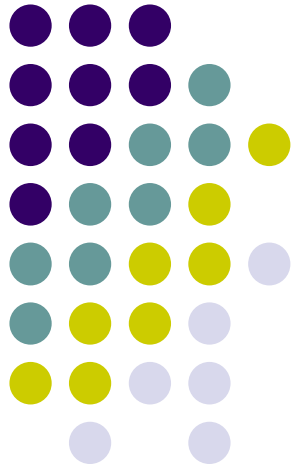
Program Design

Textbook:

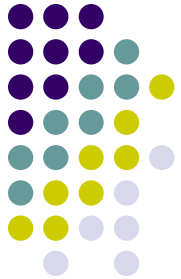
Simple Program Design:

A Step-by-Step Approach

by Lesley Anne Robertson

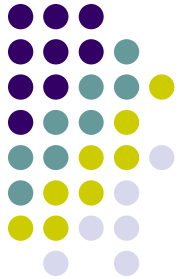


Program Design



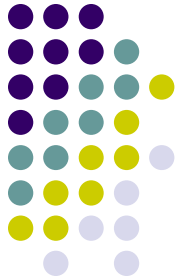
- Objectives:
  - Describe the steps in the program development process
  - Introduce program design methodology
  - Introduce procedural and object-oriented programming
  - Introduce algorithms
  - Describe program data

Steps in Program Development



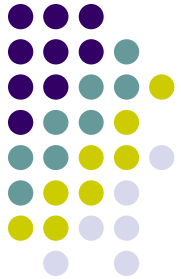
- Programming can be defined as the development of a solution to an identified problem, and the setting up of a related series of instructions that will produce the desired results.
  - 1. Define the problem
    - To help with initial analysis, divide the problem into three basic components:
      - Inputs
      - Outputs
      - The processing steps to produce the required outputs

2. Outline the Solution



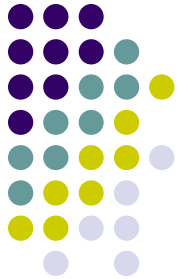
- This initial outline is usually a rough draft of the solution and may include:
  - The major processing steps involved
  - The major subtasks (if any)
  - The user interface (if any)
  - The major control structures (e.g. repetition loops)
  - The major variables and record structures
  - The mainline logic

3. Develop the Outline into an Algorithm



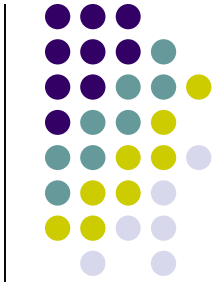
- The solution outline developed in Step 2 is expanded into an algorithm: a set of precise steps that describe exactly the tasks to be performed and the order in which they are to be carried out.

4. Test the Algorithm for Correctness



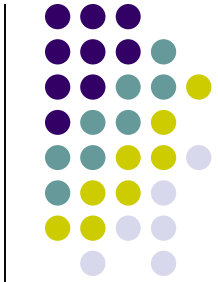
- This step is one of the most important in the development of a program, yet it is the step most often forgotten.
- The main purpose of “desk checking” the algorithm is to identify major logic errors early, so that they may be easily corrected.

5. Code the Algorithm into a Specific Programming Language



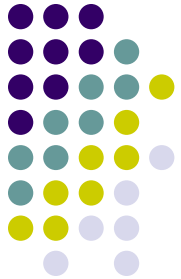
- Only after all design considerations have been met should you actually start to code the program into your chosen programming language.

6. Run the Program



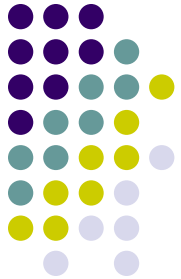
- This step uses the program, the language's development environment, and programmer-designed test data to test the code for syntax and logic errors.

7. Document and Maintain the Program



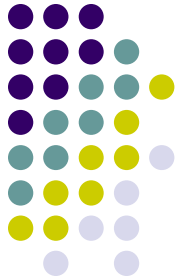
- Program documentation should not be listed as the last step in the program development process, as it is really an ongoing task from the initial definition of the problem to the final test result.
- Documentation involves both external documentation and internal documentation that may have been coded in the program.

Program Design Methodology



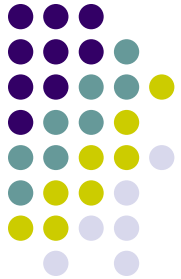
- There are several basic approaches to program design, including:
  - Procedure-driven
  - Event-driven
  - Data-driven

Procedure-Driven Program Design



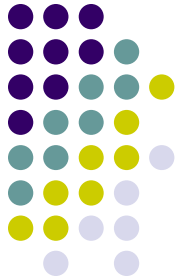
- The procedure-driven approach to program design is based on the idea that the most important feature of a program is “what” it does – that is, its processes or functions.

Event-Driven Program Design



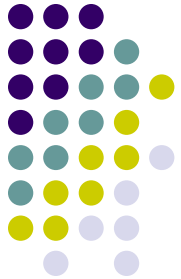
- The event-driven approach to program design is based on the idea that an event or interaction with the outside world can cause a program to change from one known state to another.
- The initial state of a program is identified, then all the triggers that represent valid events for that state are established.

Data-Driven Program Design



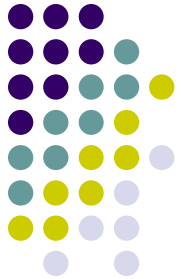
- The data-driven approach to program design is based on the idea that the data in a program is more stable than the processes involved.
- It begins with an analysis of the data and the relationships between the data, in order to determine the fundamental data structures.
- The choice between procedure-driven, event-driven, or data-driven program design methodologies is usually determined by the selection of a programming language.

Procedural Programming



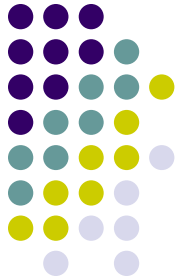
- Procedural programming is based on a structured, top-down approach to writing effective programs.
- The procedural approach concentrates on “what” a program has to do and involves identifying and organizing the “processes” in the program solution.

Top-Down Development



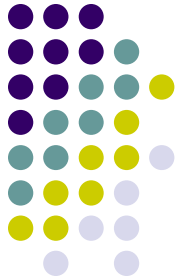
- In the top-down development of a program design, a general solution to the problem is outlined first.
- This is then broken down gradually into more detailed steps until, finally, the most detailed levels have been completed.

Modular Design



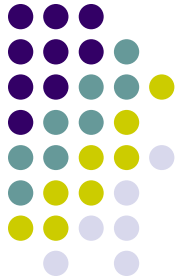
- Procedural programming also incorporates the concept of modular design, which involves grouping tasks together because they all perform the same function.
- Modular design is connected directly to top-down development.

Object-Oriented Programming



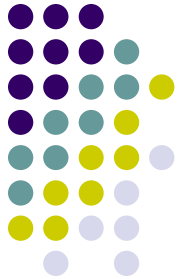
- Object-oriented programming is also based on decomposing the problem; the primary focus in on the “things” that make up the program.

Introduction to Algorithms and Pseudocode



- A program must be systematically and properly designed before coding begins.
- This design process results in the construction of an algorithm.

What is an Algorithm?



- An algorithm is like a recipe: it lists the stops involved in accomplishing a task.
- It can be defined in programming terms as a set of detailed, unambiguous and ordered instructions developed to describe the process necessary to produce the desired output from a given input.

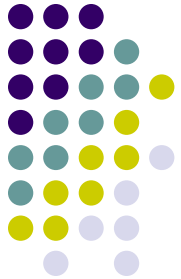
What is Pseudocode?

- Pseudocode is a way of representing algorithms.
- It's easy to read and write and allows the programmer to concentrate on the logic of the problem.
- Pseudocode, really, is structured English.
- It is English that has been formalized and abbreviated to look like high-level computer languages.

Program Data

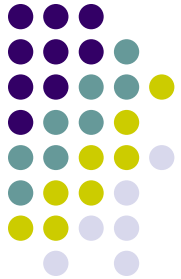
- Data within a program may be a single variable, such as an integer or a character, or a group item (sometimes called an aggregate), such as an array, a file, or a record.

Variables, Constants and Literals



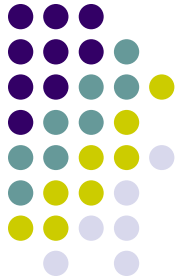
- A variable is a name given to a collection of memory cells, designed to store a particular data item.
- It is called a variable because the value stored in those memory cells may change or vary as the program executes.
- A constant is a data item with a name and a value that remain the same during the execution of the program.
- A literal is a constant whose name is the written representation of its value.

Data Types



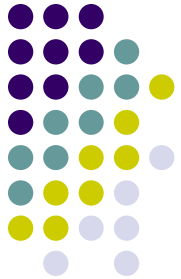
- At the beginning of a program, the programmer must clearly define the form or type of data to be collected.
- The data types can be elementary data items or data structures.

Data Types: Elementary Data Items



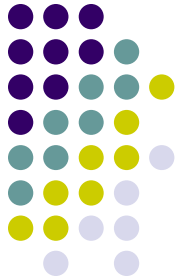
- An elementary data item is one containing a single variable that is always treated as a unit.
- A data type consists of a set of data values and a set of operations that can be performed on those values.
- The most common elementary data types are: Integer, Real (Float), Character, and Boolean

Data Types:
Data Structures



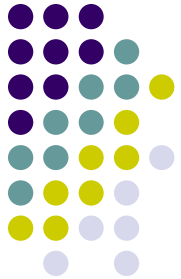
- A data structure is an aggregate of other data items.
- The data items that it contains are its components, which may be elementary data items or another data structure.
- The most common data structures are: Records, Files, Linked Lists, Arrays, Strings, and Trees.

