did you do? Compare your procedure against this one, called **t-itlist.p,** which you can retrieve with GET (F5):

t-itlist.p

```
FOR EACH customer WHERE max-credit > 500:
  DISPLAY name address city st zip max-credit.
  FOR EACH order OF customer:
    DISPLAY order-num pdate sdate.
    FOR EACH order-line OF order:
      FIND item OF order-line.
      DISPLAY line-num item.item-num idesc qty price.
    END.
  END.
END.
```

This procedure is made up of three nested blocks:

- `FOR EACH customer WHERE max-credit > 500:`
- `FOR EACH order OF customer:`
- `FOR EACH order-line OF order:`

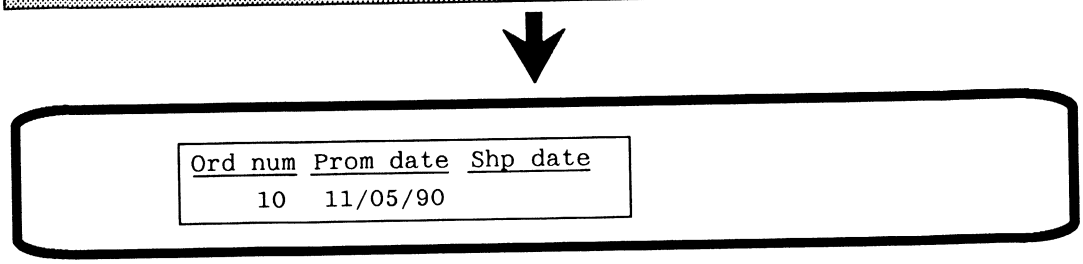
The outer FOR EACH block selects each customer record that has a maximum credit limit that is greater than $500. The DISPLAY statement that follows displays the name, address, city, state, zip code, and max-credit fields.

```
FOR EACH customer WHERE max-credit > 500:
    DISPLAY name address city st zip max-credit.
```

Name	Addr	City	State	Zip	Max-cred
Second Skin Scuba	79 Farrar Ave	Yuma	AZ	85369	1,500

The second block displays the order number, promised delivery date, and shipping date for each order belonging to that customer:

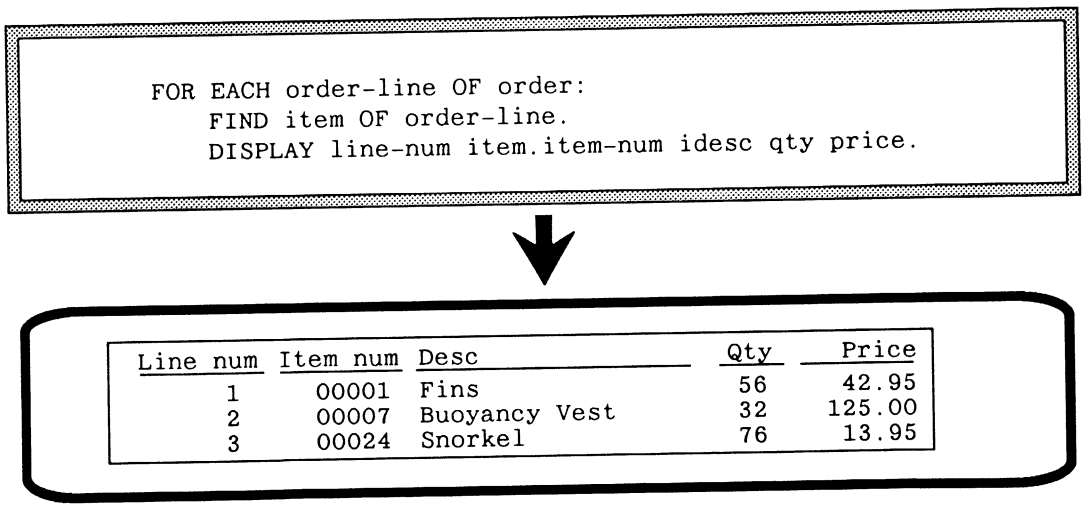
```
FOR EACH order OF customer:
    DISPLAY order-num pdate sdate.
```



Ord num	Prom date	Shp date
10	11/05/90	

The third block header statement finds each order-line associated with the current order record. The FIND statement following the block header finds the item record associated with the order-line record you just read.

```
FOR EACH order-line OF order:
    FIND item OF order-line.
    DISPLAY line-num item.item-num idesc qty price.
```



Line num	Item num	Desc	Qty	Price
1	00001	Fins	56	42.95
2	00007	Buoyancy Vest	32	125.00
3	00024	Snorkel	76	13.95

6.3 PRINTING A REPORT

So far, all the procedures we've worked with have sent output to the screen. PROGRESS doesn't require that you learn one set of statements for screen displays and another for printer output. The statement

```
OUTPUT TO PRINTER.
```

directs all of the output of your procedure to the printer. When the procedure is finished, the statement

```
OUTPUT CLOSE.
```

redirects output back to the default output destination. The default output destination is usually your terminal.

Adding the OUTPUT TO PRINTER statement to t-itlist.p lets you create a printed list of customer orders and the items in each order (this modified version of t-itlist.p is called t-rplist.p).

t-rplist.p

```
OUTPUT TO PRINTER.

FOR EACH customer WHERE max-credit > 500:
  DISPLAY name address city st zip max-credit.
  FOR EACH order OF customer:
    DISPLAY order-num pdate sdate.
    FOR EACH order-line OF order:
      FIND item OF order-line.
      DISPLAY line-num item.item-num idesc qty price.
    END.
  END.
END.

OUTPUT CLOSE.
```

If your printer is on-line and ready to go, run t-rplist.p to get a printed report of customer and order information.

6.4 PUTTING IT ALL TOGETHER WITH MENUS

You've worked with a number of procedures that let you display, update, or delete customer and order information. Now let's tie them together with the menu procedure called t-csmenu.p.

t-csmenu.p

```
DEFINE VARIABLE Selection AS INTEGER FORMAT "9".                    } 1
REPEAT:
  FORM                                                              }
      SKIP(2) "      M A I N  M E N U                 "             }
      SKIP(1) "   1) Add a new customer               "             }
      SKIP(1) "   2) Change customer information      "             }
      SKIP(1) "   3) Display orders                   "             } 2
      SKIP(1) "   4) Delete a customer                "             }
      SKIP(1) "   5) EXIT                             "             }
      WITH CENTERED                                                 }
           TITLE "Maintenance and Reporting".                       }
  UPDATE SKIP(2) SPACE(1) selection AUTO-RETURN                     } 3
    WITH SIDE-LABELS.
4 { HIDE.
       IF selection EQ 1 THEN RUN t-adcust.p.                       }
  ELSE IF selection EQ 2 THEN RUN t-chcust.p.                       }
  ELSE IF selection EQ 3 THEN RUN t-itlist.p.                       } 5
  ELSE IF selection EQ 4 THEN RUN t-delcus.p.                       }
  ELSE IF selection EQ 5 THEN QUIT.                                 }
  ELSE MESSAGE "Incorrect selection-please try again".              }
END.
```

There's a lot of new information in this procedure, but don't worry about learning it all just yet. For now, we'll concern ourselves with gaining a general understanding of menu procedures and how they work:

1 When you choose an item from a menu, you want to perform a task based on that menu selection. PROGRESS needs a place to store that choice. You can store values like menu choices in temporary fields called **variables**. The Selection variable is a temporary place in which to store integer data, such as the number of a menu selection.

2 The FORM statement describes the design of the menu. SKIP(2) tells PROGRESS to skip two lines; SKIP(1) skips one line. The CENTERED option tells PROGRESS to center the menu on your screen. The TITLE option gives the menu a title of "Maintenance and Reporting".

❸ The UPDATE statement prompts for a menu selection and stores the supplied value in the selection variable. For example, if you type "3," the selection variable has a value of 3.

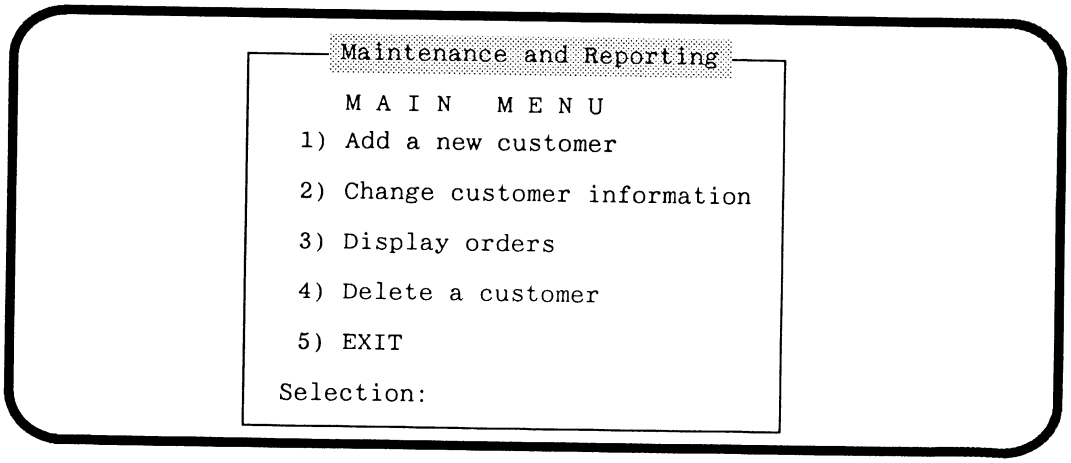
Normally, you have to press RETURN after entering data into a field. AUTO-RETURN tells PROGRESS to do a RETURN automatically. SIDE-LABELS puts the label of the selection variable to the left of (instead of above) the input area.

❹ The HIDE statement removes the menu from the screen after you've made a selection.

❺ The IF...THEN...ELSE statement tests the value of the selection variable. If the value is 1 (you selected menu option 1), the RUN statement runs the t-adcust.p procedure. If the value is 2, the RUN statement runs the t-chcust.p procedure. And so on.

If you select a value that is not between 1 and 5, the MESSAGE statement displays an error message telling you try again. Selection 5 lets you end the PROGRESS session.

When you run t-csmenu.p, your screen looks like this:



Try out the different menu options.

Here's a fancier version of the above menu program that illustrates the powerful CHOOSE statement.

t-csmen1.p

```
DEFINE VARIABLE selection  AS CHARACTER.
DEFINE VARIABLE menuchoice AS CHAR EXTENT 5
         FORMAT "x(30)" INITIAL
    ["1) Add a new customer",
     "2) Change customer information",
     "3) Display orders",
     "4) Delete a customer",
     "5) EXIT"].

REPEAT:
    FORM menuchoice WITH FRAME x CENTERED TITLE
         "Maintenance and Reporting"
         ATTR-SPACE 1 COLUMN NO-LABELS ROW 10.
    DISPLAY menuchoice WITH FRAME x.
    CHOOSE FIELD menuchoice AUTO-RETURN WITH FRAME x.
    selection = SUBSTR(FRAME-VALUE,1,1).
    HIDE FRAME x.
         IF selection EQ "1" THEN RUN t-adcust.p.
    ELSE IF selection EQ "2" THEN RUN t-chcust.p.
    ELSE IF selection EQ "3" THEN RUN t-itlist.p.
    ELSE IF selection EQ "4" THEN RUN t-delcus.p.
    ELSE IF selection EQ "5" THEN QUIT.
END.
```

❶ Once again, we use the `selection` variable, but we have added an array variable `menuchoice` to store the actual menu choice character strings that are displayed.

❷ The FORM statement describes the design of the menu. The FRAME phrase supplies an arbitrary frame name for the menu. The CENTERED and TITLE options are as in the first example. The ATTR-SPACE option lets the procedure work on both terminals that require an attribute space ("space-taking") and those that do not. The menu choices display in a single column without any field labels at row 10 on the screen.

❸ The DISPLAY statement actually displays the menu in the frame defined in the FORM statement.

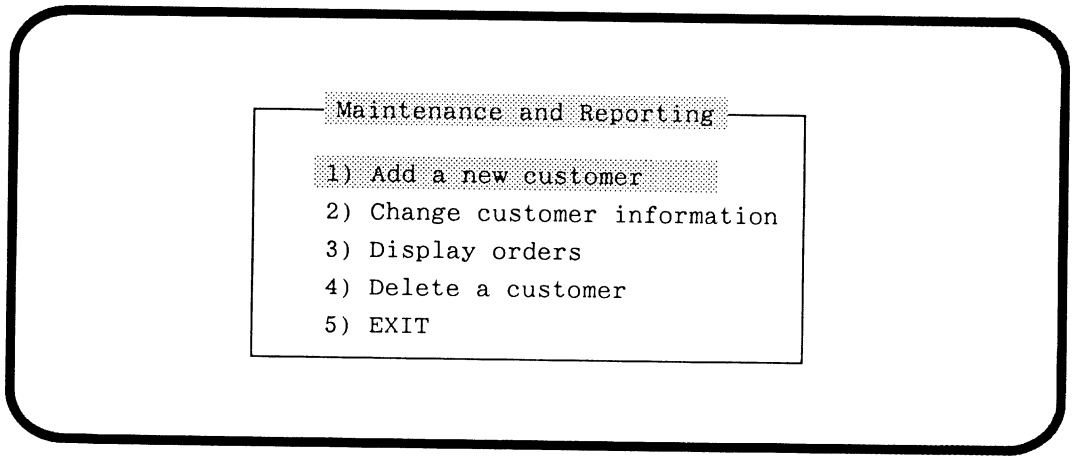
❹ The CHOOSE statement permits you to move a highlight bar through the displayed fields of the `menuchoice` variable. Furthermore, CHOOSE lets you select an option by either pressing RETURN with your selection highlighted, or by pressing the first character of an option (a digit in this case). AUTO-RETURN tells PROGRESS to do a RETURN automatically when you press a digit. SIDE-LABELS puts the label of the selection variable to the left of (instead of above) the input area.

❺ When you make a selection, the cursor is in one of the displayed fields; the FRAME-VALUE function returns the value stored in this field. This statement assigns to `selection` the first character (a digit) in the field the cursor was last in.

❻ The HIDE statement removes the menu from the screen after you have made a selection.

❼ The IF...THEN...ELSE statement is unchanged from the first example.

When you run `t-csmen1.p`, your screen looks like this:



See any difference between this screen and the one produced by `t-csmenu.p`? This time, CHOOSE highlights the first menu selection, and updates `selection` behind the scenes, rather than on the screen.

6.5 SUMMARY

This chapter described the different types of relationships that can exist between files:

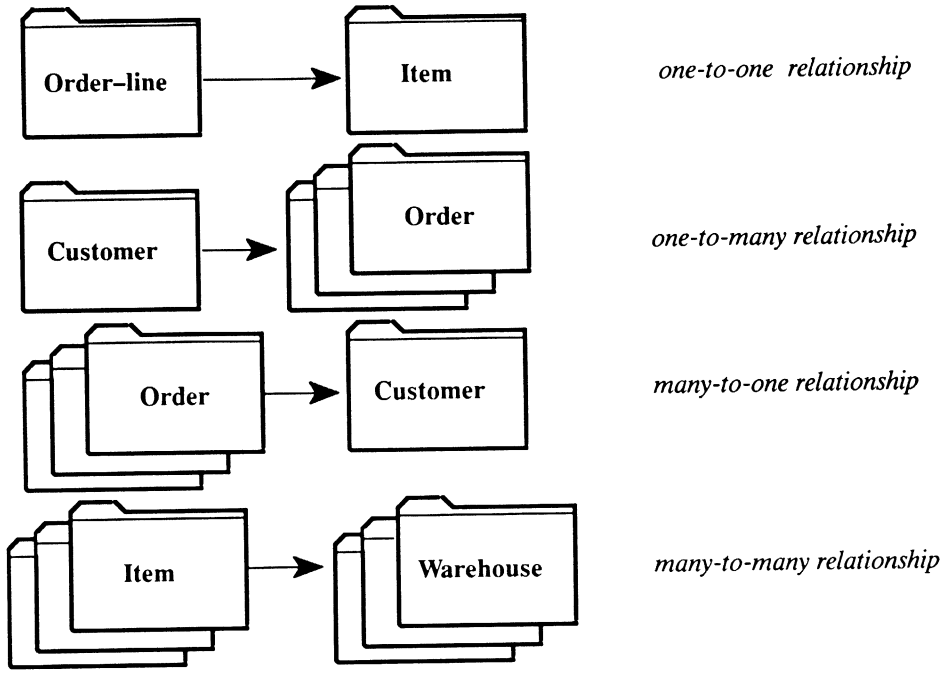


Figure 6–5: Relationships Between Files

This chapter also showed you how to:

- Write procedures using multiple related files.
- Write procedures that selectively read records from a database file.
- Print a report.
- Tie it all together with an application menu.

You have already built your own simple application, and in only four chapters. The next chapter takes a look at some of the issues you need to address as you develop more complex applications.

Chapter 7
Using PROGRESS to Develop Applications

In the space of a few short chapters, you have written and run several PROGRESS procedures and combined those procedures into your first application. Since building applications is what PROGRESS is all about, this chapter looks at some of the application design issues you should consider as you develop PROGRESS applications for your own use or for use by others.

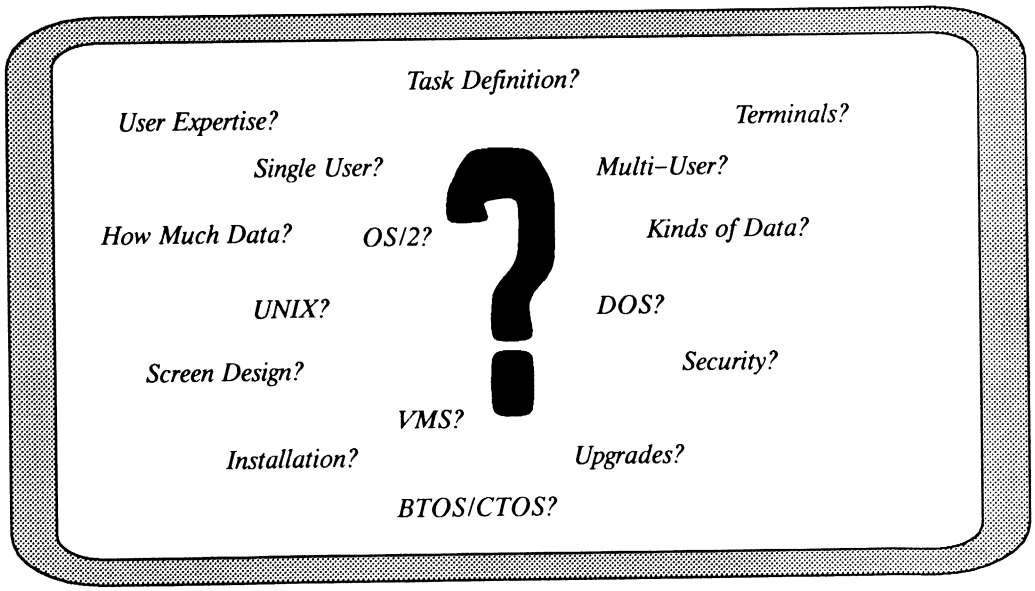


Figure 7–1: Some Factors to Consider When Developing an Application

In this chapter you will learn to:

- Design an application.
- Analyze your application needs.
- Design and build your application database.
- Write and test procedures.

7.1 APPLICATION DESIGN

Before you roll up your sleeves and start writing your application, ask yourself the following questions:

- What kind of application is needed?
- Who will be using the application? What is their level of technical expertise?
- How much and what kind of data do the users of the application require?
- How should the application task(s) be performed? What should the application procedures do?
- On which operating system will the application be run? DOS? OS/2? UNIX? XENIX? VMS? BTOS/CTOS? Will the application be transported between these systems?
- Will the application be used in a multi-user environment?
- What are the installation requirements for the application?
- What kind of terminals will the users be working with?
- What level of data security is needed?
- How will upgrades or modifications to the application be handled?

The remaining chapters of this Tutorial take a closer look at PROGRESS procedures, data movement and record reading, screen design, data entry control, report generation, transaction processing, and error handling. They are the groundwork for your PROGRESS applications. These topics are also covered in detail in the *PROGRESS Language Reference*. The *PROGRESS Programming Handbook* provides you with complete information on how PROGRESS handles issues like printer or file input and output, multi-user system design, and security. But before you can jump into some of these topics, it's important to build a solid foundation for your application. Begin by looking at the application development cycle.

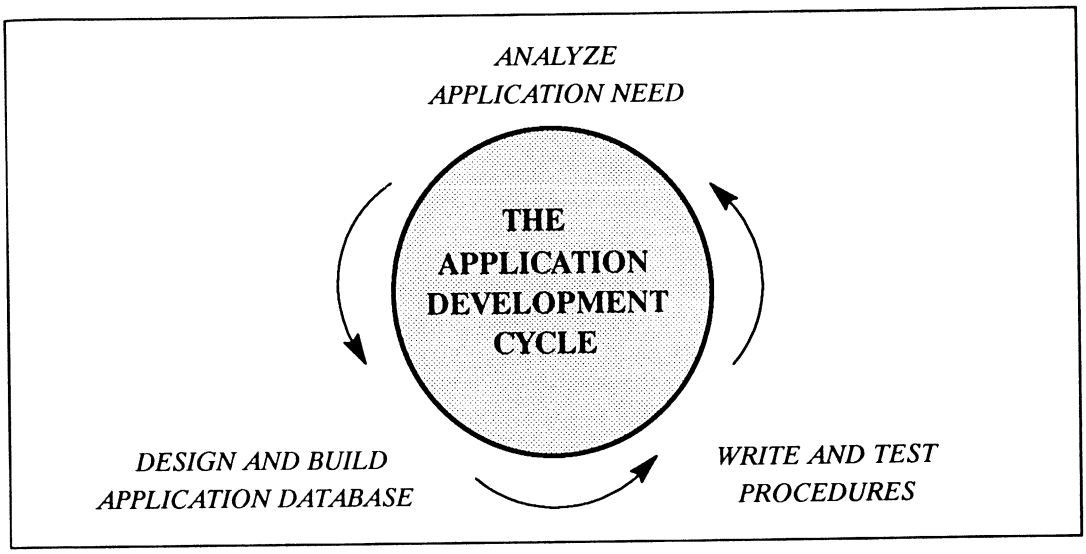


Figure 7–2: The Application Development Cycle

7.2 ANALYZE YOUR APPLICATION NEED

What kind of application is needed?
Who are the users?
What is their level of technical expertise?

Talk to your users. Find out what they're trying to do. Even if you're writing an application for your own use, these issues are worth thinking about.

Consider the small application you built in the last chapter. Its overall objective was to update and maintain accurate information about retail sporting goods customers and the merchandise they order. Tasks specific to that application include adding new customers to the database, changing customer information as needed, displaying orders placed by customers, and deleting customers as they become inactive.

7.3 DESIGN AND BUILD YOUR APPLICATION DATABASE

How much and what kind of data do the users require?

Once you've analyzed the goals of your application, your next step is to create a database for your application. What kind of information does the application require? Customer names? Addresses? Credit histories? Order information? How many files does the database require? How should those files be indexed?

You use the PROGRESS Data Dictionary to define the structure for the application data. You've already had a look at the Dictionary in Chapter 3. The *PROGRESS Programming Handbook* gives you more information on the Dictionary and on database design.

7.4 WRITE AND TEST PROCEDURES

What should your procedures do?

When your task analysis is complete and your database has a preliminary design, you can begin to break your overall application into smaller, more easily managed tasks.

As you write procedures to perform certain tasks, you may discover that you need to modify your Data Dictionary definitions. In fact, you'll probably continue to fine-tune both the definitions and your procedures as you build your application.

7.5 SUMMARY

A well planned application design strategy is critical to developing a functional and robust database application. Application development is an iterative process of analyzing application needs, defining data, and writing procedures. The remainder of this Tutorial shows you how to use PROGRESS and all of its components as your application development tools.

Chapter 8
A Closer Look at PROGRESS Procedures

The statements you have used in the procedures so far form the foundation of all the procedures you will write with PROGRESS. Before you begin writing more sophisticated procedures, it is important to understand what each of these statements is doing internally. Chapter 8 covers the following topics:

- How PROGRESS statements move data.
- Choosing between single and compound statements.
- Doing conditional processing.
- Grouping statements into blocks.

If you are still in the PROGRESS editor, use the QUIT statement to return to the operating system:

```
QUIT.

Enter PROGRESS procedure.  Press F1 to run.
```

Once you have exited from PROGRESS, make a fresh copy of the demo database. This is done with one of the commands in the following table:

Table 8–1: Command to Start PROGRESS

Operating System	To Copy Demo Database
UNIX, DOS, and OS/2	prodb mywork2 demo
VMS	PROGRESS/CREATE mywork2 demo
BTOS/CTOS	PROGRESS Create Database New Database Name mywork2 Copy From Database Name demo

Now you will be working with data that is unaffected by any changes you may have made so far.

8.1 HOW PROGRESS STATEMENTS MOVE DATA

PROGRESS uses two types of **data buffers** to control the movement of data:

- When you read a record from the database, PROGRESS places that record in a **record buffer**, which is a temporary storage area in data memory for a record, field, or variable. When you write a record to the database, PROGRESS gets that record from the **record buffer**.

- When you prompt for information or display information for the user, PROGRESS places that information in a **screen buffer**, which is a display area for a field, variable, or the result of a calculation.

Many PROGRESS statements use buffers to move data back and forth between the database, record buffers, and screen buffers. These statements are FIND, FOR EACH, DISPLAY, PROMPT-FOR, ASSIGN, SET, UPDATE, CREATE, INSERT, DELETE, and RELEASE.

You used many of these statements in earlier chapters; you'll learn about CREATE, ASSIGN, and SET in this chapter. We'll examine how each of these statements performs a particular task.

8.1.1 Displaying Data

From the PROGRESS editor, retrieve the t-disp.p procedure. First press GET (F5), then type **t-disp.p** and press RETURN.

t-disp.p

```
PROMPT-FOR customer.cust-num.
FIND customer USING cust-num.
DISPLAY name address city st zip.
```

This procedure prompts the user for a customer number, finds the customer record that has that number, and displays the record on the screen. Let's look at how these three statements move data between the database, the record buffer, and the screen buffer.

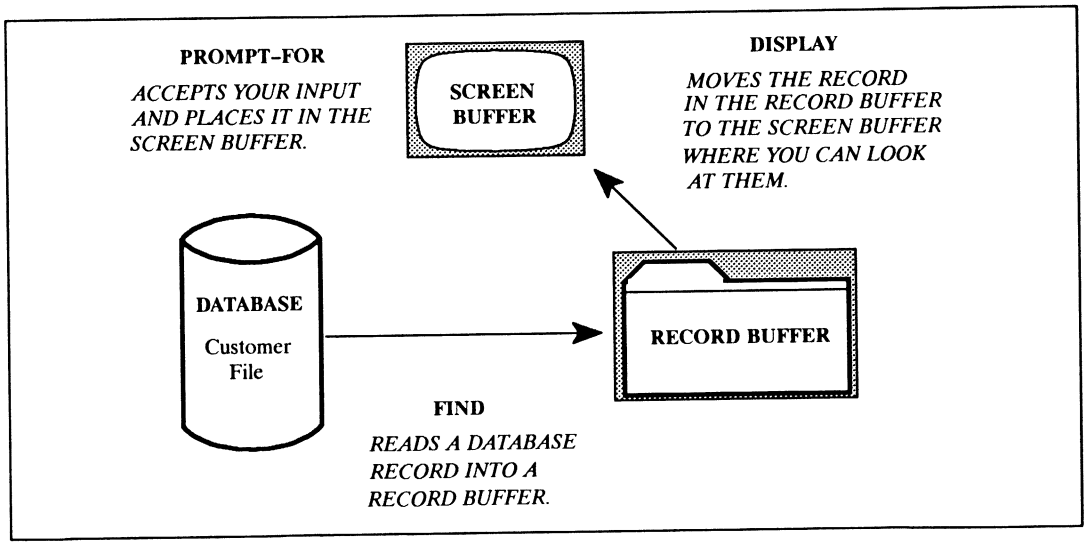


Figure 8–1: How Statements Move Data

These statements let you look at just one customer record having a specific customer number. Now let's look at what happens in these data buffers when you look at multiple records from multiple files. For example, suppose you want to:

1. Display the name, address, city, state, zip code, and maximum credit limit of every customer with a credit limit greater than $1500.

2. Display the order number, promised ship date and the payment terms of the orders placed by those customers.

Press CLEAR (F8) to clear the edit area, then press GET (F5), type **t-ordlst.p**, and press RETURN to retrieve the following procedure:

t-ordlst.p

```
FOR EACH customer WHERE max-credit > 1500:
  DISPLAY name address city st zip max-credit.
  FOR EACH order OF customer:
     DISPLAY order-num pdate terms.
  END.
END.
```

In the t-disp.p procedure, you saw that the FIND statement reads a single record from the database into the record buffer. FOR EACH statements read a copy of a single record from the database into the record buffer *each time* the FOR EACH block iterates.

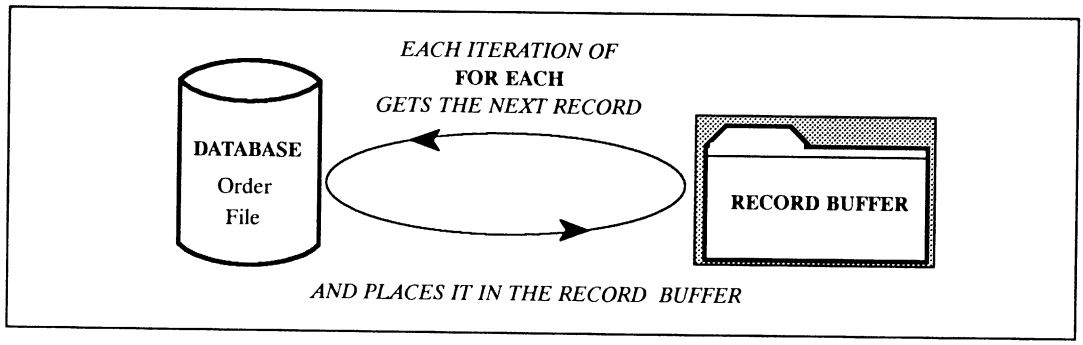


Figure 8–2: Iterations of the FOR EACH Statement

The first line of the t-ordlst.p procedure

```
FOR EACH customer WHERE max-credit > 1500:
```

tells PROGRESS on each iteration of the FOR EACH block to retrieve a customer record whose max-credit field holds a value greater than $1500. PROGRESS puts a copy of that record in a record buffer.

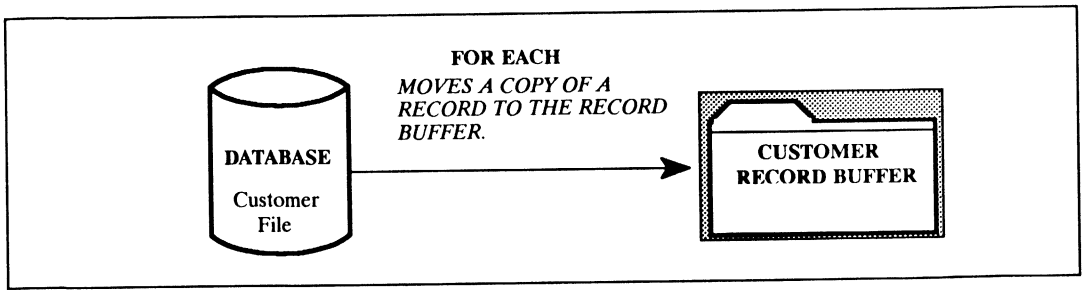


Figure 8–3: The FOR EACH Statement

The next line

```
DISPLAY name address city state zip max-credit.
```

moves the data from the record buffer to the screen buffer where you can look at it.

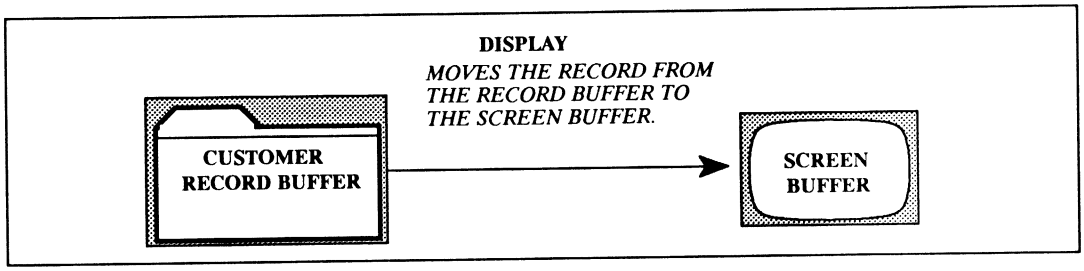


Figure 8–4: The DISPLAY Statement

In a similar way, the inner, or nested FOR EACH block retrieves a copy of that customer's first order record, and puts it into a record buffer. The DISPLAY statement that follows moves the data to the screen buffer, displaying the record on the terminal screen. The END statement for the inner FOR EACH block

```
FOR EACH order OF customer:
    DISPLAY order-num pdate terms.
END.
```

clears the order record buffer, leaving it ready for another order record when the block loops again.

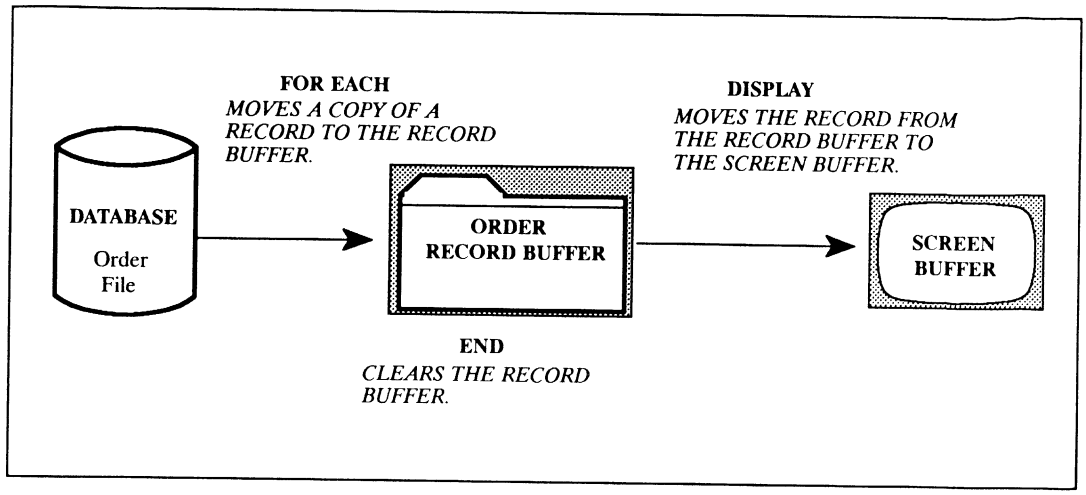


Figure 8–5: The FOR EACH/DISPLAY Loop

When all of the order records for a customer have been displayed, the END statement for the outer FOR EACH block clears the customer record buffer, leaving it ready for the next customer record.

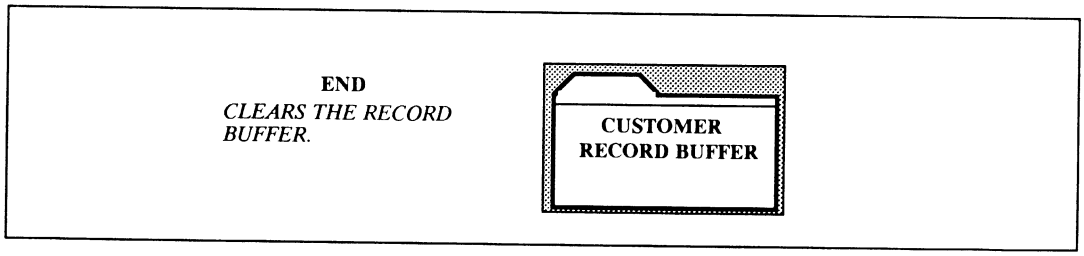


Figure 8–6: The END Statement

8.1.2 Adding and Changing Records

Let's assume you want to make changes to the customer file. You learned how to add records when you worked with the t-adcust.p procedure in Chapter 4. This procedure became part of your Reporting and Maintenance menu in Chapter 6.

Clear the edit area by pressing CLEAR (F8). Press GET (F5) and type **t-adcust.p** to retrieve this procedure, then press GO (F1) to run it. Try adding a customer record to your database.

t-adcust.p

```
REPEAT:
  INSERT customer WITH 2 COLUMNS.
END.
```

The INSERT statement creates a new, empty customer record. PROGRESS moves a copy of that empty record to a record buffer and then to the screen buffer, where you can enter and view the customer information.

When you press GO (F1) to complete entry of the data, PROGRESS moves the data from the screen buffer to the record buffer. You can also press END-ERROR (F4) to cancel your work and leave the procedure.

At the end of each REPEAT block iteration, the record is written to the database and the record buffer is cleared. The record buffer is then ready for a new customer record during the next block iteration.

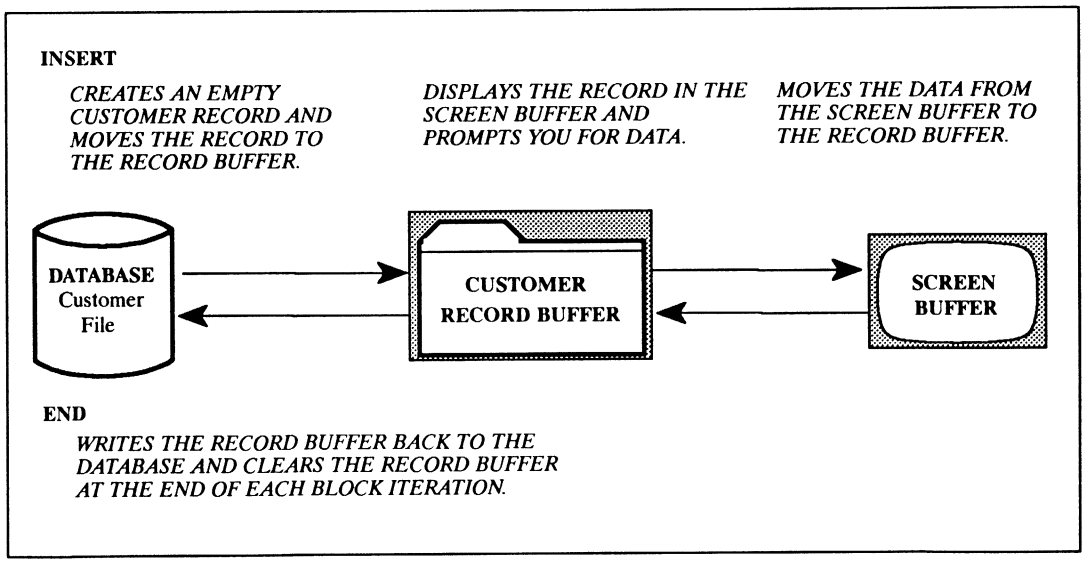


Figure 8–7: The INSERT Statement

If, at some point, you want to change information in that record, you can use the UPDATE statement. Retrieve the t-chinfo.p procedure, which uses PROMPT-FOR, FIND, and UPDATE to let you update customer records.

t-chinfo.p

```
PROMPT-FOR customer.cust-num.
FIND customer USING cust-num.
UPDATE customer WITH 2 COLUMNS.
```

The PROMPT-FOR statement asks you to enter a customer number and puts that number into the screen buffer where you can view it. PROGRESS uses that number to read into a record buffer a copy of the appropriate customer record. The UPDATE statement displays that record in the screen buffer and lets you make the desired changes. When you press GO (F1) to save your changes, PROGRESS moves the record back to the record buffer.

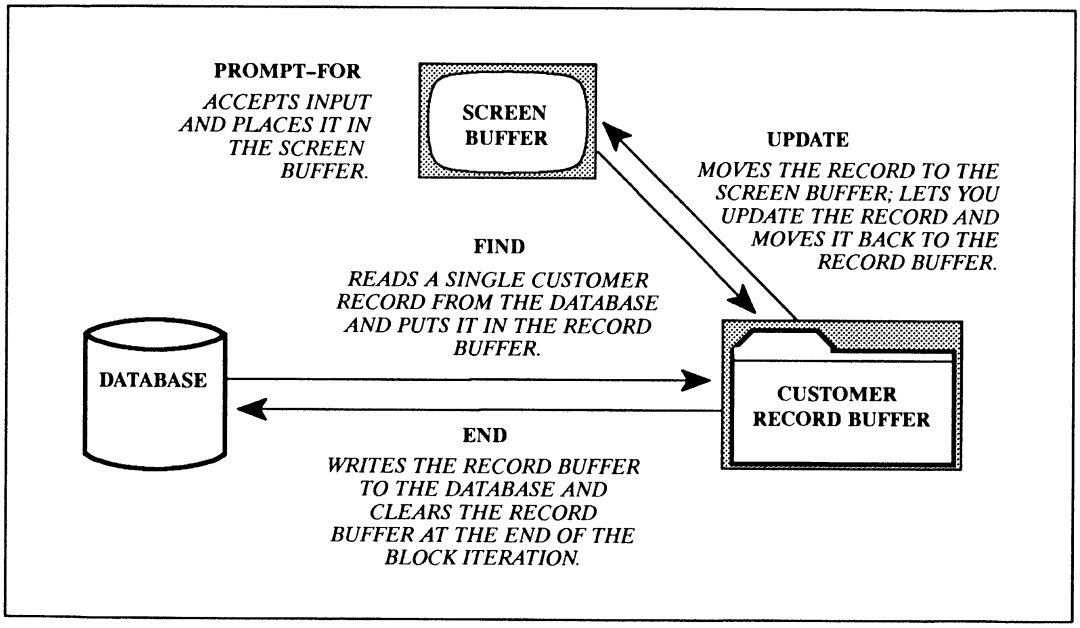


Figure 8–8: Updating a Record

Suppose you try to add a customer record with a customer number already in use, like 2. If you enter 2 in the cust-num field, and then enter information in the remaining fields, PROGRESS gives you an error message when you press GO (F1) to save the information:

```
** Customer already exists with Cust-num 2.
```

It would be nice to know at the outset that customer number **2** already exists, and it would also be useful to be able to update the record if you want to. Here's a procedure that lets you add *or* update a customer record:

t-mkcus.p

```
REPEAT:
❶ { PROMPT-FOR customer.cust-num.
    FIND customer USING cust-num NO-ERROR. } ❷
    IF NOT AVAILABLE customer THEN
    DO:
❸     MESSAGE "New customer record created.".
      CREATE customer.
      ASSIGN cust-num.
    END.
    ELSE DISPLAY customer. } ❹
❺ { SET customer WITH 2 COLUMNS.
❻ { END.
```

This procedure:

❶ Asks you for a customer number (PROMPT-FOR).

❷ Checks the database to see if a customer with that number exists (FIND). Since the goal is to create a new customer record, you don't want PROGRESS to give you an error message if a record with that customer number does not exist. The NO-ERROR option does this.

❸ If there is no record with that customer number, PROGRESS:

— Displays a message saying that a new record has been created (MESSAGE).

— Creates a new customer record (CREATE).

— Assigns to that new record the customer number you entered at the start of the procedure (ASSIGN).

❹ Displays the customer record if it already exists (DISPLAY).

❺ Lets you add information to the new record, or change the record if it's an existing record (SET).

❻ At the end of each block iteration, PROGRESS writes the contents of the record buffer to the database and clears the record buffer.

There are three new statements in this procedure:

- CREATE produces a new, empty customer record in the database and moves it to a record buffer.
- ASSIGN moves the data you entered (cust-num) from the screen buffer to the record buffer.
- The SET statement is a combination of the PROMPT-FOR and ASSIGN statements. It accepts your input (customer data) into the screen buffer, and then assigns, or moves that information to the record buffer.

Let's look at how this procedure moves data between the record buffer, screen buffer, and the database. There are two possibilities. First, a customer record may already exist in the database. If so, the procedure displays the record and uses the SET statement to let you change the information in that record. When you have finished with that record, the END statement writes the record to the database and clears the record buffer for the next iteration of the REPEAT block.

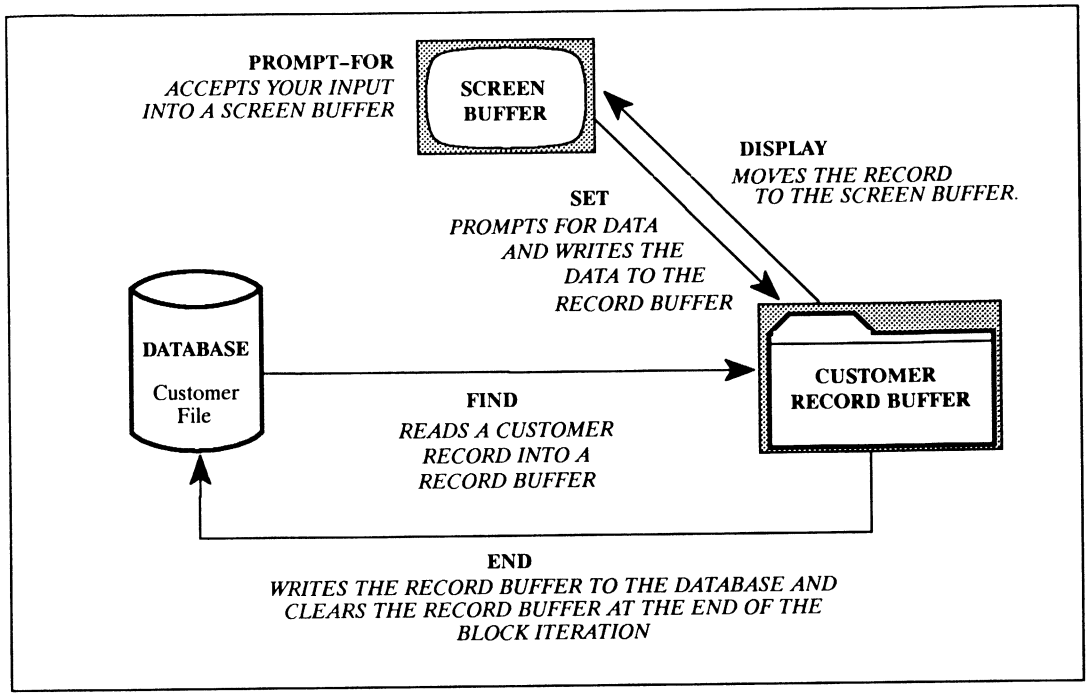


Figure 8–9: Changing an Existing Record

The second possibility is that no customer record exists with the customer number that you have entered. If this is the case, the procedure lets you add this new record to the database. It uses the CREATE statement to create an empty record and place that record in the record buffer. Then it uses ASSIGN to move the information you enter from the screen buffer to the record buffer. When you have finished entering the information, the END statement writes the record to the database and clears the record buffer for the next iteration of the REPEAT block.

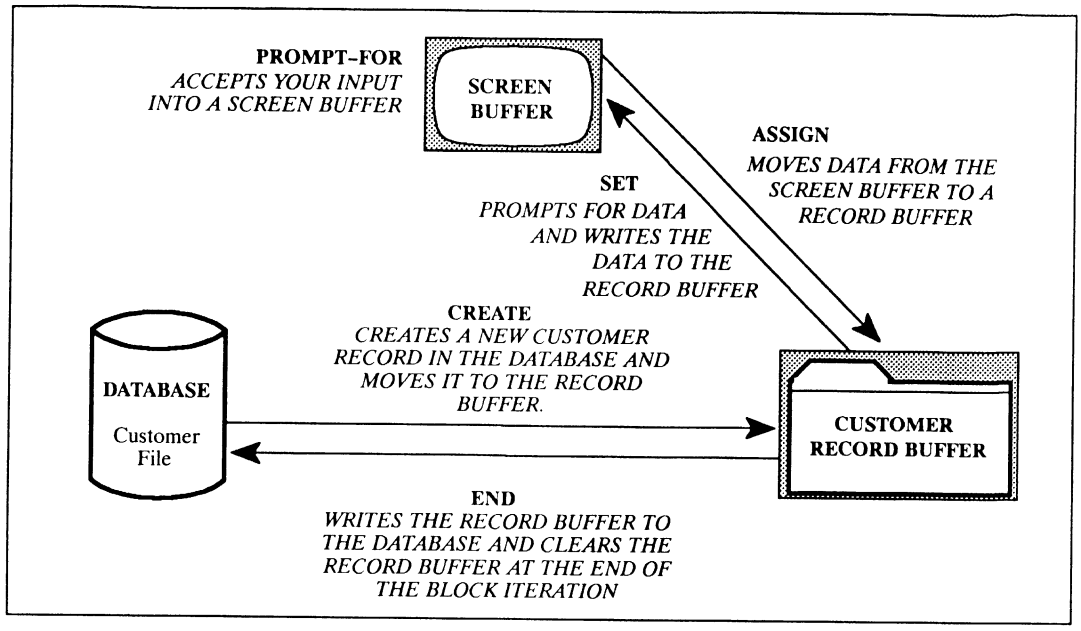


Figure 8–10: Entering a New Record

8.1.3 Removing Records

In Chapter 4, you worked with the t-delcst.p procedure to remove selected customer records from your database.

t-delcst.p

```
PROMPT-FOR customer.cust-num.
FIND customer USING cust-num.
DELETE customer.
```

Here's how PROGRESS uses buffers to delete a customer record:

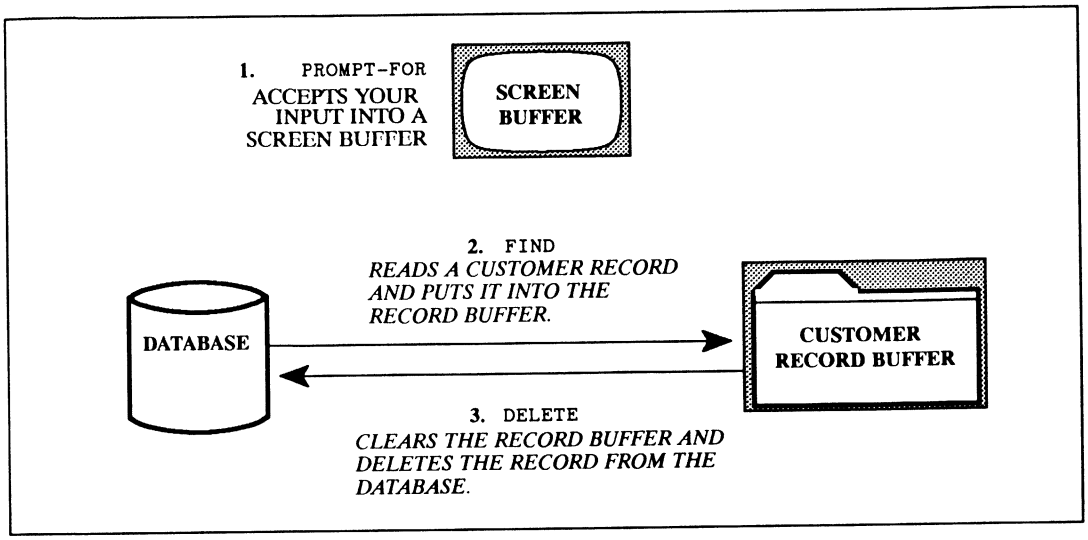


Figure 8–11: Deleting a Record

8.2 CHOOSING BETWEEN SINGLE AND COMPOUND STATEMENTS

You've probably noticed that some statements seem to do the job of 2 or even 3 other statements. Specifically:

- INSERT does the work of CREATE, DISPLAY, PROMPT-FOR and ASSIGN.
- UPDATE does the work of DISPLAY, PROMPT-FOR, and ASSIGN.
- SET does the work of PROMPT-FOR and ASSIGN.

