

Procedures, functions and library routines

Read online at pastopic.com/cambridge/computer-science-0478/notes/procedures-and-functions

Last checked 30 September 2026

Read first: [Programming concepts](#)

© 2026 Pastopic. Free for personal study. Please share the link, not the file.

What you need to know

By the end of this topic you should be able to:

1. **Say what procedures, functions and parameters are:** a procedure and a function are both named blocks of code (subroutines) that the main program can run when it needs them; a function also hands a value back; a parameter carries a value into the subroutine.
2. **Write and use procedures and functions,** with or without parameters (up to three parameters each): define them, then call them from the program.
3. **Use local and global variables:** know where in a program each can be used, and choose the right one.
4. **Use the library routines** MOD, DIV, ROUND and RANDOM: say what each one does, and write statements that use them.

Declaring variables, data types and the operators are in the note on **Programming concepts**; this note uses them where a question needs them.

Understand it

Why split a program into subroutines

Think of a recipe book. The recipe for a cake says "make the icing (see page 40)" instead of repeating the icing method every time. A **subroutine** is the same idea in a program: a block of statements with a **name**, written once, that the program can ask to run whenever it needs that job done. Running a subroutine is called **calling** it.

Cambridge pseudocode has two kinds of subroutine:

- a **procedure** does a job, such as displaying a menu or saving a record;
- a **function** does a job **and hands back one value**, such as working out a price or checking whether a code is valid.

Write every procedure and function **first**, above the main program that uses it.

Procedures: defining and calling

A procedure with no parameters:

```
PROCEDURE ShowMenu
    OUTPUT "1. New game"
    OUTPUT "2. Load game"
    OUTPUT "3. Quit"
ENDPROCEDURE
```

To run it, the program uses the keyword `CALL` and the name, as a statement on its own line:

```
CALL ShowMenu
```

Writing the definition does not run anything. The statements inside only run when the procedure is called, and they run afresh every time it is called. So a program can define `ShowMenu` once and call it ten times.

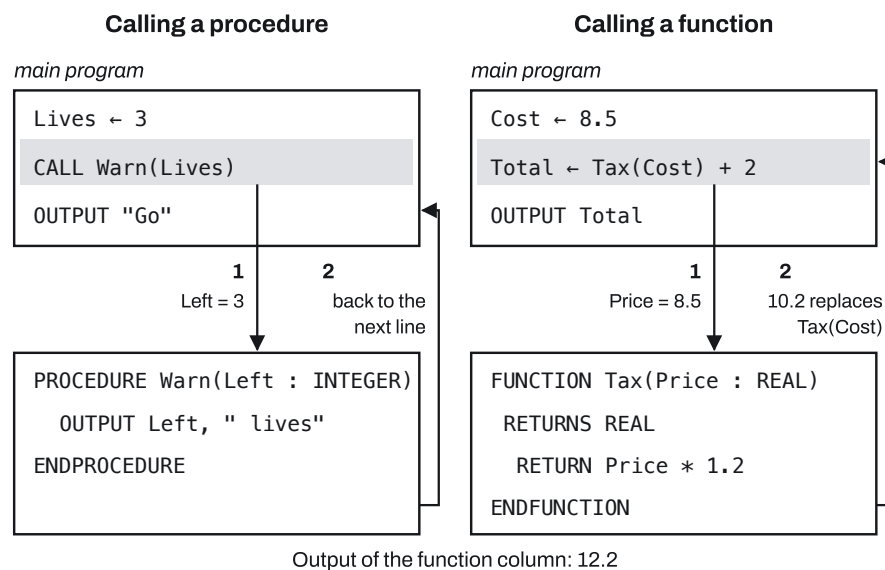
Parameters

A **parameter** lets the calling program pass a value in, so the same subroutine can work on different data each time. In the definition, each parameter gets a name and a data type, in brackets after the procedure's name:

```
PROCEDURE Warn(Left : INTEGER)
    OUTPUT Left, " lives"
ENDPROCEDURE
```

In the call, you put the actual value (or a variable, or an expression) in the same position:

```
Lives ← 3
CALL Warn(Lives)
OUTPUT "Go"
```



What happens on the call (the left column of the diagram):

1. The program **jumps** to the procedure named `Warn`.
2. The parameter `Left` is given the value passed in, 3.
3. The procedure's statements run: it outputs 3 lives.
4. At `ENDPROCEDURE`, control **returns** to the line **after** the call, and `OUTPUT "Go"` runs.

