

Structured Programming in Pascal

Tak Auyeung, Ph.D.

December 3, 2003

Change log:

- TA 20031203 0929: add assignment 18.5
- TA 20031119 0905: add assignment 18.4
- TA 20031029 0839: add assignment 14.4
- TA 20031022 0815: add assignment 13.4
- TA 20031008 0911: add assignment 12.3
- TA 20030918 1233: add assignment 10.7
- TA 20030908 2038: change second assignment, refer to 8.3 for more details
- TA 20030905 2328: add figure 6.1 to illustrate nested conditional statements in a flowchart form
- TA 20030827 2316: add appendix A to list all homework assignments
- TA 20030512 0016: add section 22.3 for linked lists (an application of dynamically allocated memory)
- TA 20030509 0948: add chapter 22 for pointers
- TA 20030507 0936: add section 21.4 to help clarify recursion a little
- TA 20030505 0052: add chapter 20 for convenient Pascal constructs and chapter 21 for recursion
- TA 20030416 1649: add chapter 19 for file I/O, need to add examples at the end
- TA 20030401 1746: add chapter 18 for string discussion, work in progress.
- TA 20030330 1336: add chapter 17 for abstract data type
- TA 20030324 1412: add section 15.4 as an assignment.
- TA 20030323 1356: add chapter 16.
- TA 20030316 0020: add chapter 15, not done yet!
- TA 20030316 0020: fixed a few problems in chapter 14
- TA 20030312 2131: add chapter 13
- TA 20030310 1154: add chapter 12
- TA 20030310 0927: add assignment at the end of chapter 9.
- TA 20030309 0251: add chapter 14, not done yet!
- TA 20030303 1551: add chapter 11

- TA 20030223 1629: add chapter 10
- TA 20030222 2149: add chapter 9
- TA 20030219 1205: fix addition sample in subsection 8.1.4
- TA 20030215 1356: add subsection 8.1.6
- TA 20030212 0850: add subsection 8.1.4
- TA 20030211 2328:
 - change part title of III
 - add new chapter 8 for integer representation (not done)
 - add new chapter 9 for real numbers (not done)
- TA 20030211 2318: add change log to pin point change manually

Contents

0.1	Copyright Notice	11
I	Background	13
1	Programming in General	15
1.1	Not Just for Computers!	15
1.2	What Can a Program Do?	15
2	Tools for Programming in Pascal	17
2.1	Getting Free Pascal	17
2.2	Your First Program	18
2.3	Command Line Interface	19
2.4	Alternative Method to Invoke FreePascal	21
2.5	Online Resources	21
II	Basic Pascal	23
3	Printing to the Output	25
3.1	begin...end	25
3.2	writeln	25
3.3	Adding More Statements	26
3.4	Commenting	26
3.5	Formatting	26
4	Simple Variables	29
4.1	Tedious Counting	29
4.2	Variables	30
4.2.1	Name	30
4.2.2	Type	30
4.2.3	Value	30
4.2.4	Declaring a Variable	31
4.3	Assignment Statements	31
4.4	Sample Program Revised	32

5	Reading from a “File”	35
5.1	What You Can Do	35
5.1.1	read	35
5.1.2	readln	35
5.1.3	eof	36
5.2	Standard Input and Standard Output	36
 III Problem Solving with Numbers		 39
6	Control Structures	41
6.1	Conditions	41
6.1.1	Comparisons	41
6.1.2	Compound Logical Operators	42
6.2	Control Constructs	42
6.2.1	Conditional Statement	43
6.2.2	Prechecking Iteration	44
6.2.3	Postchecking Iteration	45
6.3	The Block Statement	45
6.4	Nested Statements	46
6.4.1	Conditional Statements	46
7	Examples of Problem Solving	49
7.1	Error Checking	49
7.1.1	First Cut	49
7.1.2	Refinement	50
8	Integer Representation	53
8.1	Binary Numbers	53
8.1.1	Revisit Base-10 (Decimal) Numbers	53
8.1.2	Breaking Down a Binary Number	53
8.1.3	Converting from Decimal to Binary	54
8.1.4	Addition	54
8.1.5	Multiplication	55
8.1.6	Negative Values and Negation	56
8.2	Range of Values	57
8.3	Assignment (200 pts, due one week from assignment date)	58
8.3.1	What Does the Program Do?	58
8.3.2	The Approach	58
8.3.3	Hints	58
8.3.4	Trinary Conversion Example	58
8.3.5	How to Turn it In?	59

<i>CONTENTS</i>	7
9 Real Numbers	61
9.1 What is Real ?	61
9.1.1 The Binary Point	61
9.1.2 Exponent	62
9.1.3 Actual Implementation	62
9.2 Real Vs. Integer	63
9.2.1 Efficiency	63
9.2.2 Range of Magnitude	63
9.2.3 Fractions	63
9.3 Operators for Real Versus Integer	63
9.4 Assignment	64
9.4.1 Getting a Random Number	64
9.4.2 Hint	64
9.4.3 How to submit?	65
IV Subroutines	67
10 Simple Subroutines	69
10.1 Example	69
10.2 Why Use Subroutines	70
10.3 Introduction to Subroutine	70
10.4 Global Variables and Subroutines	71
10.5 Our Example	72
10.6 What's Next?	73
10.7 Assignment	73
10.7.1 The New Program	73
10.7.2 How to proceed?	74
10.7.3 How to turn it in	75
11 Parameter Passing for Subroutines	77
11.1 What is a parameter?	77
11.2 Parameter Definition Syntax	78
11.3 First Contact with Scope	78
11.4 Invoking Subroutines with Parameters	79
11.5 Nested Invocation	80
12 Functions	83
12.1 What is a Function? How is It Different from a Procedure?	83
12.2 More about the Syntax	84
12.3 Homework Assignments (100 points, due one week from assignment date)	85

13 Local Variables	87
13.1 The Concept	87
13.2 The Syntax	87
13.3 Local Variables and Parameters Passed by Value	88
13.4 Homework assignment, 100 points (due in one week)	89
V Congregates	91
14 Arrays	93
14.1 What is an Array?	93
14.2 Pascal Syntax	93
14.3 Common Array Techniques	94
14.3.1 Going Through an Array	94
14.4 Homework Assignment (200 points)	95
14.4.1 Requirements	95
14.4.2 How to turn in	96
14.4.3 Warning	96
15 Array Related Algorithms	97
15.1 Searching in an Unsorted Array (Linear Search)	97
15.2 Searching in a Sorted Array	98
15.3 Sorting	99
15.4 Homework Assignment	101
15.4.1 Hint	102
15.4.2 How to submit?	102
16 Records	103
16.1 Concepts	103
16.2 Syntax	104
16.2.1 <i>Standalone</i> Record Definition and Reference	104
16.2.2 <i>Named</i> Record Definition and Reference	105
16.2.3 <i>Named</i> Type Definition and Reference	105
17 Abstract Data Types	107
17.1 Circular Queue	107
17.1.1 General Queue (FIFO)	107
17.1.2 Circular Queue	107
17.1.3 Pascal <code>record</code> Implementation	108
17.1.4 Operations for Circular Queues in Pascal	108
17.2 Using <code>CirQ</code> <i>not</i> As an ADT	110
17.3 Using <code>CirQ</code> as an ADT	112
17.4 Homework Assignment	113
17.4.1 Stack Revisited	114
17.4.2 Interactive Tester	114
17.4.3 How to Turn in	114

VI	Text and File Processing	115
18	Strings	117
18.1	Character Encoding	117
18.2	The <code>char</code> Type	118
18.3	String	119
18.3.1	String Comparison	119
18.3.2	String Concatination	119
18.3.3	String Length	119
18.3.4	Indexing in a String	120
18.4	Homework Assignment (200 points)	120
18.4.1	What to do?	120
18.4.2	Requirements	122
18.4.3	How to turn in	122
18.5	Extra Credit Assignment	122
18.5.1	What to do?	122
18.5.2	How to modify the program?	123
18.5.3	Tips	123
18.5.4	How to submit?	124
19	File Operations	125
19.1	Files	125
19.2	Pascal File Operations	126
19.2.1	The <code>File</code> Type	126
19.2.2	Associating a <code>File</code> variable to a <code>File</code>	127
19.2.3	Opening a file	127
19.2.4	Moving to a Position	127
19.2.5	Knowing more about a file and file operations	128
19.2.6	Reading and writing	128
19.2.7	Flushing a file	128
19.2.8	Closing a file	129
19.2.9	Deleting a file	129
19.3	Merge Sort	129
19.3.1	Splitting	129
19.3.2	Merging	130
19.3.3	The Loop	130
VII	Advanced Discussion	131
20	Handy but not Necessary Constructs	133
20.1	<code>for</code>	133
20.2	<code>case</code>	134

21 Recursion	135
21.1 Call Frame	135
21.2 Self-Similar Patterns	136
21.2.1 Factorial	136
21.3 Recursive Function	136
21.3.1 Tracing Recursion	137
21.4 Demystifying Recursion	138
21.5 Assignment (Extra Credit 200 Points)	138
22 Dynamically Allocated Memory and Pointers	141
22.1 Pointers	141
22.2 Allocating and Deallocating Memory	142
22.3 Practical Use of Pointers	143
22.3.1 A Linked List	143
VIII Appendices	149
A Assignments	151

0.1 Copyright Notice

All materials in this document are copyrighted. The author reserves all rights. Infringements will be prosecuted at the maximum extent allowed by law.

You are permitted to do the following:

1. add a link to the source of this document at www.mobots.com
2. view the materials online at www.mobots.com.
3. make copies (electronic or paper) for *personal* use only, given that:
 - (a) copies are not distributed by *any* means, you can always refer someone else to the source
 - (b) copyright notice and author information be preserved, you cannot cut and paste portions of this document without also copying the copyright notice

Part I

Background

Chapter 1

Programming in General

You are reading this because you want to learn how to program. This chapter introduces some very general concepts so you understand more about programming in general.

1.1 Not Just for Computers!

Although the word “programming” implies computer programming, concepts used in programming are much more general than their application in computers. Consider instructions in a recipe, or instructions to assemble a piece of furniture (like a bookshelf). Those are also programs, except they were written to be followed by people, not executed by computers.

A program, in its most general sense, is merely a sequence of instructions arranged to solve a particular problem.

If you know how to teach a younger sibling or a child *how* to perform a task, you already know, to an extent, how to program.

A program is simply a sequence of instructions arranged to solve a particular problem.

1.2 What Can a Program Do?

As mentioned in the previous section, a program is a sequence of instructions. What can a program do, then?

A program does *exactly* what the programmer specifies. A sophisticated program written by an experienced programmer can do a lot (correctly), a half-baked program written by a beginner is likely to do simple tasks wrong.

Everything that a computer does is the result of executing one or more programs. This should give you some idea of what kind of complicated tasks can be done by a program. In this class, however, we will focus more on simple programs that are more educational than practical.

Chapter 2

Tools for Programming in Pascal

In this course, we use Free Pascal to write and test programs. As the name implies, Free Pascal is 100% free for you to download, distribute and install. The home site of Free Pascal is <http://www.freepascal.org>.

2.1 Getting Free Pascal

You can download Free Pascal from the internet. Go to <http://www.us.freepascal.org/download.html>, then click on the link to your operating system. Most of you will click on “Win32” for most versions of Windows.

Once you are at the Win32 download page, you can right-click on the link to `w32???.zip`, where ??? is the version number (106 at the time this is written). From the pop-up menu, select “Save Link As File” or “Save Target as File” (depending on your browser). Pick a place to save the file, I suggest `C:\Windows\Temp`.

You need a utility program to open a ZIP file. Some of you may have WinZIP or other ZIP utility programs installed already, others may not know what to with a ZIP file at all. Don’t worry, just follow these instructions:

- right-click <http://www.drta.org/util/unzip.exe>, and select “Save Link As File”.

Unless you know how to change environment variables, you need to save this file as `unzip.exe` in a the Windows folder. It is most likely `C:\Windows` for most versions of Windows, and `C:\Winnt` for NT4 and Windows 2000.

- Click the “Start” button, then select “Run...”, in the dialog box, type `command` and press ENTER.
- Change directory to where you saved the ZIP file. If you followed my instructions, it should be in `C:\Temp`. Type the following command:

```
cd \Windows\Temp
```

- Now unzip the file with the following command (replace ??? with an actual version number):

```
unzip w32???.zip
```

- After the ZIP file is uncompressed, type the following command:

```
install
```

- Accept all default values in the installation screen, click the “Continue” button.

Once Free Pascal is installed, you can invoke it using different methods. For convenience, I suggest you make a shortcut on the desktop. This is rather easy to do.

- Right-click any where on the desktop, click “New”, then select “Shortcut”
- Either browse to the folder or simply enter

```
C:\pp\bin\win32\fp.exe
```

- Click “Next”, then enter “Free Pascal” as the name.
- Click “Finish” and you are done!

2.2 Your First Program

It is easier to do your homework assignments if you can view the FreePascal or Borland Pascal screen and the browser side-by-side. If you are using Borland Pascal from the lab, it defaults to full screen mode. You can turn it into a window by pressing **alt-ENTER**. However, the computers at the lab are set up to make it easier for web authoring students (and harder for you). You need to do the following:

1. turn Borland Pascal to a window by **alt-ENTER**
2. click on the upper left “DOS” icon
3. select “properties”
4. deselect QuickEdit (uncheck the check box)
5. press OK
6. select apply changes to all future invocations

7. press OK

Furthermore, the screens are defaulted to 800x600. If you have good eyes, you can consider changing the resolution to 1024x768. A higher resolution means effectively smaller fonts, but you can also fit more information on one screen (such as the browser *and* FreePascal) at once. Another trick is to change the font of your DOS box (with Borland Pascal or FreePascal running in it). This is done by the following steps:

1. click on the upper left “DOS” icon
2. select “properties”
3. click the “Font” tab
4. select a smaller font, I suggest you do not go below 6x8
5. click OK

You have to do this every time you log in at the lab because those machines reset the defaults after every reboot.

With Free Pascal installed, you can now write your first program. Invoke Free Pascal (use the shortcut set up in the previous section). By default, Free Pascal opens a blank editor screen for you.

Type the following program in the editor. Be sure to observe the punctuations.

```
begin
  writeln('test!')
end.
```

You have to save a program before running it. Click the “File” menu and select “Save”. Although the filename is not important, I will make sure it is up to eight characters long with an extension of `.PAS`.

Now click the “Run” menu and select “Run”. Your screen will flicker a little when the program runs (because it is completed so quickly). To see the output of your program, click the “Debug” menu, then select “Output”, you should see `test!` in this screen.

You have just written your first Pascal program that did something!

When you are done, click the “File” menu and select “Exit”.

2.3 Command Line Interface

Some of you may have difficulties getting the IDE (integrated development environment) working, especially in Windows XP. This is not a big problem. FreePascal is distributed with a command line compiler that is very easy to use, and that works in Windows XP just fine.

You do need your own text editor for editing the program. You *can* use the built-in DOS editor `edit` or the Windows editor `notepad`. You can also download a free DOS editor with syntax highlighting from <http://setedit.sourceforge.net>.

In order for this to work, you need to add the path to FreePascal to your system path. Follow these steps (only need to do this once):

1. Using the Classic View, click the “Start” button, then “Control Panel”, then “System”. If you are using the category view, click the “Start” button, “Control Panel”, “Performance and Maintenance”, then “System”.
2. click the “Advanced” tab
3. click the “Environment Variables” button (near the bottom)
4. select “PATH” from the variables
5. click “Edit” to change it
6. append (add to the end) `;C:\pp\bin\win32` to the original setting. Be sure to include the semicolon!
7. click “OK” when you are done

After this step, Windows XP knows where to find the command line compiler. You can use a similar trick to add the path to your favorite editor.

When you do your homework assignment, do the following:

1. open a DOS box by clicking the “Start” button, select “run...”, then specify `command`, and press ENTER, change to the directory (use the command `cd`) where your Pascal files are located
2. open one more DOS box if you use a DOS-type editor (such as the built-in `edit`), otherwise, start a notepad
3. in the first DOS box, type

```
ppc386 test.pas
```

to compile a program, then type

```
test
```

to run the compiled program. Replace `test` with the name of your program.

4. if the compiler complains, switch to the editor to edit the Pascal program, make sure you save it first, then repeat the compilation step

2.4 Alternative Method to Invoke FreePascal

Download <http://www.mobots.com/ARC/CIS31/samples/mypas.bat> and save it to a folder where you put your Pascal (.pas) files. If you installed FreePascal at a drive or folder other than the default one, you need to edit this batch file to reflect your custom settings. Use the DOS editor instead of notepad to edit the batch file.

When you are ready to compile a program, say `test.pas`, do the following when the command line interface is at the directory where you put your Pascal files. The following is an example of what you may want to do. (It assumes your Pascal files are in `c:\homework\cis31\project1`, and `mypas.bat` is already saved in that directory.)

```
c:  
cd \homework\cis31\project1  
mypas test.pas  
test
```

Of course, if your program does not compile (with syntax errors), don't bother with running the program.

2.5 Online Resources

You can find documentations for Free Pascal at <http://www.freepascal.org/docs.html>. Do *not* print these documents in the lab! These documents are huge and difficult to understand for a beginner. Use them as reference material instead.

Part II

Basic Pascal

Chapter 3

Printing to the Output

Let us take a closer look at our first program.

```
begin
  writeln('test!')
end.
```

Although this program does not do much, it still illustrates features of Pascal. This chapter breaks this program down to explain each component.

3.1 begin...end

The *keywords* `begin` and `end` act like a pair of parentheses to enclose a Pascal program. Everything between these two keywords is considered the *main program*, which gets executed when the program is run.

Like all other reserved words that we will encounter, the words `begin` and `end` have special meanings.

3.2 writeln

`writeln('test!')` is a statement. It invokes a method called `writeln` and supplies a *parameter* `'test!'` to it. `writeln` means “write line”, it optionally prints something to the output and ends the current line. If a program has anything for `writeln` to output before it ends the current line, the text to print must be enclosed by parentheses.

In our example, we want to print `test!` to the output, therefore it is enclosed by parentheses after `writeln`. Note that we also use the single quote `'` to enclose `test!`. This is because contents enclosed by a pair of single quotes is considered literal. For now, don't worry about what is literal; just remember to enclose what you want to appear in the output in single quotes.

3.3 Adding More Statements

You can extend this program so it outputs multiple lines instead of one. For example, if I want the program to output my address, I can change the program as follows:

```
begin
  writeln('Tak Auyeung');
  writeln('999 Milky Way');
  writeln('Galaxy, UV 99999')
end.
```

I just changed the original `writeln` statement, and added two more `writeln` statements. Note that I use the semicolon (;) to separate the statements. In Pascal, a semicolon is used as a statement *separator*. This is why when I only had one statement, there was no need to use a semicolon.

3.4 Commenting

For simple programs, you probably don't need any help reminding you what the program does. I can probably write hundreds of lines and still remember what each line does (in the context of the program).

Unfortunately, *real* programs have at least thousands of lines. This makes it difficult to just read the program and remember what each statement does in the context of the entire program. Almost all programming languages allow you to add notes to your program. Such notes is called *comments*, they are completely ignored by the computer, so you can use any format you like.