The relationship between these statements can be summarized in the following diagram:

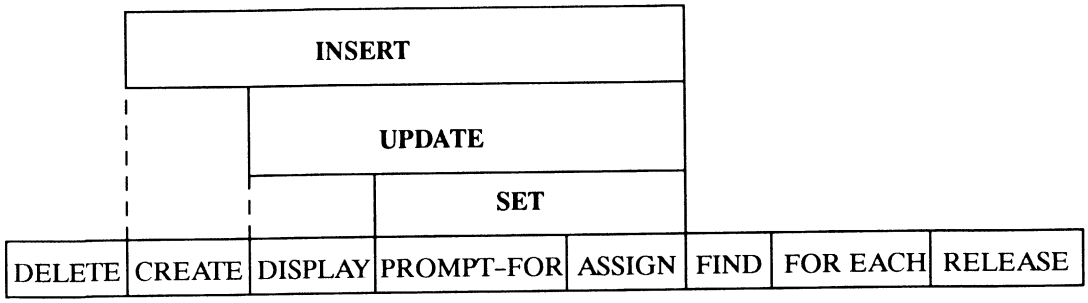
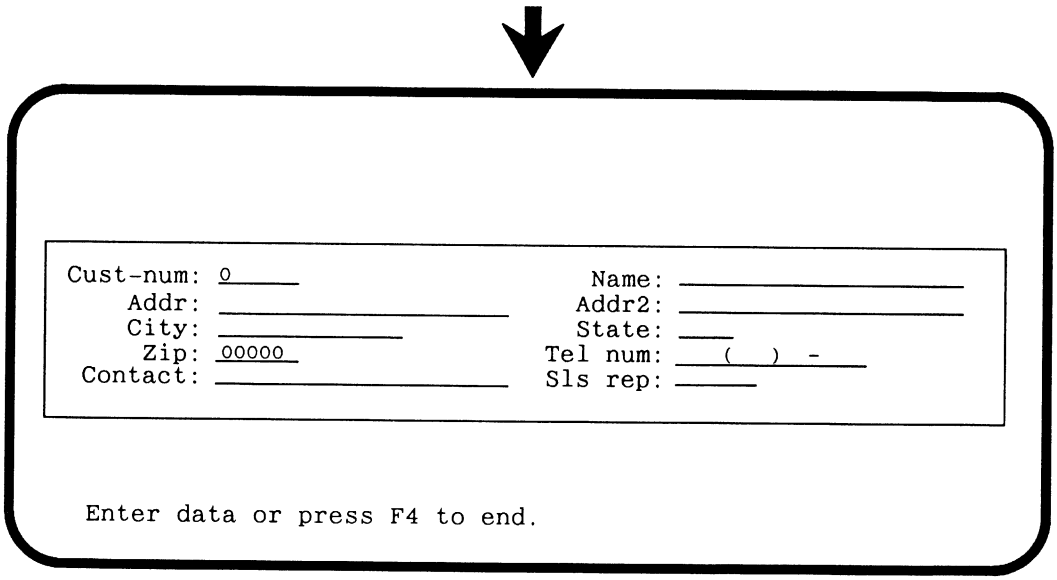


Figure 8–12: Relationships Between PROGRESS Statements

Certain application tasks suggest that you use the individual, component statements instead of the compound data-handling statements. For example, suppose you want a data entry clerk to insert new customer records into your database. However, you don't want that clerk to be able to input information into every field you've defined for that file. If you use the INSERT statement (which is a combination of CREATE, DISPLAY, and SET), the order entry person can enter data into all of the fields defined for that file.

t-adcust.p

```
REPEAT:
  INSERT customer WITH 2 COLUMNS.
END.
```



If you break down the INSERT statement into its component statements, you can control which fields the user can modify. In t-create.p, the only fields enabled for input are the name and contact fields:

t-create.p

```
REPEAT:
  CREATE customer.
  DISPLAY customer WITH 2 columns.
  SET name contact.
END.
```

```
Cust-num: 0                    Name: _____________________
    Addr:                     Addr2:
    City:                     State:
     Zip: 00000               Tel num: (    ) -
 Contact: _____________       Sls rep:

Enter data or press F4 to end.
```

8.3 DOING CONDITIONAL PROCESSING

Sometimes you want to do processing based on one or more **conditions**. A condition is any expression whose value is true or false. Conditional processing is often represented by a flow chart such as this:

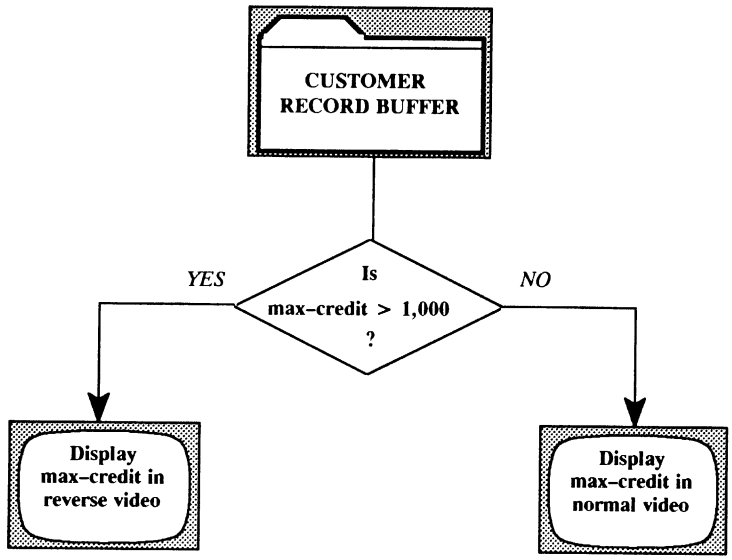
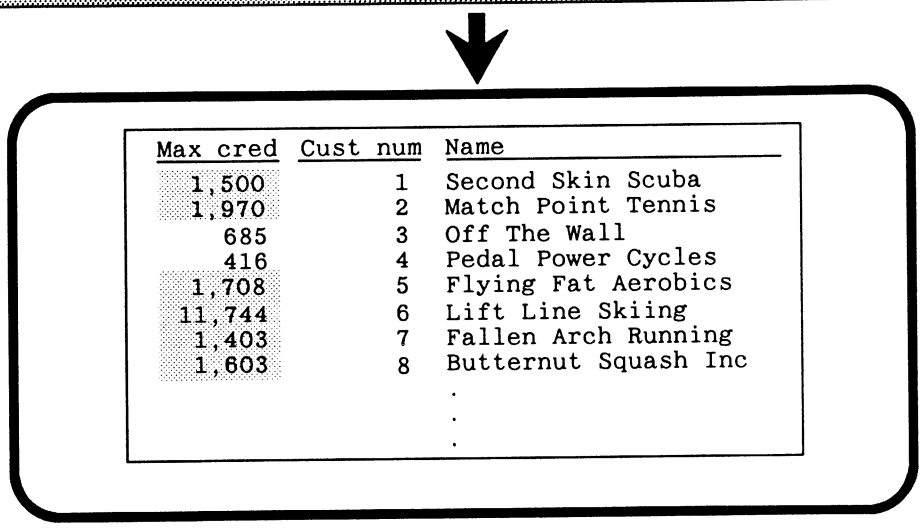


Figure 8–13: Conditional Processing

In this example, you want to display the number, name, and max-credit of every customer, but if the max-credit value is over $1,000, you want to display it in MESSAGES mode (usually in reverse video). If the max-credit value is 1,000 or less, you want to display it with normal video attributes. The t-ifels.p procedure accomplishes this task:

t-ifels.p

```
FOR EACH customer:
  IF max-credit > 1000
  THEN COLOR DISPLAY MESSAGES max-credit.
  DISPLAY max-credit cust-num name.
END.
```



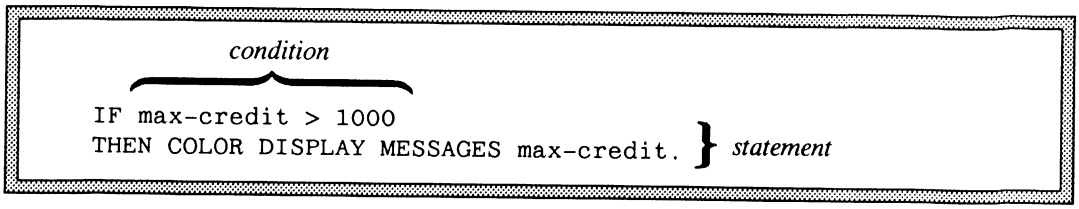
Max cred	Cust num	Name
1,500	1	Second Skin Scuba
1,970	2	Match Point Tennis
685	3	Off The Wall
416	4	Pedal Power Cycles
1,708	5	Flying Fat Aerobics
11,744	6	Lift Line Skiing
1,403	7	Fallen Arch Running
1,603	8	Butternut Squash Inc
		.
		.
		.

You use the IF...THEN...ELSE statement to perform conditional processing. There are two forms of this statement. The first, used in t-ifels.p, contains an IF condition followed by a THEN statement or block:

```
IF  condition THEN { statement }
                   { block     }
```

The statement or block is executed only if the IF condition is true. Otherwise, the statement or block is skipped and the procedure goes on to execute the statements following the THEN statement or block.

The t-ifels.p procedure uses the first of these forms:



The second form lets you also specify a statement or block of statements to execute if the first IF condition is false:

```
IF  condition THEN  { statement | block }
    ELSE { statement | block }
```

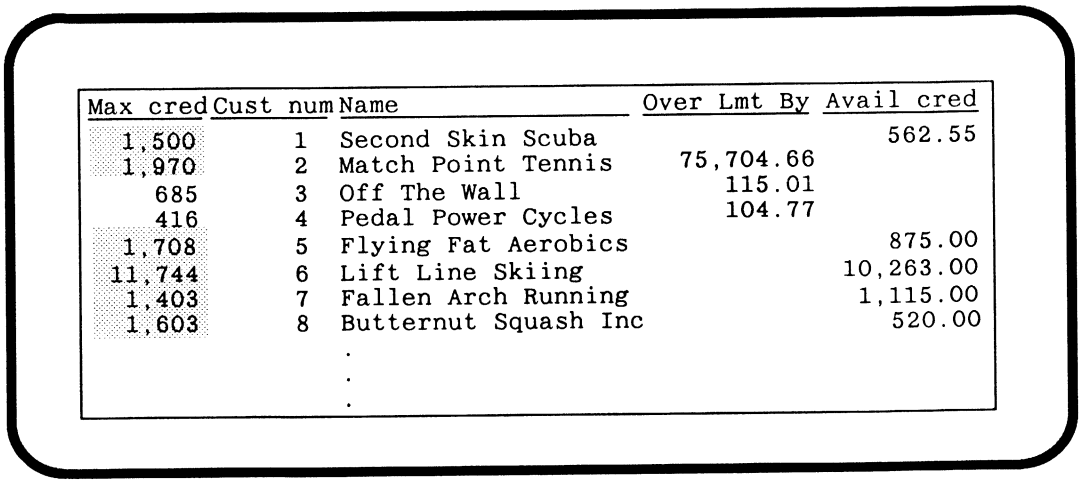
The t-ifels2.p procedure uses both forms of the IF...THEN...ELSE statement:

t-ifels2.p

```
FOR EACH customer:
  IF max-credit > 1000
  THEN COLOR DISPLAY MESSAGES max-credit.
  DISPLAY max-credit cust-num name.

  IF curr-bal > max-credit
  THEN DISPLAY curr-bal - max-credit
       LABEL "Over Lmt By".
  ELSE DISPLAY max-credit - curr-bal
       LABEL "Avail cred".
END.
```

First form: IF...THEN

Second form: IF...THEN...ELSE



Max cred	Cust num	Name	Over Lmt By	Avail cred
1,500	1	Second Skin Scuba		562.55
1,970	2	Match Point Tennis	75,704.66	
685	3	Off The Wall	115.01	
416	4	Pedal Power Cycles	104.77	
1,708	5	Flying Fat Aerobics		875.00
11,744	6	Lift Line Skiing		10,263.00
1,403	7	Fallen Arch Running		1,115.00
1,603	8	Butternut Squash Inc		520.00
	.			
	.			
	.			

Here, if a customer has exceeded their maximum credit limit, PROGRESS displays the amount by which they have exceeded that limit. Otherwise, the remaining credit is displayed.

8.4 GROUPING STATEMENTS INTO BLOCKS

You've learned that the data handling statements used in different combinations and in different situations handle the movement of information in your database. The FIND statement, for example, retrieves a single, specific record from the database. FOR EACH blocks, however, automatically retrieve a set of records.

Blocks also let you:

- Group statements for flow of control. By grouping statements into blocks, you can tell PROGRESS to process a group of statements under certain circumstances but not under other circumstances.
- Identify a context for a block. For example, you can use blocks to identify a specific file to be used by an entire group of statements.
- Identify points in a procedure where you want PROGRESS to perform a processing service. The block structure of the PROGRESS language offers a wide range of processing services:
  - Record reading
  - Iteration, or looping
  - Clearing of record buffers

- Default screen designs
- Transaction recovery
- Error processing

8.4.1 Grouping Statements to Control Program Flow

Block Benefit #1: You can use blocks to group statements for flow of control. You may want to tell PROGRESS to process a group of statements under certain circumstances but not under others or tell PROGRESS what to do when exiting from the block.

The following procedure is a modified version of the t-mkcus.p procedure from the previous section. Like all PROGRESS procedures, the entire procedure is a block. In addition, this procedure also contains a DO block.

t-mkcus2.p

```
PROMPT-FOR customer.cust-num.
FIND customer USING cust-num NO-ERROR.
IF NOT AVAILABLE customer THEN
DO:
  MESSAGE "New customer record created.".
  CREATE customer.
  ASSIGN cust-num.
END.
ELSE UPDATE customer WITH 2 COLUMNS.
```

The t-mkcus2.p procedure begins by prompting you for a customer number. It uses that number to retrieve a record from the customer file. In the case where a record is not available, the procedure must:

- Tell you that a new customer record is being created.
- Create the new record.
- Assign to that record the customer number you entered at the start of the procedure.

A DO block, whose only job is to group statements, handles this task.

```
DO:
  MESSAGE "New customer record created.".
  CREATE customer
  ASSIGN customer
END.
```

Within a block , you can also use the NEXT and LEAVE statements to control the program flow when exiting from a block. For example:

t-flow.p

```
DEFINE VARIABLE selection AS CHARACTER FORMAT "x".

FOR EACH customer:
  DISPLAY cust-num name WITH FRAME a.
  DISPLAY "O     - Order Processing" SKIP
          "N     - Next Customer   " SKIP
          "L     - Leave this task"  SKIP(1)
          WITH FRAME b COLUMN 45 NO-HIDE.

  SET selection LABEL "Selection" AUTO-RETURN
        WITH FRAME b SIDE-LABELS.
  IF selection = "L" THEN LEAVE.
  ELSE IF selection = "N" THEN NEXT.
  ELSE IF selection = "O" THEN
  DO:
    FOR EACH order OF customer WITH TITLE "Orders":
      DISPLAY order-num.
      UPDATE odate pdate sdate.
    END.
  END.
END.
```

```
Cust num  Name
      1   Second Skin Scuba

O   - Order Processing
P   - Next Customer
L   - Leave this Task

Selection:
```

Here, if you enter an "N," the NEXT statement starts the next iteration of the FOR EACH block. If you enter an "L," the LEAVE statement leaves the FOR EACH block.

8.4.2 Identifying a Context for a Block

Block Benefit #2: You can use blocks to identify a specific file to be used by an entire group of statements, thereby specifying a context for the block.

Suppose you want to look at information about a particular customer:

t-cntxt.p

```
REPEAT:
  PROMPT-FOR cust-num.
  FIND customer USING cust-num.
  DISPLAY name address city st zip.
END.
```

When you press GO (F1) to run this procedure, this message appears on your screen:

```
** cust-num is ambiguous with Customer.Cust-num and Order.Cust-num.
```

PROGRESS isn't sure whether you want to be prompted for the cust-num value in the customer file or the order file. You must tell PROGRESS which cust-num field you are referring to. To do this, you can modify the t-cntxt.p procedure to read:

t-cntxt2.p

```
REPEAT FOR customer:
  PROMPT-FOR cust-num.
  FIND customer USING cust-num.
  DISPLAY name address city st zip.
END.
```

By specifying that the REPEAT block should perform its activities FOR the customer file, PROGRESS knows that the customer file is the appropriate file in which to locate cust-num.

If you do not want to make customer the default file for the entire block, you can use the full name of the cust-num field:

```
customer.cust-num
```

In fact, if you were using both the customer and order files in the block, you would have to use the full name.

8.4.3 Using Blocks to Perform PROGRESS Processing Services

Block Benefit #3: You can use blocks to identify points in a procedure where you want PROGRESS to perform some processing service.

Different kinds of blocks provide a particular processing service or set of services. Automatic record reading and looping are the processing services associated with FOR EACH blocks. FOR EACH blocks automatically read a record each time the block loops; you need not use a FIND statement. The other processing service associated with a FOR EACH block is iteration. The FOR EACH statement in this procedure reads each record in the file.

Take another look at the t-cntxt2.p procedure we worked with earlier, which contains a REPEAT block:

t-cntxt2.p

```
REPEAT FOR customer:
  PROMPT-FOR cust-num.
  FIND customer USING cust-num.
  DISPLAY name address city st zip.
END.
```

A REPEAT block, like a FOR EACH block, automatically iterates. In the t-cntxt.p procedure, you are prompted for a customer number each time the block loops, unless you press END-ERROR (F4) to stop the iteration. However, a REPEAT block does not automatically read records; you must use a FIND statement within the REPEAT block if you want to retrieve a record.

In later chapters, we'll take an in-depth look at other processing services performed automatically by blocks, such as writing records back to the database, designing screens, recovering transactions, and handling errors.

8.5 SUMMARY

Knowing how PROGRESS works with buffers to move data adds a new dimension to your knowledge of PROGRESS. The following diagram summarizes how each of the data handling statements moves data. In the diagram, the dot ● indicates a record's starting point and the arrows ➔ show the direction of movement as the record is handled by a particular statement.

The data handling statements we've examined are the foundation statements for all procedures. These statements perform the fundamental tasks required by all business applications — adding, changing, displaying, and deleting records.

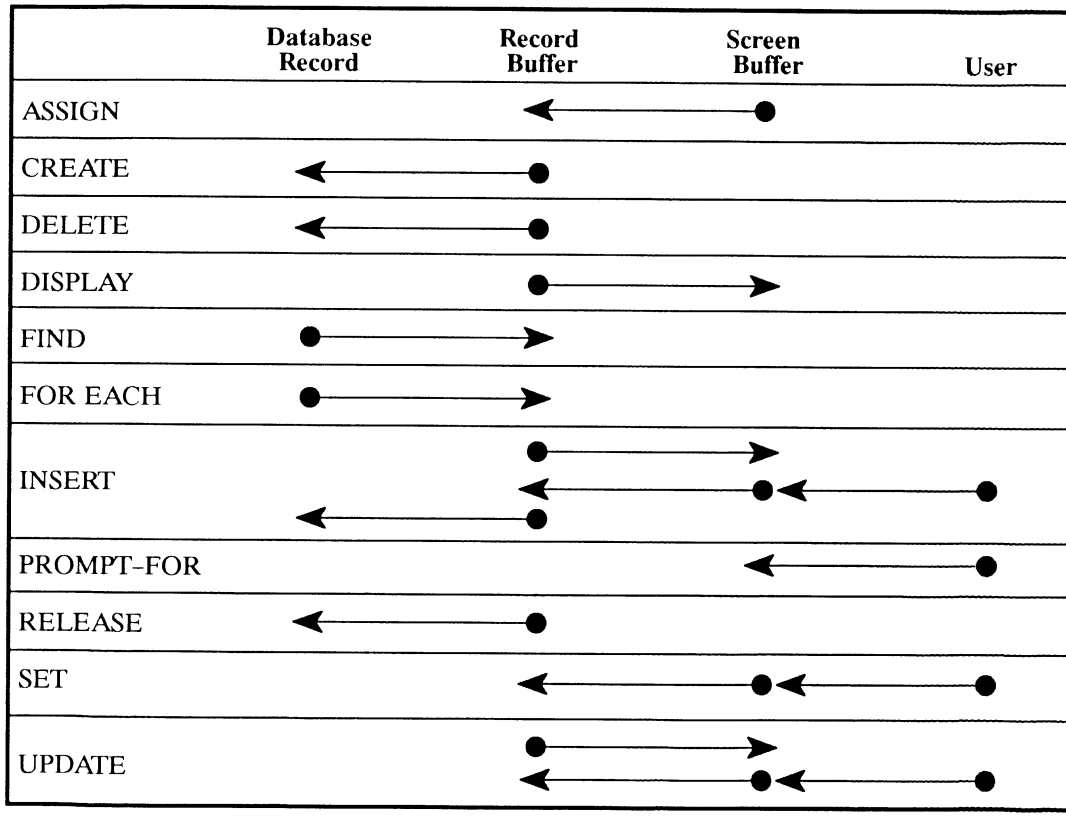


Figure 8–14: How Data Handling Statements Move Data

PROGRESS lets you fine tune the way these jobs are done by using either "combination" statements like INSERT, or the components statements such as CREATE, DISPLAY, and SET. The statement you use depends upon the requirements of your application. As you continue through this Tutorial, we'll show you many additional ways to use these statements.

This chapter also showed you how to perform conditional processing with IF...THEN...ELSE and how to use blocks to group statements to control the program flow, to define a context for a block, and to perform record processing services.

Chapter 9
Record Reading

You saw in Chapter 8 how FIND and FOR EACH move records between the database and the record buffer. FIND reads a copy of a single, unique record from the customer database file into the record buffer.

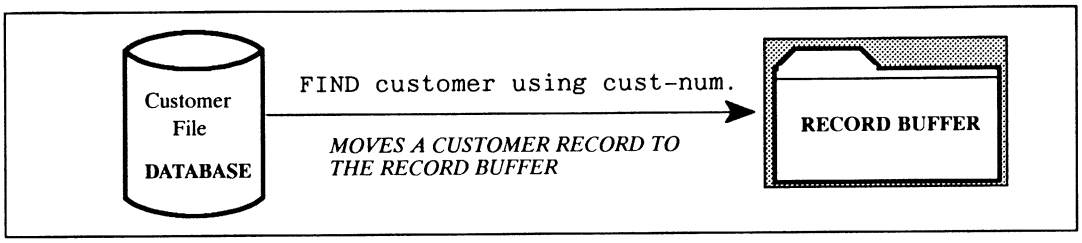


Figure 9–1: The FIND Statement

At the end of the block to which the record is scoped, PROGRESS writes the record back to the customer database file if the record was updated and clears the record buffer.

FOR EACH automatically reads a record each time the block iterates. That is, each time a FOR EACH block loops, it reads a single record from a database file into the record buffer. At the end of each block iteration, PROGRESS writes the record back to the customer database file if the record was updated and clears the record buffer to make way for the next record FOR EACH reads from the database.

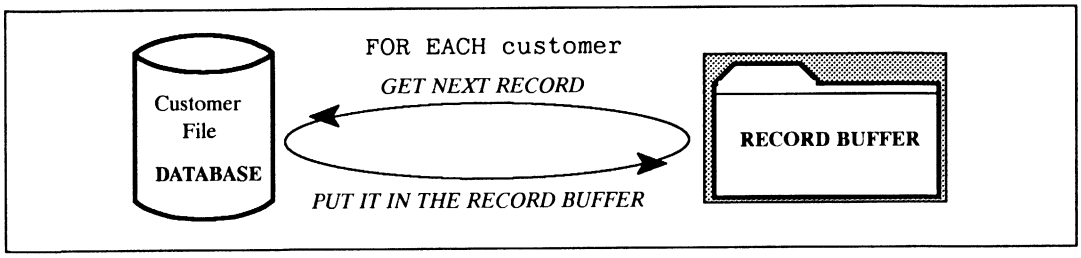


Figure 9–2: Iterations of the FOR EACH Statement

The FIND statement (used alone or within a REPEAT block) and the FOR EACH block are your principal record reading tools. By using options on these statements, you can be very specific about the records you want to read.

In this chapter, you will learn how to more closely control what records are read from the database. You will also start to learn how to control how the user interacts with your application by defining actions to take when the user presses specific keys. You will learn about the "scope" of records and how to read multiple records from the same file at the same time.

If you left PROGRESS after reading the last chapter, be sure you are in your working directory and type

```
pro mywork2
```

9.1 QUALIFYING RECORD SELECTION

You can control which records PROGRESS reads, and even the order in which they are read, by:

- Supplying a set of conditions for record reading.
- Reading records that match a value the user supplies.
- Reading the first, last, next, or previous record in a file.
- Comparing field values with a character expression.
- Identifying a specific index you want PROGRESS to use to read a record or group of records.
- Sorting records in a specific order.

9.1.1 Reading Records Conditionally: WHERE

You will often want to read only the records that satisfy a certain condition or set of conditions. You use the WHERE option to describe these conditions. For example, the following FOR EACH statement reads and displays only those customer records whose max-credit value is greater than 1500.

t-where1.p

```
FOR EACH customer WHERE max-credit > 1500:
  DISPLAY name.
END.
```

```
Name

Match Point Tennis
Flying Fat Aerobics
Lift Line Skiing
Butternut Squash Inc
Spikes Volleyball
    .
    .
    .

Procedure complete.  Press space bar to continue.
```

You know you can also use the FIND statement to read records, so let's modify this procedure to use the FIND statement instead of FOR EACH.

t-where3.p

```
FIND customer WHERE max-credit > 1500.
DISPLAY name.
```

When you press GO (F1) to run this procedure, you get a message from the PROGRESS compiler:

```
** Insufficient index information for a unique find.
```

You may be thinking, "I thought I could use *either* FIND or FOR EACH to get records out of my database files." You're right. You can. However, FIND works a little differently than FOR EACH.

The purpose of the FIND statement is to retrieve a *single, unique* record. In order to locate this record, the FIND statement must refer to an indexed field. Therefore, for FIND to succeed, it must be able to use an index and the data in the indexed field must be unique. The FIND statement in our example failed because there is no unique index for the max-credit field.

Since FOR EACH blocks loop, they need to be able to read multiple records. Therefore, if you think multiple records will satisfy the WHERE condition, use a FOR EACH block to read those records. If you know that just one record will satisfy the condition and you have a unique index to support it, use the FIND statement.

Two important points to remember are:

- You are always safe using a FOR EACH block. FOR EACH works whether one record or many records satisfy the WHERE condition. FIND works only when exactly one record satisfies the condition, and you have a unique index to support it.
- If you want to be sure that just one record satisfies the WHERE condition, make sure that the condition uses a unique index (such as the cust-num field in your customer file).

This example uses a unique index with the FIND statement:

t-unique.p

```
PROMPT-FOR customer.cust-num.
FIND customer WHERE
     customer.cust-num EQ INPUT cust-num.
DISPLAY customer.
```

In this example, PROMPT-FOR asks you for a value and stores that value in the screen buffer. The FIND statement tells PROGRESS to read from the database the customer record whose cust-num equals the cust-num stored in the screen buffer. (The INPUT function refers to that screen buffer). This is a common application situation: you enter a value and want to retrieve the record that matches that value. Because this is such a common situation, PROGRESS gives you a shorthand way to do it.

9.1.2 Reading Records that Match a Value the User Supplies: USING

The USING option tells PROGRESS to find a record that has a field that matches the value the user supplied. For example:

t-using.p

```
PROMPT-FOR customer.cust-num.
FIND customer USING cust-num.
DISPLAY customer WITH 2 COLUMNS.
```

The statement

```
FIND customer USING cust-num
```

is exactly the same as

```
FIND customer WHERE cust-num = INPUT cust-num
```

In addition to the USING option, several other options are available with FIND that let you fine tune the record selection process.

9.1.3 Finding the Next, Previous, First, or Last Record in a File

When you're looking at a particular record in a file, you may decide that you want to look at the record that follows it or precedes it or you may want to look at the first or last record. PROGRESS will read those records if you modify the FIND statement with the NEXT, PREV, FIRST, and LAST options, as in the t-finds.p procedure.

t-finds.p

```
DISPLAY
"You may update each customer. After making your changes:"
         SKIP "Press one of:" SKIP(1)
        " F1  -  Make the changes permanent"   SKIP
        " F4  -  Undo changes and exit"        SKIP
        " F9  -  Undo changes and try again"   SKIP
        " F10 -  Find next customer"           SKIP
        " F11 -  Find previous customer"       SKIP
        " F12 -  Find first customer"          SKIP
        " F13 -  Find last customer"           SKIP
            WITH CENTERED FRAME instr.
PROMPT-FOR customer.cust-num.
FIND customer USING cust-num.
REPEAT:
  UPDATE cust-num name address city st
  GO-ON(F9 F10 F11 F12 F13)
    WITH 1 DOWN.
  IF LASTKEY = KEYCODE("F9")  THEN UNDO, RETRY.
  ELSE
  IF LASTKEY = KEYCODE("F10") THEN FIND NEXT customer.
  ELSE
  IF LASTKEY = KEYCODE("F11") THEN FIND PREV customer.
  ELSE
  IF LASTKEY = KEYCODE("F12") THEN FIND FIRST customer.
  ELSE
  IF LASTKEY = KEYCODE("F13") THEN FIND LAST customer.
END.
```

❶ The first part of this procedure sets up the screen display for a menu for updating customer records. The menu lists a number of function keys and the action each performs. After updating a customer record, the user can save the changes made to the record, undo the changes and exit the menu, or undo the changes and update the record again. The user can also retrieve the next, previous, first, or last customer record in the file and update any one of those records.

❷ PROGRESS prompts the user for a customer number, using the primary index (cust-num) to find the appropriate customer record.

FIND reads a copy of the record into the customer record buffer.

The UPDATE statement moves the copy of the record from the record buffer to the screen buffer so the user can make changes. After updating the record, the user can press GO (F1) to save the changes, or press

END (F4) to undo the changes and exit the menu. GO (F1) and END (F4) perform their usual PROGRESS functions in this procedure.

❸ The GO-ON option tells PROGRESS to go on to the rest of the procedure based on one or more conditions. Here, in addition to the F1 key which is the default GO key, PROGRESS goes on to process the remainder of the procedure if the user presses F9, F10, F11, F12, or F13.

The LASTKEY function always returns the internal key code of the last key pressed. The KEYCODE function returns the internal code of a specified key. For example, if the user presses F9, then the value returned by LASTKEY equals the value returned by KEYCODE ("F9"). In this procedure, the following actions are defined for function keys 9 through 13:

F9	UNDO, RETRY	PROGRESS undoes all changes the user has made to the record, letting the user update the record again.
F10	FIND NEXT	PROGRESS uses the primary index, cust-num, to retrieve the next customer record from the file and read it into the record buffer.
F11	FIND PREV	PROGRESS finds the previous record in the file.
F12	FIND FIRST	PROGRESS reads customer #1 into the record buffer.
F13	FIND LAST	PROGRESS finds the last customer record in the file.

Each of these keys has been redefined to perform a task other than its default PROGRESS function.

You saw several new PROGRESS capabilities in this procedure. The important point to remember is that you can modify the FIND statement with words like FIRST, LAST, NEXT, and PREV to let you navigate around a file to retrieve specific records.

9.1.4 Comparing Field Values with a Character Expression: BEGINS and MATCHES

Most character comparisons are case-insensitive in PROGRESS. By default, all characters are converted to upper case prior to comparisons. However, it is possible to define fields and variables as case-sensitive (though it is not advised, unless strict ANSI SQL adherence is required). If either expression is a field or variable defined to be case-sensitive, the comparison is case-sensitive and "Smith" does not equal "smith."

Note that case-sensitivity for columns and variables applies only to the stored data values; the actual file names, column names, and variable names are never case-sensitive, nor are comparisons involving these names. For example, the following comparison is never case-sensitive.

```
. . . WHERE field-name BEGINS 'ny'
```

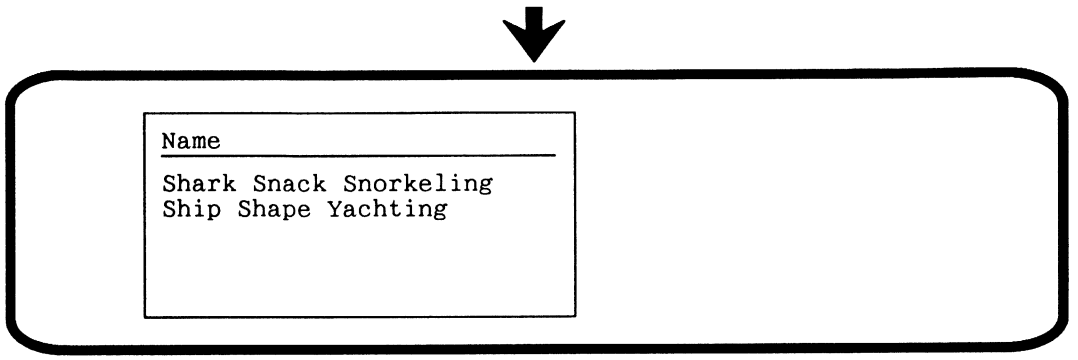
Suppose you want a list of customers whose names begin with the letters **"Sh"**. You can use the BEGINS function to accomplish this. Because it is possible for more than one customer name to begin with **Sh**, you write the procedure with a FOR EACH statement, not a FIND statement.

Because **Sh** is a character string, you must enclose it in quotation marks. PROGRESS will find the records beginning with **SH** (or **Sh**, or **sH**, or **sh**) whether you specify **SH**, **Sh**, **sH** or **sh**. PROGRESS searches through the customer file using the name field to find the appropriate customer records. Each customer record whose name field value begins with **Sh** is read into a record buffer on each iteration of the FOR EACH block and displayed in the screen buffer.

NOTE: If a field is defined as case-sensitive (see Chapter 5), BEGINS and MATCHES comparisons are case-sensitive. That is, **SH** is different from **Sh** and **sh**. Case-sensitive fields are generally not recommended. See also the ANSI SQL (-Q) startup option in Chapter 3 of *System Administration II: General* and the BEGINS and MATCHES functions in the *PROGRESS Language Reference.*

t-begins.p

```
FOR EACH customer WHERE name BEGINS "Sh".
  DISPLAY name.
END.
```



Another way to retrieve records by comparing field values is to use the MATCHES function. Notice that an asterisk (*) follows **Sh**. The asterisk tells PROGRESS to do a wildcard search to retrieve any and all records that begin with **Sh**. Here, MATCHES works the same way as BEGINS because the character expression following MATCHES includes an asterisk.

/* t-match1.p */

```
FOR EACH customer WHERE name MATCHES "Sh*":
  DISPLAY name.
END.
```

Now try writing the same procedure without adding an asterisk after **Sh**:

t-match2.p

```
FOR EACH customer WHERE name MATCHES "Sh":
  DISPLAY name.
END.
```

When you run this procedure, nothing happens. MATCHES searches for a field that *exactly matches* the specified pattern and PROGRESS can't find any records where the name field contains only the characters **Sh**. However, the wildcard character * can make MATCHES more flexible so that it works like BEGINS.

The difference between BEGINS and MATCHES is that BEGINS searches for a field that *begins* with the characters you specify. If you use an indexed field, BEGINS uses that index to find the appropriate records and does not have to search through every record in the file. In contrast, MATCHES retrieves and scans *all* records in a file in an effort to find records that match the character expression you specify.

Though BEGINS performs a more efficient search than MATCHES if you are using an indexed field, MATCHES has more flexibility than BEGINS because you can optionally use a wildcard character to control the way MATCHES performs its search. See the *PROGRESS Language Reference* manual for more information about BEGINS and MATCHES.

9.1.5 Identifying an Index for Record Selection: USE-INDEX

You learned earlier that when PROGRESS retrieves a record from a database file with either a FIND or FOR EACH statement, it always attempts to use an index to find the record. USE-INDEX lets you name a specific index to both find and sort the records you want read from the database.

t-useind.p

```
FOR EACH customer USE-INDEX zip:
  DISPLAY name address city st zip max-credit.
END.
```

This procedure uses the zip index to read customer records on each iteration of the FOR EACH block, and sorts them for display in zip code order.

9.1.6 Sorting Records in a Specified Order: BY

You've seen that PROGRESS uses an indexed field to read and sort records. If you want, you can then sort those records using any field you specify with the BY option.

t-sortby.p

```
FOR EACH customer WHERE max-credit > 1500
  BY name DESCENDING:
  DISPLAY name city zip max-credit.
END.
```

In this procedure, PROGRESS uses the primary index, cust-num, to find the customer records satisfying the WHERE condition. It sorts the customer records BY name in DESCENDING alphabetical order.

9.1.7 Reading Related Records: OF

Suppose you read a customer record and then want to find all of the orders, if any, placed by that customer. The OF option lets you read records from related files. For example, both the customer file and the order file have a cust-num field, which is an index for both files. Therefore, you can use the OF option to relate, or "join," the customer file to the order file. PROGRESS uses the common indexed field to find the right records.

The OF option, like the USING option, is a shorthand form of the WHERE option. That is,

```
FIND order OF customer.
```

is the same as

```
FIND order WHERE order.cust-num = customer.cust-num.
```

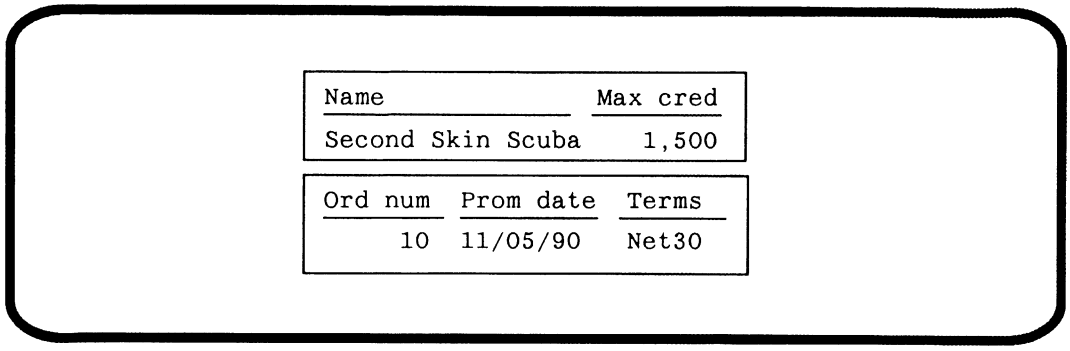
Let's look first at the t-order1.p procedure:

t-order1.p

```
FOR EACH customer:
  DISPLAY name max-credit.
  FOR EACH order OF customer:
    DISPLAY order-num pdate terms.
  END.
END.
```

On the first iteration of the outer FOR EACH block, PROGRESS reads a customer record and displays the customer's name and maximum credit limit. The inner FOR EACH block retrieves each of the orders related to that customer, if there are any, and displays each order number, promised ship date, and payment terms.

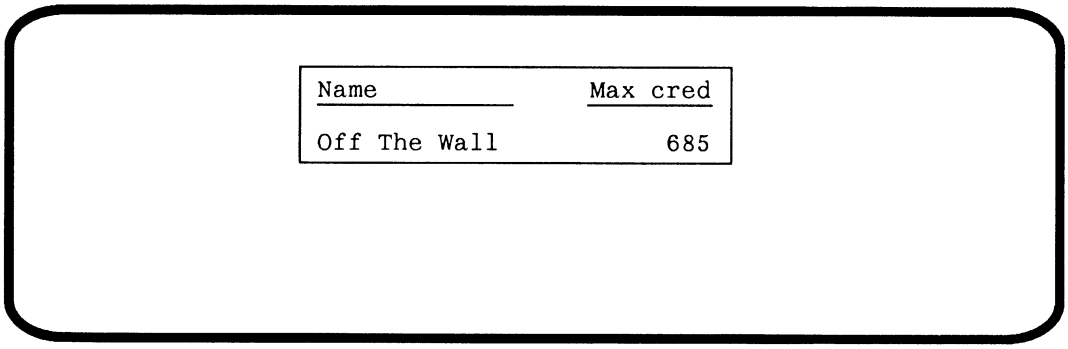
When the outer FOR EACH block reads a record for a customer who has placed orders, the inner FOR EACH block displays information about those orders:



Name	Max cred
Second Skin Scuba	1,500

Ord num	Prom date	Terms
10	11/05/90	Net30

When the outer FOR EACH block reads a record for a customer who has not placed any orders, the inner FOR EACH block exits without displaying any order information:



Name	Max cred
Off The Wall	685

A different result is produced when you combine both the customer file and the order file on the same FOR EACH block header statement.

t-order2.p

```
FOR EACH customer, EACH order OF customer:
  DISPLAY name max-credit order-num pdate terms.
END.
```

When you try to run this procedure, PROGRESS displays an error message:

```
** name is ambiguous with order.Name and customer.Name
```

Because the procedure specifies two files on the same FOR EACH block header statement and because both of these files have a name field, you need to identify from which file(s) PROGRESS should display this field. If you resolve the ambiguity in the name field and try to run the procedure again, you find that the the terms field is also ambiguous.

```
** terms is ambiguous with order.Terms and customer.Terms
```

Let's have PROGRESS display both fields from the customer file:

t-order4.p

```
FOR EACH customer, EACH order OF customer:
  DISPLAY customer.name max-credit order-num pdate customer.terms
END.
```

When you run this procedure, you see only those customers who have orders associated with them. For example, since Off The Wall hasn't placed any orders, you don't see its name displayed on this list.

```
Second Skin Scuba    1,500        10   11/05/90   2% 10/Net30

 Name                Max cred   Ord num  Prom date  Terms
Second Skin Scuba      1,500        10  11/05/90   2% 10/Net30
Match Point Tennis     1,970         6  10/01/90   Net30
Pedal Power Cycles       185         1  10/02/90   2% 10/Net30
                          .
                          .
                          .
Press space bar to continue.
```

The FOR EACH statement in the t-order4.p procedure specifies that customer records and order records are to be treated as a set; PROGRESS must find each customer *and* that customer's orders before displaying records. Customers who have not placed orders are not displayed.

9.1.8 Reading a Unique Related Record

Suppose you read an order record and then want to find the one customer record to which that order belongs. The OF option lets you read a unique related record:

t-cust1.p

```
FOR EACH order, customer OF order:
  DISPLAY order-num pdate customer.name.
END.
```

The OF option in this procedure tells PROGRESS to:

- Relate, or "join," the order file to the customer file.
- Use a common indexed field to find the right record.

Here is another example of using the OF option to find a unique, related record:

t-cust1.p

```
FOR EACH order, EACH order-line OF order, item OF order-line:
  DISPLAY order.order-num line-num item.item-num idesc.
END.
```

In this example, the FOR EACH statement tells PROGRESS to read a single record from the order file on each iteration of the block. After reading that record, PROGRESS reads the first related record from the order-line file. After reading the order-line record, PROGRESS reads the single related record from the item file. Finally, the DISPLAY statement displays information from each of the three files.

9.2 RECORD SCOPE

We began this chapter by looking at how PROGRESS reads records from your database files. Now we're going to examine this process in terms of **record scope**. **Record scope** is the length of time PROGRESS holds a record in the record buffer. The scope influences:

- When PROGRESS writes that record back to the database and when its record buffer is cleared.
- What criteria PROGRESS uses to resolve references to field names.
- What validation tests PROGRESS performs when updating records.

9.2.1 Writing Records to the Database and Clearing the Record Buffer

Look again at the t-order1.p procedure you worked with earlier in this chapter:

t-order1.p

```
FOR EACH customer:
  DISPLAY name max-credit.
  FOR EACH order OF customer:
    DISPLAY order-num pdate terms.
  END.
END.
```

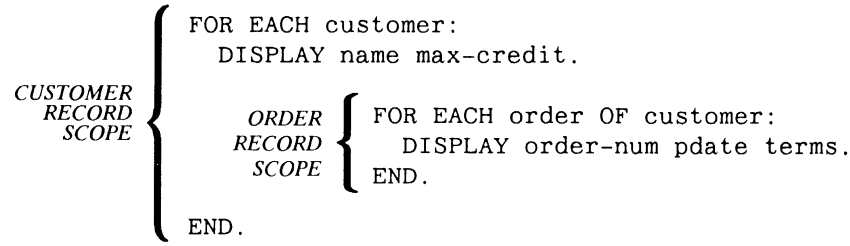
When it executes the first FOR EACH statement, PROGRESS reads a customer record into the customer record buffer. It then displays the specified fields from that record.

While the customer record is still in the record buffer, PROGRESS then looks at the next FOR EACH statement and reads an order record belonging to that customer into the order record buffer.

After displaying order information, PROGRESS clears the order record buffer, marking the end of the scope of the order record. It then reads the next order record into the record buffer and displays that order information.

When the last order record is displayed, the iteration of the outer FOR EACH block ends. At that point, PROGRESS clears the customer record buffer, marking the end of the scope of the customer record.

PROGRESS repeats these steps until all customer records and all order records related to those customers have been processed.



PROGRESS automatically scopes records to FOR EACH blocks, REPEAT blocks, and to procedure blocks.

9.2.2 Resolving Field Name References

Suppose you want to look at the promised ship date and payment terms for a particular order. You'd probably write a procedure that prompts for the order number, finds the order using that number, and then displays the desired fields.

t-order5.p

```
PROMPT-FOR order-num.
FIND order USING order-num.
DISPLAY pdate terms.
```

When you press GO (F1) to run this procedure, PROGRESS displays an error message telling you that the order-num field you specified is ambiguous; both the order and order-line files have a field named order-num.

You've seen that one way to resolve this problem is to specify the name of the file that you want to use:

```
PROMPT-FOR order.order-num.
```

An alternative solution in this case is to enclose the procedure in a DO block that has been modified by the keyword FOR.

t-order6.p

```
DO FOR order:
   PROMPT-FOR order-num.
   FIND order USING order-num.
   DISPLAY pdate terms.
END.
```

A DO block without the FOR keyword simply groups statements together. A DO block modified by the FOR keyword tells PROGRESS to scope a specific record to the DO block. The order record read by the FIND statement is cleared from the record buffer when PROGRESS encounters the end of the block. Since DO blocks don't iterate, only one order record is read by the FIND statement.

If you want to look at several records from the order file, modify this procedure so that it becomes a REPEAT block:

t-order7.p

```
REPEAT FOR order:
   PROMPT-FOR order-num.
   FIND order USING order-num.
   DISPLAY pdate terms.
END.
```

The FOR option on the REPEAT block header statement removes any ambiguity between the order and order-line files. Because REPEAT blocks loop, you'll be able to display all of the orders you want until you press END (F4). At the end of each iteration of the REPEAT block, the order record buffer is cleared.

9.2.3 Validating Records

At the end of any statement that changes an index component, PROGRESS checks to see that the record complies with the unique index definitions established in the Data Dictionary. For example, look at the t-order8.p procedure.

t-order8.p

```
REPEAT FOR order:
  PROMPT-FOR order-num.
  FIND order USING order-num.
  UPDATE order-num pdate terms.
END.
```

Press GO (F1) to run this procedure, and enter **1** for the order number:

```
Ord num Prom date  Terms
      1 10/02/90 Net30

Enter an order number between 1 and 99999
```

Change the ord-num to **2** and change the prom-date and terms fields to any values.

```
Ord num Prom date  Terms
      2 10/12/90 Net30

Enter an order number between 1 and 99999
```

When you press RETURN or GO (F1) after changing the terms field, PROGRESS displays this message:

```
** order already exists with Ord num 2.
```

At the end of the UPDATE statement, PROGRESS first checked to see if a unique index in the Data Dictionary had been modified in that record. Since you tried to change the value of the ord-num field (a unique index field) to a number already in use by another order record, PROGRESS did not allow the update. It **validated**, or checked the index against your data definitions for unique indexed fields.

You've seen that record scope is useful for writing records back to the database and clearing record buffers, resolving ambiguous field name references, and validating fields against Data Dictionary definitions. Understanding record scope is particularly important to developing multi-user applications. We'll take a closer look at record scope when we explore multi-user issues in Chapter 12 of the *PROGRESS Programming Handbook.*

9.3 PROCESSING MULTIPLE RECORDS FROM A SINGLE FILE AT ONE TIME

Every example of record reading that we've looked at has shown how PROGRESS uses the FIND or FOR EACH statement to read a single record at a time from a database file into a record buffer. But what do you do when you need to work with two records from the same file at the same time? Suppose you want to list every item you carry that has a designated substitute you can sell when the original is not in stock. In addition, you'd like to look at the original item record and the substitute item record at the same time.

PROGRESS automatically gives you one record buffer for each file that you use in a procedure. It uses that buffer to store one record at a time as needed during that procedure. If you need more than one record at a time from a single file, you can use the DEFINE BUFFER statement to define additional record buffers for that file. The t-defbuf.p procedure defines an additional buffer for you to use to look at substitute item records.

t-defbuf.p

```
DEFINE BUFFER alt-item FOR item.

FOR EACH item:
  IF subs-item NE 0
  THEN
  DO:
    FIND alt-item WHERE
    alt-item.item-num = item.subs-item.
    DISPLAY item.item-num item.idesc
            alt-item.item-num label "Alternate"
            alt-item.idesc.
  END.
END.
```

In this procedure, the statement:

```
DEFINE BUFFER alt-item FOR item.
```

creates a record buffer named alt-item that can hold records from the item file.

PROGRESS reads the first item record into the record buffer automatically reserved for the item file using the primary index, item-num, to retrieve that record. Each item record has a field called subs-item, which is the item number for that item's substitute. If the subs-item field does not have a value of 0, PROGRESS finds the substitute item and reads it into the alt-item buffer.

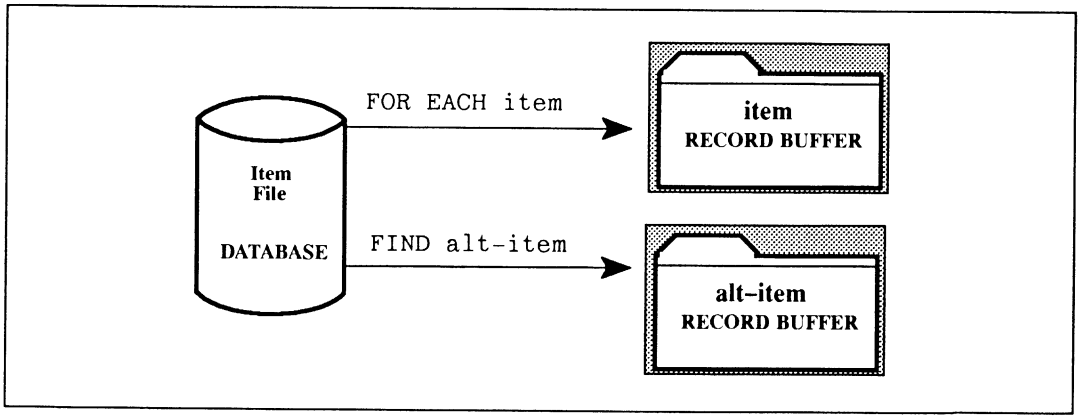


Figure 9-3: Reading Two Records

The item number and item description for each item with a designated substitute is then displayed along with the number and description of the substitute.

```
Item num Desc          Alternate Desc
00028    Ski Gloves    00029     Gloves
00032    Tennis Shorts 00033     Shorts
00033    Shorts        00032     Tennis Shorts


Procedure complete. Press space bar to continue.
```

Scoping rules for additional record buffers that you define are the same as those for record buffers automatically provided by PROGRESS.

9.4 SUMMARY

In this chapter you learned:

- How to qualify record selection with different PROGRESS statements, options, and keywords.
- How PROGRESS assigns scope to a record and how that scope is used when writing records to the database and when resolving references to fields in related files.
- How to process multiple records from a single database file at one time.

Understanding the variety of ways you can control the record-reading process is important for creating procedures for your applications.

Chapter 10
Using Variables, Expressions, Functions, and Arrays

You have now seen how to use different PROGRESS statements to add, change, display, and delete information. You've used these statements to work with database records and fields. However, your application may require procedures that work with the following:

- Temporary data that isn't stored in a database field.
- Data that is produced as a result of manipulating a single database field or as a result of combining multiple database fields.
- Groups of data stored as an array.

This chapter introduces **variables, expressions,** and **arrays,** which are the tools you use to handle these tasks. The chapter covers the following topics:

- Using variables for temporary data.
- Using expressions
- Using functions in expressions.
- Precedence of functions and operators
- Using arrays.

10.1 USING VARIABLES FOR TEMPORARY DATA

When you write a procedure, you often need a place where you can temporarily store data. A temporary field for storing data is called a **variable.**

Let's assume you want to look at a customer record and then decide whether or not to delete that record. To handle this task, the procedure you write must do the following:

1. Prompt for information which allows PROGRESS to find the customer record you want and then display the record on your screen.
2. Ask you whether you want to delete the record.
3. Use your response to determine whether to delete the record or leave it alone.

Here's one approach to performing this task:

t-var1.p

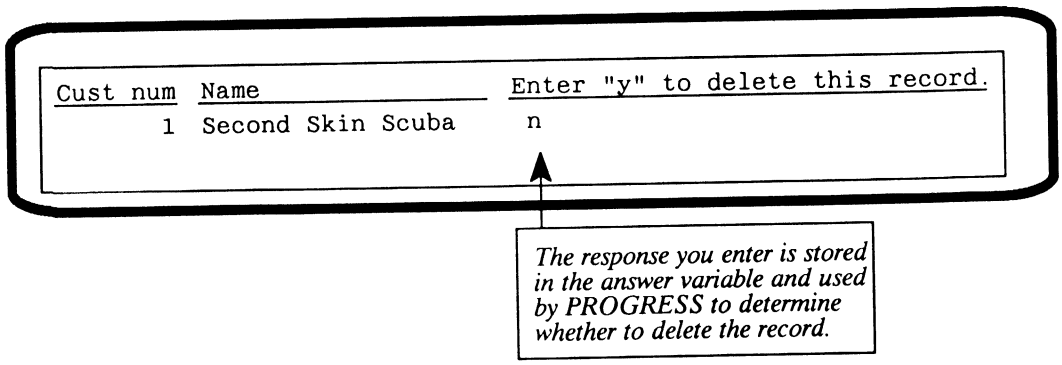
```
DEFINE VARIABLE answer AS LOGICAL FORMAT "y/n".

REPEAT:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  DISPLAY customer.name.
  answer = NO.
  UPDATE answer LABEL
     "Enter ""y"" to delete this record".
  IF answer THEN DELETE customer.
END.
```

The procedure begins by defining a variable called answer. It then prompts you for a customer number and displays the corresponding customer name. The statement

```
answer = NO.
```

sets the default value of the answer variable to No, so you won't accidentally delete a record. If you want to delete the record, you type **y** in response to the prompt "Enter 'y' to delete this record." Your response is stored in the answer variable and is used by PROGRESS to determine whether to delete the record.

Run t-var1.p, and enter **1** when you are prompted for a customer number. PROGRESS displays the name "Second Skin Scuba" and prompts you to "Enter 'y' to delete this record." For now, keep Second Skin Scuba as a customer, so accept **n**, the default response to this prompt.



The type of variable created by the DEFINE VARIABLE statement in this procedures is a **local** variable. Data in a local variable is available only within the procedure in which it is defined; that is, when this procedure ends, your response to the deletion question will no longer be stored in the answer variable.

