CS 9B Study Guide Table of Contents

Topic ... page
Introduction ... 1
Structure of quizzes and programs in CS 9B ... 2
Orientation ... 3
Quiz—Program comprehension ... 4
Program—Flow-of-control project ... 5
Quiz—Arrays ... 18
Quiz—Records ... 19
Program—Exercises with arrays and records ... 20
Quiz—Recursion ... 25
Program—Recursion exercises ... 26
Quiz—Pointers ... 34
Program—Pointer exercises ... 35
Sample quiz questions ... 47
Solutions to sample quiz questions ... 50
Running Pascal programs ... 53
Aspects of good programming style ... 54
Illustration of Pascal's parameter mechanisms ... 59
dbx manual pages ... 62
Debugging with dbx ... 67
Emacs reference ... 73
Notes from tutoring sessions ... 88

© 2002 University of California

Introduction

In CS 9B, you learn to program in Pascal. Course material consists of quizzes, which test your knowledge of language and low-level conceptual details, and programming assignments, which exercise your overall command of the language. This volume supplies a framework for the course. It contains the following:

Study guides. Each study guide focuses on a particular programming topic. It provides references to textbook material describing the topic, and suggests exercises for self-study. The study guides reference the following text.

Condensed Pascal, D. Cooper (Norton, 1987).

Programming assignments. Each one has a header page (this tells you the title and related topics) that is followed by the actual assignment.

System information. Some programming assignments require you to use features of Pascal not adequately described in the standard documents. More information is included with the corresponding study guide.

Sample quiz questions, with solutions. These help you prepare for the quizzes.

Structure of quizzes and programs in CS 9B

The following table outlines the relationship between quizzes and programs. All the material for a particular grouping must be completed before material in the next grouping; however, quizzes and programs within a group may be done in any order.

group	*quizzes*	*programs*
A	Program comprehension	Orientation
		Flow-of-control project
B	Arrays	Array and record exercises
	Records	
C	Recursion exercises	Recursion exercises
	Pointer exercises	Pointer exercises

Note that this breakdown is different from what's required to satisfy course deadlines. For information about deadlines, consult the "Information and Regulations" document.

Orientation

In this activity, you familiarize yourself with software to compute deadline penalties in the self-paced courses and supply us with some administrative information that makes it easier to contact you.

Readings

The "Information and Regulations" pamphlet.

Problem

The "Orientation" assignment will be distributed at the Self-Paced Center. You must complete it to enable any of your other work to be recorded. Complete it as early in the semester as possible.

Quiz—Program comprehension

You learn the “art” of using Pascal not only by writing programs of your own, but also by reading those of others. Quiz questions ask you to understand, debug, and criticize an already-written program; thus, this topic might be called “program appreciation”. There is a lot of reading for this first topic, so get started right away. It will cover most of the Pascal syntax you will learn: input/output, loops, procedures, and functions.

Readings

Pascal readings

Chapters 1 through 6 in Cooper, or sections on loops, testing, procedures, and functions in whatever Pascal text you are using.

The section "Illustration of Pascal's parameter mechanisms" later in this document.

Exercises

Sample quiz questions appear at the back of this document.

Program—Flow-of-control project

This program introduces you to Pascal by giving you practice with procedures, functions, and the various flow-of-control constructs.

Related quizzes

Program comprehension.

Programming assignment

Do *one* of the problems described on the following pages.

Readings

Pascal readings

Chapters 1 through 6 in Cooper, or sections on loops, testing, procedures, and functions in whatever Pascal text you are using.

Study Guide (this document)

"Aspects of good programming style", and "Illustration of Pascal's parameter mechanisms" later in this document.

If you're using EECS instructional computers, also read the sections "Running Pascal programs", "dbx manual pages" and "Debugging with dbx".

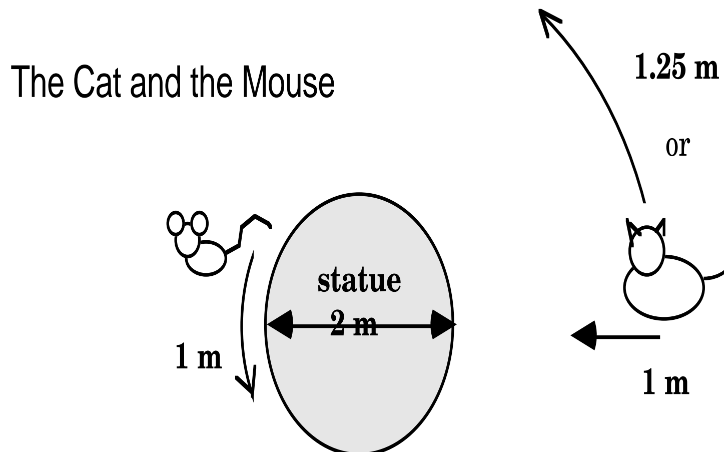
The document "Before You Begin …" available online in the self-paced course web site (for students who are not familiar with EECS computing facilities).

We don't care what editor you use. The section "Emacs reference" contains a brief summary of commands for the emacs editor.

CS 9B programming assignment
Flow of control project A

Background

The scene is an urban park, where a cat watches a mouse run around the base of a statue of Klaus Wirth. Over the course of a minute, the somewhat witless mouse moves one meter counterclockwise around the statue's base, which is circular and two meters in diameter. Every sixty seconds, the cat pursues the mouse as follows. If the cat can see the mouse, the cat moves one meter in toward the statue. If the cat can't see the mouse, the cat circles 1.25 meters counterclockwise around the statue. The cat plans eventually to get close enough to the mouse to make it a juicy lunch.



Write a program to keep track of the cat's pursuit of the mouse. Your main program should contain statements to input and check the initial positions of the cat and mouse; within the program, all positions should be represented in polar coordinates, and the mouse's radius should be assumed to be one meter. It should also contain a loop, each repetition of which represents the passage of one minute, and which contains calls to procedures to move the mouse and the cat.

Your program should contain, within its loop, statements to print the new positions of the cat and mouse after each minute, and to keep track of how much time the cat is taking to catch the mouse. The output should be in the form of a table for maximum readability. (Informative output at other places in the program may also prove useful.)

Here are an assortment of useful facts and things to watch out for. Derivations the formulas given below for all appear in the Engineering 77S study guide, available for examination at the Self-Paced Center.)

1. The cat sees the mouse if

   (cat radius) * cos (cat angle – mouse angle)

   is at least 1.0.

2. When the cat circles distance d around the statue, its radius does not change, and the change in its angle can be calculated from the relationship

   d = angle * (radius of arc)

3. The cat catches the mouse when it (the cat) moves past the mouse while at the base of the statue, i.e. when the cat radius is 1.0 and the mouse angle lies between the old cat angle and the new cat angle. Your program should include a boolean function

   ```
   Between (angleA, angleB, angleC)
   ```

   that returns true if angleB is between angleA and angleC, and false otherwise. If angleA and angleC are less than two radians apart, the "between" condition holds exactly when

   cos (angleB – angleA) > cos (angleC – angleA), and

   cos (angleC – angleB) > cos (angleC – angleA).

4. Note that the cat cannot move inside the statue's base; hence if the cat is, say, at radius 1.7 and sees the mouse, it can move in only 0.7 meters, up to the base of the statue.

5. The mouse by accident may manage to keep completely out of sight of the cat, since both the cat and the mouse are moving counterclockwise. The cat will eventually tire of the chase if this happens. If the cat has not caught the mouse after thirty minutes of moving, you should print an appropriate message and stop the program.

6. Remember that angles a, $a + 2\pi$, $a + 4\pi$, ... are all equal.

Miscellaneous requirements

Your program is to include procedures to move the cat if it sees the mouse, to move the cat if it doesn't see the mouse, and to move the mouse; a real function to reduce an angle to the equivalent angle less than 2π; and boolean functions to determine if the cat sees the mouse and if an angle is between two others (see 3 above). Pass the information these subprograms need through parameters. You should then test these subprograms singly, using appropriate test data, to see if they work independently before putting them all together. Tutors may ask to see the results of these tests. In no case should any of your routines contain more than twenty-four lines of Pascal.

The test data for your complete program should include the following values:

cat radius	*cat angle*	*mouse angle*
0.5	—	—
−6.0	—	—
1.0	35.0	396.0
8.1	0.0	45.0
8.1	150.0	240.0
4.0	0.0	−57.0

plus whatever other values are necessary to ensure that all statements in your program have been executed at least once.

Checklist

program listing;

program appropriately split into subprograms, including a Between function, a Sees function, an angle reduction function, and movement procedures, using parameters as specified;

evidence of independent tests of the functions and subroutines;

no routine more than twenty-four lines of code;

variable and subprogram names which reflect their use;

comments at the head of the main program and subprograms which explain the purpose of each;

informative tabular output;

cat radius less than 1 detected and dealt with appropriately;

clean case analysis and simple loop structuring;

appropriate use of Pascal's various looping and testing constructs;

correct execution: loop termination conditions, loop entry conditions, case statement adequately guarded;

test data as specified.

MJC:8/96

CS 9B programming assignment
Flow of control project B

Background

Computers are good at drill. They never get tired or annoyed at a student's lack of progress. Thus they have been used for many years in instructional settings to pose short exercises for a student and to check the student's answers.

More recently, the use of artificial intelligence techniques has produced computer "tutors" that maintain an internal model of the student's knowledge. They vary the difficulty of the exercises based on their diagnosis of the student's responses and analysis of his or her ability. We consider a simple version of such a tutor in this problem.

Problem

You are to write a program that poses simple addition problems for a student and keeps track of the answers he or she types. Your program will start by asking the student for the level at which to start asking problems. There are seven levels:

- level 1: 1-digit number + 1-digit number, without a carry (e.g. 2+4);
- level 2: 1-digit + 1-digit, with a carry (e.g. 6+7);
- level 3: 1-digit + 2-digit, in either order, without carry (e.g. 72+3 or 5+41);
- level 4: 1-digit + 2-digit, in either order, with a carry (e.g. 72+9 or 5+46);
- level 5: 2-digit + 2-digit, without carry (e.g. 46+23);
- level 6: 2-digit + 2-digit, with a single carry (e.g. 46+44 or 64+44);
- level 7: 2-digit + 2-digit, two carries (e.g. 46+54).

It will then pose problems at the specified level until the student answers five consecutive problems correctly or three consecutive problems incorrectly. In the former case, the program should move the student to the next higher level if possible. In the latter case, the program should move her to the next lower level if possible.

Whenever the student's level changes, the program should ask the student whether or not to continue. If the student answers yes, the program should continue to pose problems at the new level, again moving to a new level when the student gives five consecutive right answers or three consecutive wrong answers. If the student wishes not to continue, or graduates from level 7, the program should print a message indicating the last level reached and quit.

Miscellaneous requirements

You should represent the number of consecutive right or wrong answers required to change levels as Pascal constants in your program, and use them to determine whether the student user should be moved to a different problem level. It will be much easier to test your program with both these constants being 2; set them to 5 and 3 only for your final run.

Your program should be divided into subprograms, with the information each subprogram needs passed through parameters. None of your subprograms should be longer than twenty-four lines of code.

One of them should be a procedure named Administer. Administer, when given the tens and units digits of two values to add, asks the student to type the sum of these two values, and returns in a boolean var parameter an indication of whether or not the student got the problem right. The problem should be posed to the student as in the following example:

```
  56
+ 98
-----
```

The student would type the answer on the line following the problem.

Administer should be called by a set of seven procedures named GiveLevel1Problem, GiveLevel2Problem, ..., GiveLevel7Problem. The first five of these are in the file ~cs9b/lib/giveProblem.i, and are listed on an accompanying page; you supply the other two. You should test GiveLevel6Problem and GiveLevel7Problem separately and bring the results of the tests to the Self-Paced Center when you have this program graded.

You may assume that all values typed by the student are integers. You should, however, check that the level the student specifies initially is an integer between 1 and 7, and continue asking for the level until it is.