Files



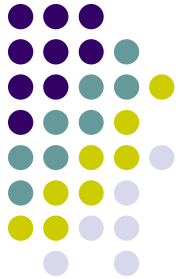
- The major advantages of using files are:
  - Several different programs can access the same data.
  - The data can be entered once and re-used many times.
  - The data can be easily updated and maintained.
  - The accuracy of the data is easier to enforce.
- There are two basic methods of storing/retrieving data in files:
  - Sequential/text files (open to read OR write).
  - Direct/random-access files (open to read AND write).

Data Validation



- Data should always undergo a validation check before it is processed by a program.
- Different types of data require different checks. For example:
  - Correct type
  - Correct length
  - Correct date
  - Correct range
  - Completeness

Summary

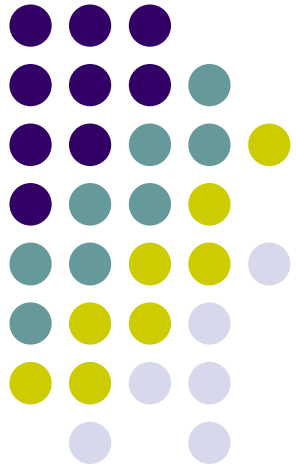


- The steps in program development were introduced and are briefly described below:
  - Define the problem
  - Outline the solution
  - Develop the outline into an algorithm
  - Test the algorithm for correctness
  - Code the algorithm into a specific programming language
  - Run the program on a computer
  - Document and maintain the program

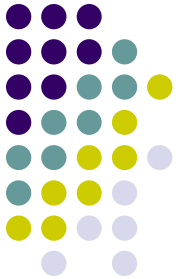
Summary (continued)

- Three different approaches to program design were introduced: procedure-driven, event-driven, and data-driven.
- An algorithm was defined as a set of detailed, unambiguous and ordered instructions developed to describe the processes necessary to produce the desired output from the given input.
- Pseudocode is an English-like way of representing the algorithm.

Pseudocode

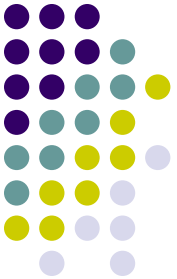


Objectives



- Introduce common words, keywords, and meaningful names when writing pseudocode.
- Define the three basic control structures as set out in the Structure Theorem.
- Illustrate the three basic control structures using pseudocode.

How to Write Pseudocode



- When designing a solution algorithm, you need to keep in mind that a computer will eventually perform the set of instructions you have written.
- If you use words and phrases in the pseudocode which are similar to basic computer operations, the translation from pseudocode/algorithm to a specific programming language becomes quite simple.

Six Basic Computer Operations

- 1. A computer can receive information.
  - When a computer is required to receive information or input from a particular source, whether it is a terminal, a disk or any other device, the verbs **Read** and **Get** are used in pseudocode.
- 2. A computer can put out information.
  - When a computer is required to supply information or output to a device, the verbs **Print**, **Write**, **Put**, **Output**, or **Display** are used in pseudocode.
  - Usually an output **Prompt** instruction is required before an input **Get** instruction.

Six Basic Computer Operations (continued)

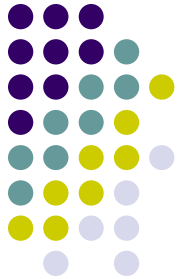
- 3. A computer can perform arithmetic.
  - Most programs require the computer to perform some sort of mathematical calculation, or formula, and for these a programmer may use either actual mathematical symbols or the words for those symbols.
  - When writing mathematical calculations for the computer, standard mathematical “order of operations” apply to pseudocode and most computer languages.
  - To be consistent with high-level programming languages, the following symbols can be written in pseudocode:

+ for add	- for subtraction
* for multiplication (asterisk)	/ for division

Six Basic Computer Operations (continued)

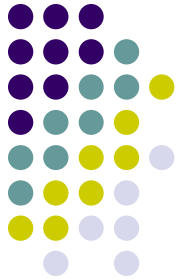
- 4. A computer can assign a value to a variable or memory location.
  - There are three cases where you may write pseudocode to assign a value to a variable or memory location:
    1. To give data an initial value in pseudocode, the verbs **Initialize** or **Set** are used.
    2. To assign a value as a result of some processing the symbols "=" or "←" are written.
    3. To keep a variable for later use, the verbs **Save** or **Store** are used.

Six Basic Computer Operations (continued)



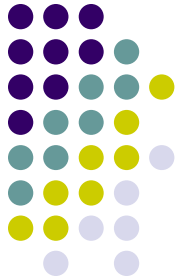
- 5. A computer can compare two variables and select one or two alternate actions.
  - An important computer operation available to the programmer is the ability to compare two variables (values) and then, as a result of the comparison, select one of two alternate actions.
  - To represent this operation in pseudocode, special keywords are used: e.g. **IF**, **THEN**, and **ELSE**.

Six Basic Computer Operations (continued)



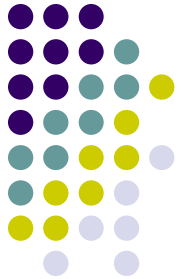
- 6. A computer can repeat a group of actions.
  - When there is a sequence of processing steps that need to be repeated, two special keywords, **DOWHILE** and **ENDDO**, are used in pseudocode.
  - The condition for the repetition of a group of actions is established in the **DOWHILE** clause, and the actions to be repeated are listed beneath it.

Meaningful Names



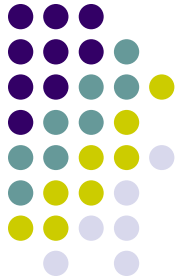
- All names should be meaningful.
- A name given to a variable is simply a method of identifying a particular storage location in the computer.
- The uniqueness of a name will differentiate it from other locations.
- Often a name describes the type of data stored in a particular variable.
- Most programming languages don’t tolerate a space in a variable name; a space would signal the end of the variable name and thus imply that there were two variables.

The Structure Theorem



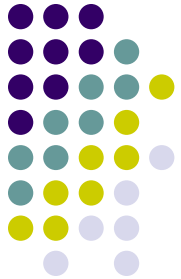
- The Structure Theorem states that it is possible to write any computer program by using only three basic control structures that are easily represented in pseudocode:
  - Sequence
  - Selection
  - Repetition

The Three Basic Control Structures



- 1. Sequence
  - The sequence control structure is the straightforward execution of one processing step after another.
  - In pseudocode, we represent this construct as a sequence of pseudocode statements.
- 2. Selection
  - The selection control structure is the presentation of a condition and the choice between two actions; the choice depends on whether the condition is true or false.
  - In pseudocode, selection is represented by the keywords **IF**, **THEN**, **ELSE**, and **ENDIF**.

The Three Basic Control Structures (cont.)

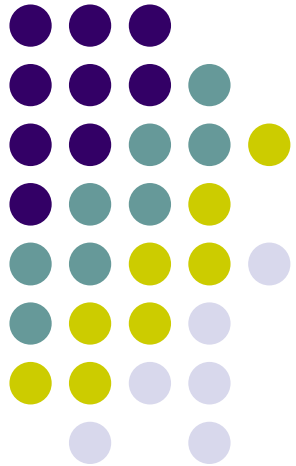


- 3. Repetition
  - The repetition control structure can be defined as the presentation of a set of instructions to be performed repeatedly, as long as a condition is true.
  - The basic idea of repetitive code is that a block of statements is executed again and again, until a terminating condition occurs.
  - This construct represents the sixth basic computer operation, namely to repeat a group of actions.

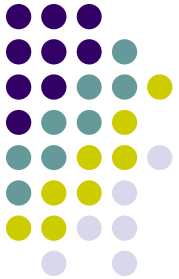
Summary: Pseudocode

- Six basic computer operations were listed, along with pseudocode words and keywords to represent them.
- These operations were: to receive information (input) to send information (output), perform arithmetic, assign a value to a variable, decide between two alternate actions, and repeat a group of actions.
- The Structure Theorem was introduced. It states that it is possible to write any computer program by using only three basic control structures: sequence, selection, and repetition.

Developing an Algorithm



Objectives

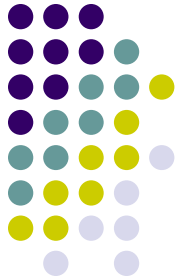


- Introduce methods of analyzing a problem and developing a solution.
- Develop simple algorithms using the sequence control structure.
- Introduce methods of manually checking the developed solution.

Defining the Problem

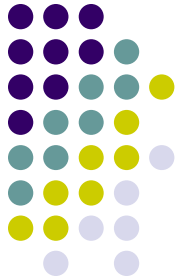
- To help with the initial analysis, the problem should be divided into three separate components:
  - Input: A list of the source data provided to the problem.
  - Output: A list of the outputs required.
  - Processing: A list of actions needed to produce the required outputs.
- When dividing a problem into its three different components, you should simply analyze the actual words used in the specification, and divide them into those that are descriptive and those that imply actions.

Example: Add Three Numbers



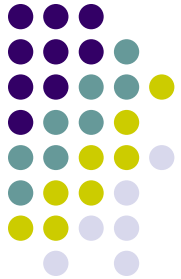
- A program is required to read three numbers, add them together, and print their total.
  - Tackle this problem in two stages:
    - Underline the nouns and adjectives used in the specification.
      - This will establish the input and output components, as well as any objects that are required.
      - Example: A program is required to read three numbers, add them together, and print their total.
    - Underline (in a different color) the verbs and adverbs used in the specification.
      - Example: A program is required to read three numbers, add them together, and print their total.

Example: Add Three Numbers (cont.)



- Example: A program is required to read three numbers, add them together, and print their total.
  - You can see that the processing verbs are “read,” “add together,” and “print.”
  - These steps can now be added to our defining diagram to make it complete.
  - When it comes to writing down the processing steps in an algorithm, you should use words that describe the work to be done in terms of single, specific tasks or functions.
  - There is a pattern in the words chosen to describe these steps.
  - Each action is described as a single verb followed by a two-word object.