Rules for parameters:

- The values in the call are matched to the parameters **by position**: the first value goes to the first parameter, and so on. Give them in the right order and the right number.
- Each value must have the parameter's data type: a procedure expecting an `INTEGER` must not be sent `"three"`.
- A string value in a call needs quotes: `CALL Greet("Mia")`.
- In the syllabus a procedure or function has **at most three** parameters, separated by commas: `PROCEDURE Box(Width : INTEGER, Height : INTEGER, Fill : CHAR)`.
- Inside the subroutine, a parameter is used just like a variable.

Functions

A function is defined like a procedure, with two extra things: the header says **what type of value it gives back** (`RETURNS <type>`), and the body uses `RETURN` to send that value back.

```
FUNCTION Tax(Price : REAL) RETURNS REAL
    RETURN Price * 1.2
ENDFUNCTION
```

A function call stands for a value, so put it somewhere a value makes sense, such as the right-hand side of an assignment. Never write `CALL` in front of it:

```
Cost ← 8.5
Total ← Tax(Cost) + 2
OUTPUT Total
```

When `Tax(Cost)` is reached (the right column of the diagram), control jumps to the function with `Price = 8.5`, the function works out $8.5 \times 1.2 = 10.2$, and `RETURN` sends 10.2 back. Think of `Tax(Cost)` as being swapped for **10.2**: the line now reads `Total ← 10.2 + 2`, and `Total` is 12.2.

Other ways to use a function's result: `OUTPUT Tax(20)`, or `IF Tax(Cost) > 100 THEN ....` A call on a line by itself, such as `Tax(Cost)`, would throw the answer away, so it is always wrong.

A function can have several `RETURN` statements, for example one in each branch of an `IF`; the first one reached ends the function.

```
FUNCTION Grade(Mark : INTEGER) RETURNS CHAR
    IF Mark >= 50
    THEN
```

```

        RETURN 'P'
    ELSE
        RETURN 'F'
    ENDIF
ENDFUNCTION

```

Procedure or function?

	Procedure	Function
Hands back a value?	No (it may still output or change things)	Always one value
Header	PROCEDURE Name(Param : TYPE)	FUNCTION Name(Param : TYPE) RETURNS TYPE
Ends with	ENDPROCEDURE	ENDFUNCTION, with a RETURN inside
Called by	CALL Name(values) as its own statement	using Name(values) inside an expression, with no CALL

Defining a subroutine means writing it: its header, its statements and its end keyword. This is done once.

Calling it means running it from somewhere else in the program, and this can happen as many times as needed.

Local and global variables

Every variable has a **scope**: the part of the program where it can be used.

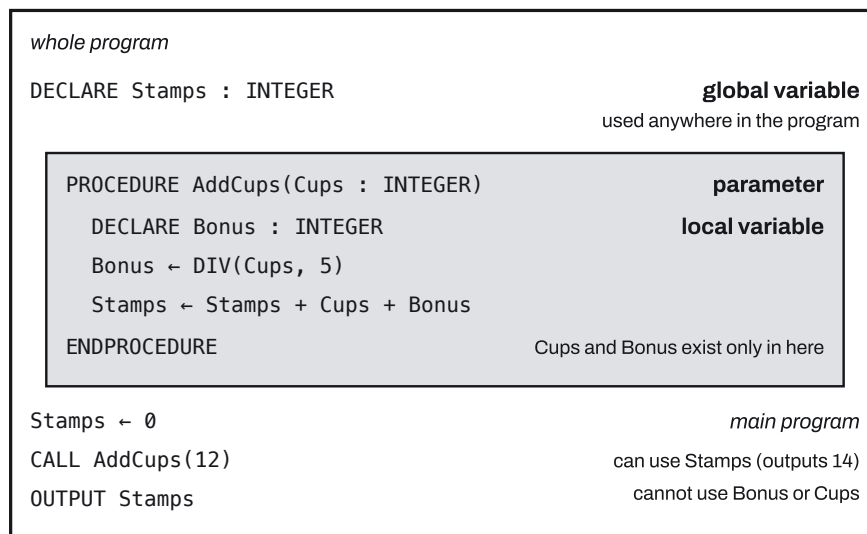
```

DECLARE Stamps : INTEGER          // global

PROCEDURE AddCups(Cups : INTEGER)
    DECLARE Bonus : INTEGER       // local
    Bonus ← DIV(Cups, 5)
    Stamps ← Stamps + Cups + Bonus
ENDPROCEDURE

Stamps ← 0
CALL AddCups(12)
OUTPUT Stamps