The code in ~cs9b/lib/giveProblem.i uses a random number generator defined in the file ~cs9b/lib/random.i. How you use this code depends on what computing platform you're using. On mingus.eecs, you should copy the files random.i and HP.giveProblem.i to your own directory, add a declaration for an integer variable named seed to your variable declarations, add the lines

```
$include 'random.i'$
$include 'HP.giveProblem.i$
```

after the variable declarations in your main program, and add an assignment statement to seed to the start of your main program. The file ~cs9b/lib/HP.giveProblem.test.p provides an example of how to do this.

To test your program, run it through each level, with at least one attempt to move back from the first level and at least one attempt to move back from a subsequent level. Also provide enough evidence for a tutor to see that your random generation of values to add is working correctly.

Checklist

program listing;

program appropriately split into subprograms, with all information needed passed as parameters;

procedures Administer, GiveLevel1Problem, ..., GiveLevel7Problem as specified;

no routine more than twenty-four lines of code;

use of Pascal constants for the number of consecutive right answers to advance a level and the number of consecutive wrong answers to go back a level;

variable and subprogram names that reflect their use;

comments at the head of the main program and subprograms that explain their purpose;

problems posed as described, enough to support the program's correctness;

error-checking of the initially-specified level;

separate tests of GiveLevel6Problem and GiveLevel7Problem;

clean case analysis and simple loop structuring;

appropriate use of Pascal's various looping and testing constructs;

correct execution: loop termination conditions, loop entry conditions, case statement adequately guarded;

each level tested;

at least one attempt to move back from level 1;

at least one attempt to move back from a subsequent level;

sufficient evidence of correct random value generation.

MJC:8/97

```
{
    Return a random value between j and k, inclusive.
}

function RandomInt (j, k: integer):
    integer;
var
    x: real;
begin
    RandomInt :=
        k1 + trunc ((k2-k1+1)*random
(0.0));
end;

{
    Pose the problem of adding two one-digit values
    with no carry.
}

procedure GiveLevel1Problem (var
    rightAnswer: boolean);
var
    u1, u2: integer;
begin
    u1 := RandomInt (1, 8);
    u2 := RandomInt (1, 9-u1);
    {to ensure no carry}
    Administer (0, u1, 0, u2, rightAnswer);
end;

{
    Pose the problem of adding two one-digit values
    with a carry.
}

procedure GiveLevel2Problem (var
    rightAnswer: boolean);
var
    u1, u2: integer;
begin
    u1 := RandomInt (2, 9);
    u2 := RandomInt (10-u1, 9);
    {to ensure carry}
    Administer (0, u1, 0, u2, rightAnswer);
end;

{
    Pose the problem of adding a one-digit value and a
    two-digit value with no carry.
    The odds of having the first value have two digits
    should be roughly the same as the odds of having
    the second value have two digits.
}

procedure GiveLevel3Problem (var
    rightAnswer: boolean);
var
    u1, u2, t, t1, t2: integer;
begin
    u1 := RandomInt (1, 8);
    u2 := RandomInt (1, 9-u1);
    {to ensure no carry}
    t := RandomInt (1, 9);
    {flip a coin}
    if RandomInt (1, 2) = 1 then begin
        t1 := t;    {first value has two digits}
        t2 := 0;
    end else begin
        t1 := 0;
        t2 := t;    {second value has two digits}
    end;
    Administer (t1, u1, t2, u2,
rightAnswer);
end;

{
    Pose the problem of adding a one-digit value and a
    two-digit value with a carry.
    The odds of having the first value have two digits
    should be roughly the same as the odds of having
    the second value have two digits.
}

procedure GiveLevel4Problem (var
    rightAnswer: boolean);
var
    u1, u2, t, t1, t2: integer;
begin
    u1 := RandomInt (2, 9);
    u2 := RandomInt (10-u1, 9);
    {to ensure carry}
    t := RandomInt (1, 8);
    if RandomInt (1, 2) = 1 then begin
        t1 := t;
        t2 := 0;
    end else begin
        t1 := 0;
        t2 := t;
    end;
    Administer (t1, u1, t2, u2,
rightAnswer);
end;

{
    Pose the problem of adding two two-digit values
    with no carry.
}

procedure GiveLevel5Problem (var
    rightAnswer: boolean);
var
    u1, u2, t1, t2: integer;
begin
    u1 := RandomInt (1, 8);
    u2 := RandomInt (1, 9-u1);
    {to ensure no carry}
    t1 := RandomInt (1, 8);
    t2 := RandomInt (1, 9-t1);
    {to ensure no carry}
    Administer (t1, u1, t2, u2,
rightAnswer);
end;
```

CS 9B programming assignment
Flow of control project C

Background

For this assignment, you write a program to play the game of "Twenty-one" (also called "blackjack").

The general idea of Twenty-one is as follows. There are two players, the *customer* and the *dealer*. The dealer deals two cards to the customer, both face down, and two cards to herself, the first face down and the other face up. In most cases, the customer may then ask for more cards, one at a time. The dealer does the same, and the hand is resolved. The object of the game is to be dealt a set of cards whose total value is as close to 21 as possible without going over.

Cards are dealt from a deck of 52. There are four cards in the deck with value 2, four with value 3, and so on up to value 9; there are sixteen cards with value 10; and there are four cards ("aces") whose value may be either 1 or 11 depending on the value of a player's remaining cards. For both players, an ace counts 11 unless the hand total is more than 21, in which case it counts 1.

For our purposes, the actual play of the game proceeds as follows. Each player receives two cards as described above. If the dealer's cards total 21—that is, one has value 10 and the other is an ace—the dealer has a "blackjack" and wins the hand. Otherwise, if the customer's cards total 21, she has a blackjack and wins the hand. Otherwise, the customer may ask for more cards one by one, until either she goes over 21—in which case she loses, or "busts"—or has a total that she thinks is high enough to beat the dealer. After the customer has stopped with a total at most 21, the dealer draws, taking another card if her total is 16 or less, and stopping if her total is over 16. If the dealer busts, she loses. If neither player has bust, the totals of the two sets of cards are compared. If the customer's is higher, she wins; if the dealer's is higher, she wins; and if the two are equal, no one wins.

Examples

Some annotated sample hands are shown below. Only the values of the cards are listed, except for aces.

Customer hand = 10 ace
Dealer hand = ace 10
Dealer has blackjack and wins immediately,
even though the customer also has blackjack.

Customer hand = 10 ace
Dealer hand = 4 9
Customer has blackjack and wins immediately.

Customer hand = 4 ace
Dealer hand = 4 9
Customer continues, and is dealt a 7; her hand is now 4 1 7.
Customer continues, and is dealt a 10; dealer wins.

Customer hand = 4 ace
Dealer hand = 4 9
Customer continues, and is dealt a 7; her hand is now 4 1 7.
Customer continues, and is dealt an ace; her hand is now 4 1 7 1.
Customer continues, and is dealt a 5; her hand is now 4 1 7 1 5.
Dealer draws a 10 and loses.

Customer hand = 4 ace
Dealer hand = 4 9
Customer continues, and is dealt a 7; her hand is now 4 1 7.
Customer continues, and is dealt an ace; her hand is now 4 1 7 1.
Customer continues, and is dealt a 5; her hand is now 4 1 7 1 5.
Dealer's total is less than 17, so she draws a 3; her hand is now 4 9 3.
Dealer's total is less than 17, so she draws an ace; her hand is now 4 9 3 1.
Dealer has 17 and must stand. Customer has more than dealer, so customer wins.

Customer hand = 4 ace
Dealer hand = 4 9
Customer continues, and is dealt a 7; her hand is now 4 1 7.
Customer continues, and is dealt an ace; her hand is now 4 1 7 1.
Customer continues, and is dealt a 5; her hand is now 4 1 7 1 5.
Dealer's total is less than 17, so she draws a 3; her hand is now 4 9 3.
Dealer's total is less than 17, so she draws a 5; her hand is now 4 9 3 5.
Dealer has 21 and must stand. Dealer has more than customer, so dealer wins.

Problem

You are to write a program to play Twenty-one, in order to test two strategies for the customer. Your program will play fifty games of Twenty-one and report how many games were won by dealer and customer.

Your program will include calls to two subprograms we supply, declared as follows:

```
procedure Shuffle;

function NextCard: integer;
```

Shuffle should be called at the start of each game to shuffle the deck of cards. NextCard should be called for each card dealt; it returns an integer between 2 and 11 inclusive that represents the value of the next card dealt. The file $lib/cards.i contains the Pascal code for these subprograms. The easiest way to use it is probably to copy it to your own directory and then add your own code to it.

Your program will also include a boolean function named CustomerStillDrawing with the following heading:

```
function CustomerStillDrawing (handTotal, dealerUpCard: integer): boolean;
```

This function incorporates the customer's strategy. For one of the runs you bring in for grading, this function should implement a strategy that's the same as the dealer's. For another of the runs you bring in for grading, this function should implement the following strategy:

> Take a card if and only if your hand total is less than 17 and the dealer's up card is a 7, 8, 9, 10, or ace, or if your hand total is less than 12 and the dealer's up card is a 2, 3, 4, 5, or 6. (The idea is that in the second case, the dealer is much more likely to bust since there are more 10's than any other card.)

Finally, you need to incorporate a random number generator in your program, since it's used by the Shuffle procedure. The code is in the file ~cs9b/lib/random.i. How you use this code depends on what computing platform you're using. On HP workstations, you should copy the files random.i and HP.cards.i to your own directory, add a declaration for an integer variable named seed to your variable declarations, add the lines

```
$include 'random.i'$
$include 'HP.cards.i'$
```

after the variable declarations in your main program, and add an assignment statement to seed to the start of your main program. (For an example of how to do this, examine the file ~cs9b/lib/HP.giveProblem.test.p, which $include's random.i and HP.giveProblem.i from "Flow of control project B".)

Miscellaneous requirements

Your program should be divided into subprograms, with the information each subprogram needs passed through parameters. As described above, one of these subprograms should be named CustomerStillDrawing and represent the customer's strategy. Another should be called DealerStillDrawing and represent the dealer's strategy. In no case should any of your subprograms contain any more than twenty-four lines of code, when coded according to the style guidelines described earlier in this document.

This program simulates the result of actually playing Twenty-one. Your program runs must also provide enough evidence that your simulation is accurate: that hands are dealt and evaluated correctly, and that the rules and strategies described above are correctly implemented. Include in your program sufficient output for the tutor to be able to verify its correctness quickly. One way to improve the readability of your output is to indent the intermediate output and not to indent the result of each game. For instance:

```
   customer hand:  4 9
   dealer hand:  10 8
   customer draws 7
   customer stands
   dealer stands
customer wins 20 to 18
```

Your output should include examples of each case of the dealer's strategy, as well as the customer's strategy as displayed in the following table:

your total	*dealer's up card*
< 17	7, 8, 9, 10, A
= 17	7, 8, 9, 10, A
> 17	7, 8, 9, 10, A
< 17	2 through 6
= 17	2 through 6
> 17	2 through 6
< 12	2 through 6

= 12	2 through 6
> 12	2 through 6
< 12	7, 8, 9, 10, A
= 12	7, 8, 9, 10, A
> 12	7, 8, 9, 10, A

It is probably easier to provide this output by testing your function in isolation than to rely on the random number generator to produce the right numbers.

Checklist

program listing;

program appropriately split into subprograms,
including functions that implement the dealer and customer strategies, using parameters as specified;

no routine more than twenty-four lines of code;

variable and subprogram names that reflect their use;

comments at the head of the main program and subprograms that explain their purpose;

informative and readable output, enough to support the program's correctness;

sufficient test data as described above;

clean case analysis and simple loop structuring;

appropriate use of Pascal's various looping and testing constructs;

correct execution: loop termination conditions, loop entry conditions, `case` statement adequately guarded.