For the test program, I may want to add just a little bit of comments so the program now looks like this.

```
begin
  writeln('Tak Auyeung');      { my name }
  writeln('999 Milky Way');    { street address}
  writeln('Galaxy, UV 99999') { city and state}
  { Of course, you realize this address is
    completely fake. Don't try to send me
    anything via this address! }
end.
```

In Pascal, the curly braces ({}) enclose comments. Comments can span any number of lines. Pascal also accept (* to begin comments and *) to end comments, but that's more typing.

3.5 Formatting

Pascal is a free form language. This means unless they are within single quote pairs, multiple spaces is considered the same as one space, line breaks are also

considered spaces. I could have written the program like the following with no change of behavior:

```
begin
  writeln('Tak Auyeung');
  writeln('999 Milky Way');
  writeln('Galaxy, UV 99999')
end.
```

I can even write everything on one single line. The formatting of a program does not bother the computer at all, but it may bother you. This is especially the case when a program gets large.

As a good habit, you should always indent everything between **begin** and **end** by a fixed number of spaces. The importance of doing this becomes more important when we talk about structured programming.

Chapter 4

Simple Variables

In the previous chapter, we learned how to write a program with multiple statements. So far, the only statement we have learned is the `writeln` statement, which outputs text to the output.

You can write some boring programs with just the `writeln` statement and text to print quoted by single quotes. That's why this chapter will discuss concepts that can create more interesting programs.

4.1 Tedious Counting

Let us assume you have a program that looks like the following:

```
begin
  writeln(' 1. make breakfast');
  writeln(' 2. send kid to school');
  writeln(' 3. pick up coffee');
  writeln(' 4. go to work');
  writeln(' 5. call dentist');
  writeln(' 6. lunch with spouse');
  writeln(' 7. book ticket to visit parents');
  writeln(' 8. leave work');
  writeln(' 9. pick up kids');
  writeln('10. pick up dinner');
  writeln('11. go home');
end.
```

Hardly exciting, I know. Now, imagine that you need to insert an item between “call dentist” and “lunch with spouse”. Although it is as easy as inserting one more `writeln` statement, you have to renumber all the items after “lunch with spouse”. It'd be nice if somehow the program can automatically count the number of points.

With variables, the program can.

4.2 Variables

A variable is an *object* in a program that contains a value that can change during the course of the execution of a program. Every variable has three attributes:

- *Name*: the name of a variable uniquely identifies the variable from all others.
- *Type*: the type of a variable determines what kind of value the variable is allowed to store.
- *Value*: the value of a variable can change.

Now, let us get a little more specific.

4.2.1 Name

The name of a variable is also called an identifier. This is an *alphanumeric* sequence that uniquely identifies the variable. A name must follow these rules:

- must contain at least one character
- the first character must be an underscore or a letter (uppercase or lowercase)
- the rest (if any) must be an underscore, a letter or a digit

For example, `JamesBond` is a valid name, as is `_007`. However, `007` is not a valid name.

Of course, `_` is a valid name, and it is different from `__`. Although such names are allowed, you should avoid them because they are difficult to read.

Pascal, unlike C, C++ and Java, is case insensitive. This means the name `JamesBond` is the same the name `jamesBOND`. I recommend that you stay consistent with you choice of case and do not exploit the case insensitivity of Pascal. This allows an easier transition to case sensitive languages.

Last but not least, you cannot use keywords as names. For example, `begin` is already reserved as a keyword, so you may not use it as the name of a variable.

4.2.2 Type

As mentioned earlier, the type of a variable limits what kind of value a variable may store. For now, let us only be concerned about variables that store integers. `Integer` is a built-in type that Pascal recognizes.

4.2.3 Value

The value of a variable can change as a program executes. However, the value of a variable is limited by the type of a variable. For example, a variable of type `Integer` cannot store the value 23.44 because 23.44 is not an integer.

4.2.4 Declaring a Variable

Special sections of a Pascal program declare variables. A variable must be declared before it can be used. This requirement of declaration is also true for C, C++ and Java. There are languages, such as Visual Basic and Perl, that do not need declaration before use.

A variable declaration section in a Pascal program begins with the keyword `var`, then followed by one or more declarations. The following is an example:

```
var
  counter : Integer;
  age : Integer;
```

You can combine two variable declarations into one line if they have the same type. The previous example can be rewritten as follows.

```
var
  counter, age : Integer;
```

You can put a variable declaration section anywhere before the main program (`begin...end`).

4.3 Assignment Statements

When a variable is declared, all the computer knows is that “there exists a variable of a certain name, and type.” There is, officially, no fixed initial value. Note that I said “no fixed initial value”, I did not say “no initial value”. This means once a variable is declared, it has *some* value, we just don’t know what that is.

This means it is important to initialize a variable before referring to it. Assignment statements are used to store values to a variable. For example, if we want `counter` to begin at 0, we can use the following statement (in the main program, between `begin` and `end`):

```
counter := 0
```

In this statement, the left-hand-side (LHS) is a variable. The symbol `:=` is the assignment symbol, and the right-hand-side (RHS) specifies what value should be stored to it.

What if we want to increase the value of `counter` by one? This can be done by the following statement:

```
counter := counter + 1
```

In this statement, the computer takes a snapshot of `counter` first, and use that snapshot on the RHS. The result of the addition is then stored into `counter` as the new value.

4.4 Sample Program Revised

The following is the modified program.

```

var
  counter : Integer;
begin
  counter := 1;
  writeln(' 1. make breakfast');
  counter := counter + 1;
  writeln(' 2. send kid to school');
  counter := counter + 1;
  writeln(' 3. pick up coffee');
  counter := counter + 1;
  writeln(' 4. go to work');
  counter := counter + 1;
  writeln(' 5. call dentist');
  counter := counter + 1;
  writeln(' 6. lunch with spouse');
  counter := counter + 1;
  writeln(' 7. book ticket to visit parents');
  counter := counter + 1;
  writeln(' 8. leave work');
  counter := counter + 1;
  writeln(' 9. pick up kids');
  counter := counter + 1;
  writeln('10. pick up dinner');
  counter := counter + 1;
  writeln('11. go home');
  counter := counter + 1;
end.

```

Of course, the program is not done yet. All have done is to declare a variable called `counter`, initialize it to one, then add one to it every time I print an item. Somehow, I need to print the *value* of `counter` to the output instead of the fixed numbers.

Instead of using a constant for each item number, we want to use the value in `counter` instead. The original statement is as follows:

```
writeln(' 1. make breakfast');
```

We will change that to the following statement.

```
writeln(counter, '. make breakfast');
```

What have we done here? First of all, there are now two components in `writeln`. The first component is `counter`, and the second component is `'. make breakfast'`. A comma is used to separate the two components.

The first component, `counter`, is *not* quoted in single quotes. This is because we don't want the program to display `counter` at the output, but rather the *value* of `counter`. Remember, anything in single quotes are printed verbatim.

The second component is just like the original text, except it does not include the item number anymore. This is because the item number is now provided by `counter`.

We can now apply the same change to all other `writeln` statements. But wait, why don't we just change one statement, and see if that works first?

Ah, it kind of worked. However, you can see that the first item is not longer aligned with the rest. This is because the original statement had a space before 1 so that we can line it up with items with a two-digit item number. Of course, you can brute force this and just include the space like the following statement:

```
writeln(' ',counter,'. make breakfast');
```

However, this approach only works for items with one-digit item numbers.

You can ask `writeln` to always use two characters to print a number, like the following statement:

```
writeln(counter:2,'. make breakfast');
```

Now, this approach works for all the other `writeln` statements.

You have just written the first program that utilizes a variable!

Chapter 5

Reading from a “File”

Most useful programs do not just print something to the output. Instead, most read data from some source, process the data, and generate some output. This chapter discusses the simplest method to read some information from the standard input device: the keyboard.

5.1 What You Can Do

5.1.1 `read`

The method to read data is simply `read`. It requires at least one variable in parentheses so it knows where the data read should be placed. For example, if you want to read an integer from the keyboard and put the number into the integer variable `mystery`, you can use the following statement:

```
read(mystery);
```

`read` can also read multiple items at once. The following statement reads two integers, `mystery` and `unknown`:

```
read(mystery, unknown);
```

The end user can type the two integers separated by a space or one two lines. `read` knows how to skip spaces and newlines to read the next item. However, if the end user types a non-number, such as the name “Tak”, the Pascal program becomes very confused and will crash.

5.1.2 `readln`

`readln` is very similar to `read`, except that `readln` *skips to end of the current line after the items are read*. For example, assume the program has the following statements.

```
readln(mystery);  
readln(unknown);
```

If the end user types two number on one line, the first number is read into `mystery`, then the program skips everything until the end of line. This means the second number will not be read into `unknown`!

Unlike `read` which *requires* at least one parameter (a variable to read into), `readln` can be used without any parameter. When `readln` is used without any parameter, it simply instructs the computer to skip to the end of the current line.

5.1.3 eof

`eof` stands for end-of-file. This is a very useful function that returns a boolean (true or false) value. Since we have not discussed any control logic, this function is fairly useless for now.

5.2 Standard Input and Standard Output

Let us consider a very simple program as follows.

```
var  
  number1, number2 : integer;  
begin  
  read(number1, number2);  
  writeln(number1+number2)  
end.
```

As you quickly realize, this program reads two integers, add them up and print the sum to the output.

At this point, you have a program that reads from the keyboard and prints to the screen. This program, however, can be used without any keyboard/screen interaction. It turns out `read` and `readln` read from a “standard input” file. Similarly, `writeln` writes to a “standard output” file. Normally, the standard input file is the keyboard, and the standard output file is the screen.

There are cases in which you don’t want to interact with a program. Consider a grading program that takes answers from an exam. and output the scores for each student. It is more convenient if the program can read the answers from a file created by the ScanTron machine instead. It is also more convenient if the output can be sent directly to a spreadsheet file.

In order to do this, you should read section 2.3 first. This is because the IDE (integrated development environment) does not usually allow you to change the standard input and output files.

Go ahead and run this program from a DOS command line interface. You can expect the same behavior as in the IDE. Now here comes the fun part.

Create a text file with one line, and write two numbers on that line, for example:

56 109

Let us assume this file is called `input.txt` and it is in the same directory as the program itself. Let us also assume the program is called `sum.pas`, which creates a file called `sum.exe` after you successfully compile the program.

Now, type the following (verbatim) in a DOS command line interface:

```
sum < input.txt
```

Do you need to type the input? What is the output of the program?

Now, this step is even more fun:

```
sum < input.txt > output.txt
```

Where is the output? What is the content of `output.txt`?

The less than sign, `<` has a special meaning on a command line. It *redirects* the standard input file of the program to the file specified to the right of `<`. Instead of reading from the keyboard, the redirection tells the program to read from the specified file instead.

Similarly, the greater than sign, `>`, redirects the standard output file to the file specified to the right of `>`. With this redirection, the program writes to the specified file instead of the screen.

Part III

Problem Solving with
Numbers

Chapter 6

Control Structures

With variables, input and output, a program can perform some useful tasks. However, most tasks worth doing are not a straight sequence of operations. Many useful tasks that a computer does involve decision making. This chapter discusses the most important decision making constructs in Pascal.

6.1 Conditions

Before we discuss decision making, let us first discuss conditions. A condition is “an expression that evaluates to a boolean value”. Since we know about variables already, the following is an example of a condition (assume `x` is an integer variable):

```
x = 50
```

Note how I used `=` and not `:=`. The `:=` symbol means assignment (copy a value from right to left), the `=` symbol means equality in a comparison. This condition is true if and only if `x` has a value of 50.

At this point, we will only need to understand two types of conditions. The first type is based on comparisons, the second type is based on compound logical operations.

6.1.1 Comparisons

For `Integer` values, Pascal understands the following comparison results. Note that the two sides of comparison operators can be literal constants, integer variables or complicated expressions that evaluate to some integer values.

- `=` means equal. `x = y` is true if and only if `x` has the same value as `y`.
- `<>` means not equal. `x <> y` is true if and only if `x` and `y` evaluate to different values.

- $<$ means less than. $x < y$ is true if and only if x evaluates to a value that is less than the value of y .
- $>$ means greater than. $x > y$ is true if and only if x evaluates to a value that is greater than the value of y .
- $<=$ means less than or equal to. $x <= y$ is true if and only if y is at least as big as x .
- $>=$ means greater than or equal to. $x >= y$ is true if and only if x is at least as big as y .

6.1.2 Compound Logical Operators

Logical operators apply to conditions, and they result in a value that is true or false.

Negation

Negation means flipping true to false, and flipping false to true. This is denoted by `not` in Pascal. For example, $x \neq y$ is exactly the same as `not (x = y)`. The operator `not` requires a condition on its right hand side.

Conjunction

Conjunction means “and” in English. It requires one condition on each side of the operator. A conjunction is true if and only if both sides of the operator evaluate to true. Otherwise, the conjunction is false. Pascal uses the reserved word `and` for conjunction.

For example $x = y$ can be clumsily expressed as $(x <= y) \text{ and } (x >= y)$.

Disjunction

Disjunction means “or” in English. It requires one condition on each side of the operator. A disjunction is true if and only if at least one of the two sides evaluate to true. If both sides evaluate to false, a disjunction is false.

For example, $x <= y$ can be rewritten as $(x < y) \text{ or } (x = y)$.

6.2 Control Constructs

This section introduces the most basic three control constructs in Pascal. These constructs exist in *most* high level computer languages.

6.2.1 Conditional Statement

A conditional statement has two *alternative* statements to execute, depending on the value of a condition. If the condition evaluates to true, the *then* statement is executed. Otherwise, the *else* statement is executed.

The general format of a conditional statement is as follows:

```
if condition then
    then-statement
else
    else-statement
```

In this notation, text appearing in **courier** font are reserved words and should be typed as-is. Text appearing in *italic* font are simply place-holders and should be replaced by something else.

Note that *then-statement* must be a *single* statement. The same applies to *else-statement*.

For example, we may use a conditional statement to indicate whether a variable has a value that is negative:

```
if x < 0 then
    writeln('x is negative')
else
    writeln('x is non-negative')
```

Note how a semicolon is *not* used in this code. This is because the two `writeln` statements are considered a part of the conditional statement. As such, they are *already separated* by the reserved word `else`.

Homework Assignment 1

This assignment is due one week from its official announcement date. It carries 100 points.

Please follow these instructions closely!

Your first assignment involves writing a Pascal program that does the following:

- read three integers
- print the smallest (minimum) of the three integers

Obviously, you'll need at least two variables and a conditional statement. You'll also need `read` and `writeln`.

This is a list of concepts you'll need to use in this assignment:

- `read` or `readln` to read the numbers
- `writeln` to output the smallest number

- comparison operators to compare numbers
- conditional statements to perform alternative statements depending on the comparisons
- variables so you can store values to be compared and printed
- you don't need conjunction
- you don't need nested conditional statements
- you don't need block statements
- you only need two variables (not three)

I'll leave it up to you to design the rest of this program.
To submit your assignment, do the following:

- send to my ARC account at auyeunt@arc.losrios.edu
- attach the Pascal file with the extension `.pas` to the email, do *not* attach the other files, I don't need them
- the subject line of the email should read "CIS31 project1 by *your name*", do not include the double quotes and replace *your name* with your actual name
- do not leave any space between CIS and 31, or between "project" and 1
- carbon copy the email to yourself so you have a copy as well
- you can send this email from any account, I reply the message after I grade it to the originating account

6.2.2 Prechecking Iteration

This construct is more commonly known as a while-loop. A while-loop is a construct for repetition that checks for a condition before each iteration. If this condition is true, another iteration occurs. If this condition is false, there will be no more iterations (and the computer proceeds to the next statement).

The syntax of a while-loop is as follows:

```
while condition do
  statement
```

Similar to the conditional statement, *statement* in the syntax description is a single statement.

The following code adds 20 to a number `sum` until it is at least 200.

```
while sum < 200 do
  sum := sum + 20
```

If `sum` has a value that is greater than or equal to 200 to begin with, say, 215, the loop does not perform a single iteration.

6.2.3 Postchecking Iteration

This is a construct more commonly known as a repeat-until-loop. A repeat-until-loop is also a construct for repetition (like a while-loop), but it checks an *exit condition* at the end of every iteration. If the exit condition is true, the computer exits the loop. Otherwise, the computer executes another iteration.

The syntax of a repeat-until-loop is as follows:

```
repeat
  statements
until exit-condition
```

Note that we use *statements* instead of *statement* between the reserved words `repeat` and `until`. This means we can have any number of statements in a repeat-until-loop.

The following code asks the user to input a number (to variable `inputNumber`) until the user inputs a non-negative number.

```
repeat
  readln(inputNumber)
until inputNumber >= 0
```

6.3 The Block Statement

You may have noticed that the limitation of only allowing one statement as the *then-statement*, *else-statement* or *statement* (of a while-loop). This limitation makes it very difficult to express some useful logic that involves multiple statements.

Fortunately, Pascal has a statement *wrapper* that makes a single statement out of a sequence of statements. This wrapper is called a block statement and it has the following syntax:

```
begin
  statements
end
```

Between the reserved words `begin` and `end`, you can put any number (including zero) of statements. And, yes, the main program is a block statement with a period at the end!

For example, the following code (fragment of a larger program) *attempts* to print numbers from 1 to 20, but it will fail.

```
number := 1;
while number < 21 do
  writeln(number);
  number := number + 1
```

The failure mode is that it keeps printing 1 forever. This is because even though the indentation suggests that the increment of `number` is part of the repeated code, it's not! A while-loop only repeat one statement, and that is the statement that prints the number in our code.

To fix this problem, we modify the program as follows:

```
number := 1;
while number < 21 do
  begin
    writeln(number);
    number := number + 1
  end
```

This time, the program works as expected because Pascal treats everything between and including the reserved words `begin` and `end` as a block statement. This entire block statement is executed for each iteration of the while-loop.

6.4 Nested Statements

Let us revisit nested statements. While nested statements let Pascal programs perform complicated operations out of individually simple constructs, they can also make programs more difficult to understand. This section discusses some of the common issues with nested statements in Pascal programs.

6.4.1 Conditional Statements

For beginners, nested conditional statements are particularly tricky. For example, consider the following example:

```
if x < 0 then
  writeln('x is negative')
else
  writeln('x is non-negative')
```

It is important *not* to use a semi-colon after `writeln('x is negative')`! This is because the reserved word `else` *does not begin another statement*. The reserved word is already acting as a separator between the statment to execute when the condition is true and the statement to execute when the condition is false. If you put a semicolon after `writeln('x is negative')`, you end up with the following program:

```
if x < 0 then
  writeln('x is negative');
else
  writeln('x is non-negative')
```

The compiler chokes. Although it is okay up to the semicolon, it also thinks the semicolon is separating the conditional statement itself from the next statement. Unfortunately, `else` is still is part of the conditional statement. No statement can begin with the reserved word `else`, and this is why the compiler chokes.

Another common issue with conditional statements is how `else` is matched with `if` reserve words. Consider the following *poorly formatted* program.

```
if x < a then
if x > b then
writeln('hahaha')
else
writeln('hehehe')
else if x < c then
writeln('hohoho')
else
writeln('hihihi')
```

This program fragment contains the same number of `ifs`, `thens` and `elses`. This code even compiles! However, can you tell what conditions lead to the printing of “hahaha”, “hehehe”, “hohoho” and “hihihi”?

The reserved word `if` is like an open parenthesis. The reserved word `then` is like a close parenthesis. This means it is going to close the most recent unclosed `if`. The same applies to `else`, it close the most recent unclosed `then`.