Example:
Find Average Temperature



- A program is required to prompt the terminal operator for the maximum and minimum temperature readings on a particular day, accept those readings as integers, and calculate and display to the screen the average temperature, calculated as (maximum temperature + minimum temperature) / 2.

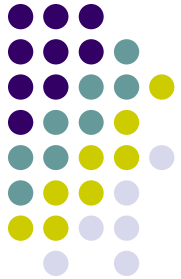
Example: Find Average Temperature (c)

- First, establish the input and output components by underlining the nouns and adjectives in the problem statement:
  - A program is required to prompt the terminal operator for the maximum and minimum temperature readings on a particular day, accept those readings as integers, and calculate and display to the screen the average temperature, calculated as (maximum temperature + minimum temperature) / 2.

Example:
Find Average Temperature (c)

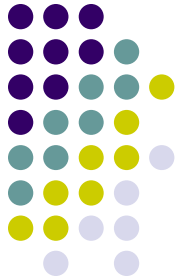
- Second, establish the processing steps by underlining the verbs in the problem statement:
  - A program is required to prompt the terminal operator for the maximum and minimum temperature readings on a particular day, accept those readings as integers, and calculate and display to the screen the average temperature, calculated as (maximum temperature + minimum temperature) / 2.

Example:
Find Average Temperature (c)



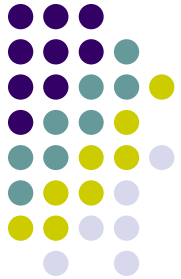
- Finding the associated objects of the verbs allows the completion of the defining diagram.

Example: Compute Mowing Time



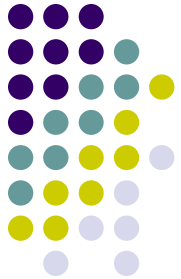
- A program is required to read from the screen the length and width of a rectangular house block, and the length and width of the rectangular house that is built on the block. The algorithm should then compute and display the mowing time to cut the grass around the house, at the rate of two square meters per minute.

Designing a Solution Algorithm



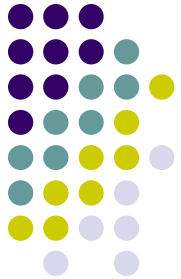
- Designing a solution algorithm is the most challenging task in the life cycle of a program.
- Once the problem has been properly defined, you usually begin with a rough sketch of the steps required to solve the problem.
- The first attempt at designing a particular algorithm usually does not result in a finished product.
- Pseudocode is useful in this trial-and-error process, since it is relatively easy to add, delete, or alter an instruction.

Checking the Solution Algorithm



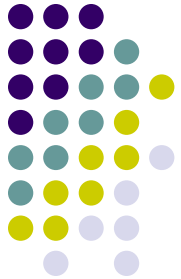
- After a solution algorithm has been established, it must be tested for correctness.
- This step is necessary because most major logic errors occur during the development of the algorithm, and if not detected these errors can be passed on to the program.
- Desk checking involves tracing through the logic of the algorithm with some chosen test data.

Selecting Test Data



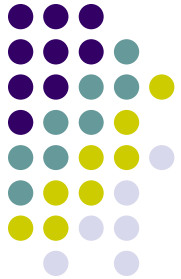
- When selecting test data to desk check an algorithm, you must look at the program specification and choose simple test cases only, based on the requirements of the specification, not the algorithm.
- By doing this, you will still be able to concentrate on *what* the program is supposed to do, not *how*.

Steps in Desk Checking an Algorithm



- There are six simple steps to follow when desk checking an algorithm:
  - 1. Choose simple input test cases that are valid. Two or three test cases are usually sufficient.
  - 2. Establish what the expected result should be for each test case. This is one of the reasons for choosing simple test data in the first place: it is much easier to determine the total of 10, 20 and 30 than 3.75, 2.89 and 5.31!
  - 3. Make a table on a piece of paper of the relevant variable names within the algorithm.
  - 4. Walk the first test case through the algorithm, line by line, keeping a step-by-step record of the contents of each variable in the table as the data passes through the logic.
  - 5. Repeat the walk-through process using the other test data cases, until the algorithm has reached its logical end.
  - 6. Check that the expected result established in Step 2 matches the actual result developed in Step 5.

Steps in Desk Checking an Algorithm (continued)

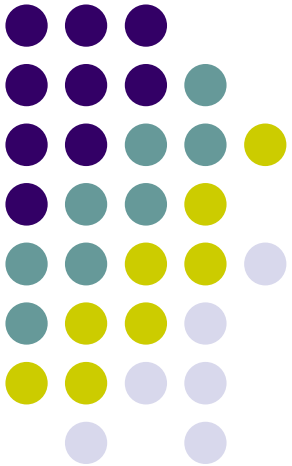


- By desk checking an algorithm, you are attempting to detect errors early.
- It is a good idea for someone other than the author of the solution algorithm to design the test data for the program, because they won't be influenced by the program logic.

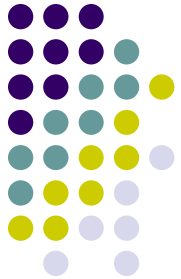
Summary: Developing an Algorithm

- We learned methods of analyzing and defining a programming problem.
  - You must fully understand a problem before you can attempt to find a solution.
  - The method suggested was to analyze the actual words used in the specification with the aim of dividing the problem into three separate components: input, output, and processing.
- We discussed the establishment of a solution algorithm.
  - After the initial analysis of the problem, you must attempt to find a solution and express the solution as an algorithm.
- Finally, we learned how to check the algorithm for correctness.

Selection Control Structures

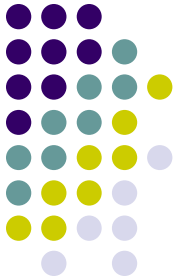


Objectives



- Eleborate on the uses of simple selection, multiple selection, and nested selection in algorithms.
- Introduce the case construct in pseudocode.
- Develop algorithms using variations of the selection control structure.

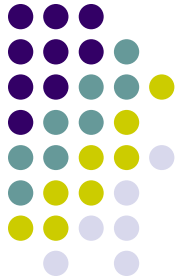
The Selection Control Structure



- You can use the selection control structure in pseudocode to illustrate a choice between two or more actions, depending on whether a condition is true or false.
- The condition in the **IF** statement is based on a comparison of two items, and is usually expressed with one of the following relational operators:

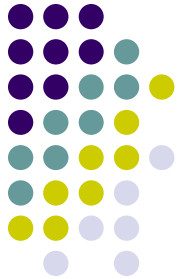
<	less than	>	greater than
=	equal to	<=	less than or equal to
>=	greater than or equal to	<>	not equal to

1. Simple Selection (Simple IF Statement)



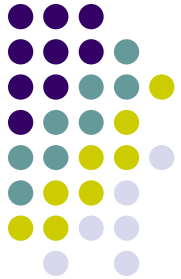
- Simple selection occurs when a choice is made between two alternate paths, depending on the result of a condition being true or false.
- The structure is represented in pseudocode using the keywords **IF**, **THEN**, **ELSE**, and **ENDIF**.
- Only one of the **THEN** or **ELSE** paths will be followed, depending on the result of the condition in the **IF** clause.

2. Simple Selection with Null False Branch (Null ELSE Stmt)



- The null **ELSE** structure is a variation of the simple **IF** structure.
- It is used when a task is performed only when a particular condition is true.
- If the condition is false, then no processing will take place and the **IF** statement will be bypassed.

3. Combined Selection (Combined IF Statement)



- A combined **IF** statement is one that contains multiple conditions, each connected with the logical operators **AND** or **OR**.
- If the connector **AND** is used to combine the conditions, then *both* conditions must be true for the combined condition to be true.
- If the connector **OR** is used to combine any two conditions, then only *one* of the conditions need to be true for the combgined condition to be considered true.

The NOT Operator

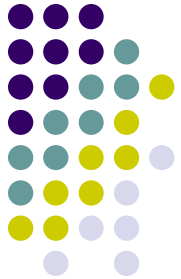
- The **NOT** operator can be used for the logical negation of a condition, as follows:

```
IF NOT (record_code = “23”) THEN
        update customer record
ENDIF
```

- Note that the **AND** and **OR** operators can also be used with the **NOT** operator, but great care must be taken and parentheses should be used to avoid ambiguity. For example:

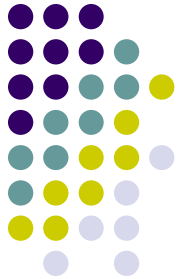
```
IF NOT (record_code = “23” AND update_code = delete) THEN
        update customer record
ENDIF
```

4. Nested Selection (Nested IF Statement)



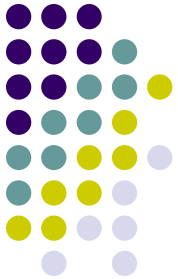
- Nested selection occurs when the word IF appears more than once within an IF statement.
- Nested IF statements can be classified as linear or non-linear.

4. Nested Selection (cont.)
Linear Nested IF Statements



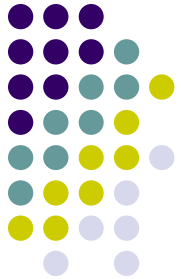
- The **linear nested IF** statement is used when a field is being tested for various values and a different action is to be taken for each value.
- This form of nested **IF** is called linear because each **ELSE** immediately follows the IF condition to which it corresponds.

