Procedures

Chapter 6
Structured Design Using Procedures

- **Structured Design** is an approach used to design solutions to larger problems in a systematic manner.
- In structured design, a problem is broken up into smaller pieces, each of which is solved individually.
- The solution to an individual piece of the problem is called a **procedure**.
- A procedure is a series of instructions that are grouped together and treated as a single unit.
- A **procedure** can be **called** from elsewhere in the (main) solution.
Structured Design Using Procedures continued...

- When a procedure is called, the statements control flows from the (main) solution to the statements in the procedure.
- When the procedure is finished, control returns to the calling statement.
Figure 6-12: Flow of a procedure call

Begin

Instruction

Instruction

DoWork

Instruction

Instruction

Instruction

End

DoWork

Instruction

Instruction

Instruction

Instruction

End of DoWork
Rotating Flags Problem

• Draw a flag
• Turn 360/8 degrees
• Draw flag
High Level Structured Design

- For Images = 1 to 8
  - Draw One Flag
  - Turn Right: 360/8
- EndFor

- This solution represents the highest level of the structured design. The details of how a flag is drawn are not important at this level—we’ll assume the OneFlag procedure is correct.
Creating a Procedure

- To create a procedure in Visual Logic, select “Procedures, Add New Procedure” from the main menu.
One Flag Procedure

Always end at the starting position!!
Another Program using OneFlag procedure

Begin

Colors Forward: 1

Pen Width: 3

Count 1 to 6

Begin

Count 1 to 6

Pen Width: 3

Colors Forward: 1

End

End

MoveTo: Count*20, -Count*30

OneFlag

End

Colors Forward: 256

MoveTo: Count*20, -Count*30

OneFlag
Write a Program to draw this!

DrawFlag procedure – OneFlag with Parameters
DrawFlag procedure – OneFlag with Parameters

Begin

Flags 1 to 8

DrawFlag(5*Flags, 20)

Turn Right: 360/8

End

Actual Parameters

Formal Parameters

DrawFlag(PoleLength, FlagLength)

Forward: PoleLength

Side 1 to 3

Back: PoleLength

End of DrawFlag(PoleLength, FlagLength)

Forward: FlagLength

Turn Right: 360/3
2 Parameters: PoleLength, FlagLength
Procedures with Parameters

- Procedures can be made more flexible by using parameters.
- A parameter is a means of sharing information between the calling program (main) and a procedure.
- The information in the calling code is the actual parameter.
- Since procedures can be called more than once, the actual parameters can be different in different procedure calls.
The corresponding variable in the procedure body is called a **formal parameter**. Formal parameters are declared in the Procedure Edit dialog box at the same time as the procedure name. Formal parameters are displayed in parentheses after the procedure name **in the procedure’s header and footer elements**.

The calling program specifies the **actual parameters** at the time of the procedure call. Actual parameters are displayed in parentheses after the procedure name **in the procedure call element**.
Formal Parameters/Actual Parameters

• List the names of the Formal parameters in the DrawFlag procedure:
  Parameter 1 is __________
  Parameter 2 is: __________

• How many times is this procedure called in the program?

• List the actual parameters in each procedure call:
  • Call 1:
  • Call 2:
Rotating Shapes Program

- This program draws a polygon as specified by the user as many times as the user wants. The 1\textsuperscript{st} input value is the number of times the user wants the polygon to be drawn. The 2\textsuperscript{nd} input value is the number of sides in the polygon to be drawn.

- Top-Level Structured Design:

- For N = 1 to 1\textsuperscript{st} input Value
  
  DrawFigure (2\textsuperscript{nd} input value)
  
  Turn Right: 360/1\textsuperscript{st} input value

- EndFor
Rotating Shapes Program

- Top-Level Structured Design:
- For $N = 1$ to $1^{st}$ input Value
  - DrawFigure ($2^{nd}$ input value)
  - Turn Right: $360/1^{st}$ input value
- EndFor
DrawFigure – procedure with 1 parameter “NumSides”
Global Variables – in Main

Begin

Input: balance

Input: interestRate

UpdateBalance

Output: "Your updated balance is: "

& formatcurrency(balance)§

End

UpdateBalance

balance = balance + balance*interestRate/100

End of UpdateBalance
Global Variables

- In Visual Logic, variables in main can be accessed by all other procedures.
- These are called global variables.
Local Variables – in Procedures

Begin

Input: balance

Input: interestRate

UpdateBalance

Output: "Your updated balance is: " & formatcurrency(balance)