If I am to use indentation to indicate how the code is nested, the previous code should be reformatted as follows:

```
if x < a then
  if x > b then
    writeln('hahaha')
  else
    writeln('hehehe')
else
  if x < c then
    writeln('hohoho')
  else
    writeln('hihihi')
```

Ah, much better! Now we understand the program behaves as follows:

- if `x` is less than `a`
 - if `x` is less than `a` and `x` is greater than `b`, print “hahaha”
 - if `x` is less than `a` and `x` is not greater than `b`, print “hehehe”
- else (if `x` is not less than `a`)
 - if `x` is not less than `a` and `x` is less than `c`, print “hohoho”

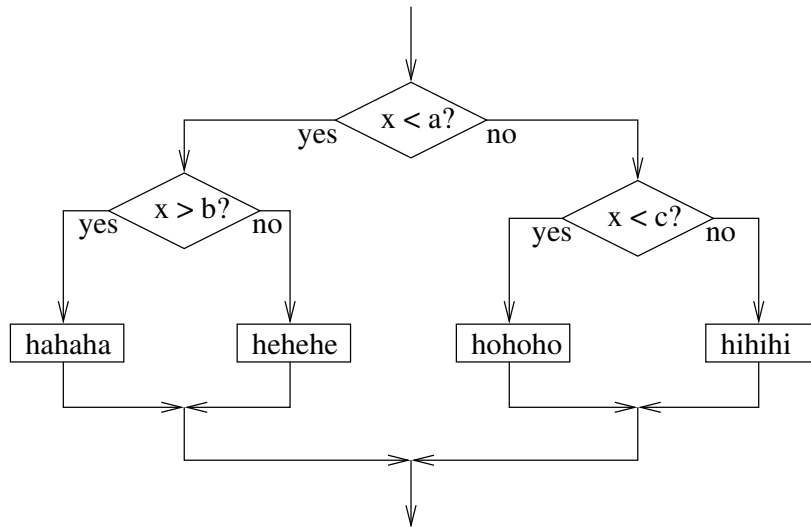


Figure 6.1: The flowchart of a nested structure of conditional statements.

- if x is not less than a and x is not less than c , print “hihihi”

If you prefer a more visual representation, figure 6.1 illustrates the same logic.

Chapter 7

Examples of Problem Solving

This chapter applies concepts that we have learned from previous chapters to solve certain problems. While most of these “problems” are not realistic or have been solved by many people, the exercise of solving these problems helps to illustrate the process of writing programs in Pascal.

7.1 Error Checking

Let us first work on a program that asks a user to enter a number between 1 and 100. The program does not stop until the user has entered a number between 1 and 100. Let us see how we approach this problem.

7.1.1 First Cut

Instead of worrying about the prompt and whether to use `read` or `readln`, let us first focus on the main logic.

We first have to choose among the following for the overall code of this logic:

- A sequence.
- A conditional statement.
- A loop.

Because the end user has to input a number until the number is between 1 and 100, there is repetition. What is done repeatedly? How do we exit the loop?

What each iteration should do to ask the user for an input. This is what need to be done possibly repeatedly.

The repetition stops as soon as the user enters a number between 1 and 100.

There are two kinds of loops, prechecking and postchecking. Is there at least one iteration? (A equivalent question with an opposite answer: Is it possible not to go through one single iteration?) Since we have ask the user to input at least once, we have to perform at least one iteration. This means we cannot check the condition before an iteration, we have no choice but to use a postchecking loop.

Our final code should look like the following:

```
var
  num : integer;
begin
  repeat
    readln(num);
  until (1 <= num) and (num <= 100)
end.
```

7.1.2 Refinement

Although this program appears to work, it is very crude. We can at least add a prompt so the end user knows what to do (the following is just the repeat-until loop and not the entire program):

```
repeat
  writeln('please enter a number between 1 and 100:');
  readln(num)
until (1 <= num) and (num <= 100)
```

This program works, but the prompt and the cursor are not on the same line. We can use `write` instead of `writeln` because `write` does not end a printed line with an end-of-line character.

If an end user does enter numbers out of the range, this program does not produce any warning or error message, it simply prompts the user to enter again. It is advisable to print a warning or error message so the end user understands why he/she is prompted again.

The message should be printed only if the number is out of range. This implies two things. First, the error message printing logic is a part of the loop. Second, the error message printing logic should occur after the `readln` statement.

How do we print the error message? We can use a simple `writeln` statement like the following:

```
repeat
  write('please enter a number between 1 and 100:');
  readln(num);
  writeln('error: your number is less than 1 or greater than 100!')
until (1 <= num) and (num <= 100)
```

This is wrong because the error message is printed unconditionally. It should be printed only if the number is out of range. As a result, we need to a conditional statement to “guard” the printing of the error message when the number is in-range. The modified code is as follows:

```
repeat
  write('please enter a number between 1 and 100:');
  readln(num);
  if (num < 1) or (100 < num) then
    writeln('error: your number is less than 1 or greater than 100!')
until (1 <= num) and (num <= 100)
```

This last modification completes this code.

Chapter 8

Integer Representation

So far, we have used the `Integer` type for most of our programs. The mathematical definition of “integer” includes zero, all positive whole numbers and all negative whole numbers. In other words, there is such thing as the largest integer or the smallest integer.

However, in a Pascal program, a variable of `Integer` type has a limited range of number that it can represent. This is due to the internal representation of an `Integer` in a computer. This chapter discusses this representation and how it determines the range of values that a variable of `Integer` type can represent.

8.1 Binary Numbers

8.1.1 Revisit Base-10 (Decimal) Numbers

Let us first revisit how decimal numbers represent values. For example, let us consider the number 537. Let us rewrite this number as follows:

$$537 = 5 \times 100 + 3 \times 10 + 7 \times 1 \quad (8.1)$$

$$= 5 \times 10^2 + 3 \times 10^1 + 7 \times 10^0 \quad (8.2)$$

Of course, this is hardly surprising, except for the fact that $10^0 = 1$. It is important to note, however, that each *digit* of a base-10 number indicates the quantity of a particular power of 10. The right-most digit is also called the least significant digit, while the left-most digit is also called the most significant digit.

8.1.2 Breaking Down a Binary Number

Now, let us consider binary numbers. In order to make it clear which base we are using, binary numbers are marked by a subscript of 2. In other words, 100101_2 is a binary number.

Let us take a look at 100101_2 as a binary number. The digits have about the same meanings as digits in a decimal number. However, instead of indicating the quantity of a particular power of *ten*, each digit of a binary number indicates the quantity of a particular of *two*. We can rewrite our binary number as follows:

$$100101_2 = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \quad (8.3)$$

$$= 1 \times 32 + 0 \times 16 + 0 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 \quad (8.4)$$

$$= 32 + 4 + 1 \quad (8.5)$$

$$= 37 \quad (8.6)$$

Hey, that wasn't so difficult!

Each digit in a binary number has a special name, it is called a *bit*.

8.1.3 Converting from Decimal to Binary

The previous section shows a method to convert from binary to decimal. It is also possible to reverse the process. Given a decimal number, we can use division to convert it to a binary number.

Let us consider the number 53. This is how we come up with the bits:

$$53 \div 2 = 26r1 \quad (8.7)$$

$$26 \div 2 = 13r0 \quad (8.8)$$

$$13 \div 2 = 6r1 \quad (8.9)$$

$$6 \div 2 = 3r0 \quad (8.10)$$

$$3 \div 2 = 1r1 \quad (8.11)$$

$$1 \div 2 = 0r1 \quad (8.12)$$

$$0 \div 2 = 0r0 \quad (8.13)$$

$$0 \div 2 = 0r0 \quad (8.14)$$

$$\dots \quad (8.15)$$

If you read the remainders bottom-up, you have a binary number of 00110101_2 . The leading zeros (to the left) are useless. This means the process could have stopped when the quotient becomes one.

8.1.4 Addition

Addition in binary is about the same as addition in decimal. Let us review. In decimal, $4 + 9 = 11$, this can also be read as “3 with a carry of 1”. In binary, we only have two possible numbers per digit, 0 and 1.

It is obvious that $0_2 + 0_2 = 0_2$, afterall, zero is zero. We also know that $0_2 + 1_2 = 1_2$ and $1_2 + 0_2 = 1_2$, any value added to zero gets the original value. However, $1_2 + 1_2 = 10_2$. This is because 10_2 is the binary representation of two. Of course, you can also read this as “zero with a carry of one”.

Just like you use carries in decimal addition, you use carries in binary addition as well. It helps to make the carry from the previous (less significant) digit explicit and represent the partial sum explicitly. The following is an example of the addition of two binary numbers:

num1		1	0	0	1
num2	+	0	0	1	1
partial sum		1	0	1	0
carry from previous digit	+	0	1	1	0
answer		1	1	0	0

8.1.5 Multiplication

Multiplication in decimal requires the memorization of a rather big table. Binary multiplication, on the other hand, only has four rules:

- $0_2 \times 0_2 = 0_2$
- $0_2 \times 1_2 = 0_2$
- $1_2 \times 0_2 = 0_2$
- $1_2 \times 1_2 = 1_2$

Now that we have discussed single digit binary multiplication, let talk about multiple digit binary multiplication. Before we discuss binary multiplication, let's talk about decimal multiplication, first.

Let us consider the following:

$$215 \times 71 = 200 \times 71 + 10 \times 71 + 5 \times 71 \quad (8.16)$$

$$= 2 \times 71 \times 100 + 1 \times 71 \times 10 + 5 \times 71 \times 1 \quad (8.17)$$

$$= 2 \times 71 \times 10^2 + 1 \times 71 \times 10^1 + 5 \times 71 \times 10^0 \quad (8.18)$$

Of course, there is nothing really surprising in the previous derivation. It is, however, important to understand it anyway. Next, we present an example in binary multiplication.

$$\begin{aligned} 1011_2 \times 0101_2 &= 1000_2 \times 0101_2 + 000_2 \times 0101_2 + 10_2 \times 0101_2 + 1_2 \times 0101_2 \\ &= 0101000_2 + 0 + 01010_2 + 0101_2 \end{aligned} \quad (8.20)$$

The reason why $1000_2 \times 0101_2 = 0101000_2$ is exactly why $100 \times 13 = 1300$ in base 10.

When written in vertical form, the same multiplication can be done as follows:

num1					1	0	1	1
num2	×				0	1	0	1
leftmost			1	0	1	0	0	0
second rightmost	+				1	0	1	0
partial sum			1	1	0	0	1	0
rightmost	+					1	0	1
product			1	1	0	1	1	1

8.1.6 Negative Values and Negation

Up to this point, we have only discussed positive integers (represented in base-2). A computer needs to store and process negative values as well. We need a method to represent negative values. Before we discuss the actual method, let's think about what a negative number is.

The essence of “negativity” is the notion of negation. The symbol of negation is a minus sign. In other words, the negation of a value x is $-x$. x can be positive, negative or zero to start with. Let us take a look at the effects of negation.

The most common notation of negation is the $-$ symbol. In other words, in $x = -y$, x is the negation of y , and vice versa. If an operator fulfills the job of negation, then $x = -(-x)$ and $x + (-x) = 0$.

In our usual “people” math, negation is simple: we just prefix the value being negated by a $-$ symbol. In a computer, however, negation is a little more complicated. This is because the memory of a computer can only represent zeros and ones. There is no provision to represent the negation operator.

Fortunately, there is a quick-and-easy method to represent negative numbers and negate numbers. This method is called “two's complement”. We'll use $C_2(x)$ to represent the two's complement of x . If there is an operation called two's complement, there must be an operation called one's complement. We'll use $C_1(x)$ to represent the one's complement of x .

For one's complement and two's complement, it is important that we fix the number of digits being considered. Let us assume from now on that our binary numbers have exactly 8 bits.

One's complement is an operation to “flip” each binary digit of a number. For example, $C_1(01101101_2) = 10010010_2$. One's complement is meaningful only to binary representations.

Two's complement depends on one's complement. The definition of two's complement is as follows:

$$C_2(x) = C_1(x) + 1 \quad (8.21)$$

This means we can compute the two's complement of 01101101_2 as follows:

$$C_2(01101101_2) = C_1(01101101_2) + 1 \quad (8.22)$$

$$= 10010010_2 + 1 \quad (8.23)$$

$$= 10010011_2 \quad (8.24)$$

Now, let us test some of the properties of two's complement and see if it fits as a negation operator.

First, let's check if $x = C_2(C_2(x))$.

$$C_2(C_2(01101101_2)) = C_2(10010011_2) \quad (8.25)$$

$$= 01101100_2 + 1 \quad (8.26)$$

$$= 01101101_2 \quad (8.27)$$

Next, we check if $x + C_2(x) = 0$:

$$01101101_2 + C_2(01101101_2) = 01101101_2 + 10010011_2 \quad (8.28)$$

$$= (1)00000000_2 \quad (8.29)$$

Note that although the result is 100000000_2 , because we restrict ourselves to the least significant 8 digits, the answer is 00000000_2 .

Although none of this is any formal proof that two's complement can be used as negation for binary presentations, we did demonstrate that two's complement has some of the properties of negation.

Our current question is, then, which one is positive and which one is negative? In our example, does 01101101_2 represent a positive quantity, or does 10010011_2 represent this quantity?

The rule to determine the sign (negative or otherwise) is that if the most significant bit is a one, the number represents a negative value. Using this rule, 01101101_2 is non-negative because the most significant bit (leftmost) is zero. 10010011_2 , on the other hand, represents a negative value.

Our next question is, what is the value represented by 01101101_2 ? For *non-negative* values, we can directly apply the method as discussed in section 8.1.2. $01101101_2 = 64 + 32 + 8 + 4 + 1 = 109$. Because 10010011_2 represents the negation of this value, it represents -109 .

8.2 Range of Values

Now that we understand how integer values are represented in binary, we can return to our original question. What is the range of numbers as **Integer** variables?

In Borland Pascal and Free Pascal, an **Integer** has 16 bits and it is signed. This means we interpret the most significant bit to know if the number is negative or not.

The largest value that can be represented is 0111111111111111_2 , which is $2^{15} - 1$, or 32767. The smallest value that can be represented is 1000000000000000_2 . If you take the two's complement, it is still 1000000000000000_2 . This means the smallest value is -2^{15} or -32768.

In general, given n bits to represent a signed integer, the largest value that can be presented is $2^{n-1} - 1$, whereas the smallest value is -2^{n-1} .

8.3 Assignment (200 pts, due one week from assignment date)

8.3.1 What Does the Program Do?

In this assignment, you ask the user to input a non-negative integer that is less than or equal to 32767 (because it is the maximum value that can be represented by the `Integer` type). You only need to check if the number is negative. If the number is negative, ask the user to enter another number. Repeat this until the number is non-negative.

Once your program acquires a non-negative number, it should convert this number to quinary. Use a fixed overall width of 7 base-5 digits to print the number in quinary because you will need 7 quinary digits for 32767. The most significant digit must be on the left hand side, while the least significant digit must be on the right hand side.

8.3.2 The Approach

Refer to <http://www.drta.org/teaches/ARC/cisp365/samples/printbin.pas> for a program that prints in binary.

Instead of using the division method, you should use the compare-and-subtract method. Unlike binary numbers, which only have two choices for each digit (0 and 1), quinary numbers have 5 choices for each digit (0, 1, 2, 3 and 4). You need to use the compare-and-subtract method repeatedly to find out number of power of five in the number being converted. Do not use any division in this program except to divide the power of five by five.

I suggest you use $5^6 = 15625$ as the first power of five to try. After each step, divide the power of five by five. Remember to use the operator `div` instead of `/` for integer division.

8.3.3 Hints

You need two `Integer` variables, one for the power of five, one for the number being converted.

You need one loop to input the number and perform error checking, and another loop to print the number in quinary.

You need an *inner* loop in the main loop to find out the number of power of five in the number being converted. Then, you print the number of power of five using a `write` statement.

The *outer* loop to print the number in quinary should also include an assignment statement to divide the power of five for each iteration.

8.3.4 Trinary Conversion Example

Let consider the number 154 (in base 10), and try to convert this to base 3 (instead of base 5). The largest power of 3 to worry about is 81.

8.3. ASSIGNMENT (200 PTS, DUE ONE WEEK FROM ASSIGNMENT DATE)59

How many 81s are there in 154? There is one. The first (most significant) digit to print is 1. After this, we subtract 81 from 154, so the remaining value to convert is $154 - 81 = 73$. We also need to divide the power of 3 by 3, so 81 becomes $81 \div 3 = 27$.

How many 27s are there in 73? Since $73 \geq 27$, there is at least one. The remaining value, $73 - 27 = 46$, has another 27. There are now two 27s in 73. The remaining value, $73 - 27 - 27 = 19 < 27$, so we can conclude there are only two 27s in 73. We print 2. The remaining value to convert is now 19, and we divide the power of 3 so that $27 \div 3 = 9$.

How many 9s are there in 19? Through the same boring process as described in the previous paragraph, there are two. The remaining value to convert is now 1. The power of 3 is divided again so $9 \div 3 = 3$.

How many 3s are there in 1? Since $1 < 3$, there is none! We print 0. We divide the power of 3, $3 \div 3 = 1$.

How many 1s are there in 1? Since $1 \geq 1$, we know there is at least one. The remaining value is 0, and we know $0 < 1$. We conclude there is one 1 in 1. Print 1. We divide the power of 3 again, $1 \div 3 = 0$ (because we only keep the integer part).

As soon as the power of 3 becomes 0, we stop. The number printed is 12201 (base 3).

8.3.5 How to Turn it In?

Use "CIS31 Project2 by *your name*" as the subject of your email. Send only the .pas file. You have one week from the date of assignment to complete this assignment.

Chapter 9

Real Numbers

`Integer` is a very useful type. Variables of this type can be used for counting and represent positive as well as negative quantities. However, in real life, many numbers are not integers. For example, the value of π is an irrational number. Three quarter of an inch is 0.75 inch, which is not an integer, either.

Pascal provides another type, in addition to `Integer`, to represent numbers that are not integers. This type is predefined (like `Integer`) and has a name of `Real`. To declare the `x` is a real number, for example, you can do so with the following code:

```
var
  x : Real;
```

9.1 What is Real?

In mathematics, “real numbers” is the union of rational numbers and irrational numbers. Examples of real numbers include π , 2, $-\frac{12}{23}$, and etc. However, the type name `Real` in Pascal is misleading because a variable of `Real` type may not be able to represent a fraction exactly, and it is guaranteed not to be able to represent an irrational number exactly.

It is better to see the `Real` type as a “floating point” number type. To be more specific, it is a type of number represented by a binary point number with a binary exponent.

9.1.1 The Binary Point

In decimal numbers, the “decimal point” separates places in a number into the whole number group and fractional group. For example in the number 12.34, the decimal point indicates that “2” is the least significant digit for the whole number group, while “3” is the most significant digit for the fractional group.

We can extend our interpretation of numbers to include the decimal point:

$$12.34 = 1 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} \quad (9.1)$$

$$= 10 + 2 + 0.3 + 0.04 \quad (9.2)$$

For *binary* numbers, a period is also used to separate the whole number group and the fractional group. For example, the binary number 11.101_2 really means the following:

$$11.101_2 = 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \quad (9.3)$$

$$= 2 + 1 + \frac{1}{2} + 0 + \frac{1}{8} \quad (9.4)$$

$$= 3\frac{5}{8} \quad (9.5)$$

9.1.2 Exponent

We can use the concept of exponent to allow the specification of very large and small magnitudes. Consider the number 30,000,000,000 (our state deficit). Instead of writing all these zeros, I can also write $30000000000 = 30 \times 10^9$. In this alternative representation, 9 is the exponent (with a base of 10).