4. Nested Selection (cont.)
Linear Nested IF Statements (cont.)



```
IF record_code = “A” THEN
        increment counter_A
ELSE
        IF record_code = “B” THEN
                increment counter_B
        ELSE
                IF record_code = “C” THEN
                        increment counter_C
                ELSE
                        increment error_counter
                END IF
        ENDIF
ENDIF
```

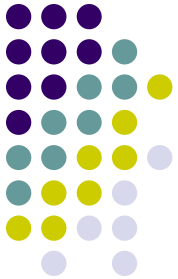
4. Nested Selection (cont.)
Non–Linear Nested IF Statements



- A **non-linear** nested **IF** occurs when a number of different conditions need to be satisfied before a particular action can occur.
- It is termed **non-linear** because the **ELSE** statement may be separated from the **IF** statement with which it is paired.

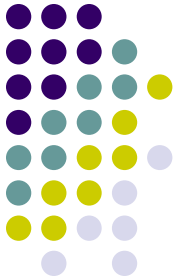
4. Nested Selection (cont.)

Non–Linear Nested IF Statements (cont.)



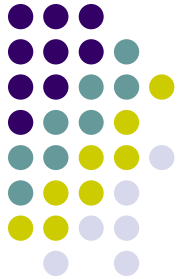
```
IF student_attendance = part_time THEN
        IF student_gender = female THEN
                IF student_age > 21 THEN
                        add 1 to mature-female_part_time_students
                ELSE
                        add 1 to young_female_part_time_students
                ENDIF
        ELSE
                add 1 to male_part_time_students
        ENDIF
ELSE
        add 1 to full_time_students
ENDIF
```

Algorithms Using Selection



- In each of the following examples:
  - The problem will be defined
  - A solution algorithm will be developed
  - The algorithm will be manually tested.

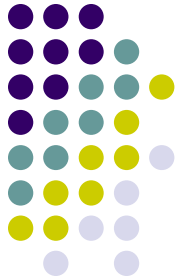
Example 1: Read Three Characters



- Design an algorithm that will prompt a terminal operator for three characters, accept those characters as input, sort them into ascending sequence, and output them to the screen.
- A: The Defining Diagram

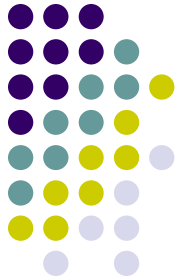
Input	Processing	Output
Char_1	Prompt for characters	Char_1
Char_2	Sort three characters	Char_2
Char_3	Sort three characters	Char_3
	Output three characters	

Example 1: Read Three Characters (cont.)



- B: The Solution Algorithm
  - The solution algorithm requires a series of IF statements to sort the three characters into ascending sequence.
  - In this solution, most of the logic of the algorithm is concerned with the sorting of the three characters into alphabetic sequence.
  - See the following slide for the example…

Example 1: Read Three Characters (cont.)



```
Prompt the operator for char_1, char_2, char_3
Get char_1, char_2, char_3
IF char_1 > char_2 THEN
        temp = char_1
        char_1 = char_2
        char_2 = temp
ENDIF
IF char_2 > char_3 THEN
        temp = char_2
        char_2 = char_3
        char_3 = temp
ENDIF
IF char_1 > char_2 THEN
        temp = char_1
        char_1 = char_2
        char_2 = temp
ENDIF
Output char_1, char_2, char_3 to the screen
```

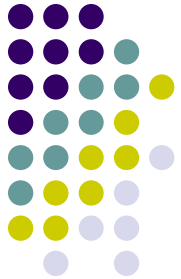
Example 1: Read Three Characters (cont.)

- C: Desk Checking
  - Two sets of valid characters will be used to check the algorithm:

INPUT DATA	First Data Set	Second Data Set
char_1	k	z
char_2	b	s
char_3	g	a

EXPECTED RESULTS	First Data Set	Second Data Set
char_1	b	a
char_2	g	s
char_3	k	z

Example 1: Read Three Characters (cont.)

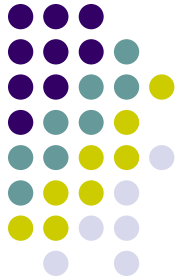


- C: Desk Checking (continued)
  - Examine the input data, expected results, and desk check table illustration as shown on the following slide…
  - Line numbers have been used to identify each statement within the program
  - Note that when desk checking the logic, each IF statement is treated as a single statement.

Example 1: Read Three Characters (cont.)

Statement #	char_1	char_2	char_3	temp
1st pass				
1, 2	k	b	g	
3	b	k		k
4		g	k	k
5				
6	output	output	output	
2nd Pass				
1, 2	z	s	a	
3	s	z		z
4		a	z	z
5	a	s		s
6	output	output	output	

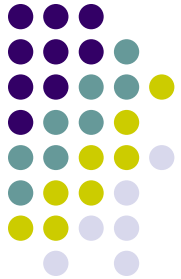
Example 2: Process Customer Record



- A program is required to read a customer's name, a purchase amount, and a tax code. The tax code is an integer 0–4 (exempt, sales tax, federal, and state, respectively). The program must then compute the tax and the total amount due, and print the customer's name, purchase amount, tax, and total amount due.
- A: Defining Diagram

Input	**Processing**	**Output**
cust_name	Read customer details	cust_name
purch_amt	Compute tax	purch_amt
tax_code	Compute total amount	Tax
	Print customer details	total_amt

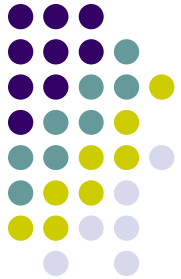
Example 2: Process Customer Record (c)



- B. Solution Algorithm
  - The solution algorithm requires a linear nested IF statement to calculate the tax:

```
Read cust_name, purch_amt, tax_code
IF tax_code = 0 THEN
        tax = 0
ELSE
        IF tax_code = 1 THEN
                tax = purch_amt * 0.03
        ELSE
                IF tax_code = 2 THEN
                        tax = purch_amt * 0.05
                ELSE
                        tax = purch_amt * 0.07
                ENDIF
        ENDIF
ENDIF
Total_amt = purch_amt + tax
Print cust_name, purch_amt, tax, total_amt
```

Example 2: Process Customer Record (c)



- C: Desk Checking
  - Two sets of valid input data for purchase amount and tax code will be used to check the algorithm.
  - Examine the input data, expected results, and desk check tables shown on the following slides.
  - As the expected result for the two test cases matches the calculated result, the algorithm is correct.

Example 2: Process Customer Record (c)

- Input Data

	First Data Set	Second Data Set
purch_amt	$10.00	$20.00
tax_code	0	2

- Expected Results

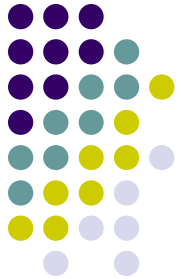
	First Data Set	Second Data Set
tax	$0.00	$1.00
total_amt	$10.00	$21.00

Example 2: Process Customer Record (c)

- Desk Check Table

Statement #	purch_amt	tax_code	tax	total_amt
1st Pass				
1	$10.00	0		
2			$0.00	
3				$10.00
4	print		print	Print
2nd Pass				
1	$20.00	2		
2			$1.00	
3				$21.00
4	print		print	print

The Case Structure



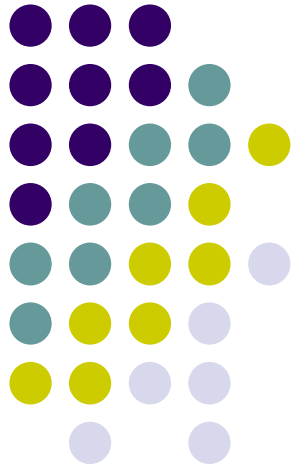
- The Case control structure in pseudocode is another way of expressing a linear nested IF statement. For example, we could represent Example 2's pseudocode much more simply (and clearly) with a Case structure:

```
CASE OF tax_code
        "0":    tax = 0
        "1":    tax = purch_amt * 0.03
        "2":    tax = purch_amt * 0.05
        "3":    tax = purch_amt * 0.07
ENDCASE
Total_amt = purch_amt + tax
Print cust_name, purch_amt, tax, total_amt
```

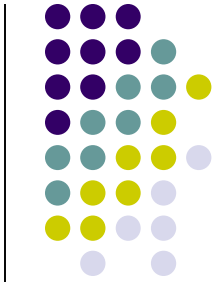
Summary: Selection Control Structures

- The selection control structure was described in detail.
- Descriptions and pseudocode examples were given for simple selections, combined IF and nested IF statements.
- The case structure was introduced as a means of expressing a linear nested IF statement in a simpler and more concise form.

Repetition Control Structures

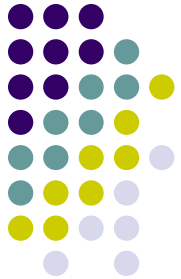


Objectives



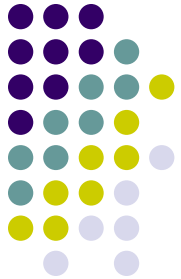
- Develop algorithms that use the DOWHILE and REPEAT…UNTIL control structures.
- Introduce a pseudocode structure for counted repetition loops.
- Develop algorithms using variations of the repetition construct.

Repetition using the DOWHILE Structure



- There are three different ways that a set of instructions can be repeated, and each way is determined by where the decision to repeat is placed:
  - At the beginning of the loop (leading decision loop)
  - At the end of the loop (trailing decision loop)
  - A counted number of times (counted loop)

Leading Decision Loop



- The DOWHILE construct is a leading decision loop – that is, the condition is tested before any statements are executed.
- When designing a DOWHILE loop, be aware:
  - The testing of the condition is at the beginning (top) of the loop.
  - The only way to terminate the loop is to render the DOWHILE condition false.

Using DOWHILE to Repeat a Set of Instructions a Known Number of Times

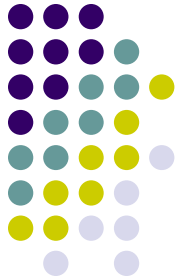
- When a set of instructions is repeated a specific number of times, a counter can be used in pseudocode, which is initialized before the DOWHILE statement and incremented just before the ENDDO statement.

