AS 伪代码参考手册

Contents

1 伪代码的书写规范 **3**
- 1.1 关键字、标识符与大小写 . . . 3
- 1.2 缩进与注释 . . . 4

2 数据类型、变量与常量 **6**
- 2.1 六种数据类型 . . . 6
- 2.2 DECLARE 声明与赋值 . . . 7
- 2.3 CONSTANT 常量 . . . 8

3 输入、输出与运算符 **10**
- 3.1 INPUT 与 OUTPUT . . . 10
- 3.2 算术运算、DIV 与 MOD . . . 11
- 3.3 关系运算符与逻辑运算符 . . . 12

4 选择结构 **14**
- 4.1 IF、THEN、ELSE、ENDIF . . . 14
- 4.2 嵌套 IF . . . 15
- 4.3 CASE OF 多分支 . . . 16

5 循环结构 **20**
- 5.1 FOR 循环与 STEP . . . 20
- 5.2 WHILE 前置条件循环 . . . 21
- 5.3 REPEAT 后置条件循环 . . . 22
- 5.4 如何选择循环 . . . 23

6 数组 **25**
- 6.1 一维数组 . . . 25
- 6.2 二维数组 . . . 26
- 6.3 求和、计数、最大值与最小值 . . . 28

7 字符串与内置函数 **31**
- 7.1 用 & 连接字符串 . . . 31
- 7.2 LEFT、RIGHT、MID 与 LENGTH . . . 32
- 7.3 大小写与字符编码 . . . 33
- 7.4 数字与字符串的转换 . . . 35

8 过程与函数 **37**
- 8.1 PROCEDURE 与 CALL . . . 37
- 8.2 FUNCTION 与 RETURN . . . 38
- 8.3 BYVAL 与 BYREF . . . 39
- 8.4 局部变量与全局变量 . . . 41

9 记录与用户自定义类型 **44**
9.1 定义记录类型 44
9.2 记录数组 46
9.3 其他用户自定义类型 47

10 文本文件 **49**
10.1 打开与关闭文件 49
10.2 写入与追加 49
10.3 读取文件到末尾 51

11 标准算法 **54**
11.1 线性查找 54
11.2 冒泡排序 55
11.3 输入验证 58

12 栈、队列与链表 **62**
12.1 用数组实现栈 62
12.2 用数组实现循环队列 64
12.3 用数组实现链表 66

13 速查表 **69**
13.1 关键字一览 69
13.2 从 IGCSE 0478 过渡 71
13.3 如何写好 Paper 2 答案 72

1 伪代码的书写规范

1.1 ■ 关键字、标识符与大小写

伪代码（pseudocode）是一种书写**算法**（algorithm）的方式：它看起来像程序，但不依赖任何一种具体的编程语言。在 9618 考试中，题目里的算法都用伪代码给出，很多答案也必须用伪代码来写。剑桥为此出版了一份官方指南，本手册就以这份指南为准。

拿分并不需要“ 完美 ”的伪代码，但需要阅卷老师**不用猜**就能看懂的伪代码。下面这些规则能帮你做到这一点。

1.1.1 关键字全部大写

关键字（keyword）是含义固定的词，例如 `IF`、`WHILE`、`DECLARE`、`OUTPUT`。关键字一律用 **大写**（upper case）字母书写。

```
DECLARE Total : INTEGER
Total ← 10 + 5
OUTPUT Total
```

这里 `DECLARE`、`INTEGER` 和 `OUTPUT` 是关键字，`Total` 是我们自己起的名字。

1.1.2 标识符

标识符（identifier）是你给变量、常量、过程或函数起的名字。9618 的规则是：

- 必须**以字母开头**（A–Z 或 a–z），不能以数字开头。
- 第一个字母之后，可以使用字母、数字（0–9）和下划线 `_`。
- 不能有空格、带重音符号的字母或其他符号。
- **不能**是关键字。`Output` 或 `Case` 都不能用作变量名。

标识符	是否合法	原因
`TotalScore`	合法	只含字母
`Player2`	合法	数字出现在首字母之后，没问题
`max_value`	合法	9618 允许下划线，也允许小写字母开头
`2ndPlace`	**不合法**	以数字开头
`Total Score`	**不合法**	含有空格
`WHILE`	**不合法**	它是关键字

剑桥书写标识符时采用**混合大小写**（mixed case）：每个新单词的首字母大写，例如 `NumberOfPlayers`。每一份试卷里你都会看到这种写法。

1.1.3 名字要能说明自己的含义

好的名字让你的答案更容易批改。`Average` 能说明这个值是什么，`A` 就不能。只有在约定俗成的地方才适合用单个字母：数组下标用 `i` 和 `j`，坐标用 `X` 和 `Y`。

1.1.4 标识符不区分大小写

Count、COUNT 和 count 被看作**同一个**名字。所以，永远不要使用只有大小写不同的两个名字，例如 CountDown 和 Countdown。阅卷老师会把它们看成同一个变量，你的算法也就不是你想表达的意思了。

```
DECLARE Count : INTEGER
Count ← 3
count ← count + 1
OUTPUT COUNT
```

这段程序输出 4：三种拼写访问的都是同一个变量。它能运行，但风格很差。选定一种拼写，并一直使用它。

常见错误

- 关键字用小写，例如 if 或 output。应写作 IF 和 OUTPUT。
- 用关键字作名字，例如 DECLARE String : STRING。
- 名字中间有空格：应写 FirstName，而不是 First Name。

1.2 ■ 缩进与注释

1.2.1 缩进表示“ 谁在谁里面 ”

