A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

11.3.2 Parameters by value and by reference, and variable scope

Name: _____ Class: _____ Date: _____ **Total: 32 marks**

KEY WORDS return value 返回值 • pass by reference 传引用 • pass by value 传值 • pseudocode 伪代码 • procedure 过程

Objective

Build the skills to answer exam questions on the half of **structured programming** 结构化编程 that decides whether a subroutine's work reaches the caller at all: **pass by value** 传值 against **pass by reference** 传引用, and the **scope** 作用域 and lifetime of a variable. Sheet 11.3 states the difference in a line; this sheet is about tracing it, choosing the right mode for each parameter, and explaining why a **local variable** 局部变量 loses its value between calls.

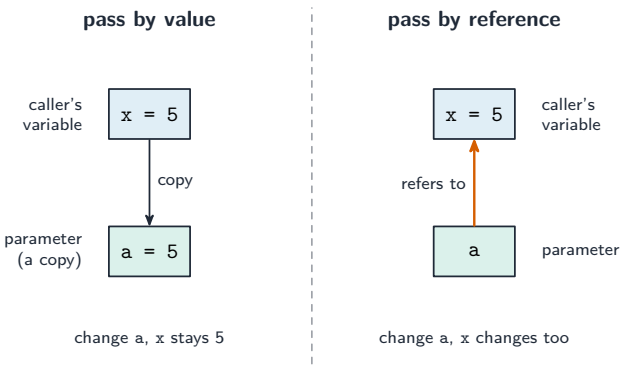
You must be able to:

- use parameters, understanding that a parameter may be passed by reference or by value
- trace what a subroutine changes in the caller, and what it does not
- choose the right passing mode for each parameter of a procedure
- use the terminology of procedures and functions when explaining your choice
- explain the difference between a local variable and a **global variable** 全局变量, and write efficient pseudocode that prefers locals and parameters

1 Worked examples

Study these first. Each one shows a **method** that a question later on this sheet needs.

■ The two modes, in one picture



- **BYVAL** – the subroutine gets a **copy** in a box of its own. Anything it writes there is thrown away when the call ends.
- **BYREF** – the parameter **is** the caller's box, under another name. Anything it writes goes straight into the caller's variable.
- Cambridge pseudocode writes the mode in the header. If neither word appears, **BYVAL is assumed**, so a subroutine that must change the caller's variable needs BYREF written in.

■ Tracing a call

```
PROCEDURE Update(BYREF Count : INTEGER, BYVAL Step : INTEGER)
    Count <- Count + Step
    Step  <- 0
ENDPROCEDURE

Total  <- 12
Amount <- 4
CALL Update(Total, Amount)
OUTPUT Total, Amount
```

Take the parameters one at a time and say which box each one is.

1. `Count` is `BYREF`, so `Count` **is** `Total`. `Count <- Count + Step` writes `16` into `Total`.
2. `Step` is `BYVAL`, so `Step` is a copy of `Amount`. `Step <- 0` empties the copy and leaves `Amount` at `4`.
3. The output is **`16, 4`**.

- Change the header to `BYVAL Count` and the output becomes **`12, 4`**: the addition still happens, but in a box nobody else can see.
- A trace question is marked on the *reason*, so write "Count is BYREF so it is

Total's own box", not just the number.

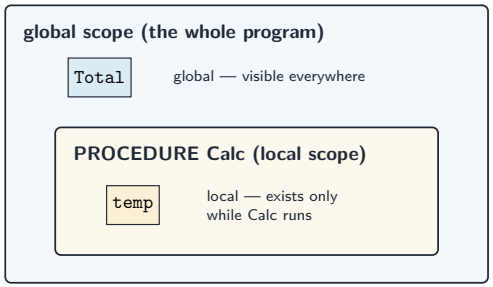
Choosing the mode for each parameter

Ask of each parameter separately: **does the caller need to see a change to this one?**

The subroutine...	Mode	Why
only reads the value	`BYVAL`	nothing needs to go back, and a copy protects the caller
must update that variable	`BYREF`	the change has to reach the caller's own box
produces exactly one value	neither – use a **function**	the return value replaces the call in an expression

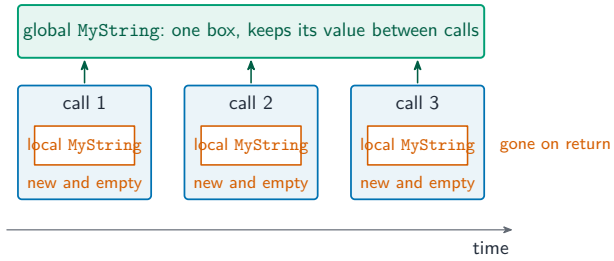
- `Swap` needs **both** parameters `BYREF`: two values change, so there is no single value to return.
- Passing a large array `BYREF` also avoids copying it, but that is a side benefit, not the reason the syllabus asks for.
- The mode is part of the subroutine's **interface** – everything a caller must know in order to use it. Changing `BYVAL` to `BYREF` therefore changes the interface, even though the **argument** written at the call site looks exactly the same.
- **Where in the construction of an algorithm** does a procedure with a `BYREF` parameter belong? At a step that has to change something the rest of the algorithm can see. If the step instead produces a single value, a function fits better, because its return value can be used where the call stands.

Scope: where a name can be seen



- A **global** variable is declared outside every subroutine and can be read or written **anywhere** in the program.
- A **local** variable is declared inside a subroutine and can only be seen **inside that subroutine**. Using the name outside it is an error.
- The one-line answer the scheme wants: *a global variable can be accessed from anywhere in the program; a local variable only inside the subroutine that declares it.*

Lifetime: why a local starts empty every time



A local variable is **created when the subroutine is called and destroyed when it returns**, so it cannot carry a value from one call to the next.