```



- A **global variable** is declared in the main program, outside every subroutine. Its scope is **the whole program**: the main program and every procedure and function can read it and change it. Here `Stamps` is global. It exists from when it is declared until the program ends.
- A **local variable** is declared **inside** a procedure or function. Its scope is **only that subroutine**: nothing outside it can see or change it. It is created when the subroutine is called and removed when the subroutine ends, so its memory is freed and its value is not kept for the next call. Here `Bonus` is local. A parameter such as `Cups` is local to its subroutine too.

The call `AddCups(12)` makes `Bonus = DIV(12, 5) = 2`, so `Stamps` becomes $0 + 12 + 2 = 14$ and 14 is output. If the main program tried `OUTPUT Bonus`, it would fail: `Bonus` doesn't exist outside `AddCups`.

Why prefer local variables where you can:

- a subroutine can't accidentally change a value that another part of the program depends on;
- two subroutines can each use a local variable with the same name (such as `Count`) without clashing;
- memory is only used while the subroutine runs;
- the subroutine is self-contained, so it can be tested alone and copied into another program.

Use a global variable only for data that many parts of the program genuinely share, such as a running score used by several procedures.

Library routines

A **library routine** is a ready-made subroutine that comes with the language. You don't write it; you just call it by name. The syllabus names four.

Routine	What it gives back	Example	Result
<code>DIV(a, b)</code>	how many whole times <code>b</code> goes into <code>a</code> (the quotient, fractional part thrown away)	<code>DIV(23, 4)</code>	5
<code>MOD(a, b)</code>	the remainder left after that whole-number division	<code>MOD(23, 4)</code>	3

Routine	What it gives back	Example	Result
ROUND(<i>x</i> , <i>places</i>)	<i>x</i> rounded to <i>places</i> decimal places	ROUND(3.14159, 2)	3.14
RANDOM()	a random number from 0 to 1 inclusive	RANDOM()	e.g. 0.7302

- DIV and MOD are a pair: $23 = 5 \times 4 + 3$. They work on integers. DIV(*a*, *b*) is also written *a* DIV *b*, and MOD(*a*, *b*) as *a* MOD *b*.
- ROUND rounds a 5 upwards: ROUND(6.75, 1) is 6.8 and ROUND(19.5, 0) is 20. *places* = 0 gives a whole number.
- RANDOM() on its own is rarely what you want; you almost always scale it and round it to get a **random whole number in a range**:

$\text{ROUND}(\text{RANDOM()} * (\text{High} - \text{Low}), 0) + \text{Low}$

gives a whole number from Low to High inclusive. For 50 to 80: $\text{ROUND}(\text{RANDOM()} * 30, 0) + 50$. Check the ends: $\text{RANDOM()} = 0$ gives 50 and $\text{RANDOM()} = 1$ gives 80. Watch out: $\text{ROUND}(\text{RANDOM()} * 6, 0)$ gives 0 to 6 (seven different values), not 1 to 6; for a dice you need $\text{ROUND}(\text{RANDOM()} * 5, 0) + 1$.

Key facts and formulas

There is no formula sheet for Paper 2; learn these.