Example 1: Fahrenheit/Celsius Conversion

- Daily, a weather station receive 15 temperatures expressed in degrees Fahrenheit. A program is to be written that will accept each Fahrenheit temperature, convert it to Celsius and display the converted temperatures to the screen. After 15 temperatures have been processed, the words "All temperatures processed." are to be displayed on the screen.
- A. Defining Diagram

	First Data Set	Second Data Set
f_temp (input)	32	50
c_temp (expected results)	0	10

Example 1: Fahrenheit/Celsius Conversion (continued)



- B. Solution Algorithm

```
1       Set temperature_count to zero
2       DOWHILE temperature_count < 15
3               Prompt operator for f_temp
4               Get f_temp
5               Compute c_temp = (f_temp – 32) * 5/9
6               Display c_temp
7               Add 1 to temperature_count
        ENDDO
8       Display “All temperatures processed.”
```

Example 1: Fahrenheit/Celsius Conversion (continued)

- C. Desk Checking
  - Although the program will require 15 records to process properly, it is still only necessary to check the algorithms, at this stage, with two valid sets of data.

Example 1: Fahrenheit/Celsius Conversion (continued)

- C. Desk Checking

Program Line	temperature_count	DOWHILE condition	f_temp	c_temp
1	0			
2		True		
3, 4			32	
5				0
6				Display
7	1			
2		true		
3, 4			50	
5				10
6				display
7	2			

Using DOWHILE to Repeat a Set of Instructions an Unknown Number of Times

- 1. When a trailer record or sentinel exists.
  - When there are an unknown number of items to process you cannot use a counter, so another way of controlling the repetition must be used.
  - Often, a trailer record or sentinel signifies the end of the data.
  - This sentinel is a special record or value placed at the end of valid data to signify the end of that data.
  - It must contain a value that is clearly distinguishable from the other data to be processed.

Example 2:
Print Examination Scores

- A program is required to read and print a series of names and exam scores for students enrolled in a mathematics course. The class average is to be computed and printed at the end of the report. Scores can range from 0 to 100. The last record contains a blank name and a score of 999 and is not to be included in the calculations.
- A. Defining Diagram

Input	**Processing**	**Output**
name	Read student details	name
exam_score	Print student details	exam_score
	Compute average score	average_score
	Print average score	

Example 2: Print Examination Scores (continued)

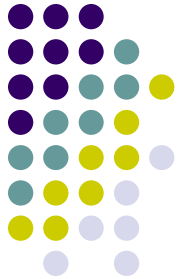
- You will need to consider the following when establishing a solution algorithm:
  - A DOWHILE structure to control the reading of exam scores, until it reaches a score of 999.
  - An accumulator for total scores, namely total_score.
  - An accumulator for the total students, namely total_students.

Example 2: Print Examination Scores (continued)

- B. Solution Algorithm

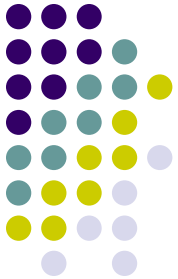
```
1       Set total_score to zero
2       Set total_students to zero
3       Read name, exam_score
4       DOWHILE exam_score not = 999
5               Add 1 to total_students
6               Print name, exam_score
7               Add exam_score to total_score
8               Read name, exam_score
        ENDDO
9       IF total_students not = zero THEN
                average_score = total_score / total_students
                Print average_score
        ENDIF
```

Example 2: Print Examination Scores (continued)



- In this example, the DOWHILE condition tests for the existence of the trailer record or sentinel (record 999).
- However, this condition cannot be tested until at least one exam mark has been read.
  - The initial processing that sets up the condition is a Read statement immediately before the DOWHILE clause.
  - This is known as a priming read.

Example 2: Print Examination Scores (continued)

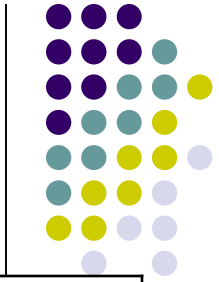


- C. Desk Checking
- Two valid records and a trailer record should be sufficient to desk check this algorithm.

INPUT DATA	First Record	Second Record	Third Record
score	50	100	999

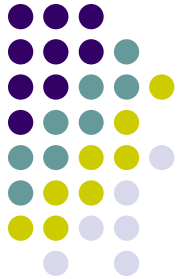
- Expected Results:
  - First name and score of 50
  - Second name and score of 100
  - Average score of 75

Example 2: Print Examination Scores (continued)



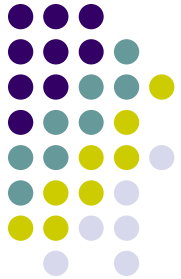
Program Line	total_score	total_students	exam_score	DOWHILE condition	average_score
1, 2	0	0			
3			50		
4				True	
5		1			
6			Print		
7	50				
8			100		
4				True	
5		2			
6			Print		
7	150				
8			999		
4				False	
9					75
					print 100

2. When a Trailer Record or Sentinel Does Not Exist



- When there is no trailer record or sentinel to signify the end of the data, the programmer needs to check for an end-of-file marker (EOF).
- This EOF marker is added when the file is created, as the last character in the file.
- The check for EOF is positioned in the DOWHILE clause, using one of the following equivalent expressions:
  - DOWHILE more data
  - DOWHILE more records
  - DOWHILE records exist
  - DOWHILE NOT EOF

Example 3: Process Student Enrollments



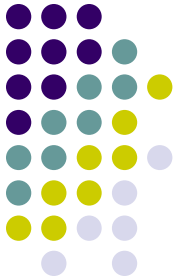
- A program is required that will read a file of student records, and select and print only those students enrolled in a course unit named Programming 1. Each student record contains student number, name, address, postcode, gender, and course unit number. The course unit number for Programming 1 is 18500.
- You will need to consider the following requirements when establishing a solution algorithm:
  - A DOWHILE structure to perform the repetition
  - IF statements to select the required students
  - Accumulators for the three total fields

Example 3: Process Student Enrollments (continued)

- A. Defining Diagram

Input	Processing	Output
student_record	Read student records	Selected student records
• student_no	Select student records	total_females_enrolled
• name	Print selected records	total_males_enrolled
• address	Compute total females enrolled	total_students_enrolled
• postcode	Compute total males enrolled	
• gender	Compute total students enrolled	
• course_unit	Print totals	

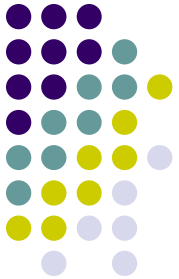
Example 3: Process Student Enrollments (continued)



- B. Solution Algorithm

```
1       Set total_females_enrolled to zero
2       Set total_males_enrolled to zero
3       Set total_students_enrolled to zero
4       Read student record
5       DOWHILE records exist
6               IF course_unit = 18500 THEN
                        print student details
                        increment total_students_enrolled
                        IF student_gender = female THEN
                                increment total_females_enrolled
                        ELSE
                                increment total_males_enrolled
                        ENDIF
                ENDIF
7               Read student record
        ENDDO
8       Print total_females_enrolled
9       Print total_males_enrolled
10      Print total_students_enrolled
```

Example 3: Process Student Enrollments (continued)

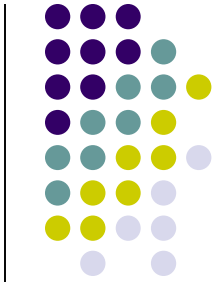


- C. Desk Checking
  - Three valid student records should be sufficient to desk check this algorithm.
  - Since student_no, name, address, and postcode are not operated upon in this algorithm, they do not need to be provided as input test data.

INPUT DATA	1st record	2nd record	3rd record
course_unit	2000	18500	18500
gender	F	F	M

  -
    - Student number, name, address, postcode, F (2nd student)
    - Student number, name, address, postcode, M (3rd student)
    - Total females enrolled: 1; Total males enrolled: 1; Total students enrolled: 2.

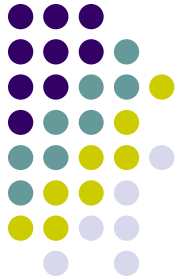
Example 3: Process Student Enrollments (continued)



- C. Desk Checking

Program Line	course_unit	gender	DOWHILE condition	total_females_enrolled	total_males_enrolled	total_students_enrolled
1, 2, 3				0	0	0
4	2000	F				
5			true			
6						
7	18500	F				
5			true			
6	print	print		1		1
7	18500	M				
5			true			
6	print	print			1	2
7	EOF					
5			false			
8, 9, 10				print	print	print

Repetition Using the REPEAT…UNTIL Structure – Trailing Decision Loop

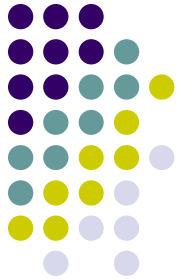


- The REPEAT…UNTIL structure is similar to the DOWHILE structure, in that a group of statements are repeated in accordance with a specified condition.
- However, where the DOWHILE structure tests the condition at the *beginning* of the loop, a REPEAT…UNTIL structure tests the condition at the *end* of the loop.
- REPEAT…UNTIL is a trailing decision loop; the statements are executed once before the condition is tested.

Trailing Decision Loop (cont.)

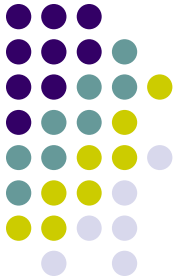
- REPEAT…UNTIL loops are executed when the condition is false.
- The statements within a REPEAT…UNTIL structure will always be executed as least once.

Example 4: Process Inventory Items



- A program is required to read a series of inventory records that contain item number, item description, and stock figure. The last record in the file has an item number of zero. The program is to produce a low stock items report, by printing only those records that have a stock figure of less than 20 items. A heading is to be printed at the top of the report and a total low stock item count printed at the end.