MJC:8/97

cards.i

```
const
   INDENT = '    ';

type
   DeckType = set of 1..52;

var
   used: DeckType;

function RandomInt (j, k: integer): integer;
var
   x: real;
begin
   RandomInt := k1 + trunc ((k2-k1+1)*random (0.0));
end;

procedure Shuffle;
begin
   used := [];
end;

procedure PrintUsed;
var
   card: integer;
begin
   write (INDENT, 'Used: ');
   for card := 1 to 52 do begin
      if card in used then begin
         write (card:3);
      end;
   end;
   writeln;
end;

function CardValue (cardNum: integer): integer;
begin
   case (cardNum-1) mod 13 of
      0:
         CardValue := 11;
      1, 2, 3, 4, 5, 6, 7, 8:
         CardValue := cardNum mod 13;
      9, 10, 11, 12:
         CardValue := 10;
   end;
end;

function NextCard: integer;
var
   cardNum: integer;
begin
   repeat
      cardNum := RandomInt (1, 52);
   until not (cardNum in used);
   used := used + [cardNum];
   NextCard := CardValue (cardNum);
   PrintUsed;
end;
```

Quiz—Arrays

Quizzes for this topic drill you on simple uses of arrays, in particular: determining when arrays are necessary; filling them in specified patterns; and using the contents of one array to index another.

Readings

Chapter 7 in Cooper, or the sections on arrays in whatever Pascal book you're using.

Exercises

Sample quiz questions appear at the back of this document.

Quiz—Records

Here you learn to construct records, which are aggregates of data of different types. Quizzes test your ability to declare records and access their fields.

Readings

Chapter 8 in Cooper, or the sections on records in whatever Pascal text you're using.

Exercises

Sample quiz questions appear at the back of this document.

Program—Exercises with arrays and records

These short programs give you practice with techniques for declaring and using arrays and records in Pascal. These techniques include the use of two-dimensional arrays, strings, characters as substripts, and records. Most languages allow the use of two-dimensional arrays and strings, so these should be familiar to you. (Pascal's details are of course slightly different.) If your experience has been mainly in Fortran or Basic, you may be unfamiliar with the use of records, and of characters as subscripts.

Related quizzes

Arrays, records.

Programming assignment

Do *all* of the four problems described on the immediately following pages.

Checklist

All: correctly working code;
indenting, naming, etc. in accordance with style guidelines in this document.

2.1: use of parameters and $lib/encode.p as specified;
appropriate testing.

2.2: additions of procedures and declarations as specified;
data that tests insertion and deletion at the ends of the string as well as the middle;
data that tests deletion from a full array.

2.3: LoadLine, driver program, and WriteLine routine as specified;
test data that includes a blank line and a full line.

2.4: procedures that set board cells and print the board as specified;
output that indicates the orientation of the board;
test data that adequately displays the correctness of the program.

Problem 2.1 — practice with arrays indexed by characters

Write a procedure to encode a word by replacement. For instance, it might encode the word “adder” as “cqqmw”, by replacing each “a” by a “c”, each “d” by a “q”, each “e” by an “m”, and each “r” by a “w”. The code will be represented as an array indexed from “a” to “z”, each cell of which contains the corresponding replacement letter. An array to encode “adder” as “cqqmw” might be

```
a b c d e f g h i j k l m n o p q r s t u v w x y z   (not encoded)
c g l q m r z h v i t d e s k b y w u x a j o p n f   (encoded)
```

The procedure will fit into the following program, which you’ll find in $lib/encode.p (you supply the declarations).

```
program exercise (input, output);

{You supply the declarations and the Encode procedure.}

procedure GetValues ( {You supply the parameter declarations.} );
var
   ch: char;
   i: integer;
begin
   writeln ('Code array:  ');
   for ch := 'a' to 'z' do begin
      read (codeArray [ch]);
   end;
   readln;
   word := '     ';         {quick way to blank out word, allowed in UNIX Pascal}
   i := 0;
   write ('Word to encode:  ');
   while not eoln do begin
      i := i + 1;
      read (ch);
      if i <= WORDSIZE then begin
         word [i] := ch;
      end;
   end;
   readln;
end;

begin
   GetValues (codeArray, word);
   Encode (codeArray, word);
   writeln ('Coded word = ', word);
end.
```

Problem 2.2 — practice with character strings

The program below, in $lib/stringproc.p, is missing its declarations and two procedures, Insert and Delete. You are to add the declarations and procedures, and devise data for testing them.

Insert takes three arguments: an array of characters, another character to insert into the array, and the position to insert it into. Inserting the character 'R' into the second position of a five-element array whose first four characters are 'F', 'I', 'L', and 'L' changes the contents of the array to 'F', 'R', 'I', 'L', and 'L'. Thus some of the characters are moved over one cell in the array, and the last character is lost.

Delete takes two arguments: an array of characters and the position of a character to delete. It moves all characters after that position back one cell, and makes the last character blank. Deleting the second character of the five-element array containing 'F', 'R', 'I', 'L', and 'L' would produce an array whose first four characters are 'F', 'I', 'L', and 'L' and whose last character is blank.

```
program exercise (input, output);

{You supply the declarations, and the procedures Insert and Delete.}

procedure ReadLine ( {You also supply the parameter declarations.} );
var
   i: integer;
   ch: char;
begin
   for i := 1 to LINESIZE do begin
      line [i] := ' ';
   end;
   writeln ('Please type a line of characters.');
   i := 0;
   while not eoln do begin
      i := i + 1;
      read (ch);
      if i <= LINESIZE then begin
         line [i] := ch;
      end;
   end;
   readln;
end;

begin
   ReadLine (line);
   write ('Character to insert?  ');
   readln (ch);
   write ('Position to insert it?  ');
   readln (position);
   Insert (line, ch, position);
   writeln (line);
   write ('Position of character to delete?  ');
   readln (position);
   Delete (line, position);
   writeln (line);
end.
```

Problem 2.3 — practice with using records

A procedure LoadLine to fill an array with a line of characters from the input is listed below. Rewrite it to return a record whose two fields contain the characters read and how many there were. Then write a main program and a WriteLine procedure to see if the new representation works.

```
const
   MAXCHARS = 50;
   BLANK = '                                                  ';

type
   LineType = array [1..MAXCHARS] of char;

procedure LoadLine (var thisLine: LineType);
var
   count: integer;
begin
   thisLine := BLANK;
   count := 0;
   while not eoln and (count < MAXCHARS) do begin
      count := count + 1;
      read (thisLine[count]);
   end;
end;
```

Problem 2.4 — practice with two-dimensional arrays

In a game like chess or checkers, the game board is normally placed between the players, and they sit on opposite sides. Thus, the two players see the board from different perspectives. Opposite views of a 3-by-3 tic-tac-toe board, for instance, might be as follows:

```
xox        oxo
ox          xo
oxo        xox
```

Write and test two procedures to manipulate a 19-by-19 game board.

```
procedure SetCell (var board: BoardType; player, row, col: integer; c: char);
```

stores the given character argument at board position [row][col], computed from the given player's perspective. For example, on the 3-by-3 board that, viewed from player 1's perspective, contains

```
xox
ox.
oxo
```

changing board cell [1][3] to 'M' from player 1's perspective would result in the board

```
xoM
ox.
oxo
```

while changing board cell [1][3] to 'M' from player 2's perspective would result in the board

```
xox
ox.
Mxo
```

```
procedure Print (board: BoardType; player: integer);
```

prints the board from the given player's perspective.

The players are numbered 1 and 2, and the rows and columns of the board are numbered from 1 to 19.

Your test cases should include calls to SetCell and Print for each of the two players.

Quiz—Recursion

Questions for this quiz ask you write some short recursive subprograms and to understand, modify, and criticize some other recursive code.

Readings

Chapter 16 in Cooper (or sections on recursion in whatever other Pascal text you're using).

Exercises

Sample quiz questions appear at the back of this document.

Program—Recursion exercises

This assignment gives you practice with several aspects of using recursion: writing your own recursive routines, debugging those written by others, and incorporating informative tracing.

You will probably benefit by doing this assignment before you take the recursion quiz.

Readings

Chapter 16 in Cooper (or sections on recursion in whatever other Pascal text you're using).

Related quizzes

Recursion

Programming assignment

Do *all* of the three problems described on the immediately following pages.

Checklist

All: correctly working code;
indenting, naming, etc. in accordance with style guidelines in this document.

3.1: bugs fixed with minimal changes;
test data which check the trivial merge (all in order) and the typical merges (first list runs out first, second list runs out first).

3.2: correct output;
test data showing various recursion levels.

3.3: recursive function, with parameters as specified;
correct output for the given examples, plus tests for consecutive *'s and *'s at the end.

Problem 3.1 — debugging a recursive program

An array may be sorted by the merge sort algorithm as follows. First, it's divided in two, and the two halves are sorted recursively. Then, the two halves are merged into one—a temporary array is useful for this—and we have everything in order. An example:

> To sort the array `3 1 8 2 5 9 4 6 0 7`,
> divide it into `3 1 8 2 5` and `9 4 6 0 7`,
> sort those, getting `1 2 3 5 8` and `0 4 6 7 9`,
> and merge those, getting `0 1 2 3 4 5 6 7 8 9`.

In the file $lib/mergesort.p is an implementation of merge sort, containing a bug or two. Fix the bugs. You will not need to add any statements, just make small changes in already-existing code.

Problem 3.2 — understanding a recursive program

Any translator for a programming language includes a *parser*, which tests a string of characters to see if it's a legal construct of the language. One of the tasks of a programming language parser is to keep track of matching parentheses (e.g., in Pascal, (), [], {}, begin-end, etc.). The rules defining legally parenthesized strings are often expressed recursively, and may be reflected in the structure of a parsing program.

The file $lib/balparens.p contains parsing code that analyzes parenthesis strings typed by the user, to see whether or not they are balanced. You are to add a trace facility which, in the Check procedure, prints the left and right indices of the balanced parenthesis string that is recognized, and in the CheckParenSeq procedure prints the left and right indices of the balanced string sequence that's recognized. Show how deep you are in the recursion in your trace output by indenting each line four spaces for each recursion level. Your program will produce output like the following. (Lines beginning with asterisks would be printed by the trace routine.)

```
Please type a paren string:  ((()())())
        **** Balanced string from 3 to 4
        **** Balanced string from 5 to 6
        **** Balanced sequence from 3 to 6
    **** Balanced string from 2 to 7
    **** Balanced string from 8 to 9
    **** Balanced sequence from 2 to 9
**** Balanced string from 1 to 10
**** Balanced sequence from 1 to 10
Balanced.

Please type a paren string:  ()(())
**** Balanced string from 1 to 2
    **** Balanced string from 4 to 5
    **** Balanced sequence from 4 to 5
**** Balanced string from 3 to 6
**** Balanced sequence from 1 to 6
Balanced.
```

Problem 3.3 — writing a recursive program

The file $lib/match.p contains most of a program to do pattern matching, much the same way as the UNIX shell matches file names. The program reads two strings, a pattern and an object string. All characters in the pattern match themselves in the object string except an asterisk, which can match *any* substring of 0 or more characters in the object string. Thus "a*a" in the pattern matches any object string that starts and ends with an "a"; "a*b*c" matches any string that starts with an "a" and ends with a "c", with a "b" somewhere in between; "*b" matches any string ending with a "b".