Fact	Status
PROCEDURE Name(<i>P1</i> : TYPE, ...) ... ENDPROCEDURE; called with CALL Name(values)	Learn it
FUNCTION Name(<i>P1</i> : TYPE, ...) RETURNS TYPE ... RETURN value ... ENDFUNCTION; called inside an expression, never with CALL	Learn it
Write every procedure and function first, above the main program that uses it; at most 3 parameters each	Learn it
Values in a call match the parameters by position and must have the right data type	Learn it
After a procedure, the program carries on from the line after the call; a function's answer is used where its name was written	Learn it
Global: declared outside subroutines, scope is the whole program. Local: declared inside a subroutine, scope is that subroutine only, removed when it ends	Learn it
DIV(<i>a</i> , <i>b</i>) whole-number quotient; MOD(<i>a</i> , <i>b</i>) remainder; $a = \text{DIV}(a, b) \times b + \text{MOD}(a, b)$	Learn it

Fact	Status
<code>ROUND(x, places)</code> rounds to <code>places</code> decimal places	Learn it
<code>RANDOM()</code> gives a number from 0 to 1 inclusive; a whole number from Low to High is <code>ROUND(RANDOM() * (High - Low), 0) + Low</code>	Learn it

How to do it

In the Paper 2 questions we have tagged for this topic (21 questions, 2022–2025), 3 mix more than one type, so the counts add up to 26.

Question type	Questions
Write a procedure or function, and call it	7
State the purpose of a library routine, with an example	4
Use subroutines or library routines in the 15-mark program	4
Describe or identify procedures, functions and parameters	4
Compare local and global variables	4
Explain why programmers use procedures, functions and parameters	2
Trace an algorithm that uses DIV and MOD	1

1. Write a procedure or function, and call it (7 questions)

Usually 4–6 marks for the definition and 1–2 for the call, one mark per feature. Often part (a) first asks you to declare the variables the question describes.

- 1. Declare any variables asked for**, one `DECLARE Name : TYPE` line each, choosing the type from the description ("a whole number" → `INTEGER`, "text" → `STRING`, "true or false" → `BOOLEAN`).
- 2. Procedure or function?** If the question says the subroutine "returns" something, it is a function; if it only outputs or stores, it is a procedure.
- 3. Write the header** with the exact name from the question and every parameter it lists, each with a data type. A function also needs `RETURNS` and the type of the value it gives back.
- 4. Write the body**: use the parameters (not fresh `INPUTS`) as the data. Declare any extra variables inside as local.
- 5. For a function, make sure every path ends in RETURN**: with an `IF`, put a `RETURN` in both branches.
- 6. Close it** with `ENDPROCEDURE` or `ENDFUNCTION`.
- 7. Write the call**: `CALL Name(values)` for a procedure; `Variable ← Name(values)` for a function, when the question says to store the result.

For example, a function that decides whether a year's rainfall in millimetres counts as "Dry":

```

FUNCTION IsDry(Rain : REAL) RETURNS BOOLEAN
    IF Rain < 250
        THEN
            RETURN TRUE
        ELSE
            RETURN FALSE
    ENDIF
ENDFUNCTION

```

```
Desert ← IsDry(Rainfall)
```

Variations you will meet:

- **Complete a part-written subroutine:** fill the header's parameter list, then use that parameter wherever the body needs the value (for example as the end value of a FOR loop).
- **Replace existing lines with a call:** turn the repeated lines into the subroutine body, with parameters for the values that change; the call then passes exactly the values or expressions that the old lines used, in the same order.
- **Input a value, then call with it:** `INPUT HowMany` then `CALL AddItems(HowMany)`.
- **Call with fixed values:** numbers as they are, strings in double quotes: `CALL Label("Oak", 3)`.
- **Use a library routine inside the body:** `DIV` and `MOD` to split a quantity into whole units and what is left over, `MOD(Value, n) = 0` to test whether a value divides exactly, `ROUND` to compare or format a real number.
- **A procedure that writes to a file:** open, write the parameter, close (see **File handling**).

2. State the purpose of a library routine, with an example (4 questions)

Worth 1 mark per purpose and 1 per example; sometimes a multiple-choice "which routine ..." item.