A procedure adds 1 to a counter and outputs it, and is called three times:

- with the counter **global**, one box survives between calls: the outputs are `1`, `2`, `3`.
- with the counter **local**, a new empty box is made each time: the outputs are `1`, `1`, `1`.
- This is the standard "the procedure does not work as expected" question. The answer is the lifetime, not the spelling: the earlier value is gone because the box is gone.
- Two ways to keep a value between calls: make it **global**, or pass it as a **`BYREF` parameter** so the box belongs to the caller.

Why locals are still preferred

Even though globals look easier, the scheme credits these for using locals and parameters instead:

- the same identifier can be reused in another subroutine with no clash,
- the value cannot be changed accidentally by another part of the program,
- the memory is released when the subroutine ends,
- the subroutine is **self-contained**, so it can be tested on its own and reused in another program.

2 Practice

Now use the methods above.

2.1 State what is meant by passing a parameter **by value**, and what is meant by passing it **by reference**. [2]

2.2 For the `Update` procedure in the worked example, `Total` is `12` and `Amount` is `4` before the call. State the two values output, and give the reason for each. [2]

2.3 The header is changed so that **both** parameters are `BYVAL`. State the two values now output, and explain the change. [2]

2.4 State what is meant by the **scope** of a variable. [1]

2.5 A procedure's header is changed from `BYVAL` to `BYREF` for one parameter. State whether the **interface** of the procedure has changed, and state whether the **argument** written at the call site has to change. [2]

2.6 A programmer declares a variable inside a procedure and then tries to use it in the main program. State what happens, and why. [2]

2.7 State **two** advantages of using a local variable rather than a global variable. [2]

3 Exam-style questions

3.1 A procedure `Swap()` is intended to exchange the values of two integer variables. Before the call, `A` is `7` and `B` is `2`.

(a) The header is `PROCEDURE Swap(BYVAL X : INTEGER, BYVAL Y : INTEGER)`. State the values of `A` and `B` after the call, and explain your answer. [2]

(b) Write a header that makes the procedure work as intended. [2]

3.2 A subroutine must give the calling program **two** values. Describe **two** different ways of doing this, and state one reason to prefer each. [4]

3.3 A program stores a running total in a global variable that three different procedures update.

(a) State **two** problems this can cause. [2]

(b) Describe one change that removes those problems. [2]

3.4 A procedure `Record()` takes three parameters: a customer's name, which it only reads; a running total, which it must increase; and an array of prices, which it only reads.

State, with a reason, whether each parameter should be passed `BYVAL` or `BYREF`. [4]

3.5 A procedure declares a counter as a local variable, adds 1 to it and outputs it. The procedure is called three times.

State the three values output, explain why, and name **two** ways to make the counter keep its value between calls. [3]

Solutions

Each answer shows the steps, then the **result**.

2.1

- **by value** – the subroutine receives a **copy** of the value, so changes it makes do not affect the caller's variable
- **by reference** – the parameter refers to the caller's **own** variable, so changes it makes do affect the caller

2.2

- `Total` is **16**: `Count` is `BYREF`, so it is `Total`'s own box and the addition is written into it
- `Amount` is **4**: `Step` is `BYVAL`, so setting it to 0 only empties the copy

2.3

- `Total` is **12** and `Amount` is **4**
- both parameters are now copies, so the addition happens inside the procedure and nothing reaches the caller

2.4

- the scope of a variable is the part of the program in which that name can be accessed

2.5

- the **interface has changed**: the passing mode is part of the interface, which is everything a caller must know in order to use the procedure
- the **argument does not change**: it is still written as the variable's name at the call site, exactly as before – what changes is what the procedure may do to it

2.6

- the variable is **local** to the procedure, so it does not exist outside it
- the main program cannot access the name, so this is an error (and the value is destroyed when the procedure returns)

2.7

- any two: the same identifier may be used in another subroutine without a clash; the value cannot be changed accidentally by another part of the program; the memory is released when the subroutine ends; the subroutine is self-contained, so it can be tested on its own and reused

3.1

(a) `A` is still **7** and `B` is still **2**

- both parameters are copies, so the exchange happens inside the procedure and is lost when it returns

(b) both parameters must be passed by reference `PROCEDURE Swap(BYREF X : INTEGER, BYREF Y : INTEGER)`

3.2

- **two `BYREF` parameters** – the procedure writes into the caller's own variables
- prefer this when neither value is "the answer": both are updates the caller asked for
- **a function returning one value, with the other as a `BYREF` parameter**, or returning a record/array holding both
- prefer a function when one value is clearly the result, because the return value can be used directly in an expression

3.3

(a) any two: any procedure can change it, so a fault is hard to trace to one place; a change made by one procedure can be overwritten by another; the procedures cannot be tested on their own; the name cannot be reused elsewhere

(b) make the total a **local** variable in the calling program and pass it to each procedure as a `BYREF` parameter

- each procedure is then self-contained and can be tested alone, and only a procedure actually given the total can change it

3.4

- the name: **`BYVAL`** – it is only read, and a copy protects the caller's variable
- the running total: **`BYREF`** – the procedure must increase it and the new value has to reach the caller
- the array of prices: **`BYVAL`** – it is only read (passing it `BYREF` would avoid copying it, but nothing needs to go back)

3.5

- the outputs are **1, 1, 1**
- a local variable is created when the procedure is called and destroyed when it returns, so each call starts with a new, empty counter and the earlier value is gone
- to keep it: make the counter **global**, or pass it in as a **`BYREF` parameter** so the box belongs to the caller