You are to write the Match function, which takes a pattern string and an object string as arguments and determines if the pattern *exactly* matches the object string. (Note that your function may take only two arguments; you may, however, write an auxiliary function for it to call.) The table below gives some examples of what your function will return.

pattern	*object*	Match *returns*
a	a	true
a	ab	false
ab	a	false
a*b	ab	true
a*b	abc	false
a*b	cab	false
a*b	abcdb	true
*b	b	true
*b	aabb	true
*b	bc	false
a*b*c	abc	true
a*b*c	ababc	true
a*b*c	aabbcc	true
a*b*c	aaacbb	false

mergesort.p

```
program exercise (input, output);

type
   ArrayType = array [1..10] of integer;

var
   list: ArrayType;
   i: integer;

procedure Trace (list:ArrayType; left,right:integer; entering:boolean);
begin
   if entering then begin
      writeln ('**** Entering mergesort:');
   end else begin
      writeln ('**** Leaving mergesort:');
   end;
   if right = left then begin
      writeln (left:3);
      writeln (list[left]:3);
   end else if right > left then begin
      write (left:3);
      for i := left+1 to right-1 do begin
         write (' ':3);
      end;
      writeln (right:3);
      for i := left to right do begin
         write (list[i]:3);
      end;
      writeln;
   end;
end;

procedure Merge (source: ArrayType; var destination: ArrayType;
   index1, bound1, index2, bound2: integer);
var
   merging, i: integer;
begin
   merging := index1;
   while (index1 <= bound1) and (index2 <= bound2) do begin
      if source[index1] < source[index2] then begin
         destination[merging] := source[index1];
         index1 := index1 + 1;
         merging := merging + 1;
      end else begin
         destination[merging] := source[index2];
         index2 := index2 + 1;
         merging := merging + 1;
      end;
   end;
   for i := index1 to bound1 do begin
      destination[merging] := source[i];
      merging := merging + 1;
   end;
   for i := index2 to bound2 do begin
      destination[merging] := source[i];
      merging := merging + 1;
   end;
end;
```

```
procedure Mergesort (var list: ArrayType; left, right: integer);
const
   ENTERING = true;
   LEAVING = false;
var
   middle, i: integer;
   tempList: ArrayType;
begin
   Trace (list, left, right, ENTERING);
   if left < right then begin
      middle := (left+right) div 2;
      for i := left to right do begin
         tempList[i] := list[i];
      end;
      Mergesort (tempList, left, middle);
      Mergesort (tempList, middle, right);
      Merge (list, tempList, left, middle, middle+1, right);
   end;
   Trace (list, left, right, LEAVING);
end;

begin
   write ('Please type ten numbers:  ');
   for i := 1 to 10 do begin
      read (list[i]);
   end;
   readln;
   Mergesort (list, 1, 10);
end.
```

balparens.p

```
{
   This program determines whether or not a string of parentheses
   input from the user is balanced.  It uses the following recursive
   definition for nonempty balanced parenthesis strings:
      A balanced_string is either ( ), or a balanced_string_sequence
         between ( and ).
      A balanced_string_sequence is a sequence of one or more
         balanced_strings.
   The procedure Check recognizes balanced_strings, and the procedure
   CheckParenSeq recognizes balanced_string_sequences.
}
program parentest (input, output);

type
   StringType = array [1..40] of char;

var
   parens: StringType;
   index: integer;
   okSoFar: boolean;


procedure CheckParenSeq (var parens: StringType; var index:
integer;
   var okSoFar: boolean);  forward;
{
   Check is entered with index pointing to a left parenthesis,
   and updates index so that it points to the matching right parenthesis.
   Example:  if parens contains (()()()) and Check is entered
   with index at 2, a value of 3 will be returned in index.
   The variable okSoFar indicates whether or not the string is legal.
}
procedure Check (var parens: StringType; var index: integer;
   var okSoFar: boolean);
begin
   assert (parens[index] = '(');
   okSoFar := true;
   index := index + 1;
   while okSoFar and (parens[index] = '(') do begin
      CheckParenSeq (parens, index, okSoFar);
   end;
   if parens[index] <> ')' then begin
      okSoFar := false;
   end;
end;
```

```
{
    CheckParenSeq is entered with index pointing to a left parenthesis,
    and updates index so that it points one past the right parenthesis
    of the last balanced string in the sequence.
    Example:  if parens contains (()()()) and CheckParenSeq is entered
    with index at 2, a value of 8 will be returned in index.
    The variable okSoFar indicates the legality of the string.
}
procedure CheckParenSeq {var parens: StringType;
   var index: integer;
   var okSoFar: boolean};
begin
   assert (parens[index] = '(');
   okSoFar := true;
   repeat
      Check (parens, index, okSoFar);
      if okSoFar then begin
         index := index + 1;
      end;
   until not okSoFar or (parens[index] <> '(');
end;

{
    Read a string (terminated by carriage return).
    We omit error checks to present the recursion more clearly.
}
procedure GetString (var s: StringType);
var
   length: integer;
begin
   length := 0;
   s := '  ';
   while not eoln (input) do begin
      length := length + 1;
      read (input, s[length]);
   end;
   readln (input);
end;


begin
   while true do begin
      write (output, 'Please type a paren string:  ');
      GetString (parens);
      index := 1;
      okSoFar := true;
      CheckParenSeq (parens, index, okSoFar);
      if okSoFar and (parens[index] <> ')') then begin
         writeln (output, 'Balanced.');
      end else begin
         writeln (output, 'Unbalanced at position ', index:1, '.');
      end;
   end;
end.
```

match.p

```
program patternmatch (input, output);

const
   MAXSTRLEN = 10;
   MATCHANYTHING = '*';
   BLANKS = '  ';

type
   StringType = record
      length: integer;
      contents: array [1..MAXSTRLEN] of char;
   end;

var
   myPattern, myObject: StringType;

{
   Read a line of characters from the user, keeping track
   of how many there are.
}
procedure GetString (var myString: StringType);
var
   ch: char;
begin
   myString.length := 0;
   myString.contents := BLANKS;
   while not eoln do begin
      read (ch);
      if myString.length < MAXSTRLEN then begin
         myString.length := myString.length + 1;
         myString.contents [myString.length] := ch;
      end;
   end;
   readln;
end;

{You write this.}
function Match (myPattern, myString: StringType): boolean;
begin
end;

begin {main program}
   write ('Pattern: ');
   GetString (myPattern);
   while myPattern.length > 0 do begin
      write ('Object: ');
      GetString (myObject);
      while myObject.length > 0 do begin
         writeln (Match (myPattern, myObject));
         write ('Object: ');
         GetString (myObject);
      end;
      write ('Pattern: ');
      GetString (myPattern);
   end;
end.
```

Quiz—Pointers

Recursion is conceptually hard. With pointers, on the other hand, the programmer must merely keep track of a lot of detail. Questions on this quiz ask you to write, modify, or debug short subprograms that use linked lists, and to relate linked structures to pointer diagrams.

You will probably benefit by doing the pointers programming assignment before you take this quiz.

Readings

Chapters 11 and 17 in Cooper (or sections on pointers in whatever other Pascal text you're using).

Exercises

Sample quiz questions appear at the back of this document.

Program—Pointer exercises

This assignment should familiarize you with Pascal pointers. You will run, look at, modify, and write some programs that deal with pointers.

Three of the four programs for this assignment involve some operations on sorted singly-linked lists; the other uses a more complicated structure. A number of routines for manipulating lists are online and available for your use in the file $lib/listutils.i, including procedures for creating, copying, and printing lists, finding an element in a list, and inserting an element into a list. The code for these routines appears on the following pages; read it, trying several examples and drawing pictures for them, before you start on the problems described below.

Readings

Chapters 11 and 17 in Cooper (or sections on pointers in whatever other Pascal text you're using).

Related quizzes

Pointers.

Programming assignment

Do *all* of the four exercises on the following pages.

Checklist

All: correctly working code;
indenting, naming, etc. in accordance with CS 9B style sheet.

4.1: proper parameterization of the PrintAfter procedure;
use of Find and WriteList as directed;
sufficient test cases, including those specified.

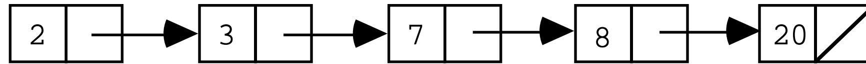
4.2: proper parameterization of the Delete procedure;
sufficient test cases, including those specified.

4.3: data structure diagrams as directed;
all requested procedures;
sufficient test cases.

4.4: clean bug fixes;
sufficient test cases.

Problem 4.1 — using procedures that work with linked lists

Write a procedure PrintAfter that, given a list of integers list and another integer k as arguments, prints the sublist of all elements of list that follow k. For example, printing out all the elements after the 7 in the list



yields as output

```
        8         20
```

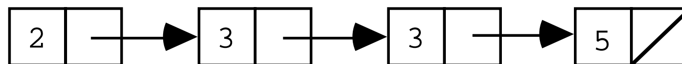
Hint: use the Find and WriteList procedures in the file $lib/listutils.i. Also use the following type declarations:

```
type
   ElementPtrType = ^ElementNode;
   ElementNode = record
      data: integer;
      next: ElementPtrType;
   end;
```

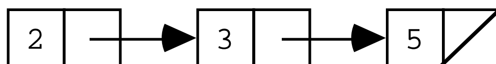
You must provide enough test cases to convince a tutor that your program works. Cases should include a list where k is at the beginning, another where k is at the end, and a third where k isn't in the list at all.

Problem 4.2 — deleting an element

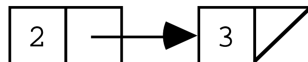
Write a procedure Delete which deletes the first occurrence of an integer from a list. Delete will be given two parameters: a pointer to a singly-linked list whose integer elements are in nondecreasing order, and an integer itemToDelete to delete from that list. For example, deleting 3 from the list



results in the list



Deleting 5 from that list results in the list



You must provide enough test cases to convince a tutor that your program works. Cases should include the following:

a list that does not contain the item to be deleted;
a list whose first item is the item to be deleted;
a list whose last item is the item to be deleted;
a list that contains two copies of the item to be deleted.

For your list elements, use the type declarations from problem 4.1.

Problem 4.3 — amoeba family trees[*]

An amoeba family tree is simpler than a normal family tree because amoebas do not get married. An amoeba has one parent, perhaps a million siblings (brothers and sisters), and billions and billions of children. An amoeba also has a name.

Amoebas (or amoebae) live dull lives. All they do is reproduce. So all we need to keep track of them are the following declarations:

```
type
   AmoebaName = array [1..AMOEBANAMESIZE] of char;
   AmoebaPtr = ^ Amoeba;
   Amoeba = record
      name : AmoebaName;
      parent : AmoebaPtr;
      olderSibling : AmoebaPtr;
      youngestChild : AmoebaPtr;
      oldestChild : AmoebaPtr;
   end;

var
   me : AmoebaPtr;
```

This declaration and some handy routines are in the file $lib/amoeba.p. A listing accompanies this handout.

a. Draw a picture of the "family" constructed by the main program. Arrows in the picture will show all the family relationships. Make sure that your drawing shows the tree exactly as the program builds it (a misleading drawing is worse than none at all).

b. Suppose you were the amoeba pointed to by the pointer me. Write a routine to print out the name of your grandparent. Try it out. What happens? Why? Can you do what's necessary to make this routine work properly? Hint: look at your drawing.

c. Write a procedure called PrintChildren that, given a pointer to an amoeba as argument, prints the names of all the amoeba's children.

d. Write a procedure called PrintGrandChildren that, given a pointer to an amoeba as argument, prints the names of all the amoeba's grandchildren. Hint: Use the routine from part c.

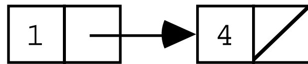
e. Write a procedure called PrintDescendants that, given a pointer to an amoeba as argument, prints the names of all the amoeba's descendants (children, grandchildren, etc.). Hint: Use the routine from part c.

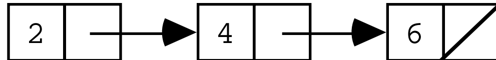
---

[*] Note: this is a computer science class, not a biology class. Any similarity between this lab and micro-organisms living or dead is purely coincidental.