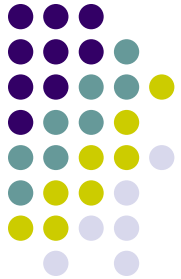
Example 4: Process Inventory Items (continued)



- A. Defining Diagram

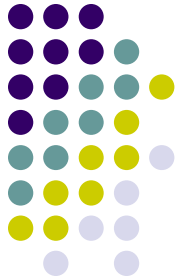
Input	Processing	Output
Inventory record	Read inventory records	Heading
• item_number	Select low stock items	Selected records
• item_description	Print low stock records	• item_number
• stock_figure	Print total low stock items	• item_description
		• stock_figure
		total_low_stock_items

Example 4: Process Inventory Items (continued)



- You will need to consider the following requirements when establishing a solution algorithm:
  - A REPEAT…UNTIL to perform the repetition
  - An IF statement to select stock figures of less than 20
  - An accumulator for total_low_stock_items
  - An extra IF, within the REPEAT loop, to ensure the trailer record is not processed

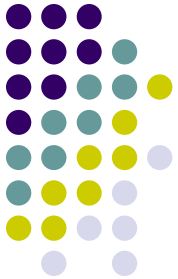
Example 4: Process Inventory Items (continued)



- B. Solution algorithm

```
1       Set total_low_stock_items to zero
2       Print “Low Stock Items” heading
        REPEAT
3           Read inventory record
4           I   F item_number > zero THEN
                    IF stock_figure < 20 THEN
                        print item_number, item_description, stock_figure
                        increment total_low_stock_items
                    ENDIF
                ENDIF
5       UNTIL item_number = zero
6       Print total_low_stock_items
```

Example 4: Process Inventory Items (continued)



- C. Desk Checking
  - Two valid records and a trailer record (item number equal to zero) will be used to test the algorithm; then examine input data, expected results, and desk check tables.

Input Data	1st Record	2nd Record	3rd Record
item_number	123	124	0
stock_figure	8	35	

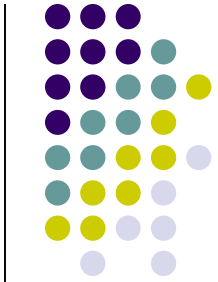
  - Expected results

    Low Stock Items

    123 8 (first record)

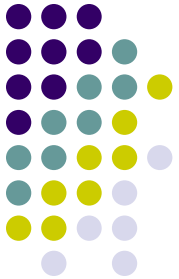
    Total Low Stock Items = 1

Example 4: Process Inventory Items (continued)



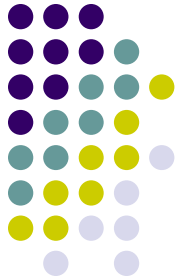
Program Line	item_number	stock_figure	REPEAT UNTIL	total_low_stock_items	Heading
1				0	
2					print
3	123	8			
4	print	print		1	
5			false		
3	124	25			
4					
5			false		
3	0				
4					
5			true		
6				print	

Counted Loop



- Counted repetition occurs when the exact number of loop iterations is known in advance.
- The execution of the loop is controlled by a loop index, and instead of using DOWHILE or REPEAT…UNTIL, the simple keyword DO is used.
- Revisiting Example 1, the Fahrenheit/Celsius conversion problem, note that we can re-write it using a DO loop.

Counted Loop (continued)



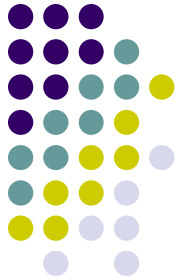
- The DO loop does more than just repeat the statement block:
  - Initialize the loop_index to the required initial_value.
  - Increment the loop_index by 1 for each pass through the loop.
  - Test the value of loop_index at the beginning of each loop to ensure that it is within the stated range of values.
  - Terminate the loop when the loop_index has exceeded the specified final_value.

Example 5: Fahrenheit/Celsius Conversion

- Review the problem statement from Example 1 (slide #90).
- A. Defining Diagram

Input	**Processing**	**Output**
f_temp	Get Fahrenheit temperatures	c_temp
(15 temperatures)	Convert temperatures	(15 temperatures)
	Display Celsius temperatures	
	Display screen message	

Example 5: Fahrenheit/Celsius Conversion (continued)



- B. Solution Algorithm

```
1       DO temperature_count = 1 to 15
2               Prompt operator for f_temp
3               Get f_temp
4               Compute c_temp = (f_temp – 32) * 5/9
5               Display c_temp
        ENDDO
6       Display “All temperatures processed.” to the screen
```

Example 5: Fahrenheit/Celsius Conversion (continued)

- Note that the DO loop controls all the repetition:
  - It initializes temperature_count to 1
  - It increments temperature_count by 1 for each pass through the loop.
  - It tests temperature_count at the beginning of each pass to ensure that it is within the range 1 to 15.
  - It automatically terminates the loop once temperature_count has exceeded 15.
- C. Desk Checking
  - Two valid records should be sufficient to test the algorithm for correctness; it isn't necessary to check the DO loop construct for all 15 records.

Example 5: Fahrenheit/Celsius Conversion (continued)

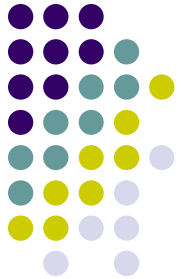
INPUT DATA	1st Data Set	2nd Data Set
f_temp	32	50
EXPECTED RESULTS	**1st Data Set**	**2nd Data Set**
c_temp	0	10

Example 5: Fahrenheit/Celsius Conversion (continued)

- Desk Check Table

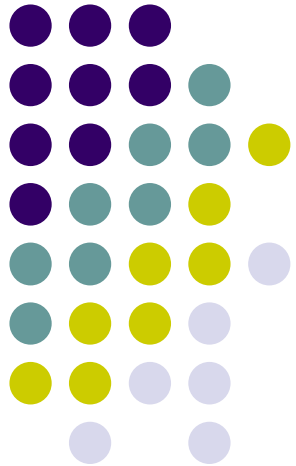
Program Line	temperature_count	f_temp	c_temp
1	1		
2, 3		32	
4			0
5			display
1	2		
2, 3		50	
4			10
5			display

Summary

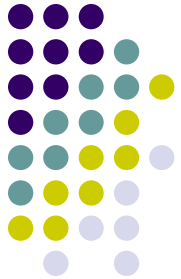


- We covered the repetition control structure in detail.
- Descriptions and pseudocode examples were given for leading decision loops (DOWHILE), trailing decision loops (REPEAT…UNTIL) and counted loops (DO).
- We saw that most of the solution algorithms that used the DOWHILE structure had the same general pattern.

First Steps in Modularization

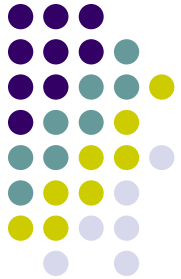


Objectives



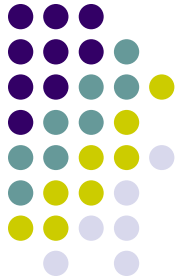
- Introduce modularization as a means of dividing a problem into subtasks.
- Present hierarchy charts as a pictorial representation of modular program structure.
- Discuss intermodule communication, local and global variables, and the passing of parameters between modules.
- Develop programming examples that use a simple modularized structure.

Modularization



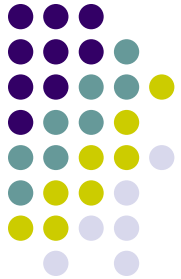
- Modularization is the processes of dividing a problem into separate tasks, each with a single purpose.
- Top-down design methodology allows the programmer to concentrate on the overall design of the algorithm without getting too involved with the details of the lower-level modules until it’s necessary (later).

The Modularization Process



- The division of a problem into smaller subtasks, or modules, is a relatively simple process.
- When you are defining the problem, write down the activities or processing steps to be performed.
- A module must be large enough to perform its task, and must include only the operations that contribute to the performance of that task.

The Modularization Process (continued)

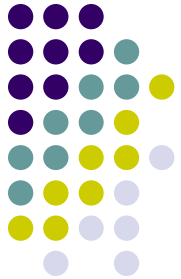


- A module should have a single entry and a single exit with a top-to-bottom sequence of instructions.
- The name of the module should describe the work to be done as a single specific function.
- The convention of naming a module by using a verb, followed by a two-word object, is particularly important here, as it helps to identify the separate task or function that the module is to perform.

The Mainline

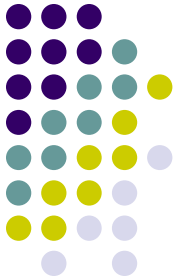
- A mainline routine must provide the master control that ties all the modules together and coordinates their activity.
- This program mainline should show the main processing functions, and the order in which they are to be performed.
- It should also show the flow of data and the major control structures.
- The mainline should be easy to read, be of manageable length, and show sound logic structure.

Benefits of Modular Design



- There are a number of benefits from using modular design:
  - Ease of understanding
  - Reusable code
  - Elimination of redundancy
  - Efficiency
  - Easier debugging
  - Easier maintenance

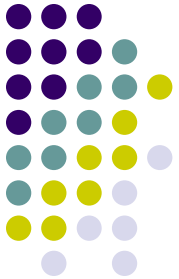
Example 1: Process Three Characters



- Design a solution algorithm that will prompt a terminal operator for three characters, accept those characters as input, sort them into ascending sequence, and output them to the screen. The algorithm is to continue to read characters until "XXX" is entered.
- A. Defining Diagram

Input	Processing	Output
char_1	Prompt for characters	char_1
char_2	Accept three characters	char_2
char_3	Sort three characters	char_3
	Output three characters	

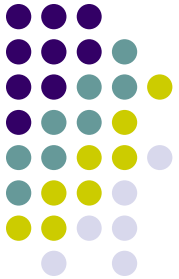
Example 1: Process Three Characters (cont.)