Similarly, we can use a similar concept to handle large and small magnitudes for binary numbers. $11.101_2 = 1.1101_2 \times 2^1$. The only difference between a binary exponent and a decimal exponent is that we use 2 as a base for binary exponent, whereas we use 10 as a base for decimal exponent.

For binary point numbers except 0, we can always adjust the exponent until the number becomes $1 \cdot \dots \times 2^n$. n can be positive, negative or zero. This is why this type of number is called “floating point” because the binary point “floats” as we adjust the exponent.

9.1.3 Actual Implementation

The actual implementation of the `Real` type in Pascal is as follows. A `Real` number represents the following value:

$$sign = 0 \Rightarrow 1frac \times 2^{exp-127} \quad sign = 1 \Rightarrow -1frac \times 2^{exp-127} \quad (9.6)$$

sign is a single bit in a `Real` variable. *frac* is a 23-bit quantity. Note that *frac* only represents the *fractional* part of the mantissa. The only digit of the whole number part is implied to be a one. *exp* is similar to the exponent, except it is biased by 128. *exp* has 8 bits, but the values 0000000_2 and 1111111_2 is reserved.

With this representation, there is no way to represent 0 itself. The standard says when all the bits of a `Real` value are 0, the value it represents is also zero.

In this class, we don’t need to know the *exact* details of the IEEE 32-bit floating point numbers. It is, however, important to remember that the `Real` type is not exactly suitable for storing all real numbers in mathematics.

9.2 Real Vs. Integer

With *both* `Real` and `Integer` types, a programmer needs to decide which type to use in programs. From an abstract perspective real numbers include integers, so it is always better to use the type `Real`. However, from the practical standpoint, the `Real` type is not always suitable.

9.2.1 Efficiency

Most processors can handle both integers and real numbers. However, integer mathematics is still considerably more efficient than real number mathematics. As a result, if you want a program to execute faster, you should avoid using real numbers.

9.2.2 Range of Magnitude

If your program needs to handle numbers of a very wide magnitude (such as for scientific or astronomical applications), you have no choice but to use real numbers. On the other hand, if you know the range of number is within that of integers, you should consider using integers.

9.2.3 Fractions

It appears that real numbers can handle fractions but decimal numbers cannot. In reality, even real numbers cannot handle fractions precisely. For example, there is no finite binary representation of $\frac{1}{3}$. When you execute the statement “`x := 1/3`”, it stores an approximation of the actual quantity of one third to variable `x`.

If an application needs to handle fractions, there are other presentations that are either more efficient or more precise than `Real` numbers. We will talk about some of these other methods later.

9.3 Operators for Real Versus Integer

Both `Real` and `Integer` are number types, they share the same notations for addition, subtraction and multiplication. However, they do not share the notation of division.

For `Integers`, if you want to know the quotient of a division, you use the `div` operator. For example, in `q := dn div dr`, variable `q` receives the quotient of `dn` divided by `dr`. For `Integer` division, you can also receive the remainder of a division using the `mod` (for modulus) operator. For example in the statement `r := dn mod dr`, variable `r` receives the remainder of `dn` divided by `dr`.

If you have a `Real` variable `x` to receive the result of a division, you use the `/` operator. For example, in `x := 1/3`, `x` receives a value of approximately 0.33333333.

This practice of using a different operator for real number division from that for integer division is not a standard. In C, C++ and Java, the same operator, `/`, is used for both. In Visual Basic, however, `/` is used for real number division, and `\` is used for integer division.

9.4 Assignment

In this (new) assignment, write a program that plays a simple game with the end user. Your program has to choose an integer between zero and 50 (inclusively), and let the end user guess which number the program had selected.

If and only if the user enters a value of -1, the program should exit.

Allow the user to guess a number up to 6 times. If the user cannot guess the number in 6 times, print a message and start the game all over again (use another random number).

9.4.1 Getting a Random Number

In this program, you need to come up with some “random” numbers. It is actually extremely difficult to come up with true random numbers in computers. You can, however, use the `random` function in Borland Pascal.

The function `random` returns a different `real` number every time you call it. The values returned is somewhat random. The number returned by `random` is guaranteed to be greater than or equal to 0 and less than 1.

In order to come up with an integer between 0 and 50 inclusively, you need to multiply the random number by 51. You can assign a `real` value to an integer variable, Pascal automatically takes the floor of the value (just the integral part).

9.4.2 Hint

You can write this program as one big main program, or you can write this program in smaller bits (as in procedures). I recommend the use of subroutines (procedures) to help you break the program down to more manageable pieces.

You will need the following in this program:

- An integer variable to store the number being guessed.
- An integer variable to store the number of guesses already used.
- An integer variable to store whatever guess the user types.
- An overall loop to keep the game going until the user indicates he/she want to quit.
- A loop to make sure the user only inputs valid values.
- A loop to guess one number until all guesses are used up, the user wants to quit, or the correct number is entered.

- use `51*random` in Borland Pascal 7 to come up with a real number greater than or equal to zero and less than 51.
- use `trunc(x)` to get the integer part of a real number `x`.

If you want to practice using **procedures**, I'd recommend the following **procedures**:

- A procedure to read a number in the valid range in. Note that the player can enter -1 at any time to give up.
- A procedure to guess one number. This procedure should get a “random” number and let the user guess until all guesses are used up, the user wants to quit or the correct number is entered. Obviously, this procedure needs to call the previous one.
- The main program should call the previous procedure repeatedly until the user wants to quit.

You don't have to use parameters in this program, but it certainly will not hurt if you try. If you are unsure about parameters, write the program without parameters first, get it to work, save it in a different name, *then* rewrite it using parameters.

9.4.3 How to submit?

Use “CIS31 Project3 by *your name*” as the subject of your email. Send only the `.pas` file. You have two weeks from the date of assignment to complete this assignment.

Part IV

Subroutines

Chapter 10

Simple Subroutines

Let us consider the previous homework assignment that prints an integer in binary format. This is a fairly useful piece of code, a program may want to print several numbers in binary.

With all we know at this point, the only way to print more than one numbers in binary is to duplicate the code several times. While this approach works, we will find out that it is not the best method.

10.1 Example

Let us consider the following program. This program is to read two numbers as the first and the last number of a range. The program then prints numbers from the first number to the last number inclusively. In addition, the program must ensure the following rules:

- The first number must be inclusively between 1 and 100.
- The second number must be inclusively between the first number and 100.

The program needs to ask the user to enter a number again if the number is out of range.

While this program is a little tedious, it is not out of reach with what we have discussed up to this point. The program may look like the following:

```
var
  first, last, i : Integer;
begin
  repeat
    writeln('please enter a number between 1 and 100');
    readln(first);
    if (first < 1) or (first > 100) then
      writeln('your number is out of range')
```

```

until (1 <= first) and (first < 100);
repeat
  writeln('please enter a number between ',first,' and 100');
  readln(last);
  if (last < first) or (last > 100) then
    writeln('your number is out of range')
until (first <= last) and (last < 100);
while (first <= last) do
  begin
    writeln(first);
    first := first + 1
  end
end.

```

Do you notice the similarity between the first and the second `repeat-until` loop?

10.2 Why Use Subroutines

Given our example, we can now discuss why subroutines are useful. Well, we will discuss what a pain it is without subroutines.

There is a bug in the previous program. I should have used `<=` instead of `<` for the upper bound check of the exit conditions of the `repeat-until` loops. Of course, I can easily fix this error, but I have to fix this error *twice* because the second loop is a modified duplicate of the first. In other words, although there is only one *conceptual* mistake, I need up to update all of the copies of the original code.

What if I forget to update one of the duplicate(s)? The compiler cannot tell! My program will be partially fixed and remain partially wrong.

It is much better if I only need to fix the problem once.

Now, the other disadvantage of *not* using subroutines is the tedium of copying and pasting. Reading a number that is bound checked is a very common algorithm. Each time I need to read a number and ensure the bounds are checked, I need to copy and cut the original `repeat-until` loop. This is tedious, and it makes my program long.

Long programs take up more space (in memory and in the harddisk). However, the worst part is that lengthy programs are also more difficult to read and debug. For beginners, each self-contained block of code should be between 7 to 15 lines, excluding the `begin` and `end` for block statements.

10.3 Introduction to Subroutine

A subroutine is basically a block of code that can be “invoked” from anywhere in the program (including another subroutine). A subroutine can be invoked many times in a program.

Consider the following program with a simple subroutine `sub1`:

```
{ first, define the subroutine }
procedure sub1;
begin
    writeln('in sub1')
end;

{ this is the main program }
begin
    sub1;
    sub1
end.
```

This program differs from all of our previous programs because it contains the definition of subroutine `sub1`. All subroutines must be declared or defined before they can be invoked. Note that we never start execution in the definition of a subroutine. All Pascal programs start execution in the main program (with its own independent `begin-end` block statement).

In the main program, `sub1` is invoked twice. Each invocation is a statement. In this simple Pascal program, to invoke `sub1` simply requires its name.

This output of this program is the output of “`in sub1`” twice. You can add more invocations of `sub1` to print the boring message more times.

What have we learned so far?

In order to define a subroutine, we need the following template:

```
procedure subname ;
begin
    ...
end;
```

In order to invoke a subroutine, the statement consists of the name of the subroutine. Each invocation passes control to the named subroutine, let the subroutine run its code, and return control to the statement after the invocation.

10.4 Global Variables and Subroutines

A subroutine can “see” all global variables defined before its own definition. This *can* be used as a mechanism to pass information to a subroutine. For example, if we want to use a subroutine to print a number, we could write a program as follows:

```
var
    numToPrint: Integer;
    x, y : integer;
```

```

procedure printNum;
  begin
    writeln(numToPrint)
  end;

begin
  readln(x,y);
  numToPrint := x + y;
  printNum;
  numToPrint := x - y;
  printNum
end.

```

This program clumsily utilize subroutine `printNum` to print the sum and difference of `x` and `y`. Note how the value to be printed must be assigned to the global variable `numToPrint` first?

In the next chapter, we'll discuss better methods to pass information into and out of subroutines.

10.5 Our Example

For our example, we can *generalize* both `repeat-until` loops into one common algorithm and utilize this in the program. Our program becomes the following:

```

var
  readMax, readMin, readNum : integer; { for subroutine }
  first, last : integer; { for the main program }

procedure readBoundChecked;
  begin
    repeat
      writeln('please enter a number between ',readMin,' and ',readMax);
      readln(readNum);
      if (readNum < readMin) or (readMax < readNum) then
        writeln('your number is out of range')
      until (readMin <= readNum) and (readNum < readMax)
    end;

begin
  readMin := 1;
  readMax := 100;
  readBoundChecked;
  first := readNum;
  readMin := first;
  readMax := 100;
  readBoundChecked;

```

```

last := readNum;
while (first <= last) do
  begin
    writeln(first);
    first := first + 1
  end
end.

```

The subroutine `readBoundChecked` utilizes variables `readMax` and `readMin` to set up the boundary. Then it attempts to read a number into `readNum`. Its logic is very similar to our original `repeat-until` loops, except it is now more general and uses variables instead of constant as bounds.

The main program no longer requires its own `repeat-until` loops. Instead, it invokes `readBoundChecked` to read a bound-checked number. Note how the program needs to set up `readMax` and `readMin` before invoking `readBoundChecked`. The program also needs to utilize `readNum` by copying its value to `first` and `last` after each invocation.

Even though this program is still tedious due to the setting up of `readMin` and `readMax` as well as the explicit utilization of `readNum`, it is now cleaner than the original code. Furthermore, Even though I made the same mistake in the exit condition of the `repeat-until` loop (using `readNum < readMax` instead of `readNum <= readMax`), I only need to fix the code once!

10.6 What's Next?

The use of global variables to pass information from an invoker to the invoked subroutine is *possible* in almost all programming languages. However, this approach is considered dangerous and unstructured. In the next chapter, we will discuss the concept of *parameters*. Parameters are the recommended method to let the invoker exchange information with the invoked subroutine.

10.7 Assignment

200 points, due one week from assignment date.

In this assignment, you need to modify the program already available at <http://www.drta.org/teaches/ARC/cisp365/samples/factor.pas>. Download this program and make sure you understand it.

The output of this program is rather simple. For example, the following is the output when 100 is the input:

```
100=2*2*5*5
```

10.7.1 The New Program

You are to modify the program so it uses the exponent notation whenever it makes sense. To be more specific:

- When a prime factor has a power more than one, use the exponent notation
- When a prime factor only has a power of one, use the normal notation, *do not display as a number to the first power!*

This means for 100, we want the new output to look like the following:

$$100=2^2*5^2$$

The caret symbol (^) is used to indicate “to the power of”.
For 30, however, we want the output to be just like before:

$$30=2*3*5$$

Your program must work for all numbers between 1 and 32767.

10.7.2 How to proceed?

First, download my `factor.pas` and study it. Make sure you understand the logic, especially the main program. Add additional comments as necessary. Then, save this `factor.pas` and leave it alone.

Make a copy and call it another name. This way, if you break the program, at least you can fall back on the original program.

The loop of the main program only finds factors. It does not, however, keep track of the exponent of a particular factor. There are several ways to solve our problem.

Method 1

The first method is to leave the logic intact, but use two additional variables and a conditional statement to decide how to print. Use one variable to keep track of the *previous* factor and another variable to keep track of the exponent. When the previous factor is different from the current factor, you print the exponent of the previous one and reset the exponent for the current factor.

The method needs a special case for the first factor. For the first factor, there are no previous factor to speak of. As a result, the logic to print exponent must not be active for the first factor. This can be done by initializing the previous factor to zero, then use a conditional statement to block the interpretation of the “previous” factor and its exponent when it is zero. This method also needs to handle the special case of the last factor. How are we going to print the exponent (if necessary) of the last factor?

The nice part about this method is that there is no new loop. However, you do need an additional conditional statement, and the printing of a factor and its exponent (if necessary) are off by one iteration. The last point makes the first and last factors special cases.

Method 2

The second method is to introduce an additional inner loop in the current loop. Once a factor is found, this inner loop continues to test if there are more of this factor in the number. Before this inner loop, you need to initialize the exponent counter, then each iteration increments the exponent counter by one.

Ask yourself these questions about this inner loop:

- What does each iteration do?
- What condition(s) lead(s) you out of the loop?
- What needs to be initialized before the loop?
- Do you need at least one iteration?

The advantage of this approach is that you don't have special cases for the first and last factors. As a result, you don't need special conditional statements that handle the first and last factors. You still need an exponent counter, but you don't need to track the previous factor as in the previous method. The extra loop is relatively easy.

10.7.3 How to turn it in

Send this one (only the `.pas` file) to tauyeung@drtak.org. The subject line should be "CIS31 Project3 *by ...*".

Chapter 11

Parameter Passing for Subroutines

In the previous chapter, we discussed subroutines in general. We discovered that we can use subroutines to share and reuse code in a program. We also discovered a limitation: we have to rely on global variables to let the code invoking the subroutine communicate with the invoked subroutine.

This chapter presents the concept of “parameters”. Parameters is a powerful concept that allows a clean, elegant and flexible linkage between a subroutine and the code that invokes it.

11.1 What is a parameter?

Imagine the *definition* of a subroutine as an office/cubicle with a worker who specializes in a particular task. A parameter is essentially an in-box and an out-box for the office/cubicle. Each parameter has a unique name so the worker in the office/cubicle can refer to it.

There are two types of parameters. One type of parameter is *a copy of* some original item, while the other type is *a reference to* an original item.

The first type of parameters is called “passed by value”. When a parameter is passed by value, the invoker makes a copy of some value, then give the subroutine the copy to process. The main feature of this type of parameters is that the subroutine can modify the copy as much as possible during processing, but it can never modify the original item. This type of parameter is useful mostly for information that is given to a subroutine.

The second type of parameters is called “passed by reference”. When a parameter is passed by reference, the invoker gives the subroutine a “reference” to an object (i.e., a variable). A “reference” is essentially some instruction to access the original item. When a parameter is passed by reference, the subroutine accesses the original item directly. As a result, any changes the subroutine makes to the parameter is reflected directly at the original item.

This type of parameter is useful for passing information that a subroutine needs to modify and return to the invoker.

11.2 Parameter Definition Syntax

When a subroutine specifies parameters, such parameters are usually called “formal parameters”. The definition of a subroutine specifies formal parameters in parentheses after the name of the subroutine. Each parameter looks somewhat like a variable declaration. Formal parameters are separated by semicolons.

For example, the following is the definition of a subroutine that prints a number:

```
procedure printNum(number : integer);
begin
  writeln(number)
end;
```

Formal parameters with a `var` reserved word in front of the name are passed by reference. Otherwise, a formal parameter is passed by value. `number` in the previous example is passed by value.

The following subroutine has a formal parameter that is passed by reference:

```
procedure increment(var number : integer);
begin
  number := number + 1
end;
```

You can mix and match parameters passed by value and reference. The following code defines a subroutine that computes the difference of two numbers given to it:

```
procedure difference(num1, num2 :
  integer; var diff : integer);
begin
  diff := num1 - num2
end;
```

11.3 First Contact with Scope

You *may* define a parameter with the same name as a global variable. For example, the following program is syntactically correct.

```
var
  x, y, z : integer;
procedure sub1(x : integer; var y : integer; z : integer);
begin
```

```

    y := x + z
end;

begin
  x := 2;
  y := 4;
  z := 10;
  sub1(x+1, z, y-1);
  writeln(x, ', ', y, ', ', z)
end.

```

From the perspective of subroutine `sub1`, there is only one `x`, one `y` and one `z`. Because the formal parameters are defined “locally”, they “hide” the global variables of the same name. In other words, inside a subroutine, the Pascal compiler resolves definitions of symbols using more local definitions first.

We will have more discussions of scoping when we discuss local variables.

11.4 Invoking Subroutines with Parameters

Depending on whether a formal parameter is passed by value or passed by reference, there are different limitations applied to the invoking code.

When a statement invokes a subroutine with parameters, Pascal uses position to match the provided parameters with the formal parameters. Let us consider the following program.

```

var z : integer;
procedure difference(a, b : integer; var diff : integer);
begin
  diff := a - b
end;

begin
  z := 12;
  difference(24, z+1, z)
end.

```

When `difference` is invoked, the compiler needs to match the provided parameters with the formal parameters in the definition. Because Pascal uses position for matching, we end up matching the value `24` with formal parameter `a`, the value `z+1` with formal parameter `b` and the variable `z` with formal parameter `diff`.

This means that during execution, formal parameter `a` is a copy of the value `24`. Similarly, formal parameter `b` is a snapshot of the expression `z+1`. Because `z` has a value of `12` when `difference` is invoked, formal parameter `b` is a copy of the value `13`. Formal parameter `diff` is a *reference* to variable `z`.

Ah, what does this mean (that `diff` is a reference to variable `z`)? This means that whenever we refer to formal parameter `diff` in subroutine `difference`, we are actually referring to global variable `z`. A formal parameter passed by reference does not have its own value, it is a reference to *something else* that has a value.

Note that Pascal *requires* that you pass *something with storage* to formal parameters passed by reference. In other words, you cannot pass the constant `243` to formal parameter `diff` in the previous example. This makes sense because we cannot use `243` on the left hand side of an assignment statement.