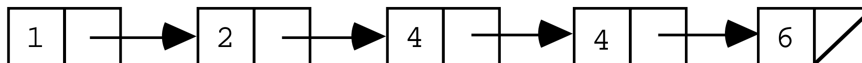
Problem 4.4 — debugging a merge routine

The file $lib/merge.p (a listing appears later in this document) contains a procedure Merge which is supposed to merge two sorted lists, that is, produce a single sorted list whose elements are those of the two lists. For example, when given the two lists





it should produce the list



without changing either of the two component lists. The merge procedure has a bug or two or three. Find the bugs and fix them, using available routines where possible.

listutils.i

```
{
   Create a new element record with data field n and next field nil,
   and return a pointer to it in newElement.
}
procedure MakeNewElement (var newElement: ElementPtrType;
   n: integer);
begin
   new (newElement);
   newElement^.data := n;
   newElement^.next := nil;
end;

{
   Insert itemToInsert into the list at the proper place.
   There are three cases:  if the list is empty, we create a list consisting
   of the single item; if the item is smaller than the first item of the list,
   we put it there; otherwise we find the position where itemToInsert should
   go and put it there. The last case requires us to keep track of the list
   element before the insertion point.
}

procedure Insert (itemToInsert: integer;  var list: ElementPtrType);
var
   newElement, insertAfter: ElementPtrType;
   done: boolean;
begin
   MakeNewElement (newElement, itemToInsert);
   if list = nil then begin
      list := newElement;
   end else if itemToInsert <= list^.data then begin
      newElement^.next := list;
      list := newElement;
   end else begin
      insertAfter := list;
      done := false;
      while not done do begin
         if insertAfter^.next = nil then begin
            insertAfter^.next := newElement;
            done := true;
         end else if itemToInsert <= insertAfter^.next^.data then begin
            newElement^.next := insertAfter^.next;
            insertAfter^.next := newElement;
            done := true;
         end else begin
            insertAfter := insertAfter^.next;
         end;
      end;
   end;
end;
```

{Write out the elements of a list.}

```
procedure WriteList (list: ElementPtrType);
var
   p: ElementPtrType;
begin
   if list = nil then begin
      writeln ('Empty list.');
   end else begin
      p := list;
      while p <> nil do begin
         write (p^.data:4);
         p := p^.next;
      end;
      writeln;
   end;
end;
```

{Build a sorted list from data values provided by the user.}

```
procedure ReadList (var list: ElementPtrType);
var
   n: integer;
begin
   writeln ('Please type list items followed by a 0.');
   list := nil;
   read (n);
   while n <> 0 do begin
      Insert (n, list);
      read (n);
   end;
   readln;
   WriteList (list);
end;
```

{Copy listSource into listDest.}

```
procedure MakeCopy (var listDest: ElementPtrType;
   listSource: ElementPtrType);
var
   tailDest: ElementPtrType;
begin
   if listSource = nil then begin
      listDest := nil
   end else begin
      MakeNewElement (listDest, listSource^.data);
      tailDest := listDest;
      listSource := listSource^.next;
      while listSource <> nil do begin
          MakeNewElement (tailDest^.next, listSource^.data);
          tailDest := tailDest^.next;
          listSource := listSource^.next;
      end;
   end;
end;
```

```
{
   Find an item in a list, and either return a pointer to it or indicate
   that it isn't in the list.
}

procedure Find (itemToFind: integer;  list: ElementPtrType;
   var foundPtr: ElementPtrType;  var found: boolean);
begin
   foundPtr := nil;
   found := false;
   while (list <> nil) and not found do begin
      if list^.data = itemToFind then begin
         foundPtr := list;
         found := true;
      end else begin
         list := list^.next;
      end;
   end;
end;
```

amoeba.p

```
program amoeba(input,output);

const
   AMOEBANAMESIZE = 10; { maximum number of characters in amoeba name }

type
   AmoebaName = array [1..AMOEBANAMESIZE] of char;
   AmoebaPtr = ^ Amoeba;
   Amoeba = record
      name : AmoebaName;              { this one's name }
      parent : AmoebaPtr;             { good old mom (or is it dad?) }
      olderSibling : AmoebaPtr;       { the next older brother/sister }
      youngestChild : AmoebaPtr;      { the youngest kid }
      oldestChild : AmoebaPtr;        { the oldest kid }
   end;

{ NewAmoeba: create a new amoeba with given name }
function NewAmoeba(name : AmoebaName) : AmoebaPtr;
var
   newOne : AmoebaPtr;
begin
   { create the new amoeba }
   new(newOne);
   { set name as specified and pointers to nil }
   newOne^.name := name;
   newOne^.parent := nil;
   newOne^.olderSibling := nil;
   newOne^.youngestChild := nil;
   newOne^.oldestChild := nil;

   NewAmoeba := newOne;
end;

{ PrintName: print the name of an amoeba }
procedure PrintName(which : AmoebaPtr);
begin
   { watch out for nil amoeba }
   if which = nil then begin
      writeln(output,'PrintName: Sorry.  That one has no name.');
   end else begin
      writeln(output,which^.name);
   end;
end;
```

```
{ AddChild: add a child to a parent. Make it the youngest child }
{   connect all pointers as appropriate }
procedure AddChild(parent, newChild : AmoebaPtr);
var
   otherSibling : AmoebaPtr;
begin
   { make child point to parent }
   newChild^.parent := parent;

   { check for parent having other children }
   otherSibling := parent^.youngestChild;
   if otherSibling = nil then begin
      { only child. Make parents youngest and oldest point to this one }
      parent^.youngestChild := newChild;
      parent^.oldestChild := newChild;

      { there is no older sibling }
      newChild^.olderSibling := nil;
   end else begin
      { other children. Make this one youngest }
      parent^.youngestChild := newChild;

      { no younger siblings, but this one points to others as older siblings }
      newChild^.olderSibling := otherSibling;
   end;
end;

var
   me : AmoebaPtr;                    { me }
   grandparent : AmoebaPtr;           { my grandparent }
   parent : AmoebaPtr;                { my parent }
   brother, sister : AmoebaPtr;       { my brother and sister }
   child1, child2, child3 : AmoebaPtr;     { my children }
   grandchild11, grandchild12 : AmoebaPtr;       { grandchildren }
   grandchild31, grandchild32 : AmoebaPtr;

begin
   { create everybody }
   me :=    NewAmoeba('me');
   parent :=      NewAmoeba('mom/dad');
   grandparent :=       NewAmoeba('Amos McCoy');
   brother :=     NewAmoeba('Fred');
   sister :=      NewAmoeba('Wilma');
   child1 :=      NewAmoeba('Larry');
   child2 :=      NewAmoeba('Curly');
   child3 :=      NewAmoeba('Moe');
   grandchild11 :=      NewAmoeba('Laurel');
   grandchild12 :=      NewAmoeba('Hardy');
   grandchild31 :=      NewAmoeba('Rowan');
   grandchild32 :=      NewAmoeba('Martin');

   { This will seem a little backwards, because we have an "add child" }
   {   instead of an "add parent". We have to put the grandchildren on }
   {   the children and then add the children to me, then add me to my parent. }

   { first do Larry's kids }
   AddChild(child1,grandchild11);
   AddChild(child1,grandchild12);
```

```
   { next do Moe's kids }
   AddChild(child3,grandchild31);
   AddChild(child3,grandchild32);

   { now add Larry, Curly, and Moe to me }
   AddChild(me,child1);
   AddChild(me,child2);
   AddChild(me,child3);

   { now add me to my parent }
   AddChild(parent,me);

   { now add my brother and sister }
   AddChild(parent,brother);
   AddChild(parent,sister);

   { we don't need these any more, do we? }
   parent := nil;
   grandparent :=  nil;
   brother := nil;
   sister := nil;
   child1 := nil;
   child2 := nil;
   child3 := nil;
   grandchild11 := nil;
   grandchild12 := nil;
   grandchild31 := nil;
   grandchild32 := nil;

   { print out my and my parent's name }
   write(output,'Hi.  My name is ');
   PrintName(me);
   write(output,'Meet my parent ');
   PrintName(me^.parent);
end.
```

merge.p

```
program mergeTest (input, output);

type
   ElementPtrType = ^ElementType;
   ElementType = record
               data: integer;
               next: ElementPtrType;
           end;

var
   list1, list2, list3: ElementPtrType;

{Add a line here to include $lib/listutils.i.
 See the file $lib/INCLUDE to find out how to do this.}

{
   Copy the smaller of the heads of the two nonnull lists listSource1
   and listSource2 into a node pointed to by destElement.
   Then move the appropriate source pointer up.
}
procedure CopySmallerAndMoveUp (var destElement,
   listSource1, listSource2: ElementPtrType);
begin
   assert (listSource1 <> nil);
   assert (listSource2 <> nil);
   if listSource1^.data <= listSource2^.data then begin
      MakeNewElement (destElement, listSource1^.data);
      listSource1 := listSource1^.next;
   end else begin
      MakeNewElement (destElement, listSource2^.data);
      listSource2 := listSource2^.next;
   end;
end;

{
   Merge listSource1 and listSource2 into listDest, without modifying
   the lists pointed to by listSource1 and listSource2.
}
procedure Merge (listSource1, listSource2: ElementPtrType;
   var listDest: ElementPtrType);
var
   tailDest: ElementPtrType;
begin
   if listSource1 = nil then begin
      MakeCopy (listDest, listSource2);
   end else if listSource2 = nil then begin
      MakeCopy (listDest, listSource1);
   end else begin
      listDest := nil;
      tailDest := listDest;
      while (listSource1 <> nil) and (listSource2 <> nil) do begin
         CopySmallerAndMoveUp (tailDest^.next, listSource1, listSource2);
      end;
      MakeCopy (tailDest^.next, listSource1);
      MakeCopy (tailDest^.next, listSource2);
   end;
end;
```

```
begin
   ReadList (list1);
   ReadList (list2);
   Merge (list1, list2, list3);
   writeln ('*** Merged list');
   WriteList (list3);
   writeln ('*** Argument lists');
   WriteList (list1);
   WriteList (list2);
end.
```

Sample quiz questions

Sample questions for “Program comprehension” quiz

1. Each of the following two programs is supposed to print the Fibonacci numbers up to and including a certain limit. (The Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, ..., with each value in the sequence being the sum of the previous two values.) Each of the programs has bugs. Provide test data for which they fail, and fix each program with as few changes as possible. Compare the programs in other ways, e.g. ease of reading and debugging, or appropriateness of looping constructs.

```
program quiz (input, output) ;
var
   limit, current, next: integer ;
begin {main program}
   write ('Please type limit:  ') ;
   readln (limit) ;
   current := 0 ;
   next := 1 ;
   writeln ('Fibonacci numbers up to ', limit:1, ' are') ;
   repeat
      writeln (current) ;
      next := next + current ;
      current := next;
   until current > limit;
end   {main program}.

program quiz (input, output) ;
var
   limit, oldfib, newfib: integer ;
begin {main program}
   write ('Please type limit:  ') ;
   readln (limit) ;
   oldfib := -1 ;
   newfib := 0 ;
   writeln ('Fibonacci numbers up to ', limit:1, ' are') ;
   while newfib < limit do begin
      newfib := newfib + oldfib ;
      oldfib := newfib ;
      writeln (newfib) ;
   end;
end   {main program} .
```

Sample questions for “Array manipulation” quiz

1. Suppose you have two arrays filled as follows:

```
a[1] = 3.14        a[2] = 2.78        a[3] = -27.6
m[1] = 3           m[2] = 1           m[3] = 2
k[1,1] = 2         k[1,2] = 3         k[1,3] = 1
k[2,1] = 6         k[2,2] = 4         k[2,3] = 3
```

What are the values of `a[m[3]]`, `m[k[1,m[2]]]`, and `a[k[k[1,1],k[1,2]]]`?

2. Write a for loop to initialize the elements of a 50-element array m as follows:

   ```
   m[1] = m[2] = 0, m[3] = m[4] = 3, m[5] = m[6] = 6,
   ```
   and so on up to `m[49] = m[50] = 72.`

Sample questions for "Records" quiz

1. Define a record for storing a person's name. The two fields of the record will be the person's first and last names, and each of these will be no more than 20 characters. Then define a record that contains information for ordering a textbook: the author's name, the name of the text, the publisher (assume the last two will fit in 30-character arrays), and the year of publication. The author's name should be stored in the first type of record you defined.
2. Given records like those you just defined (give them types NameType and TextbookType), write code to do the following:

   see if Abook was published later than Bbook;
   see if the last name of famousPerson is the same as the last name of the author of Abook;
   copy the information in Abook to Bbook;

   assuming the following declarations:

   ```
   var Abook, Bbook: TextbookType;  famousPerson: name;
   ```

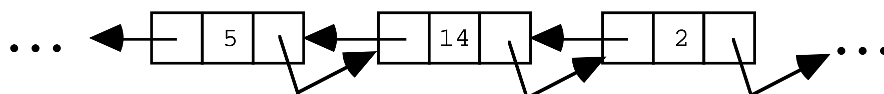
Sample questions for "Recursion" quiz

1. The following function is supposed to return the sum of the digits in its integer argument. Find and fix the bug in it.

   ```
   function DigitSum (n: integer): integer;
   begin
      if n = 0 then begin
         DigitSum := 0;
      end else begin
         DigitSum := n mod 10 + DigitSum (n div 10);
      end;
   end;
   ```
2. Write a recursive procedure PrintWithCommas to print its integer argument with commas in the correct places. For instance, PrintWithCommas (1928764) should print 1,928,764.