- B. Original Solution Algorithm

```
Prompt the operator for char_1, char_2, char_3
Get char_1, char_2, char_3
DOWHILE NOT (char_1 = “X” AND char_2 = “X”
AND char_3 = “X”)
        IF char_1 > char_2 THEN
                temp = char_1
                char_1 = char_2
                char_2 = temp
        ENDIF
        IF char_2 > char_3 THEN
                temp = char_2
                char_2 = char_3
                char_3 = temp
        ENDIF
        IF char_1 > char_2 THEN
                temp = char_1
                char_1 = char_2
                char_2 = temp
        ENDIF
        Output to the screen char_1, char_2, char_3
        Prompt operator for char_1, char_2, char_3
        Get char_1, char_2, char_3
ENDDO
```

Example 1: Process Three Characters (cont.)



- C. Solution Algorithm Using a Module
  - Use two modules: a mainline module called ProcessThreeCharacters and a submodule called SortThreeCharacters.

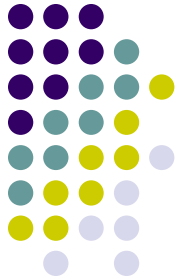
ProcessThreeCharacters

```
    Prompt the operator for char_1, char_2, char_3
    Get char_1, char_2, char_3
    DOWHILE NOT (char_1 = “X” AND char_2 = “X” AND char_3 = “X”)
        SortThreeCharacters
        Output to the screen char_1, char_2, char_3
        Prompt operator for char_1, char_2, char_3
        Get char_1, char_2, char_3
    ENDDO
```

SortThreeCharacters

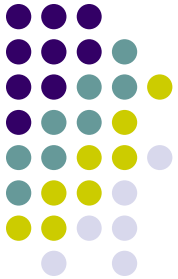
```
    IF char_1 > char_2 THEN
        temp = char_1
        char_1 = char_2
        char_2 = temp
    ENDIF
    IF char_2 > char_3 THEN
        temp = char_2
        char_2 = char_3
        char_3 = temp
    ENDIF
    IF char_1 > char_2 THEN
        temp = char_1
        char_1 = char_2
        char_2 = temp
    ENDIF
```

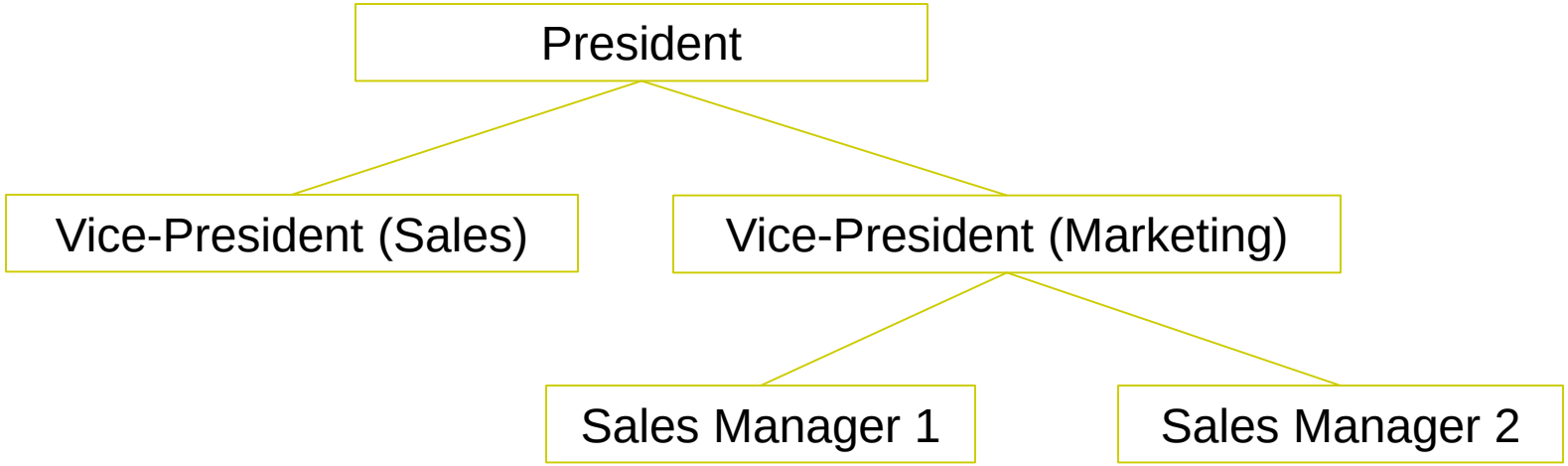
Hierarchy or Structure Charts



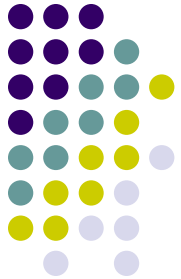
- After the program tasks have been grouped into functions or modules, present the modules graphically in a diagram.
- This diagram is a Hierarchy or Structure chart. It shows the names of the modules and their hierarchical relationships.
- An example is on the following page.

Hierarchy or Structure Charts (Example)



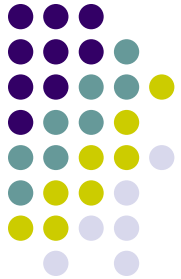


Example 2: Process Three Characters



- Using the same task and Defining Diagram as on slide 130, further modularization…
- B. Solution Algorithm
  - The processing steps in the Defining Diagram can be divided into three separate tasks, each task becoming a module in the algorithm.
  - These tasks are:
    - Read three characters
    - Sort three characters
    - Print three characters

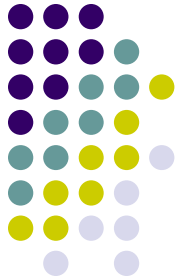
Example 2: Process Three Characters (continued)



- B. Solution Algorithm (continued)

```
Process_three_characters
        Read_three_characters
        DOWHILE NOT (char_1 = “X” AND char_2 = “X” AND char_3 = “X”)
                Sort_three_characters
                Print_three_characters
                Read_three_characters
        ENDDO
END

Read_three_characters
        Prompt the operator for char_1, char_2, char_3
        Get char_1, char_2, char_3
END
```

Example 2: Process Three Characters (continued)

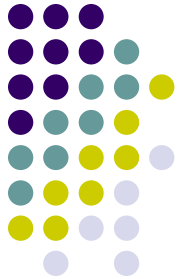


- B. Solution Algorithm (continued)

```
Sort_three_characters
        IF char_1 > char_2 THEN
                temp = char_1
                char_1 = char_2
                char_2 = temp
        ENDIF
        IF char_2 > char_3 THEN
                temp = char_2
                char_2 = char_3
                char_3 = temp
        ENDIF
        IF char_1 > char_2 THEN
                temp = char_1
                char_1 = char_2
                char_2 = temp
        ENDIF
```

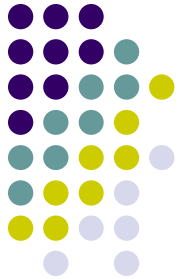
```
Print_three_characters
        Output to the screen char_1, char_2, char_3
END
```

Communication Between Modules



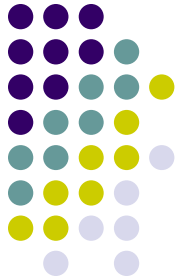
- When designing solution algorithms, you should consider not only the division of the problem into modules but also the flow of information between the modules.
- The fewer and simpler the communications between modules, the easier it is to understand and maintain one module without reference to other modules.

Scope of a Variable



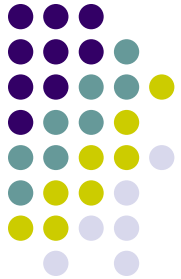
- The scope of a variable is the portion of a program in which that variable has been defined and to which it can be referred.
- If a list is created of all the modules in which a variable can be referenced, that list defines the scope of the variable.

Global Data



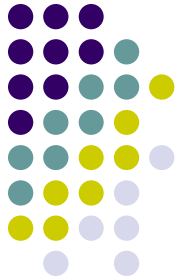
- Global data is data that can be used by all the modules in a program.
- The scope of a global variable is the whole program, because every module in the program can access the data.

Local Data



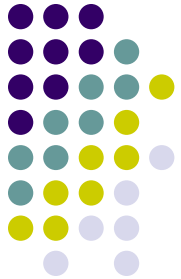
- Variables that are defined within a subordinate module are called local variables.
- These local variables are not known to the calling module, or to any other module.
- The scope of a local variable is simply the module in which it is defined.

Side Effects



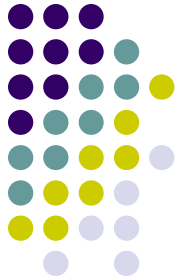
- A side effect is a form of cross-communication of a module with other parts of a program.
- It occurs when a subordinate module alters the value of a global variable inside a module.
- Side effects are not necessarily detrimental, but they do tend to decrease the manageability of a program.

Passing Parameters



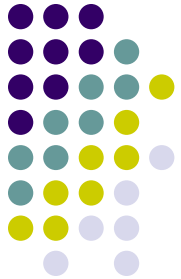
- A particularly efficient method of intermodule communication is the passing of parameters or arguments between modules.
- Parameters are simply data items transferred from a calling module to its subordinate module at the time of calling.
- When the subordinate module terminates and returns control to its caller, the values in the parameters are transferred back to the calling module.
- This method of communication avoids any unwanted side effects.

