4.6: Communication between modules, cohesion and coupling

IT1406 - Introduction to Programming

Level I - Semester 1

4.6. Communication between modules, cohesion and coupling

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 referenced.
- If a list is created of all the modules in which a variable can be referenced, that list defines the scope of the variable.
- Variables can be global, where the scope of the variable is the whole program, or local, where the scope of the variable is simply the module in which it is defined.

Scope of a 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 and change that data.
- The lifetime of a global variable spans the execution of the whole program.

Scope of a variable

Local data

- Variables that are defined within a submodule 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.
- The lifetime of a local variable is limited to the execution of the single submodule in which it is defined.
- Using local variables can reduce what is known as program side effects.

Scope of a variable

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;
- however, they do tend to decrease the manageability of a program.
- A programmer should be aware of their impact.
- Sometimes, a programmer may need to modify an existing program, and in doing so, may make a change to a global variable.
- This change could cause side effects or erroneous results because the new programmer is unaware of other modules that also alter that global variable.

Scope of a variable

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 may be transferred back to the calling module.
- This method of communication avoids any unwanted side effects, as the only interaction between a module and the rest of the program is via parameters.

Scope of a variable

Passing parameters

To pass parameters between modules, two things must happen:

1. The calling module must name the parameters that it wants to pass to the submodule, at the time of calling.

2. The submodule must be able to receive those parameters and return them to the calling module, if required.

- In pseudocode and most programming languages, when a calling module wants to pass parameters to a submodule, it simply lists the parameters, enclosed in parentheses, beside the name of the submodule, for example:

Print_page_headings (pageCount, lineCount)

Scope of a variable

- The submodule must be able to receive those parameters, so it, too, lists the parameters that it expects to receive, enclosed in parentheses, beside the submodule name when it is defined, for example:

  Print_page_headings (pageNumber, lineNumber)

- The names that the respective modules give to their parameters need not be the same – in fact, they often differ because they have been written by a different programmer – but their number, type and order must be identical.
- In the above example, the parameter pageCount will be passed to pageNumber, and the parameter lineCount will be passed to lineNumber.
- In this book, parameters will be named without underscores, to differentiate them from variables.

Formal and actual parameters

- For example, a mainline may call a module with an actual parameter list, as follows:

  Calculate_amount_owing (gasFigure, amountBilled)

- while the module may have been declared with the following formal parameter list:

  Calculate_amount_owing (gasUsage, amountOwing)

- Although the parameter names are different, the actual and formal parameters will correspond.

Value and reference parameters

Parameters may have one of three functions:

1. To pass information from a calling module to a subordinate module. The subordinate module would then use that information in its processing, but would not need to communicate any information back to the calling module.

2. To pass information from a subordinate module to its calling module. The calling module would then use that parameter in subsequent processing.

3. To fulfil a two-way communication role. The calling module may pass information to a subordinate module, where it is amended in some fashion, then passed back to the calling module.

Value and reference parameters

- Let’s look at an example that illustrates the passing of parameters by value:

Value and reference parameters

Defining diagram

Input	Processing	Output
numerator denominator	Get numerator, denominator Convert fraction to percentage Display percentage	percentage

Value and reference parameters

Solution algorithm

```
Calculate_percentage_value
    Prompt for numerator, denominator
    Get numerator, denominator
    Convert_fraction_value (numerator, denominator, percentage)
    IF percentage NOT = 0 THEN
    Output to screen, percentage, '%'
    ELSE
    Output to screen 'invalid fraction'
    ENDIF
END
```

Value and reference parameters

Solution algorithm

```
Convert_fraction_value (numerator, denominator,
calculatedPercentage)
    IF denominator NOT = 0
        calculatedPercentage = numerator / denominator * 100
    ELSE
        calculatedPercentage = 0
    ENDIF
END
```

Value and reference parameters

Solution algorithm

- In this example, copies of the numerator and denominator values are passed as parameters to the module Convert_fraction_value, which will use those values to calculate the percentage.
- When the percentage is calculated, a copy of the value in the parameter calculatedPercentage will be passed to the parameter percentage, which will be displayed to the screen.
- This is an example of passing by value.

Value and reference parameters

- Now let's look at an example that illustrates the passing of parameters by reference.

Value and reference parameters

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.

Value and reference parameters

Defining diagram

Input	Processing	Output
counter1 counter2	Increment counters Output counters	counter1 counter2

Value and reference parameters

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
```

Value and reference parameters

Solution algorithm

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

Value and reference parameters

- In this example, the module Increment_counter is defined with a formal parameter named counter.
- Increment_counter is called by the mainline, first, with the actual parameter counter1, and then with the actual parameter counter2.
- At the first call, the reference address of counter1 is passed to the parameter counter, and its value is changed.
- Then the reference address of counter2 is passed to the parameter counter, and its value is also changed.

Value and reference parameters

- The values of counter1 and counter2 are displayed on the screen and the process is repeated, so that each time the module Increment_counter is called the values in the parameters counter1 or counter2 are increased by 1.
- The screen output would be as follows:

1 1

2 2

3 3

4 4 etc.

Module cohesion

- A module has been defined as a section of an algorithm that is dedicated to the performance of a single function.
- It contains a single entry and a single exit, and the name chosen for the module should describe its function.
- Programmers often need guidance in determining what makes a good module.
- Common queries include: ‘How big should a module be?’, ‘Is this module too small?’ and ‘Should I put all the read statements in one module?’

Module cohesion

- There is a method you can use to remove some of the guesswork when establishing modules.
- You can look at the cohesion of the module. Cohesion is a measure of the internal strength of a module; it indicates how closely the elements or statements of a module are associated with each other.
- The more closely the elements of a module are associated, the higher the cohesion of the module.
- Modules with high cohesion are considered good modules, because of their internal strength.
- Edward Yourdon and Larry Constantine established seven levels of cohesion and placed them in a scale from the weakest to the strongest.

Module cohesion

Cohesion level	Cohesion attribute	Resultant module strength
Coincidental	Low cohesion	Weakest
Logical		
Temporal		
Procedural		
Communicational		
Sequential		
Functional	High cohesion	Strongest