1. Give what the routine **hands back**, in one precise phrase: `DIV`, the whole-number part of a division; `MOD`, the remainder; `ROUND`, a value rounded to a given number of decimal places; `RANDOM`, a random number (from 0 to 1, usually scaled to a range).
2. Give an example as a **complete pseudocode statement** with real values, and state its result if asked:
`Weeks ← DIV(365, 7)` stores 52; `Spare ← MOD(365, 7)` stores 1; `Price ← ROUND(2.675, 2)` stores 2.68; `Card ← ROUND(RANDOM() * 12, 0) + 1` stores a whole number from 1 to 13.

"Divides" alone is not enough for `DIV`: say it gives the whole-number quotient, and that `MOD` gives the remainder. "Rounds a number" without saying to how many places is vague too; be exact about what comes back.

3. Use subroutines or library routines in the 15-mark program (4 questions)

The scenario may say "each action is completed by a procedure with a parameter" or "create one function for each ...", or need random numbers or rounding. The marking is the same as any 15-mark program (see **Programming concepts**), but a named requirement you skip costs marks.

1. Write the subroutines **first**, with the names and parameters the question gives.
2. Keep each one to a single job, with its data passed in as parameters and any working variables declared inside it.
3. In the main program, call each one at the right point: procedures with `CALL`, functions inside an assignment or `OUTPUT`.
4. For random data use the range formula, checking both ends; for "rounded to n decimal places" wrap the calculation or the output in `ROUND (... , n)`.
5. Comment each subroutine to say what it does.

4. Describe or identify procedures, functions and parameters (4 questions)

Short definitions, matching lines or a multiple-choice tick.

- **Procedure**: a named subroutine that carries out a task and doesn't have to hand back a value.
- **Function**: a named subroutine that always hands back one value to where it was called.
- **Parameter**: a value passed into a subroutine when it is called, for the subroutine to use.
- **Defining vs calling**: defining writes the subroutine once; calling runs it, and can be done many times.
- **What happens when a function is called**: describe the steps in time order, as in the diagram: the program reaches the function's name in an expression; any values given are passed to its parameters; control moves to the function and its statements run; `RETURN` sends one value back, which takes the place of the call in the expression, and the program carries on from there.

5. Compare local and global variables (4 questions)

1. For "describe differences", give **pairs**: say what is true of a local variable **and** the matching fact for a global one. Useful pairs:
 - scope: local only within the subroutine it is declared in; global anywhere in the program;
 - who can change it: a local variable can't be altered by code outside its subroutine; a global one can be altered by any part of the program;
 - lifetime: a local variable is created when the subroutine runs and its memory is released when it ends; a global one lasts until the program finishes.
2. To classify a variable in given code, look at **where it is declared**: inside `PROCEDURE ... ENDPROCEDURE` or `FUNCTION ... ENDFUNCTION` means local (parameters too); in the main program means global.

6. Explain why programmers use procedures, functions and parameters (2 questions)

Give reasons, each with its effect. A good answer covers several of these, in your own words:

- **Written once, used many times:** a task needed in several places is called instead of copied, so the program is shorter and there is only one copy to correct or update.
- **Smaller parts:** each subroutine does one clearly named job, so the program is easier to read, test and debug one piece at a time, and a team can share the work.
- **Reuse elsewhere:** a tested subroutine can go into other programs or a library.
- **Parameters** let one subroutine work on different data each time it is called, because the values are passed in rather than fixed inside it.

For a question about maintenance, link each point to how it helps the **next programmer** who has to read or change the code.

7. Trace an algorithm that uses DIV and MOD (1 question)

1. Work out each DIV and MOD separately and write it in the trace table: for 347 and 25, $\text{DIV}(347, 25) = 13$ and $\text{MOD}(347, 25) = 22$, since $347 = 13 \times 25 + 22$.
2. When one result feeds the next line (a remainder that is divided again), use the value just stored, not the original input.
3. Rows for inputs rejected by an earlier check get no DIV/MOD values.

Worked examples

The bold codes show where marks are earned, as in Cambridge mark schemes. The number is how many marks.

- **B1** mark for a correct result on its own, with no method needed.
-

Example 1

A shop's program uses the function `Discount(Price, Percent)`. It returns the price after taking off Percent per cent, rounded to 2 decimal places.