11.5 Nested Invocation

The allowance of nested invocation is what makes structured programming languages (like Pascal, C and others) powerful. You can invoke a subroutine from a subroutine. There is no theoretical limitation as to how many levels of invocations a program can nest.

Parameter passing in nested invocation requires some additional explanation. This is especially the case with passing by reference.

Let us consider the following program:

```
var gx, gy : integer;

procedure sub1(x1 : integer; var y1 : integer);
begin
  y1 := y1 + x1
end;

procedure sub2(var x2 : integer; var y2 : integer);
begin
  x2 := x2 * 2;
  sub1(x2, y2)
end;

begin
  gx := 6;
  gy := 11;
  sub2(gx, gy)
end.
```

Let's see what happens when this program executes:

- `gx` and `gy` are initialized to 6 and 11, respectively.
- `sub2` is invoked, `x2` becomes a reference to `gx`, `y2` becomes a reference to `gy`.

- `x2` is doubled. Since `x2` is a reference to `gx`, the value of `gx` is doubled from 6 to 12.
- `sub1` is invoked, the value of `x2` is passed by value of `x1` of `sub1`, which means `x1` has a copy of the value of `x2`. What is `x2`? `x2` is a reference to `gx`. As a result, formal parameter `x1` of `sub1` has a value of 12. `y2` is passed by reference to `y1`. `y2` is a reference to `gy`. This means `y1` is also a reference to `gy`.
- we add `x1` to `y1`. `x1` has a value of 12. `y1` is a reference to `gy`, which has a value of 11. The sum is 23, and it is stored to `y1`. Since `y1` is a reference to `gy`, 23 is actually store to `gy`!
- we now return from `sub1`.
- we now return from `sub2`.
- in the main program `gx` has a value of 12, and `gy` has a value of 23.

What if we change formal parameter `y2` from passed-by-reference to passed-by-value? It becomes a snapshot of the value of `gy`, but changes to `y2` is no longer reflected in `gy`. When we pass `y2` by reference to `y1` of `sub1`, we make `y1` a reference to `y2` *instead of* `gy`. This means the outcome of the program is that `gx` still has a value of 12, but `gy` has a value of 11 instead of 23 because changes to `y1` in `sub1` changes `y2`, which is only a copy of `gy`, leaving `gy` unmodified.

Chapter 12

Functions

Procedures are handy and flexible, but they are also clumsy to use in some cases. This is because a procedure has only one method of returning information to the caller: passing parameters by reference. While this method is flexible (because a procedure can use multiple parameters passed by reference to return multiple pieces of information), it is clumsy when there is only one result to return.

Consider the procedure that computes the difference two numbers:

```
procedure difference(num1, num2 : integer; var diff : integer);
begin
    diff := num1 - num2
end;
```

This works, but it also requires the caller to pass a variable to formal parameter `diff`. For example, just to print the difference of two numbers, we need the following code:

```
difference(myvar1, myvar2, temp);
writeln('the difference of ',myvar1,' and ',myvar2,' is ',temp);
```

Does it work? Yes. But this is clumsy.

12.1 What is a Function? How is It Different from a Procedure?

A function can *return* a value, whereas a procedure cannot.

What does that mean? Let's try to implement `difference` as a function:

```
function difference(num1, num2 : integer) : integer;
begin
    difference := num1 - num2
end;
```

The definition is altered a little bit from the `procedure` version. First, the reserved word `function` is used instead of `procedure`. `diff` is no longer a parameter. There is the `: integer` notation following the declaration. Instead of assigning to `diff`, we assign to the name of the function, which is `difference`.

The `: integer` notation means the function “returns a value of `integer` type”. In order to specify what to return, we assign to the name of the function.

When we need to compute and print the difference of two values, we can now use `difference` as a function:

```
temp := difference(myvar1, myvar2);
writeln('the difference of ',myvar1,' and ',myvar2,' is ',temp);
```

This first attempt is correct. Because a function returns a result, the result itself can be treated as a value. As a result, we can simply assign the value (returned by a function) to a variable.

But this means we can also directly utilize the returned value of a function in the `writeln` statement:

```
writeln('the difference of ',myvar1,' and ',myvar2,' is ',
        difference(myvar1, myvar2));
```

This is why functions are more convenient than procedures! We can utilize the returned value of a function directly without having to use a variable to hold the value!

12.2 More about the Syntax

The general form of the definition of a function is as follows:

```
function func_name [ (params) ] : ret_type ;
begin
...
end
```

In this notation, [and] encloses a section that is optional. This means the formal parameters are optional.

It is important that the programmer remember to assign a returned value to the name of a function. The Pascal compiler does not check and make sure a function always returns a value! In other words, the following function is perfectly correct:

```
function useless : integer;
begin
  writeln('I am useless.')
end;
```

What do you think the following code prints?

```
writeln(useless);
```

Because the function definition of `useless` does not specify any particular value to return (as an assignment to `useless`), its actual returned value is undetermined. This means there *is* a value returned every time `useless` is called, it's just that we can not determine what that value is. This is *not* to say that `useless` returns a *random* value because the value returned by `useless` does depend on the useage of `useless` and other factors in the program.

12.3 Homework Assignments (100 points, due one week from assignment date)

Use the program at <http://www.drta.org/teaches/ARC/cisp365/samples/factor.pas> as a basis.

This program works (as you know already). However, it is somewhat of a taboo to refer to global variables in a subroutine (be it a `function` or a `procedure`). As a result, procedure `find_factor_f_in_n` is not as ideal as it could have been.

Your assignment is to fix this problem. You are to turn `find_factor_f_in_n` into a function that returns the factor. As a function, it should also take one (and only one) parameter.

You also need to change the main program so it calls a function instead of a procedure. In addition, you need to *do something* about the value returned by the function.

This is a summary of what you need to do (as a suggestion):

- turn `find_factor_f_of_n` from a procedure to a function
- add a local variable to the function
- add a parameter to the function
- change the main program to call the function
- do something about the value returned by the function

These are the *absolute* requirements of your program:

- the main program must call `find_factor_f_of_n` to find a prime factor of `n`
- the subroutine `find_factor_f_of_n` must be a function
- the subroutine `find_factor_f_of_n` must use one parameter passed by value
- the subroutine `find_factor_f_of_n` must not access *any* global variable directly

- the behavior of the program must remain the same as the original program

Submit the assignment to tauyeung@drtak.org with a subject line of “CIS31 Project4 *by My Name*”.

Chapter 13

Local Variables

So far, we have used global variables. The “global” in global variables implies there must be “local” variables. Local variables are important concepts in programming because they make it possible to define variables of the same name for different subroutines.

13.1 The Concept

The concept of local variables is rather simple: a local variable is local to a particular subroutine. There are actually two aspects of being “local”.

The first aspect of “local” is the scope. Recall that the scope of a name is defined by which part of the program can see it. A local variable has a local scope, which means it can only be seen (accessed) from the subroutine that defines it.

The second aspect of “local” is lifespan. *In Pascal*, all local variables start to exist when the containing subroutine is invoked, and they all cease to exist when the subroutine returns. In other languages like C and C++, you can specify a persistent lifespan for local variables.

13.2 The Syntax

You can define local variables for **procedures** and **functions**. To do this, you need to insert a **var** block between the header of a subroutine (the line containing either **procedure** or **function**) and its body (the **begin end** block).

For example, the following procedure has a local variable **i** to keep track of the number being printed.

```
procedure printnums(first, last : integer);
  var
    i : integer;
  begin
```

```

i := first;
while (i <= last) do
begin
  writeln(i);
  i := i + 1
end
end;

```

You can define any number of local variables after the `var` reserved word and before the `begin` reserved word.

The nice part of local variables is that different subroutines can use the same name for their local variables. Consider the following program:

```

function readANum : integer;
var
  i : integer;
begin
  readln(i);
  readANum := i
end;

procedure printnums(first, last : integer);
var
  i : integer;
begin
  i := first;
  while (i <= last) do
  begin
    writeln(i);
    i := i + 1
  end
end;

begin
  printnums(readANum, readANum)
end.

```

The local variable `i` defined and used in `readANum` has *nothing* to do with the local variable `i` defined in used in `printnums`! This is rather handy because now a programmer can reuse “popular” variable names like `i`, `j`, `k` and etc.

13.3 Local Variables and Parameters Passed by Value

Local variables are very similar to parameters passed by value. Both have storage can change their values as the program executes. The only differences

13.4. HOMEWORK ASSIGNMENT, 100 POINTS (DUE IN ONE WEEK)89

between a local variable and a parameter passed by value are the syntax of their definition and that a parameter passed by value has an initial value determined by the caller of the subroutine. A local variable does *not* have any initial value. As a result, it is important for a programmer to initialize a local variable before using its value.

Just like parameters, local variables are “destroyed” when a subroutine completes. This means the value of a local variable is *not preserved* across invocations.

13.4 Homework assignment, 100 points (due in one week)

The program at <http://www.drta.org/teaches/ARC/cisp365/samples/factor.pas> works, but it is inefficient. To be more specific, the subroutine `find_factor_f_in_n` searches for a prime factor from 2 every time.

Start with downloading the program from the link listed above. Similar to the previous homework assignment, you are to add parameters and turn `find_factor_f_in_n` into a function, however, there is a twist in this homework assignment.

Make the program more efficient by letting `find_factor_f_in_n` modify a variable, storing the prime factor, that belongs to the main program. To do this, you’ll need to perform the following changes to the program:

- turn `find_factor_f_in_n` into a function
- specify the return value of the function
- add any necessary parameters to the function
- add any necessary local variables to the function
- change the main program so it calls the function (instead of a procedure)
- change the main program to pass the appropriate parameters
- remove one of the initializations from `find_factor_f_in_n`
- add the initialization to the main program

These are the absolute requirements of the homework assignment. If your program does not conform to *all* of these requirements, your assignment will receive 0 point.

- *no* global variable can be accessed from `find_factor_f_of_n`
- `find_factor_f_of_n` must be a function, not a procedure
- `find_factor_f_of_n` cannot print, it cannot contain `write` or `writeln`

- once the main program has found a prime factor p , all subsequent calls to `find_factor_f_of_n` cannot reexamine all prime factors that are less than p .
- only pass by reference when it is necessary, otherwise, pass by value (this means that if you pass by reference when you could have passed by value, points will be deducted!)
- the overall behavior (as far as input and output is concerned) must remain unchanged from the original program

Part V

Congregates

Chapter 14

Arrays

In previous chapters, we have use “atomic” types. A value of atomic type cannot be broken into smaller components. This chapter discusses the first of two ways to assemble congregate types.

14.1 What is an Array?

An array is a collection of elements of the same type in which each element is designated by an index. Indices of elements in an array must be consecutive integer values.

Okay, that’s kind of abstract. Imagine you have a box of coffee mugs, each coffee mug is tagged with a unique integer value. Each coffee mug is *like* a variable because it can contain changing contents as a program executes. However, specifying a mug in a box involves the following:

- First, specify the name of the box.
- Second, specify the number of the mug in the named box.

There is some restrictions. The integers tagging each mug in the box must be consecutive. In other words, you cannot use just any random integers. The “type” of each mug in a box must be the same. In other words, you cannot mix coffee mugs along with beer jugs in the same box.

To get to a particular mug, we can use the phrase “mug number 23 in box cheapPlasticMugs”.

14.2 Pascal Syntax

To define an array in Pascal, we can use the following code:

```
var
  aBunchOfIntegers : array [0..23] of integer;
```

In this particular definition, we define an array called “aBunchOfIntegers”. The first element has an index of 0, while the last element has an index of 23. Each element in this array has the type of `integer`.

How many elements do we have in the array? Since we use 0 for the first index and 23 as the last, we have 24 elements in the array.

Most people count from 1, it seems to make more sense with the following definition with 24 `integers`:

```
var
  aBunchOfIntegers : array [1..24] of integer;
```

In fact, in Pascal, *you* can specify any integer as the index of the first element in an array, including negative integers! For example, you *can* define the following array which also has 24 `integers`:

```
var
  aBunchOfIntegers : array [-12..11] of integer;
```

However, keep in mind that most other programming languages do not allow an arbitrary first index. Instead, most programming languages *require* that the first index be 0 (as in zero).

After an array is defined, a program can refer to the elements in it. The indexing operator is represented by square brackets. For example the following statement initializes the element at index 0 of array `numbers` to 100.

```
numbers[0] := 100
```

Note that you only need to supply an integer value in the square brackets. This means it is perfectly okay (syntactically) to have the following statement, assuming `i` is an integer variable:

```
numbers[i] := 100
```

14.3 Common Array Techniques

14.3.1 Going Through an Array

One of the strengths of arrays (as compared to individual variables) is that it is easy to access individual elements of an array. If we have variables `num01`, `num02` and etc. up to `num50` (a total of 50 individual variables), the initialization of all 50 variables will be tedious because it requires 50 assignment statements.

However, if we use an array of 50 `integers`, however, the initialization of all 50 `integers` is easy. Observe the following code:

```
var
  nums : array [0..49] of integer;
  i : integer;
```

```

begin
  i := 0;
  while (i < 50) do
    begin
      nums[i] := 0;
      i := i + 1
    end
  end.

```

This program with only three lines of code initializes all 50 `integers` in array `nums` to zero. In fact, this code can initialize elements in `nums` to 0, regardless of the number of items in the array. All we need to do is to change the bound.

If we want this program to be easy to maintain, with respect to the number of items in the array, we can define a constant. The following program utilizes a constant for the size of the array:

```

const
  nums_size = 50;
var
  nums : array [0..nums_size-1] of integer;
  i : integer;
begin
  i := 0;
  while (i < nums_size) do
    begin
      nums[i] := 0;
      i := i + 1
    end
  end.

```

Note how the definition of `nums_size` uses an equal sign (and not a colon). This is because this is a definition of a constant, and we are equating the symbolic name `nums_size` with the value 50.

By simply changing the definition of `nums_size`, we can change the size of the array *without* having to alter any other line in the program.

14.4 Homework Assignment (200 points)

In this homework assignment, write a program that finds the first 50 prime numbers. You can use my program at <http://www.drta.org/teaches/ARC/cisp365/samples/factor.pas> as a starting point.

14.4.1 Requirements

Requirements of this assignment:

- only use prime numbers to test for factors; for example, do not check if 4 is a factor of 49.
- store the prime numbers in an array of integers
- use a symbolic constant to define the size (number of items) of the array
- define and use at least one of the following two subroutines:
 - define a function that returns a boolean value, indicating whether a number is prime or not
 - define a function that returns a prime factor of a given number
- do *not* use global variables in any subroutine
- 2 is the only known prime number that you can initialize the array with, all other prime numbers must be computed
- make use of the following statement to make your program efficient: “if you cannot find a factor f for n such that $f^2 \leq n$, n has no factor except 1 and itself.”
- name subroutines, parameters and variables appropriately so that I can tell what each one does in the program.
- the program should print *the array of prime numbers* at the end of the main program, do not print each prime number as it is discovered.

14.4.2 How to turn in

Submit as “CISP365 Project6 by *your name*” to tauyeung@drtak.org. Submissions not having the correct subject line or sent to the other email address are subject to point deduction!

14.4.3 Warning

This is an open ended program with lots of flexibilities. As a result, I do not expect any two programs to be structurally the same. If you plan to help someone else in the class, by all means do so. However, do so *without* disclosing any of your own Pascal code. When you help a fellow student, do the following:

- reexplain Pascal constructs, using examples totally unrelated to the homework assignment.
- reexplain the homework assignment, but only explain so the other student understands *what* is required, but not *how* to do it.

I am aware that many of you work in groups to help each other, that is fine. However, I am going to make sure that each one of you work on your program individually. Code copying/sharing is not allowed. If any two submitted programs look structurally “too similar”, I will, at my discretion, given zero points to the owners of the similar programs.

Chapter 15

Array Related Algorithms

Arrays are important constructs in any programming language because their properties. First, each element in an array is accessed by its index number. It is just as quick to access the first element and the last element in an array. This property is often known as “random access”. Second, it is possible to use a variable to serve as an index to an array. This provides the means to use algorithms to perform operations on elements in an array. Useful operations include searching and sorting.

15.1 Searching in an Unsorted Array (Linear Search)

Provided that an array is not sorted, searching for a particular element in it is time consuming. Imagine you need to look up a name in a phone book that is not arranged by first name or last name. Such a task is difficult because you need to read and check every single item in the phone book to search for the one that you want to find. The worst case is when the name being found is not in the phone book. You will need to read and match every item in the phone book before concluding the name you want to search for doesn't exist!

Given an array `nums` with `n` elements that are unsorted, we can write a function to return whether an element `e` exists in the array or not.

```
function findElement(var nums : array of integer;
                    n : integer;
                    e : integer) : boolean;
var
    found : boolean;
    i : integer;
begin
    found := false;
    i := 0;
```

```

while (not found) and (i < n) do
  begin
    found := (nums[i] = e);
    i := i + 1
  end;
findElement := found
end;

```

Note that in this function, we passed `nums` by reference even though it is not modified in the subroutine. This is because passing by reference is much more efficient than passing by value, especially when it comes to big items like arrays.

15.2 Searching in a Sorted Array

If an array is sorted, the previous algorithm still works. We can, however, easily refine it so it searches slightly more efficiently. Assuming the given array `nums` is sorted in a non-decreasing order, the function can be rewritten as follows:

```

function findElement(var nums : array of integer;
                    n : integer;
                    e : integer) : boolean;

var
  found : boolean;
  i : integer;
begin
  found := false;
  i := 0;
  while (not found) and (i < n) and (e >= nums[i]) do
    begin
      found := (nums[i] = e);
      i := i + 1
    end;
  findElement := found
end;

```

The modification here is the introduction of `(e >= nums[i])` in the `while` condition. This additional condition lets the algorithm exit as soon as `e` is greater than the current element in the array. This is wise because if there is no match for `e` up to an element that is greater than `e` in a sorted array, all other elements must be greater than `e`.

There is a far more efficient method to search in a sorted array called “binary search”. In this method, we keep track of a range of elements in the array that can contain the one that we are looking for. In each step, we select the middle element in the range to match with the value being searched. Based on the comparison, we can either locate the element being found or eliminate at least half of the remaining elements in the array.

Let us assume the array `nums` is sorted in a non-decreasing fashion. This time, we implement the binary search algorithm:

```
function binsrch(var nums : array of integer;
                n : integer;
                e : integer) : boolean;
var
  lbound, hbound, midpt : integer;
begin
  lbound := 0;
  hbound := n - 1;
  while (lbound <= hbound) and (nums[(lbound + hbound) div 2] <> e) do
    begin
      midpt := (lbound + hbound) div 2;
      if (e < nums[midpt]) then
        hbound := midpt - 1
      else
        lbound := midpt + 1
      end;
      binsrch := lbound <= hbound;
    end;
end;
```

This algorithm is best demonstrated by a few concrete examples.

15.3 Sorting

Because a sorted array facilitates *much* faster search algorithms, it is desirable to sort an array. There are many algorithms for sorting arrays. Unfortunately, the efficient ones tend to be difficult to understand.

This section presents an inefficient method to sort an array, but it is considered a little easier to understand. This is called selection sort.

In selection sort, the algorithm keeps track of an unsorted range in the array (to be sorted). In this range, the algorithm finds the least element, and swap it with the first element in the range. Then the algorithm makes the unsorted range one element smaller and repeat the same process.