10.1.1 Defining a Data Type for a Variable

Though variables are not part of any database file, they have many of the same characteristics as database fields. You can define a variable with the same data types and format options as you use to define a field. For example, in the procedure you just saw, the variable answer is defined as a logical data type:

t-var1.p

```
DEFINE VARIABLE answer AS LOGICAL FORMAT "y/n".
                          .
                          .
                          .
```

You use the AS option to indicate the data type for a variable. The FORMAT option defines the display format for the variable. If you use the AS option and you do not use the FORMAT option, the variable uses the default display format for its data type. To refresh your memory, Table 10–1 gives the default display formats for each PROGRESS data type.

Table 10-1: Default Display Formats for PROGRESS Data Types

Data Types	Default Display Formats
Character	X(8)
Date	99/99/99
Decimal	->>,>>9.99
Integer	->,>>>,>>9
Logical	yes/no

Chapter 4 of the *PROGRESS Programming Handbook* provides a complete summary of the display formats you can use for both fields and variables.

10.1.2 Defining a Variable LIKE a Field

Suppose you want to see a list of customers whose names begin with two letters you specify. Here's a procedure that uses a variable to help you with this task:

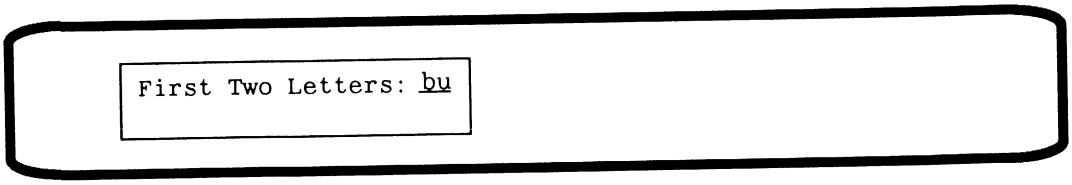
t-like.p

```
DEFINE VARIABLE cname LIKE customer.name
  FORMAT "x(2)" LABEL "First Two Letters".

REPEAT:
  SET cname WITH SIDE-LABELS.
  FOR EACH customer WHERE name BEGINS cname:
    DISPLAY name address city st zip.
  END.
END.
```

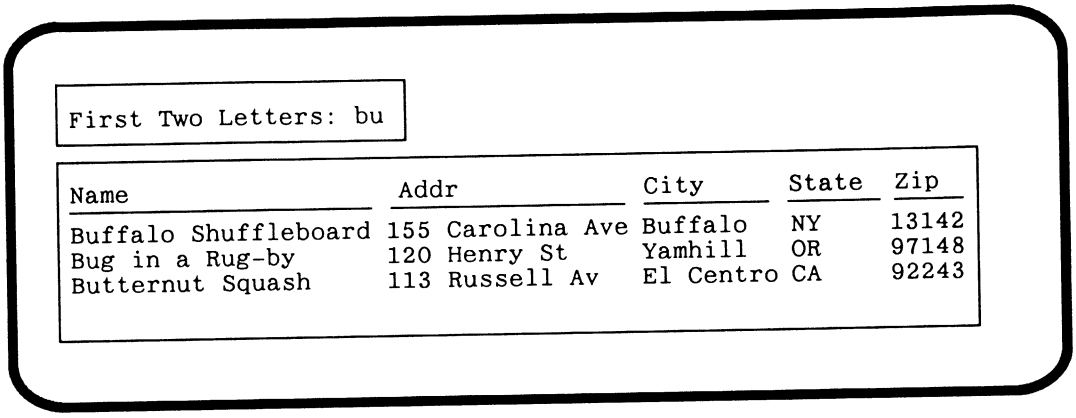
You saw in our last example that you can define a variable as any data type you want. You can also define a variable with the same characteristics as a database field.

The t-like.p procedure defines a variable called cname, and uses the LIKE option to give this variable the same characteristics as your customer file's name field with just two modifications. Instead of using the name field's Dictionary-defined display format of x(20), the variable uses the FORMAT option to specify a display format of only 2 characters. (You can also use options such as LABEL, INITIAL, DECIMALS and EXTENT to override these Dictionary-defined characteristics for a field.) The cname variable is distinguished from the name field by the label "First Two Letters." If you look at the data definition for the name field, you'll see that "name" is the label defined for that field.

When you run t-like.p, the procedure begins by letting you enter two characters, such as **bu**:



PROGRESS stores the characters in the cname variable and displays all the customers whose names begin with those letters.



The cname variable, like the answer variable you worked with in this chapter's first procedure, is a local variable; the information in a local variable is available only within the current procedure and disappears when the procedure is complete. If you want to share information among many procedures, you use *shared* and *global shared* variables. You'll learn more about these kinds of variables in Chapter 14 when you write **subprocedures**.

10.2 USING EXPRESSIONS

Suppose you are considering giving all of your customers a new maximum credit limit based upon the following:

- If a customer's maximum credit is greater than $500, their new maximum credit limit should be increased by $1000.
- If a customer's maximum credit is less than or equal to $500, their new maximum credit limit should be increased by only $100.

Before you commit to the new maximum credit limit, you want to be able to say **yes** or **no** to each customer's prospective new max-credit value. The t-expl.p procedure lets you do this.

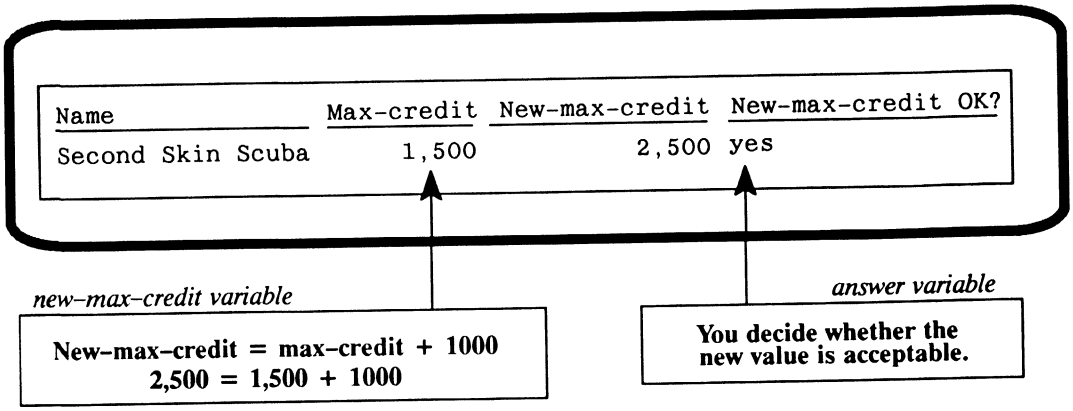
t-expl.p

```
DEFINE VARIABLE new-max-credit LIKE max-credit
                LABEL "New max-credit".
DEFINE VARIABLE answer AS LOGICAL.

FOR EACH customer:
  DISPLAY name max-credit LABEL "Max-credit".
  IF max-credit > 500
  THEN new-max-credit = max-credit + 1000.
  ELSE new-max-credit = max-credit + 100.
  DISPLAY new-max-credit.
  answer = YES.
  UPDATE answer LABEL "New max-credit OK?"
  IF answer
  THEN DO:
     max-credit = new-max-credit.
     MESSAGE "Max-credit for" name "changed to"
             max-credit.
  END.
  ELSE MESSAGE "Max-credit for" name "not changed".
END.
```

This procedure displays each customer's max-credit value and, if the maximum credit limit is greater than $500, adds $1000 to that max-credit value, and stores the new value in the variable new-max-credit. If the customer has a credit limit less than or equal to $500, $100 is added to max-credit and is stored in the new-max-credit variable.

When you decide whether the new maximum credit limit is acceptable, your response is stored in the answer variable. If the new-max-credit value is agreeable, PROGRESS changes the customer's max-credit value to the value stored in new-max-credit.

Let's run t-exp1.p. The first record you see, Second Skin Scuba, has a max-credit limit of $1,500. PROGRESS adds $1,000 to that amount, yielding a value of $2,500. It stores this amount in the new-max-credit variable. It then asks if this new value is acceptable. You can answer **yes** or **no**.



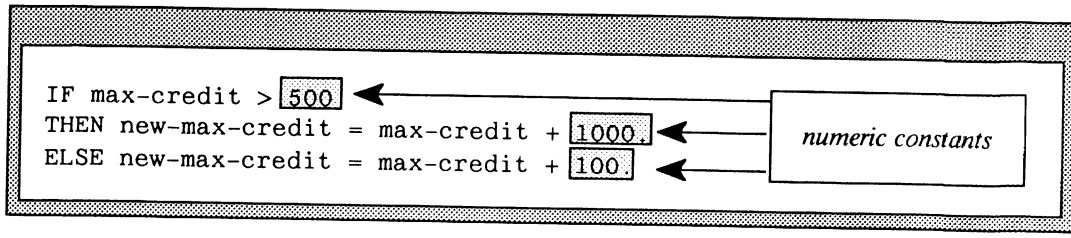
As in this procedure, you'll often want to work with a value that is the result of processing one or more database fields. Here, to create a new-max-credit value, you added either 1000 or 100 to the existing max-credit values, using two **expressions**.

```
THEN new-max-credit = max-credit + 1000.
ELSE new-max-credit = max-credit + 100.
```

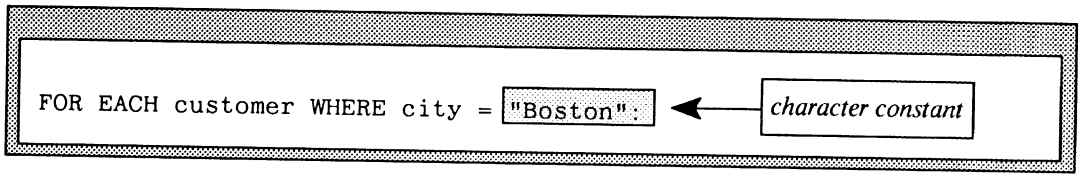
An expression consists of constants, field names, and variable names combined with operators or functions to produce a results. Expressions can perform arithmetic, comparisons, or other functions on any of the PROGRESS data types. The remaining sections of this chapter show you how to use **constants, operators,** and **functions** to form expressions.

10.2.1 Using Constants in Expressions

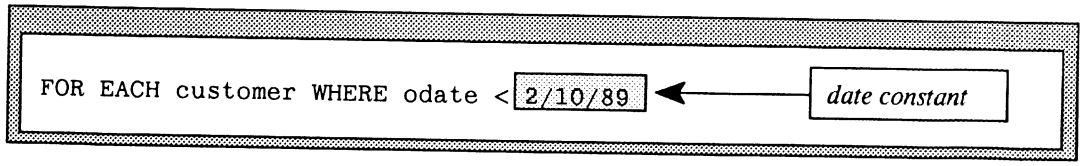
Some expressions use **constant** values to process information. Constants in a procedure can be numeric, character, date, or logical data. You added numeric constants to the max-credit field in this last procedure:



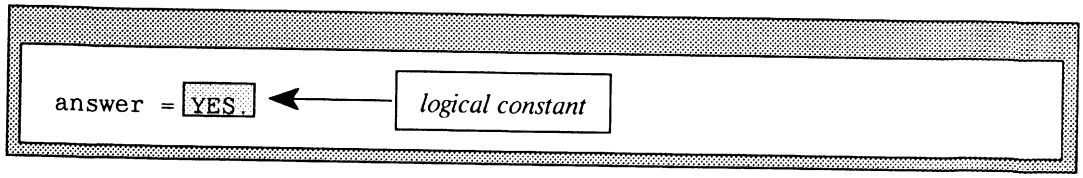
Character constants are made up of non-numeric or character data, or a combination of numeric and non-numeric data. They must be enclosed in quotation marks when you use them in a procedure. For example, you can compare the city field to a character constant to see if they are equal:



Date constants in a procedure are represented in month/day/year format. This expression looks at the date in the odate field to see if it is before February 10, 1989:



Logical constants are represented by either yes, true, no, or false:



10.2.2 Using Operators in Expressions

Operators are the symbols you use to perform numeric calculations, date calculations, character string manipulations, or data comparisons. Table 10–2 lists the operators you can use with PROGRESS and the operations they perform.

You saw how the addition (+) operator worked in t-exp1.p to create new max-credit values. Now we'll look at a few other operators and how you might use them in different application situations.

10.2.3 Using Operators for Numeric Calculations and Data Comparisons

Let's assume you want to know the value of each item currently stocked in your company's inventory. In addition, if the inventory value of any item is over $1000, you want to find out why such a high value is being kept in inventory. The t-op1.p procedure performs this function.

t-op1.p

```
DEFINE VARIABLE inv-value AS DECIMAL
  LABEL "Inventory Value".

DISPLAY "Inventory Report - Check Highlighted Items"
  WITH CENTERED.

FOR EACH item WITH CENTERED:
  inv-value = on-hand * cost.
  DISPLAY item.item-num idesc on-hand cost inv-value.

  IF inv-value GT 1000
  THEN COLOR DISPLAY MESSAGES inv-value.
END.
```

To determine the inventory value for each item, t-op1.p uses the multiplication operator (*) to multiply the value in each item's on-hand field by the value in the cost field. It stores the result in the inv-value variable. Notice the space on either side of the multiplication operator in this procedure. When you use an operator, you must leave a space to the left and to the right of that operator.

PROGRESS then compares inv-value with the constant 1000, using the logical operator GT (you can also use the > symbol). The expression "inv-value GT 1000" is called a logical expression because when PROGRESS evaluates it, the expression yields either a true or false value. If inv-value is greater than 1000, the result is true; otherwise, the result is false. If the result of this expression is true, the statement COLOR DISPLAY MESSAGES tells PROGRESS to highlight the value stored in inv-value.

Table 10–2: PROGRESS Operators

PROGRESS Operator	Operation
- (UNARY NEGATIVE)	Reverses the sign of a numeric expression.
+ (UNARY POSITIVE)	Returns the positive or negative value of a numeric expression.
/ (DIVISION)	Divides one numeric expression by another to produce a decimal result.
* (MULTIPLICATION)	Multiplies two numeric expressions.
- (DATE SUBTRACTION)	Subtracts a number of days from a date to produce a date result; or subtracts one date from another to return the number of days between the two dates.
- (SUBTRACTION)	Subtracts one numeric expression from another.
+ (DATE ADDITION)	Adds a number of days to a date to produce a date result.
+ (CONCATENATION)	Concatenates two character strings or character expressions.
+ (ADDITION)	Adds two numeric expressions.
< or LT	Returns a true value if the first of two expressions is less than the second.
< = or LE	Returns a true value if the first of two expressions is less than or equal to the second.
> or GT	Returns a true value if the first of two expressions is greater than the second.
> = or GE	Returns a true value if the first of two expressions is greater than or equal to the second.
= or EQ	Returns a true value if two expressions are equal.
< > or NE	Compares two expressions and returns a true value if they are not equal.
NOT	Reverses the true or false value of an expression.
AND	Returns a true value if each of two logical expressions is true.
OR	Returns a true value if either of two logical expressions is true.

When you run this procedure, PROGRESS displays the item number, quantity on hand, unit cost, and total inventory value for each item and highlights values that are over 1000:

Inventory Report - Check Highlighted Items

Item num	Desc	On hand	Cost	Inventory Value
00001	Fins	0	42.95	0.00
00002	Tennis Racquet	0	64.50	0.00
00003	Sweat Band	142	2.55	362.10
00004	Cycle Helmet	141	75.00	10,575.00
00005	Leotard, Black	56	19.95	1,117.20
00006	Ski Poles	15	27.50	412.50
00007	Buoyancy Vest	37	125.00	4,625.00
	:			
	:			

10.2.4 Performing Date Calculations

Suppose you want information about customers whose orders will be shipped a week later than the promised delivery date. To find out which customers have overdue orders, you need to work with two date values:

- The date the orders were promised for delivery (pdate).
- The date the orders will actually be shipped (sdate).

You use the date addition (+) operator and the logical operator GT (>) to compare the shipping date against the promised delivery date:

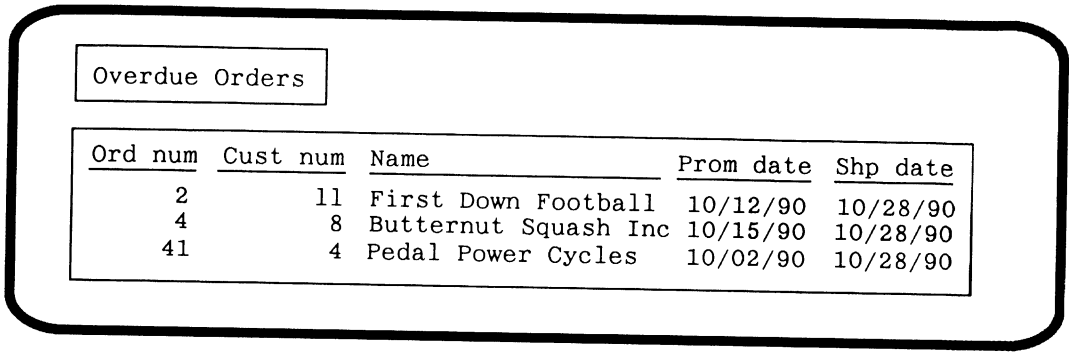
t-op2.p

```
DISPLAY "Overdue Orders".

FOR EACH order WHERE shipped = "":
  IF sdate GT (pdate + 7)
  THEN
  DO:
    FIND customer OF order.
    DISPLAY order.order-num order.cust-num
            customer.name pdate sdate.
    END.
END.
```

In this procedure, PROGRESS retrieves each order record whose shipped field indicates that the order has not yet been shipped. The double quotes "", called a **null string**, indicate there is no data in the shipped field. The procedure then checks to see if the shipping date for each of those orders is later than one week after the promised date. This is done with the following expression:

```
sdate GT (pdate + 7)
```

If this expression returns a true result, PROGRESS displays information about those orders and the customers who ordered them.



Overdue Orders

Ord num	Cust num	Name	Prom date	Shp date
2	11	First Down Football	10/12/90	10/28/90
4	8	Butternut Squash Inc	10/15/90	10/28/90
41	4	Pedal Power Cycles	10/02/90	10/28/90

For further information about all the operators described in Table 10–2 refer to the *PROGRESS Language Reference* manual.

10.3 USING FUNCTIONS IN EXPRESSIONS

When you write an application, you typically have both a large number and wide variety of jobs to do. Rather than write lengthy procedures to handle many of these tasks, you can use a special set of PROGRESS tools called **functions**. Functions are "shortcuts" to handling tasks as diverse as calculating the logarithm of an expression and determining the day of the week on which a particular date falls. There are many types of functions built into the PROGRESS language:

- **Aggregate value** functions evaluate groups of values.
- **Arithmetic** functions perform mathematical operations on numeric values.
- **Character** functions manipulate character strings or expressions.
- **Date** functions provide day, month, and year information for an application.
- **Data conversion** functions change data from one data type to another.
- **Decision** functions evaluate one of two expressions, depending on the value of a specified condition.

- **Record status** functions determine record conditions such as availability for processing.
- **Screen status** functions return information about data on the screen.
- **System status** functions give operating system, terminal, and keyboard information.

Table 10–3 presents a list of PROGRESS functions, grouped by function type. We'll spend the remainder of this chapter looking at ways you can use some of these functions to handle different kinds of application tasks.

Table 10–3: PROGRESS Functions

Function Type	PROGRESS Functions		
Aggregate value	ACCUM COUNT-OF	MAXIMUM MINIMUM	NUM-ENTRIES
Arithmetic	EXP LOG MODULO	RANDOM ROUND SQRT	TRUNCATE
Character	BEGINS CAPS ENTRY FILL	INDEX KEYWORD LC LENGTH	LOOKUP MATCHES R-INDEX SUBSTRING
Date	DATE DAY MONTH	TODAY WEEKDAY YEAR	
Data conversion	ASC CHR DECIMAL	INTEGER STRING TRIM	
Decision	IF-THEN-ELSE NOT OR		
Record status	AMBIGUOUS AVAILABLE CAN-FIND FIRST	FIRST-OF LAST LAST-OF LOCKED	NEW RECID SEEK
Screen status	ENTERED FRAME-COL FRAME-DB FRAME-DOWN FRAME-FIELD FRAME-FILE	FRAME-INDEX FRAME-LINE FRAME-NAME FRAME-ROW FRAME-VALUE GO-PENDING	INPUT MESSAGE-LINES NOT ENTERED OVERLAY SCREEN-LINES
System/Database Info.	CAN-DO CONNECTED DBNAME DBRESTRICTIONS DBTYPE GATEWAYS KBLABEL KEYCODE KEYFUNCTION KEYLABEL	LASTKEY LDBNAME LINE-COUNTER NUMALIASES NUMDBS OPSYS PAGE-NUMBER PAGE-SIZE PDBNAME PROGRAM-NAME	PROGRESS PROPATH RETRY SDBNAME SEARCH SETUSERID TERMINAL TIME USERID

10.3.1 Performing Arithmetic Calculations

Suppose you want to calculate the square root of a number. One way to do this calculation is to use the operators you saw earlier and develop a formula that gives you the square root. A faster approach would be to use the arithmetic function SQRT. The arithmetic functions listed in Table 10-3 let you quickly perform calculations that would otherwise take much longer if you had to write your own formulas or procedures to get the same results.

Here's a procedure that uses the ROUND function to round adjusted max-credit values to the nearest hundred dollars:

t-func1.p

```
DEFINE VARIABLE new-credit LIKE max-credit.
  LABEL "New credit limit".

FOR EACH customer WHERE max-credit > 2500:
  DISPLAY cust-num name max-credit
    LABEL "Current credit limit".
  PAUSE 1 NO-MESSAGE.
  new-credit = ROUND( (max-credit * 1.1) / 100, 0) * 100.
  DISPLAY new-credit.
  PAUSE 1 NO-MESSAGE.
END.
```

PROGRESS displays the customer number, name, and maximum credit limit for each customer who has a credit limit greater than $2500. Each time a customer record is displayed, the procedure pauses for a second (without displaying a message about the pause), displays a new credit limit, and pauses again. This new credit limit, which is stored in the new-credit variable, is the result of a calculation in which the max-credit value is increased by 10 percent and then rounded to the nearest hundred dollars.

Try running t-func1.p for yourself to see how this procedure works:

Cust num	Name	Current credit limit	New credit limit
6	Lift Line Skiing	11,744	12,900
16	Thundering Surf Inc.	3,290	3,600
21	Ship Shape Yachting	3,266	3,600
22	Pocket Billiards Co.	2,732	3,000
24	On Target Rifles	5,051	5,600
29	Chip's Poker	5,866	6,500
50	Stay Afloat Swimming	2,776	3,100

New credit values rounded to the nearest hundred.

10.3.2 Evaluating Character Expressions

At certain points in an application, you may want to take a particular action based on the result of a character expression. For example, you may want to convert uppercase letters in some string to lowercase letters, or vice-versa. Or, you might want to find a group of records that begin with some character string that you specify. Character functions like LC, CAPS, BEGINS, and MATCHES are helpful for jobs like these. You probably remember this procedure from Chapter 10:

t-func2.p

```
FOR EACH customer WHERE name MATCHES "Sh*":
  DISPLAY name.
END.
```

The MATCHES function checks each customer record to see if the name field matches the pattern "Sh*". The asterisk (*) tells PROGRESS to do a wildcard search to retrieve all records that begin with Sh.

Name
Ship Shape Yachting
Shark Snack Snorkeling

Records beginning with Sh

10.3.3 Evaluating Date Expressions

You'll often want to build into an application a way to determine dates, such as the number of days between when an order is placed and when it is shipped. By writing procedures that use date functions you can easily incorporate date information into an application.

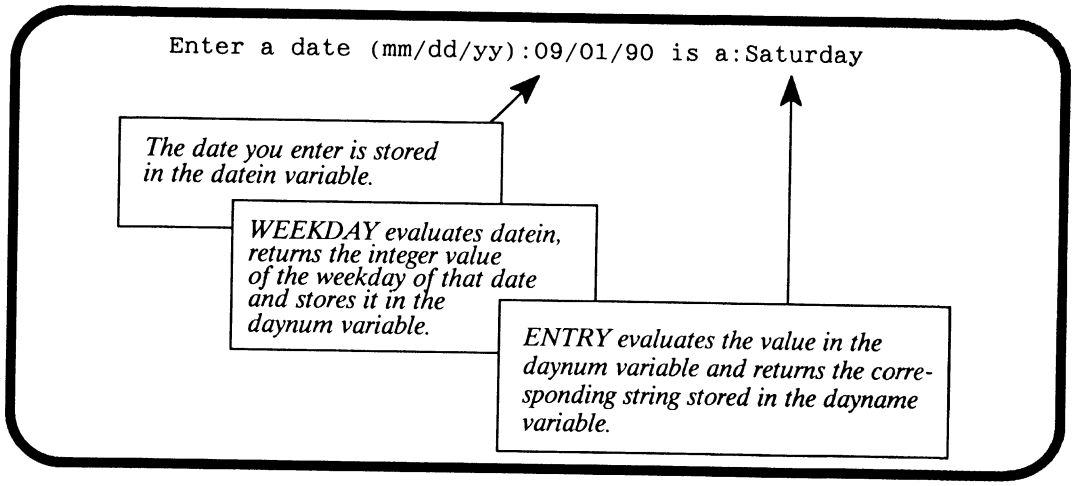
Here's a procedure that tells you the day of the week on which a certain date falls:

t-func3.p

```
DEFINE VARIABLE datein AS DATE.
DEFINE VARIABLE daynum AS INTEGER.
DEFINE VARIABLE dayname AS CHARACTER INITIAL
  "Sunday,Monday,Tuesday,Wednesday,Thursday,
   Friday,Saturday".

SET datein LABEL
  "Enter a date (mm/dd/yy)".
daynum = WEEKDAY(datein).
DISPLAY ENTRY (daynum, dayname)
  LABEL "is a" WITH SIDE-LABELS.
```

The procedure lets you enter a date, which it then stores in the `datein` variable. The WEEKDAY function evaluates the date you enter, returns the day of the week as an integer from 1 (Sunday) to 7 (Saturday), and stores that integer in the `daynum` variable.

Notice that the `dayname` variable has been defined as a single character string with each day of the week separated from the next by a comma. The character function ENTRY evaluates the integer stored in the `daynum` variable, and returns the character string whose position in the `dayname` variable list corresponds to the integer in `daynum`. Run t-func3.p to see how it works. Enter the date **08/09/86**. When you press RETURN, **Saturday** is displayed.



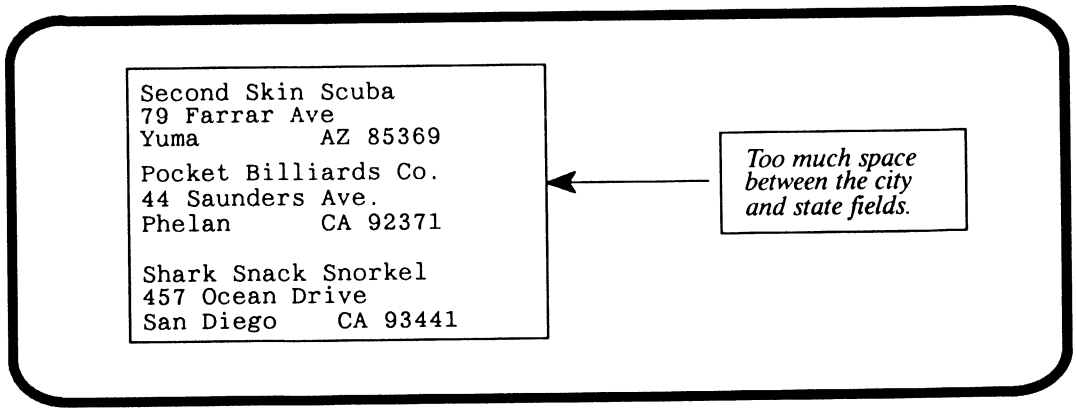
10.3.4 Converting Data from One Data Type to Another

To perform certain kinds of application tasks, it is often necessary to change data from one type to another. For example, suppose you want to create mailing labels for all of the customers in your customer file. One approach would be to use a simple procedure like the t-mailcs.p procedure.

t-mailcs.p

```
FOR EACH customer BY st:
  DISPLAY name SKIP
          address SKIP
          city st zip SKIP(1)
          WITH NO-LABELS NO-BOX.
END.
```

The mailing labels you get from this procedure look like these:



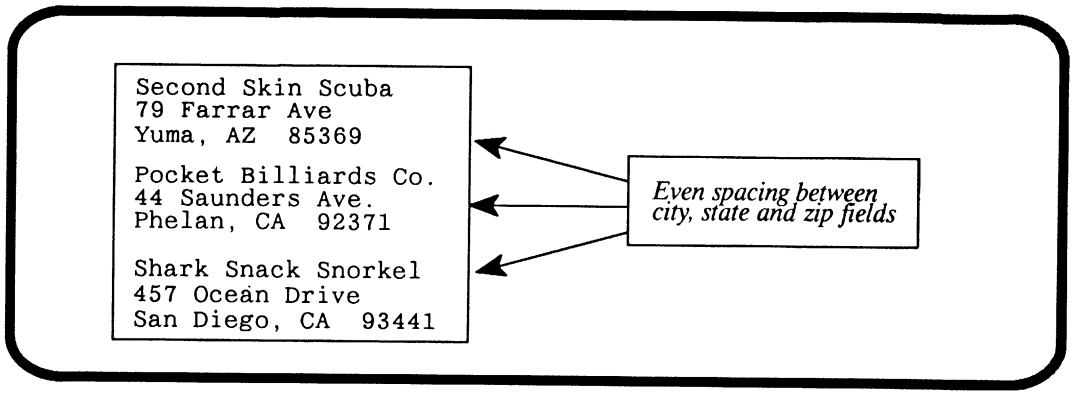
You can improve the way these labels look by slightly modifying this procedure:

t-func4.p

```
FOR EACH customer BY st:
  DISPLAY name SKIP
          address SKIP
          city + ", " + st + "  " + STRING(zip,"99999")
          FORMAT "x(30)" SKIP(2) WITH NO-LABELS.
END.
```

The procedure uses the concatenation operator(+) to join, or concatenate, the city and state fields and separate these fields by a comma and a space. Since only character strings can be concatenated, you must convert the zip code field, which is an integer expression, to a character value. This is done with the STRING function, which converts the value of any data type into a character value. Here, the zip code is converted to a character value with a format of "99999". The entire concatenated string is formatted for up to 30 characters.

The labels you get when you run this procedure look much better than those generated by t-mailcs.p:



10.3.5 Determining the Status of a Record

There are several functions you can use to quickly determine record conditions such as whether a record is available for processing or whether a record can be found using a specified selection criteria.

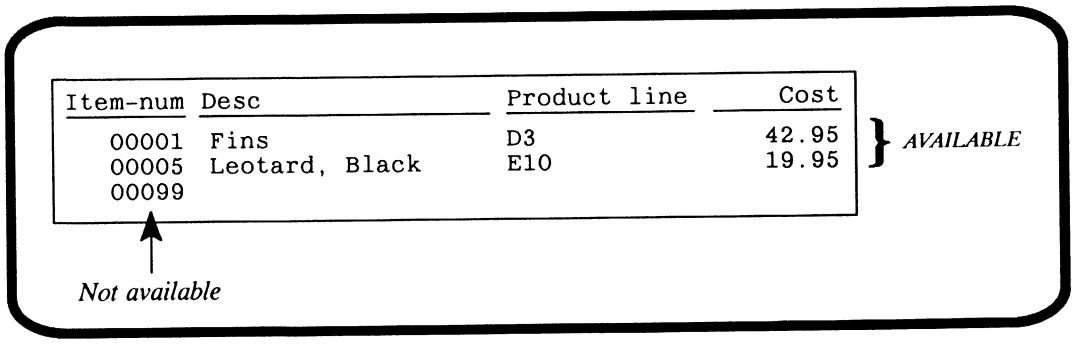
The AVAILABLE function lets you determine the action to be taken based upon the availability of a particular record, as shown in t-func5.p.

t-func5.p

```
REPEAT:
  PROMPT-FOR item.item-num.
  FIND item USING item-num NO-ERROR.
  IF AVAILABLE item
  THEN DISPLAY idesc prod-line cost.
  ELSE MESSAGE "Not Found".
END.
```

This procedure prompts you for an item number and uses that number to retrieve the appropriate item record. The FIND statement in this procedure uses the NO-ERROR option to bypass both PROGRESS' default error action and the message you normally get when a record does not exist.

If a record is found that corresponds to the item number, PROGRESS displays a description of the item, the product line it belongs to, and the item's cost. Otherwise, the message "Not Found" is displayed. See how this procedure works by running t-func5.p:



You will sometimes want to use the AMBIGUOUS function in conjunction with AVAILABLE to determine whether a record is unavailable because more than one record met the specified FIND criteria. In the procedure you just saw, item-num is a unique index field, and therefore AMBIGUOUS is not needed.

10.3.6 Checking the Status of Your System

System status functions provide you with information about your system, such as data about your terminal, keyboard, your current database, or the operating system you are using.

Suppose you plan to run an application on any of the PROGRESS-supported operating systems and you want to be able to list the files in your working directory without having to be concerned about which operating system you are currently using. You can use the OPSYS function to evaluate the operating system you are running the application on, and then use the appropriate command to list the files in your directory.

For example:

t-func6.p

```
IF OPSYS = "unix" then UNIX ls.
ELSE IF OPSYS = "msdos" then DOS dir.
ELSE IF OPSYS = "os2" then OS2 dir.
ELSE IF OPSYS = "vms" then VMS directory.
ELSE IF OPSYS = "btos" then  BTOS "[sys]<sys>files.run" files.
ELSE MESSAGE  OPSYS  "is an unsupported operating system".
```

Go ahead and run this procedure if you'd like to look at the files in your working directory.

As you continue to work through this Tutorial, you'll see uses for many of the functions shown in Table 10–3. In addition, you can find detailed descriptions of each of these functions in the *PROGRESS Language Reference* manual.

10.4 PRECEDENCE OF FUNCTIONS AND OPERATORS

The order in which PROGRESS evaluates an expression depends on the precedence of its operators. Consider this arithmetic expression:

```
3 * 5 + 2
```

The result of this expression is different, depending on the order in which you process the multiplication and addition operators:

```
3 * (5 + 2) = 21
```

but

```
(3 * 5) + 2 = 17
```

Table 10–4 shows the precedence, or order of processing, of PROGRESS functions and operators.

If an expression contains two operators of equal precedence, PROGRESS evaluates the expression from left to right. If the operators are not of equal precedence, PROGRESS evaluates the operator of the higher precedence first. You can change the default order used to evaluate an expression by grouping expressions and operators with parentheses.

10.5 USING ARRAYS

Arrays contain multiple elements. For example, the Mnth-shp field of the item file is an array field. It contains one element for every month of the year. The *extent* of an array is the number of elements contained in an array.

Both fields and variables can be arrays. In procedures, you can:

- Refer to all the elements in the array at once.
- Refer to specific array elements.
- Refer to a range of array elements.

You can use the FRAME-INDEX function to determine the array element in which the cursor was last positioned.

Table 10–4: Precedence of Functions and Operators

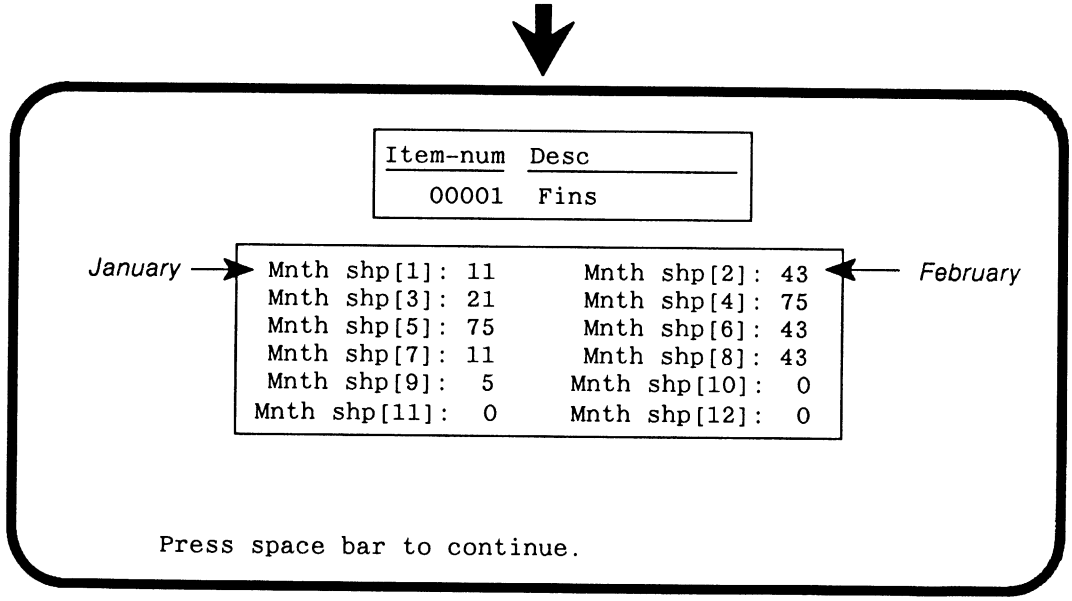
Name of Operator	Precedence
- UNARY NEGATIVE + UNARY POSITIVE	7 (highest)
MODULO / DIVISION * MULTIPLICATION	6
- DATE SUBTRACTION - SUBTRACTION + DATE ADDITION + CONCATENATION + ADDITION	5
MATCHES LT or < LE or < = GT or > GE or > = EQ or = NE <> BEGINS	4
NOT	3
AND	2
OR	1 (lowest)

10.5.1 Using an Entire Array

The mnth-shp field of the item file in the demo database is an array field containing one element for each month of the year. Suppose you want to display, for each item in the database, how many of that item have been shipped in each of the 12 months of the year.

t-array.p

```
FOR EACH item:
  DISPLAY item-num idesc WITH FRAME a CENTERED.
  DISPLAY mnth-shp WITH 2 COLUMNS CENTERED 1 DOWN.
END.
```



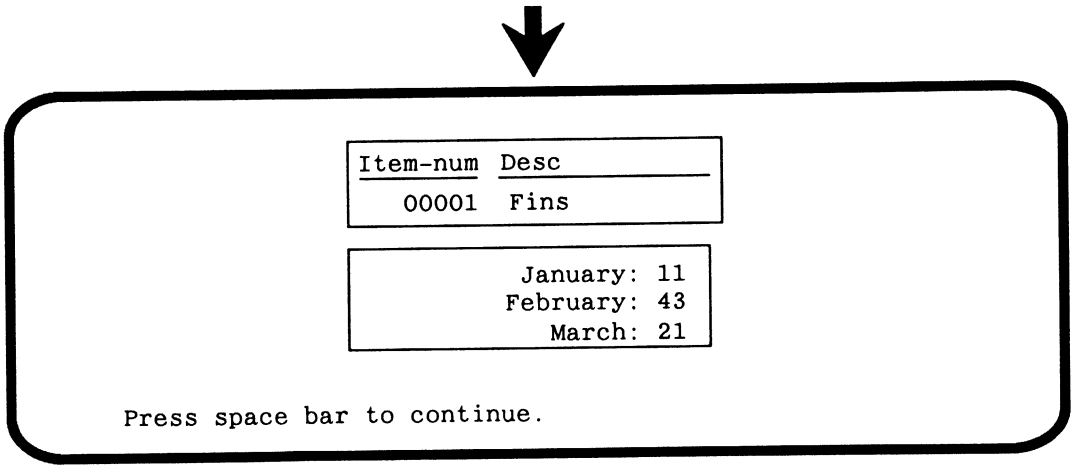
The second DISPLAY statement in the procedure displays the Mnth-shp field. Since the procedure does not name a specific array element, the DISPLAY statement displays all the elements in the Mnth-shp array field.

10.5.2 Using Specific Array Elements

If you don't want to see all the elements in the array, you can refer to just selected elements. For example, suppose you just want to see how many items were shipped in January, February, and March.

t-array2.p

```
FOR EACH item:
  DISPLAY item-num idesc WITH FRAME a CENTERED.
  DISPLAY mnth-shp[1] LABEL "January"
          mnth-shp[2] LABEL "February"
          mnth-shp[3] LABEL "March"
          WITH 1 COLUMN CENTERED 1 DOWN.
END.
```



As with fields and variables, you can use Format phrases to describe formatting characteristics for array elements. In the t-array2.p procedure, the LABEL option defines a label for each of the array elements being displayed.

10.5.3 Using a Range of Array Elements

You may want to look at a range of array elements, such as the first through the third elements. For example, you might want to see for each item, the amount shipped for that item over a period of three months starting with a specific month.

t-array3.p

```
DEFINE VARIABLE beg-month AS INTEGER FORMAT "99".

SET beg-month LABEL "Starting month" WITH SIDE-LABELS.
IF beg-month > 10 THEN DO:
   beg-mnth = 10.
   MESSAGE "Showing 4th quarter.".
END.

FOR EACH item WITH 1 DOWN:
   DISPLAY item-num idesc WITH FRAME a.
   DISPLAY beg-mnth LABEL "Starting Month"
      mnth-shp [beg-mnth FOR 3] FORMAT "zzz9".
END.
```

This procedure asks you for the number of a month and then shows you, for each item, how many items were shipped in that month and the following 2 months. The specification

```
beg-mnth FOR 3
```

tells PROGRESS to start with the array element that is equal to the value stored in the beg-mnth variable and to work with that and the next two elements. Here is the output of this procedure:

```
Starting month: 06

Item-num  Desc
   00001  Fins

Starting Month  Mnth-shp  Mnth-shp  Mnth-shp
            06        43        11        43

   Press space bar to continue.
```

10.5.4 Defining Array Variables

If you are defining an array variable, you can supply initial values for each element in the array. For example:

```
DEFINE VARIABLE array-var
  AS CHARACTER EXTENT 3 INITIAL ["Add","Delete","Update"].
```

If you do not supply enough values to fill up the elements of the array, PROGRESS puts the last value you named into the remaining elements of the array. If you supply too many values, you receive an error message.

10.6 SUMMARY

In this chapter, you learned that

- You can use variables as temporary fields for storing information required by a procedure.
- You can use operators and functions to manipulate expressions. An expression can be made up of constants, field names, and variable names combined with operators to produce a result.
- You can define arrays to hold collections of data of the same data type.

You'll see additional examples which use variables, expressions, operators, functions, and arrays as you continue your work through this Tutorial.

Chapter 11
Understanding and Modifying Screen Designs

Throughout this book you have been using the DISPLAY statement to display information on the screen. This chapter takes a closer look at how PROGRESS displays data and how you can control the way your screen looks. The chapter covers the following topics:

- Changing overall frame characteristics.
- Changing the field and variable characteristics.
- Using frames for input.
- Defining frame layout and processing properties.
- Using the message area.
- Understanding and changing frame behavior
- Frame and terminal relationships.
- Using color screen displays.

PROGRESS uses **frames** to display data. If you don't tell PROGRESS how you want to display data, it uses default display characteristics. However, you can include in your procedures either general or specific instructions about how you want to display data.

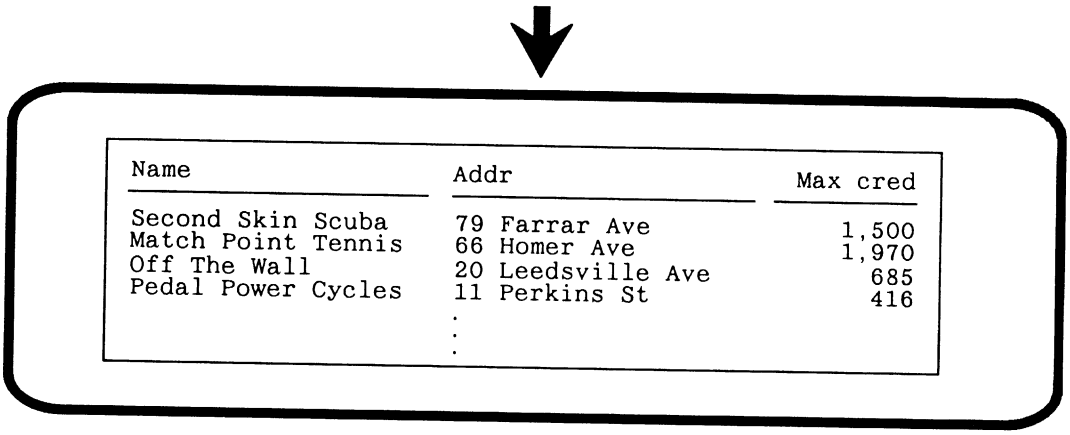
A frame can use all but the bottom three lines of the screen. Two of these lines are used for PROGRESS error messages and any messages produced by your procedure with the MESSAGE statement. The last line is used for status and help messages. Some terminals have 24 display lines, leaving 21 lines available for frames. Other terminals have 25 display lines, leaving 22 lines available for frames.

The length of a frame must be less than or equal to the length of the display area available on the screen. If you think some users might run your procedures on terminals that have 24 lines and some on terminals that have 25 lines, then design your frames to fit on the smaller screen. That way, you can be sure that displays work the way you planned.

Let's look at a simple procedure to understand how PROGRESS goes about designing a frame.

t-frame.p

```
FOR EACH customer:
  DISPLAY name address max-credit.
END.
```



Name	Addr	Max cred
Second Skin Scuba	79 Farrar Ave	1,500
Match Point Tennis	66 Homer Ave	1,970
Off The Wall	20 Leedsville Ave	685
Pedal Power Cycles	11 Perkins St	416
	.	
	.	
	.	

PROGRESS decides how to format the data displayed because the procedure contains no explicit formatting statements. PROGRESS lays out the frame when it compiles the procedure. That is, in a top to bottom pass of a procedure during compilation, PROGRESS lays out the frames necessary for the procedure to do any displays to the screen. Incorporated in these frame layouts are the PROGRESS display defaults along with any formatting characteristics you may have specified in the procedure.

There are two levels of frame characteristics:

- Characteristics that affect the entire frame, such as whether it uses a box or whether labels should be above the data or beside the data.
- Characteristics that affect individual fields.

11.1 CHANGING OVERALL FRAME CHARACTERISTICS

Overall frame characteristics affect the way an entire frame appears. You use a **Frame phrase** to describe the overall characteristics of a frame. A Frame phrase always begins with the word WITH and is followed by one or more of the options listed in Table 11–1. In Table 11–1, the text in the Default column indicates how PROGRESS displays the frame if you do not specify a particular characteristic.

Table 11–1: Frame Phrase Options

WITH Option	Purpose	Default
ATTR-SPACE	Reserves spaces for field attributes such as underlining and highlighting.	Depends on the type of terminal, unless you specify ATTR-SPACE or NO-ATTR-SPACE.
CENTERED	Centers the frame on the terminal screen	Does not center the frame.
COLOR	Specifies a video attribute or color to use for the background of the frame, and an optional video attribute or color for fields enabled for input.	The default color for the background of a frame is NORMAL.
COLUMN *expression*	Positions the frame in a specified column on the screen.	Positions the frame in column 1.
n COLUMNS	Formats data into *n* columns with side-labels.	Places fields in the frame across the screen, wrapping onto as many lines as needed to accommodate all the fields. Labels are placed above the fields.
DOWN	Displays multiple instances or repetitions of the data in the frame.	Displays multiple instances of the data if the frame is the default frame for the innermost iterating block(s) of the procedure. Otherwise, displays a single instance in the frame.
expression DOWN	Displays a specified number of records in a single frame.	Displays multiple instances of the data if the frame is the default frame for the innermost iterating block(s) of the procedure. Otherwise, displays a single instance in the frame.
FRAME *frame*	Names a frame.	Uses the default frame for a block.

Table 11-1: Frame Phrase Options (Continued)

WITH Option	Purpose	Default
NO-ATTR-SPACE	Does not reserve spaces for field attributes such as underlining and highlighting.	Depends on the type of terminal, unless you specify ATTR-SPACE or NO-ATTR-SPACE.
NO-BOX	Does not display a box around the frame.	Displays a box around the frame.
NO-HIDE	Suppresses the automatic hiding of the frame.	Automatically hides the frame according to a set of default hiding rules.
NO-LABELS	Does not display labels.	Displays labels.
NO-UNDERLINE	Does not underline labels appearing above fields.	Underlines labels appearing above fields.
NO-VALIDATE	Disregards all validation conditions specified in the Dictionary for the fields in the frame.	Validates all fields in the frame according to specifications in the Dictionary.
OVERLAY	Indicates that the frame can overlay any other frame that does not use the TOP-ONLY option.	The frame cannot overlay other frames.
PAGE-BOTTOM	Displays the frame at the bottom of the page each time a page is filled.	Does not display the frame at the bottom of the page each time a page is filled.
PAGE-TOP	Displays the frame each time output begins on a new page.	Does not redisplay the frame on each page.
RETAIN *n*	Retains a specified number of iterations when the frame scrolls.	Does not retain on the screen any iterations already displayed.
ROW *expression*	Positions the frame in a specified row.	Positions the frame in the next available row on the screen.

Table 11–1: Frame Phrase Options (Continued)

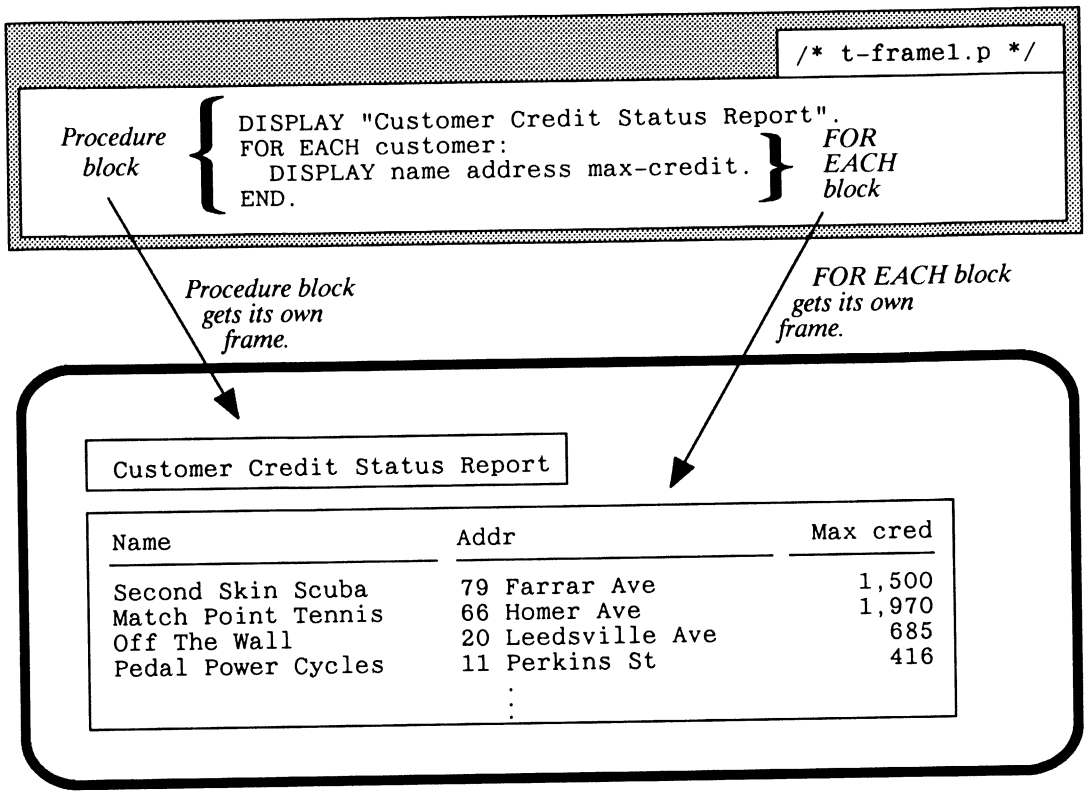
WITH Option	Purpose	Default
SCROLL *n*	Displays a scrolling rather than a paging frame.	Displays a paging frame.
SIDE-LABELS	Displays field labels to the left of the data.	Displays field labels above the corresponding field.
TITLE *expression*	Displays a title as part of the top line of the box around a display frame.	No title.
TOP-ONLY	Indicates that no other frame can overlay this frame.	Other frames that use the OVERLAY option can overlay this frame.
WIDTH *n*	Specifies the number of columns in a frame.	Uses a width based on the fields you are displaying and the width of the terminal you are using.

Let's look at how you can use some of these options to describe overall frame characteristics.

11.1.1 Naming the Frames You Want to Use

If you don't name any frames in a procedure, PROGRESS decides which displays go into which frames.

PROGRESS uses blocks as a way of deciding when to design a frame. Whenever PROGRESS sees a FOR EACH block, a REPEAT block, or a procedure block, it automatically sets up a frame for that block to use. Therefore, in the following procedure, PROGRESS displays two frames — one for the procedure block and one for the FOR EACH block.