Sample questions for "Pointers" quiz

1. Write a declaration for a record ListNode that represents a node in the diagram below.



2. Write a procedure that inserts an element into its correct place in a list linked as in the previous problem, whose elements are arranged in order.

Solutions to sample quiz questions

Solutions to questions for "Program comprehension" quiz

1. Check first what happens the first couple of times through each loop.

   The top program prints a 0, sets next to 1 and current to 1. Then it prints 1, and sets next to 2 and current to 2. Then it prints 2. This is wrong. The bottom program prints first a –1, then a –2. This is also wrong. There is a bug in the intended "parallel assignment" to the Fibonacci variables—we change next and newfib before we're through with them.

   Fixing this problem in each program causes another problem to show up. In the top program, the test for stopping the loop is wrong (we find this out by checking the *last* iteration of the loop). In the bottom program, the writeln statement should be at the top of the loop, or else it should be printing oldfib instead of newfib.

   The bottom program has slightly better variable names, but its initialization is somewhat trickier. The top program should use a while loop, just in case a negative value for limit is read.

Solutions to questions for "Array manipulation" quiz

1.

```
a[m[3]] => a[2] => 2.78;

m[k[1,m[2]]] => m[k[1,1]] => m[2] => 1;

a[k[k[1,1],k[1,2]]] => a[k[2,3]] => a[3] => -27.6.
```

2.

```
for j := 1 to 25 do begin
   m[2*j] := (j-1) * 3;
   m[2*j-1] := m[2*j];
end;
```

Solutions to questions for "Records" quiz

1.

```
string20 = array [1..20] of char;
string30 = array [1..30] of char;
name = record
   firstname: string20;
   lastname: string20
end;

textbookinfo = record
   author: name;
   title: string30;
   publisher: string30;
   pubyear: integer
end;
```

2. Assume the following additional declaration:

```
AlaterThanB, sameName: boolean.
```

Then we have

```
AlaterThanB := Abook.pubyear > Bbook.pubyear;
sameName := famousperson.lastname = Abook.author.lastname;
Bbook := Abook;
```

Solutions to questions for “Recursion” quiz

1. 0 unfortunately is a 1-digit number. A way to fix the function is to define an auxiliary function to handle the special case, as follows:

```
function DigitSum (n: integer): integer;
begin
   if n = 0 then begin
      DigitSum := 1;
   end else begin
      DigitSum := DigitSum1 (n);
   end;
end;
```

where the code for DigitSum1 is exactly that of the DigitSum function in the question.

2.

```
procedure WriteWithCommas (n: integer);
begin
   if n < 1000 then begin
      write (n:1);
   end else begin
      WriteWithCommas (n div 1000);
      if n mod 1000 < 10 then begin
         write ('00', n mod 1000:1);
      end else if n mod 1000 < 100 then begin
         write ('0', n mod 1000:2);
      end else begin
         write (n mod 1000:3);
      end;
   end;
end;

procedure PrintWithCommas (n: integer);
begin
   WriteWithCommas (n);
   writeln;
end;
```

Solutions to questions for “Pointers” quiz

1.

```
type
   ListPtrType = ^ListNode;
   ListNode = record
      previous: ListPtrType;
      info: integer;
      next: ListPtrType;
   end;
```

2.

```
procedure Insert (itemToInsert: integer;  var list: ListPtrType);
var
   newElement, insertAfter: ListPtrType;
   done: boolean;
begin
   MakeNewElement (newElement, itemToInsert);    {Similar to that in listutils.i}
   if list = nil then begin
      list := newElement;
   end else if itemToInsert <= list^.data then begin
      newElement^.next := list;
      list^.previous := newElement;
      list := newElement;
   end else begin
      insertAfter := list;
      done := false;
      while not done do begin
         if insertAfter^.next = nil then begin
            insertAfter^.next := newElement;
            newElement^.previous := insertAfter;
            done := true;
         end else if itemToInsert <= insertAfter^.next^.data then begin
            newElement^.next := insertAfter^.next;
            newElement^.previous := insertAfter;
            insertAfter^.next := newElement;
            done := true;
         end else begin
            insertAfter := insertAfter^.next;
         end;
      end;
   end;
end;
```

Running Pascal programs

Using pc

pc is the standard UNIX Pascal Compiler. pc is not available on all EECS instructional systems; for more information, read the file /usr/pub/pascal.help.

To compile a program with pc, give the following command.

```
pc file.p
```

where *file.p* should be replaced by the name of the file containing the program you want to compile. The file name must end in the letters ".p". For example, if you want to compile the program first.p, the command would be

```
pc first.p
```

If the program contains errors, a list of the errors along with the associated line numbers will be displayed. Unfortunately, there is no way to get pc to produce a program listing with line numbers. You must therefore record the errors and then use an editor to find the errors in the program.

Once you resolve all compiler errors, you can execute your program with the following command:

```
a.out
```

The dbx debugger can help you find errors in your program while it is running; the following section, "Debugging with dbx", in this document is a guide to using dbx.

Using "include files" on HP workstations

"Include files" provide a nice way of breaking up programs into separate files so that common subprograms can be used in different programs without having to copy the subprograms. One could create packages of functions that perform various tasks such as string operations, and statistics; when the routine is needed in a program, one merely includes those files and add the desired calls to the subprograms. This is done by adding a line to the program that uses the "include file"; the Pascal compiler retrieves the file, with the result that the compiled code is the same as if the user had typed the contents of the included file at the point of the include line.

File inclusion on HP workstations

Include files are referenced by the file that uses them by including a line where the contents of the include file should occur in the program. For example, the file random.i would be included with the line

```
$include 'random.i'$
```

This acts as if the contents of random.i had been included at that spot. The program that contains the $include line is then be compiled with a pc command as usual; the $included file is not mentioned in the command.

Aspects of good programming style

by M. Clancy and M. Powell

Good programming style effectively conveys the *organization* of both a program and its data: the roles of the variables and constants, and the structure of subprograms and the data and control flow among them. Good style also incorporates a certain degree of *redundancy*. To err is human, so goes the proverb. Redundancy reduces the chance for error in understanding and modifying a program. Lastly, good style maximizes *locality*, to let the reader gain as much information as possible without flipping to and fro throughout the program.

Comments

Comments are perhaps the most obvious form of redundancy. They explain in English what the program's code "explains" in Pascal. In CS 9B, we will describe our program and data structure in comments as follows:

Guidelines for comments

Comments should appear before each subprogram to indicate

a. what it does;

b. how to use it;

c. any special things about it (global variables, error conditions).

A comment should appear on the line of a constant, type, or variable declaration to clarify its usage if necessary.

Identifiers

There are several uses for identifiers in Pascal: reserved words, and names of functions, procedures, constants, types, and variables. All identifiers are either predeclared or declared by the programmer, and thus can be looked up by the reader. Good style, however, allows us to see at a glance what role a given identifier is playing. In CS 9B, we will use upper and lower case letters (recall that Pascal identifiers contain only letters and digits) to distinguish the various roles an identifier may play.

Guidelines for identifiers

Reserved words and standard functions should be all lower case.

```
maxint, writeln, if ... then
```

Constants should be in all capital letters.

```
NAMESIZE, MAXNUMBEROFENTRIES
```

Variable and parameter names should be mostly lower case, start with lower case, and have word breaks indicated with upper case.

```
stringTable, sumOfTheSquares, nextMove
```

Type, function and procedure names should be mostly lower case, start with upper case, and have word breaks indicated with upper case.

```
TableIndex, NameString, SortArray
```

Naming

Names have two uses: as tags (visual marks to distinguish one item from another) and as descriptors. Where there is something to describe, good style will describe it as completely as possible. Thus:

Guideline for naming

Generally, names should be spelled out, unless there is an obvious and consistently used abbreviation. Loop indices and temporary variables may have short names, but global names must be longer and more descriptive, indicating the module and purpose of the variable.

Layout in general

The static structure of a program is most easily shown with "white space"—blank lines around conceptual units, and indentation to highlight the components of those units.

General layout guidelines

Indentation should be in units of four spaces.

Blank lines should appear between major sections, several blank lines between procedures.

Spaces should appear around operators, except when closeness indicates grouping.

```
deviation := (median-mean) / (number-1);

for entry := 0 to numEntries-1 do begin
```

Generally, the main program and procedures just inside it start at the left edge as follows.

```
|<- left edge
|function SumList(list : ListOfNumbers) : integer;
|var
|    index : ListIndex;
|    sum : integer;
|begin
|    sum := 0;
|    for index := 1 to LISTSIZE do begin
|        sum := sum + list[index];
|    end;
|    SumList := sum;
|end;
```

Layout of control structures

At least one "action" is associated with any Pascal control structure. For a loop, it's the repeated statement(s); for a selection, the actions are the various alternatives. The rules of Pascal, which allow an action to be a simple statement (e.g. an assignment or a procedure call), have caused many an obscure bug. Suppose we want to add code to such an action, but forget to enclose the expanded code in a begin … end. The inserted statement will most likely be indented "correctly", misleading us into thinking we've done the right thing. Redundancy in this situation is useful not so much to aid understanding as merely to save ourselves a lot of time and trouble.

Self-protection rules for control structures

Always put in begins and ends.

Always put a ";" after each statement.

We indent the statements of an action (relative to the associated control structure) to indicate its scope. The control statement itself will mark the start of the action. Some indenting schemes use the following statement to mark the action's end. In CS 9B, however, we will mark the boundary of the action with the compound statement itself, by "outdenting" its end statement. This redundant marking displays the structure of a program segment even more clearly. The examples that follow illustrate the proper layout of Pascal's control structures.

Pascal constructs to avoid

Two constructs of Pascal violate the principle of locality: the goto statement and the with statement. There's no easy way for the reader to see, given a goto, where it goes, or (even worse) given a label, what statement transfers control to it. Nor can one easily determine, within the scope of a with, what a given identifier refers to. Avoid the use of these two statements.

Length of routines

A long subprogram is probably a badly organized one. A subprogram should perform a *single* function; if it's more than about thirty lines long, chances are it's doing two or three things and should be broken up into subsidiary routines. A small routine is also likely to be more easily understood than a large one.

As for how *short* a routine should be, follow this guideline: if you can convey significantly more meaning by naming a routine than by writing the code in-line, do so. You may often do this with a boolean function, for example:

```
function No (value: integer): boolean;
begin
   No := value = 0;
end;
```

One often represents a nonexistent value with a 0, for instance, when looking up an entry in an array. Saying “if No (index) then …” is almost English, and requires no additional interpretation.

Examples of control structure layout

```
for index := 1 to topOfStack do begin
   PrintElement (index);
end;

if name = 'Clancy' then begin
   writeln (output,'Instructor');
end else if name = 'Claire' then begin
   writeln (output,'Head TA');
end else begin
   writeln (output,'TA');
end;

found := FALSE;
test := 1;
while (test <= NUMENTRIES) and not found do begin
   if key = entry[test].key then begin
      found := TRUE;
   end else begin
      test := test + 1;
   end;
end;

repeat
   read (input,nextValue);
   if nextValue <> 0 then begin
      Process (nextValue);
   end;
until nextValue = 0;

case command of
   ADD :       DoAdd;
   SUBTRACT :  DoSubtract;
   TRACE :     DoTrace;
end;

case numberOfNeighbors of
   0, 1, 4, 5, 6, 7, 8 : begin
      cell[newGen,row,col] := NOCELL;
   end;
   2 : begin
      cell[newGen,row,col] := cell[oldGen,row,col];
   end;
   3 : begin
      cell[newGen,row,col] := LIVECELL;
   end;
end;
```

Illustration of Pascal's parameter mechanisms

The following example illustrates Pascal's mechanisms for allocating memory and handling parameters. This model is useful for understanding exactly how a Pascal program with parameters operates.

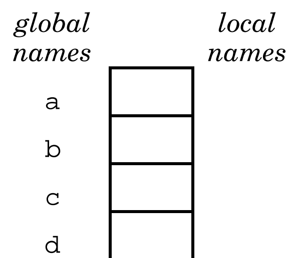
A sample program

```
 1      program example (output);

 2      var
 3         a, b, c, d: integer;

 4      procedure Test (x: integer; var a, b: integer);
 5      var
 6         c: integer;
 7      begin
 8         c := 10;
 9         a := c - b;
10         b := x;
11         x := d + 5;
12      end;

13      begin
14         a := 4;
15         b := 5;
16         c := 7;
17         d := 9;
18         Test (d, b, a);
19         writeln ('Answer is: ', a, b, c, d);
20      end;
```

Line numbers are referred to in the step-by-step solution below.