- (a) Write the function in pseudocode. **[4]**
- (b) Write a statement that uses the function to store in `NewPrice` the value of `Cost` after a 15% discount. **[1]**
- (c) State the value returned by `Discount(24.99, 15)`. **[1]**

(a)

```

FUNCTION Discount(Price : REAL, Percent : INTEGER) RETURNS REAL
    RETURN ROUND(Price * (100 - Percent) / 100, 2)
ENDFUNCTION

```

- FUNCTION Discount ... ENDFUNCTION with RETURNS REAL. **B1**
- Both parameters in the header, each with a suitable data type. **B1**
- The discounted price worked out from the parameters and rounded with ROUND (... , 2). **B1**
- The result sent back with RETURN. **B1**

(b) $\text{NewPrice} \leftarrow \text{Discount}(\text{Cost}, 15)$ **B1** (no CALL, and the result is stored).

(c) $24.99 \times 85 \div 100 = 21.2415$, which rounds to **21.24**. **B1**

Example 2

A program for a primary school outputs times tables.

(a) Write a procedure TimesTable with two parameters, Number and Rows. It outputs Rows lines of the times table for Number, each in the form $9 \times 5 = 45$. **[4]**

(b) Write the pseudocode to use TimesTable to output the first 12 lines of the 7 times table. **[2]**

(c) Identify one local variable in your procedure and give one reason why it should be local. **[2]**

(d) State the last line output by CALL TimesTable(9, 5). **[1]**

(a)

```

PROCEDURE TimesTable(Number : INTEGER, Rows : INTEGER)
    DECLARE Row : INTEGER
    FOR Row ← 1 TO Rows
        OUTPUT Number, " x ", Row, " = ", Number * Row
    NEXT Row
ENDPROCEDURE

```

- PROCEDURE TimesTable ... ENDPROCEDURE with both parameters and their types. **B1**
- A loop that runs from 1 to Rows, using the parameter as its end value. **B1**
- The product Number * Row worked out. **B1**
- Each line output in the required form. **B1**

(b) CALL TimesTable(7, 12) earns **B1** for CALL and **B1** for the name with 7 and 12 in the right order (its last line is $7 \times 12 = 84$).

(c) Row is local **B1**. It is only needed while the procedure runs, so keeping it local means code outside can't change it by accident and its memory is freed when the procedure ends. **B1**

(d) $9 \times 5 = 45$ **B1** (the loop's fifth and last pass).

Example 3

(a) State the value of each of these: (i) `ROUND(18.4567, 2)` (ii) `DIV(58, 8)` (iii) `MOD(58, 8)`. **[3]**

(b) Items are packed in boxes. Write the function `BoxesNeeded(Items, Capacity)`, which returns the number of boxes needed for `Items` items when each box holds `Capacity` items. A part-filled box still counts as a box. **[4]**

(c) State the value returned by `BoxesNeeded(58, 8)`. **[1]**

(a) (i) 18.46 **B1** (ii) 7 **B1** (iii) 2, because $58 = 7 \times 8 + 2$ **B1**.

(b) `DIV` counts the full boxes; any remainder means one more box.

```
FUNCTION BoxesNeeded(Items : INTEGER, Capacity : INTEGER) RETURNS INTEGER
    DECLARE Boxes : INTEGER
    Boxes ← DIV(Items, Capacity)
    IF MOD(Items, Capacity) > 0
        THEN
            Boxes ← Boxes + 1
        ENDIF
    RETURN Boxes
ENDFUNCTION
```

- Header with both parameters, their types and `RETURNS INTEGER`, and `ENDFUNCTION`. **B1**
- Full boxes found with `DIV`. **B1**
- `MOD` used to test for items left over, adding one box when there are some. **B1**
- The number of boxes returned. **B1**

(c) 8 **B1** (7 full boxes and one holding the last 2 items). For 56 items it would return 7, because `MOD(56, 8)` is 0.

Example 4

A maths practice program asks five addition questions using random numbers. The global variable `Score` holds the number answered correctly.