$ 

Output: "Variable x in procedure? " & x

$ 

End

Please type a value for BALANCE: 100
Please type a value for INTERESTRATE: 5
x is 9999
Your updated balance is: $105.00
Variable x in procedure? 0

variable Watch

UpdateBalance

balance = balance + balance*interestRate/100

x = 9999

Output: "x is " & x

$ 

End of UpdateBalance
Local Variables

- The variables in a procedure are accessible ONLY in the procedure and cannot be accessed by main or other procedures.
- These are called **local** variables.
This is NOT updating the balances! We need parameters

```
Please type a value for CHECKACCTBALANCE:100
Please type a value for CHECKINGINTERESTRATE:2
Please type a value for SAVINGSACCTBALANCE:1000
Please type a value for SAVINGSINTERESTRATE:4
Your updated Checking Acct balance is: $100.00
Your updated Savings Acct balance is: $1,000.00
```

```
UpdateBalance

balance = balance + balance*interestRate/100

End of UpdateBalance
```

```
Begin

Input: checkAcctBalance

Input: checkinginterestRate

Input: savingsAcctBalance

Input: savingsInterestRate

UpdateBalance

Output: "Your updated Checking Acct balance is: " & formatCurrency(checkAcctBalance) $

Output: "Your updated Savings Acct balance is: " & formatCurrency(savingsAcctBalance) $

End
```
Parameters – passed by value

Balances did NOT update—what happened?
Parameters – passed by reference

ONLY CHANGE: Balance Passed by Reference

UpdateBalance(balance, interestRate)

balance = balance + balance*interestRate/100

End of UpdateBalance(balance, interestRate)

Balances updated correctly!
Parameters – Actual / Formal

- Procedures allow code to be written once and called many times.
- Procedures allow for code to be organized by logical function.
- **Actual parameters** are the parameters in the procedure call, usually in main, that are “passed” to the procedure.
- **Formal parameters** are the parameters defined in the procedure. They act like local variables in the procedure.
- When a procedure is called in main, the formal parameters are assigned values before the statements in the procedure are executed.
Parameters – Value and Reference

- If a formal parameter is a **value parameter**: then the value of the formal parameter in the procedure becomes the value of the corresponding actual parameter in the procedure call – **before the code** in the procedure is executed.

- If a formal parameter is a **reference parameter**: then the formal parameter is the exact SAME variable as the actual parameter. A formal reference parameter refers to the same memory location as the actual parameter (in the procedure call) passed.

- An actual parameter is a value or reference passed from the calling code (usually main) to a procedure.

- A formal parameter is the corresponding variable in the procedure that receives the value or reference.
Procedures calling Procedures!

Procedures can call other procedures
(see pages 118 – 120 in text book)

• List the names of all the procedures

• Which procedure is called from another procedure?
Procedures calling Procedures!

- Can a Procedures call ITSELF ??!!

- A **Recursive procedure** is one that calls itself.
Recursive Call Stack

Target: 180

LooseWeight (200)

LooseWeight (190)

LooseWeight (180)

Target: 155

LooseWeight (200)

LooseWeight (190)

LooseWeight (180)

LooseWeight (170)

LooseWeight (160)

LooseWeight (150)
Recursion is “repetition” in disguise

**RECURSIVE**

Procedure **LoseWeight** (*currWt*)

if (*currWt* > Target)
    then Output: “Losing Weight”
        *currWt* = *currWt* – 10
        **LoseWeight**(*currWt*)
    endif

**ITERATIVE**

Procedure LoseWeight (*currWt*)

while (*currWt* > Target)
    Output: “Losing Weight”
    *currWt* = *currWt* – 10
endwhile

Warning!
Must make sure the Recursion STOPS!!
Or will have an ___________________
BentLine

Begin

r
0 to 3

MoveTo: 100°r, 0

BentLine(200, r)

End

BentLine(Size, Count)

Count > 0

Forward: Size

False

BentLine(Size/3, Count - 1)

Turn Left: 60

BentLine(Size/3, Count - 1)

Turn Right: 120

BentLine(Size/3, Count - 1)

Turn Left: 60

BentLine(Size/3, Count - 1)

BentLine(Size/3, Count - 1)

BentLine(Size/3, Count - 1)

BentLine(Size/3, Count - 1)

End of BentLine(Size, Count)
BentLine

BentLine(200, 2)
BentLine(200/3, 1)
BentLine(200/9, 0) = Forward(200/9)

BentLine(Size, Count)
Count > 0

Forward: Size

BentLine(Size/3, Count - 1)
Turn Left: 60

BentLine(Size/3, Count - 1)
Turn Right: 120

BentLine(Size/3, Count - 1)
Turn Left: 60

End of BentLine(Size, Count)