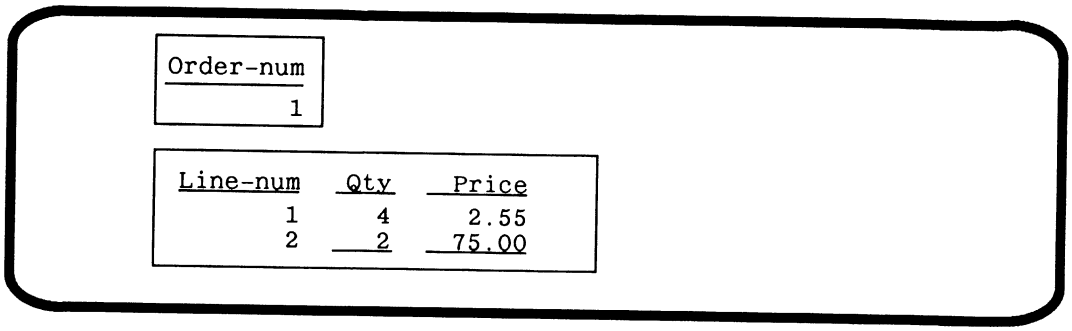


It's easy to use frames other than those PROGRESS sets aside automatically. Take a look at another example:

t-frame2.p

```
REPEAT:
  PROMPT-FOR order.order-num.
  FIND order USING order-num.
  FOR EACH order-line OF order:
    DISPLAY line-num.
    UPDATE qty price.
  END.
END.
```

Knowing that PROGRESS automatically sets up frames for procedure blocks, FOR EACH blocks, and REPEAT blocks, you can tell before running this procedure that there should be 3 frames — one for the procedure block, one for the REPEAT block, and one for the FOR EACH block.



As you can see, the REPEAT block has its own frame for displaying the order number and the FOR EACH block has its own frame for displaying the line number, quantity, and price. The procedure block does have its own frame but, because nothing is being displayed in that block, the frame does not appear on the screen. That frame is called a **null frame**.

Now suppose you want to display the line number in the same frame as the order number:

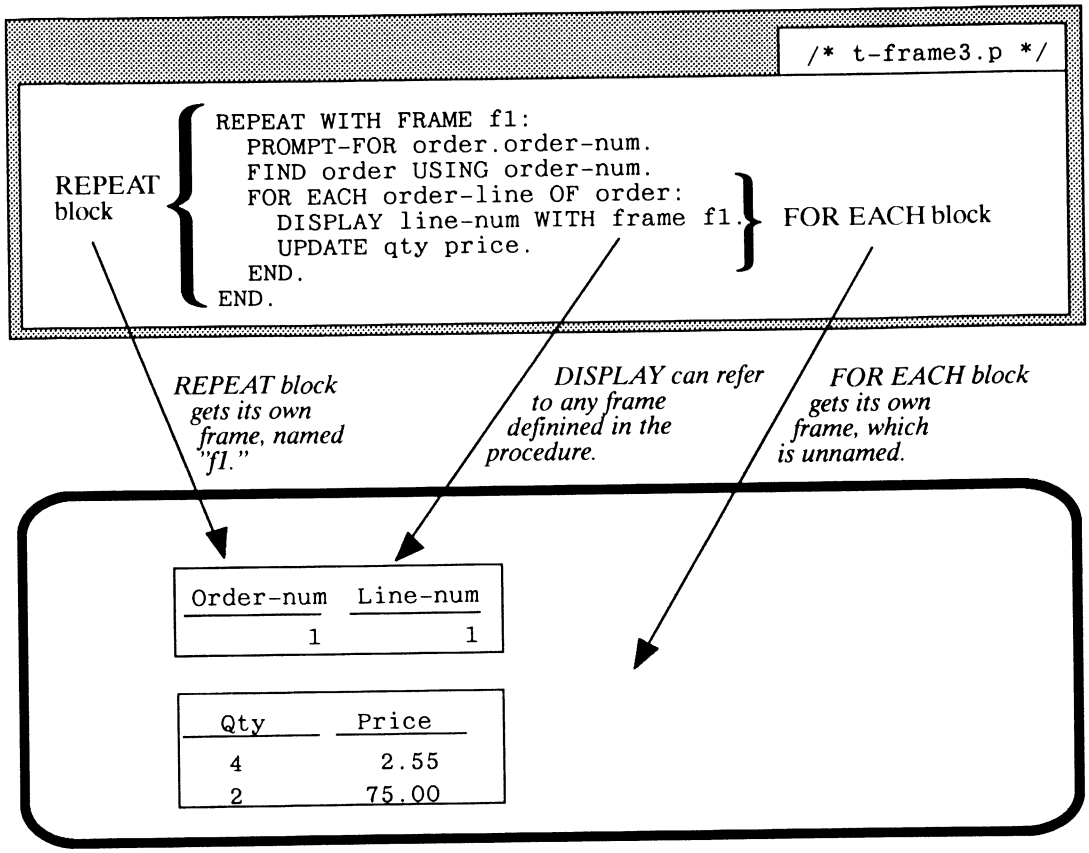
t-frame3.p

```
REPEAT WITH FRAME f1.
  PROMPT-FOR order.order-num.
  FIND order USING order-num.
  FOR EACH order-line OF order:
    DISPLAY line-num WITH FRAME f1.
    UPDATE qty price.
  END.
END.
```

Using the FRAME option in the REPEAT block header statement tells PROGRESS to set up a frame for the REPEAT block as it ordinarily would, but to name that frame "f1." The reason for naming the frame is simply so that you can refer to that frame elsewhere in the procedure. You can use any name you want for a frame as long as the name includes at least one non-numeric character. This example uses "f1" as the frame name.

Using the FRAME option in the DISPLAY statement tells PROGRESS to use a frame other than the one that has been automatically set up for the FOR EACH block. In particular, it tells PROGRESS to display the line number using the f1 frame that was designed for the REPEAT block.

Look at how these two FRAME options affect the output of the procedure:



So, you can use a frame that is not a default frame, simply by naming it.

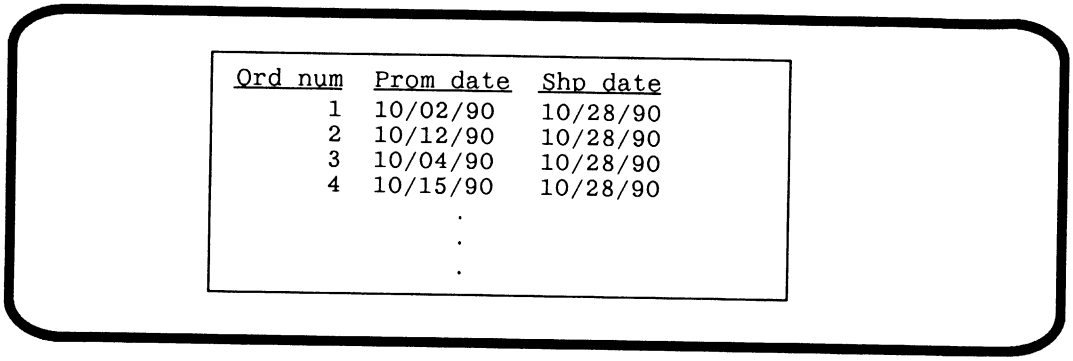
11.1.2 Determining How Many Records to Display in a Frame

You've probably noticed that sometimes PROGRESS displays multiple records in a frame at a time and sometimes displays just one record. For example:

t-frame4.p

```
FOR EACH order:
  DISPLAY order-num pdate sdate.
END.
```

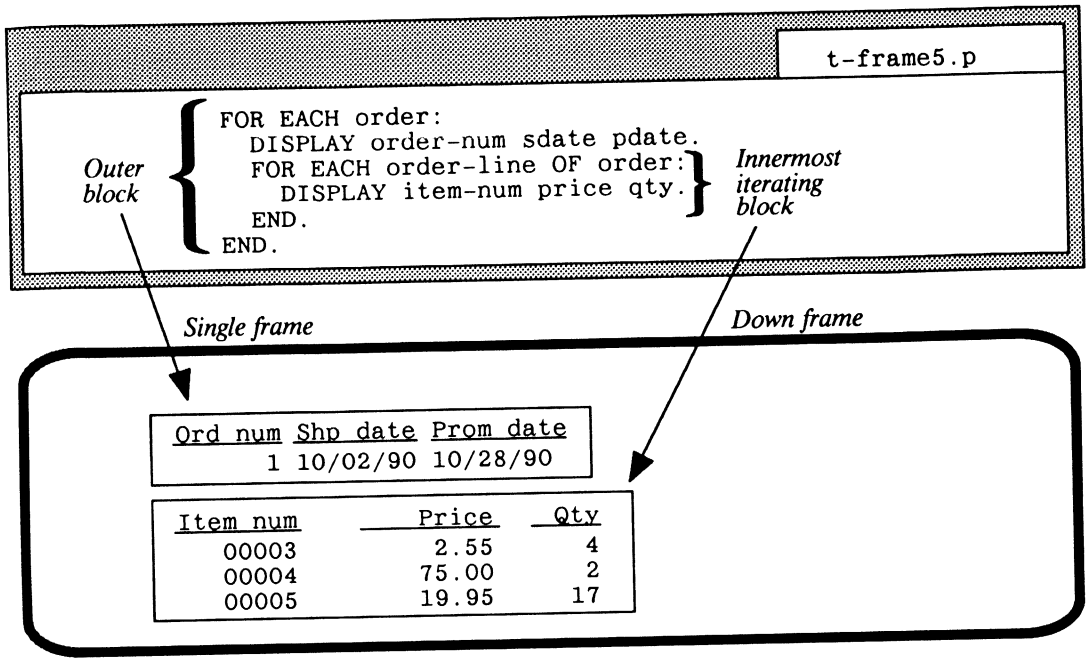
In the first iteration of the FOR EACH block, PROGRESS displays information for the first order and then advances to the next display position on the screen. On the next iteration, PROGRESS displays information for the next order in the new display position. When the screen is full, PROGRESS displays the "Press space bar to continue" message.



Ord num	Prom date	Shp date
1	10/02/90	10/28/90
2	10/12/90	10/28/90
3	10/04/90	10/28/90
4	10/15/90	10/28/90
	.	
	.	
	.	

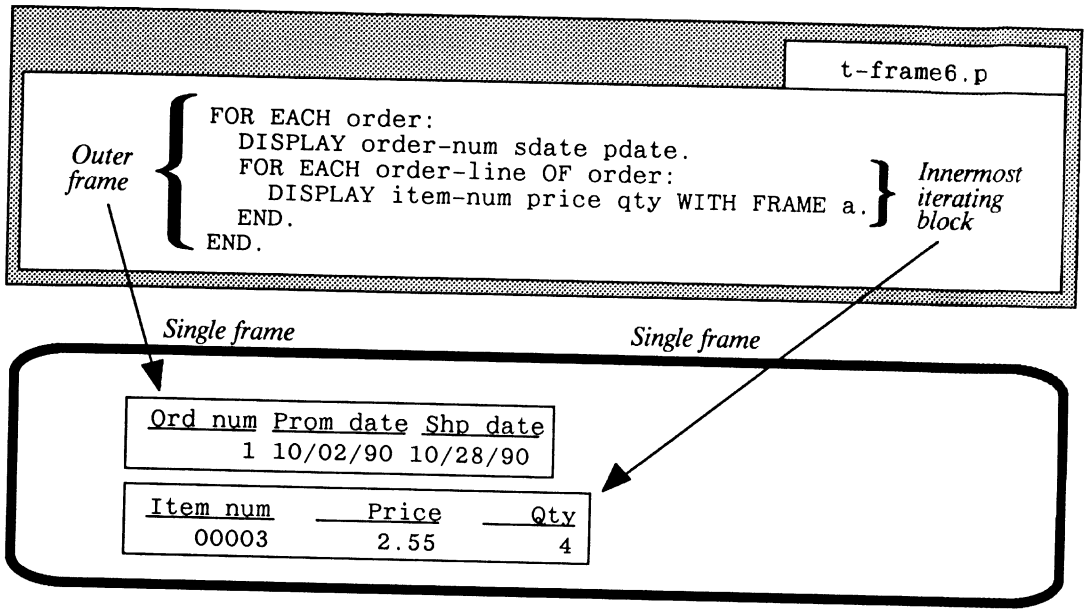
When PROGRESS displays multiple records in one frame, that frame is called a multi frame, or a down frame. When PROGRESS displays a single record in a frame, that frame is called a single frame, or a 1-down frame. Unless you say otherwise, a frame is a single frame except when it is the default frame for an innermost iterating block and it is scoped to the block, in which case the frame is a down frame.

In the t-frame5.p procedure, PROGRESS makes both a single frame and a down frame:



Here, PROGRESS displays the first order record. Then in the next FOR EACH block, PROGRESS displays the first order-line of that order. In the second iteration of the inner FOR EACH block, PROGRESS displays the next order line in the next available screen position. In this example, the inner FOR EACH block is the innermost iterating block of the procedure. Therefore, the default frame for that block is a down frame. The DISPLAY statement inside the block uses that default frame.

You can have the DISPLAY statement use a different frame by simply naming a frame in the statement. For example:

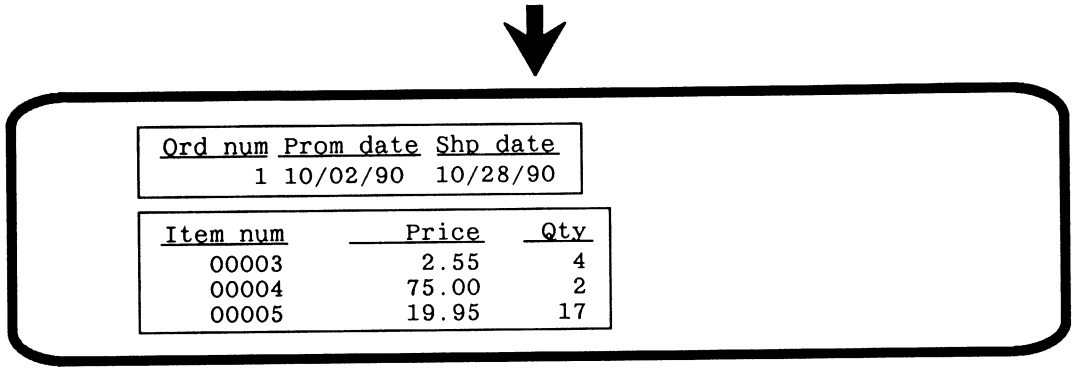


In this example, the default frame for the inner FOR EACH block is a down frame because that block is the innermost iterating block in the procedure. But the DISPLAY statement in that block does not use the default frame. Instead, it uses frame a. Since frame a is not the default frame for the innermost iterating block, it is a single frame instead of a down frame.

You use the DOWN option to tell PROGRESS to create a down frame:

t-frame7.p

```
FOR EACH order:
  DISPLAY order-num sdate pdate.
  FOR EACH order-line OF order:
    DISPLAY item-num price qty WITH FRAME a DOWN.
  END.
END.
```



Ord num	Prom date	Shp date
1	10/02/90	10/28/90

Item num	Price	Qty
00003	2.55	4
00004	75.00	2
00005	19.95	17

For more information about down and single frames, see Chapter 7 of the *PROGRESS Programming Handbook.*

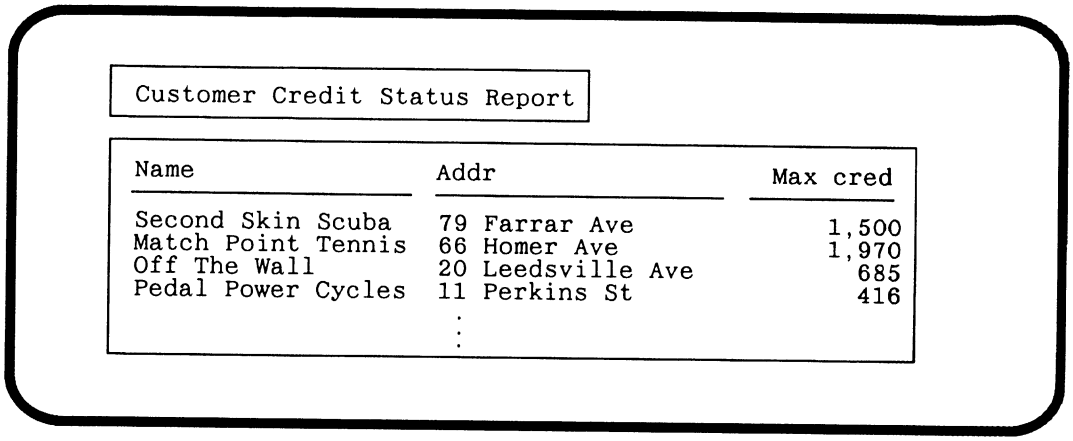
11.1.3 Describing Other Overall Frame Characteristics

Suppose you want to produce a report on the screen showing the customer's credit status. You want to display a title at the top of the screen and then list each customer's name, address, and max-credit. You could produce your report using PROGRESS' default frames, like this:

t-frame1.p

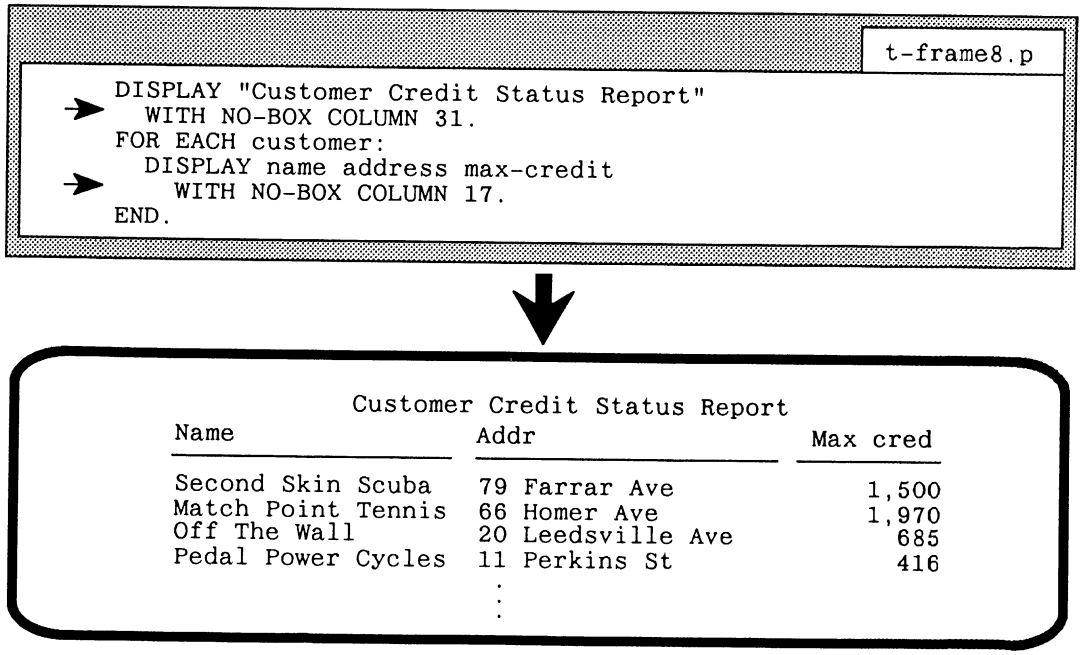
```
DISPLAY "Customer Credit Status Report".
FOR EACH customer:
  DISPLAY name address max-credit.
END.
```

The first DISPLAY statement uses the default frame for the procedure block. The second DISPLAY statement uses the default frame for the FOR EACH block. Since it is the innermost FOR EACH block, the second frame is a down frame. When you run t-frame1.p, it displays the following:



```
Customer Credit Status Report

Name                Addr                  Max cred
------------------  --------------------  --------
Second Skin Scuba   79 Farrar Ave          1,500
Match Point Tennis  66 Homer Ave          1,970
Off The Wall        20 Leedsville Ave       685
Pedal Power Cycles  11 Perkins St           416
                      .
                      .
                      .
```

Now let's use some frame options to change the procedure slightly:



t-frame8.p

```
DISPLAY "Customer Credit Status Report"
  WITH NO-BOX COLUMN 31.
FOR EACH customer:
  DISPLAY name address max-credit
    WITH NO-BOX COLUMN 17.
END.
```

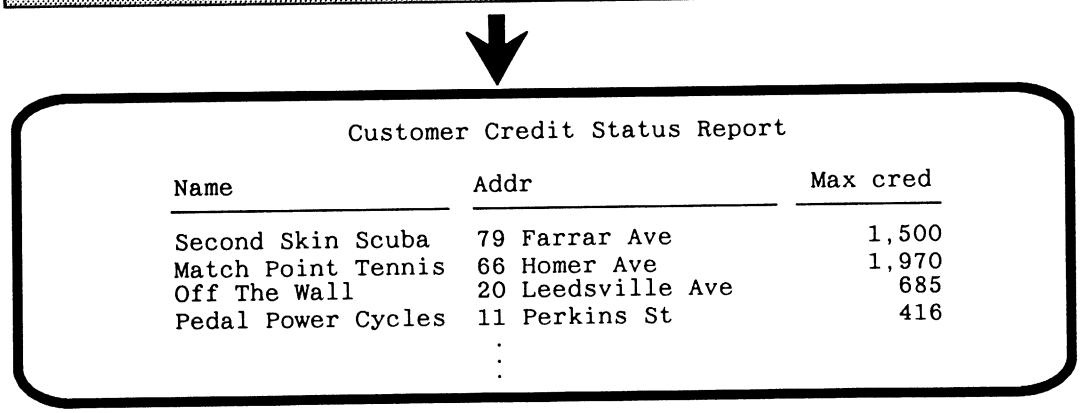
```
                Customer Credit Status Report
Name                Addr                  Max cred
------------------  --------------------  --------
Second Skin Scuba   79 Farrar Ave          1,500
Match Point Tennis  66 Homer Ave          1,970
Off The Wall        20 Leedsville Ave       685
Pedal Power Cycles  11 Perkins St           416
                      .
                      .
                      .
```

The Frame phrase in the first DISPLAY statement (WITH NO-BOX COLUMN 31) tells PROGRESS to display the frame containing the words "Customer Credit Status Report" starting in column 31 on the screen. NO-BOX tells PROGRESS to not use the box that normally surrounds a frame. The Frame phrase in the second DISPLAY statement is similar.

This display looks better than the one produced by the procedure that used default frames. However, there is at least one problem you'd probably want to fix. The report header is too close to the body of the report. That's easy to change. By adding SKIP(2) to the first DISPLAY statement, you tell PROGRESS to skip 2 lines after the report header:

t-frame9.p

```
DISPLAY "Customer Credit Status Report" SKIP(2)
  WITH NO-BOX COLUMN 31.
FOR EACH customer:
  DISPLAY name address max-credit
    WITH NO-BOX COLUMN 17.
END.
```



Customer Credit Status Report

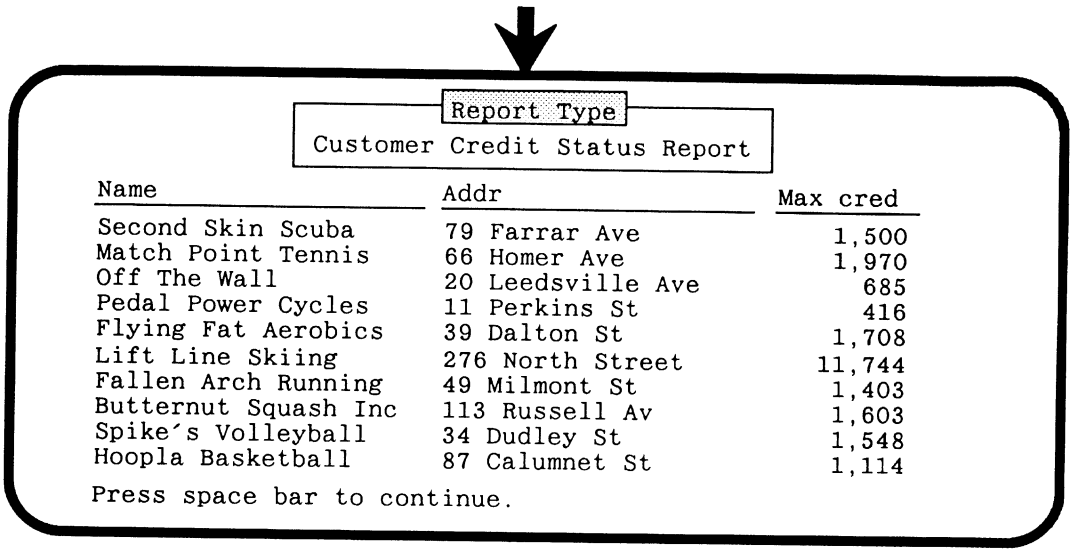
Name	Addr	Max cred
Second Skin Scuba	79 Farrar Ave	1,500
Match Point Tennis	66 Homer Ave	1,970
Off The Wall	20 Leedsville Ave	685
Pedal Power Cycles	11 Perkins St	416
	.	
	.	
	.	

You can use any number you like with the SKIP option. Go ahead and try a few different values and see what happens to the display.

Let's use some other options with the Frame phrase:

t-fram10.p

```
DISPLAY "Customer Credit Status Report"
  WITH TITLE "Report Type" CENTERED.
FOR EACH customer:
  DISPLAY name address max-credit
    WITH NO-BOX 10 DOWN CENTERED RETAIN 2.
END.
```



In this example, the first DISPLAY statement uses the Frame phrase WITH TITLE "Report Type" CENTERED:

- The CENTERED option tells PROGRESS to center the frame on the display screen.
- The TITLE option names a title to be used with the DISPLAY. PROGRESS always places the title in the center of the top line of the box that surrounds the frame being displayed.

The second DISPLAY statement displays customer information:

- The 10 DOWN option tells PROGRESS to display only 10 customers on the screen at a time.
- The RETAIN 2 option tells PROGRESS to redisplay the last two customers at the top of the screen after clearing the screen to make room for the next set of customers.
- The CENTERED option centers the display of customer records.

11.1.4 Some Rules for Using Frame Phrases

When you use a Frame phrase, that phrase applies to the entire frame regardless of the statements that use that frame. Different frame phrases that apply to the same frame cannot conflict. For example:

```
DISPLAY name WITH FRAME a COLUMN 15.
```

conflicts with:

```
DISPLAY address WITH FRAME a COLUMN 30.
```

11.2 CHANGING THE FIELD AND VARIABLE CHARACTERISTICS

When you display a field or variable, either for a display only or for user input, PROGRESS applies special characteristics to that field or variable. You can use a Format phrase to override these characteristics. Table 11-2 lists the options you can use in the Format phrase.

Table 11-2: Format Phrase Options

FORMAT Phrase	Purpose	Default
AT *n*	Starts the frame field in column *n* within the frame.	Starts the display in the next available column.
AS *datatype*	Creates a frame field and variable with the specified data type.	Does not create a new frame field and variable.
ATTR-SPACE	Reserves spaces for field attributes such as underlining and highlighting.	Depends on the type of terminal, unless you specify ATTR-SPACE or NO-ATTR-SPACE.

Table 11–2: Format Phrase Options (Continued)

FORMAT Phrase	Purpose	Default
AUTO-RETURN	When you fill the field with input data, PROGRESS automatically moves to the next input field.	Does not automatically move out of the field. Use TAB or RETURN to move on to the next field.
BLANK	Tells PROGRESS to display blanks for the frame field instead of the actual value.	Displays the frame field value.
COLON *n*	Positions the frame field so the colon between the label and the data is in column *n*.	The placement of the colon is dependent on the default placement of the label.
COLUMN-LABEL *label[!label]*	Names the label you want to display above the field, optionally using more than one line.	Displays the label above the field, using one line.
DEBLANK	Removes leading blanks from character fields on input.	Does not remove leading blanks.
FORMAT *string*	Defines the format in which values will be displayed or entered.	A field uses the format in the Dictionary, a variable uses the format of the variable definition, and an expression uses the default format for the data type of the expression.
HELP *string*	Defines the help message to be displayed when the user is entering data into the frame field.	PROGRESS displays a help string specified in the Dictionary, if any.
LABEL *string*	Specifies the label for a field, variable, or expression.	Uses the label from the Dictionary or variable definition.
LIKE *field*	Creates a frame field and variable with the same definition as a specified field.	Does not create a new frame field and variable.

Table 11–2: Format phrase Options (Continued)

FORMAT Phrase	Purpose	Default
NO-ATTR-SPACE	Does not reserve spaces for field attributes such as underlining and highlighting.	Depends on the type of terminal, unless you specify ATTR-SPACE or NO-ATTR-SPACE.
NO-LABEL	Does not use a label.	Uses the default label.
TO *n*	Positions the frame field so the last position is in column *n*.	The column in which the display ends is dependent on the column in which the display began and the display format.
VALIDATE	Validates the frame field value against specified criteria and displays a message if the validation fails.	Performs no validation unless validation was specified in the Dictionary.

Let's take a look at how you can use some of these options to describe the display characteristics of a field or variable.

11.2.1 Describing the Display Format of a Field or Variable

If you don't include any formatting statements in a procedure, PROGRESS uses default formatting to display fields and variables. For example, the t-fmt1.p procedure uses default formatting for the name and max-credit fields:

t-fmt1.p

```
FOR EACH customer:
  DISPLAY name.
  UPDATE max-credit.
END.
```

Here, the name field is 20 characters long and the max-credit field is 10 spaces long. PROGRESS determines how to display these fields by first checking to see if any formatting phrases were used in the procedure. If not, it uses the Data Dictionary display formats to display the name and max-credit fields.

```
Name                 Max cred
Second Skin Scuba        1,500
```

To change a display format, you can change the Dictionary definition, but you probably don't want to do that every time you want to display a field differently. Instead, you can use the FORMAT option with the DISPLAY and UPDATE statements, like this:

t-fmt2.p

```
FOR EACH customer:
  DISPLAY name FORMAT "x(10)".
  UPDATE max-credit FORMAT ">,>>9".
END.
```

```
Name       Max cred
Second Ski     1,500
```

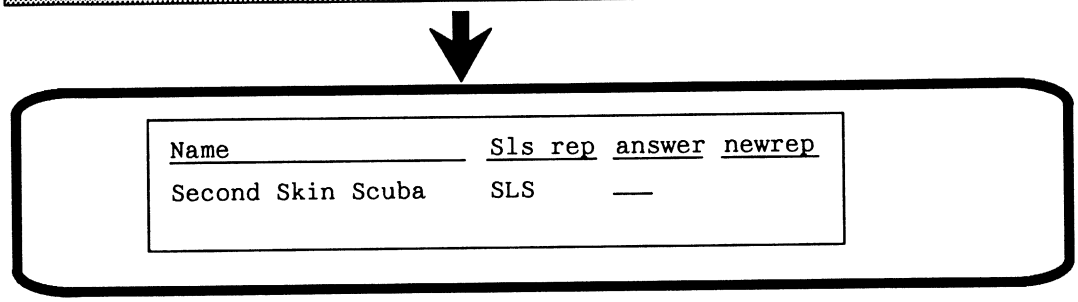
You can see that there are now only 10 spaces for the name field, but there are still 8 spaces for the max-credit field even though the new format for the field should take up only 5 spaces. That's because the label of the field, Max cred, requires 8 spaces. PROGRESS makes enough room for either the data or the field label, whichever is larger.

Because we changed the format of the name field, it appears as if only Second Ski is stored in the database and that the remainder of the name has been deleted. That's not true. When you change the format of a field, you are merely changing the way that field is displayed, not the data that is actually stored in the database. In fact, if you run t-fmt1.p again, you will see that the full name of Second Skin Scuba is still stored in the database.

When displaying variables, PROGRESS uses the same formatting rules as for fields. For example:

t-fmt3.p

```
DEFINE VARIABLE newrep AS CHARACTER FORMAT "x(3)".
DEFINE VARIABLE answer AS LOGICAL.

FOR EACH customer:
  DISPLAY name sales-rep.
  SET answer.
  IF answer THEN DO:
    SET newrep.
    sales-rep = newrep.
    DISPLAY sales-rep.
  END.
END.
```



Name	Sls rep	answer	newrep
Second Skin Scuba	SLS	—	

In this example, PROGRESS allocates 6 spaces for the label of the answer variable, only 3 spaces for the actual variable, and six spaces for the newrep label. How does PROGRESS determine how much space to allocate?

- First, PROGRESS determines whether any format phrases were used in the procedure.
- If not, it uses the default display format for the variable.
- Table 11–3 lists the default display formats for each of the data types.

Table 11-3: Default Display Formats For Data Types

Data Type of Expression	Default Format
Character	x(8)
Date	99/99/99
Decimal	->>,>>9.99
Integer	->,>>>,>>9
Logical	yes/no

Given these defaults, PROGRESS would have used a display format of x(8) for the newrep variable (a character variable) and a display format of yes/no for the answer variable (a logical variable). However, since newrep was given a format of x(3), PROGRESS accepts 3 characters of input. You override these defaults in the same way you override the default display formats for fields, by using the FORMAT option. For example:

t-fmt4.p

```
DEFINE VARIABLE newrep AS CHARACTER FORMAT "x(3)".
DEFINE VARIABLE answer AS LOGICAL FORMAT "Change/Keep".

FOR EACH customer:
  answer = no.
  DISPLAY name sales-rep.
  UPDATE answer.
  IF answer THEN DO:
    SET newrep.
    sales-rep = newrep.
    DISPLAY sales-rep.
  END.
END.
```

In this example, PROGRESS allocates enough spaces in the answer variable for the value "Change" and enough spaces in the newrep variable for the label of the variable newrep.

```
Name                Sls rep  answer  newrep
Second Skin Scuba   SLS      Keep
```

You don't have to use the FORMAT option in the DEFINE VARIABLE statement. You can use it with any data handling statement that displays data:

t-fmt5.p

```
DEFINE VARIABLE newrep AS CHARACTER.
DEFINE VARIABLE answer AS LOGICAL.

FOR EACH customer:
  answer = NO.
  DISPLAY name sales-rep.
  UPDATE answer FORMAT "Change/Keep".
  IF answer THEN DO:
    SET newrep FORMAT "x(3)".
    sales-rep = newrep.
    DISPLAY sales-rep.
  END.
END.
```

11.2.2 Describing the Label for a Field or Variable

If you don't include any statements about labels in a procedure, PROGRESS uses default labels for fields and variables.

For example:

t-fmt1.p

```
FOR EACH customer:
  DISPLAY name.
  UPDATE max-credit.
END.
```

PROGRESS determines what label to give to a field by first checking to see if any format options were used. If not, it checks the Dictionary to see if you specified labels or column labels for the fields. If a field has both a label and a column label, PROGRESS uses the column-label as the default label. In this example, the name field has a label of "Name" in the Dictionary; the max-credit field has a label of "Max cred." If these fields did not have labels, PROGRESS would use the field name as the label.

```
Name                  Max cred
Second Skin Scuba        1,500
```

So, how can you change the label that PROGRESS uses when displaying a field? You have two choices:

1. Change the label or column label in the Dictionary. Although this will work, you probably don't want to have to change the label for a field every time you want to display that field a little differently.

2. Use the LABEL or COLUMN-LABEL option with the DISPLAY, PROMPT-FOR , SET, and UPDATE statements. This is the more common practice. For example:

```
                                                    t-lbl1.p
FOR EACH customer:
  DISPLAY name LABEL "Customer Name".
  UPDATE max-credit LABEL "Maximum Credit".
END.
```

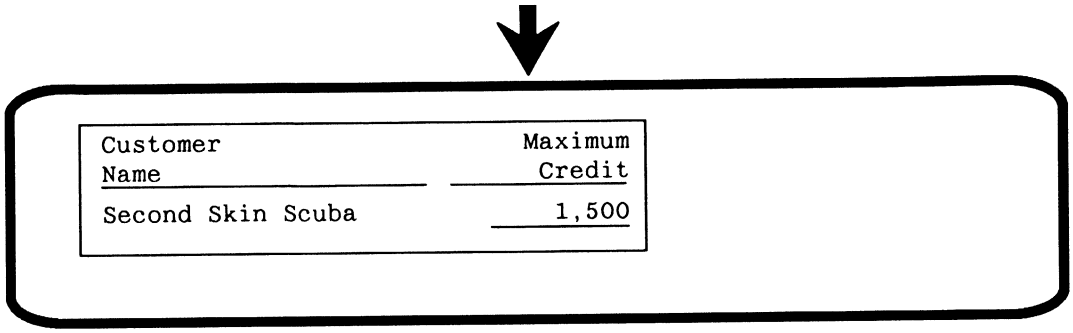
```
Customer Name         Maximum Credit
Second Skin Scuba        1,500
```

When you run this procedure, you can see that the name and max-credit fields now have the new labels, "Customer Name" and "Maximum Credit."

To use a "stacked" or column label, change the t-lbl1.p procedure this way.

t-lbl2.p

```
FOR EACH customer:
  DISPLAY name COLUMN-LABEL "Customer!Name".
  UPDATE max-credit COLUMN-LABEL "Maximum!Credit".
END.
```



You can see that the labels for the name and max-credit fields are stacked. When you want to use a column label for a field, separate the words in the label with an exclamation point (!) and the enclose the entire label in quotes. Note that there are no spaces before or after the exclamation point; any spaces you insert also appear in the label.

When displaying variables, PROGRESS uses the same labeling rules as for fields. For example:

t-fmt3.p

```
DEFINE VARIABLE newrep AS CHARACTER FORMAT "x(3)".
DEFINE VARIABLE answer AS LOGICAL.

FOR EACH customer:
  DISPLAY name sales-rep.
  SET answer.
  IF answer THEN DO:
    SET newrep.
    sales-rep = newrep.
    DISPLAY sales-rep.
  END.
END.
```

When displaying the name and sales-rep fields, PROGRESS uses the labels specified in the Dictionary ("Name" and "Sls rep"). When displaying the newrep and answer variables, PROGRESS uses the names of those variables as the labels.

```
Name                 Sls rep  answer  newrep
Second Skin Scuba    SLS      ____
```

You override default labels for variables the same way you override default labels for fields, by using the LABEL option, as in t-fmt6.p:

t-fmt6.p

```
DEFINE VARIABLE newrep AS CHARACTER
  LABEL "New Sales Rep".
DEFINE VARIABLE answer AS LOGICAL.

FOR EACH customer WITH 1 DOWN:
  answer = NO.
  DISPLAY name sales-rep LABEL "Current Sales Rep".
  UPDATE answer
    LABEL "Do you want to name a new sales rep?"
    WITH FRAME a.
  IF answer THEN DO:
    SET newrep.
    sales-rep = newrep.
    DISPLAY sales-rep.
  END.
END.
```

In this example, PROGRESS assigns new labels to the newrep and answer variables ("New Sales Rep" and "Do you want to name a new sales rep?"). Notice that the new label for the newrep variable is specified in the DEFINE VARIABLE statement while the new label for the answer variable is specified in the UPDATE statement. You can specify a new label for a variable in either the DEFINE VARIABLE statement or directly in the data handling statement that displays that variable.

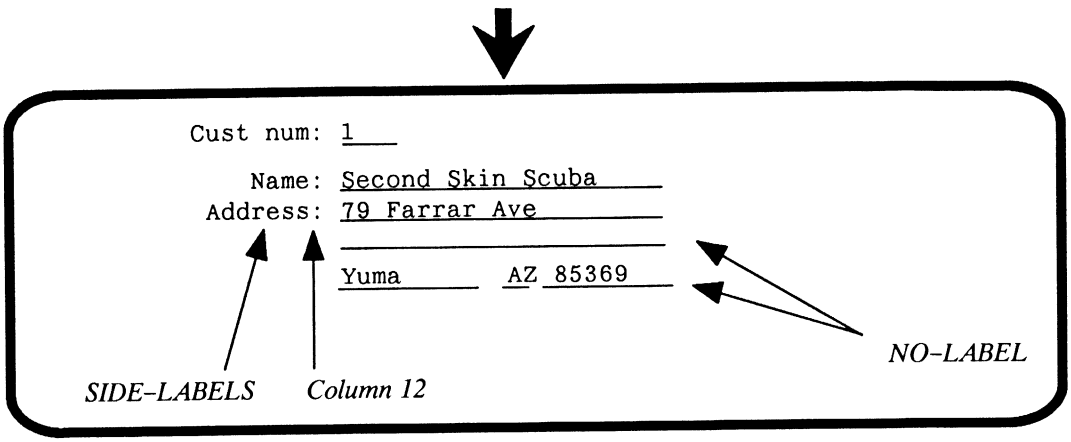
```
Name                    Current Sales Rep New Sales Rep
Second Skin Scuba       SLS

Do you want to name a new sales rep?
no
```

11.2.3 Describing Other Field and Variable Characteristics

There are many other options you can use to describe fields and variables. For example, the t-fdvr1.p procedure displays customer data in a format that is useful for entering data from forms or mailing labels.

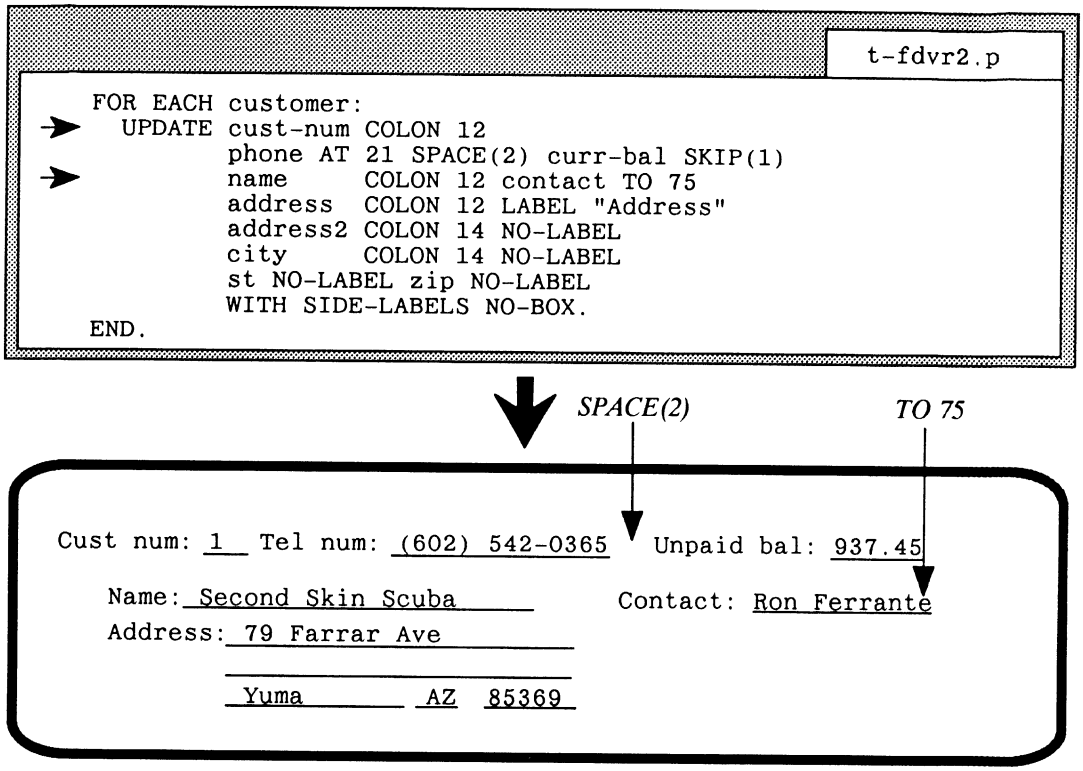
t-fdvr1.p

```
FOR EACH customer:
  UPDATE cust-num COLON 12 SKIP(1)
         name     COLON 12
         address  COLON 12 LABEL "Address"
         address2 COLON 14 NO-LABEL
         city     COLON 14 NO-LABEL
         st NO-LABEL zip NO-LABEL
         WITH SIDE-LABELS NO-BOX.
END.
```



Try removing the COLON options from the procedure to see how the display looks without them.

The t-fdvr2.p procedure shows how to put spaces between fields and make sure that a field ends in a certain column:



The SPACE option puts two spaces between the telephone number and the unpaid balance. The TO option ends the display of the contact field in column 75.

11.2.4 Displaying Multiple Fields in the Same Frame Field

Often you want to display one of several different values in the same frame field, depending on some condition. For example, the t-fdvr4.p procedure displays "Extend special offer" for those customers whose max-credit is greater than 500 and it displays "Recheck credit" for all other customers.

t-fdvr4.p

```
FOR EACH customer:
  IF max-credit > 500 THEN
    DISPLAY name "Extend special offer"
            WITH NO-LABEL NO-UNDERLINE.
  ELSE DISPLAY name "Recheck credit".
  DISPLAY SKIP(1).
END.
```

```
Second Skin Scuba     Extend special offer
Match Point Tennis    Extend special offer
Off The Wall          Extend special offer
Pedal Power Cycles                          Recheck credit
Flying Fat Aerobics   Extend special offer
                            .
                            .
                            .
```

The procedure displays information in three columns. When designing the frame for the procedure, PROGRESS starts at the top of the procedure to determine how much space to allocate for various fields. In this top-to-bottom pass, PROGRESS allocates space for:

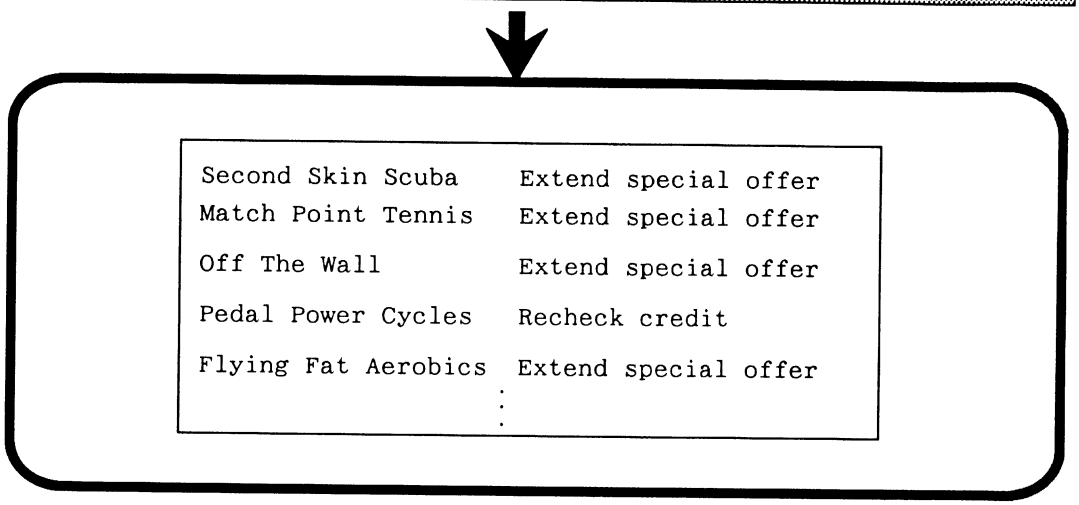
- The name field. (Both references to the name field are put in the same frame field.)
- The string "Extend special offer."
- The string "Recheck credit."

Although there are three fields in the frame, you probably want to display the "Extend special offer" and the "Recheck credit" strings in the same column.

Here is another version of the procedure:

t-fdvr5.p

```
FOR EACH customer:
  IF max-credit > 500 THEN
    DISPLAY name
            "Extend special offer" @ msg-area
              AS CHARACTER SKIP(1)
            WITH NO-LABEL NO-UNDERLINE.
  ELSE DISPLAY name "Recheck credit" @ msg-area.
END.
```



```
Second Skin Scuba     Extend special offer
Match Point Tennis    Extend special offer
Off The Wall          Extend special offer
Pedal Power Cycles    Recheck credit
Flying Fat Aerobics   Extend special offer
                      :
                      :
```

This procedure creates a variable as part of the DISPLAY statement and places it in the frame field called msg-area. Just as you would with the DEFINE VARIABLE statement, you must define the data type and format of the frame field. In this example, the frame field is a constant, so you do not need to define a display format. The data type is CHARACTER. The procedure displays the string "Extend special offer" in this new frame field. The second DISPLAY statement displays the string "Recheck credit" in the same frame field.

11.3 USING FRAMES FOR INPUT

In addition to the options you can use to tell PROGRESS how to display fields and variables, there are several options you can use to help the user input data. You can define a help message to display when the user is positioned on a field to update. You can validate a field using criteria other than the criteria defined in the Data Dictionary. You can simplify how the user enters data by moving the cursor automatically to the next field or by defining an array of fields which the user can edit like text in a word processor file.

11.3.1 Providing Help, Validation, and Automatic Return on Input

Here is an example that shows how to provide help information for a field, validate the data in a field, and automatically move to the next field when it is filled with data.

t-fdvr3.p

```
FOR EACH customer:
  DISPLAY name     COLON 6 sales-rep AT 50 SKIP
          address  COLON 6 max-credit AT 50 SKIP
          city     COLON 8 NO-LABEL
          st NO-LABEL zip NO-LABEL SKIP(2)
          WITH SIDE-LABELS.
  SET sales-rep AUTO-RETURN
                HELP "Enter one of SLS, BBB, or DKP"
      max-credit
        VALIDATE(max-credit > 100,
                 "Enter a value greater than 100").
END.
```

```
Name: Second Skin Scuba       Sls rep: SLS
Addr: 79 Farrar Ave           Max cred: 1,500
      Yuma        AZ 85369

Enter one of SLS, BBB, or DKP
```

When your cursor is on the sales-rep field, the HELP option displays the message "Enter one of SLS, BBB, or DKP." Even though the sales-rep field has a Dictionary-defined help message of "Enter initials for a sales rep," the message specified with the HELP option overrides the Dictionary message.

The sales-rep field is only three characters long in order to store a first, middle, and last initial for the sales rep. The AUTO-RETURN option is used on this field to make it easier to enter data. When you enter the third character into the sales-rep field, PROGRESS automatically moves the cursor to the max-credit field. You do not have to press RETURN.

Try entering a max-credit that is less than 100. The VALIDATE option displays the message "Enter a value greater than 100". The max-credit field has a Dictionary specified validate expression of "max-credit > = 0 and max-credit < = 9999999." The validation criteria specified in the VALIDATE option override the Dictionary validation expression.

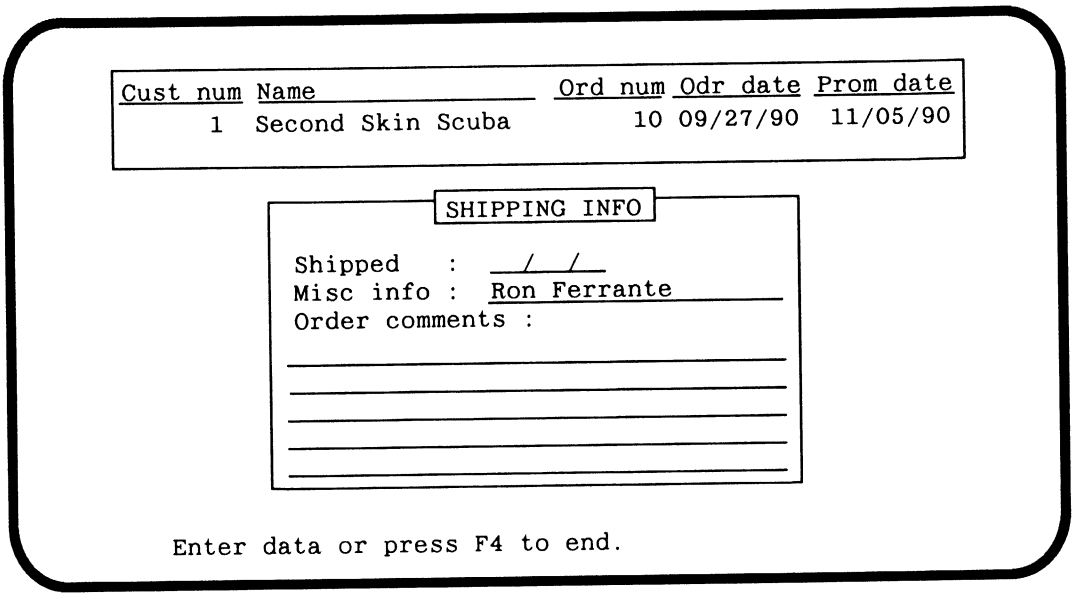
11.3.2 Describing Word Processing Style Data Entry

Suppose you want to be able to enter several lines of text in a field as though you are using a word processor. Run the t-text.p procedure to see how this is done.

t-text.p

```
DEFINE VARIABLE ship-comments AS CHARACTER FORMAT "x(40)" EXTENT 5.

FORM "Shipped    :" order.sdate AT 13 SKIP
     "Misc Info :" order.misc-info AT 13 SKIP(1)
     "Order Comments :" ship-comments AT 1
  WITH FRAME ord-comm CENTERED NO-LABELS TITLE " SHIPPING INFO ".

FOR EACH customer, EACH order OF customer:
  DISPLAY cust.cust-num cust.name order.order-num order.odate
          order.pdate WITH FRAME order-hdr CENTERED.
  UPDATE sdate misc-info TEXT(ship-comments) WITH FRAME ord-comm.
  ship-comments = "".
END.
```

Now type some text into the Order comments area. Notice that you can type continuously and the text wraps onto subsequent lines. Try deleting and adding text in the middle of text you typed. PROGRESS wraps data back to fill empty areas and pushes text forward to make space for text you insert. You can also use the APPEND-LINE, BREAK-LINE, CLEAR, DELETE-LINE, NEW-LINE, RECALL, RETURN, and TAB. These keys work the same as they do in the PROGRESS editor.



The t-text.p procedure defines an array, ship-comments, with an extent of 5 and a format allowing up to forty characters in each element. This array is the area for order comments. Look at the UPDATE statement in the procedure. The TEXT option is used to update the ship-comments array variable. Normally, you would assign the value of the variable to a field in the database, but we only want to show the word processing feature here.

You can use the TEXT option for character fields or variables (including array elements). The character fields must be in the format "x(*n*)". For more information, see the *PROGRESS Reference* manual under the TEXT option for the UPDATE, SET, and PROMPT-FOR statements.