The algorithm is presented as the following Pascal procedure:

```
procedure selectSort(var nums : array of integer;
                    n : integer);
var
  t : integer;
  i : integer;
  minIndex : integer;
  firstUnsorted : integer;
begin
  firstUnsorted := 0;
```

```

while (firstUnsorted < n - 1)
  begin

    {first the least element from firstUnsorted}
    minIndex := firstUnsorted;
    i := firstUnsorted+1;
    while (i < n) do
      begin
        if nums[minIndex] < nums[i] then
          minIndex := i;
          i := i + 1
        end;

        {swap the least element with what's occupying its place}
        t := nums[firstUnsorted];
        nums[firstUnsorted] := nums[minIndex];
        nums[minIndex] := t;

        {advance firstUnsorted}
        firstUnsorted := firstUnsorted + 1
      end;
    end;
end;

```

This algorithm is a little difficult to read because it is too long. We can convert this long procedure into two to make it more readable. We can extract the inner `while` loop because it serves a particular purpose: to find the index of the least element from `nums[firstUnsorted]` to `nums[n-1]`:

```

function findMinIndex(var nums : array of integer;
                     n : integer;
                     firstUnsorted : integer) : integer;
var
  minIndex : integer;
  i : integer;
begin
  {first the least element from firstUnsorted}
  minIndex := firstUnsorted;
  i := firstUnsorted+1;
  while (i < n) do
    begin
      if nums[minIndex] > nums[i] then
        minIndex := i;
        i := i + 1
      end;
    findMinIndex := minIndex
  end;
end;

```

It also helps to define a small procedure to encapsulate the swap operation:

```
procedure swap(var x, y : integer);
  var
    t : integer;
  begin
    t := x;
    x := y;
    y := t
  end;
```

We now have the function `findMinIndex` to find the index of the least element from index `firstUnsorted` to the end of the array. We also have a procedure `swap` to swap the values of two integers. Now, we can specify the overall sorting algorithm more clearly:

```
procedure selectSort(var nums : array of integer;
                    n : integer);
  var
    t : integer;
    minIndex : integer;
    firstUnsorted : integer;
  begin
    firstUnsorted := 0;
    while (firstUnsorted < n - 1)
    begin
      minIndex := findMinIndex(nums, n, firstUnsorted);
      swap(nums[minIndex], nums[firstUnsorted]);
      firstUnsorted := firstUnsorted + 1
    end;
  end;
```

15.4 Homework Assignment

In this assignment, the computer plays the game of number guessing using binary search technique. In other words, the following is a scenario of a game play:

```
I have 6 guesses, is it 25?
-1
I have 5 guesses, is it 12?
1
I have 4 guesses, is it 18?
1
I have 3 guesses, is it 21?
-1
```

```
I have 2 guesses, is it 19?  
0  
I guessed the number in 5 guesses.
```

In this program, the computer needs to keep track the number of guesses remaining (starting with 6) and use the binary search technique to guess a number between 0 and 50. After the computer presents a guess, the user can answer with -1 if the guess is too little, 0 if the guess is correct and 1 if the guess is too large.

The program needs to indicate that the user is cheating if the user inputs inconsistent hints.

15.4.1 Hint

You can start with the binary search code in this chapter. Make sure you understand how that works first. However, you do not need any array in this program. You do need to add some `read`, `readln`, `write` and `writeln` statements to make the code interactive.

You don't need to make this program loop. In other words, each time you run the program, it only needs to play one game.

You can turn the binary search subroutine into a `procedure` because there is nothing to return to the main program. The main program can just call the binary search subroutine once.

15.4.2 How to submit?

Use "CIS31 Project4 by *your name*" as the subject of your email. Send only the `.pas` file. You have one week from the date of assignment to complete this assignment.

Chapter 16

Records

Arrays represent one of the two congregates that are useful in programming. As a quick reminder, an array contains multiple elements *of the same type*. This means it is impossible for an array to contain some integers, some real numbers and some booleans. Furthermore, each element in an array is selected by its *index*, which is an integer. In most languages, elements in an array must start with index 0 and use consecutive integers thereafter.

Sometimes, however, it is helpful to group items of different types together into a single unit. For example, consider a real student record. On a student record, the `firstname`, `lastname` and `address` are “text” fields. The `GPA`, however, is a real number. The `student number` is an integer field, while the `birthdate` is a data field. It is helpful to see the entire record as one single object sometimes. For example, it is convenient to say “make a copy of Joe’s student record”. The alternative is to specify making a copy of every field that applies to Joe, which is tedious and error prone.

This chapter discusses the general concept, syntax and uses of records in Pascal. Most of the conceptual discussions apply to Pascal and most other modern programming languages, while the syntax and sample programs are more Pascal specific.

16.1 Concepts

A record, as discussed earlier, is a collection of items of potentially different types. This section discusses more formally what a record is and its components.

First of all, in Pascal, a **record** (called a **struct** in C and a **class** in C++ and Java) is a type. You *can* make record variables from a record type, just like you can make integer variables from the **integer** type. This is analogous to treating “student record” as the name of the template empty record, rather than an actual record of a particular student.

In a record, a programmer can specify any number of **fields**. A field has two attributes. First, a field has a name. The name of a field must obey the

same rules as variable names. The name of a field uniquely identifies a field from all other fields in the same record. As a result, names of fields of the same record must be unique. On the other hand, because a field name is “local” to the definition of a record, the same name can be reused as the name of a global variable, the name of a subroutine, or the name of a field in another record.

The other attribute of a field is its type. A field can have any type that a variable can have. In other words, a field can be an **integer**, a **real**, an **array** of something, or even a record.

16.2 Syntax

16.2.1 *Standalone* Record Definition and Reference

In Pascal, you can directly define a record when you define variables. For example, if I want to define two variables to represent two points in a plane, I can do the following:

```
var
  pointA, pointB : record
    x : real; { x coordinate}
    y : real { y coordinate}
  end;
```

In the above definition, **pointA** and **pointB** are two variables being defined. Their type is a record that contains two fields. One field is named **x** while the other one is named **y**.

With this definition, I can write some code to compute **pointB** given both points are on a linear line with slope **m** and the x-coordinate of **pointB** is **someX**:

```
pointB.x := someX;
pointB.y := m * (pointB.x - pointA.x) + pointA.y;
```

The dot **.** is used to specify the name of a field. **pointB** refers to the variable that is a record with two fields. **pointB.x**, however, refers to the field named **x** in variable **pointB**. As a result, **pointB.x** is a storage location of **real** type. **pointB.x** can go anywhere a variable of **real** type goes.

While this definition works, it is cumbersome if we need to define more objects of this record type. For example, we may need to define local variables and parameters of this record type.

Every time we need to define another object (variable or parameter) of this record type, we need to include the details of the record. Not only is this cumbersome, it is also error prone because we are effectively copying and pasting the definition of the record all over a program.

16.2.2 *Named* Record Definition and Reference

In Pascal and many other languages, you can associate a symbolic name to a record type. This is very handy because now we only need one single definition of the record type, then simply refer to the symbolic name of the record type throughout the program. For example, the following code defines a symbolic name `Point2D` for our previously mentioned record type:

```
type
  Point2D = record
    x : real;
    y : real
  end;
```

Note that when we are defining a symbolic name as a type, we use the equal sign = and not a colon :. The `type` reserved word tells the Pascal compiler that the following definition(s) is/are type definitions (as opposed to `constants` or `variables`).

With this definition, we can refer to the record type using its symbolic name. For example, to define the two variables `pointA` and `pointB`, we can use the following code:

```
var
  pointA, pointB : Point2D;
```

In fact, we can use the name `Point2D` much like any other type name (such as `integer`). For instance, if we need to define a procedure to compute slope of intercept between two points, we could use the following definition:

```
procedure getSlopeIntercept(a, b : Point2D;
                           var slope : real;
                           var intercept : real);
begin
  slope := (b.y - a.y) / (b.x - a.x);
  intercept := a.x - slope * (b.x - a.x)
end;
```

16.2.3 *Named* Type Definition and Reference

As it turns out, we can define a name for *any* type, including arrays! For example, we can use the following type definitions for two array types:

```
const
  bigSize = 100;
  smallSize = 10;
type
  BigIntArray = array [0..bigSize-1] of integer;
  SmallIntArray = array [0..smallSize-1] of integer;
```

If we need two `BigIntArrays` and three `SmallIntArrays`, we can use the following variable definitions:

```
var
  bigA, bigB : BigIntArray;
  littleC, littleD, littleE : SmallIntArray;
```

Note that the type names can be used to define parameters and local variables as well as global variables. This helps both in terms of making a program more legible (due to the lack of cluttering `array` or `record` definitions) and easier to maintain (only need one change to the definition of a type name instead of changes to individual `array` or `record` variable definitions throughout the program).

In most other languages, you can associate a symbolic name with record types but not array types.

Chapter 17

Abstract Data Types

With the introduction of `records` in the previous chapter, we are moving into more complicated programs. This chapter first presents the concept of a circular queue, then proceeds to discuss what abstract data type is and why it is an important concept in programming.

17.1 Circular Queue

17.1.1 General Queue (FIFO)

A queue is a first-in-first-out (FIFO) container. In other words, the first item put into a FIFO is the first item to remove. FIFO can be found in real life, such as lines at banks, supermarkets and warehouse stores. FIFO can also be found in computer applications. For example, the buffer that allows smooth playing of streaming video or audio over an internet connection with even speed is a FIFO. Frames arriving earlier are always displayed before frames that arrive later.

Every queue has a head. The head of a queue is the next item to remove. Every queue also has a tail. The tail of a queue is the most recent item added to the queue. When one removes an item from a queue, it is always removed from the head (so the next item becomes the new head). Similarly, whenever one adds a new item to a queue, it is always added to the tail so the new item becomes the new tail.

17.1.2 Circular Queue

FIFO does not imply “circular”. A FIFO simply means a container has the particular behavior of removing items in the same order as the arrivals of the items. A *circular* FIFO, or circular queue, has all the properties of a FIFO. In addition, it also reuses empty slots that have been emptied.

Although a circular queue also has head and a tail, it is not easy to track them. This is because when the head and the tail are at the same slot, it is impossible to tell if the circular queue is actually full or empty. This makes

perfect sense. When a circular queue is empty, the head and the tail are at the same slot. This is because the “head” really means “the next item to remove”, whereas the “tail” means “the next slot to be filled”. When a circular queue is full, however, the head and the tail are the same again!

This means it is better to represent a circular queue remembering the head and a counter that count the number of used slots. The means the tail is computed from the head and the in-use-counter.

17.1.3 Pascal record Implementation

Given the understanding from the previous subsection, the definition of a circular queue is as follows:

```
const
  QCapacity = 8;
type
  CirQ = record
    head : integer;
    usedCount : integer;
    buffer : array [0..QCapacity-1] of integer
  end;
```

This is a definition for a circular queue to store integers. You can substitute `integer` in the definition of other types, including other user defined types.

17.1.4 Operations for Circular Queues in Pascal

Given the definition of `CirQ` from the previous subsection, we can write subroutines to perform certain operations to a `CirQ` variable.

Initialize

Even though some Pascal implementations guarantees that variables are filled with zeros upon creation, this is a risky assumption. As a result, it is better to provide our own subroutine to initialize variables of `CirQ` type.

```
procedure CirQInitialize(var cq : CirQ);
begin
  cq.head := 0; { force head to be 0 }
  cq.usedCount := 0 { force queue to be empty }
end;
```

In this subroutine, we reset the head and the used counter both to 0. Although technically speaking, we can reset the head to any value between 0 and `QCapacity-1`, it is more “usual” to reset it to zero.

Check for Empty

We should provide a subroutine to check if a circular is empty. The following Pascal code does that:

```
function CirQIsEmpty(var cq : CirQ) : boolean;
begin
  CirQIsEmpty := (cq.usedCount = 0)
end;
```

This function returns a value of true if and only if the circular queue `cq` is indeed empty. With a counter for the number of used items, it is easy to determine whether a circular queue is empty or not. Note that we are passing `cq` by reference *only* before we want to improve efficiency.

Check for Full

Checking for full is simple:

```
function CirQIsFull(var cq : CirQ) : boolean;
begin
  CirQIsFull := (cq.usedCount >= QCapacity)
end;
```

Add an Item

This function adds an item `item` to a circular queue `cq`:

```
function CirQAddItem(var cq : CirQ;
  item : integer) : boolean;
begin
  if not CirQIsFull(cq) then
    begin
      cq.buffer[cq.head + cq.usedCount] := item;
      cq.usedCount := cq.usedCount + 1;
      CirQAddItem := true { successful }
    end
  else
    CirQAddItem := false; { failed! }
  end;
```

This subroutine is a function because it is not always possible to add an item to a circular queue. If the queue is full already, we cannot add another item. This function returns true if and only if the item is successfully added to the tail of circular queue `cq`.

Remove and Item

This function removes an item `item` from a circular queue `cq`:

```
function CirQRemoveItem(var cq : CirQ;
    var item : integer) : boolean;
begin
    if not CirQIsEmpty(cq) then
        begin
            item := cq.buffer[cq.head];
            cq.head := (cq.head + 1) mod QCapacity;
            cq.usedCount := cq.usedCount - 1;
            CirQRemoveItem := true
        end
    else
        CirQRemoveItem := false
    end;
end;
```

Again, this is a function because it is not always possible to remove an item from a circular queue. If a circular queue is already empty, no item can be removed. This function returns true if and only if an item is successfully removed from the queue.

17.2 Using CirQ *not* As an ADT

Let's see what happens when we don't treat `CirQ` as an abstract data type.

You may have noticed the simplicity of `CirQIsEmpty` and its counterpart `CirQIsFull`. In a program, one can easily use `cq.usedCount = 0` instead of `CirQIsEmpty(cq)`. For example, I can write the following program to test whether my circular queue logic is working:

```
const
    QCapacity = 8;

type
    CirQ = record
        head : integer;
        usedCount : integer;
        buffer : array [0..QCapacity-1] of integer
    end;

{ ... definitions of the 5 subroutines discussed earlier }

var
    cq : CirQ;
    selection : integer;
```

```

    item : integer;
    result : boolean;

begin
    cq.head := 0;
    cq.usedCount := 0;
    repeat
        repeat
            writeln('select one of the following:');
            writeln('-1: exit');
            writeln('0: add an item');
            writeln('1: remove an item');
            readln(selection);
            if (selection < -1) or (selection > 1) then
                writeln('error: invalid selection')
            until (selection >= -1) and (selection <= 1);
            if selection = 0 then
                begin
                    writeln('enter number to add');
                    readln(item);
                    if cq.usedCount = QCapacity then
                        writeln('queue is full already, operation failed!')
                    else
                        result := CirQAddItem(cq, item)
                    end
                end
            else
                if selection = 1 then
                    begin
                        if cq.usedCount = 0 then
                            writeln('queue is empty, operation failed!')
                        else
                            begin
                                result := CirQRemoveItem(cq, item);
                                writeln('removed value is ',item);
                            end
                        end
                    end
                until selection = -1;
            writeln('this concludes our test.')
        end.

```

While this program works, what happens when I change the implementation of `CirQ`? In other words, what if I simply change the field name `head` to `nextToRemove` and the field name `usedCount` to `numOfUsedItems`?

Of course, I am expected to change the five subroutines defined earlier. However, I also have to change all the direct references to `head` and `usedCount` throughout the program.

This program is a “bad” program because it uses the internals of `CirQ` instead of just using the five subroutines defined for the type `CirQ`. Because of the bad programming practice, the program is sensitive to changes to the definition of `CirQ`.

Although this test program is short and it is relatively easy to fix all the references to the internals of `CirQ`, a realistic program is far more difficult to fix when there are easily tens of thousands of lines of code. Even a search-and-replace approach may not work if `head` and `usedCount` are names used in other contexts.

The problem of using the internals of a type is more significant when changes are done to the basic implementation of a type instead of just names of fields. For example, if `CirQ` is implemented by pointers and dynamically allocated memory, no editor feature can help to bring the rest of the program up-to-date.

17.3 Using `CirQ` as an ADT

Instead of using `cq.usedCount = 0` in the main program, we should use `CirQIsEmpty(cq)` instead. As long as we do not peek into the internal structure of `CirQ` in the program *other than* the five subroutines defined to handle `CirQ`, we are treating `CirQ` as an abstract data type.

After we apply the techniques of ADT, our program becomes the following:

```

const
  QCapacity = 8;

type
  CirQ = record
    head : integer;
    usedCount : integer;
    buffer : array [0..QCapacity-1] of integer
  end;

{ ... definitions of the 5 subroutines discussed earlier }

var
  cq : CirQ;
  selection : integer;
  item : integer;
  result : boolean;

begin
  CirQInitialize(cq);
  repeat
    repeat
      writeln('select one of the following:');

```

```

writeln('-1: exit');
writeln('0: add an item');
writeln('1: remove an item');
readln(selection);
if (selection < -1) or (selection > 1) then
  writeln('error: invalid selection')
until (selection >= -1) and (selection <= 1);
if selection = 0 then
  begin
    writeln('enter number to add');
    readln(item);
    if CirQIsFull(cq) then
      writeln('queue is full already, operation failed!')
    else
      result := CirQAddItem(cq, item)
    end
  end
else
  if selection = 1 then
    begin
      if CirQIsEmpty(cq) then
        writeln('queue is empty, operation failed!')
      else
        begin
          result := CirQRemoveItem(cq, item);
          writeln('removed value is ',item);
        end
      end
    end
  until selection = -1;
  writeln('this concludes our test.')
end.

```

Although the changes seem subtle and worthless, this program is now insulated from changes to the internal structure of `CirQ`.

17.4 Homework Assignment

In this assignment, write a program that implements and tests the stack ADT. You can borrow some code from the implementation of the circular queue ADT. To be more specific, first define the type “`StackType`”, then write the following interface subroutines to operate on a stack:

- `initializeStack(var s : StackType)` is a procedure to initialize a stack object so it is empty
- `isEmpty(var s : StackType) : boolean` is a function that returns true if and only if the stack has no elements

- `isFull(var s : StackType) : boolean` is a function that returns true if and only if the stack is full
- `push(var s : StackType; v : integer) : boolean` is a function that attempts to put an integer into a stack, returns true if and only if the operation is successful
- `pop(var s : StackType; var v : integer) : boolean` is a function that attempts to remove an integer from a stack, returns true if and only if the operation is successful

In addition, you also need to write an interactive test program to test whether the five subroutines are working or not.

17.4.1 Stack Revisited

A stack is a container where items follow the last-in-first-out order. In an array implementation, you only need an integer that indicates the “top” of a stack. There are two definitions of “top”. “Top” can refer to the index of the element that will hold the *next* item pushed. “Top” can also refer to the index of the element that was last pushed. Either way, as long as you use the same meaning consistently throughout the program, it should not matter much.

For this program, assume the stack has a capacity of 5 elements, and each element is of type integer. In other words, use an array of 5 integers to store the values.

You need to define your own type for a stack. This type should be a record that has fields for each necessary component for a stack.

17.4.2 Interactive Tester

The interactive tester should be a loop. For each iteration, allow the user to select from the following choices:

- 0 to exit
- 1 to initialize (no feedback)
- 2 to check empty (report whether the stack is empty)
- 3 to check full (report whether the stack is full)
- 4 to push an item (ask for value to push)
- 5 to pop an item (report whether an item was popped, and if so, value of the popped item)

17.4.3 How to Turn in

Use “CIS31 Project5 by *your name*” as the subject of your email. Send only the `.pas` file. You have two week from the date of assignment to complete this assignment.