Formal and Actual Parameters



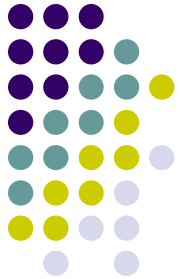
- Parameter names that appear when a submodule is defined are known as *formal parameters.*
- Variables and expressions that are passed to a submodule in a particular call are called *actual parameters.*
- A call to a submodule will include an *actual* parameter list, one variable for each formal parameter name.
- There is one-to-one correspondence between formal and actual parameters, which is determined by the relative position in each parameter list.

Value and Reference Parameters



- Parameters may have one of three functions:
  - To pass information from a calling module to a subordinate module.
  - To pass information from a subordinate module to its calling module.
  - To fulfill a two-way communication role.

Value Parameters



- Value parameters only pass data one way, the *value* of the parameter is passed to the called module.
- When a subordinate module is called, each actual parameter is assigned into the corresponding formal parameter, and from then on, the two parameters are independent.

Reference Parameters

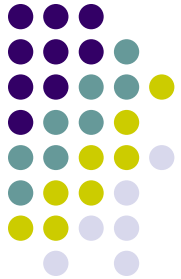
- Reference parameters can pass data to a called module where that data may be changed and then passed back to the calling module, the *reference address* of the parameter is passed to the called module.
- Each actual parameter is an ‘alias’ for the corresponding formal parameter; the two parameters refer to the same object, and changes made through one are visible through the other.
- As a result, the value of the parameter may be referenced and changed during the processing of the submodule.

Example 3: Increment Two Counters

- Design an algorithm that will increment two counters from 1 to 10 and then output those counters to the screen. Your program is to use a module to increment the counters.
- A. Defining Diagram

Input	Processing	Output
counter1	Increment counters	counter1
counter2	Output counters	counter2

Example 3: Increment Two Counters (cont)



- B. Solution Algorithm

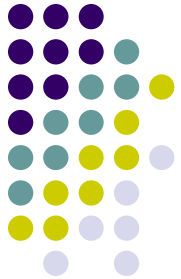
```
Increment_two_counters
        Set counter1, counter2 to zero
        DO      I = 1 to 10
                Increment_counter (counter1)
                Increment_counter (counter2)
                Output to the screen counter1, counter2
        ENDDO
END

Increment_counter (counter)
        counter = counter + 1
END
```

Hierarchy Charts and Parameters

- Parameters, which pass between modules, can be incorporated into a hierarchy chart or structure chart using these symbols:

- Data parameters contain the actual variables or data items that will be passed between modules.
- Status parameters act as program flags and should contain just one of two values: true or false.

Hierarchy Charts and Parameters (cont.)



- When designing modular programs, the programmer should avoid using data parameters to indicate status as well, because this can affect the program in two ways:
  - It may confuse the reader of the program because a variable has been overloaded.
  - It may cause unpredictable errors when the program is amended at some later date, as the maintenance programmer may be unaware of the dual purpose of the variable.

Using Parameters in Program Design / Process Three Characters

- Look again at example 2 and change the solution algorithm so that the parameters are used to communicate between modules.
- Process Three Characters
  - Design a solution algorithm that will prompt a terminal operator for three characters, accept those characters as input, sort them into ascending sequence, and output them to the screen. The algorithm is to continue to read characters until “XXX” is entered.

Process Three Characters (continued)

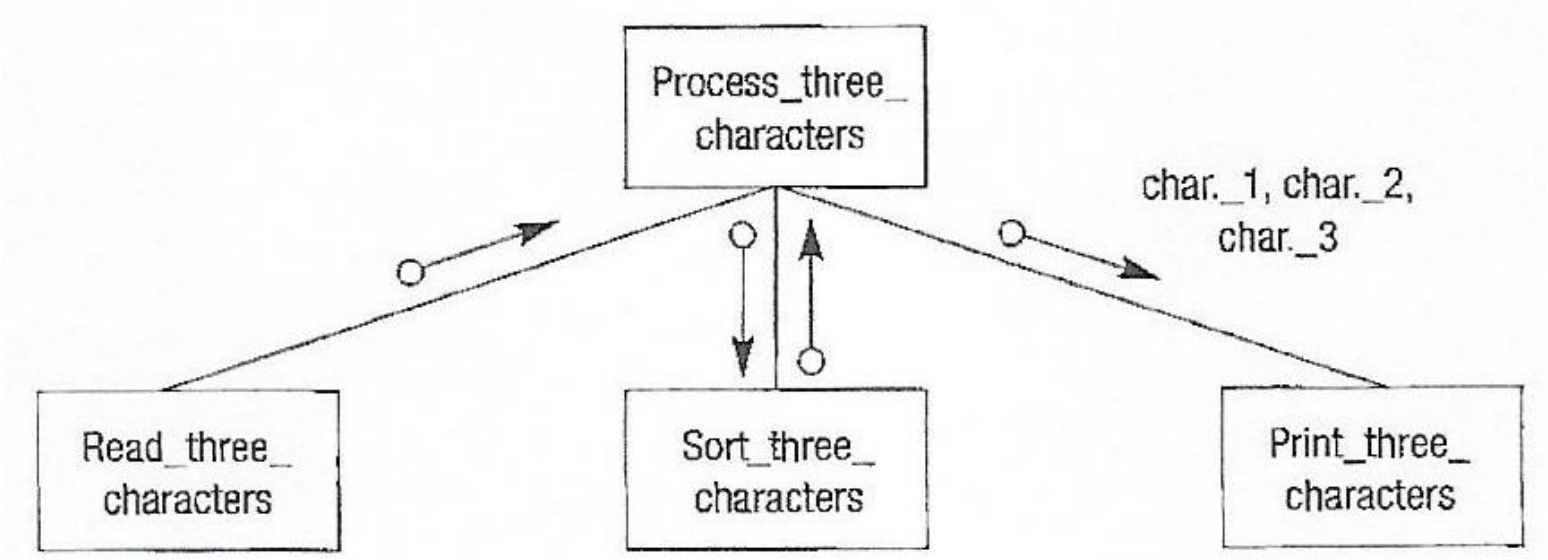
- A. Defining Diagram

Input	Processing	Output
char_1	Prompt for characters	char_1
char_2	Accept three characters	char_2
char_3	Sort three characters	char_3
	Output three characters	

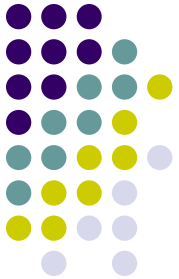
- B. Group the activities into modules.
  - A module will be created for each processing step in the defining diagram.

Process Three Characters (continued)

- C. Construct a Hierarchy Chart



Process Three Characters (continued)



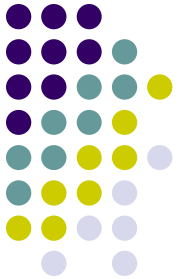
- D. Establish the Logic of the Solution Algorithm Using Pseudocode
- The mainline will call the module Read_three_characters, which will get the three input characters and send them to the mainline as parameters.
- These parameters will then be passed to the module Sort_three_characters, which will sort the three characters and send these sorted values back to the mainline as parameters.

Process Three Characters (continued)

- D. Establish the Logic of the Solution Algorithm Using Pseudocode
- Code:

```
Process_three_characters
        Read_three_characters (char_1, char_2, char_3)
        DOWHILE NOT (char_1 = “X” AND char_2 = “X” AND char_3 = “X”)
                Sort_three_characters (char_1, char_2, char_3)
                Print_three_characters (char_1, char_2, char_3)
                Read_three_characters (char_1, char_2, char_3)
        ENDDO
END
Read_three_characters (char_1, char_2, char_3)
        Prompt the operator for char_1, char_2, char_3
        Get char_1, char_2, char_3
END
Print_three_characters (char_1, char_2, char_3)
        Output to the screen char_1, char_2, char_3
END
```

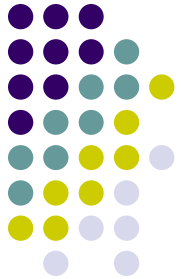
Process Three Characters (continued)



- Code (continued):

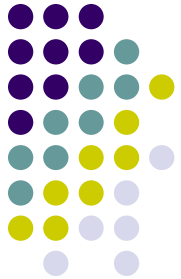
```
Sort_three_characters (char_1, char_2, char_3)
        IF char_1 > char_2 THEN
                temp = char_1
                char_1 = char_2
                char_2 = temp
        ENDIF
        IF char_2 > char_3 THEN
                temp = char_2
                char_2 = char_3
                char_3 = temp
        ENDIF
        IF char_1 > char_2 THEN
                temp = char_1
                char_1 = char_2
                char_2 = temp
        ENDIF

END
```

Further Modularization



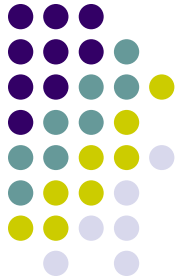
- The module Sort_three_characters contains some repeated code, which looks cumbersome.
- Further modularization could be achieved by the introduction of a new module, called Swap_two_characters, which is called by the module Sort_three_characters.

Steps in Modularization



1. Define the problem by dividing it into its three components: input, output, and processing.
2. Group the activities into subtasks or functions to determine the modules that will make up the program.
3. Construct a hierarchy chart to illustrate the modules and their relationship to each other.
4. Establish the logic of the mainline of the algorithm in pseudocode.
5. Develop the pseudocode for each successive module in the hierarchy chart.
6. Desk check the solution algorithm.

Programming Examples Using Modules



- Examples 8.5 and 8.6 on pages 124–132 of the text depict the six steps in modularization.

Summary

- This section introduced a modular approach to program design.
- A module was defined as a section of an algorithm that is dedicated to the performance of a single function.
- Top-down design was defined as the process of dividing a problem into major tasks and then into further subtasks within those major tasks until all the tasks have been identified.
- Hierarchy charts were introduced as a method of illustrating the structure of a program that contains modules.
- The steps in modularization that a programmer must follow were listed.
- Programming examples using these six steps in modularization were then developed in pseudocode.