Allocate memory

When the program starts, the four global variables are allocated as in the diagram below.



Do global assignments (through line 17)

global names		*local names*	
a	4		`a := 4;`
b	5		`b := 5;`
c	7		`c := 7;`
d	9		`d := 9;`

Call procedure (line 18), deal with parameters, and allocate local variables

To deal with the first parameter, x, we add a memory location to the display since it is a value parameter. You can think of this as an initialized local variable. The other two parameters are var parameters. The global variable b is referred to as a while we are in the procedure. We note this in the diagram by writing the local name a next to the box for global b. Likewise, for the third parameter: global a is referred to as b. We add one more memory location for the local variable c declared within the procedure.

Make sure you understand why all the variables in the diagram have the values they do at this point.

global names		*local names*	
a	4	b	{var parameter b is an alias for global a}
b	5	a	{var parameter a is an alias for global b}
c	7		
d	9		
	9	x	{value parameter x gets a copy of global c}
		c	

Handle assignment statements within the procedure

global names		*local names*	
a	9	b	`b := x;` {line 10}
b	6	a	`a := c - b;` {line 9}
c	7		
d	9		
	14	x	`x := d + 5;` {line 11}
	10	c	`c := 10;` {line 8}

The assignments of course are really done in the order that they occur in the program (see the listing at the start of this section). Note that since d is neither the name of a local variable nor the name of a parameter, it refers to the global variable.

Return from call

global names		*local names*
a	9	b
b	6	a
c	7	
d	9	
	14	x
	10	c

Variables and names allocated within the procedure—i.e. those shaded above—vanish after return to the main program.

Output

```
Answer is:   9   6   7   9
```

NAME

dbx – debugger

SYNOPSIS

dbx [**–r**] [**–i**] [**–k**] [**–I** *dir*] [**–c** *file*] [*objfile* [*coredump*]]

DESCRIPTION (edited for E 77S and CS 9A)

Dbx is a tool for source level debugging and execution of programs under UNIX. The *objfile* is an object file produced by *f77*.

The object file contains a symbol table that includes the name of the all the source files translated by the compiler to create it. These files are available for perusal while using the debugger.

If a file named ‘‘core’’ exists in the current directory or a *coredump* file is specified, *dbx* can be used to examine the state of the program when it faulted.

If the file ‘‘.dbxinit’’ exists in the current directory then the debugger commands in it are executed. *Dbx* also checks for a ‘‘.dbxinit’’ in the user’s home directory if there isn’t one in the current directory.

The command line options and their meanings are:

–r Execute *objfile* immediately. If it terminates successfully *dbx* exits. Otherwise the reason for termination will be reported and the user offered the option of entering the debugger or letting the program fault. *Dbx* will read from ‘‘/dev/tty’’ when **–r** is specified and standard input is not a terminal.

–i Force *dbx* to act as though standard input is a terminal.

–I *dir* Add *dir* to the list of directories that are searched when looking for a source file. Normally *dbx* looks for source files in the current directory and in the directory where *objfile* is located. The directory search path can also be set with the **use** command.

–c *file* Execute the *dbx* commands in the *file* before reading from standard input.

Unless **–r** is specified, *dbx* just prompts and waits for a command.

Execution and Tracing Commands

run [*args*] [< *filename*] [> *filename*]
rerun [*args*] [< *filename*] [> *filename*]

Start executing *objfile*, passing *args* as command line arguments; < or > can be used to redirect input or output in the usual manner. When **rerun** is used without any arguments the previous argument list is passed to the program; otherwise it is identical to **run**. If *objfile* has been written since the last time the symbolic information was read in, *dbx* will read in the new information.

trace [**in** *subroutine/function*] [**if** *condition*]
trace *source-line-number* [**if** *condition*]
trace *subroutine/function* [**in** *subroutine/function*] [**if** *condition*]
trace *expression* **at** *source-line-number* [**if** *condition*]
trace *variable* [**in** *subroutine/function*] [**if** *condition*]

Have tracing information printed when the program is executed. A number is associated with the command that is used to turn the tracing off (see the **delete** command).

The first argument describes what is to be traced. If it is a *source-line-number*, then the line is printed immediately prior to being executed. Source line numbers in a file other than the current one must be preceded by the name of the file in quotes and a colon, e.g. "mumble.p":17.

If the argument is a subroutine or function name then every time it is called, information is printed telling what routine called it, from what source line it was called, and what parameters were passed to it. In addition, its return is noted, and if it's a function then the value it is returning is also printed.

If the argument is an *expression* with an **at** clause then the value of the expression is printed whenever the identified source line is reached.

If the argument is a variable then the name and value of the variable is printed whenever it changes. Execution is substantially slower during this form of tracing.

If no argument is specified then all source lines are printed before they are executed. Execution is substantially slower during this form of tracing.

The clause ‘‘**in** *subroutine/function*’’ restricts tracing information to be printed only while executing inside the given subroutine or function.

Condition is a boolean expression and is evaluated prior to printing the tracing information; if it is false then the information is not printed.

stop if *condition*
stop at *source-line-number* [**if** *condition*]
stop in *subroutine/function* [**if** *condition*]
stop *variable* [**if** *condition*]

Stop execution when the given line is reached, subroutine or function called, variable changed, or condition true.

status [> *filename*]

Print out the currently active **trace** and **stop** commands.

delete *command-number ...*

The traces or stops corresponding to the given numbers are removed. The numbers associated with traces and stops are printed by the **status** command.

cont Continue execution from where it stopped. The process is continued as though it had not been stopped.

Execution cannot be continued if the process has ‘‘finished’’, that is, called the standard procedure ‘‘exit’’. *Dbx* does not allow the process to exit, thereby letting the user to examine the program state.

step Execute one source line.

next Execute up to the next source line. The difference between this and **step** is that if the line contains a call to a subroutine or function the **step** command will stop at the beginning of that block, while the **next** command will not.

return [*subroutine*]

Continue until a return to *subroutine* is executed, or until the current subroutine returns if none is specified.

call *subroutine(parameters)*

Execute the object code associated with the named subroutine or function.

Printing Variables and Expressions

Names are resolved first using the static scope of the current function, then using the dynamic scope if the name is not defined in the static scope. If static and dynamic searches do not yield a result, an arbitrary symbol is chosen and the message "[using *qualified name*]" is printed. The name resolution procedure may be overridden by qualifying an identifier with a block name, e.g., "*module.variable*".

Expressions are specified with an approximately common subset of C and Pascal (or equivalently Modula-2) syntax. Array expressions are subscripted by brackets ("[ ]").

Types of expressions are checked; the type of an expression may be overridden by using "*type-name*(*expression*)".

assign *variable* = *expression*
: Assign the value of the expression to the variable.

dump [*subroutine*] [> *filename*]
: Print the names and values of variables in the given subroutine, or the current one if none is specified. If the subroutine given is ".", then the all active variables are dumped.

print *expression* [**,** *expression* ...]
: Print out the values of the expressions.

whatis *name*
: Print the declaration of the given name, which may be qualified with block names as above.

which *identifier*
: Print the full qualification of the given identifier, i.e. the outer blocks that the identifier is associated with.

up [*count*]
down [*count*]
: Move the current function, which is used for resolving names, up or down the stack *count* levels. The default *count* is 1.

where Print out a list of the active subroutines and functions.

whereis *identifier*
: Print the full qualification of all the symbols whose name matches the given identifier. The order in which the symbols are printed is not meaningful.

Accessing Source Files

/*regular expression*[/]
?*regular expression*[?]
: Search forward or backward in the current source file for the given pattern.

edit [*filename*]
edit *subroutine/function-name*
: Invoke an editor on *filename* or the current source file if none is specified. If a *subroutine* or *function* name is specified, the editor is invoked on the file that contains it. Which editor is invoked by default depends on the installation. The default can be overridden by setting the environment variable EDITOR to the name of the desired editor.

file [*filename*]
: Change the current source file name to *filename*. If none is specified then the current source file name is printed.

func [*subroutine/function*]

Change the current function. If none is specified then print the current function. Changing the current function implicitly changes the current source file to the one that contains the function; it also changes the current scope used for name resolution.

list [*source-line-number* [**,** *source-line-number*]]
list *subroutine/function*
List the lines in the current source file from the first line number to the second inclusive. If no lines are specified, the next 10 lines are listed. If the name of a subroutine or function is given lines *n-k* to *n+k* are listed where *n* is the first statement in the subroutine or function and *k* is small.

use *directory-list*
Set the list of directories to be searched when looking for source files.

Command Aliases and Variables

alias *name name*
alias *name* "*string*"
alias *name* (*parameters*) "*string*"
When commands are processed, dbx first checks to see if the word is an alias for either a command or a string. If it is an alias, then dbx treats the input as though the corresponding string (with values substituted for any parameters) had been entered. For example, to define an alias "rr" for the command "rerun", one can say

alias rr rerun

To define an alias called "b" that sets a stop at a particular line one can say

alias b(x) "stop at x"

Subsequently, the command "b(12)" will expand to "stop at 12".

unalias *name*
Remove the alias with the given name.

Miscellaneous Commands

gripe Invoke a mail program to send a message to the person in charge of *dbx*.

help Print out a synopsis of *dbx* commands.

quit Exit *dbx*.

sh *command-line*
Pass the command line to the shell for execution. The SHELL environment variable determines which shell is used.

source *filename*
Read *dbx* commands from the given *filename*.

FILES

a.out	object file
.dbxinit	initial commands

SEE ALSO

cc(1), f77(1), pc(1), mod(l)

COMMENTS

Some problems remain with the support for individual languages. Fortran problems include: inability to assign to logical, logical∗2, complex and double complex variables; inability to represent parameter constants which are not type integer or real; peculiar representation for the values of dummy subroutines.

Debugging with dbx

Bill Tuthill

Sun Microsystems, Inc.
2550 Garcia Avenue

Kevin J. Dunlap

Computer Systems Research Group
University of California
Berkeley, CA 94720

Introduction

This short paper discusses *dbx*, a symbolic debugger that is vastly superior to *adb*. It may be as good as the debuggers you remember from those non- UNIX† systems you worked on before. The advantage of symbolic debuggers is that they allow you to work with the same names (symbols) as in your source code.

Like *adb*, *dbx* is interactive and line-oriented, but *dbx* is a source-level rather than an assembly-level debugger. It allows you to determine where a program crashed, to view the values of variables and expressions, to set breakpoints in the code, and to run and trace a program. Source code may be in C, Fortran, or Pascal.

Mark Linton wrote *dbx* as his master's thesis at UC Berkeley. Along with Eric Schmidt's Berknet, *dbx* is among the most successful master's theses done on UNIX. Since *dbx* required changes to the symbol tables generated by the various compilers, you need to compile programs for debugging with the *−g* flag. For example, C programs should be compiled as follows:

% cc −g *program*.c −o *program*

Programs compiled with the *−g* option have good symbol tables, while programs compiled without *−g* have old-style symbol tables intended for *adb*. Stripped programs have no symbol tables at all. Invoke the debugger as follows, where *program* is the pathname of the executable file that dumped core:

% dbx *program*

The core image should be in the working directory; if it isn't, specify its pathname in the argument after the program name. Among the great advances of *dbx* is that it has a help facility; type the *help* request to see a list of possible requests. You can obtain help on any *dbx* request by giving its name as an argument to *help*.

† UNIX is a trademark of Bell Laboratories.