11.4 DEFINING FRAME LAYOUT AND PROCESSING PROPERTIES

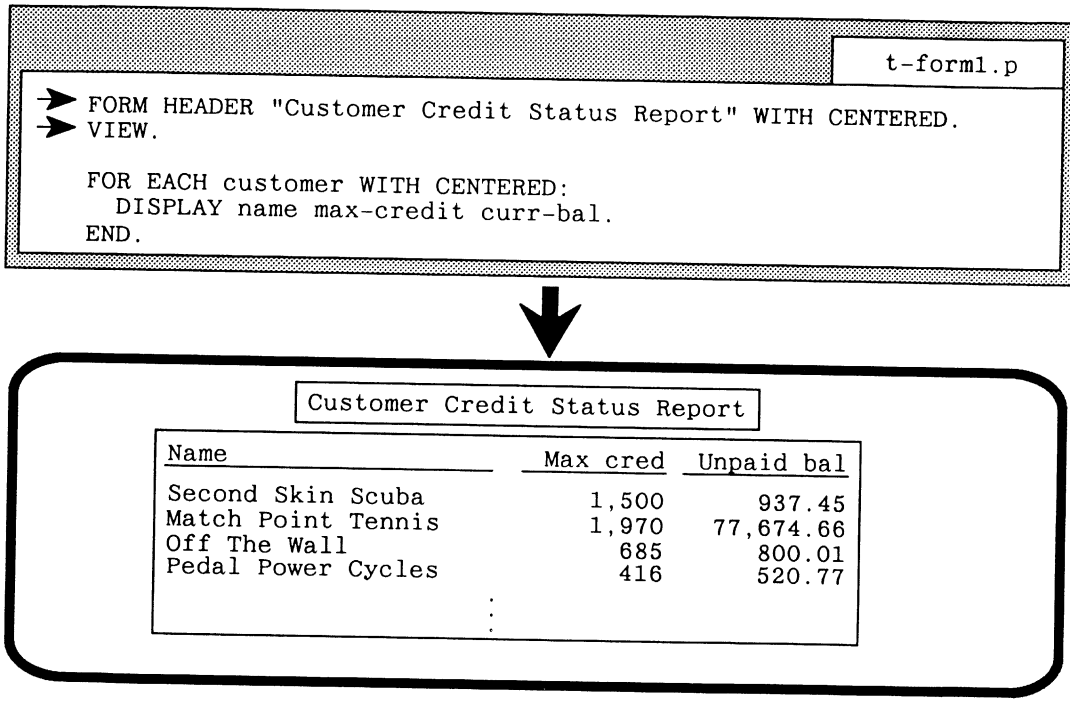
Often you want to describe characteristics of a frame but don't want to include that description in the Frame phrase of a data handling statement. This is especially true if the frame characteristics are very complex or if you want to use the same frame frequently in the same or in other procedures. You can use the FORM statement to describe the layout and processing properties of a certain frame. Specifically, you use the FORM statement to:

- Describe frame headers.
- Lay out frame fields in one order when you are going to process them in another order.
- Design a frame that doesn't necessarily match the flow of your procedure.

It is important to remember that the FORM statement describes only the layout of a frame. It does not actually bring that frame into view. To see the frame, you must either use a data handling statement that uses that frame or you must use the VIEW statement.

11.4.1 Describing Frame Headers

You can use the FORM statement to describe a header that appears at the top of a frame. For example, The FORM statement in this procedure defines a frame that consists of nothing but a header. The VIEW statement brings that frame into view.



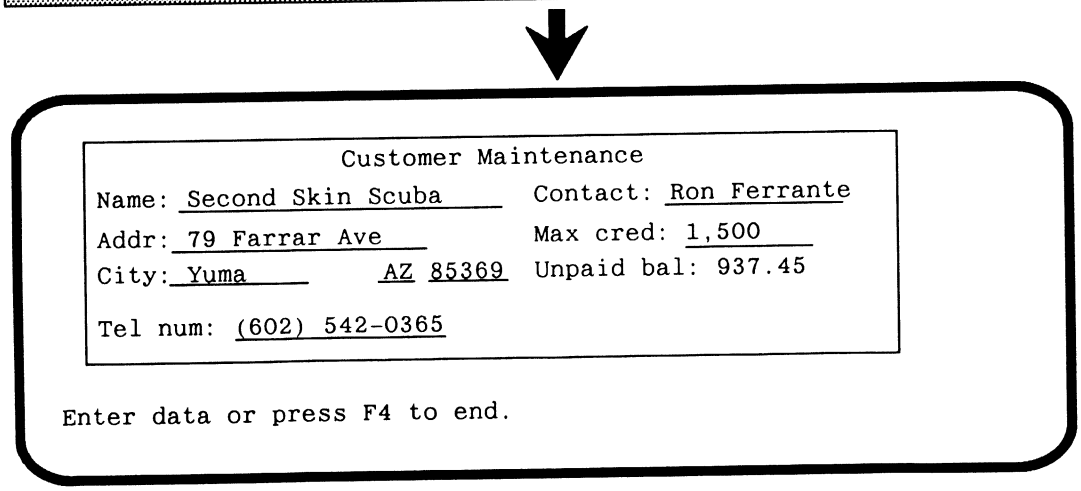
11.4.2 Laying Out Frame Fields

When you are using a frame that has a very complex layout or when you want to use the same frame layout many times in a single procedure, you can use the FORM statement to describe the frame. That way, you need only describe the frame once, instead of in each data handling statement that uses the frame.

In this example, the FORM statement describes the layout of frame a. Although the fields are displayed in the order described in the FORM statement, you update those fields in the order listed in the UPDATE statement. That is, the order in which fields are described in a FORM statement affects only the way those fields are positioned in the frame and not the order in which they are actually processed. When you run t-form2.p, press TAB to move through the fields. Notice that the cursor moves among the fields in the order they are listed in the UPDATE statement.

t-form2.p

```
FORM customer.name contact AT 40 SKIP
     customer.address max-credit AT 40 SKIP
     customer.city customer.st NO-LABEL
     customer.zip NO-LABEL curr-bal AT 40 SKIP(1)
     phone
     HEADER "Customer Maintenance" AT 25 SKIP(1)
     WITH SIDE-LABELS NO-UNDERLINE FRAME a.

FOR EACH customer WITH FRAME a:
  DISPLAY curr-bal.
  UPDATE name address city st zip phone contact
         max-credit.
END.
```



11.5 USING THE MESSAGE AREA

In addition to using the regular display area on your terminal screen, you can use the message area at the bottom of the screen.

The message area consists of three lines:

- The first two lines are for procedure-specific messages.
- The third line (the last line on the screen) is for PROGRESS system messages and help messages.

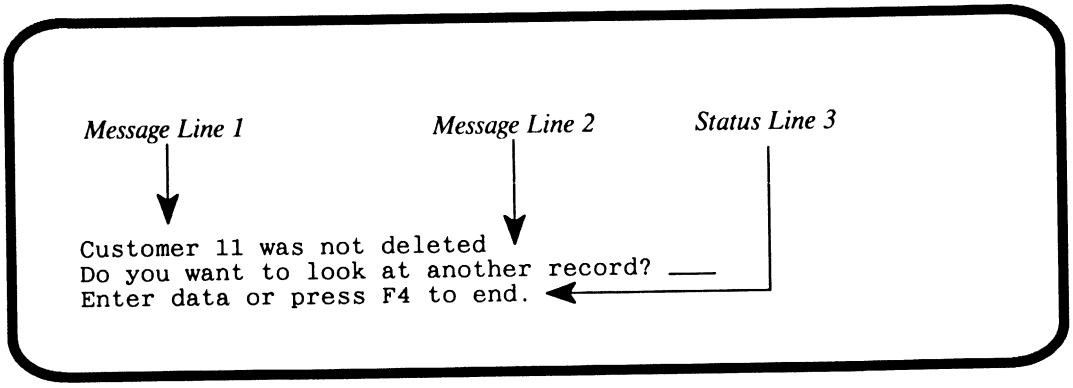
11.5.1 Displaying Procedure-Specific Messages

You use the MESSAGE statement to display messages in the message area. For example:

t-msg1.p

```
DEFINE VARIABLE answer AS LOGICAL.

block1:
REPEAT WITH 1 DOWN:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  DISPLAY name address city st zip.
  SET answer LABEL "Delete this customer record?"
             WITH FRAME a NO-HIDE.
  IF answer THEN DO:
    MESSAGE "Customer" cust-num "has been deleted".
    DELETE customer.
  END.
  ELSE MESSAGE "Customer" cust-num "was not deleted".
  MESSAGE "Do you want to look at another record?"
    SET answer.
  IF answer THEN NEXT block1.
  ELSE LEAVE block1.
END.
```

This procedure displays a different message depending on whether you choose to delete a customer record. After displaying a message indicating whether or not the record was deleted, the procedure uses the second line of the message area to ask if you want to look at another record.



11.5.2 Changing the PROGRESS System Message

You use the STATUS statement to specify the message that appears on the bottom line of the screen. There are three ways you can use the STATUS statement:

- STATUS DEFAULT expression

  Specifies a status message to be displayed when you are running a procedure but not entering any data. The default status message in this situation is all blanks.

- STATUS INPUT OFF

  Tells PROGRESS not to display any messages while you are entering data into an input field.

- STATUS INPUT expression

  Specifies a status message to be displayed when you are entering data into an input field. The default status message is "Enter data or press F4 to end." For specific fields, this message is replaced by any help message defined for the field in the Dictionary or any help message you specify with the HELP option.

For example, the t-msg2.p procedure uses the STATUS statement to override both the default status message displayed when you are just running the procedure and the default status message displayed when you are entering data.

t-msg2.p

```
DEFINE VARIABLE answer AS LOGICAL.

block1:
REPEAT WITH 1 DOWN:
  CLEAR FRAME a.
  STATUS INPUT "Customer file cleanup".
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  DISPLAY name address city st zip.
  STATUS DEFAULT
    "Information for customer number you entered".
  PAUSE 3 NO-MESSAGE.
  STATUS INPUT "Confirmation".
  SET answer LABEL "Delete this customer record?"
             WITH FRAME a NO-HIDE.
  IF answer THEN DO:
    MESSAGE "Customer" cust-num "has been deleted".
    DELETE customer.
  END.
  ELSE MESSAGE "Customer" cust-num "was not deleted".
  MESSAGE "Do you want to look at another record?"
    SET answer.
  IF answer THEN NEXT block1.
  ELSE LEAVE block1.
END.
```

When the PROMPT-FOR statement prompts you for a customer number, the default status message is "Enter data or press F4 to end." This procedure redefines that status message to be "Customer file cleanup." When the DISPLAY statement displays customer information, the status message line is all blanks. The STATUS DEFAULT statement resets that line to read "Information for customer number you entered." When the SET statement asks you for an answer to the question "Delete this customer record?" the default status message would be "Customer file cleanup" because the default set by the first STATUS statement stays in effect. The STATUS INPUT statement redefines that message to be "Confirmation."

NOTE: A NOTE ABOUT USING THE STATUS STATEMENT

Once you use either the STATUS DEFAULT, STATUS INPUT OFF, or STATUS INPUT statement during a PROGRESS session, that statement is in effect for all the procedures run in that session, unless you override it by using other STATUS statements in those procedures or until you return to the editor. STATUS DEFAULT (with no message) resets the default status message to blanks. STATUS INPUT (with no message) resets the input status message to "Enter data or press F4 to end."

11.6 UNDERSTANDING AND CHANGING FRAME BEHAVIOR

Just as PROGRESS uses defaults for designing frames, it uses defaults for viewing frames, advancing frames, and hiding frames. And, just as you can override frame design defaults, you can override these frame behavior defaults. For more information about frame behavior, see Chapter 7 of the *PROGRESS Programming Handbook*.

11.6.1 Bringing Frames into View

You bring a frame into view either by displaying data in that frame or by using the VIEW statement. For example:

t-form2.p

```
FORM customer.name contact AT 40 SKIP
     customer.address max-credit AT 40 SKIP
     customer.city customer.st NO-LABEL
     customer.zip NO-LABEL curr-bal AT 40 SKIP(1)
     phone
     HEADER "Customer Maintenance" AT 25 SKIP(1)
     WITH SIDE-LABELS NO-UNDERLINE FRAME a.

FOR EACH customer WITH FRAME a:
  DISPLAY curr-bal.
  UPDATE name address city st zip phone contact
         max-credit.
END.
```

Here, the FORM statement describes a frame named "a." PROGRESS brings that frame into view when the DISPLAY statement displays data in the frame. But if you want to bring the frame into view before the DISPLAY statement, you can use the VIEW statement.

In this example, the VIEW statement brings frame a into view:

t-form4.p

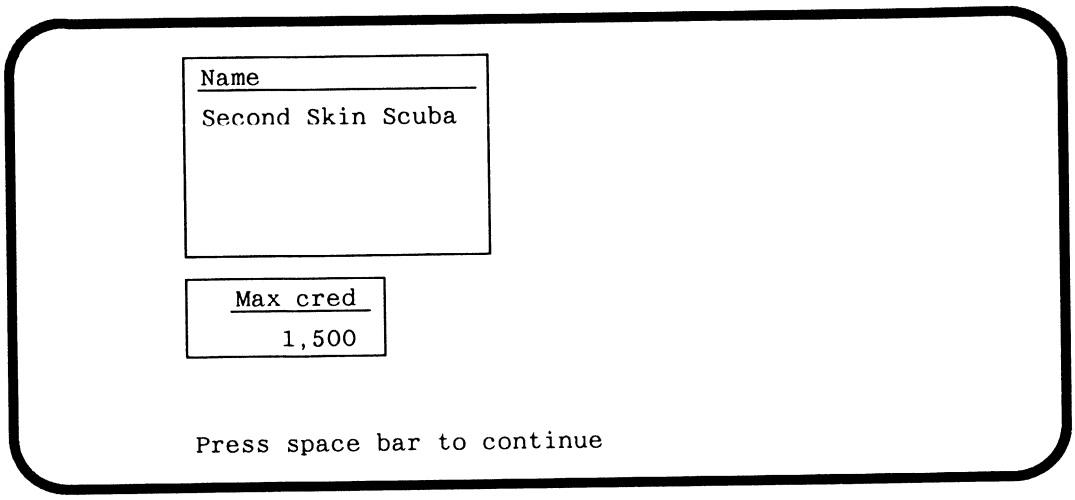
```
FORM customer.name contact AT 40 SKIP
     customer.address max-credit AT 40 SKIP
     customer.city customer.st NO-LABEL
     customer.zip NO-LABEL curr-bal AT 40 SKIP(1)
     phone
     HEADER "Customer Maintenance" AT 25 SKIP(1)
     WITH SIDE-LABELS NO-UNDERLINE FRAME a.

VIEW FRAME a.
MESSAGE "Before the DISPLAY statement...".
PAUSE.
FOR EACH customer WITH FRAME a:
  DISPLAY curr-bal.
  UPDATE name address city st zip phone contact
         max-credit.
END.
```

Throughout this manual, you have seen procedures where PROGRESS automatically removes frames from the screen. This is called "hiding." For example:

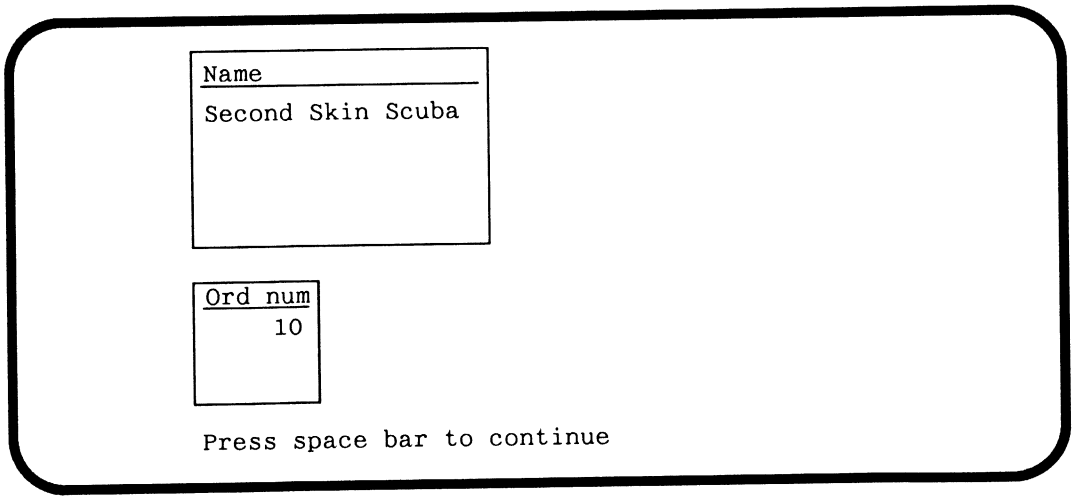
t-hide.p

```
FOR EACH customer WHERE cust-num < 5:
  DISPLAY name SKIP(8) WITH FRAME f1.
  DISPLAY max-credit WITH FRAME f2.
  FOR EACH order OF customer WITH FRAME f3:
    DISPLAY order-num.
  END.
END.
```

Go ahead and run this procedure. PROGRESS displays the following screen:



At this point, the procedure has displayed the customer name and the max-credit. But there is no more space on the screen in which to display order numbers. Press the space bar.



You can see that PROGRESS automatically hides frame f2 to make room for frame f3.

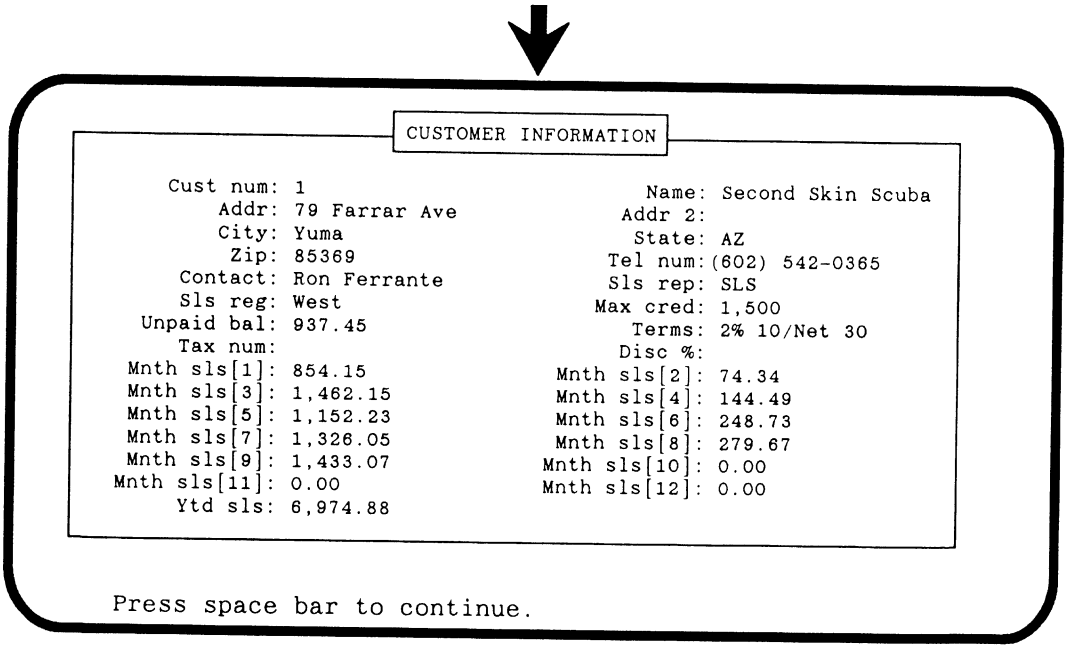
When it runs out of space, PROGRESS starts from the bottom of the screen and erases existing frames. If the frame PROGRESS is about to hide has not yet been seen, PROGRESS automatically pauses (as it did in the previous example when you were asked to press the space bar to continue after the max-credit was displayed). If you do not want PROGRESS to automatically hide frames, you can overlay frames.

11.6.2 Using Overlay Frames

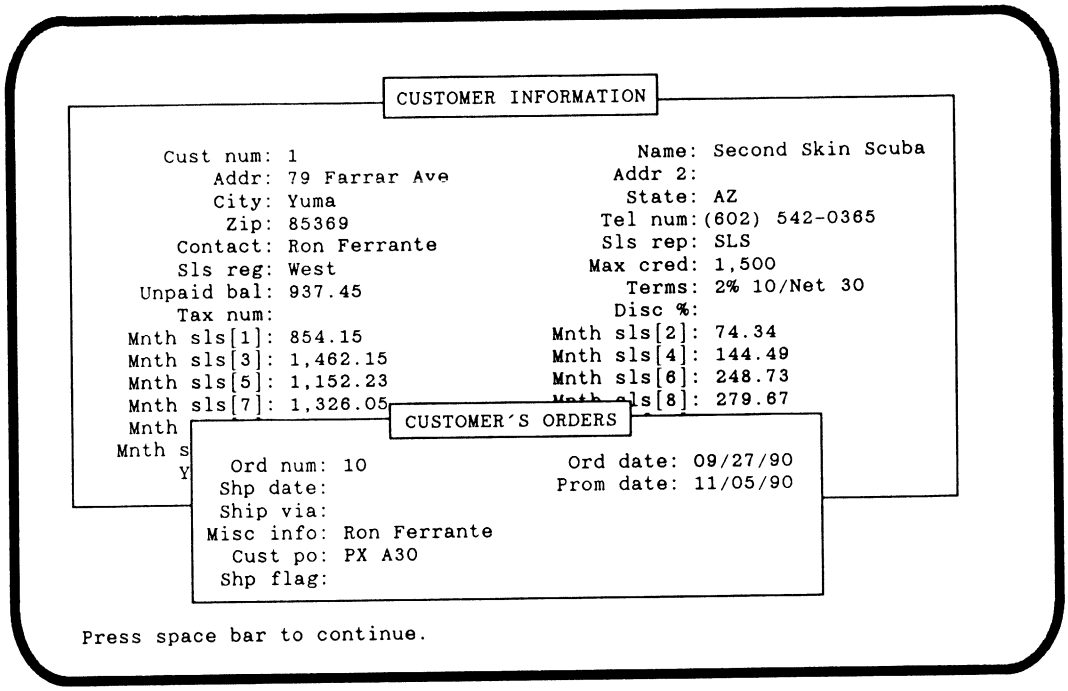
Suppose you want to display two frames but there is not enough room for both. PROGRESS allows you to create a frame that overlays another frame. For example, suppose you want to display all the information for a customer, and then see that customer's orders, one at a time, while keeping the customer information in view. Here is a procedure to do this.

t-ovrlay.p

```
FOR EACH customer:
  DISPLAY customer WITH 2 COLUMNS TITLE "CUSTOMER INFORMATION".
  FOR EACH order OF customer:
    DISPLAY order-num odate sdate pdate shp-via misc-info
            cust-po shipped WITH 2 COLUMNS 1 DOWN OVERLAY
            TITLE "CUSTOMER'S ORDERS" ROW 14 COLUMN 10.
  END.
END.
```



When you press the space bar, if the customer has orders, you see the customer's first order in a frame that overlays the first frame on your screen.



When you press the space bar, you see the next order for the customer, the order frame is cleared and the next customer is displayed. PROGRESS automatically refreshes a covered frame when an overlay frame clears.

Look back at the t-ovrlay.p procedure. PROGRESS displays the frame with the order information over the frame with the customer information because the frame-phrase for the order frame includes the OVERLAY option. You can use the ROW and COLUMN options to control where the overlay frame is placed on the screen. If you do not include the ROW and COLUMN options, the overlay frame is displayed with the upper left corner in row 1, column 1.

See the *PROGRESS Reference* manual for more information about the OVERLAY option on the frame-phrase.

11.6.3 Using FRAME-ROW and FRAME-COL to Control the Position of Frames

When you design an application, you can control where a frame appears on the screen and where one frame overlays another. PROGRESS has several functions that let you control the position of frames in relation to the screen, and in relation to other frames.

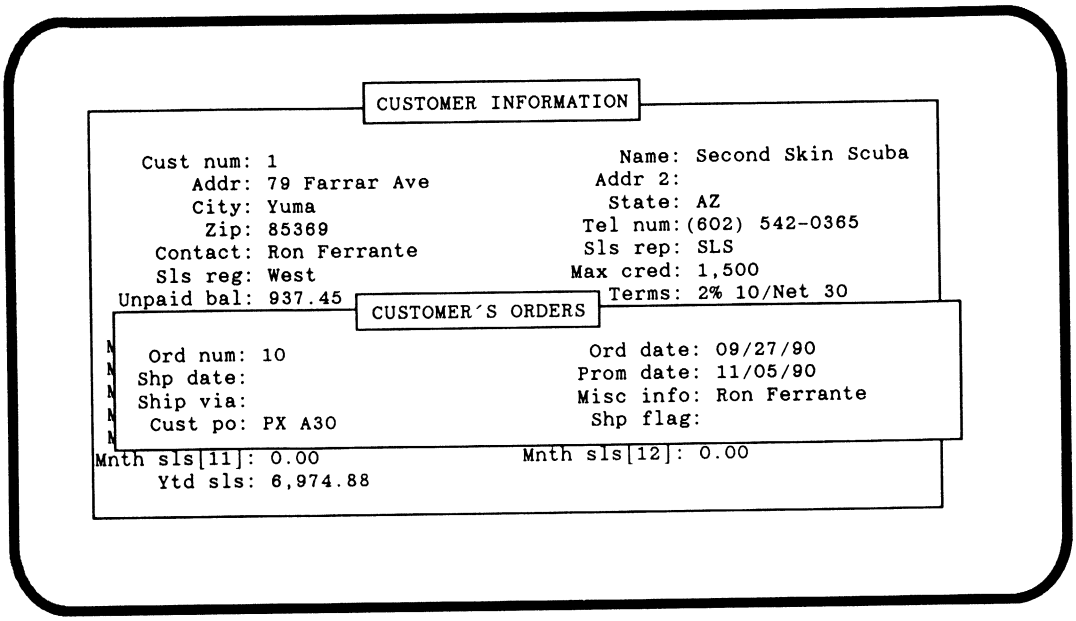
In the t-ovrlay.p procedure, we used the ROW and COLUMN options on the frame-phrase to control the position of the overlay frame. Suppose you want to move the customer information frame and you want the order frame to overlap it at the same place no matter where you move it. We can change the t-ovrlalay.p procedure so that the order information frame overlays the customer information frame at the same position regardless of where the customer information frame is on the screen.

t-frrow.p

```
FOR EACH customer:
  DISPLAY customer WITH FRAME cust-frame
          2 COLUMNS TITLE "CUSTOMER INFORMATION".
  FOR EACH order OF customer:
      DISPLAY order-num odate sdate pdate shp-via misc-info
           cust-po shipped WITH 2 COLUMNS 1 DOWN OVERLAY
              TITLE "CUSTOMER'S ORDERS"
              ROW FRAME-ROW(cust-frame) + 8
              COLUMN FRAME-COLUMN(cust-frame) + 1.
  END.
END.
```

In this modified procedure, the FRAME-ROW and FRAME-COL functions indicate the location of the order information frame on the customer information frame.

The FRAME-ROW and FRAME-COL functions each return an integer value that represents the row or column position of the upper left corner of the named frame. The FRAME-ROW function returns the value of the row position of the uppermost corner of the cust-frame. Then the procedure adds 8 to that value and displays the overlay frame at that row position. PROGRESS figures the column position the same way. If you run the t-frrow.p procedure, you see the following screen.



Now, suppose we display the cust-frame at the third row on the screen. Because the FRAME-ROW and FRAME-COL functions place the order information frame in relation to the customer information frame, the order information frame is still displayed at the same position on the customer information frame; below the eighth row of the cust-frame and next to the first column of the cust-frame. Run the t-frrow1.p procedure to see this behavior.

t-frrow1.p

```
FOR EACH customer:
  DISPLAY customer WITH FRAME cust-frame ROW 3
          2 COLUMNS TITLE "CUSTOMER INFORMATION".
  FOR EACH order OF customer:
      DISPLAY order-num odate sdate pdate shp-via misc-info
                cust-po shipped WITH 2 COLUMNS 1 DOWN OVERLAY
              TITLE "CUSTOMER'S ORDERS"
              ROW FRAME-ROW(cust-frame) + 8
              COLUMN FRAME-COLUMN(cust-frame) + 1.
  END.
END.
```

See the *PROGRESS Reference* manual for more information on the FRAME-ROW and FRAME-COL functions.

11.6.4 Using FRAME-LINE to Control the Position of Frames

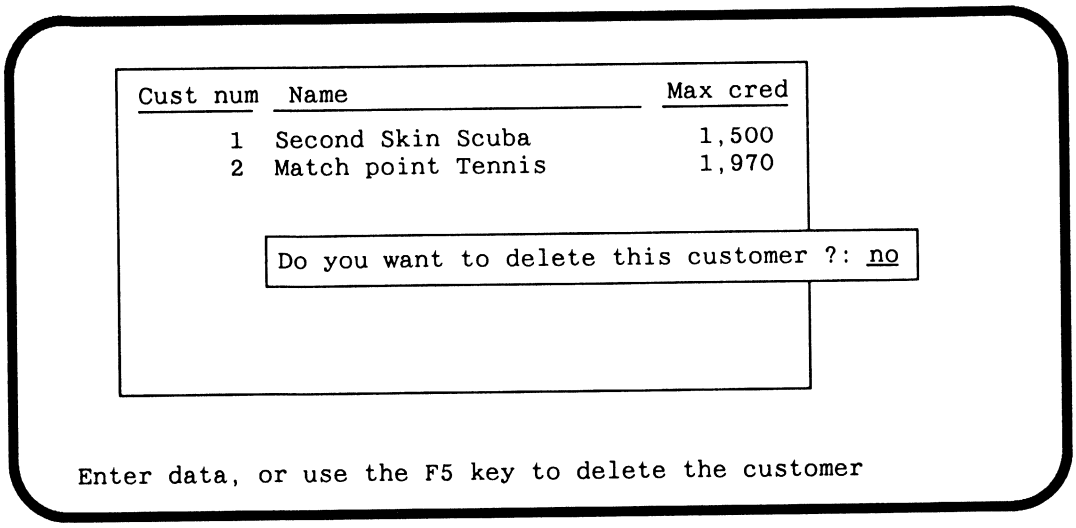
The FRAME-LINE function gives you information about the position of the current display line in a frame. You can use this information to determine how to display other information in the frame. For example, look at the following procedure.

t-frline.p

```
DEFINE VARIABLE ans AS LOGICAL LABEL
       "Do you want to delete this customer?".

STATUS INPUT "Enter data, or use the "
             + KBLABEL("get")
             + " key to delete the customer.".

get-cust:
FOR EACH customer WITH 10 DOWN:
  UPDATE cust-num name max-credit
  editing:
    READKEY.
    IF KEYFUNCTION(LASTKEY) = "get"
    THEN DO:
      UPDATE ans WITH ROW FRAME-ROW + 3 +
                          FRAME-LINE + 5 COLUMN 10
             SIDE-LABELS OVERLAY FRAME del-frame.
      IF ans
      THEN DO:
        DELETE customer.
        NEXT get-cust.
      END.
    END.
    APPLY LASTKEY.
  END.
END.
```

The t-frline.p procedure lets you update a customer's number, name, and max credit. The procedure displays information in the frame for one customer at a time. You can press the GET key (F5) at any time to see a message and a prompt that asks if you want to delete the customer. The message always appears in its own frame, five lines below the last customer displayed.

When you run the procedure, you see this screen:



Look back at the second UPDATE statement in the t-frline.p procedure. The FRAME-LINE function controls where the ans variable is displayed. The position of the prompt is calculated from the upper right corner of the frame and the current line within the frame. That is, FRAME-ROW + 3 + FRAME-LINE gives the position of the current line in the frame, taking into account the 3 lines for the frame box and the labels. The prompt is placed 5 lines below the current line.

See the *PROGRESS Reference* manual for more information on the FRAME-LINE function.

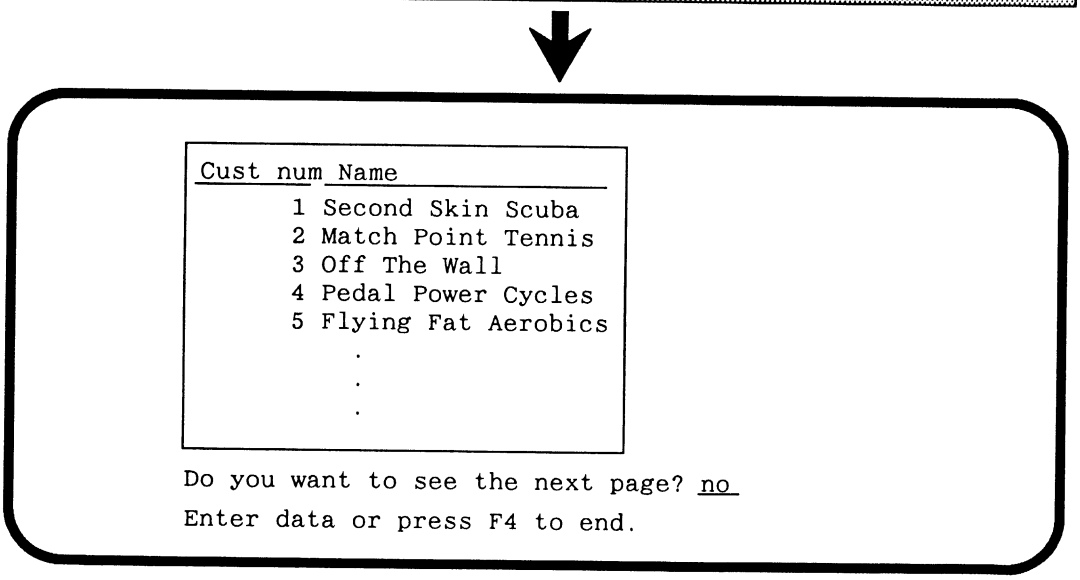
11.6.5 Using FRAME-DOWN to Position Frames

The FRAME-DOWN function returns the number of iterations that can fit in a frame. You can use this function to determine if a down frame is full. Then PROGRESS can perform an action based on the value of FRAME-DOWN.

For example, suppose you want to display all the customers in the database. This can be a long list that takes several screens. If the customer you want to see is on the first screen, it is time consuming to press RETURN until every customer in the database is listed and the procedure ends. The following procedure uses the FRAME-DOWN function to solve this problem.

t-frdown.p

```
DEFINE VARIABLE ans AS LOGICAL.

FOR EACH customer:
  DISPLAY cust-num name.
  IF FRAME-LINE = FRAME-DOWN
  THEN DO:
    MESSAGE "Do you want to see the next page?"
    UPDATE ans.
    IF ans
    THEN NEXT.
    ELSE LEAVE.
  END.
END.
```



Each full screen includes the question "Do you want to see the next page?" If you press no, the procedure ends.

Look at the IF...THEN...ELSE statement in the t-frdown.p procedure. Remember that the FRAME-DOWN function returns the number of iterations that can fit in a frame and the FRAME-LINE function returns the current logical line in a frame. So when the current logical line equals the number of lines in the frame, the procedure displays the message "Do you want to see the next page?" and gives the user the option to continue or to end the procedure.

See the *PROGRESS Reference* manual for more information on the FRAME-DOWN function.

11.7 FRAME AND TERMINAL RELATIONSHIPS

There are two ways a terminal can handle screen formatting. It can either:

- Reserve a character position on both sides of every field for special screen field attributes, such as underlining or highlighting. These terminals are called "spacetaking" terminals.
- Not reserve a character position for special field attributes. These are called "nonspacetaking" terminals.

All PC terminals are nonspacetaking terminals. However, this aspect of screen formatting is still relevant if you are planning to develop applications that will run on any PROGRESS supported operating system.

For more information on designing screens for spacetaking and nonspacetaking terminals, see Chapter 7 of the *PROGRESS Programming Handbook.*

11.8 USING COLOR IN SCREEN DISPLAYS

One way you can enhance the appearance of your application is to use different colors or other video attributes such as reverse video or highlighting. You can use different colors when displaying data, messages, or frames, or when you are getting input from the user.

PROGRESS uses three different colors or attributes by default. These are assigned the following standard names, which correspond to different video attributes depending on the terminal being used:

- NORMAL

  Used to display fields and labels.

- INPUT

  Used to indicate input areas.

- MESSAGES

  Used to display information in the message area.

Table 11–4 describes the default colors for each of these modes. You can modify the defaults for DOS and OS/2 by using the -v startup option (see Chapter 3 of the *PROGRESS System Administration Guide II: General* for details).

Table 11–4: NORMAL, INPUT, And MESSAGES Colors

Monitor	NORMAL	INPUT	MESSAGES
DOS or OS/2 color monitor	Blue background and white foreground.	Light gray background and blue foreground.	Light gray background and blue foreground.
DOS or OS/2 monochrome monitor	Standard background and foreground depending on monitor.	Underlined	Reverse video
UNIX, BTOS/CTOS, and VMS	Whatever your terminal defines as normal mode.	Depends on the terminal type and how INPUT is defined in the protermcap file; usually underlining.	Depends on the terminal type and how MESSAGES is defined in the protermcap file; usually reverse video.

You use the COLOR statement to specify different colors you want to use while displaying data or getting input from the user, as shown in the t-color1.p procedure.

The COLOR statement indicates that the curr-bal field should be displayed in MESSAGES mode when the current balance is greater than the maximum credit. The actual display of the curr-bal is caused by the DISPLAY statement. The COLOR statement simply indicates what color to use when the data is actually displayed.

t-color1.p

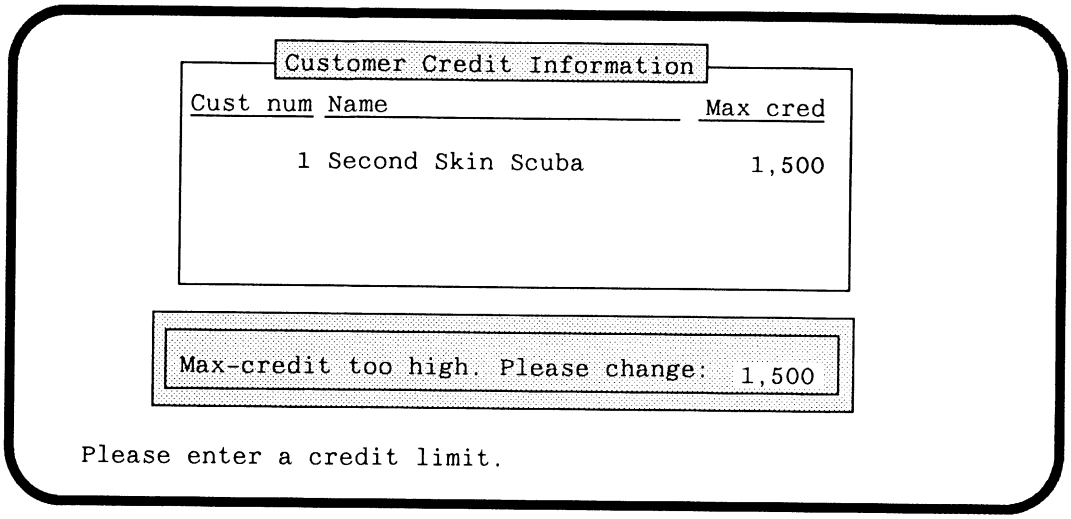
```
FOR EACH customer WITH FRAME a:
  FORM customer.name contact AT 40 SKIP
       customer.address max-credit AT 40 SKIP
       customer.city customer.st NO-LABEL
       customer.zip NO-LABEL curr-bal AT 40 SKIP(1)
       phone
       HEADER "Customer Maintenance" AT 25 SKIP(1)
       WITH SIDE-LABELS NO-UNDERLINE 1 DOWN FRAME a.
  IF curr-bal > max-credit THEN DO:
    COLOR DISPLAY MESSAGES curr-bal.
    MESSAGE
      "Current balance greater than maximum credit".
  END.
  DISPLAY curr-bal.
  UPDATE name address city st zip phone contact
         max-credit.
END.
```

If you have a non-spacetaking terminal, you can change the color of the frames on your screen with the COLOR option on the frame-phrase. For example, look at the t-bkcol.p procedure.

t-bkcol.p

```
FOR EACH customer:
  DISPLAY cust-num name max-credit WITH 10 DOWN FRAME cust
          CENTERED TITLE "Customer Credit Information".
  IF max-credit >= 1500
  THEN DO:
    UPDATE max-credit LABEL "Max-credit too high. Please change"
           WITH SIDE-LABELS FRAME x
           COLOR MESSAGES PROMPT NORMAL ROW 17 CENTERED.
    DISPLAY max-credit WITH FRAME cust.
  END.
END.
```

When you run this procedure you see the following screen.



In this procedure, PROGRESS displays customers and their maximum credit. If the credit is above $1,500, PROGRESS displays frame x with the maximum credit message and a prompt for the user to change the maximum credit. PROGRESS displays the entire frame in the MESSAGES color.

Look back at the t-bkcol.p procedure. In the UPDATE statement, the frame phrase for frame x includes the option COLOR MESSAGES. This option tells PROGRESS to display frame x in the MESSAGES color. The PROMPT NORMAL option tells PROGRESS to display the prompt in the NORMAL color.

For more information on using different colors in your procedures, see the COLOR option on the Frame phrase and the COLOR statement in the *PROGRESS Reference* manual.

11.9 SUMMARY

Chapters 1 through 10 explained the basics of writing PROGRESS procedures. This chapter showed you how to design the output of your procedures. Specifically, it explained:

- The definition of a frame.
- How to change default frame characteristics.
- The differences between single frames and down frames.
- How to change the display characteristics of fields and variables.
- How to use the message area at the bottom of the terminal screen.
- The behavior of frames and how to change that behavior.
- How to use color in your screen displays.

Chapter 12
Writing Reports

The goal of any database management system is to provide information that helps users run an organization more effectively. A **report**, whether printed on paper, viewed on a screen, or stored on other media is the most common tool for presenting information.

Using the frame formatting techniques described in Chapter 11, this chapter shows you how to create reports ranging from the simple to the highly detailed. You'll also learn how to direct reports to alternate output destinations, such as a printer. Chapter 12 covers the following topics:

- Generating a simple report.
- Generating categorized and sorted reports.
- Using data from multiple files in a report.
- Overriding default layouts.
- Sending reports to different devices.
- Using running headers and footers in a report.

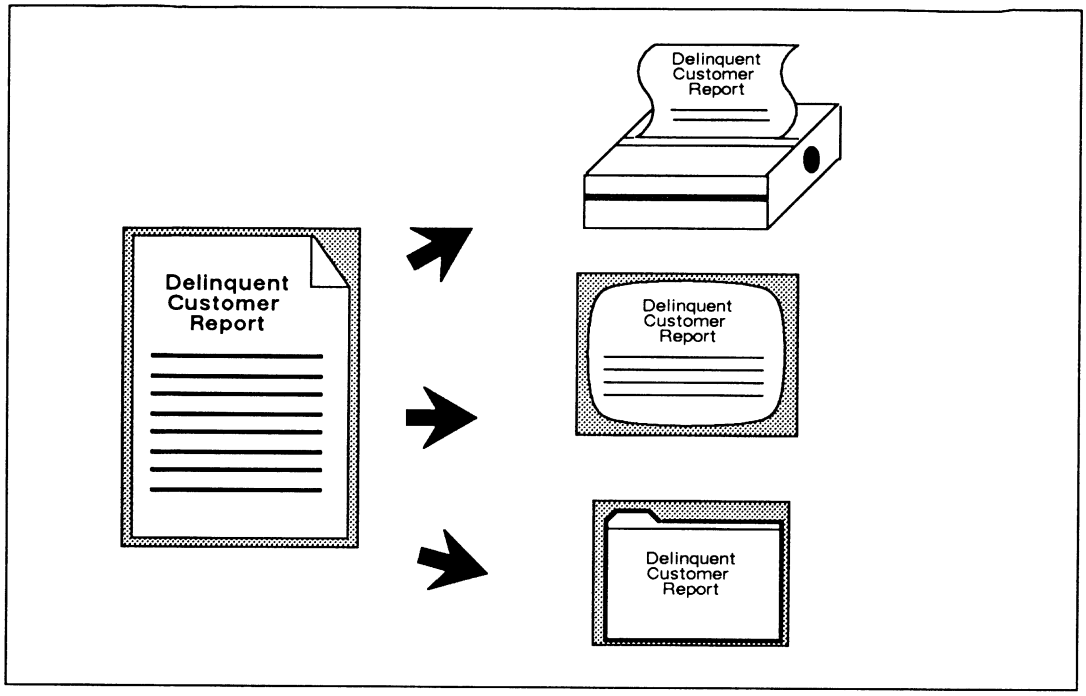


Figure 12–1: Output Destinations

12.1 GENERATING A SIMPLE REPORT

You can view a report on the screen simply by using the DISPLAY statement. Here, PROGRESS displays the name, telephone number, unpaid balance, and sales rep for all customers with balances greater than $1,400:

t-rept1.p

```
FOR EACH customer WHERE curr-bal >= 1400:
  DISPLAY name phone curr-bal sales-rep.
END.
```

Since there are no formatting statements in the t-rept1.p procedure, PROGRESS uses a default display format. This default display format includes a label for each field and a box to enclose the display.

```
Name                    Tel num          Unpaid bal   Sls rep
Match Point Tennis      (817) 498-2801   77,674.66    DKP
Lift Line Skiing        (617) 250-0087    1,481.00    BBB
Spike's Volleyball      (702) 272-9264    1,696.00    SLS
First Down Football     (603) 499-2544    2,011.00    BBB
Hard Knock's Skating    (307) 311-9788    3,080.00    SLS
Pocket Billiards Co.    (818) 666-4063    2,764.00    SLS
Sub Par Golf            (303) 444-6387    1,493.00    SLS
On Target Rifles        (601) 625-0070    2,565.00    DKP
Chip's Poker            (904) 567-8223    4,542.00    BBB

Procedure complete.  Press space bar to continue.
```

This type of report is rather generic; it is simply a list of all delinquent customers, organized in no specified order. A more practical approach might be to group the customers according to the sales rep who services each customer, and perhaps to display the unpaid balance totals for each sales rep.

12.2 GENERATING CATEGORIZED AND SORTED REPORTS

A report that shows totals for subgroups or categories in a file is called a **control break** report. For example, each sales rep is a subgroup. We can easily expand and modify the t-rept1.p procedure to produce a control break, or categorized report. The first step is to make some changes to the FOR EACH block header:

```
FOR EACH customer WHERE curr-bal >= 1400
    BREAK BY sales-rep:
```

Here, PROGRESS groups the customer records BY sales representative, using the BREAK option. Each customer record that belongs to a particular sales rep is a member of that sales rep's **break group.**

In addition to grouping the customer records by sales rep, our second objective is to produce a total of the unpaid balances for each sales rep's delinquent customers. We do this by modifying the DISPLAY statement:

```
DISPLAY name phone curr-bal
    (TOTAL BY sales-rep) sales-rep WITH NO-BOX.
```

The break group calculation is performed by the TOTAL BY option. TOTAL BY is an **aggregate-phrase** option. An aggregate-phrase identifies one or more values to be calculated based on a change in a break group. That is, each time the break group (sales-rep) changes, the TOTAL BY option calculates the total unpaid balances for all customers belonging to the last sales rep. It also provides a grand total of the unpaid balances for all delinquent customer accounts for all sales-reps at the end of the report.

Here is the modified t-rept1.p (now called t-rept2.p) and its output:

t-rept2.p

```
FOR EACH customer WHERE curr-bal >= 1400
    BREAK BY sales-rep:
  DISPLAY name phone curr-bal
      (TOTAL BY sales-rep) sales-rep WITH NO-BOX.
END.
```

```
Name                 Tel num          Unpaid bal  Sls rep
Lift Line Skiing     (617) 450-0087    1,481.00   BBB
First Down Football  (603) 499-2544    2,011.00   BBB
Chip's Poker         (904) 567-8223    4,542.00   BBB
                                       8.034.00      TOTAL

Match Point Tennis   (817) 498-2801   77,674.66   DKP
On Target Rifles     (601) 625-0070    2.565.00   DKP
                                      80,239.66      TOTAL

Spike's Volleyball   (702) 272-9264    1,696.00   SLS
Hard Knocks Skating  (307) 311-9788    3,080.00   SLS
Pocket Billiards Co. (818) 666-4063    2,764.00   SLS
Sub Par Golf         (303) 444-6387    1,493.00   SLS
                                       9,033.00      TOTAL

                                      97,306.66      TOTAL


 Press space bar to continue.
```

The procedure produces a report that lists the name, telephone number, and unpaid balance of all customers whose curr-bal field is over $1,400, grouped by sales-rep. The procedure produces a subtotal for each sales-rep and a grand total for all the customers in the report.

Although the t-rept2.p procedure sorts the customers in order by sales-rep, it is also possible to sort on other levels. Here is another version of the procedure that sorts both by sales-rep and by curr-bal:

t-rept2a.p

```
FOR EACH customer WHERE curr-bal >= 1400
    BREAK BY sales-rep BY curr-bal DESCENDING:
  DISPLAY name phone curr-bal
      (TOTAL BY sales-rep) sales-rep WITH NO-BOX.
END.
```

```
Name                 Tel num           Unpaid bal  Sls rep
Chip's Poker         (904) 567-8223      4,542.00  BBB
First Down Football  (603) 499-2544      2,011.00  BBB
Lift Line Skiing     (617) 450-0087      1,481.00  BBB
                                         8.034.00     TOTAL

Match Point Tennis   (817) 498-2801     77,674.66  DKP
On Target Rifles     (601) 625-0070      2.565.00  DKP
                                        80,239.66     TOTAL

Hard Knocks Skating  (307) 311-9788      3,080.00  SLS
Pocket Billiards Co. (818) 666-4063      2,764.00  SLS
Spike's Volleyball   (702) 272-9264      1,696.00  SLS
Sub Par Golf         (303) 444-6387      1,493.00  SLS
                                         9,033.00     TOTAL

                                        97,306.66     TOTAL

 Press space bar to continue.
```

The t-rept2a.p procedure produces a report that not only groups the customers by sales-rep but also lists the customers in descending order by curr-bal within each sales-rep break group.

12.3 USING DATA FROM MULTIPLE FILES IN A REPORT

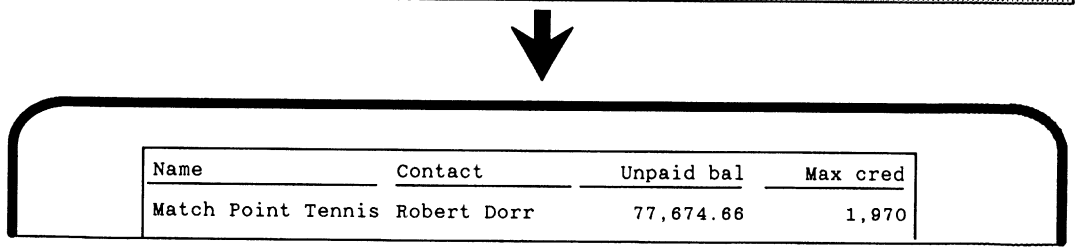
The reports you have seen so far use a single file, the customer file. To make reports more descriptive, you often need information from several files.

For example, if you are generating an analysis of delinquent customer accounts, it would be helpful to look at information about a customer's credit limit, the number and kinds of orders placed to date, and the items ordered. This kind of report involves four files — the customer, order, order-line, and item files.

12.3.1 Reporting Customer File Information

We'll start by working with the customer file. Suppose you want your report to show each delinquent customer's maximum credit limit, as well as that customer's unpaid balance. In addition, you want to list the name of the person to contact should you decide to notify the customer of the delinquency. Your procedure begins with a FOR EACH block to find the delinquent customers and display information from the customer record. The CENTERED option centers the frame on the screen.

```
FOR EACH customer WHERE curr-bal >= 1400:
  DISPLAY name contact curr-bal max-credit
          WITH CENTERED.
```

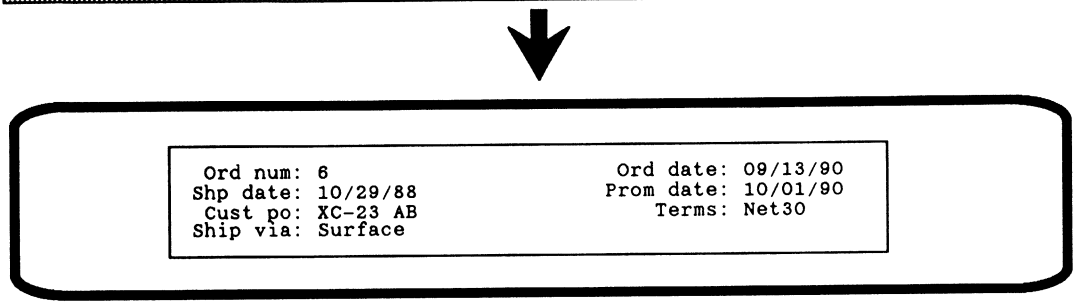


Name	Contact	Unpaid bal	Max cred
Match Point Tennis	Robert Dorr	77,674.66	1,970

12.3.2 Reporting Order File Information

Next, we'll provide some order information. Here, the DISPLAY statement is modified by the option WITH 2 COLUMNS CENTERED. It tells PROGRESS to format the specified fields in 2 columns.

```
FOR EACH order OF customer:
  DISPLAY order-num odate sdate pdate cust-po
          terms shp-via
          WITH 2 COLUMNS CENTERED.
```



```
 Ord num: 6                          Ord date: 09/13/90
Shp date: 10/29/88                  Prom date: 10/01/90
 Cust po: XC-23 AB                      Terms: Net30
Ship via: Surface
```

12.3.3 Reporting the Order-line and Item File Information

To get information about items ordered by a customer, the procedure must direct PROGRESS to look at each of the order-lines on each of the customer's orders. After finding out which item is on each order-line, it uses the item file to obtain more information about each item:

```
FOR EACH order-line OF order:
  FIND item OF order-line.
  DISPLAY qty order-line.item-num idesc
          disc price LABEL "Unit Price"
          WITH NO-BOX CENTERED ROW 13.
```

PROGRESS displays the quantity ordered, the item's stock number and description, any applicable discount, and the item's unit price, centered at row 13 without a box.

```
 Qty  Item num  Desc            Disc%  Unit Price
  34    00002  Tennis Racquet      0       64.50
  37    00013  Squash Glove        0       10.99
  10    00003  Sweat Band          0        2.55
```

12.3.4 Displaying the Result of a Calculation

On a report like the one we've been looking at, you often want to know the "extended" price; that is, the price for the total quantity of a particular item ordered. To obtain this price for the items in our report, we can perform a calculation using three fields from the item file. The procedure simply multiplies the item quantity by the item price, and applies a discount, if any. Let's incorporate this formula into the DISPLAY statement:

```
DISPLAY qty order-line.item-num idesc
        disc price LABEL "Unit Price"
        price * qty * (1 - disc / 100) (TOTAL)
        LABEL "Price" FORMAT "$zz,zzz,zz9.99 CR"
        WITH NO-BOX CENTERED ROW 13.
```

Extended price formula

Display Calculation

```
 Qty  Item num  Desc            Disc%  Unit Price       Price
  34    00002  Tennis Racquet      0       64.50 $  2,193.00
  37    00013  Squash Glove        0       10.99 $    406.63
  10    00003  Sweat Band          0        2.55 $     25.50
                                                  $  2,625.13 TOTAL
```

In a DISPLAY statement, when you do a calculation involving one or more fields, PROGRESS reserves space in the frame to hold the result. Simply follow the calculation with the characteristics you would normally apply to any field or variable being displayed.