Part VI

Text and File Processing

Chapter 18

Strings

Up to this point, we have only used numerical or boolean types in our programs (as atomic types). Most practical programs, on the other hand, are likely to process text information (such as names and addresses) as well. This chapter discusses how a Pascal program can handle text information.

18.1 Character Encoding

Most media and transmission means of computers are digital. To be more specific, they are binary. In other words, we can only store information or transmit (and receive) information in 0s and 1s. This is why we had to discuss the internal binary representation of integers and floating point numbers (real numbers in Pascal).

How are going to use 0s and 1s to represent text? The American Standard Code for Information Interchange (ASCII) is a lookup table to map *characters* to/from bit patterns. Characters in this context include letters in the English alphabet, digits (0 to 9), punctuations (comma, semi-colon, colon and etc.) and certain unprintable characters like carriage return and line feed.

For example, to start a new line, a DOS/Windows environment requires the use of a carriage return and a line feed character. The bit pattern for carriage return is 00001101_2 , while the bit pattern for line feed is 00001010_2 . This means to make a new line in a file, the bit pattern 00001101000010101_2 should be used. You can find a complete table of ASCII encoding on most computers or online. For example, <http://www.asciitable.com> has such a table (but not in bit patterns).

ASCII only requires 8 bits per character. This is no longer sufficient now that computers are used in most countries using different languages. This is why Unicode is going to replace ASCII as the standard encoding scheme. In Unicode, each character has 16 bits. This means there can be up to 65536 distinct characters. On the other hand, because Pascal is an old language that is not updated anymore, all of our programs will use ASCII encoding.

18.2 The char Type

In Pascal, you can use variables of `char` type to store single characters. Because `char` is an internal type, Pascal knows how to compare, read and write values of this type. When characters are compared, the ordering is based on the *value* of the bit pattern representing the characters. For example, the character 'a' has an ASCII code of 01100001_2 , while the character 'A' has an ASCII code of 01000001_2 . This means Pascal considers 'A' to be less than 'a' because the bit pattern 01100001_2 can be interpreted as 97, while the bit pattern 01000001_2 can be interpreted as 65, and 65 is less than 97.

The following code illustrate how one may use a variable of character type to handle user interaction:

```

var
  answer : char;
begin
  ...
  repeat
    write('please answer y for yes, n for no: ');
    readln(answer);
    if (answer <> 'y') and (answer <> 'n') and
       (answer <> 'Y') and (answer <> 'N') then
      writeln('invalid answer!')
  until (answer = 'y') or
        (answer = 'Y') or
        (answer = 'n') or
        (answer = 'N');
  if (answer = 'y') or (answer = 'Y') then
    begin
      { do whatever when the user answers yes }
    end
  else
    begin
      { do whatever when the user answers no }
    end;
end.

```

Characters, like any other type, can also be assigned.

Two special functions apply to values of `char` type. The function `chr` converts an ASCII code (as an `integer`) into the corresponding character, while the function `ord` converts a character (as a `char`) to its ASCII code (as an `integer`).

This is rather handy because it is impossible to type a carriage return character, but you can specify `chr(13)` instead.

18.3 String

While a single character is useful mostly for selecting ‘yes’ or ‘no’, single characters cannot be used to represent names or address. In order to represent text information that contains multiple characters, both Borland Pascal and FreePascal offer the special type `string`.

A `string` variable is *similar* to an array of `char`. However, a `string` can do more because it is recognized as an internal type in Pascal.

The following code defines a variable that is a `string` of up to 20 characters:

```
var
  firstname : string [20];
```

Because `string` is an internal type, you can use `readln(firstname)` to read a string or `writeln(firstname)` to print a string. However, there is more operations that you can do with string values.

18.3.1 String Comparison

You can compare strings using the intuitive symbols `<`, `>`, `=`, `<>`, `<=` and `>=`. The comparison of strings is based on the character encoding scheme.

18.3.2 String Concatination

Two strings can be concatenated (the second one appended to the end of the first one) to form a third string. For example, you can write the following code:

```
var
  firstname : string [20];
  lastname  : string [25];
  wholename : string [46];
begin
  {...}
  wholename := lastname + ', ' + firstname;
  writeln(wholename);
  {...}
end.
```

18.3.3 String Length

One advantage of the `string` type over an array of `char` is that a `string` automatically tracks the number of characters actually being used. In other words, you may define a variable (or parameter) as `name : string[20]`, but this only means you can store *up to* 20 characters. You can certainly store an empty string in `name`.

In FreePascal, you can check the length of a string using the built-in function `length`. The following code illustrates how to use `length`:

```

var
  whatYouType : string [80];
begin
  readln(whatYouType);
  writeln('you typed a string with ',length(whatYouType),' characters')
end.

```

18.3.4 Indexing in a String

Although a string is most useful when you treat it as a whole, you can also break it apart, character by character. Indexing into a string is the same as indexing into an array.

Because Pascal was originally designed for beginning programmers, the first character in a `string` type variable has an index of 1, not 0!

The following code counts the number of spaces in a string:

```

var
  inputLine : string [80];
  spaceCount : integer;
  i : integer;
begin
  readln(inputLine);
  spaceCount := 0;
  i := 1;
  while i <= length(inputLine) do
  begin
    if inputLine[i] = ' ' then
      spaceCount := spaceCount + 1
    end;
  writeln('the input line has ',spaceCount,' space(s)')
end.

```

18.4 Homework Assignment (200 points)

This assignment is due two weeks from its assignment date.

18.4.1 What to do?

In this assignment, apply concepts of abstract data type and also algorithms for sorting and searching. You will need to do this assignment in steps.

Step 1

Implement an abstract data type called student record. A student record type must store the following information

- student ID: this is an integer, let us assume a student ID is never going to exceed 32767.
- lastname: this is a string, allocate at least 20 characters
- firstname: this is also a string, allocate at least 20 characters
- GPA: this is a real number

As an abstract data type, you need to implement the following subroutines:

- a subroutine to put values into a student record, it will need a passed-by-reference parameter for the student record, then followed by parameters representing the data items in a student record.
- a subroutine to print fields of a student record. For efficiency, pass the parameter by reference
- a subroutine to compare two records. This is a special subroutine, it must be do the following:
 - the two student records to compare must be passed by reference for efficiency
 - an additional integer parameter indicates which field is used for comparison, use 0 for student ID, 1 for lastname and 2 for GPA
 - the subroutine must return one of three integer values. It returns 0 if the two records are the same, returns 1 if the first is greater than the second, and return -1 if the first is less than the second.

Step 2

Now, design a new type to represent a class. This new type should be a record by itself. It should contain an array of student records (with a minimum capacity of 10), and an integer that indicates the number of students in a class. Note that the size of a class is less than or equal to the capacity of the array of student records.

Treat the class type as an abstract data type. Define the following subroutines:

- a subroutine to read information of a class from user input. Use a student ID of -1 to indicate end-of-class. A single invocation of this subroutine should read in an entire class.
- a subroutine to print information of a class, it should print all student records in the class.
- a subroutine to sort a class based on a given sort criterion. The sort criterion code is the same as the comparison code for student records. The subroutine should sort in non-decreasing order.

Step 3

Now that we have all the abstract data types defined, write the following logic in the main program:

- read a class section in, assume there is at least one student
- sort the class by student ID, print the class
- sort the class by student lastname, print the class
- sort the class by GPA, print the class

18.4.2 Requirements

Both the student record type and the class type must be treated as abstract data types. In other words, only the designated subroutines can refer to the internals of the types.

In addition, use symbolic names for constants whenever possible. This includes the length of firstname, lastname, capacity of a class and etc. You can even use symbolic constants for the sorting/comparison criterion.

Subroutines cannot refer to global variables directly. Every bit of information required by a subroutine must be passed by parameters.

You can borrow any code in my classnotes or the sample folder. You cannot borrow code from your fellow classmates!

18.4.3 How to turn in

Send your Pascal program by email to tauyeung@drtak.org. Do *not* send to my ARC account! The subject should read “CISP365 Project7 by *your name*”.

18.5 Extra Credit Assignment

This is an extra credit extension to the previous assignment. In this assignment, the behavior of the program does not change much, but the implementation is a bit more efficient.

18.5.1 What to do?

Resorting the array of student records just to display records in a particular order is an expensive operation. As a result, it is more efficient to keep the student records sorted all the time, but only with the indices to the student records.

In other words, leave the original array of student records alone, unsorted. Add three arrays of integers to the class (or class section) type. Each array tracks the indices of student records sorted by a particular comparison criterion.

For discussion purposes, let me call these arrays of indices `ididx`, `lnidx` and `gidx` for the ID, lastname and GPA, respectively. Let me assume the array of student records is called `stus`. This means `stus[ididx[0]]` is the first student record when sorted by student ID, `stus[lnidx[0]]` is the first student record when sorted by lastname, and `stus[gidx[0]]` is the first student record when sorted by GPA.

On the other hand, `stus[0]` is the first student record entered by the user.

Do not use the same names as in this notes because they are way too concise! Invent your names for these fields. Points will be deducted if you use the same names as mine!

18.5.2 How to modify the program?

First, you need to modify the procedure to read in student records for the class. The arrays of indices need to be updated as new student records are entered. Before the indices are sorted, they should use the default sequential indices (0, 1, 2, and etc.).

Next, you need to modify the sorting algorithm. Instead of swapping and moving the records around, you need to swap indices around. Change the sorting algorithm to accept a new parameter passed by reference. This parameter should be the array of integers that acts as an array of indices.

Once you have the sorting algorithm modified, you need to call the sorting algorithm three times whenever a student record is added in the procedure that reads in students for the whole class. Each call to the sorting algorithm should handle the resorting of a particular comparison criterion and its corresponding array of indices. This way, as the program reads in students, the arrays of indices remain sorted all the time.

Last, you need to change the way student records in a class is printed. Add a new parameter to select by which order the student records should be sorted when they are printed. Do not sort the student records on the fly when you print a class, you just need to know how to use the sorted indices.

18.5.3 Tips

It is helpful to use a two-dimensional array in this assignment, although it is not completely necessary. As an example, the following is a variable that is a two dimensional array:

```
var
  twoD : array [0..2, 0..3] of integer;
```

The available elements are as follows:

```
twoD[0,0]
twoD[0,1]
twoD[0,2]
twoD[0,3]
```

```
twoD[1,0]
twoD[1,1]
...
```

The cool thing about a two-dimensional array is that you can pass a portion of it as a one-dimensional array. For example, consider the following procedure:

```
procedure p(var a : array of integer);
...
```

You can invoke `p` with portions of `twoD`. For example, the following invocation is perfectly fine:

```
p(twoD[j])
```

With this invocation, `a[i]` accessed in procedure `p` actually accesses `twoD[j, i]`.

This concept is helpful when you need to print the class according to a particular sorting criterion. You can always use a nested conditional statement if you don't want to use a two-dimensional array.

18.5.4 How to submit?

As usual, submit the Pascal file to `tauyeung@drtak.org` with a subject of “cisp365 project8 by *your name*”.

Chapter 19

File Operations

Although missing from the original Pascal language, extensive and practical file operations can be performed in Borland Pascal as well as FreePascal. Most of these operations have parallel operations in more popular languages such as C, C++ and Java.

This chapter first explains what files are, and what kind of file are available in the language. Then, this chapter introduces the built-in functions and procedures that a programmer can use to operate on files. The last portion of this chapter explains some of the most popular and useful algorithms related to files and data processing.

19.1 Files

This is, by no means, our first contact with files. As soon as we discussed `readln` and `writeln`, we have been using files all along! Yes, we have been using two special files called standard-in and standard-out.

In general, a “file” in a computer is rather abstract and does not have to resemble a physical file (as in files in a file cabinet). In reality, files are divided into pages, and each file has a limited capacity. In the computer world, a “file” is *much* more than that.

In the computer world, it is better to think of a “file” as a stream of bytes. There are three main type of such streams. An input stream is a stream that sources (provides) information to a program. An output stream is a stream that sinks (stores) information from a program. Often, we also use an input/output stream that can both source and sink at the same time.

The stream analogy works well for the special file called “console”. When the console is used an input stream, it is really the keyboard of a computer. When the console is used as an output stream, it becomes the screen of a computer. Note that neither the keyboard nor the screen has an inherit capacity. In other words, you can type as much as you want on a keyboard, and the screen can continuously display information (and scroll old materials off).

Other types of “files” that also has no inherit capacity includes the parallel port, the serial port, a network connection (for example, an ICQ connection) or the audio out port.

Of course, *most* files on a computer are more or less “regular”. This is “regular” in the sense of having a capacity and physically storing information on a medium.

19.2 Pascal File Operations

19.2.1 The File Type

In Pascal, as opposed to other languages, you can declare a file of a particular type. For example, if you have a student record type defined as follows:

```
type
  studentRecord = record
    name : string [30];
    ssn : string[9];
    gpa : real
  end;
```

You can store student records into a file declared as follows:

```
var
  studentFile : file of studentRecord;
```

Yes, `studentFile` is a variable. However, it is a special kind of variable that is accessible only via functions and procedures defined for file variables. Is this reminding you of abstract data types?

Also, note that `studentFile` is not a text file. It is a file of *fixed size records*. This means all records in this file have exactly the same size. As a result, the computer can locate records by the record number and size of record without needing separators. As a result, there is no end-of-line markers Between records.

If you don't want to specify what you can containing in a file, you can simply use `file` as a type:

```
var
  mysterious : file;
```

In this example, Pascal doesn't know what type of information is contained in `mysterious`. For such files, you can specify the size of each “record” in the file opening procedures (see later subsections).

For handling text, there is a special type of file called `text`. This kind of file can use special functions and procedures to handle text information. The standard input and standard output files, for example, are of `text` type. To make your own file variable to handle text, you can do the following variable delcaration:

```
var
  studentNames : text;
```

19.2.2 Associating a File variable to a File

With the declaration from the previous subsection, we have a variable that *can* be used to represent a file, but it is not associated to any particular file. This is okay if you just want to use the standard input and standard output files (keyboard and screen). However, if you want to associated a file variable to a particular file, you need to use the **assign** procedure.

The **assign** procedure expects a single parameter: the name of the actual file as a **string**. The following code associates **studentFile** to a file called **cis31.dat**:

```
assign(studentFile, 'cis31.dat');
```

Note that this operation only makes the association, but it doesn't do anything to the file. In other words, we do not specify whether we want to read from, write to, erase or append to the existing file with **assign**.

19.2.3 Opening a file

Once a file variable is associated with a particular filename, we can open the file using one of the following procedures:

- **reset**. This procedure opens a file for reading. It expects at least one parameters. The first parameter (mandatory) is a file variable. The second parameter, if provided, indicates the size of each record. Note that this procedure always places the file pointer to the first byte in the file.
- **rewrite**. This procedure is similar to **reset** in terms of expected parameters, but **rewrite** opens a file for writing and places the file pointer at the first byte. In FreePascal, a **reset** can follow a **rewrite** so the file can be both read from and written to.
- **append**. The procedure is similar to **rewrite**, but it specifies that the program wants to write starting at the end of the current file.

19.2.4 Moving to a Position

Sometimes, an application does not need to rewrite an entire file. It may need to update just a few bytes in the file. If this is the case, the file can be opened with **rewrite** followed by **reset** in FreePascal (or just **rewrite** in Borland Pascal).

Once a file is opened for modification, a program can move to a particular position using the **seek** procedure. The **seek** procedure expects two parameters. The first parameter is a file variable, while the second parameter indicates the index of record to move to.

Note that the index of record is specified much like indices for arrays. The first record is number 0, the second record is number 1 and etc.

The `seek` procedure doesn't work for files of `text` type.

19.2.5 Knowing more about a file and file operations

There are several functions that indicate the state of a file:

- `eof`: you can pass a file variable to the `eof` function so it returns true if and only if the file has reached its end. Note that this function only works for file variables of type `Text`.
- `eoln`: you can pass a file variable to the `eoln` function so it returns true if and only if the file has reached the end of the current line. Note that this function only works for file variables of type `Text`.
- `filepos`: call this function with a file variable and it returns the current position in the provided file. This function returns a `Longint` (instead of an `integer`) because file sizes can be big. Note that it returns the index of the current record in the file, *not* the byte position
- `filesize`: call this function with a file variable and it returns the size of the associated file. It returns the total number of records (not necessarily bytes) in the file.

19.2.6 Reading and writing

Most file operations are done with `write`, `writeln`, `read` and `readln`. However, with the introduction of files (esp. files or records), these procedure require some new explanation.

First of all, `read` and `write` work for all types of file, while `readln` and `writeln` only work for file variables of type `text`. This makes sense because a `file` or `studentRecord` does not use end-of-line to mark the end of one record.

To make these procedure write to a particular file instead of the standard input and standard output files, a program only needs to pass a file variable to these procedures as the first parameter. In our current example, if `myStudentRec` is of type `studentRecord`, we can use the following code to write this record to `studentFile`:

```
write(studentFile,mySTudentRec);
```

19.2.7 Flushing a file

The computer “buffers” most file related operations for efficiency. This, however, can cause problems. For example, in most communication programs, a query is sent to another computer before the originating computer waits for a reply. This becomes impossible when the originating computer “buffers” the query so it never leaves the originating computer.

This problem is easy to solve with `flush`. The procedure `flush` expects on file variable parameter. It clears out all the contents still in memory buffer to the actual file opened. Not only does `flush` work for output file, it also works for input files. After calling `flush`, an input file updates its buffer from the actual device (network, serial port or a regular file). This makes any “pending” information stored in a buffer for efficiency purposes available for the program to process.

19.2.8 Closing a file

When a file is opened by `reset`, `rewrite` or `append`, it is ready to be used (read, write or modify). When a program is done using a file, it should immediately close it. Closing a file is telling the operating system that all operations with the file are completed. This allows the operating system to perform whatever is necessary to “finalize” the file. This “finalization” is important because otherwise, some contents of the file may be stored in RAM and will be lost as soon as the computer powers down.

19.2.9 Deleting a file

The procedure `erase` is used to erase a file. Pass a file variable to the procedure to erase the file associated with the file variable. Note that this procedure should be called when a file is closed.

19.3 Merge Sort

Merge sort is a powerful sorting algorithm because it operates on files, not just arrays. This makes it possible to run merge sort on computers with little memory but lots of disk space.

There are two steps to each iteration in merge sort. The first phase is the splitting of a file into two, the second phase is the merging of the two files back into one.

19.3.1 Splitting

In this step, a file is split into two files. Contiguous “runs” of sorted sequences are identified in the original (input) file. Consecutive “runs” are output to the two output file alternately.

For example, if the following are numerical values in a file:

2,6,4,1,30,2,-1,3,0,1,2

It is chopped into runs (non-decreasing) as follows:

- 2,6
- 4

- 1,30
- 2
- -1,3
- 0,1,2

These runs are written to two output file in an alternating fashion. This results in the first file having the following contents:

2,6,1,30,-1,3

The second file has the contents of

4,2,0,1,2

19.3.2 Merging

In the merge step, both temporary output files from the split step are read into one output file. The algorithm picks whatever is smaller (when sorting in non-increasing order) from the current item of the two input files and write that to the output file.