Examining Core Dumps

Much of the time, programmers use *dbx* to find out why a program dumped core. As an example, consider the following program *dumpcore.c*, which dereferences a NULL pointer. This is a legal operation on VAX/UNIX, but not on VAX/VMS or on MC68000-based UNIX systems, on one of which this example was run:

```
#include <stdio.h>

#define LIMIT 5

main()              /* print messages and die */
{
     int i;

     for (i = 1; i <= 10 ; i++) {
          printf("Goodbye world! (%d)\n", i);
          dumpcore(i);
     }
     exit(0);
}

int *ip;

dumpcore(lim)            /* dereference NULL pointer */
int lim;
{
     if (lim >= LIMIT)
          *ip = lim;
}
```

The program core dumps because of a segmentation violation or memory fault — on most machines it is illegal to assign to address zero. Once the program has produced a core dump, here's how you can find out why the program died:

```
% dbx dumpcore
dbx version 3.17 of 4/24/86 15:04 (monet.Berkeley.EDU).
Type 'help' for help.
reading symbolic information ...
[using memory image in core]
(dbx) where
dumpcore.dumpcore(lim = 5), line 22 in "dumpcore.c"
main(0x1, 0x7fffe904, 0x7fffe90c), line 11 in "dumpcore.c"
```

The *where* request yields a stack trace. As you can see, the *dumpcore()* routine was called from line 11 of the program, with the argument *lim* equal to 5. You can look at the *dumpcore()* procedure by invoking the *list* request as follows:

```
(dbx) list dumpcore
  18   dumpcore(lim)          /* dereference NULL pointer */
  19   int lim;
  20   {
  21        if (lim >= LIMIT)
  22             *ip = lim;
  23   }
```

We immediately suspect that the program's failure had something to do with **ip*, so we use the *print* request to retrieve the value of the pointer and what it points to:

```
(dbx) print *ip
reference through nil pointer
(dbx) print ip
(nil)
```

This tells us the program has dereferenced a null pointer. It is possible to run the program again from inside the debugger. The first line tells you name of the running program, and successive lines give output from the program:

```
(dbx) run
Goodbye world! (1)
Goodbye world! (2)
Goodbye world! (3)
Goodbye world! (4)
Goodbye world! (5)

Bus error in dumpcore.dumpcore at line 22
  22          *ip = lim;
(dbx) quit
```

In this example the program dies with a Bus error at line 22. This method of running the program does not produce a core dump, but the *where* request will still behave properly, because the debugger is in the same state as if it had just read the core file.

Setting Breakpoints

With *dbx* you can set breakpoints before each line of a program, not just at function and procedure boundaries, as with *adb*. The *stop* request sets a breakpoint. After setting a breakpoint, use the *run* request to execute the program. The *cont* request continues execution from the current stopping point until the program finishes or another breakpoint is encountered. The *step* request executes one source statement, following any function calls. The *next* request executes one source statement, but does not stop inside any function calls. The *status* request lists active breakpoints, while the *delete* request removes them if required.

The *stop* request can take a conditional expression to avoid needless single-stepping. We will use a conditional in our example to make things simpler. Of course you can use *print* and *list* requests at any time during statement stepping if you want to print the value of variables or list lines of source code. This sample session shows a mixture of requests as we verify that the program fails when it tries to assign to **ip*:

```
(dbx) stop at 10 if (i == 5)
[1] if i = 5 { stop } at 10
(dbx) run
Goodbye world! (1)
Goodbye world! (2)
Goodbye world! (3)
Goodbye world! (4)
[1] stopped in main at line 10
  10            printf("Goodbye world! (%d)\n", i);
(dbx) next
Goodbye world! (5)
stopped in main at line 11
  11            dumpcore(i);
(dbx) step
stopped in dumpcore at line 21
  21        if (lim >= LIMIT)
(dbx) step
stopped in dumpcore at line 22
  22            *ip = lim;
(dbx) step
Bus error in dumpcore.dumpcore at line 22
  22        *ip = lim;
```

Running the program with breakpoints assures us that our intuition was correct. We shouldn't be assigning anything to a null pointer — *ip* should have been initialized to point at an object of the proper type. To exit from the debugger, use the *quit* request.

It is possible to set variables from inside *dbx*. The previous breakpoint session, for example, could have gone like this:

```
% dbx dumpcore
dbx version 3.17 of 4/24/86 15:04 (monet.Berkeley.EDU).
Type 'help' for help.
reading symbolic information ...
[using memory image in core]
(dbx) stop at 10
[1] stop at 10
(dbx) run
Running: dumpcore
stopped in main at line 10
  10            printf("Goodbye world! (%d)\n", i);
(dbx) assign i = 5
(dbx) next
Goodbye world! (5)
stopped in main at line 11
  11            dumpcore(i);
(dbx) next
Bus error in dumpcore.dumpcore at line 22
  22        *ip = lim;
```

It is often useful to assign new values to variables to draw conclusions about alternative conditions. We can't fix the bug in this program, however, because there is no declared variable to which *ip* should point.

Conclusion

Expressions in *dbx* are similar to those in C, except that there is a distinction between / (floating-point division) and *div* (integer division), as in Pascal. The table on the following page shows *dbx* requests organized by function:

Like *adb*, *dbx* can disassemble object code. It can also examine object files and print output in various formats; but *dbx* requires the proper symbol tables, so *adb* is more useful to examine arbitrary binary files. The most important thing *adb* can do that *dbx* cannot is to patch binary files — *dbx* has no write option. Despite these shortcomings, *dbx* is much easier to use than *adb*, so it contributes much more to individual programmer productivity.

Acknowledgements

Material presented in this document was first presented in ‘‘C Advisor’’, *Unix Review 4*, 1, pp 78−85. The Regents of the University California expresses their gratitude to Unix Review for allowing them to reprint this document.

This document is a good starting point for a more thorough tutorial. Those with the ambition to expand on this document are encouraged to contact the Computer Systems Research Group at ‘‘4bsd-ideas@Berkeley.Edu.’’

Groups of *dbx* Requests	
	execution and tracing
`run`	execute object file
`cont`	continue execution from where it stopped
`trace`	display tracing information at specified place
`stop`	stop execution at specified place
`status`	display active *trace* and *stop* requests
`delete`	delete specific *trace* or *stop* requests
`catch`	start trapping specified signals
`ignore`	stop trapping specified signals
`step`	execute the next source line, stepping into functions
`next`	execute the next source line, even if it's a function
	displaying data
`print`	print the value of an expression
`whatis`	print the declaration of a given identifier or type
`which`	print outer block associated with identifier
`whereis`	print all symbols matching identifier
`assign`	set the value of a variable
	function and procedure handling
`where`	display active procedures and functions on stack
`down`	move down the stack towards stopping point
`up`	move up the stack towards *main*
`call`	call the named function or procedure
`dump`	display names and values of all local variables
	accessing source files and directories
`edit`	invoke an editor on current source file
`file`	change current source file
`func`	change the current function or procedure
`list`	display lines of source code
`use`	set directory list to search for source files
`/.../`	search down in file to match regular expression
`?...?`	search up in file to match regular expression
	miscellaneous commands
`sh`	pass command line to the shell
`alias`	change *dbx* command name
`help`	explain commands
`source`	read commands from external file
`quit`	exit the debugger

Emacs reference

by Drew Roselli ripping off Dan Garcia's idea and expanding upon it

Starting out

To begin, type emacs at the % prompt.

In the command lists below, C means the "control" key. To type the command C-x u, type the control key and x at the same time, then type u. M means the "meta" key. If your terminal doesn't have a meta key, use the escape key followed by a brief pause.

command	*explanation*
`C-x u`	Undo
`C-g`	Go away, or cancel a command
`C-x C-c`	Close session (save files, then exit emacs)
`C-z`	suspend emacs (or C-x C-z)
`C-h`	Help
`?`	"describe mode" (tell what the keys mean in the current mode)

Files

Emacs has auto-saving. Every 300 keystrokes, it will save the file you are editing into a temporary file called #filename#. In addition, every time you edit a file, a backup is made called file~.

When finding a file, use the tab key for filename completion and the space key for a list of possible completions.

command	*explanation*
`C-x C-f`	Find (load) a file
`C-x s`	Save a file (or C-x C-s, but use C-x s)
`C-x C-w`	Write a file
`C-x C-c`	Close session (save files, then exit emacs)

Movement

Mouse and arrow keys will work if your terminal has them.

command	*explanation*
`C-f`	Forward one character
`C-b`	Backward one character
`C-n`	Next line
`C-p`	Previous line
`C-a`	beginning of line
`C-e`	End of line
`C-v`	scroll up
`M-v`	scroll down
`M->`	end of file
`M-<`	beginning of file
`M-C-a`	beginning of function
`M-C-e`	End of function
`C-d`	Delete dharacter
`M-d`	Delete word

Cut and paste

Everything deleted or copied goes to the "Kill Ring" which you can retrieve ("yank back") later. To define a region, set the mark at the beginning and position the cursor at the end.

command	*explanation*
`C-k`	Kill line (copies to the kill ring)
`C-SPC`	set mark (or C-@)
`C-w`	Wipe region (kill region, copied to kill ring)
`M-w`	copy region (copy to kill ring)
`C-y`	Yank back last kill
`M-y`	Yank pop (yank back previous kill from kill ring)

Windows and buffers

command	*explanation*
`C-x 2`	split window vertically
`C-x 5`	split window horizontally
`C-x 1`	close all other windows
`C-x 0`	kill the window the cursor is in
`C-x o`	other window (put cursor in the other window)
`C-x b`	switch Buffers
`C-x k`	Kill buffer

Miscellany

I-searches are incremental searches—that is, emacs searches for the pattern as you type. When you find what you are looking for, type the escape key.

command	*explanation*
`C-s`	i-search forward
`C-r`	i-search backward
`M-%`	query replace
`M-;`	go to line
`C-x m`	mail
`M-x rmail`	read mail (also send and reply)
`M-x shell`	will start a Unix shell in emacs
`M-x man`	manual entry (same as man, but in *editor* mode, not *more* mode)
`M-x run-lisp`	run a Lisp listener in a buffer
`M-x` *command-name*	general way to execute a command

Useful commands for interacting with C

command	*explanation*
`M C-\`	indent region (according to C convention)
`M-x compile`	compiles
``C-x ` ``	(i.e. control-x backquote) next error (will find your compiler errors for you)
`M-x gdb`	runs the gdb debugger and follows the source code with an => in another window

How to learn more

The online help system is extensive. Typing C-h (help) will offer you several different kinds of help. Type C-h again for help with help, etc. Some of the more useful options are

`b`	describe all the key bindings
`i`	info mode. This is the entire emacs manual in hypertext (it contains its own instructions).

Customization

Emacs allows you to create a .emacs file and enter customizations such as the following. The .emacs file is executed when emacs is invoked. Customizations can be set differently for different modes. Modes are set by the file name, i.e., any file ending with .c will be assumed to be a C program file and will invoke the C mode automatically. A sample .emacs file appears on the next page.

Sample .emacs file

```
(setq inhibit-startup-message t)                    ; don't print emacs startup message
(setq default-major-mode text-mode)

; rebind some keys for convenience
(global-set-key "\M-h" 'help-for-help)              ; reset help to M-h key.
(global-set-key "\C-h" 'delete-backward-file)       ; C-h (backspace) deletes
(global-set-key "\M-r" 'rmail)                      ; read mail mode
(global-set-key "\M-m" 'mail)
(global-set-key "\M-s" 'save-buffer)                ; save current buffer
(global-set-key "\M-g" 'goto-line)
(global-set-key "\M-c" 'compile)                    ; to avoid typing "make" or "compile"
(global-set-key "\M-q" 'query-replace)
(global-set-key "\M-i" 'indent-region)              ; useful for indenting programs

; set default indentations for C mode
(defconst c-indent-level 4)
(defconst c-brace-offset 0)
(defconst c-arg-decl-indent 0)
(defconst c-contiued-statement-offset 4)
(defconst c-auto-newline nil)
(defconst c-label-offset -2)

; set return to indent automatically while in C mode
(setq c-mode hook (function (lambda nil
  "Customize c-mode"
   (interactive)
   (setq-indent-tabs-mode t)
   (define-key "\r" 'c-newline-indent)
   (define-key "\n" 'c-newline-indent)
   (define-key c-mode-map "\n" 'c-newline-indent))))

(defun c-newline-indent (arg)
   "user defined function  to make new line and indent"
    (interactive "P")
    (newline arg)
     (c-indent-line))

(setq-default compile-command "make")
```

Notes from tutoring sessions

Notes from tutoring sessions

Notes from tutoring sessions

Notes from tutoring sessions

Notes from tutoring sessions