Now, let's assemble all of these components into one procedure called t-rept3.p:

t-rept3.p

```
FOR EACH customer WHERE curr-bal >= 1400:
  DISPLAY name contact curr-bal max-credit
          WITH CENTERED.
  FOR EACH order OF customer:
    DISPLAY order-num odate sdate pdate cust-po
            terms shp-via
            WITH 2 COLUMNS CENTERED.
    FOR EACH order-line OF order:
       FIND item OF order-line.
       DISPLAY qty order-line.item-num idesc
               disc price LABEL "Unit Price"
               price * qty * (1 - disc / 100) (TOTAL)
               LABEL "Price" FORMAT "$z,zzz,zz9.99 CR"
               WITH NO-BOX CENTERED ROW 13
                    NO-ATTR-SPACE.
    END.
  END.
END.
```

Name	Contact	Unpaid bal	Max cred
Match Point Tennis	Robert Dorr	77,674.66	1,970

```
 Ord num: 6                          Ord date: 09/13/90
Shp date: 10/28/90                  Prom date: 10/01/90
 Cust po: XC-23 AB                     Terms: Net30
Ship via: Surface
```

Qty	Item num	Desc	Disc%	Unit Price	Price
34	00002	Tennis Racquet	0	64.50	$ 2,193.00
37	00013	Squash Glove	0	10.99	$ 406.63
10	00003	Sweat Band	0	2.55	$ 25.50
					$ 2,625.13 TOTAL

Press space bar to continue.

12.4 OVERRIDING DEFAULT LAYOUTS

Some reports may require specific formats. For example, if users working with your application are converting from a manual information management system, you may want to provide them with report forms that resemble their existing manual forms. To do this, simply override the formatting layouts PROGRESS provides. Here's a variation of our delinquent customer report that uses some of the options you saw in Chapter 11:

t-rept4.p

```
FOR EACH customer WHERE curr-bal >= 1400:
  DISPLAY SKIP(1)
          name NO-LABEL curr-bal AT 52
          FORMAT "$zzz,zz9.99 CR" SKIP
          contact max-credit AT 54

          address NO-LABEL ytd-sls AT 52
          FORMAT "$zzz,zz9.99 CR" SKIP
          city NO-LABEL st NO-LABEL zip NO-LABEL SKIP(1)
          phone AT 28 SKIP(1)
          WITH 1 DOWN CENTERED SIDE-LABELS ROW 6
          TITLE "DELINQUENT".
END.
```

```
                        DELINQUENT

Match Point Tennis                      Unpaid bal: $77,674.66
Contact: Robert Dorr                      Max cred: $ 1,970.00
66 Homer Ave                            Ytd sls: $ 21,890.06
Como          TX  75431

                    Tel num: (817) 498-2801


Press space bar to continue.
```

12.5 SENDING REPORTS TO DIFFERENT DEVICES

So far, we've used PROGRESS' default output destination, the terminal screen, to display reports. Often you'll want to generate printed reports or send the output to other destinations, such as a file or a printer.

You use the OUTPUT TO statement to send data to a destination other than your screen. Output destinations can include a printer, an ASCII file, or a DOS, OS/2, UNIX, VMS, or BTOS/CTOS device, or to a VMS printer queue. Chapter 9 of the *PROGRESS Programming Handbook* describes how PROGRESS works with all of these output destinations. For now, we'll focus on how to direct reports to an ASCII file or printer.

12.5.1 Storing Mailing Labels in a File

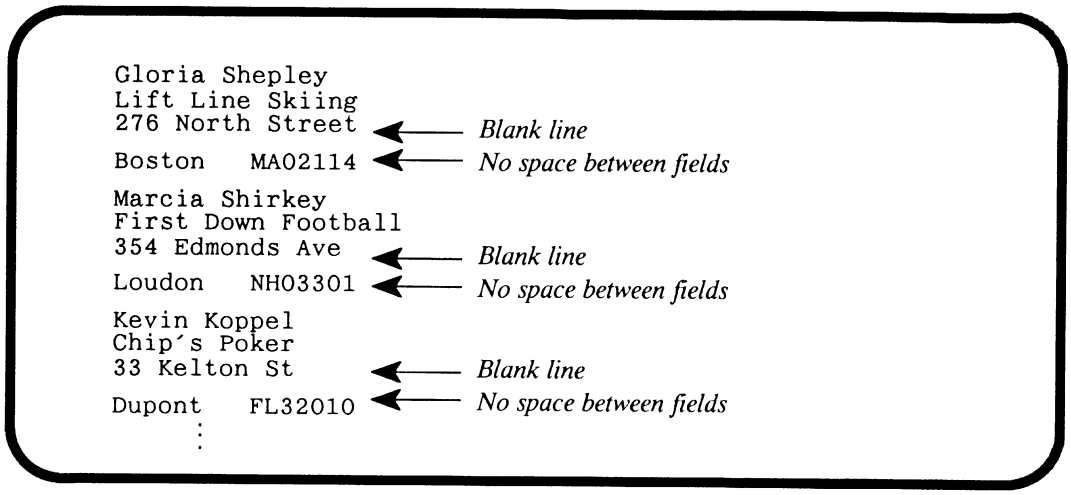
Let's assume that once you have a comprehensive list of delinquent customers, you plan to send them notices. You'll want a procedure that creates mailing labels for sending the notices. The customer file has two address fields, address and address2. Suppose your mailing label procedure looked like t-rept5.p.

t-rept5.p

```
OUTPUT TO mail.lst.

FOR EACH customer WHERE curr-bal >= 1400 BY zip:
  PUT contact SKIP
      name SKIP
      address SKIP
      address2 SKIP
      city st zip SKIP(1).

END.
```

The OUTPUT TO statement directs the output in this procedure to an ASCII file named mail.lst instead of to your screen. You can use the PUT statement in addition to the DISPLAY statement when sending data to a file, a printer, or any destination other than the screen.

If you run this procedure, you won't see the customer labels displayed on your terminal. However, once you have run the procedure, you can press GET (F5) and retrieve mail.lst to view the labels.



Notice that customer records that don't have a second address line are displayed with a blank line in the address2 frame field. In addition, the layout for the city, state, and zip fields needs to be improved. The PUT statement does not automatically leave a space between fields.

Here is a mailing label procedure that takes care of the blank line problem and also corrects the layout for the city, state, and zip fields:

t-rept6.p

```
OUTPUT TO mail.lst.

FOR EACH customer WHERE curr-bal >= 1400 BY zip:
  PUT contact SKIP
      name SKIP
      address SKIP.
  IF address2 NE "" THEN PUT address2 SKIP.
  PUT city + ", " + st + "  " + STRING(zip, "99999")
      FORMAT "x(23)" SKIP(1).
  IF address2 EQ "" THEN PUT SKIP(1).
END.
```

In this procedure, PROGRESS checks the address2 field. If the second address field is not blank (address2 NE ""), the PUT statement sends the address2 value to the file.

To improve the layout of the city, state, and zip fields, the procedure uses the concatenation operator (+) to separate these fields by a comma and a space. The STRING function converts the zip field to a character value which can be concatenated with the city and state fields.

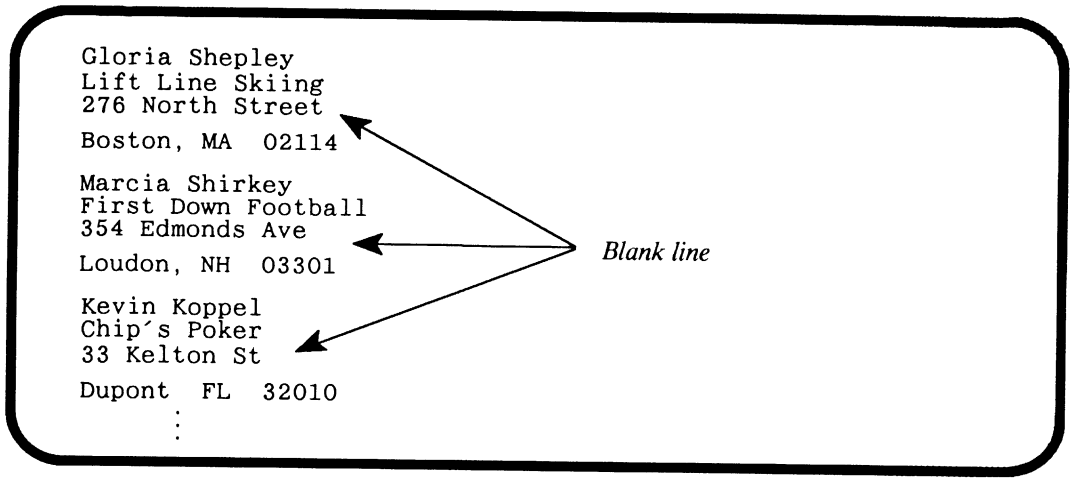
```
Gloria Shepley
Lift Line Skiing
276 North Street
Boston, MA  02114

Marcia Shirkey
First Down Football
354 Edmonds Ave
Loudon, NH  03301

Kevin Koppel
Chip's Poker
33 Kelton St
Dupont, FL  32010
        .
        .
        .
```

You may be wondering why, after using DISPLAY in so many of the procedures you've seen in this manual, we are now using PUT? Let's substitute DISPLAY for PUT in t-rept6.p, calling the new procedure t-rept7.p:

t-rept7.p

```
OUTPUT TO rept7.dat.

FOR EACH customer WHERE curr-bal >= 1000 BY zip:
→DISPLAY contact SKIP
      name SKIP
      address SKIP WITH NO-LABELS NO-BOX.
  IF address2 ne "" THEN DISPLAY address2 SKIP
  DISPLAY city + ", " + st + "  " STRING(zip, "99999")
      FORMAT "x(23)" SKIP(1).
  IF address2 EQ "" THEN DISPLAY SKIP(1).
END.
```

Now, run t-rept7.p and then GET rept7.dat:



This procedure uses just one frame. The FOR EACH block automatically gets a frame and all of the DISPLAY statements use that frame because they do not name any other frames. When designing the frame for the procedure, PROGRESS starts at the top of the procedure, determining how much space to allocate for various displays. In this top-to-bottom pass, PROGRESS allocates space for:

- The contact field.
- The name field.
- The address field.
- The address2 field. The fact that the display of the address2 field is conditional (it is only displayed if it contains text) doesn't matter. PROGRESS assumes that the field could be displayed and so makes space for it in the frame.
- The concatenation of the city, st, and zip fields.
- A blank line for the SKIP(1) on the last line inside the FOR EACH block.

When PROGRESS is designing a frame, it does not consider data output by the PUT statement. The PUT statement simply puts out the data one line at a time.

We have shown how to produce "1-up" labels (the labels are all in one column). An example procedure to produce labels 1, 2, or 3 up is included in the prodemo subdirectory with the PROGRESS software. You can use that and the other demo procedures in developing your application, as described in Chapter 14 of the *PROGRESS Programming Handbook*.

12.5.2 Sending Output to the Printer

If your printer is on line, you may want to print the mailing labels to see how they look. Run t-rept8.p, which is simply a modified version of t-rept6.p in which the output destination is changed from a file to a printer.

t-rept8.p

```
OUTPUT TO PRINTER.
FOR EACH customer WHERE curr-bal >= 1400 BY zip:
  PUT contact SKIP
      name SKIP
      address SKIP.
  IF address2 ne "" THEN PUT address2 SKIP.
  PUT city + ", " + st + "  " STRING(zip, "99999")
      FORMAT "x(23)" SKIP(1).
  IF address2 EQ ""THEN PUT SKIP(1).
END.
```

12.5.3 Sending Control Sequences to the Printer

When you send output to a printer, you may want to modify the way the printer generates that output. Many printers have a set of "control sequences" you can use to specify different print characteristics. You may, for example, want to change the number of printed characters per inch.

When you write a procedure that sends output to a printer, you can include printer control sequences within that procedure. Many control sequences involve special characters that can be represented by their octal (base 8) equivalent. To distinguish these octal codes, you precede the three octal digits by an escape character. PROGRESS then converts the octal number to a single character.

If you are using DOS, OS/2, or VMS, the escape character you use is a tilde (~). If you are using UNIX or BTOS/CTOS, the escape character you use is a tilde (~) or a backslash (\).

Let's assume you want to print a report on your Brand X printer, using compressed print mode:

t-rept9.p

```
DEFINE VARIABLE start-compress AS CHARACTER
  INITIAL "~033[3w".
DEFINE VARIABLE stop-compress AS CHARACTER
  INITIAL "~032[3w".
DEFINE VARIABLE small AS LOGICAL.

SET small LABEL "Use compressed print mode?"
OUTPUT TO PRINTER.
IF small THEN PUT CONTROL start-compress.

FOR EACH customer WHERE curr-bal >= 1400:
  DISPLAY name phone curr-bal sales-rep.
END.

IF small THEN PUT CONTROL stop-compress.
OUTPUT CLOSE.
```

In this procedure, the start-compress variable contains the four character sequence that puts the printer into compressed print mode. These four characters are octal 033 (decimal 27) followed by left bracket ([), a three (3), and a lowercase w. The variable stop-compress takes the printer out of compressed print mode. The particular codes vary from one printer to another.

The procedure asks if compressed print mode should be used in sending the output to a printer. If the answer is yes, the contents of the start-compress variable (the printer control sequence) is sent to the printer using the PUT CONTROL statement.

12.6 USING RUNNING HEADERS AND FOOTERS IN A REPORT

The reports you generate in many business applications typically span several pages. To provide continuity and uniformity to the report, as well as some polish, you can include running headers and footers.

12.6.1 Using Running Headers

Let's assume that when you print a delinquent customer report, you want to print a header on the top of each page that shows the date and the name of the sales rep responsible for the customers listed on that page.

t-rept10.p

```
OUTPUT TO rept10.dat PAGE-SIZE 10.

FOR EACH customer WHERE curr-bal >= 1400:
        BREAK BY sales rep:
    FORM HEADER TODAY "Delinquent Customer Report" AT 25
        "Page" AT 70 PAGE-NUMBER FORMAT ">>9"
        WITH PAGE-TOP FRAME a.
    VIEW FRAME a.
    DISPLAY cust-num name sales-rep.
END.
```


Get the rept10.dat file that was generated by the t-rept10.p procedure. Here is what that data file looks like:

```
11/07/90    Delinquent Customer Report          Page    1

Cust num Name                 Sls rep
-------- -------------------- -------
       6 Lift Line Skiing     BBB
      11 First Down Football  BBB
      29 Chip's Poker         BBB
       2 Match Point Tennis   DKP
      24 On Target Rifles     DKP

11/07/90    Delinquent Customer Report          Page    2

Cust num Name                 Sls rep
-------- -------------------- -------
       9  Spike's Volleyball   SLS
      17  Hard Knocks Skating  SLS
                  .
                  .
                  .
```

There are several parts of the procedure that cause the header to appear multiple times in the file:

- The PAGE-SIZE option in the OUTPUT TO statement ensures that you will see multiple pages in the output file. Without this option, there wouldn't be enough customers in the report to illustrate the form headers. (You can also send paged output to the terminal.)
- The FORM statement defines a frame (frame a) that is a PAGE-TOP frame. The PAGE-TOP option tells PROGRESS to display the frame at the top of each new page.
- All frames have a header and a body. The header is the top part of a frame. Since you want all the information shown above to appear at the top of the frame, you define them as the header part of the frame. The HEADER option in the FORM statement tells PROGRESS to reevaluate all expressions in the FORM statement at the beginning of

each new page. For example, the page number is 1 at the start of the report but changes to 2 at the start of the second page. This changed value is displayed because PROGRESS reevaluates expressions in the HEADER portion of the frame at the start of each page.

- The only way to activate a PAGE-TOP frame is by viewing it (or displaying it). The VIEW statement brings the PAGE-TOP frame into view at the start of each new page.

12.6.2 Using Running Headers and Footers

Now let's write a procedure that includes header and footer information in a delinquent customer report. Suppose you want the top of each page of the report to look like this:

```
11/07/90     Delinquent Customer List for BBB      Page 1
```

Since you want the sales-rep and page number to be reevaluated at the start of each new page, place all this information in the header part of the frame. PROGRESS reevaluates the header part of a frame at the start of each new page of output.

Here is the first part of the procedure:

```
OUTPUT TO rept11.dat PAGE-SIZE 10.

FOR EACH customer WHERE curr-bal >= 1400
    BREAK BY sales-rep:

  FORM HEADER TODAY
       "Delinquent Customer List for" AT 25 sales-rep
       "Page" AT 70 PAGE-NUMBER
       FORMAT "z9" SKIP(1)
       WITH FRAME hdr PAGE-TOP NO-BOX NO-ATTR-SPACE.
```

The OUTPUT TO statement designates an ASCII file called rept11.dat as the output destination (you can change this destination to PRINTER if you prefer). The PAGE-SIZE option tells PROGRESS that each page of the report can contain up to 10 lines.

The FORM statement describes the layout of the frame to be printed at the top of each page, using the following options:

- HEADER tells PROGRESS to place in the header section of the frame the items named in the FORM statement.

- The TODAY function returns the current system date.

- The character constant "Delinquent Customer List for" begins AT column 25, followed by the current break group's sales-rep field value.
- The character constant "Page" begins AT column 70, followed by the current page number returned by the PAGE-NUMBER function.
- The FORMAT option provides a two-digit display format ("z9") for the page number.
- The FRAME option names the header frame hdr.
- The PAGE-TOP option tells PROGRESS to display the frame at the top of each new page.
- NO-BOX tells PROGRESS to not display a box around the frame.
- The NO-ATTR-SPACE option indicates that spaces need not be reserved in the frame layout for video attributes. These spaces are not needed since this frame will not be displayed on the terminal. In general, any frames that will not be displayed on a terminal should be designated as NO-ATTR-SPACE. The default status of ATTR-SPACE or NO-ATTR-SPACE is determined not by the device you direct the output to. Rather, it is determined by the characteristics of the terminal on which you compile the program. Therefore, including NO-ATTR-SPACE will ensure that output to a printer has a consistent appearance no matter what type of terminal the procedure is compiled on.

We'll also use a FORM statement to describe the frame layout for the delinquent customer information.

```
FORM name NO-LABEL curr-bal AT 52
     FORMAT "$zzz,zz9.99 CR" SKIP
     contact max-credit AT 54
     FORMAT "$zzz,zz9.99 CR" SKIP
     address NO-LABEL
     ytd-sls AT 55
     FORMAT "$zzz,zz9.99 CR" SKIP
     city NO-LABEL st NO-LABEL zip NO-LABEL SKIP
     phone AT 28 SKIP(4)
     WITH SIDE-LABELS NO-BOX NO-ATTR-SPACE.
```

Now you have described what you want the top of each page to look like and what you want the body of each page to look like. Suppose you want the bottom of each page to look like this:

```
Delinquent Customer List for BBB continued on next page
```

You use a FORM statement with the HEADER option to define this footer display.

```
FORM HEADER "Delinquent Customer Report for"
            sales-rep "continued on next page"
            WITH FRAME ftr PAGE-BOTTOM NO-BOX
                 NO-ATTR-SPACE.
```

- The HEADER option tells PROGRESS to place in the header section of the frame (not to be confused with the top of the page) the items named in the FORM statement.
- The character constant "Delinquent Customer Report for" is followed by the current break group's sales-rep field value, which in turn is followed by the character expression "continued on next page".
- The FRAME option names the frame ftr.
- The PAGE-BOTTOM option tells PROGRESS to display the frame at the bottom of each page. PAGE-TOP and PAGE-BOTTOM frames are activated based on DISPLAY or VIEW statements. They are deactivated when the block to which the frames are scoped iterates or ends.
- NO-BOX tells PROGRESS to not display a box around the frame.

The first three parts of the procedure you just saw simply described the frame layouts for our report. The final portion of the procedure actually produces the report:

```
VIEW FRAME hdr.
VIEW FRAME ftr.
DISPLAY name curr-bal contact max-credit address
        city st zip phone ytd-sls
        WITH NO-ATTR-SPACE.
IF LAST-OF(sales-rep) THEN DO:
   HIDE FRAME ftr.
   PAGE.
END.
```

Here is the complete report procedure:

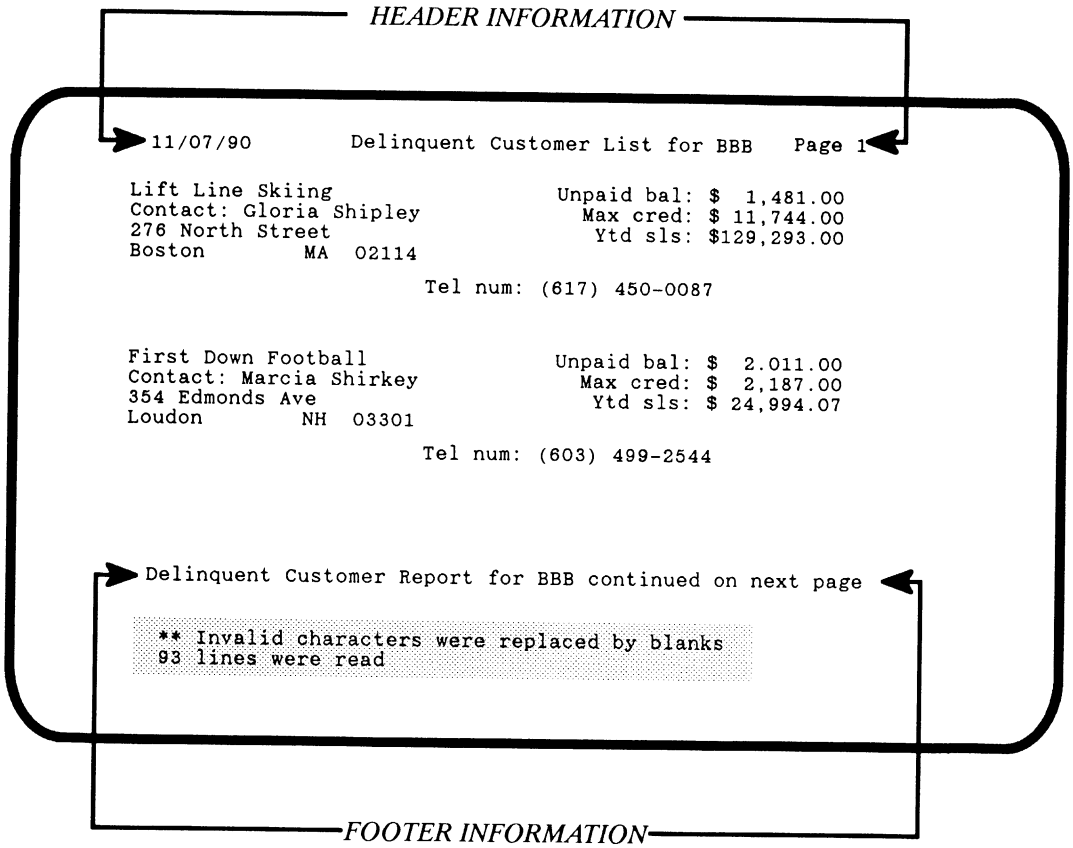
t-reptl1.p

```
OUTPUT TO reptl1.dat PAGE-SIZE 25.

FOR EACH customer WHERE curr-bal >= 1400
    BREAK BY sales-rep:

  FORM HEADER TODAY
       "Delinquent Customer List for" AT 25 sales-rep
       "Page" AT 70 PAGE-NUMBER
       FORMAT "z9" SKIP(1)
       WITH FRAME hdr PAGE-TOP NO-BOX NO-ATTR-SPACE.

  FORM name NO-LABEL curr-bal AT 52
       FORMAT "$zzz,zz9.99 CR" SKIP
       contact max-credit AT 54
       FORMAT "$zzz,zz9.99 CR" SKIP
       address NO-LABEL
       ytd-sls AT 55
       FORMAT "$zzz,zz9.99 CR" SKIP
       city NO-LABEL st NO-LABEL zip NO-LABEL SKIP
       phone AT 28 SKIP(4)
       WITH SIDE-LABELS NO-BOX NO-ATTR-SPACE.

  FORM HEADER "Delinquent Customer Report for" sales-rep
       "continued on next page"
       WITH FRAME ftr PAGE-BOTTOM NO-BOX NO-ATTR-SPACE.

  VIEW FRAME hdr.
  VIEW FRAME ftr.

  DISPLAY name curr-bal contact max-credit address
          city st zip phone ytd-sls WITH NO-ATTR-SPACE.
  IF LAST-OF(sales-rep) THEN DO:
     HIDE FRAME ftr.
     PAGE.
  END.

END.
```

Because the FORM statement only describes a frame but does not actually display it, you need a way to bring those frames into view. Whenever you have a PAGE-TOP or PAGE-BOTTOM frame, you use the VIEW statement to activate those frames for display at the beginning and end of a page.

PROGRESS then displays the customer information with the format we described earlier. The LAST-OF function checks to see if the current customer record is the last record within the present sales rep's break group. If this is the case, you do not want the footer to be displayed, so you use the HIDE statement to deactivate the frame ftr. The PAGE statement starts a new page when the sales rep field changes.

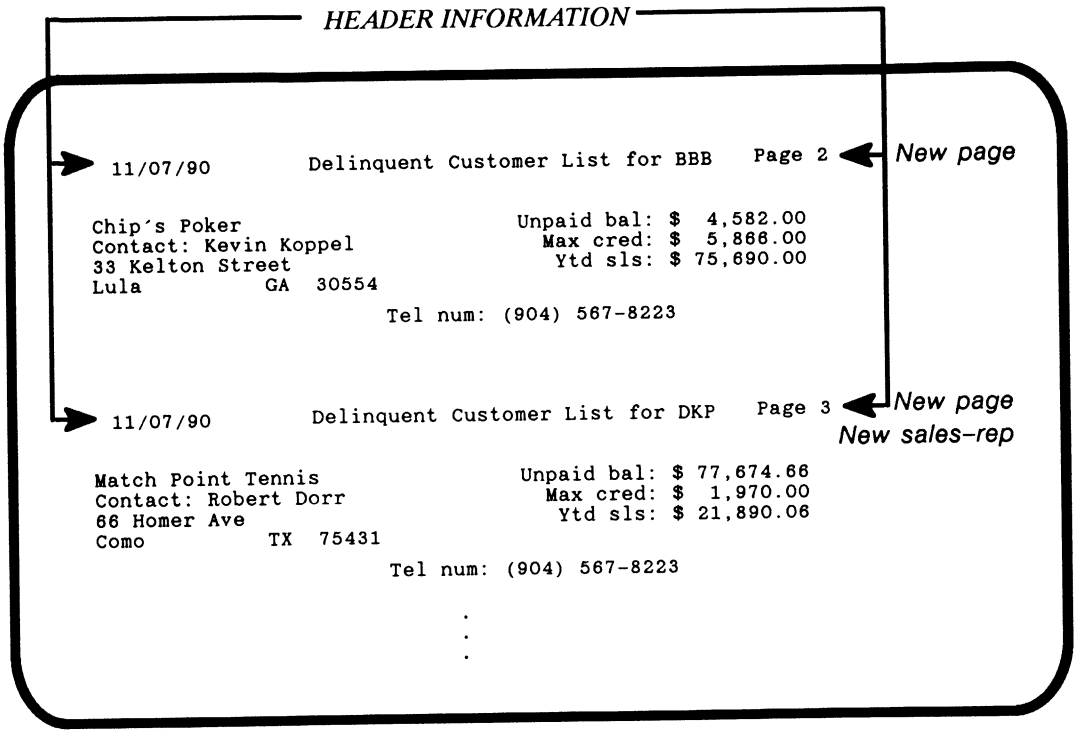
Because the output destination is a file, you won't see any activity on your screen when you run t-rept11.p. After you have run it, press GET (F5) and type the file name rept11.dat to see the output of the t-rept11.p procedure. Here is the rept11.dat file:



The message about replacing invalid characters means that the PROGRESS editor detected the new page or form feed characters that OUTPUT TO places in an output file. When you print the report, those characters cause the printer to start a new page.

Following the header, PROGRESS displays information about two of BBB's delinquent customers, using the frame specifications described in the procedure. The footer is displayed because BBB has more delinquent customer accounts.

Use the cursor keys or press PAGE DOWN to see more of the output file. When you look at the next delinquent customer for BBB, notice that the page number in the header frame has changed.



Because the last of BBB's delinquent customers (Chip's Poker) has been displayed, there is no page footer on Page 2.

Chip's Poker is the last delinquent customer in BBB's break group. PROGRESS begins a new page to start the display of delinquent customers for another sales rep, DKP. Notice that the page header on the new page uses DKP and a new page number, 3.

12.6.3 More on Page Headers and Footers

Here is another example using page headers and footers.

t-rept12.p

```
DEFINE VARIABLE cont AS CHARACTER FORMAT "X(70)".
                                                    /t-rept12.p */
OUTPUT TO rept12.dat PAGE-SIZE 17.

FORM HEADER  TODAY "Orders of Good Customers"
             AT 25 "Page" AT 74 PAGE-NUMBER TO 80
             FORMAT "z9" SKIP(1) cont
             WITH FRAME hdr PAGE-TOP NO-BOX
                  NO-ATTR-SPACE.

FOR EACH customer WHERE ytd-sls > 35000:
  FOR EACH order OF customer:

    FORM HEADER "Order number" ord-num
                "is continued on next page."
                WITH FRAME ftr PAGE-BOTTOM NO-BOX
                     NO-ATTR-SPACE.

    cont = "".
    PAGE.
    VIEW FRAME hdr.
    DISPLAY order-num name NO-LABEL AT 20 odate COLON 65
      address NO-LABEL AT 20 pdate COLON 65
      city + ", " + st + "  " + STRING(zip,"99999")
      AT 20 FORMAT "x(30)" SKIP(2) WITH SIDE-LABELS
                NO-BOX CENTERED NO-ATTR-SPACE.

    VIEW FRAME ftr.
      FOR EACH order-line OF order:
        cont = "Order number" + STRING(order-num,">>>9")
             + "continued ".
        FIND item OF order-lne.
        DISPLAY qty idesc price label "Unit Price"
                qty * price (TOTAL)
                FORMAT "$zzz,zz9.99" LABEL "Price"
                WITH NO-BOX CENTERED NO-ATTR-SPACE.
      END.
  END.
END.
```

```
DATE: 11/07/90      Current orders of Good Customers     Page 1

   Ord num: 5      Lift Line Skiing            Ord date: 09/13/90
                   276 North Street           Prom date: 10/15/90
                   Boston, MA  02114

     Qty  Desc                 Unit Price          Price
    -----  ----------------  -------------   ------------
       56  Ski Bindings             56.50    $  3,164.00
       23  Skis                    225.50    $  5,175.00
       67  Ski Poles                27.50    $  1,842.50
      125  Ski Wax, Red              2.75    $    343.50
                                             ------------
                                           $ 10,525.25 TOTAL
```

This report is similar to the one we looked at in the last section. The following list describes, step-by-step, what the procedure is doing:

- OUTPUT TO sends the output of the procedure to the rept12.dat file. The PAGE-SIZE option specifies just 17 lines to the page. (This procedure uses this option to illustrate page headers and footers.)
- The first FORM statement defines a frame called hdr. The HEADER option tells PROGRESS to reevaluate all the items in this part of the frame at the start of each page (because of the PAGE-TOP option). The items that will be reevaluated at the start of each new page are TODAY, PAGE-NUMBER, and the cont variable.
- The FOR EACH customer statement reads records of customers whose ytd-sales is greater than $35,000.
- The FOR EACH order OF customer statement reads the order records belonging to those customers.
- The second FORM statement defines a frame called ftr. The HEADER option tells PROGRESS to reevaluate all the items in this part of the frame at the end of each page (because of the PAGE-BOTTOM option). The only item that will be reevaluated at the end of each page is order-num.
- The cont variable is set to blanks.
- The PAGE statement starts a new page.

- The VIEW FRAME hdr statement activates the hdr frame. That means that the hdr frame is now available for display at the start of each new page.
- The DISPLAY statement displays order information.
- The VIEW FRAME ftr statement activates the ftr frame. That means that the ftr frame is now available for display at the end of each page.
- The FOR EACH order-line OF order statement reads each order for the current order record.
- The cont variable is set to the string "Order number *current-order-number* continued". This value will appear in the footer if a new page is needed in the middle of listing the order lines for an order.
- The FIND statement finds the item record for the item on the current order line.
- The DISPLAY statement displays some item information.

12.7 SUMMARY

This chapter showed how to write PROGRESS procedures that produce both simple and sophisticated reports. It explained how to:

- Generate categorized and sorted reports.
- Use multiple files in a single report.
- Override PROGRESS' default report layouts.
- Direct reports to devices other than the terminal screen.
- Produce mailing labels.
- Use running headers and footers in a report.

You can find additional information about frame design in Chapter 7 of the *PROGRESS Programming Handbook*.

Chapter 13
Compiling and Precompiling Procedures

So far, you have run procedures directly from the PROGRESS editor. When you ran unpack.p and, indirectly when you wrote the menu procedure, you ran procedures with the RUN statement. In this chapter, we'll look at what happens when PROGRESS runs your procedures and how you can increase the efficiency of procedure execution. We will cover the following topics:

- Compiling procedures.
- Precompiling procedures for fast execution.
- Finding procedures processed by compile, compile save, and run.
- Compiling procedures with icompile.p
- Compiling procedures for use on other computers.

13.1 COMPILING PROCEDURES

A procedure you create is called a **source procedure.** Before PROGRESS can run that source procedure, it must translate the source procedure into an internal format that can be efficiently processed by PROGRESS. This translation of your source procedure is called either the **session compile** or the **object** version of your procedure.

13.1.1 Compiling Procedures with the RUN Statement

Use the RUN statement to run t-csmenu.p, your Maintenance and Reporting menu procedure from Chapter 6 and notice the amount of time it takes for the procedure to start.

```
RUN t-csmenu.p.

   Enter PROGRESS procedure.  Press F1 to run.
```

When PROGRESS processes the RUN statement, it first creates a compiled version of the t-adcust.p procedure. It then runs the compiled version. This compiled version of the procedure is called a **session compile** because it is kept for the duration of a PROGRESS session. A PROGRESS session begins when you start PROGRESS with the pro or PROGRESS command, and ends when you run the QUIT statement.

If you run the t-csmenu.p procedure again without modifying the procedure and without leaving PROGRESS, PROGRESS uses the current session compile version of the procedure, saving the time otherwise required to recompile the source version of the procedure.

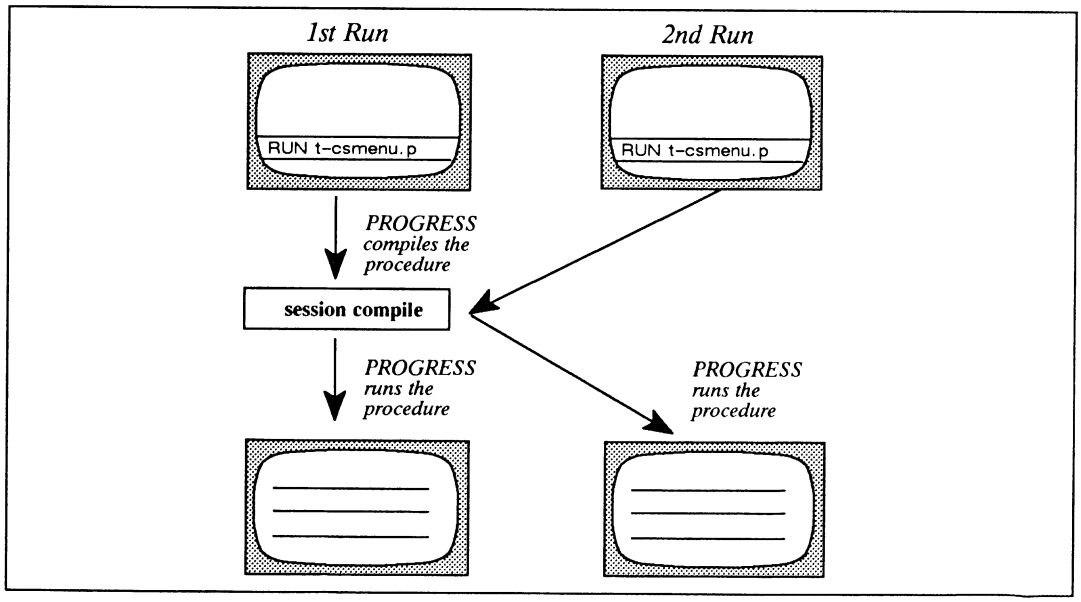


Figure 13-1: The RUN Statement

You can see this time savings for yourself. Run t-csmenu.p again. The procedure started much more quickly, didn't it? This time, PROGRESS used the session compile version of t-csmenu.p, and didn't have to spend time compiling the procedure over again.

If you leave PROGRESS after running the t-csmenu.p procedure, then start PROGRESS again and run the t-csmenu.p procedure, PROGRESS must create a new session compile version of the t-csmenu.p procedure again.

13.1.2 Explicitly Compiling Procedures

Instead of using the RUN statement to cause PROGRESS to compile your source procedure, you can use the COMPILE statement to explicitly compile that procedure.

```
COMPILE t-csmenu.p.

   Enter PROGRESS procedure.  Press F1 to run.
```


As with RUN, the COMPILE statement causes PROGRESS to create a session compile version of t-csmenu.p. Once the procedure is compiled, a RUN statement simply runs the session compile version.

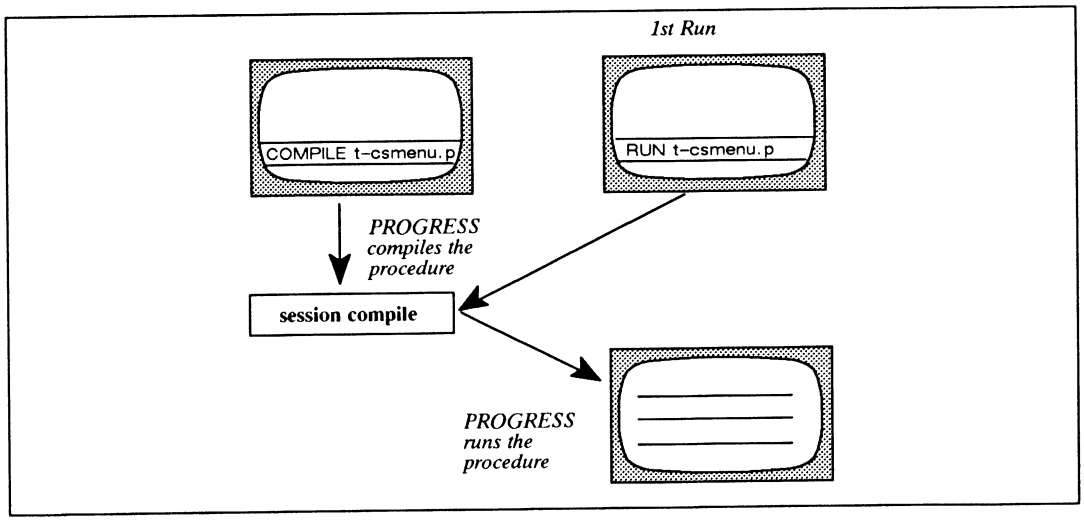


Figure 13–2: The COMPILE Statement

13.2 PRECOMPILING PROCEDURES FOR FAST EXECUTION

Let's use the t-finds.p procedure you worked with in Chapter 9 to do some more procedure "time tests." Run the t-finds.p procedure with the RUN statement, noticing the amount of time it takes for the procedure to start. Now type the following COMPILE statement and press GO (F1). Enter END (F4) to return to the PROGRESS editor.

```
COMPILE t-finds.p SAVE.

    Enter PROGRESS procedure.  Press F1 to run.
```

When you use the SAVE option with the COMPILE statement, PROGRESS produces a **precompiled** or **object** version of the procedure. PROGRESS stores this version in an object file whose name is the name of your procedure plus a .r file extension. If the procedure name has a .p file extension, such as t-finds.p, the COMPILE statement replaces the .p file extension with the .r file extension. However, you will still have your source, or .p version, in your working directory.

Now, if you check your working directory by running the **dos dir, os2 dir, unix ls, BTOS/CTOS FILES,** or **VMS DIRECTORY** command from the PROGRESS editor, you will see that a file named t-finds.r has been created. (You can use the SAVE INTO option to specify a different target directory for your object files. However, if you decide to put your object files into a library, you cannot use the SAVE INTO option. For more information on libraries, see "Building Libraries with the Prolib Utiltiy" in Chapter 4 of *System Administration II: General.*) Now use the RUN statement to run t-finds.p and notice how quickly the procedure starts.

```
RUN t-finds.p

    Enter PROGRESS procedure.  Press F1 to run.
```

Every time you run t-finds.p, PROGRESS runs the object version of your procedure.

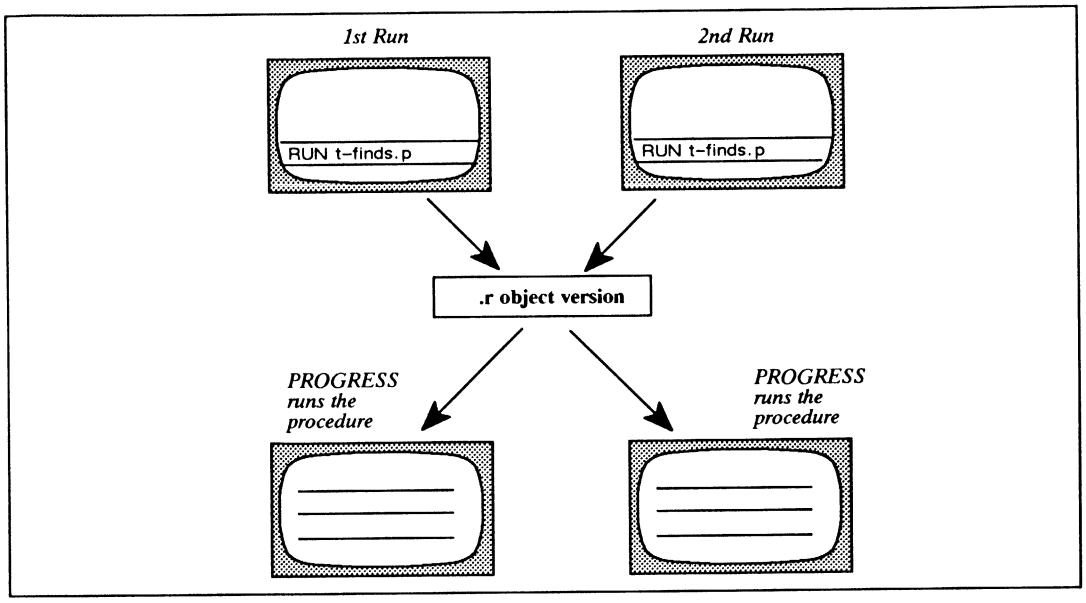


Figure 13–3: The Object File

If an object version does not exist, PROGRESS uses the source version. However, PROGRESS will have to take the time to compile it before it can be run.

Once you precompile a procedure and create an object file, that precompiled version of the procedure will be used whenever you execute the procedure using the RUN statement. If you make changes to the source procedure, you cannot test those changes unless you:

- COMPILE and SAVE again.

  OR

- Delete the object file. In this case, the source procedure will be used to do the session compile. On VMS, where you can create multiple versions of your object files, it is necessary to delete all versions of this object file.

  OR

- Run the procedure from the editor.

13.2.1 Reasons for Precompiling Procedures

PROGRESS saves object versions of procedures on disk across PROGRESS sessions. Saving the object version means that the procedure does not have to be compiled when it is run in any future PROGRESS session. Avoiding compilation each time a procedure is run is significant:

- You conserve system resources.
- You get faster response time when running the procedure.

These savings are important to any application.

In PROGRESS Version 4, once you have used the COMPILE SAVE statement to create an object version of a procedure, that object version is valid until you use the Dictionary to add or delete a file, field, or index. Any time you add or delete a database definition, PROGRESS gives that database an internal **time stamp** indicating the date and time of the most recent modifications.

When you precompile a procedure, PROGRESS gives this object version (.r file) a list of referenced files and their time stamps. If you delete or modify data definitions in one or more files in the list, the object version can no longer be run against those files. You will have to recompile the procedure to create a new object version.

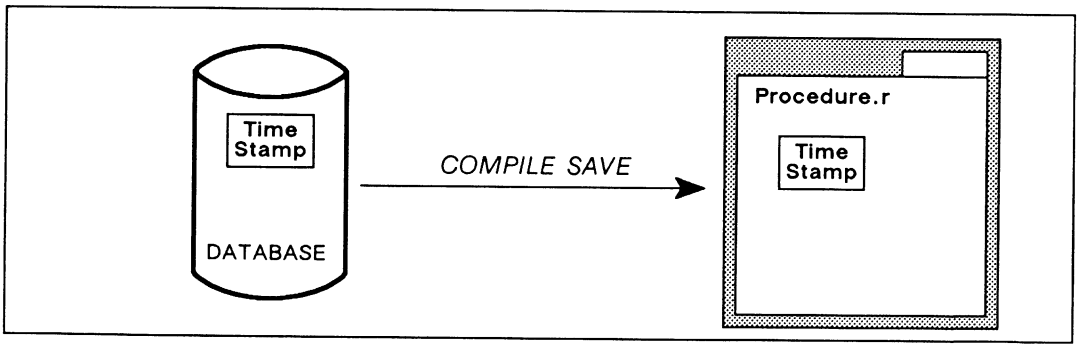


Figure 13–4: Version 4 Time Stamp

In PROGRESS Versions 5, once you have used the COMPILE SAVE statement to create an object version of a procedure, that object version is valid until you use the Dictionary to modify a database file (delete the file, or add/delete a field or index in the file) the procedure references. When a database file is added or modified, PROGRESS gives the file an internal **time stamp** indicating the date and time of the most recent modifications.

When you precompile a procedure, PROGRESS inserts in the object version (.r file) a list of referenced files and their time stamps. If you delete or modify data definitions in one or more files in the list, the object version can no longer be run against those files. You will have to recompile the procedure to create a new object version.

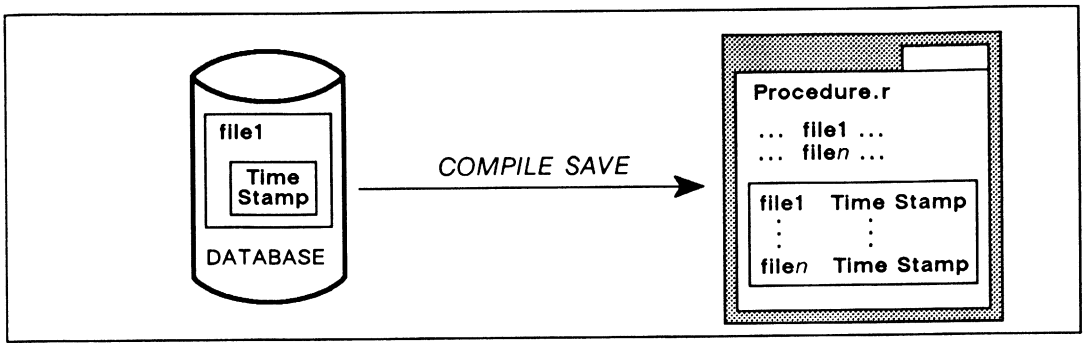


Figure 13–5: Version 5 Time Stamp

If you modify the source version of a procedure that you have previously compiled and saved, PROGRESS runs the object version unless you recompile the procedure, delete the object file, or run the procedure directly from the editor.

13.3 FINDING PROCEDURES PROCESSED BY COMPILE, COMPILE SAVE, AND RUN

When you use the COMPILE, COMPILE SAVE, or RUN statements to process a procedure, PROGRESS uses the value of the PROPATH environment variable to determine the path to use when searching for the procedure file you name. The PROPATH variable defines a list of directories that PROGRESS searches by default.

Table 13–1: Default Definitions for the PROPATH Variable

Operating System	PROPATH Variable Defaults
UNIX	:*$DLC*:*$DLC*/prodemo:*$DLC*/proguide
DOS & OS/2	;*%DLC%*;*%DLC%*\PRODEMO;*%DLC%*\PROGUIDE
VMS	",$DISK1:[DLC],$DISK1:[DLC.PRODEMO],$DISK1:[DLC.PROGUIDE]"
BTOS/CTOS	:DLC:DLC/prodemo:DLC/proguide

The *DLC* variable represents the name of the directory containing the PROGRESS system software. On DOS and OS/2 systems, this directory is usually \dlc; on UNIX systems, this directory is /usr/dlc; on VMS systems this directory is [DLC]; on BTOS/CTOS this directory is usually [sys]<dlc>. The prodemo directory holds various demonstration and utility procedures, and the proguide directory contains PROGRESS examples used in the PROGRESS documentation set.

If these default directories plus your working directory are the only ones you want PROGRESS to search, you do not need to explicitly set the PROPATH variable. However, if you want to store object files in a library (see "Building Libraries with the Prolib Utility" in Chapter 4 of the *System Administration II: General*) you must include the library in your search path for the PROPATH environment variable.

If you do define your own search path for the PROPATH variable and if you want your working directory to be searched first, be sure to start the path with either a semicolon (for DOS and OS/2 systems), a colon (for UNIX systems), or a comma (for VMS systems). The DLC directory and prodemo and proguide subdirectories are always automatically added on to any explicit PROPATH definition.

DLC represents the value of the "DLC" environment variable previously set. By default this variable is set to [sys]<dlc>.If you are using BTOS/CTOS, you explicitly set the PROPATH variable in your .env file. You cannot set the PROPATH variable interactively.

If you are using DOS, you explicitly set the PROPATH variable in your AUTOEXEC.BAT file or in any other batch file. If you are using OS/2, you explicitly set the PROPATH variable in your CONFIG.SYS file or in any other batch file. If you are using UNIX, you explicitly set the PROPATH variable in your .profile file or in another script. If you are using VMS, you explicitly set the PROPATH logical in your LOGIN.COM or in another command procedure. You can also define PROPATH interactively at either the DOS, OS/2, UNIX, or VMS system level. However, if you do, the path is no longer in effect when you log off and log back on to the system.

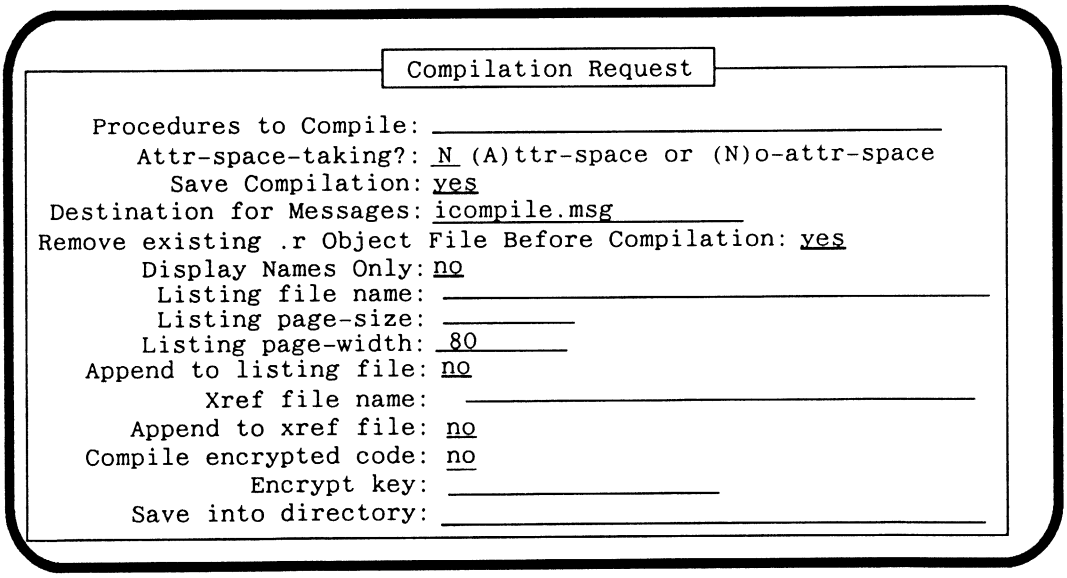
You can find detailed information on how COMPILE, COMPILE SAVE and RUN work with PROPATH in the *PROGRESS Language Reference* manual. You can find more information about operating system environments and networks in Chapter 5 of the *System Administration I: Environments* manual.