If one file has reached end-of-file before the other does, we copy the rest of the input file that has not ended to the output file.

19.3.3 The Loop

Merge sort performs the merge and split operations until the output file of merge is sorted. This can be determined by checking if any two consecutive items written to the output file during the merge step is out of order.

Part VII

Advanced Discussion

Chapter 20

Handy but not Necessary Constructs

This section covers two Pascal language constructs that can simply a program a little. These constructs are not *necessary* because they can be implemented by other Pascal constructs that we have discussed already.

20.1 for

A **for** loop in Pascal is suitable for iterating for a fixed number of times. The general syntax of a **for** loop is as follows:

```
for variable := start [ to | downto ] stop do statement
```

In this notation, it means you can either use **to** or **downto** in a **for** statement. The use of **to** implies that the *variable* should increment from *start* to *stop* inclusively. The use of **downto** implies that the *variable* should decrement from *start* to *stop* inclusively.

You should write a program to see what the following code does:

```
for i := 3 to 10 do writeln(i)
```

What if you change **to** to **downto**? What if you swap the start and the stop values? What if you do both to this program?

The **for** loop is downplayed a little in this course because it has no counterpart in languages derived from C. The **for** loop in C-based languages is different from the Pascal **for** loop. The Pascal **for** loop exists in Visual Basic and other variants of BASIC.

20.2 case

The `case` statement is similar to the `switch` statement in C-based languages. This statement makes it easier for a program to examine a scalar value and determine what to do next.

A “scalar” value is either an integer or a character in Pascal. A case statement lists statements corresponding to specific values of the scalar value. For example, the following code prints a different message depending on the value of `i*j-20`:

```
case i*j-20 of
  2: writeln('two''s company');
  5: begin
      writeln('hawaii');
      writeln('five-oh')
    end;
  7: writeln('seven years in tibet');
 12: writeln('the dirty dozen');
  else
    writeln('I don''t know a movie with this number.');
```

end

This code prints a movie/series name based on the value of `i*j-20`. Note that if there is more than one statements for a particular value, you must use a block statement to encapsulate the statements. The `else` reserved word matches all values not listed explicitly as a “catch all” mechanism.

As powerful this statement may seem, it can be implemented by a series of if-then-else statements. The `case` statement also has limitations. For example, “scalar” values do not include string values. In addition, the values to match statements in a `case` statement must be constants.

Chapter 21

Recursion

Since we have already covered most topics about subroutines, it is time to discuss recursion. In short, recursion is the act of a subroutine calling itself. At first, this concept seems almost absurd. However, as we go through the different concepts and mechanisms in this chapter, you will probably begin to appreciate recursion.

21.1 Call Frame

A call frame, also simple called a “frame” or an “invocation frame”, is a contiguous memory area to store most information related to the invocation of a subroutine. The following items are stored in a call frame:

- return address: this is the address of the statement (instruction) to return to after the subroutine is completed
- local variables: these are variables defined local to this subroutine. All such variables begin to exist when the subroutine gets control, and all such variables cease to exist when the subroutine returns.
- parameters: these are interface items for a caller to pass information to a subroutine. *Both* parameters called by reference and called by value are stored in a call frame.
- return value: this is the value returned by the subroutine (must be a function in Pascal) to its caller. Whether this item is in a call frame is somewhat up to each implementation, but for discussion purpose we can assume so.

It is important that we understand a call frame is allocated and constructed every time we invoke a subroutine. This is not the same as saying there is always one call frame for each subroutine.

In other words, call frames are allocated and constructed on-the-fly and on-demand. As soon as a subroutine returns, the call frame associated with it is destroyed so the resources can be recycled. This means a program with 200 different procedures and functions does not necessarily need 200 call frames allocated. The total size of call frames simultaneously needed depends only on the depth of nesting of subroutine invocation.

21.2 Self-Similar Patterns

Before we actually discuss recursion in programming, let us first discuss some self-similar patterns in real-life.

21.2.1 Factorial

Factorial is a function in mathematics that is used frequently in probability computations. The factorial of n is often written as $n!$, where it is defined as $n! = n \times (n - 1) \times (n - 2) \cdots 1$. For example, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

An alternative way to define factorial is the following definition:

$$n > 1 \Rightarrow n! = n \times (n - 1)! \quad (21.1)$$

$$n \leq 1 \Rightarrow n! = 1 \quad (21.2)$$

This makes sense because $4 \times 3 \times 2 \times 1$ is $4!$. We can rewrite $5! = 5 \times 4!$.

21.3 Recursive Function

Because factorial has a recursive definition, it is relatively to write a recursive function for it:

```
function factorial(n : integer) : integer;
begin
  if n <= 1 then factorial := 1
  else factorial := n * factorial(n-1)
end;
```

When n is less than or equal to 1, it is called a “base case”. This is the moment when recursion stops and an actual value is returned immediately. All recursive definitions need to have at least one base case so the recursion stops at some point.

If n is greater than one, then we rely on `factorial` to find the value of the factorial of $n-1$, then multiply that value to n for the result.

The following subsection attempts to follow the execution of a recursive definition of factorial.

21.3.1 Tracing Recursion

Tracing a recursive program is not much different from tracing a regular program. The most important point is to follow the code exactly and keep track of the current call frame.

Let us track the execution of `factorial(3)`.

1. The first invocation invokes with `n` equals to 3. The first call frame is constructed with `n=3` and the return address is whatever code we need to return to.
2. The conditional statement dictates that we need to compute `n * factorial(n-1)`. This means we need to compute `factorial(2)`.
3. The second call frame is now constructed with a return address to continue the computation of `3 * factorial(2)` and `n=2`.
4. Again, the conditional statement dictates that we need to compute `2 * factorial(1)`.
5. The third call frame is constructed with a return address to continue the computation of `2 * factorial(1)` and `n=1`.
6. With this third call frame (`n=1`), the conditional statement executes the else-part and return a value of 1. This means `factorial(1)` evaluates to 1.
7. The third invocation of `factorial` returns. The associated call frame is deallocated. The result of 1 is “plugged into” `2 * factorial(1)` so it becomes `2 * 1`.
8. The result of `2 * 1` is returned as the second invocation of `factorial` concludes. This means the call frame of the second invocation of `factorial` (with `n=2`) is now deallocated. We return to the execution of `3 * factorial(2)`.
9. From the second invocation of `factorial` we know that `factorial(2)` is 2, we plug this into `3 * factorial(2)` and get an answer of 6.
10. The first invocation of `factorial` now returns with a result of 6. This value of 6 is “plugged into” whatever invoked `factorial(6)` in the first place.

It helps when one remembers that the allocation and deallocation of call frames follows the LIFO (last-in-first-out) order. The code always refers to the call frame at the “top” of the stack.

21.4 Demystifying Recursion

At first glance, recursion is very confusing. Two things about recursion makes it confusing:

- How can we call a subroutine that's still being defined?
- How about the local variables and parameters? How can a subroutine track different parameters? More specifically, how can a parameter of a subroutine track different values?

Here are the answers:

- A recursive subroutine only *looks* being defined from your perspective (as you are creating the subroutine). To the compiler, the entire subroutine is already defined by the time the program is compiled.
- There are two different aspects to a subroutine. The first is its definition. This is a *static* description of how the subroutine should behave. This is analogous to the definition of a word in a dictionary. The second aspect to a subroutine is the run-time environment that provides storage for local variables and parameters.

There is only one description of how the subroutine should perform operations. However, there is one *environment* to store the values of parameters and local variables *for each* invocation of the subroutine. This means each invocation can track its own parameters and variables.

An environment in this context is a call frame. Recall that a new call frame is allocated and created for each invocation of any subroutine. This is how Pascal (and most other modern programming languages) can isolate the parameters and local variables of one invocation from another.

Calling recursively is really no different from calling in general. Instead of getting puzzled by the apparent mystery of recursion, one way to deal with recursion is to forget it is recursion! Just remember *what* the subroutine is supposed to do and what it requires to know (parameters) when you are ready to call it from anywhere. This approach (of focusing on *what* and not *how* when you invoke a subroutine) usually helps.

21.5 Assignment (Extra Credit 200 Points)

In this assignment, you have to implement binary search recursively. Refer to section 15.2 for the iteration version.

Define your `binsrch` to have the same interface as the one in section 15.2. In other words, it should have the same parameters and return type. However, your version (recursive) should not have any loop.

To test your subroutine, you can either hardcode and initialize the elements in the array or write a small subroutine to read values into the array. The choice

is yours. I reserve the right to change your hardcoded values to fully test the functionality of your recursive subroutine.

For testing purposes, don't put too many elements into the array. Try 7 or even 3. Make sure the array is sorted non-decreasingly. Make sure that you can search all the values in the 7-element or 3-element array and report all values that do not exist in the array as missing.

The due date is two weeks after the assignment date. No late work is accepted on and after the first day of finals.

Chapter 22

Dynamically Allocated Memory and Pointers

At this point, we only store information into variables and parameters. Both variables and parameters are “named” with identifiers. While this is sufficient for most programs, variables and parameters have certain limitations.

One problem with variables, including global array variables, is that memory may not be efficiently utilized. For example, if we need to handle the processing of “who is taking what class” in memory, we need to reserve enough memory for the maximum number of students, the maximum number of classes and the maximum number of enrollments per class.

While this *can* be done, it is a waste of memory space. This is because in most cases, we do not need the maximum allocation. Reserving more memory than necessary is, in general, no a big problem. However, if enough applications running on a computer is doing this, a computer can quickly run out of memory.

Dynamically allocated memory helps to solve this problem by reserving memory “on-the-fly” only when more memory is needed. This way, an application only needs just enough memory to get the job done. Even better, this method allows a program to deallocate memory when it is no longer needed. When utilized correctly, a program using dynamically allocated memory puts a minimal demand on systems resources while getting the job done.

22.1 Pointers

A key concept with dynamically allocated memory is the concept of a pointer. A pointer is a relatively short bit sequence (typically 32-bit) that indicates an unique address in memory. In other words, a pointer contains the address of a byte in memory.

Note that a pointer is *not* the name of a variable!

In Pascal, you can define a variable that is a pointer to something:

```
var pointerToInteger : ^integer;
```

The previous definition creates a variable called `pointerToInteger` that is a pointer to an integer. Note that it does not make sense to write the following statement:

```
pointerToInteger := 125;
```

This is because `pointerToInteger` is a pointer, it is supposed to store an address, not an integer! In other words, a pointer is the label on an envelop, but not the envelop itself.

Let us assume that you also have a variable definition as follows:

```
var anInteger : integer;
```

Then you have execute the following assignment statement (in FreePascal, at least):

```
pointerToInteger := @anInteger;
```

The `@` symbol means “address of”. In other words, the previous statement means “update the value of `pointerToInteger` so it becomes the address of `anInteger`”. After this assignment statement, then you can actually store an integer *value*:

```
pointerToInteger^ := 125;
```

The carrot symbol `^` means “location pointed to”. As a result, the previous statement means “store the value 125 to the location pointed to by `pointerToInteger`”.

Note that you can execute `pointerToInteger^ := 125` before making `pointerToInteger` meaningful. This is a very dangerous thing to do because the computer will just overwrite *whatever* location `pointerToInteger` do happens to point to. Your program may not show any symptoms, your program may crash relatively soon, or your program may do something that is totally unpredictable.

22.2 Allocating and Deallocating Memory

Assuming `pointerToInteger` is defined as pointer to integer, you can ask the operating system to *allocate* a piece of memory that fits an integer, and change `pointerToInteger` to point to that location:

```
new(pointerToInteger);
```

This operation, however, overwrites the address previously stored in `pointerToInteger`.

Once you allocate memory to `pointerToInteger`, you can refer to the allocated location as `pointerToInteger^`. In fact, you can treat `pointerToInteger^` almost as a variable (and pass it to a passed-by-reference parameter).

When you don't need the integer anymore, you can deallocate the memory using the following statement:

```
dispose(pointerToInteger;
```

This statement deallocate the memory location pointed to by `pointerToInteger`. But it also does not update `pointerToInteger` so its value becomes `NIL`. `NIL` is a pre-defined constant for pointers that indicates that the pointer is pointing to nowhere. You need an explicit assignment statement to make a pointer point to `NIL`.

You can check if a pointer is pointing to “nowhere” using a comparison to `NIL`:

```
if pointerToInteger = NIL then
  writeln('the pointer is pointing to nowhere!')
else
  writeln('the pointer points to a location with value ',pointerToInteger^)
```

22.3 Practical Use of Pointers

While one can write to code to allocate memory for single scalar variables (for example, an integer), dynamic memory allocation is often used with more complex data structures. Most of these data structures include the use of a pointer as a field in a record.

22.3.1 A Linked List

Let’s say you need to write a program that has to process many student records in memory. Furthermore, let’s say you don’t even know what the maximum number of students is.

This is a bit of a problem because no matter how large of an array of student records you allocate, it may not be big enough. Besides, preallocating a large array of student records is wasteful and leaves little resource to the rest of the system.

As a result, this is one of the many applications of dynamic memory allocation. We are not changing the size of an array of student records. Instead, we allocate *one* student record at a time.

A Link Field

In this example, we still need the definition of a student record. For example, we may choose the following definition:

```
type
  studentRec = record
    firstname : string [20];
    lastname  : string[20];
    SSN       : string[9];
    GPA       : real;
  end;
```

However, we also need to represent a “node” in a linked list. A node in a linked list represents a student record. However, it also has a field to indicate where to find the “next” node. The following definitions define a node in a linked list:

```
type
  nodePtr = ^node;
  node = record
    content : studentRec;
    next : nodePtr;
  end;
```

These two type definitions require some explanation. The first one, `nodePtr = ^node;`, is a *forward* reference. It is defining the symbol `nodePtr` to be the name of a new type which is “a pointer to a `node`”. Yes, you are correct that `node` has not been defined yet. This is the only instance in Pascal where an undefined symbol can be referred to.

The second type definition defines the `node` type. It has two components. The first component, named `content`, has a type of `studentRec`. This means this field is, by itself, a record. Since `studentRec` is defined as a type, we really don’t care that it is a `record` or just any scalar type.

The second field, named `next`, has a type of `nodePtr`. This is the important one. This field is a pointer that points to the next node. We’ll see how this field is utilized in the next section.

A Linked List

To represent a linked list, all we really need as a variable is a pointer to the first one. In other words, if we need a variable to represent a linked list, we only need a definition like the following:

```
var
  studentList : nodePtr;
```

This is because all the actual student records are stored in dynamically allocated “nodes”. Each node points to the next one. So the program only needs to find the first one to locate the rest. As a result, we only need a pointer to a node (type `nodePtr`) to represent the entire list.

Initializing a Linked List

If we want a linked list to start empty, we can define a subroutine as follows:

```
procedure linkedListInit(var list : nodePtr);
begin
  list := NIL;
end;
```

Recall the NIL is a symbol that means “an invalid address”. This is just perfect for our application because when a list is empty, there is no first node to point to!

Check for Empty

The ADT function to check if a linked list is empty is simple:

```
function linkedListIsEmpty(list : nodePtr) : boolean;
begin
  linkedListIsEmpty := list = NIL
end;
```

This makes sense because an empty linked list (as initialized) has NIL for the pointer to the first node.

Add a Node

To add an item to a linked list, we have the following steps to perform:

1. allocate store for a new node
2. copy content to the `content` field of the new node
3. make this new node point to the current first node
4. make this new node the first node

To do this in Pascal, we can use the following subroutine:

```
procedure linkedListAdd(var list : nodePtr;
  var rec : studentRec);
var
  newNode : nodePtr;
begin
  new(newNode);
  newNode^.content := rec;
  newNode^.next := list;
  list := newNode
end;
```

The notation `newNode^.content` can be interpreted step-by-step. First, `newNode` is a pointer. This makes `newNode^` the node it points to. Eventually, `newNode^.content` means the `content` field of the node pointed to by `newNode`.

Remove a Node

The linked list we are using is called a singly-linked list. This means each node only has one link to the next node. This arrangement makes it easy to remove the first node in a linked list. Note that the first node in a linked list is the last one added.

But doesn't this mean the linked list is a LIFO? Yes! This particular arrangement makes the linked list an implementation of a LIFO container. As you can see now, arrays are not the only method to implement a LIFO.

It *is* possible to write code to remove a node *other than* the first node. However, the code to do this is considerably more difficult due to more special cases. As a result, I am only including the code to remove the first node.

There are a few steps involved to delete a node:

1. make a copy of the *pointer* to the first node
2. copy the *content* field to a passed-by-reference parameter
3. make the node following the node being deleted the new first node
4. delete the (old) first node

```
function linkedListRemove(var list : nodePtr;
    var rec : studentRec) : boolean;
var
    tobeRemoved : nodePtr;
begin
    if not linkedListIsEmpty(list) then
        begin
            tobeRemoved := list;
            rec := tobeRemoved^.content;
            first := tobeRemoved^.next;
            dispose(tobeRemoved);
            linkedListRemove := true
        end
    else
        begin
            linkedListRemove := false
        end
    end;
end;
```

Getting to the Next Node

A seemingly foolish function to find the next node of a node is defined as follows:

```
function linkedListNext(ptr : nodePtr) : nodePtr;
begin
    linkedListNext := ptr^.next
end;
```

It isn't much work to use `ptr^.next` instead of `linkedListNext(ptr)`. However, the advantage of using the latter is that it is an ADT. This is also a good habit, especially for growing into the use iterators in C++.

Reading Records from a File

This is just an example of how you can utilize a linked list. Let us assume that we have the following variable definitions:

```
var
  inFile : file of studentRec;
  tempRec : studentRec;
  studentList : nodePtr;
```

Then, we can write the following code to read records from a file into a linked list:

```
linkedListInit(studentList);
while not eof(inFile) do
  begin
    read(inFile, tempRec);
    linkedListAdd(studentList, tempRec)
  end
```

Scanning a Linked List

After records are read into a linked list, we can use the following code to scan through all records. Assume `i` is of type `nodePtr`.

```
i := studentList;
while i <> NIL do
  begin
    { ... do something with i^ }
    i := linkedListNext(i)
  end
```

Note that this code examines the *last* record read from the input file because the nature of this type of linked list is LIFO.

Deallocating an Entire Linked List

The following procedure, possibly an ADT interface, deletes everything in a linked list:

```
procedure linkedListRemoveAll(var list : nodePtr);
var
  tempRec : studentRec;
begin
```

```
while not linkedListIsEmpty(list) do
  linkedListRemove(list, tempRec)
end;
```

Part VIII
Appendices

Appendix A

Assignments

- Student Records 2: 18.5
 - both classes: due one week from 12/03
- Student Records 1: 18.4
 - PM class: due two weeks from 11/18
 - AM class: due two weeks from 11/18
- Find Prime: 14.4
 - PM class: due two weeks from 10/29
 - AM class: due two weeks from 10/29
- Functionize 2 assignment: 13.4
 - PM class: due one week from 10/22
 - AM class: due one week from 10/22
- Functionize assignment: 12.3
 - PM class: due one week from 10/08
 - AM class: due one week from 10/08
- Power of factor assignment: 10.7
 - PM class: due one week from 9/19
 - AM class: due one week from 9/19
- Quinary conversion assignment: 8.3
 - PM class: due one week from 9/8
 - AM class: due one week from 9/10

- Conditional statement assignment: 6.2.1
 - PM class: due one week from 8/27
 - AM class: due one week from 8/29