(a) Write a function `RandomBetween(Low, High)` that returns a random whole number from `Low` to `High` inclusive. **[3]**

(b) Write a procedure `AskSum(Largest)` that uses `RandomBetween` to make two whole numbers from 1 to `Largest`, outputs the question, inputs the user's answer, and adds 1 to `Score` if it is correct. Declare any local variables. **[5]**

(c) Write the main program: it sets `Score` to 0, uses `AskSum` five times with numbers up to 20, and outputs the score with a message. **[3]**

(d) In one call of `AskSum(20)`, `RANDOM()` returns 0.5 and then 0.25. State the two numbers in the question. **[2]**

(e) Explain why `Score` is global but the user's answer is stored in a local variable. **[2]**

(a)

```
FUNCTION RandomBetween(Low : INTEGER, High : INTEGER) RETURNS INTEGER
    RETURN ROUND(RANDOM() * (High - Low), 0) + Low
ENDFUNCTION
```

- Header with two integer parameters and `RETURNS INTEGER`. **B1**
- `RANDOM()` scaled by `High - Low` and rounded to 0 places. **B1**
- `Low` added and the result returned. **B1**

(b)

```
PROCEDURE AskSum(Largest : INTEGER)
    DECLARE First : INTEGER
    DECLARE Second : INTEGER
    DECLARE Answer : INTEGER
    First ← RandomBetween(1, Largest)
    Second ← RandomBetween(1, Largest)
    OUTPUT "What is ", First, " + ", Second, "?"
    INPUT Answer
    IF Answer = First + Second
        THEN
            Score ← Score + 1
            OUTPUT "Correct"
        ELSE
            OUTPUT "Not quite: the answer is ", First + Second
        ENDIF
    ENDPROCEDURE
```

- `PROCEDURE AskSum(Largest : INTEGER) ... ENDPROCEDURE`. **B1**
- Local variables declared inside the procedure. **B1**
- Both numbers made by using the function in assignments, with 1 and `Largest`. **B1**
- The question output and the answer input. **B1**
- The answer compared with the sum and `Score` increased only when it is correct. **B1**

(c)

```
DECLARE Score : INTEGER
DECLARE Turn : INTEGER
Score ← 0
FOR Turn ← 1 TO 5
```

```

CALL AskSum(20)
NEXT Turn
OUTPUT "You scored ", Score, " out of 5"

```

- Score set to 0 before the questions. **B1**
- A loop that runs five times, with `CALL AskSum(20)` inside. **B1**
- The score output with a message after the loop. **B1**

(d) `RANDOM()` * 19 gives 9.5, which rounds to 10, and $10 + 1 = 11$; then $0.25 \times 19 = 4.75$, which rounds to 5, and $5 + 1 = 6$. The question is "What is $11 + 6$?" **B1 B1** (the correct answer is 17).

(e) `Score` must keep its value across all five calls and be output by the main program, so it has to be visible everywhere **B1**. `Answer` is only needed during one question; as a local variable it is created afresh for each call and nothing outside the procedure can change it by mistake **B1**.

Common mistakes

- **Putting `CALL` in front of a function**, or writing a function call on a line by itself. A function's value has to go somewhere: into an assignment, an `OUTPUT` or a condition. `CALL` is only for procedures.
- **Headers with parts missing**. Each parameter needs its own data type, and a function header needs `RETURNS` and a type. These are separate marks, and examiners often see them left out.
- **Outputting instead of returning**. If the question says a function *returns* a value, the function must use `RETURN`; an `OUTPUT` inside the function earns nothing for that point.
- **Asking for input inside the subroutine** when the value should arrive as a parameter. Use the parameter; that is what it is for.
- **Knowing what `MOD` and `DIV` do but not writing a statement**. When an example in pseudocode is asked for, give a full line such as `Left ← MOD(Pages, 16)`, not a sentence.
- **Vague library-routine answers**: saying `ROUND` "makes a number simpler" or `DIV` "divides". Say what comes back: the whole-number quotient, the value to a stated number of decimal places.
- **Using a programming language's routines** such as `int()`, `%` or `random.randint()` in a pseudocode answer. Use `DIV`, `MOD`, `ROUND` and `RANDOM`.
- **Random ranges that are one out**. `ROUND(RANDOM() * 10, 0)` gives 0 to 10, not 1 to 10. Check what the smallest and largest `RANDOM()` values produce.
- **Muddling the direction of scope**. A variable declared inside a procedure is local, even if it has the same name as one elsewhere; a global one is declared in the main program. When a question asks about "the variable declared in" a subroutine, it is asking about a local variable.
- **Changing the names given**. If the question says `PROCEDURE Label(Word)`, use exactly `Label`, and send values in the order the parameters are listed.