13.4 COMPILING PROCEDURES WITH icompile.p

Both PROGRESS 4GL/RDBMS and PROGRESS Query/Run-Time include in the `prodemo` directory a procedure, `icompile.p`, which you can use to compile procedures. The `icompile.p` procedure builds and executes a COMPILE statement and can be used as an alternative to COMPILE, either interactively or in batch mode for compiling large numbers of procedures. Use the **unpack.p** procedure to put a copy of `icompile.p` in your working directory.

13.4.1 Using icompile.p Interactively

When you run `icompile.p`, the following screen appears:



You are prompted for the following information:

- the name(s) of the procedure(s) you want to compile.

- whether to use the ATTR-SPACE or NO-ATTR-SPACE option. Your response depends on the type of terminal on which you intend to run your procedures. Remember that this option does not override any frame or field-level specifications. The default response for this question is NO-ATTR-SPACE.

- whether you want to save the results of your compilation. If you answer "yes", `icompile.p` adds the SAVE keyword to the COMPILE statement. The compiled `.r` files are stored in the current directory, unless you name another directory in the `Save into directory` field.

- the file to which the COMPILE statement can direct error messages. Because you are running `icompile.p` interactively, these messages also echo to the screen. The default file to which the error messages are directed is **icompile.msg**.

- whether to remove existing object files before the compilation proceeds. The default response to this questions is "yes".

- whether you want only to list procedures.

  By listing procedures, you can see the result of a particular request without actually running it. The default response to this question is "no".

- a listing file name. If you want the compiler to produce a listing file for the compilation, enter the name of the file where you want the listing written.

- the page-size of the listing file. Enter the number of lines per page.

- the page-width. Enter the number of characters per line, between 80 and 255. The default page-width is 80.

- whether you want the listing appended to the listing file. You should answer "yes" if you are compiling more than one procedure. The listings for each of the procedures that you compile are appended to the file. The default answer is "no."

- a compiler cross-reference listing file. Use of a cross-reference file is optional. It lists the database objects referenced by each procedure and specifies the exact type of reference made. See also the COMPILE statement in the *PROGRESS Language Reference* manual.
- whether to append a cross-reference listing to the existing cross-reference file.
- whether the source code is encrypted (and must be decrypted during compilation). If the procedure source code is encrypted and you do not specify an decryption key on the following line, the compiler attempts to decrypt the source using the standard PROGRESS default key.
- a decryption key. If you enter `yes` at the `Compile encrypted code:` _____ prompt, you must specify here the same key used to encrypt the source code. If you do not name a decryption key, the compiler attempts to decrypt the source using the standard PROGRESS default key (this is appropriate only if the source was encrypted using the default key).
- an object file directory. If you specify SAVE, compiled `.r` files are stored in the current directory, unless you name another directory in the `Save into directory` field.

13.4.2 Using icompile.p in Batch or Background Mode

You can also run the `icompile.p` procedure against a list of files. Use the same values supplied interactively above as arguments to a RUN statement. For example:

```
RUN icompile.p "file.lst" "A" "Y" "icompile.msg" "Y" "N" "icompile.lst"
                  1        2   3        4         5   6        7

    50 80 "Y" "icompile.xrf" "Y" "Y" "mykey" "DLCOBJ"
    8  9  10       11        12  13    14      15
```

1. **file.lst** contains a list of the procedure files to be compiled.
2. Compiles the procedures with the ATTR-SPACE option.
3. Specifies that the compilation should be saved.
4. Puts the compiler message into the file **icompile.msg**.
5. Removes the .r files before compiling the procedures.
6. Deselects the listing only option.
7. Names **icompile.lst** as the listing file.
8. Specifies a page length of 50 lines.
9. Specifies a page width of 80 characters (the default).
10. Specifies that the listing should be appended to the listing file.
11. Names `icompile.xrf` as cross-reference file.
12. Appends cross-reference to the existing cross-reference file.
13. Specifies that the source code is encrypted.
14. Supplies the `mykey` as the decryption key.
15. Stores the .r files in the DLCOBJ directory.

13.5 COMPILING PROCEDURES FOR USE ON OTHER COMPUTERS

When you compile a procedure on your computer, that procedure can be run only on computers of the same type. However, if you plan to distribute your application for use on other types of computers, you need to compile your procedures so they can run on those machines.

You can use either the PROGRESS 4GL/RDBMS (formerly Full PROGRESS) or the PROGRESS Developer's Toolkit to compile your procedures on the target machine. The Developer's Toolkit includes a utility, procomp, that lets you compile procedures to run on the machine for which you have purchased the Developer's Toolkit.

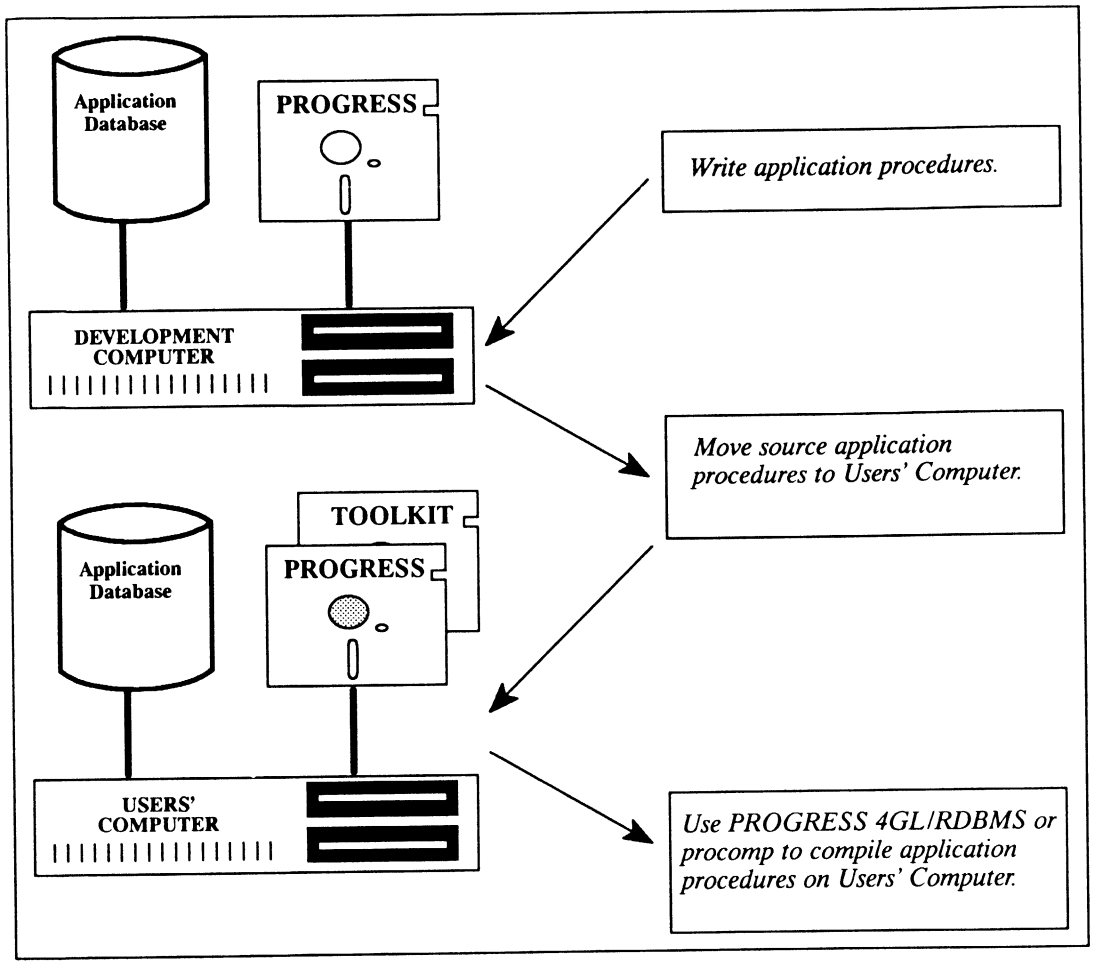


Figure 13–6: Porting an Application to Another Type of Computer

For more information about this utility, call Progress Software Corporation and ask about the Developer's Toolkit or, if you already have the Developer's Toolkit, see the *Developer's Toolkit Manual*.

13.6 SUMMARY

The way you handle procedure compilation determines how efficiently your procedures will run. In this chapter, you saw how PROGRESS compiles procedures when you use the RUN and COMPILE statements. You also learned that by modifying the COMPILE statement with the SAVE option, you can precompile your procedures. Precompilation lets you:

- Benefit from faster response times when running procedures.
- Conserve system resources.

Chapter 14
Using Procedures as Building Blocks

When you tackle an application, your first inclination might be to write a number of individual, potentially lengthy procedures where each procedure performs a complete and perhaps complex task. You can save yourself time and make your application easier to maintain by using **subprocedures** and **include files**.

Some procedures can easily be split into a series of smaller procedures that run from one main procedure. A main procedure might display a menu and run other small procedures that perform specialized tasks. For example, in the Maintenance and Reporting menu that you created in Chapter 6, one procedure adds a customer record, while another procedure deletes a customer record, depending upon the user's menu selection. Each of these smaller procedures is a **subprocedure**.

Often, different procedures in an application use the same set of statements to do a particular job. For example, many procedures might use the same FORM statement to create a standard data entry screen. You can put this FORM statement in a separate file called an **include file** that you can then, as its name suggests, include within other procedures.

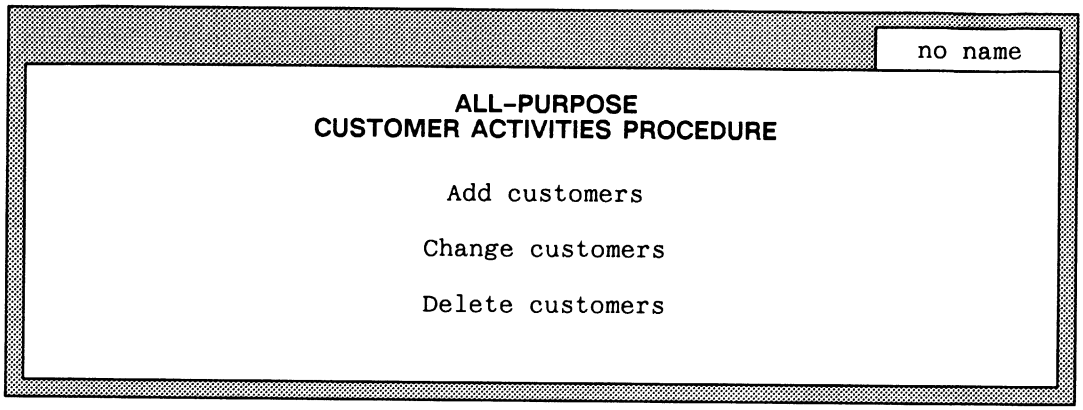
Subprocedures and include files let you:

- Minimize your application development time because you don't have to retype a sequence of statements each time you need those statements in a procedure.
- Achieve consistent application behavior because actions performed by a set of statements will behave in a predictable way when called by or included within another procedure.
- Make it easy to maintain your application because if you want to change or correct an error in a subprocedure or include file, you can change that one subprocedure or include file instead of correcting many different procedures.

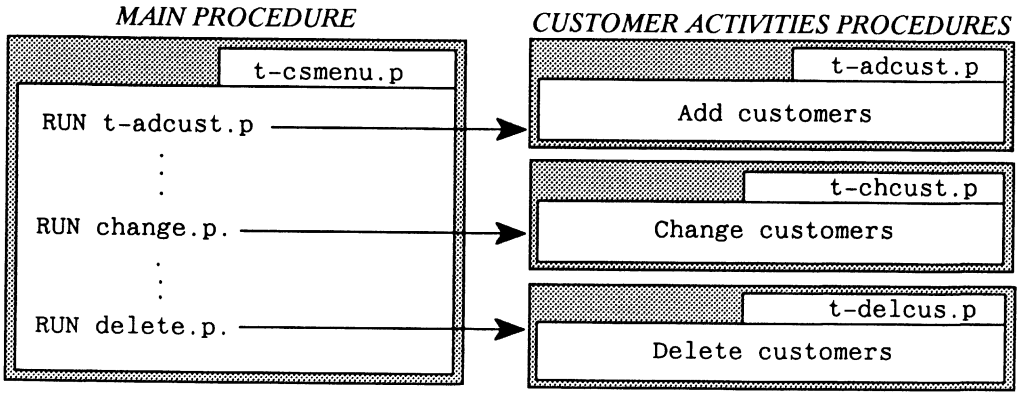
Let's take a look at how subprocedures and include files work. In this chapter, we'll cover the following topics:

- Calling subprocedures.
- Passing data to subprocedures.
- Including statements in multiple procedures.

Suppose that your application consists of a single procedure that performs many different customer-related tasks:



Instead of including all these tasks in a single procedure, you can write separate, smaller procedures that correspond to each of these tasks. You can then call, or run, these individual subprocedures from one main procedure. To run a procedure, you use the RUN statement followed by the name of the subprocedure you want to run.



If the subprocedure name is not fully qualified (that is, it does not consist of a complete directory path specification), PROGRESS searches the directories defined by the PROPATH environment variable, as described in Chapter 13.

Subprocedures are particularly useful for building menu-driven applications. When users start a menu-driven application, you usually want them to choose from a set of menu options displayed on the terminal. Here's the complete Maintenance and Reporting menu you created back in Chapter 6:

t-csmenu.p

```
DEFINE VARIABLE selection AS INTEGER FORMAT "9".

REPEAT:
  FORM
    SKIP(2)  "    M A I N  M E N U"
    SKIP(1)  " 1) Add a new customer"
    SKIP(1)  " 2) Change customer information"
    SKIP(1)  " 3) Display orders"
    SKIP(1)  " 4) Delete a customer"
    SKIP(1)  " 5) EXIT"
       WITH CENTERED TITLE "Maintenance and Reporting".
  UPDATE SKIP(2) SPACE(1) selection AUTO-RETURN
         WITH SIDE-LABELS.

  HIDE.
        IF selection EQ 1 THEN RUN t-adcust.p.
  ELSE IF selection EQ 2 THEN RUN t-chcust.p.
  ELSE IF selection EQ 3 THEN RUN t-itlist.p.
  ELSE IF selection EQ 4 THEN RUN t-delcus.p.
  ELSE IF selection EQ 5 THEN QUIT.
  ELSE MESSAGE "Incorrect selection - please try again".
END.
```

Using RUN to call a subprocedure.

The menu procedure tells PROGRESS to run one of five procedures, depending on the user's selection. For example, if a user chooses selection 1, Add a new customer, PROGRESS runs the procedure t-adcust.p. Choosing option 5 causes the procedure to run the QUIT statement to return the user to the operating system.

14.1 PASSING DATA TO SUBPROCEDURES

When you run one or more procedures from a main procedure, you'll often want to share information among the main procedure and those other procedures. There are three ways to share information among procedures:

- Use a shared variable.
- Pass the data as an argument.
- Pass the data as a parameter.

14.1.1 Using Shared Variables to Pass Data

Chapter 10 explained how to use variables to store temporary data in a procedure, such as a variable x to store an integer value:

```
DEFINE VARIABLE x AS INTEGER.
```

This statement creates a **local** variable which you use within the procedure in which it is defined. The data in the variable disappears when you leave the procedure. If you want to use variables to pass information from one procedure to another, they must be **shared** variables.

To create a shared variable, you define the variable as NEW SHARED in the calling procedure, and as SHARED in the called procedure. The diagram below outlines the relationship between the calling procedure (Procedure1) and the called procedure (Procedure2), and shows how they each define a variable called shareinfo.

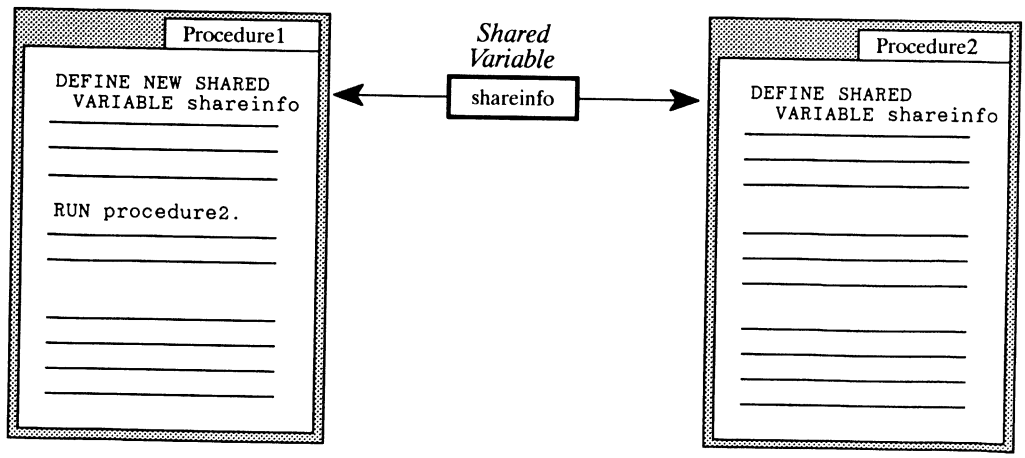


Figure 14–1: Shared Variables

When a procedure defines a variable as NEW SHARED, PROGRESS creates that variable when the procedure starts running. Other procedures, run directly or indirectly from the main procedure, have access to that variable if they define it as SHARED.

Now lets look at two complete procedures which use shared variables to pass information between themselves. These procedures are t-ordrp.p and t-ordrp2.p. The calling procedure, t-ordrp.p, creates two new shared variables, del and nrecs. It uses the del variable to pass information to t-ordrp2.p; t-ordrp2.p uses the nrecs shared variable to pass information back to t-ordrp.p.

The t-ordrp.p procedure is the controlling procedure.

t-ordrp.p

```
DEFINE NEW SHARED VARIABLE del AS LOGICAL.
DEFINE NEW SHARED VARIABLE nrecs AS INTEGER.

del = no.
MESSAGE "Do you want to delete the orders being printed (y/n)?"
    UPDATE del.

RUN t-ordrp2.p.
IF del THEN MESSAGE
  nrecs "orders have been shipped and were deleted.".
ELSE MESSAGE
  nrecs "orders have been shipped.".
```

The t-ordrp2.p procedure produces the report of all orders that have been shipped.

t-ordrp2.p

```
DEFINE SHARED VARIABLE del AS LOGICAL.
DEFINE SHARED VARIABLE nrecs AS INTEGER.

nrecs = 0.
FOR EACH order WHERE sdate <> ?:
  nrecs = nrecs + 1.
  FOR EACH order-line OF order:
    DISPLAY order-num line-num qty price.
    IF del THEN DELETE order-line.
  END.
  IF del THEN DELETE order.
END.
```

Before calling the report procedure, the t-ordrp.p procedure asks you if you want to remove shipped orders from the database after t-ordrp2.p puts them in the report. The procedure stores your response in the del variable.

```
MESSAGE "Do you want to delete the orders being printed (y/n)?"
    UPDATE del.
```

The t-ordrp2.p procedure can access the value of the del variable because both procedures define that variable as SHARED.

The t-ordrp2.p procedure keeps track of how many orders have been shipped and stores that value in the nrecs variable. The t-ordrp.p procedure has access to the value stored in nrecs (both procedures defined nrecs as SHARED), and displays it as part of a message:

```
MESSAGE nrecs "orders have been shipped and were deleted.".
```

14.1.2 Using Global Shared Variables to Pass Data

Sometimes you want to be able to share information between procedures that are not run from a common calling procedure, or you want to save information in a shared variable that retains its value even after the procedure that defined it ends. PROGRESS provides the **global shared** variable to satisfy both of these requirements. When a variable is defined as NEW GLOBAL SHARED, any procedure running in the same session that defines that variable as a SHARED variable has access to the value stored in that variable.

The following diagram shows the relationship between three procedures and the global shared variable that is accessed by two of these procedures. ProcedureA runs two procedures called ProcedureB and ProcedureC. ProcedureB defines a NEW GLOBAL SHARED variable called y. ProcedureC also defines the GLOBAL variable y. Once ProcedureB runs and defines the variable, any procedure that also defines that variable can access the data stored there. In the diagram, therefore, only ProcedureB and ProcedureC can access the data stored in the variable y. However, if another procedure called later in the PROGRESS session defines the variable, it can access the data stored there.

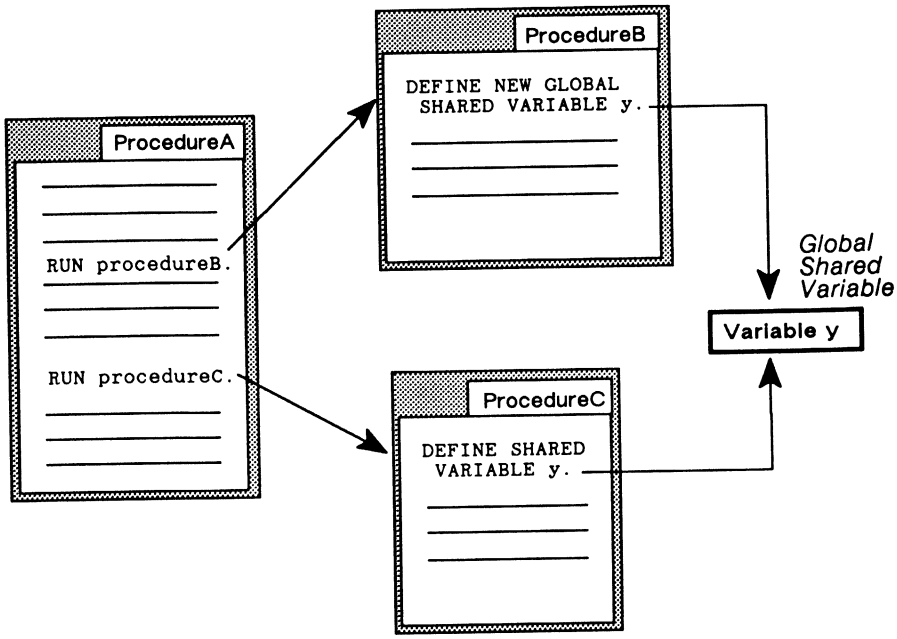


Figure 14–2: Global Shared Variables

Suppose you plan to distribute your Maintenance and Reporting application to many different companies. As part of your effort to tailor that application to each company, you may want the company name to appear throughout the application. The easiest approach to this task is to store that company name in a global shared variable, and then simply reference the global variable whenever you want to use that information.

t-global.p

```
DEFINE NEW GLOBAL SHARED VARIABLE cname AS CHARACTER
    FORMAT "x(20)".
DEFINE NEW GLOBAL SHARED VARIABLE tdate AS DATE
    INITIAL TODAY.

cname = "All Around Sports".
RUN t-csmnu2.p.
```

The t-global.p procedure creates a new global shared variable, cname, to hold the company name. Here, the value for cname is All Around Sports. A second global shared variable, tdate, stores today's date. (You might even store the company name in the database so that t-global.p could access it. That way, if the company name changed, you wouldn't have to change the t-global.p procedure.)

You can use the DEFINE SHARED VARIABLE statement to reference these variables throughout your Maintenance and Reporting system. For example:

t-csmnu2.p

```
DEFINE SHARED VARIABLE cname AS CHARACTER FORMAT "x(20)".
DEFINE SHARED VARIABLE tdate AS DATE.

DEFINE VARIABLE selection AS INTEGER FORMAT "9".
DISPLAY cname tdate TO 75 SKIP(1) WITH NO-LABELS NO-BOX.

REPEAT:
  FORM
    SKIP(2)  "    M A I N  M E N U"
    SKIP(1)  " 1) Add a new customer"
    SKIP(1)  " 2) Change customer information"
    SKIP(1)  " 3) Display orders"
    SKIP(1)  " 4) Delete a customer"
    SKIP(1)  " 5) EXIT"
       WITH CENTERED TITLE "Maintenance and Reporting".
  UPDATE SKIP(2) SPACE(1) selection AUTO-RETURN
         WITH SIDE-LABELS.
  HIDE.
        IF selection EQ 1 THEN RUN t-adcust.p.
   ELSE IF selection EQ 2 THEN RUN t-chcust.p.
   ELSE IF selection EQ 3 THEN RUN t-itlist.p.
   ELSE IF selection EQ 4 THEN RUN t-delcus.p.
   ELSE IF selection EQ 5 THEN QUIT.
   ELSE MESSAGE "Incorrect selection - please try again".
END.
```

The t-csmnu2.p procedure, which is a slightly modified version of the t-csmenu.p procedure, defines cname and tdate as shared variables. When you run t-global.p (usually at the beginning of a session as a start-up procedure), it runs t-csmnu2.p, which displays the main menu.

```
All Around Sports                              12/31/90

        Maintenance and Reporting

                M A I N   M E N U

          1) Add a new customer
          2) Change customer information
          3) Display orders
          4) Delete a customer
          5) EXIT

        Selection: __

Enter data or press F4 to end.
```

14.1.3 Using Arguments to Pass Data

In addition to using shared variables to pass information from one procedure to another, you can pass information using **arguments**. An argument is a piece of data that a calling procedure gives to a called procedure. PROGRESS uses that data as it compiles the statements within the called procedure.

In the diagram below, Procedure1 runs Procedure2 and passes the value "arg1' to Procedure2. Procedure2 uses the argument by naming it within braces. The symbol {**1**} refers to the first argument, {**2**} to the second, and so on.

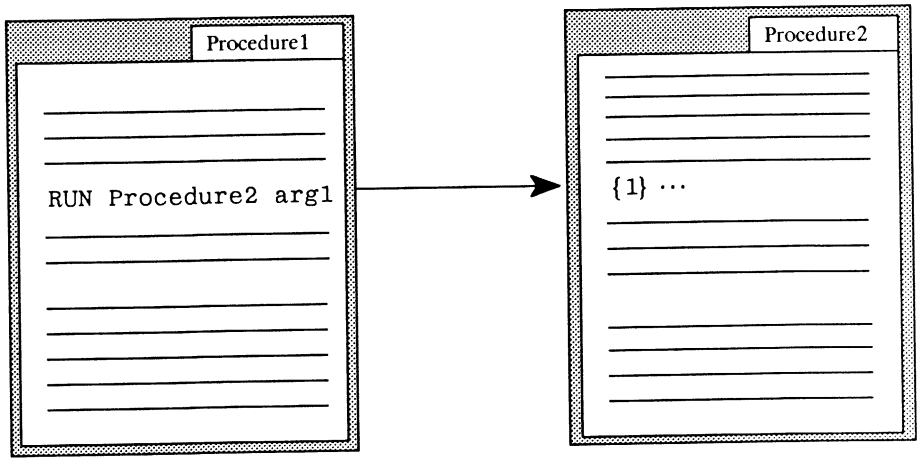


Figure 14–3: Passing an Argument

Let's look at the t-psarg1.p procedure, which calls the t-usearg.p procedure and passes the character constants "customer" and "name" as arguments:

t-psarg1.p

```
RUN t-usearg.p "customer" "name".
```

Here is the t-usearg.p procedure that is called by t-psarg1.p:

t-usearg.p

```
FOR EACH {1}:
  DISPLAY {2}.
END.
```

The t-usearg.p procedure substitutes the string "customer" as the first argument. The string "name" is substituted as the second argument. Once these substitutions are made, PROGRESS interprets t-usearg.p as follows:

```
FOR EACH customer:
  DISPLAY name.
END.
```

Next, PROGRESS compiles this procedure and displays the names of all customers. If the RUN statement passes different arguments to t-usearg.p, a different display results. For example, when you run t-psarg2.p with the following arguments, it displays the description of all items in the database:

t-psarg2.p

```
RUN t-usearg.p "item" "idesc".
```

You can see from these examples that passing arguments is a very flexible technique. However, the procedures that have arguments passed to them cannot be precompiled. This is because the compiler has no way of knowing what the arguments will be until run time. Therefore, if the called procedure being run has many statements, there will be a delay before that procedure starts to run while the called procedure is compiled. To avoid this problem, you will usually share data between procedures using shared variables, and will only pass arguments with the RUN statement when you cannot use shared variables.

14.1.4 Using Parameters to Pass Data

Runtime **parameters** provide a third alternative to using shared variables or arguments to pass information from one procedure to another. Unlike arguments, parameters passed at run-time do not cause the called procedure to be recompiled.

In the diagram below, Procedure1 runs Procedure2 and passes a parameter to Procedure2. Procedure2 defines a parameter of the same type (INPUT, OUTPUT, or INPUT-OUTPUT) and data type as `parameter1`. See the descriptions of the DEFINE PARAMETER and RUN statements in the *PROGRESS Language Reference* manual.

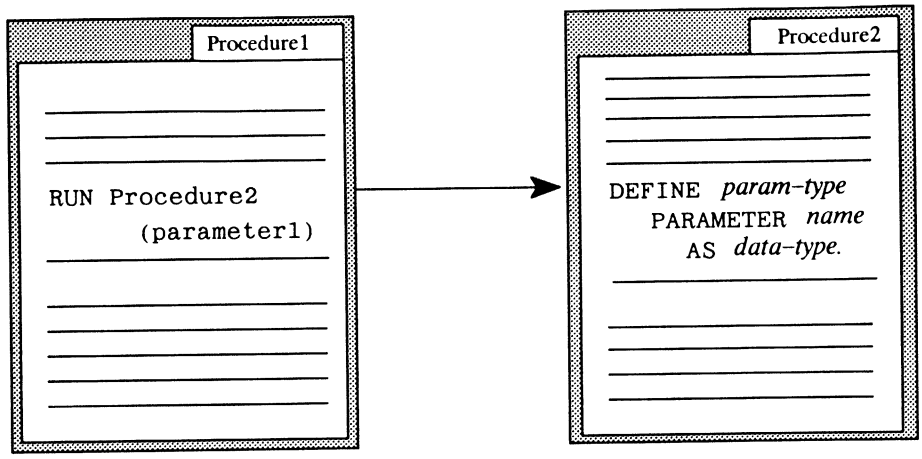


Figure 14–4: Passing a Parameter

14.2 INCLUDING STATEMENTS IN MULTIPLE PROCEDURES

Suppose there are two procedures in your application that use the same sequence of statements to perform a certain task. You could duplicate those statements in each of the procedures that requires them. For example, Procedure1 and Procedure2 both contain a FOR EACH block to display customer information.

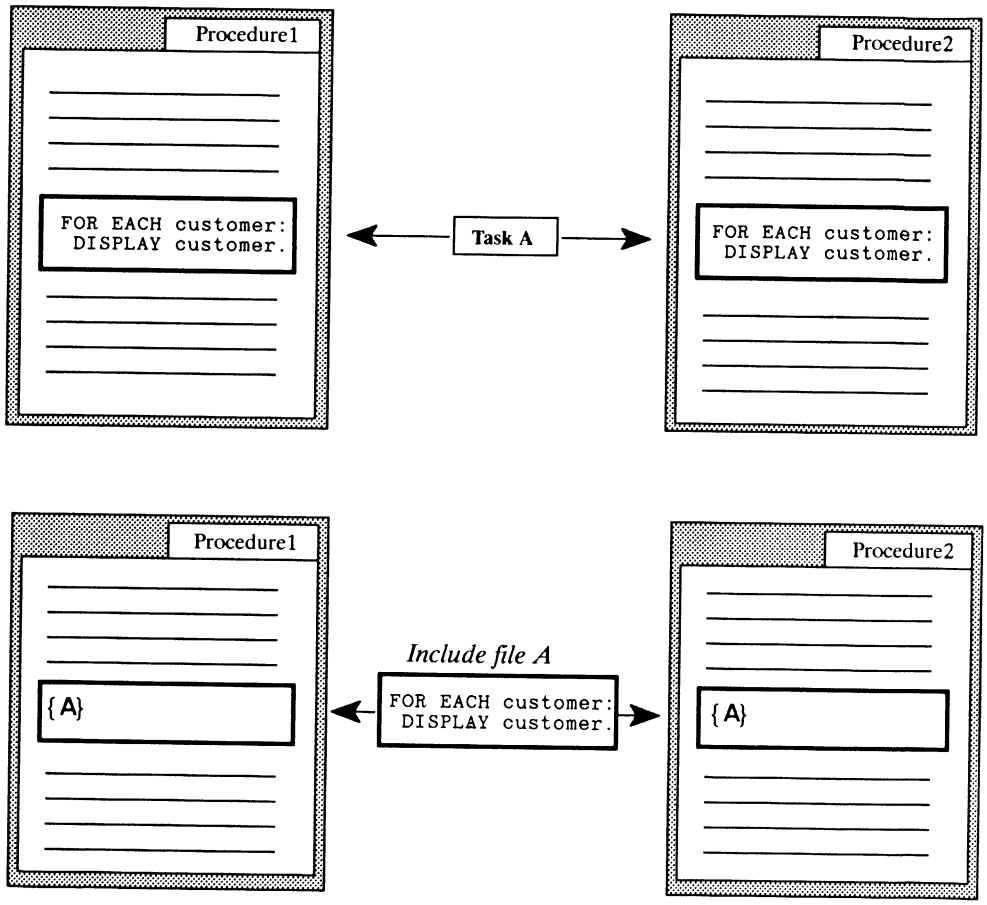


Figure 14–5: Using an Include File

To use a group of statements in more than one procedure, you can put those statements in an **include file**. When you need those statements within a procedure that you are writing, simply place the name of the include file within braces { }. The braces tell PROGRESS to retrieve the named include file at compile time and use the statements from the include file in the current procedure.

Let's look at an example of how you might use an include file in an application.

The t-prord1.p procedure prints all orders, information about the customer associated with each order, and details about each of the items on the order:

t-prord1.p

```
OUTPUT TO PRINTER.
FOR EACH order:
  DISPLAY order-num SKIP(1)
          name SPACE(20) pdate SKIP
          address SPACE(20) sdate SKIP
          city st zip SKIP(1) terms SKIP(1)
          WITH SIDE-LABELS FRAME ord.
  FOR EACH order-line OF order:
    FIND item OF order-line.
    DISPLAY qty price idesc item.item-num
          WITH 8 DOWN NO-BOX FRAME ol.
  END.
  PAGE.
END.
OUTPUT CLOSE.
```

Show Order

The t-dsord1.p procedure asks the user for an order number and displays information about the order, the customer associated with the order, and details about each of the items on the order:

t-dsord1.p

```
REPEAT FOR order:
  PROMPT-FOR order.order-num.
  FIND order USING order-num.
  HIDE.
  DISPLAY order-num SKIP(1)
          name SPACE(20) pdate SKIP
          address SPACE(20) sdate SKIP
          city st zip SKIP(1) terms SKIP(1)
          WITH SIDE-LABELS FRAME ord.
  FOR EACH order-line OF order:
    FIND item OF order-line.
    DISPLAY qty price idesc item.item-num.
      WITH 8 DOWN NO-BOX FRAME ol.
  END.
END.
```

Show Order

You can see that both the t-prord1.p and t-dsord1.p procedures use the same sequence of statements (labeled *Show Order* in the examples) to display the order information.

Here are those statements stored in their own include file. Throughout this book, include files are stored in a file with the .i extension. You do not need to use the .i extension, but you should use some naming convention so that you can easily identify the include files in your directory structure.

t-show.i

Show Order

```
DISPLAY order-num SKIP(1)
        name SPACE(20) pdate SKIP
        address SPACE(20) sdate SKIP
        city st zip SKIP(1) terms SKIP(1)
        WITH SIDE-LABELS FRAME ord.
FOR EACH order-line OF order:
  FIND item OF order-line.
  DISPLAY qty price idesc item.item-num.
    WITH 8 DOWN NO-BOX FRAME ol.
END.
```

The t-show.i procedure is a partial procedure. That is, it will not work if you run it directly. The FOR EACH statement assumes that an order record has been read and looks for the order-line records associated with that order record. Therefore, some other procedure must read the order record before the t-show.i procedure can be run.

These modified versions of the t-prord1.p and t-dsord1.p procedures use the t-show.i procedure as an include file:

t-prord2.p

```
OUTPUT TO PRINTER.
FOR EACH order:
  {t-show.i}
  PAGE.
END.
OUTPUT CLOSE.
```

t-dsord2.p

```
REPEAT FOR order:
  PROMPT-FOR order.order-num.
  FIND order USING order-num.
  HIDE.
  {t-show.i}
END.
```

14.2.1 Passing Arguments to Include Files

You can pass arguments to include files the same way you pass arguments to subprocedures. For example, the t-incarg.p procedure passes the words "customer" and "name" as arguments to the t-usearg.i include file:

t-incarg.p

```
DISPLAY "Customer List" WITH FRAME f1.
{t-usearg.i customer name}.
```

t-usearg.i

```
FOR EACH {1}:
  DISPLAY {2}.
END.
```

In the t-usearg.i include file, "customer" is substituted as argument 1 and name is substituted as argument 2. When you run t-incarg.p, it displays the name of each customer record in the database.

In addition to using numbers to identify arguments you are passing to an include file, you can use names to identify those arguments. Using named arguments to include files is often a good way to document, in your include files, the kind of information being passed to those files. For example:

t-namarg.p

```
DISPLAY "Customer List" WITH FRAME f1.
{t-namarg.i &file="customer" &field="name"}
```

t-namarg.i

```
FOR EACH {&file}:
  DISPLAY {&field}.
END.
```

In the t-namarg.i file, "customer" is substituted as the &file argument and "name" is substituted as the &field argument. When t-namarg.p is run it displays the name of each customer record in the database.

Any arguments the called procedure does not use are ignored. Passing three arguments to an include file that references only two does not cause an error.

If a called procedure references a missing argument, it is treated as a null value. To facilitate the use of include files and argument passing, a WHERE or WITH clause can be a null value; in addition, the WITH keyword can be repeated in a frame phrase. In the following example, the two expanded frame phrases, WITH WITH FRAME f1. and WITH., are both legal.

t-incar1.p

```
{t-usearg1.i "'Customer List'" "WITH FRAME f1"}.
{t-usearg2.i customer name}.
```

t-usrar1.p

```
DISPLAY {1} WITH {2}.
```

t-usear2.p

```
FOR EACH {1}:
  DISPLAY {2} WITH {3}.
END.
```

14.2.2 Precompiling Procedures with Include Files

If PROGRESS encounters an include file within a procedure that is being compiled, it retrieves the statements in the include file, substitutes the arguments, and compiles the statements as part of the main procedure. Because all references to arguments are resolved at compile time, you can precompile procedures that include other procedures, even if those include files pass arguments. For example, the procedure you just saw, t-incarg.p, can be precompiled.

14.3 SUMMARY

Subprocedures and include files are valuable time-savers during application development. This chapter showed you how to:

- Call one or more subprocedures from a main procedure.
- Pass information to subprocedures using variables and arguments.
- Use include files to incorporate commonly used statements into multiple procedures.
- Pass arguments to include files.

Chapter 15
Providing Help Information for an Application

While working through this Tutorial, you have probably taken advantage of the on-line help that PROGRESS offers you for application development. In this chapter, you'll learn how to provide help to your users as they run your application. We'll cover the follow topics:

- Using the data dictionary to provide help.
- Writing a PROGRESS procedure that provides help.

PROGRESS gives you two ways to provide application help. You can:

- Use the PROGRESS Data Dictionary to define help on a field by field basis. This help information is displayed automatically by PROMPT-FOR, SET, and UPDATE statements.
- Write a help procedure that PROGRESS can run when the user presses HELP (F2). The help you provide in this procedure can include general application help information, help information that is specific to each subsystem in the application, and information specific to database files and fields. In fact, you can have your help procedure perform any kind of processing you want.

In this chapter, we'll use these methods to provide help for a file maintenance application made up of three procedures:

- A menu procedure.
- A procedure that lets the user update order records.
- A procedure that lets the user update inventory records.

15.1 USING THE DATA DICTIONARY TO PROVIDE HELP

When you define a field, one of the characteristics you can define for that field is Help. Any time you are prompted to enter data into a field that defines a Help message, PROGRESS displays that message. For example, the order-num field in the order file defines a Help message.

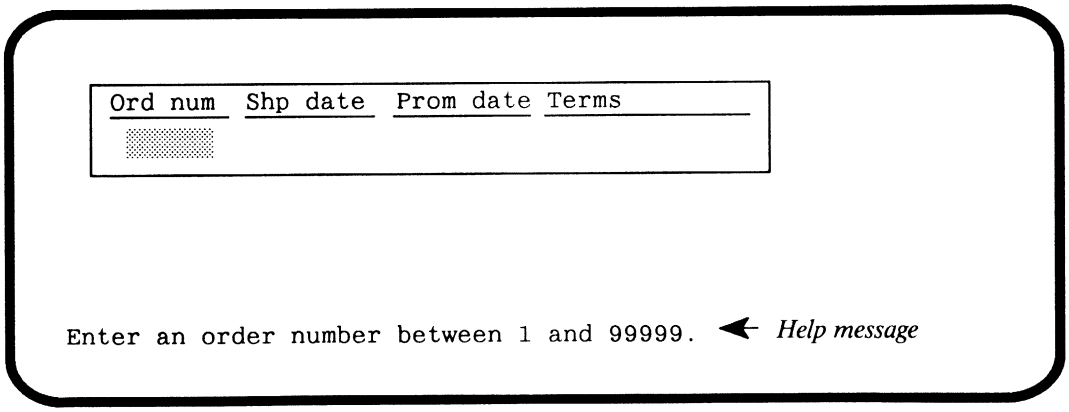
```
PROGRESS Data Dictionary                                    Field Editor
MODIFY-SCHEMA  SQL  Database  Admin  Utilities  Reports  Exit
                        Currently Defined Fields
Address       Address2    City        Cust-num    Cust-po      Misc-info
Name          Odate       Order-num   Pdate       Sales-rep    Sdate
Shipped       Shp-via     St          Terms       Zip

  Field-Name: Order-num                    Data-Type: integer
      Format: >>>>9                           Extent:
       Label: Ord num                       Decimals: ?
Column-label: ?                               Order: 10
     Initial: 0                           Mandatory: no   (Not Null)
Component of-> View: no   Index: yes     Case-sensitive: no
Valexp: order-num > 0
      :
      :
      :
Valmsg: Order number must be greater than zero
➔ Help: Enter an order number between 1 and 99999
  Desc:

  NextPage  PrevPage  Add  Modify  Delete  Copy  GoIndex  SwitchFile
    Browse  Order  Undo  Exit                          Total Fields: 17

Database: demo (PROGRESS)              File: order

Enter data or press F4 to end
```

Let's look at the t-ordmnt.p procedure, which works with the order file. The t-ordmnt.p procedure asks the user for an order number, displays the appropriate record if it is available, and lets the user update the shipping date (sdate), promise date (pdate), and payment terms (terms) fields. If there is no record that corresponds to the order number supplied, a new order record is created. This procedure also defines a shared variable called helpfile. Don't worry about this variable now; you don't need it until later in this chapter.

t-ordmnt.p

```
DEFINE SHARED VARIABLE helpfile AS CHARACTER.

helpfile = "t-ordmnt.h".

REPEAT WITH 1 DOWN CENTERED:
  PROMPT-FOR order.order-num.
  FIND order USING order-num NO-ERROR.
  IF NOT AVAILABLE order THEN DO:
      CREATE order.
      ASSIGN order-num.
  END.
  ELSE DISPLAY sdate pdate terms.
  SET sdate pdate terms.
END.
```

When you run this procedure, it displays the Dictionary Help message at the bottom of the screen. You did not have to do any special processing to display this message. It is displayed automatically by PROGRESS when it is waiting for the user to enter data into the field.



15.2 WRITING A PROGRESS PROCEDURE THAT PROVIDES HELP

In addition to the field-level help you define in the Dictionary, you can provide users with other kinds of help information which they access by pressing HELP(F2).

If the user presses HELP(F2) when being prompted for input, PROGRESS:

- Saves, but does not clear, the screen.
- Hides any frames previously displayed by your help procedure.

- Searches for and runs your help procedure, which must be named applhelp.p. If you do not provide your own version of applhelp.p, PROGRESS uses its own version, which rings the bell and displays the message "No application help is available." Otherwise, PROGRESS runs your applhelp.p, hiding whatever frames are necessary in order to display the help frames.

- When applhelp.p completes, any frames hidden in order to display help frames are redisplayed. Messages in the message area are not restored if the applhelp.p procedures used the message area or if messages are cleared as a result of an interaction during help processing.

- The user can then continue entering data. When the user is finished entering data, PROGRESS clears any frames displayed in applhelp.p that are still on the screen.

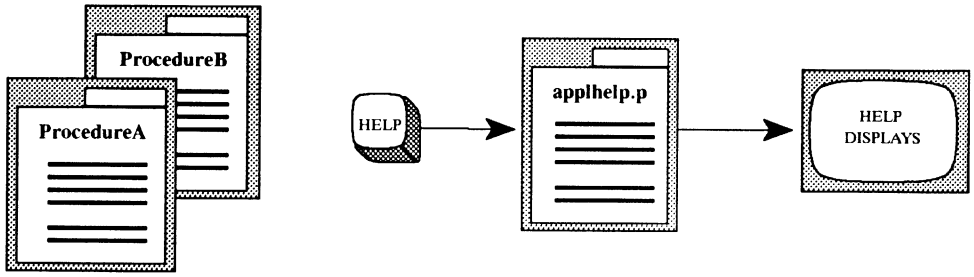


Figure 15–1: Using Application Help

The user cannot use the HELP(F2) key while the applhelp.p procedure is running.

Depending on how you write applhelp.p, that procedure can provide help at several different levels:

- Help about the application in general.

- Information about the particular subsystem or procedure the user is currently using.

- A description of the file or files with which the user is working.

- A description of the field (and possible values) of that field with which the user is working.

You can store the help information accessed by the applhelp.p procedure in several places:

- You can store help information in external ASCII text files. You can use PROGRESS to read the data in those files and display their contents on the screen or process them in some other way.

- You can define one or more database files and load help information into those files. This method is particularly useful if you want to use the PROGRESS database facilities to access a variety of help data in different ways.

- You can access file and field information. This information is stored in special files within the database which you can access with PROGRESS statements.

The next sections explain the first two of these techniques.

Your applhelp.p procedure, like all of your other PROGRESS procedures, has access to shared variables and can test the values of those variables to determine which help information to display. It can also RUN other procedures.

15.2.1 Supplying Help in Text Files

Suppose you want to build a help system for a menu-driven application. The application consists of a main menu and two subprocedures that are called from the menu. To build help into this application:

- Use a shared variable to identify the name of an external file that contains help information appropriate to each subsystem or procedure in the application.

- Display the information in the file when the user presses HELP (F2).

Here is the menu procedure for this application:

t-mntmnu.p

```
DEFINE NEW GLOBAL SHARED VARIABLE helpfile AS CHARACTER.
DEFINE VARIABLE selection AS INTEGER FORMAT "9".
DEFINE VARIABLE endlabel AS CHARACTER FORMAT "x(6)".


endlabel = KBLABEL("END-ERROR").

FORM
  SKIP(1) "Maintain Orders            1 "
  SKIP(1) "Maintain Inventory         2 "
  SKIP(2) "      Press" endlabel NO-LABEL "to exit."
  WITH FRAME menu SIDE-LABELS CENTERED
       TITLE "M A I N T E N A N C E   S Y S T E M".

REPEAT WITH FRAME menu:
   helpfile = "".
   DISPLAY endlabel.
   UPDATE SKIP(1) selection AUTO-RETURN.
   IF selection < 1 OR  selection > 2 THEN DO:
        MESSAGE "Incorrect selection. Please try again.".
        UNDO, RETRY.
   END.

   HIDE ALL.
   IF selection EQ 1 THEN RUN t-ordmnt.p.
   ELSE IF selection EQ 2 THEN RUN t-invmnt.p.
   HIDE ALL.
END.
```

❶ The global shared variable called helpfile is defined to store the name of a help file. Each procedure can have its own helpfile. Any procedure that wants to either define a helpfile or know the name of the currently defined helpfile can define this shared variable and access the data stored there.

The Selection variable stores the menu choice that the user selects.

The endlabel variable is used to customize the menu. It stores the keylabel of the END-ERROR key.

❷ This expression assigns the correct keylabel of the END-ERROR key to the endlabel variable. The various computer keyboards that this procedure may run on put different labels on this key. The KBLABEL function returns the keylabel of the key for the terminal that the procedure is currently running on.

❸ The FORM statement defines the layout of the main menu. Notice that the menu refers to the endlabel variable to display the correct keylabel for the END-ERROR key. However, the keylabel is not actually displayed until the REPEAT block, below.

❹ The REPEAT block initializes the helpfile variable to a blank, or **null string**,and displays the keylabel in the menu frame. Then, it checks to see if an incorrect menu choice was selected and, if so, displays a message, returns all variables and data fields to the values that they had at the beginning of the block (UNDO), and returns to the beginning of the block to let the user try again (RETRY).

❺ When a valid menu choice is selected, the procedure removes the menu and any other frames that are displayed on the screen, and runs either t-ordmnt.p or t-invmnt.p. The t-ordmnt.p procedure lets the user make changes to data in the order file. The t-invmnt.p procedure lets the user make changes to the data in the inventory file.

This is the menu you see when you run the procedure:

```
┌──M A I N T E N A N C E   S Y S T E M──┐
│                                       │
│ Maintain Orders                     1 │
│ Maintain Inventory                  2 │
│          Press F4  to exit            │
│ Selection:0__                         │
└───────────────────────────────────────┘

Enter data or press F4 to end.
```

When the user selects 1, **Maintain Orders**, PROGRESS runs the t-ordmnt.p procedure. When the user selects 2, **Maintain Inventory**, PROGRESS runs the t-invmnt.p procedure.

You saw earlier that the t-ordmnt.p procedure defines the helpfile shared variable and puts the value "t-ordmnt.h" into it. t-ordmnt.h is the name of the file containing help information to be used for that part of the application. The t-invmnt.p procedure, which runs the inventory maintenance subsystem, also shares data through the helpfile variable, but it assigns a different file name to the variable:

t-invmnt.p

```
DEFINE SHARED VARIABLE helpfile AS CHARACTER.

helpfile = "t-invmnt.h".

REPEAT WITH 1 DOWN CENTERED:
   PROMPT-FOR item.item-num.
   FIND item USING item-num NO-ERROR.
   IF NOT AVAILABLE item THEN DO:
      CREATE item.
      ASSIGN item-num.
   END.
   ELSE DISPLAY idesc cost.
   SET idesc cost.
END.
```

Each of these procedures requires a different help text file. The help text in the t-ordmnt.h and t-invmnt.h help files are stored as PROGRESS character strings. Therefore, each line of text is enclosed in quotation marks.

t-ordmnt.h

```
""
"You are using the order maintenance subsystem.  This"
"subsystem lets you add new orders or change existing"
"orders.  All of the information you work with in this"
"subsystem is stored in the ORDER file."
```

t-invmnt.h

```
""
"You are using the item maintenance subsystem.  This subsystem"
"lets you add new items or change existing items.  All of the"
"information you work with in this subsystem is stored in the"
"ITEM file."
```

Now all you need is an application-specific help procedure for PROGRESS to run whenever the user presses HELP (F2). If you're still running the order maintenance procedure, press END-ERROR (F4) to return to the editor. From the editor, press GET (F5) to retrieve the t-aphlp1.p procedure.

t-aphlp1.p

```
DEFINE SHARED VARIABLE helpfile AS CHARACTER.
DEFINE VARIABLE line AS CHARACTER FORMAT "x(76)".
DEFINE VARIABLE helpin AS CHARACTER.

helpin = SEARCH(helpfile).
IF helpin = ? THEN DO:
 FORM SKIP(9) SPACE(10)
 "Help is not available for this part of the application"
 SKIP(9) WITH WIDTH 80.
 VIEW.
 RETURN.
END.

INPUT FROM VALUE(helpin) NO-ECHO.

REPEAT WITH ROW 1 19 DOWN NO-LABELS ATTR-SPACE CENTERED
 TITLE "M A I N T E N A N C E  S Y S T E M  H E L P ":
 SET line. /* puts value into the frame field and
              the variable */
 DISPLAY. /* displays value previously loaded into
             frame */
END.
```

❶ The help procedure must also define the helpfile shared variable so that it can find out which file contains the appropriate help information. In addition to this variable, the procedure defines two local variables:

- The line variable stores a single line of help text.
- The helpin variable resolves to the full path name of the help text files.

❷ The SEARCH function looks for a file in the directory path defined by your PROPATH environment variable. This name is then stored in the helpin variable. If the help text file name is unknown (represented by the ?), then the procedure displays a message and returns to your application.

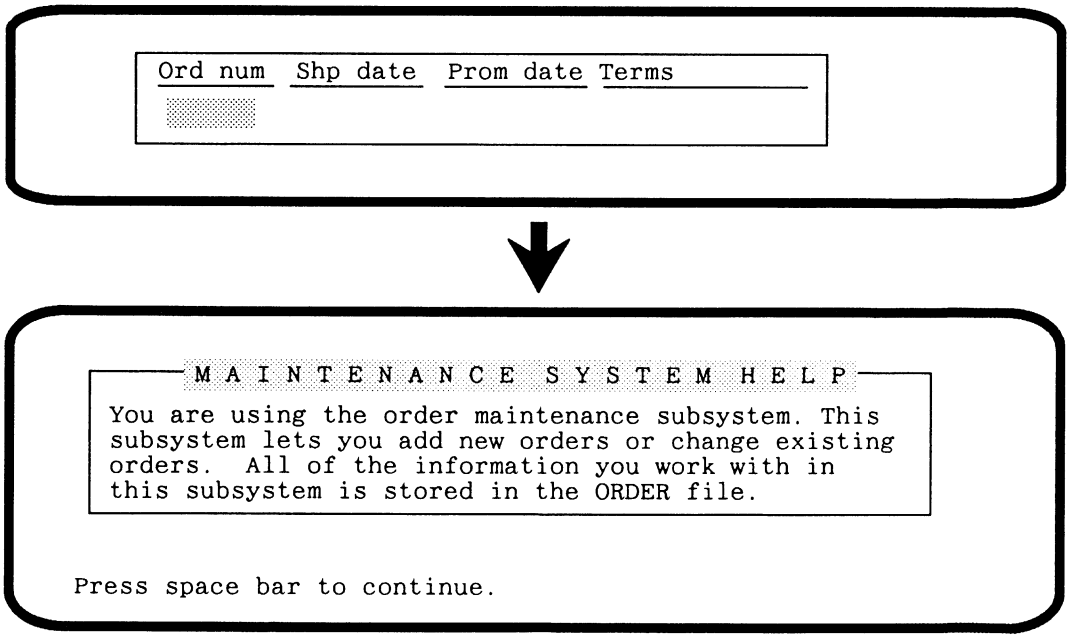
❸ If helpin contains a file name, then the INPUT FROM statement redirects the standard input device to that file. Any input statements, such as SET, PROMPT-FOR, or UPDATE, look for their input from this file rather than from the terminal. The NO-ECHO option ensures that the input is not echoed to the terminal.

❹ A REPEAT loop is used to display the help text contained in the file. The WITH ROW 1 option places the frame in the first row on the screen. Because it is the procedure's innermost iterating block, the frame defined for this block is a down frame.

Within the block, the SET statement reads a line from the help file on each iteration of the block and places the string in the screen buffer. The DISPLAY statement displays the value that was placed in the screen buffer by SET.

To use the procedure t-aphlp1.p as a help procedure, you must rename it applhelp.p. To rename the file, press GET (F5) to retrieve t-aphlp1.p if you haven't already, and press PUT (F6) to store the procedure in your working directory. Name it **applhelp.p**.

Now you can run the file maintenance application, t-mntmnu.p. To do this, first clear the edit area by pressing CLEAR (F8). You can run the procedure with the **RUN t-mntmnu.p** statement or by retrieving the procedure with GET (F5) and running it from the editor. From the application main menu, Select 1, Maintain Orders, and press HELP (F2) when you are prompted for an order number.



After the applhelp.p procedure ends, the screen is returned to the state it was in before you pressed HELP (F2). You can then continue entering data.

15.2.2 Accessing Dictionary Information in a Help Procedure

Every database you build with PROGRESS has a predefined structure, or **metaschema**, that is made up of six files:

_Db	Always contains at least one record for PROGRESS databases. If the PROGRESS database is a schema holder, this file will contain one additional record for each non-PROGRESS database defined.
_File	Contains a record for every file in the database.
_Field	Contains a record for every field in the database.
_Index	Contains a record for every index in the database.
_Index-field	Contains, for each index in the database, a record for each component of that index.
_User	Contains a record for each user who is authorized to use the application. You enter records in this file if you want to implement security features as described in Chapter 11 of the *PROGRESS Programming Handbook*.

The _Field file contains information about all the fields you define when you create a database file, such as the field name, data type, format, and so on. To see these fields, type **dict** in the edit area on your screen and press GO (F1) to run the Data Dictionary. Choose `Modify-Schema`. The Data Dictionary displays a menu. Type **m** to choose the `Modify Existing File` option. You don't see these metaschema files in the list that is displayed by the Dictionary. (They are "hidden" files; their `_File` records have `_Hidden` fields set to `yes`. You can set this field to true or false for any file if you have can-write access to the `_File` file.) By default, only your application's files appear in the list. Type **_Field** at the `File Name` prompt, and press RETURN. The Dictionary displays the definitions for the _Field file.

NOTE: Note that the definition for the _Field file is frozen. If you want to alter the file definition, you need to unfreeze it. To unfreeze a file, select the Freeze/Unfreeze option off the Utilities option on the Data Dictionary Main Menu. Refer to Chapter 5 in the *System Administration II: General* manual for more information about freezing and unfreezing files.

Press RETURN to see the list of fields defined in the _Field file. Press **b** to select the `Browse` option. Press your arrow keys to move the highlight bar to the _Valexp field, and press RETURN. The Data Dictionary displays the definition for the _Valexp field:

```
PROGRESS Data Dictionary                                  Modify Existing File
MODIFY-SCHEMA  SQL  Database  Admin  Utilities  Reports  Exit
                              Currently Defined Fields
_Can-Read     _Can-Write    _Col-label    _Data-Type    _Decimals     _Desc
_dtype        _Extent       _Field-Name   _field-rpos   _File-recid   _Fld-case
_Fld-misc1    _Fld-misc2    _Fld-res1     _Fld-res2     _Fld-stdtype  _Fld-stlen
_Fld-stoff    _Format       _Help         _Initial      _Label        _Mandatory
_Order        _sys-field    _Valexp       _Valmsg

  Field-Name: _Valexp                        Data-Type: character
      Format: x(72)                             Extent:
       Label: Valexp                           Decimals: ?
Column-label: ?                                   Order: 17
     Initial: ?                               Mandatory: no   (Not Null)
Component of-> View: no    Index: no       Case-sensitive: no
Valexp: ?
        :
        :
        :
Valmsg:
  Help:
  Desc:

  NextPage  PrevPage  Add  Modify  Delete  Copy  GoIndex  SwitchFile
    Browse  Order  Undo  Exit                                Total Fields: 18
Database: demo (PROGRESS)                 File: _Field

Press space bar to continue.
```

There is a value for _valexp for every _Field record in the database and it contains the validation criteria for the field. To access that information for a particular field, you must access both the _File record for the file and the _Field record for the field, since there may be several fields with the same name.

In your applhelp.p procedure you can ask the user what field they want help with, or you can make your help "context-sensitive" by using the FRAME-FILE, FRAME-FIELD, FRAME-NAME, and FRAME-INDEX functions. Whenever the user presses HELP (F2) during data entry, the applhelp.p procedure uses those functions to identify the field the cursor was in. Using this information, the applhelp.p procedure can then access an external file, a database file of your own design, or the Data Dictionary files to retrieve the appropriate help information.

Here is a sample applhelp.p procedure that illustrates this technique:

t-aphlp.p

```
DEFINE VARIABLE fil AS CHARACTER FORMAT "x(12)" LABEL "File".
DEFINE VARIABLE fld AS CHARACTER FORMAT "x(12)" LABEL "Field".

fil = FRAME-FILE.
fld = FRAME-FIELD.

IF fil = "" THEN DO WITH FRAME nofld ROW 18 CENTERED:
  FORM SKIP(1)
   "Help information is not available for that position"
   WITH TITLE "Field Level Help".
  VIEW.
  RETURN.
END.

FIND _file WHERE _file._file-name = fil.
FIND _field OF _file WHERE _field._field-name = fld.

IF _field._valexp <> ?
THEN DISPLAY _field._valexp
        WITH FRAME fldinfo ROW 19 CENTERED NO-LABELS
        TITLE "VALIDATION CRITERIA FOR " + fil +"." + fld.
ELSE DISPLAY "none" @ _field._valexp with frame fldinfo.

HIDE FRAME nofld.
HIDE FRAME fldinfo.
```

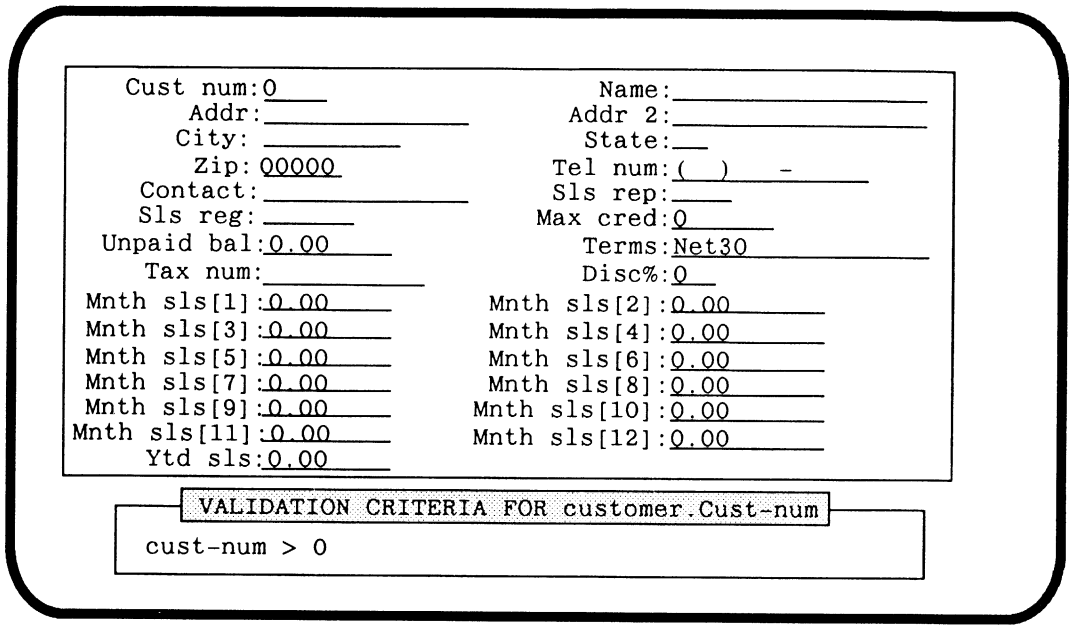
This procedure accesses the dictionary information and displays the validation criteria. This is only intended to illustrate how to access the schema. In practice you would probably not display validation expressions in an end-user application. You could use the FRAME-FILE and FRAME-FIELD functions to access records in a help database file that is part of the application database.

Copy the t-aphlp.p procedure into your edit area by pressing GET (F5). Then press PUT (F6) to save it under the name applhelp.p. Now run the following procedure:

t-ahtst.p

```
REPEAT:
  INSERT customer WITH 2 COLUMNS CENTERED.
END.
```

When the procedure prompts you for the customer number, press HELP (F2) and you see this display produced by the applhelp.p procedure:



Try doing this with the cursor in different fields, such as "Sls rep" and "Max cred," or even when it is not in a field.

15.3 SUMMARY

This chapter explained how to include information that helps users while they are running your application. It explained the different ways of providing application help information:

- Use the PROGRESS Data Dictionary to define help on a field by field basis.
- Write a help procedure that PROGRESS can run when the user presses HELP (F2).

We have illustrated several ways in which you can design a help procedure for an application.

- You can store the help information you provide in several places. Earlier in this section, we illustrated storing information in an external file. An alternate approach is to store the help information in one or more files in the database.
- You can use the FRAME-FILE, FRAME-FIELD, FRAME-NAME, and FRAME-INDEX functions to make a help procedure "context-sensitive" based on the field being entered when the user pressed HELP (F2).
- Alternatively, you might want to provide a very general help procedure that displays a menu and lets the user select specific help topics.

However, since you can use the full capabilities of PROGRESS to implement help procedures, you can provide whatever level and type of help you feel is appropriate to an application. As your skill with using PROGRESS grows, you will be able to apply the application development techniques you learn to build help systems that complement your application.

NOTE: Once you have finished using this chapter's sample applhelp.p procedure, you should erase this applhelp.p procedure from your directory so that it is not inadvertently used out of context.

Chapter 16
Preparing Your Application for Use

You've made it! You're ready to put your application into production use. This chapter discusses these considerations and explains different options for handling each. It covers the following topics:

- Providing access to your application.
- Writing a gateway procedure.
- Backing up your database.
- Final steps.
- Using the Developer's Toolkit to distribute applications.

As you take that final step, there are some things you should consider:

- You must decide how your users will access your application.
- Depending on the type of application you are packaging, you may want to write a "gateway" procedure to tie the individual procedures together for your users.
- You must make provisions for backing up the application database.

You may find that you want to use the PROGRESS Developer's Toolkit to help in the last stages of development. This chapter describes some of the utilities that are available in the Developer's Toolkit and how they can help you when packaging your application.

16.1 PROVIDING ACCESS TO YOUR APPLICATION

There are different methods you can use to give users access to your application. You can choose to have the user do one of the following:

- You can let the user start PROGRESS and run procedures from the editor the way that you have done when developing the application.
- You can let the user start PROGRESS with a start-up option to run your application's main procedure. With this method, the user doesn't have to run your application from the PROGRESS editor.
- You can supply a file that automatically invokes PROGRESS with the start-up options that you define.
- You may want to set up a "captive user" by running the application automatically when the user logs onto the system.

The option you choose depends completely on the kind of access you want the user to have to both the operating system and to the PROGRESS editor. Table 16–1 summarizes operating system and editor access for each option. Note that this table applies to PROGRESS 4GL/RDBMS and PROGRESS Query/Run-Time only; PROGRESS Run-Time does not provide access to the PROGRESS editor.

Table 16–1: Operating System and Editor Access

Option	Access to the Operating System	Access to the PROGRESS Editor
Start PROGRESS and run procedures from the editor.	✓	✓
Start PROGRESS with the -p or /STARTUP option to automatically run your application.	✓	
Invoke a script, batch file, or command procedure that contains the start-up command and options.	✓	
Log onto the system and automatically run the application.		

Although in some cases the user does not have immediate access to the PROGRESS editor, you can provide access to the editor from your application. Later sections in this chapter explain how to provide that access.

16.1.1 Running PROGRESS to Access an Application

Using this option means that the user types the same command to start PROGRESS that you've been typing while developing your application. The user types one of the following commands:

Table 16–2: Command to Start PROGRESS

Operating System	Starting PROGRESS
UNIX	pro *database–name*
DOS & OS/2	pro *database–name*
VMS	PROGRESS *database–name*
BTOS/CTOS	Progress 4GL [Options] -1 *database–name*

NOTE: For startup information for networking clients, see the *System Administration I: Environments*.

Once the user has started PROGRESS and is in the editor, the next step is to run one or more PROGRESS procedures. For example, the user might run a procedure that displays a main menu for the application and remain in the application from that point on.

This approach might be useful in situations where you want users to have access to the editor for writing queries or other kinds of procedures. However, it has the disadvantage of appearing unfriendly to beginning users. It also requires users to enter any necessary startup options on the `pro` or `PROGRESS` command line.

16.1.2 Using a Startup Procedure to Run Your Application

Rather than have end users run your application from the PROGRESS editor, let them start both PROGRESS and your application with the same command line. Specifically, supply an application startup file that starts PROGRESS with the Startup Procedure (-p) option. The -p startup option lets you name a PROGRESS procedure to run automatically at start up.

Table 16–3: Command to Start PROGRESS and an Application

Operating System	Specifying a Startup Procedure
UNIX	pro *database–name* -p *procedure–name*
DOS & OS/2	pro *database–name* -p *procedure–name*
VMS	PROGRESS/STARTUP=*procedure–name database–name*
BTOS/CTOS	PROGRESS 4GL ⋮ [Start-Up Procedure -p] *procedure–name* ⋮ [Options] -1 *database–name*

In these commands, *database–name* is the name of the database you want the user to use and *procedure–name* is the name of the start-up procedure that runs once PROGRESS has started. This start-up procedure is usually a main menu, or gateway procedure, from which the user can run other application procedures. For example:

```
pro personnel -p mainmenu.p
```

This command starts PROGRESS using the personnel database and, instead of putting the user into the editor, it runs the mainmenu.p procedure. For complete information on startup parameters and parameter files, see Chapter 3 in *System Administration II: General*.

16.1.3 Providing a Script, Batch File, or Command Procedure

This method is similar to the method covered in the previous section, but instead of requiring the user to enter the command, you put that command in a script (UNIX), batch file (DOS and OS/2), command procedure (VMS), or submit file (BTOS/CTOS). Then all the user has to do is type the name of that file. PROGRESS then starts against a predetermined database and with a predetermined set of start-up options.

Here is a sample of such a batch file called SAMPLE.BAT. The cd command ensures that the user is in the appropriate working directory. The pro command starts PROGRESS using the personnel database, and runs the mainmenu.p start-up procedure.

sample.bat

```
cd \personnel
pro personnel -p mainmenu.p
```

The user can run the application by simply running the batch file **sample.**

16.1.4 Setting Up a "Captive" User

Sometimes you want a user to have access only to an application and not to the operating system or to the editor. If you are using DOS, you can add a line to the user's AUTOEXEC.BAT file that calls a batch file like the one you saw in the previous section. Similarly, if you are using OS/2, you can add a line to the user's CONFIG.SYS file. In this case, the user automatically runs the application when the computer is turned on. When the application exits, the user returns to the operating system.

On UNIX, BTOS/CTOS, and VMS, you can set up a captive user so that they cannot access the operating system.

To set up a captive user on UNIX, put the following line in the user's .profile file:

```
exec pro database-name -p procedure-name
```

If the user is running multi-user PROGRESS, use the mpro command in this line instead of the pro command. In addition, be sure to start the multi-user server before a user tries to start multi-user PROGRESS. You can incorporate this step into separate scripts or you can always start the server whenever a user wants to run multi-user PROGRESS (that is, precede the mpro command by the proserve command in the script). If the server is already running, trying to start it again will have no effect and does no harm.

Under UNIX, this method also logs the user out upon leaving PROGRESS.

If you are using VMS:

- Create a command procedure that starts PROGRESS and your application. This procedure should contain:
  - Definitions for certain logical names.

- Use the ASSIGN or DEFINE command to assign SYS$INPUT to SYS$COMMAND so that VMS can look for commands from the command procedure and not from the terminal.

- The PROGRESS command and any start-up qualifiers necessary for your application.

- Put the following line in the user's LOGIN.COM file:

  $@*file-name*.com

  In this command, *file-name* is the name of the command procedure that starts PROGRESS and your application.

If you are using BTOS/CTOS, put the following lines in the user's ".user" file:

```
:SignonVolume:volume-name
:SignonDirectory:directory-name
:SignonFileprefix:
:SignonPassword:
:SignonChainFile:[sys]<dlc>PROGRESS.RUN
PROGRESS SINGLE USER
database-name
procedure-name
:SignonExitFile:[sys]<sys>SIGNON.RUN
:ProgressEnv:[sys]<sys>progress.env
```

NOTE: If the user executes a BTOS/CTOS command that runs a submit file, the user can access commands by pressing Action-Finish while the submit file is executing. Or, the user can execute a BTOS/CTOS command without any parameters.

16.2 WRITING A STARTUP PROCEDURE

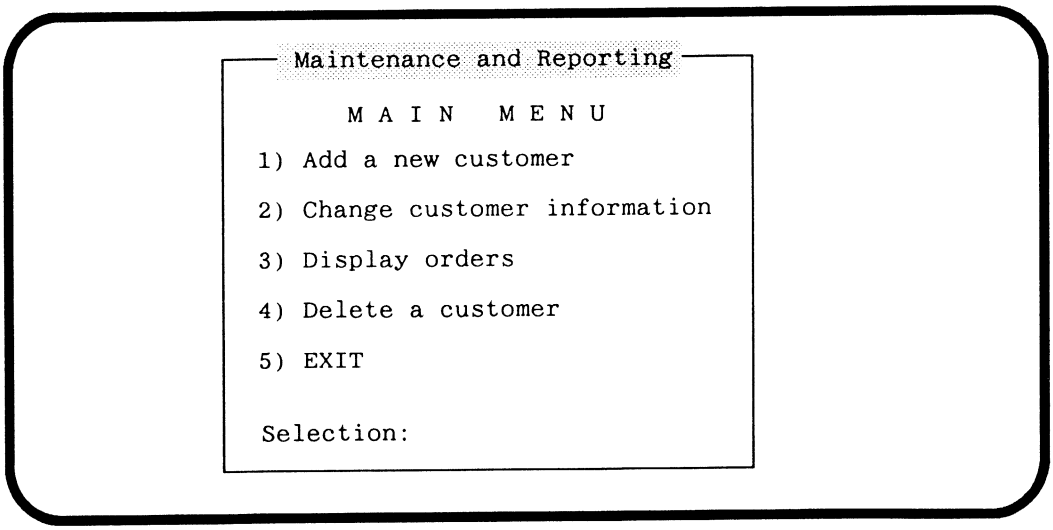
Regardless of which method you choose for user access to an application, you will need to write a "startup" procedure. A startup procedure is the first procedure a user runs (either by using the RUN command or automatically through the -p option or the /STARTUP qualifier) when entering PROGRESS. The startup procedure is usually an application main menu from which the user can run other application procedures.

A main menu procedure is much like any other menu procedure you have written. The only difference is that you need to think about the kind of access to the editor and to the operating system the procedure provides.

Here is one of the menu procedures used earlier in this manual:

t-csmenu.p

```
DEFINE VARIABLE Selection AS INTEGER FORMAT "9".
REPEAT:
  FORM
       SKIP(2) "      M A I N  M E N U              "
       SKIP(1) "  1) Add a new customer             "
       SKIP(1) "  2) Change customer information    "
       SKIP(1) "  3) Display orders                 "
       SKIP(1) "  4) Delete a customer              "
       SKIP(1) "  5) EXIT                           "
       WITH CENTERED
            TITLE "Maintenance and Reporting".
  UPDATE SKIP(2) SPACE(1) selection AUTO-RETURN
    WITH SIDE-LABELS.
  HIDE.
       IF selection EQ 1 THEN RUN t-adcust.p.
  ELSE IF selection EQ 2 THEN RUN t-chcust.p.
  ELSE IF selection EQ 3 THEN RUN t-itlist.p.
  ELSE IF selection EQ 4 THEN RUN t-delcus.p.
  ELSE IF selection EQ 5 THEN QUIT.
  ELSE MESSAGE "Incorrect selection - please try again".
END.
```



Suppose this is the main menu for your application. Let's look at each of the different kinds of access you might want to give the user in terms of the ways the user might have started the application.

16.2.1 Controlling Access for the pro or PROGRESS Command Users

Suppose the user starts the application by:

1. Using the pro or mpro command (on DOS, OS/2 or UNIX), the PROGRESS or PROGRESS/MULTI_USER=LOGIN command (on VMS), or the PROGRESS Single User or PROGRESS Multi-user command (on BTOS/CTOS) to start PROGRESS.

2. Running the application main menu from the editor.

Once you have let a user access your application in this way, that user has access to the editor, operating system and to the menus of the application. Because this user already has fairly unlimited access, there is no real need to do anything special with your gateway procedure.

NOTE: Access is controlled by the password protecting files in BTOS/CTOS.

16.2.2 Controlling Access for -p, /STARTUP, and "Captive" Users

Now suppose you have decided to:

- Have DOS, OS/2 and UNIX users type the pro or mpro command along with the -p option and the name of your start-up menu procedure.

  Have BTOS/CTOS users type the PROGRESS Single User or PROGRESS Multi-User command then supply a startup procedure name at the "Startup Procedure" option.

  Have VMS users type the PROGRESS or PROGRESS/MULTI_USER=LOGIN command along with the /STARTUP qualifier and the name of your start-up menu procedure.

OR

- Have users type the name of a script, batch file, command procedure, or submit file.

OR

- Put users directly into the application when they log in.

In the first case, users already have access to the operating system and operating system access is not an issue for "captive" users. But, how can you control access to the editor?

First, let's assume you want the user to have access to the editor.

Go ahead and run the t-csmenu.p procedure. When the procedure prompts you for a selection, press END-ERROR (F4). The procedure puts you in the editor. Why? Remember the default PROGRESS action for the END-ERROR key:

- If you press END-ERROR (F4) on the first user interaction in a block, PROGRESS takes the default ENDKEY action which is UNDO, LEAVE.
- If you press END-ERROR (F4) after the first user interaction in a block, PROGRESS takes the default ERROR action which is UNDO, RETRY.

When you press END-ERROR (F4) in response to the selection prompt, you are on the first screen interaction of the block. So PROGRESS undoes the work you've done (if any), and leaves the block. Since there are no more statements in the procedure, it ends. PROGRESS returns you to the editor when the procedure you run from the editor ends or when the procedure named in a start-up option ends. Once in the editor, the user can use the RUN statement to access the application main menu again.

Now, suppose you don't want the user to have access to the editor. To accomplish this, simply add an ON ENDKEY UNDO, RETRY phrase to the REPEAT block of the gateway procedure.

t-csmen2.p

```
DEFINE VARIABLE selection AS INTEGER FORMAT "9".

REPEAT ON ENDKEY UNDO, RETRY:
  FORM
      SKIP(2) "      M A I N  M E N U              "
      SKIP(1) "  1) Add a new customer             "
      SKIP(1) "  2) Change customer information    "
      SKIP(1) "  3) Display orders                 "
      SKIP(1) "  4) Delete a customer              "
      SKIP(1) "  5) EXIT                           "
      WITH CENTERED
           TITLE "Maintenance and Reporting".
  UPDATE SKIP(2) SPACE(1) selection AUTO-RETURN
    WITH SIDE-LABELS.
  HIDE.
       IF selection EQ 1 THEN RUN t-adcust.p.
  ELSE IF selection EQ 2 THEN RUN t-chcust.p.
  ELSE IF selection EQ 3 THEN RUN t-itlist.p.
  ELSE IF selection EQ 4 THEN RUN t-delcus.p.
  ELSE IF selection EQ 5 THEN QUIT.
  ELSE MESSAGE "Incorrect selection - please try again".
END.
```

Try running the t-csmen2.p procedure. Press END-ERROR (F4) in response to the selection prompt. Notice that the procedure no longer puts you in the editor. Instead, you remain at the selection prompt. By using the ON ENDKEY UNDO, RETRY phrase, you override the default action of UNDO, LEAVE.

Although you did not want users to have access to the editor, you did want them to be able to exit from the application. That is why the procedure includes an option (5) that runs the QUIT statement.

16.2.3 How Progress Handles the Stop Key

The way PROGRESS treats the STOP key depends on where in the application the user is when pressing that key:

- If the user is either directly in a start-up menu procedure or is running a procedure called by the start-up procedure, PROGRESS undoes the current transaction and restarts the start-up procedure. If you write your start-up procedure so that the user cannot get access to the editor with the END-ERROR (F4) key, then that user cannot access the editor with the STOP key either.

- If the user has left the application to work in the editor (assuming you allowed the user access to the editor), and then returned to running application procedures, pressing STOP returns the user to the editor.

If you run the t-csmen2.p procedure in the previous section, you can return to the editor by pressing STOP.

16.3 BACKING UP YOUR DATABASE

You should back up your application database as frequently as is necessary (usually on a daily basis) to ensure that you do not lose data due to a system failure. To back up a PROGRESS database, copy all of the files with the base name of your database (e.g. demo.*). You should never backup just the .db file. Always backup all related files as a group (.db, .bi, etc.).

NOTE: BE CAREFUL NOT TO BACKUP A DATABASE THAT IS IN USE. The exception is for performing online backups for multi-user databases. For more information about performing online backups, refer to Chapter 4 of the *System Administration II: General.*

A database is "in use" if a server is active, even if no users are currently using the database, or if the database is being used in single-user mode. Your backup scripts should use the proutil command described in Chapter 3 of the *System Administration II: General*. The proutil command should include the BUSY option to determine if the database is in use. If it is in use, do not perform the backup.

For extra protection, you should periodically create ASCII dumps of your databases using the PROGRESS Data Dictionary dump routine. You use this routine by choosing `Admin` from the Dictionary Main Menu and then choosing option D, Dump Data and Definitions, from the Admin menu. From the submenu, choose option D, to dump the Data Definitions (.df file). However, before dumping your database data, you should also dump the field and index definitions for each file in the database (first choose Admin from the Dictionary Main Menu, then choose option D from the Admin menu, and finally choose option F to dump the File Contents (.d files).

16.4 FINAL STEPS

Here is a checklist of the steps you should take when you are ready to put your application into use:

1. Choose a method for writing your start-up menu procedures.

2. Create a "basic" database. A basic database contains the schema for your application and perhaps some control data but is otherwise empty. You then store this basic database in a master directory so that you can make copies of it whenever necessary. To create a basic database:

   - From your development database, use option D, Dump Data and Definitions, from the Admin option of the Data Dictionary Main Menu. From the Dump Data and Definitions menu, choose option D to dump your data definitions. If there is control data in any files, use option F to dump the contents of those files. If you have stored any records in the _User file, choose option U to dump the contents of the _User file.

   - Create a new database from the empty database.

Table 16–4: Command to Create a Database

Operating System	To Copy The Empty Database
UNIX, DOS, and OS/2	prodb *database–name* empty
VMS	PROGRESS/CREATE *database–name* empty
BTOS/CTOS	PROGRESS Create Database New Database Name *database–name* Copy From Database Name empty

- In the new database (the "basic database"), use the L option, Load Data and Definitions, from the Admin option of the Data Dictionary menu. From the Load Data and Definitions menu, choose option D to load the dumped data definitions into the basic database. Use option F to load any data file contents, and option U to load any _User file contents.

3. Use the COMPILE SAVE statement to precompile your application procedures against the basic database.

4. Store the source versions of your application procedures, the object versions of your procedures, and the basic database in a separate directory and be sure to exercise control over changes to that directory. Also be sure to first test any changes in another directory before incorporating those changes in the master version.

   If you want to provide only object versions of procedures to users, you must be sure that you compile the procedures using a copy of the basic database that is identical to the one distributed with the application.

   When you distribute object files to users, you can place the files in a PROGRESS library. Placing object files in a library increases the overall performance of your application. To read more about placing object files in a library, see "Building Libraries with the Prolib Utility" in Chapter 4 of the *System Admininstration II: General*.

5. Whenever you need a fresh copy of the database, use the prodb command to copy the basic database from the master directory.

6. If you add or delete file, field, or index definitions in the basic database, you must recompile all procedures which access the file affected by the change against the modified database. Whenever you make these kinds of changes to the database, PROGRESS puts a new time stamp on the files that have changed. All procedures you compile against that database then contain that time stamp in their object files. The time

stamp of an object file must match the time stamp of the files referenced in the procedures against which you are trying to run that file.

If you want to provide Query/Run-Time or Run-Time users with the new database and the revised application, those users will have to dump the data from the old database and reload it into the new. This and other related considerations are covered in detail in the documentation that accompanies the Developer's Toolkit product.

7. If you make any other changes to the basic database, those changes are not reflected in procedures until you recompile those procedures against the modified basic database.

16.5 USING THE DEVELOPER'S TOOLKIT TO DISTRIBUTE APPLICATIONS

When you develop applications you plan to distribute to end users, you typically make several decisions: Do you want end users to be able to modify your application procedures and possibly add some of their own? Do you want end users to be able to access your database definitions and perhaps modify them?

Then there are the additional tools you have to supply along with your application: a facility to dump and reload the database (if the user has PROGRESS Query/Run-Time or PROGRESS Run-Time), batch files or scripts to start PROGRESS and access your applications, and so on.

The PROGRESS Developer's Toolkit consists of a complete set of tools to help you address these and other needs:

- dbrstrct Utility

  Use this utility to specify the kinds of access end users have to an application database.

- mkdump Utility

  You must supply a dump/reload facility to Query/Run-Time or Run-Time users. Use the mkdump Utility to create that facility.

- procomp Utility

  Use this utility to compile your procedures on a machine other than your development machine. By doing this compilation, users can run your application on a machine other than the machine on which you developed your application.

- tailor (UNIX only) Utility

You will need to make global changes to the values of certain variable definitions used in the Developer's Toolkit scripts. In addition, you may want to copy all scripts to /usr/bin. You run tailor to make these changes.

- xcode Utility

  Use this utility to encrypt source code for shipment to customer sites (where it can be compiled by a special version of the Developer's Toolkit compiler, procomp).

- Empty database

  You may want to make changes to the metaschema of the empty database supplied with PROGRESS. If so, you make those changes to the empty database supplied with the Developer's Toolkit.

- Templates of scripts to provide to users.

16.6 SUMMARY

This chapter showed how to pull all of your procedures together and present your application to your users. Specifically, it explained:

- Different methods for providing application access to end users.
- How to write start-up procedures.
- Database backup rules to remember.

This chapter also included a checklist of the steps you should take before putting your application into production use. The checklist is in the section called "Final Steps."

Appendix A
Demo Database Files, Fields, and Indexes

The figure below shows:

- The four main demo database files: customer, order, order-line, and item.
- The indexes for each file.
- How PROGRESS uses the indexes to relate the files to one another. The small "tabs" below each "file folder" show which fields are in each index for the file. For example, the three indexes for the customer file are based on the cust-num, name, and zip fields.

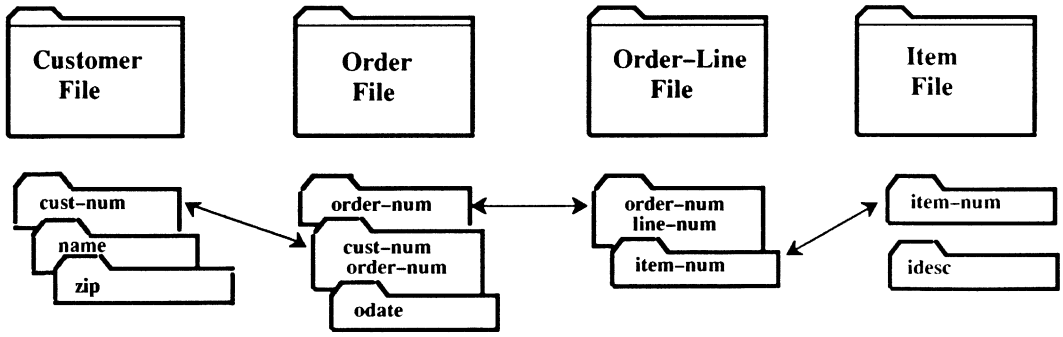


Figure A-1: Files and Indexes in the Demo Database

The following tables list the fields in the customer, order,order-line, and item files. To see a complete listing of *all* the files in the demo database, either:

- Choose option 2, Print Data Dictionary Reports, from the Data Dictionary Main Menu.

OR

- Choose option 1, List, from the Display/Change Data Definitions menu to list fields for specific files.

Table A-1: Customer Fields, Data Types, and Field Descriptions

Field	Data Type	Description
Cust-num	Integer	Customer number
Name	Character	Customer name
Address	Character	First line of customer address
Address2	Character	Second line of customer address
City	Character	City
St	Character	State
Zip	Integer	Zip code
Phone	Character	Phone Number
Contact	Character	Contact at customer site
Sales-rep	Character	Sales rep for the account
Sales-region	Character	Sales region in which account is located
Max-credit	Decimal	Customer's maximum credit allowance
Curr-bal	Decimal	Customer's current balance due
Terms	Character	Payment terms
Tax-no	Character	Tax exempt number
Discount	Integer	Customer discount percentage
Mnth-sales	Decimal	Sales in each month
Ytd-sales	Decimal	Sales year-to-date

Table A–2: Order Fields, Data Types, and Field Descriptions

Field	Data Type	Description
Order-num	Integer	Order number
Cust-num	Integer	Customer number
Name	Character	Ship to name
Address	Character	First line of address to which order is to be shipped
Address2	Character	Second line of shipping address
City	Character	City
St	Character	State
Zip	Integer	Zip
Odate	Date	Date of order
Sdate	Date	Shipping date
Pdate	Date	Promised delivery date
Shp-via	Character	Shipping service used
Misc-info	Character	Miscellaneous information
Cust-po	Character	Customer purchase order number
Terms	Character	Payment terms
Sales-rep	Character	Sales rep for the account
Shipped	Character	Shipping flag

Table A–3: Order–Line Fields, Data Types, and Field Descriptions

Field	Data Type	Description
Order-num	Integer	Order number
Line-num	Integer	Line number
Item-num	Integer	Item number
Price	Decimal	Item price
Qty	Integer	Item quantity ordered
Qty-shp	Integer	Quantity of an item shipped
Disc	Integer	Discount applied

Table A–4: Item Fields, Data Types, and Field Descriptions

Field	Data Type	Description
Item-num	Integer	Item number
Idesc	Character	Item description
Subs-item	Integer	Substitute item number
Cost	Decimal	Item cost
Loc	Character	Inventory location
Prod-line	Character	Product line to which an item belongs
On-hand	Integer	Quantity in stock
Alloc	Integer	Quantity allocated
Rop	Integer	Reorder point
Oorder	Intger	Quantity on order
Iweight	Integer	Item weight
Mnth-shp	Integer	Quantity shipped each month

Glossary

active file — the file that is currently in use.

active database — the database that is currently in use. If you have more than one database, the Data Dictionary requires you to select an active, working database.

activities file (also called a permissions file) — a file that lists the users authorized to run each procedure or group of procedures. An application can check the file to see that the current userid is in the list of authorized users for that procedure.

aggregate phrase — a PROGRESS language element that identifies one or more values to be calculated based on a change in a break group. For example, each time the break group (sales-rep for example) changes, the TOTAL BY option calculates the total unpaid balances for all customers belonging to the last sales rep. It also provides a grand total of the unpaid balances for all delinquent customer accounts for all sales-reps at the end of the report. TOTAL BY is an aggregate-phrase option.

application — a set of programming language instructions that accomplish a specific task. An application can be created from PROGRESS procedures.

argument — a value supplied to a statement or function.

array — a field or variable with multiple elements. Each element in an array has the same data type.

array extent — the number of elements contained in the array.

ASCII character — a seven-bit code representing upper and lower case letters, numbers, symbols, and punctuation marks. ASCII is an acronym for the American Standard Code for Information Interchange.

auto-connect list — a list of PROGRESS databases (with connection parameters) that connect automatically as required during program execution.

basic database — a database that contains the schema for your application and perhaps some control data but is otherwise empty. The basic database is stored in a master directory and used to make copies whenever necessary.

block — a series of statements that PROGRESS treats as a single unit. Each block begins with a block header statement and concludes with an END. PROGRESS uses four kinds of blocks: FOR EACH, REPEAT, DO, and procedure. A block can have one or more of the following properties: looping, record reading, clearing, transaction recovery, error processing, default screen design. Also called a procedure block.

block header — the statement that begins a block. It is different from other kinds of statements in two ways: it ends with a colon (:) (all other statements end with a period) and it can have a label (the label also ends with a colon).

block label — text in a procedure block that identifies the block.

break-group — a set of records having a common value in a certain database field. Break-groups are used on reports to display file and record relationships.

buffer — a small amount of memory used as a temporary storage area for data. See also **data buffer.**

captive user — a user that cannot access the operating system on exit from a PROGRESS session. A captive user has a startup file that starts a PROGRESS procedure at login time and logs the user off the operating system at the completion of the PROGRESS procedure, preventing access to the operating system.

character constant — a value made up of non-numeric or character data, or a combination of numeric and non-numeric data that remains unchanged during a procedure. Character constants must be enclosed in quotation marks when used in a procedure.

character field — a field having a character data type. An alphanumeric field that cannot be arithmetically manipulated. Although PROGRESS allows character fields of up to 255 characters, you will usually want to restrict the format length of a character field to the input/output line width of your terminal (typically 80 characters).

column — a component of a record, a field.

column label — a label displayed above a column of data (field values).

command — an instruction to the system.

comparison expression — a combination of constants, variables, operators, and parentheses used to compare values.

compiled procedure — see **session compile** and **precompiled procedure.**

concatenation operator — the plus (+) symbol used to link or join two or more character strings into a single character string.

conditional processing — a means of processing (compilation, for example) based on one or more logical expressions.

connect parameters — a subset of the large number of PROGRESS startup parameters that are relevant during a database connect operation.

connected database — a database that is accessible and can be worked on in the Data Dictionary. Connecting a database is roughly equivalent to starting it.

connection mode — one of three possible ways a database is connected: single user, multi-user client, or multi-user direct access.

constant — a value that remains unchanged during the execution of a program.

data buffer — PROGRESS uses two types of data buffers: the *record buffer*, which is a temporary storage area for a record, field, or variable; and the *screen buffer*, which is a display area for a field, variable, or the result of a calculation.

data definitions — the characteristics of the files, fields, and indexes that comprise the schema of a PROGRESS database. The structure of a given database. See also **data dictionary.**

Data Dictionary — This term has two meanings. When capitalized, it refers to the menu-driven utility program. In lower case, data dictionary is a synonym for schema; that is, it refers to the actual database structure definitions. Typically in PROGRESS, Data Dictionary refers to the PROGRESS program, while the terms schema and data definitions are used to refer to the database structure.

data integrity — certainty of data accuracy or validity.

data type — a property of a field or variable that determines the nature of data that can be stored there (integers or characters, for example). PROGRESS supports five data types: character, integer, decimal, date, and logical.

database — a collection of logically related records or files that can be accessed or retrieved.

date field — a field having a date data type. A date field can contain dates from 1/1/32768 BC through 12/31/32767 AD. You can specify dates in this century with either a two-digit year, such as 8/9/90, or a four-digit year (8/9/1990). Dates in other centuries require a four-digit year.

decimal field — a field having a decimal data type. A decimal field can contain decimal numbers up to 50 digits in length. PROGRESS allows up to 10 digits to the right of the decimal point.

demo database — a sample PROGRESS database containing files, fields, and indexes (including some data) that procedures in PROGRESS documentation use to acquaint you with the way PROGRESS operates.

device — a combination of physical components forming a unit that performs a specific set of input/output functions (a terminal or a disk drive, for example).

directory — a file system object that lists or contains files and, possibly, other directories.

display format — the way data appears on screens and in printed reports. PROGRESS automatically supplies a default display format for each data type, but you can change that default format. You can specify display formats in the Data Dictionary, or with the FORMAT option. See also **format phrase.**

DLC directory — the full pathname of the volume and directory that contains PROGRESS.

DLC variable — the environment variable that points to the directory containing the PROGRESS system software. The default installation directory is `/dlc` on DOS and OS/2 systems; `/usr/dlc` on UNIX systems; `[DLC]` on VMS systems; and `[sys]<dlc>` on BTOS/CTOS systems.

down frame — a frame that displays multiple records, one per line, one after another. See also **frame**.

dump name — a unique name assigned to each database file. PROGRESS uses the dump name if you dump the file's data. Dumped data files have a `.d` extension.

edit area — the area between the two horizontal lines in the PROGRESS screen in which you enter and edit PROGRESS code.

editor — see PROGRESS editor.

empty database — a database with no data definitions or data, having only the underlying PROGRESS system information.

escape character — a character or symbol in a stream that is used to signal the system that one or more of the following characters are to be interpreted in a special way.

expression — in a program, a combination of constants, variables, operations, and parentheses used to perform a desired computation. An expression may consist of anything from a single constant or variable to the most complicated arrangement of operators and functions that fit into a single program statement.

extent — 1) the number of elements in an array field or variable. 2) one of the volumes in a multi-volume database.

field — a component of a record that holds a data value. Also called a column.

file — a collection of logically related records treated as a unit. A file can contain a program, data, or text. For example, a payroll file is a collection of employee payroll records. Also called a table.

footer — text, such as a page number, placed on the bottom of each page of a report.

format phrase — a PROGRESS language description of the display characteristics for a field, variable, or expression.

frame — visible or invisible boxes or windows that PROGRESS uses to display data.

frozen file — a file whose field and index definitions cannot be changed. The record data is not frozen—just the data definitions. Typically, database files are frozen when application development is completed.

function — a process that generates a value. In PROGRESS, functions are shortcuts to handling tasks as diverse as calculating the logarithm of an expression and determining the day of the week a particular date falls. There are many types of functions built into the PROGRESS language. For example, **arithmetic** functions (like RANDOM or EXP) perform mathematical operations on numeric values; **character** functions (such as LENGTH and SUBSTRING) manipulate character strings or expressions; **date** functions (TODAY or MONTH, for example) provide day, month, and year information for an application.

function keys — special keys to which PROGRESS assigns certain actions. In procedures, you can reassign these actions to different keys.

global shared variable — a variable that is available to all parts of a multi-part program and has the same value wherever it is used in the program.

header — text, such as a page number, title, or date, placed on the top of each page of a report.

horizontal menu — a menu whose options appear in a horizontal line across the screen.

identifier — the name that you give a file, field, or variable. Identifiers can be up to 32 characters in length. The first character must be an uppercase or lowercase letter. The remainder of the identifier can be any combination of letters, digits, and underscore (_) characters. You cannot use any of the SQL special characters (except for the underscore character) or reserved words (except ORDER) as a part of an identifier. Identifiers are never case-sensitive.

include file — a separate file containing PROGRESS code that you can call from other procedures by placing the file's name in braces within the procedure ({`myincl.i`}, for example).

index — a directory or table containing 1) the field or fields identifying the records in a file, and 2) the locations where the records are stored.

initial value — the value to which the field is set when created.

integer field — a field having an integer data type. An integer field can contain positive or negative whole numbers, ranging in value from -2,147,483,648 through 2,147,483,647.

integrity — accuracy or completeness of data.

iteration — each repetition of a procedure block.

label — text that appears with a field or variable when it is displayed.

literal — an alphanumeric constant. Literals can be character strings, numerics, dates, or logical data.

local variable — a temporary field for storing data. Data in a local variable is available only within the procedure in which it is defined.

logical constant — a constant having the value yes, true, no, or false.

logical database name — the name of a physically connected database. The logical database name is used to resolve ambiguous database references. That is, when a procedure is compiled against a database, it is the logical database name that is stored in the procedure's object code, and when a procedure executes, its database name references must match the logical name of a connected database.

logical expression — an expression that, when evaluated, yields true or false.

logical field — a field having a logical data type. A logical field can contain one of two values (from yes/no, true/false, or a logical value pair you define).

logical operator — an operator such as AND, OR, or NOT used in an expression that yields a true or false value.

mandatory field — a field that is mandatory must be filled in and cannot have an unknown or null value in it. It can, however, have a blank as its value.

menu — a list of options (displayed on the screen) from which a user can choose.

message area of screen — three lines at the bottom of a PROGRESS screen. The first two lines are for procedure-specific messages. The third line is for PROGRESS system messages and help messages.

metaschema — internal database files that define the underlying structure of a PROGRESS database. Metaschema file names begin with the underscore (_) character.

multi-volume database — a database having multiple data segments, usually stored on separate disks.

nested blocks — procedure blocks contained in other procedure blocks.

network file server — in a network, a computer that manages file sharing and system security, coordinates station-to-station communications, and controls any attached peripherals such as printers, disk drives, modems, and plotters.

node — in a network, a computer or other device.

non-spacetaking terminal — a terminal that does not reserve character positions for special screen display attributes (highlighting and underlining, for example).

null frame — a frame that does not appear on the screen because nothing is displayed by the block that defines it.

null value — an SQL term that indicates that the field or variable data is unknown, uninitialized, or unavailable.

object file — the compiled version of a PROGRESS source file (.p). See also **precompiled file** and **session compile**.

operating system — software that controls and manages the execution of computer programs and services.

operator — the symbol you use to perform numeric calculations, date calculations, character string manipulations, or data comparisons (+, /, and GT, for example).

output destination — a file or device to which a program sends output. Output destinations can include a terminal, a printer, an ASCII file, or a printer queue.

overlay frame — a frame that PROGRESS displays on top of any other frames that overlap its display area.

parameter — a variable or constant passed to or from a subroutine and the main program. Also, a PROGRESS startup option.

parameter file — an ASCII file containing PROGRESS startup options (parameters). You use parameter files to store the appropriate startup/connect parameters for a particular database, group of users, or system configuration. Rather than supplying startup or CONNECT options explicitly on the `pro` or `PROGRESS` command line or in a CONNECT statement, you can use a parameter file.

password — a string that is known only to the user. At startup, password is used in conjunction with the userid to ensure that only known users can start the application. All PROGRESS passwords are case-insensitive and may as many as 16-characters long.

pathname — specifies the complete name of a directory or file by starting at the root directory or disk volume and tracing the hierarchy of the file.

permissions file (also called an activities file) — see **activities file.**

physical database name — the actual name of the database on a disk.

primary index — usually the most frequently used index. PROGRESS allows you to set one index as primary and uses it by default when searching for a record.

precompiled procedure — an object version (`.r` extension) of a PROGRESS procedure generated when the source version (`.p` extension) is compiled with the SAVE option.

procedure — the steps taken to perform a task. A series of programming language commands and statements that perform a desired data processing task. The procedures referred to in this book contain PROGRESS commands and statements.

procedure block — a series of statements that PROGRESS treats as a single unit. Each block begins with a block header statement and concludes with an END. PROGRESS uses four kinds of blocks: FOR EACH, REPEAT, DO, and procedure. A block can have one or more of the following properties: looping, record reading, clearing, transaction recovery, error processing, default screen design. Also called a block.

procedure editor — see **PROGRESS editor.**

PROGRESS command — the command you use to start PROGRESS. The PROGRESS command is followed by the name of the database you want to connect.

PROGRESS editor — a writing tool that allows you to create and edit PROGRESS procedures. You can also compile and test PROGRESS procedures from the PROGRESS editor.

PROGRESS session — a PROGRESS session begins when you start PROGRESS with the pro or PROGRESS command, and ends when you run the QUIT statement.

qualification — a search criteria that limits the data displayed on a report or in a query from a database. A qualification typically consists of a field name, a comparison operator, and a field name or constant. In complex qualifications, logical operators are used to join several qualifications.

query — to ask for or seek information in a database.

record — a collection of related items of data. Also called a row.

record buffer — a temporary storage area in data memory for a record, field, or variable. When you write a record to the database, PROGRESS gets that record from the record buffer.

record scope — the sequence or block of procedure code during which PROGRESS holds a particular record in the record buffer.

referential integrity — the ensured consistency of file relationships.

related files — files that share information.

relational database — data stored in one or more files such that one query can access information from one or more files.

relational operator — a symbol such as = (equal to), < (less than), or > (greater than) that is used to compare two values. It specifies a condition that can be either true or false.

report — an organized display of data from a database.

report header — text, such as a page number, title, or date placed on the top of each page of a report.

remote — a terminal that is: 1) on a different node in a network, or 2) connected via telephone lines or other communication links.

remote database — a database located on a remote machine that is networked to the machine containing the application session.

row — a collection of related items. Also called a record.

schema — a description of a database's structure—the files it contains, the fields within the files, views, etc. In addition to database structure, PROGRESS database schemas contain items such as validation expression and validation message. Also called the **data definitions**.

schema-holder — a PROGRESS database that contains the data definitions for one or more PROGRESS or non-PROGRESS databases.

screen buffer — a display area for a field, variable, or the result of a calculation. When you prompt for information or display information for the user, PROGRESS places that information in the screen buffer.

server — a master process that coordinates multi-user database access.

session compile — the active object version of a procedure for the current session. The session compile version of a procedure can be either a precompiled .r version, or a version compiled without the SAVE option (for example, by pressing GO (F1) from the PROGRESS editor).

shared variable — a variable used to pass information from one procedure to another.

side label — a label displayed immediately left of a field or variable value.

source code — the original version of a program, before it is compiled. In PROGRESS, source files have the .p extension. Also called a source procedure.

spacetaking terminal — a terminal that reserves a character position on both sides of every field for special screen field attributes, such as underlining or highlighting.

startup command — the PROGRESS or pro command.

startup options — parameters that tailor a PROGRESS session or database connection.

statement — an instruction written in the PROGRESS language.

stream — a sequence of characters or items.

string — a character expression (a constant, field name, variable name, or any combination of these that results in a character string value).

subprocedure — a procedure that is called from another procedure.

table — a collection of information organized into named columns. The following SQL statements define, modify, and delete tables: CREATE TABLE, ALTER TABLE, and DROP TABLE. Also called a file.

time stamp — an internal PROGRESS file marker that indicates the date and time of the most recent modifications to a database definition.

unique index — an indexed field where every index key must be different. For example, Social Security number could be a unique index.

userid — user identification. At startup, userid is used in conjunction with the password to ensure that only known users can start the application. At runtime, the application can check the userid that was established at startup to be sure that only authorized users run individual procedures. In PROGRESS, a userid is a 32-character string. Like file names, userids must begin with a character from a-z or from A-Z. The name can consist of alphabetic characters, digits, and the characters #,$,%,&, -, and _. Userids are not case-sensitive; they can be uppercase, lowercase, or any combination of these. A userid can be blank, written as the string " ", but it cannot be defined as such through the Data Dictionary.

unknown value — a special data value (represented by a question mark (?)), which indicates that the field or variable data is unknown, uninitialized, or unavailable.

utility — a program or subroutine designed to help perform a task.

validation expression — a test to make sure that the user does not enter invalid data in a field.

variable — a temporary field for storing data.

version file — a file that is displayed when PROGRESS starts that the users which version of PROGRESS they are running.

vertical menu — a menu style in which the menu options are lined up one above the other.

view — a "window" into one or more tables. To the user, a view looks like a table, but it does not exist as such. A view does not represent physical data of its own, instead it represents the data in the table or tables used to define it. The following SQL statements define and delete views: CREATE VIEW and DROP VIEW.

working directory — the directory you are in when you issue a command.

Index

Symbols

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X