How to get the marks

- **Definitions are marked feature by feature:** the keywords `PROCEDURE`/`FUNCTION` with their end keyword, the name, each parameter with a type, `RETURNS` and a type for a function, the processing, and the `RETURN`. Make each one visible.
- **Calls are marked separately:** the right keyword (or none, for a function), the exact name, and the right values in the right order. For a function, show where the result goes.
- **"State the purpose ... give an example":** one mark for exactly what the routine produces and one for a complete pseudocode statement with numbers in it.
- **"Describe two differences":** each difference needs both halves, local **and** global, about the same feature (scope, lifetime, who can change it).
- **"Explain why"** needs a benefit **and** its effect ("reused by calling it again, so the code isn't repeated"), not a list of words.
- **"Describe what happens when ..."** is about the order of events: give the steps in time order.
- **Pseudocode means pseudocode** except in the 15-mark program; minor syntax slips are ignored, but missing structure (no `ENDFUNCTION`, no `RETURN`) loses marks.

Quick check

1. State the value of: (i) `DIV(83, 9)` (ii) `MOD(83, 9)` (iii) `ROUND(7.349, 1)` (iv) `ROUND(12.96, 0)`.
2. Write a pseudocode statement that stores in `Pick` a random whole number from 10 to 15 inclusive.
3. This pseudocode has two errors. Identify them and correct them.

```
FUNCTION Larger(A : INTEGER, B : INTEGER)
    IF A > B
        THEN
            RETURN A
        ELSE
            RETURN B
    ENDIF
ENDFUNCTION
CALL Larger(4, 9)
```

4. For the program below: (i) name a global variable, a local variable and a parameter; (ii) state what is output; (iii) explain why adding `OUTPUT Doubled` as the last line would not work.

```
DECLARE Total : INTEGER
PROCEDURE AddOn(Amount : INTEGER)
    DECLARE Doubled : INTEGER
    Doubled ← Amount * 2
    Total ← Total + Doubled
ENDPROCEDURE
```

```
Total ← 5  
CALL AddOn(3)  
CALL AddOn(10)  
OUTPUT Total
```

5. A program prints the same receipt footer, six lines long, in four different places. Give two reasons why the programmer should put those lines in a procedure.

Answers

1. (i) 9 (ii) 2, since $83 = 9 \times 9 + 2$ (iii) 7.3 (iv) 13.
2. `Pick ← ROUND(RANDOM() * 5, 0) + 10` (`RANDOM() = 0` gives 10 and 1 gives 15).
3. The header needs a return type: `FUNCTION Larger(A : INTEGER, B : INTEGER) RETURNS INTEGER`. A function isn't called with `CALL` and its value must be used: for example `Biggest ← Larger(4, 9)`, which stores 9.
4. (i) Global: `Total`; local: `Doubled`; parameter: `Amount`. (ii) 31: `Total` starts at 5, the first call adds $3 \times 2 = 6$ and the second adds $10 \times 2 = 20$. (iii) `Doubled` is local to `AddOn`: it exists only while the procedure runs, so the main program can't use it.
5. Any two: the footer is written once and just called four times, so the program is shorter; a change to the footer (a new phone number, say) is made in one place instead of four, so no copy is missed; the procedure can be tested once and trusted everywhere; a well-named `CALL PrintFooter` makes the main program easier to read.

Past questions to try

- **0478/22/O/N/25 Q2**: complete a procedure with a parameter that fills a 2D array, then input a value and call it.
- **0478/23/M/J/25 Q3**: replace swap lines with a procedure call using two index parameters, then compare local and global variables.
- **0478/22/F/M/23 Q10**: write a Boolean function with two parameters that uses `ROUND`, call it, and say how defining differs from calling.
- **0478/23/M/J/23 Q3**: describe the steps when a function is called.