缩进（indentation）就是把一行向右移。在伪代码中，缩进表示这一行属于上面那条语句的**内部**，例如属于某个循环或某个 IF。9618 指南通常缩进**三个空格**。

```
DECLARE Counter : INTEGER
FOR Counter ← 1 TO 3
   OUTPUT "Line ", Counter
   IF Counter = 3 THEN
      OUTPUT "Done"
   ENDIF
NEXT Counter
```

- OUTPUT "Line " 和 IF 都在 FOR 循环里面，所以向右缩进 3 个空格。
- OUTPUT "Done" 在 IF 里面，所以再缩进 3 个空格。
- ENDIF 与它的 IF 对齐；NEXT 与它的 FOR 对齐。

即使完全不缩进，程序的运行结果也一样。但阅卷老师要靠排版看出每个代码块从哪里开始、到哪里结束，所以 **一定要缩进**。没有缩进的答案很难看懂，结构不清楚时还可能被扣分。

1.2.2 注释

注释（comment）是写给人看的说明，计算机会忽略它。注释以两条斜线 `//` 开头，一直到这一行结束。

```
// Work out the area of a rectangle
DECLARE Width : INTEGER
DECLARE Height : INTEGER
Width ← 4
Height ← 7
OUTPUT Width * Height   // area in square metres
```

- 注释通常**单独占一行**，写在它所解释的代码前面，缩进与那段代码相同。
- 很短的注释可以写在某一行的末尾，就像上面最后一行那样。
- 多行注释的**每一行**开头都要写 `//`。

1.2.3 行号

有些题目会在左侧印上**行号**（line numbers），方便题目说 “ 第 07 行有错误 ”。行号**不是**伪代码的一部分。改正某一行时，写出行号和新的代码，例如：

```
Line 07:  WHILE Index <= 10
```

1.2.4 练习

找出每一行的错误，然后打开答案核对。

1. `DECLARE 1stName : STRING`

2. `declare Total : INTEGER`

3. `DECLARE Average Mark : REAL`

<details> <summary> 参考答案 </summary>

1. 标识符以数字开头。应改为 `FirstName`。
2. 关键字必须大写：`DECLARE`。
3. 标识符不能含空格。应改为 `AverageMark`。

</details>

常见错误

- 忘记缩进循环体或 `IF` 的内容。
- 代码块在错误的位置结束：`ENDIF` 必须与它自己的 `IF` 对齐。
- 注释只写了一条斜线 `/`。一条斜线表示**除法**；注释要用 `//`。

2 数据类型、变量与常量

2.1 ■ 六种数据类型

每个值都有一种**数据类型**（data type）。类型说明这个值是什么，以及可以对它做什么。9618 伪代码有六种基本类型。

类型	存放的内容	示例值
INTEGER	**整数**（whole number）	5、-3、0
REAL	可以带**小数部分**（fractional part）的数	4.7、0.3、-4.0
CHAR	恰好一个**字符**（character）	'x'、'C'、'@'
STRING	零个或多个字符	"Hello"、""
BOOLEAN	TRUE 或 FALSE	TRUE
DATE	一个日历日期	02/01/2005

2.1.1 各类值的写法

直接写在代码里的值叫作**字面量**（literal）。每种类型都有自己的写法：

- **INTEGER**：普通数字，如 25 或 -3。
- **REAL**：小数点**两边**至少各有一位数字。写 0.5，不写 .5；写 4.0，不写 4.。
- **CHAR**：一个字符，用**单**引号括起来：'A'。
- **STRING**：一串字符，用**双**引号括起来："A"。**空字符串**（empty string）"" 不含任何字符。
- **BOOLEAN**：直接写 TRUE 和 FALSE，不加引号。
- **DATE**：通常写成 dd/mm/yyyy。世界各地的日期写法不同，所以最好说明你用的是哪种格式。

```
DECLARE Price : REAL
DECLARE Initial : CHAR
DECLARE Name : STRING
DECLARE Paid : BOOLEAN
Price ← 12.50
Initial ← 'M'
Name ← "Mei Lin"
Paid ← TRUE
OUTPUT Name, " (", Initial, ") paid ", Price, ": ", Paid
```

2.1.2 该选哪种类型？

- 用来**计数**的东西（人数、进球数、物品数）：INTEGER。
- 需要**测量**、可能带小数的东西（钱、身高、平均值）：REAL。
- 是/否的答案，或者记录某件事是否发生过的**标志**（flag）：BOOLEAN。
- 电话号码或 "007" 这样的编号：STRING。你从不对它做算术，而且用 INTEGER 会丢掉开头的 0。

常见错误

- 用单引号写 `'Hello'`。单引号里只能放**一个**字符；字符串要用双引号。
- 给 `"TRUE"` 加引号。那是一个由四个字母组成的字符串，不是布尔值。
- 把平均值存进 `INTEGER`。平均值通常不是整数，应该用 `REAL`。

2.2 ■ DECLARE 声明与赋值

2.2.1 DECLARE

变量（variable）是内存中一个有名字的位置，存放一个值，这个值在程序运行时可以改变。使用变量之前，你要先**声明**（declare）它：写出它的名字和数据类型。

```
DECLARE <identifier> : <data type>
```

```
DECLARE Counter : INTEGER
DECLARE TotalToPay : REAL
DECLARE GameOver : BOOLEAN
Counter ← 0
TotalToPay ← 0.0
GameOver ← FALSE
OUTPUT Counter, " ", TotalToPay, " ", GameOver
```

把变量声明写在答案的**开头**，每行一个。指南认为声明变量是好习惯，很多评分标准也会为此给分。如果题目要求 “ 声明所有变量 ”，就必须声明。

2.2.2 赋值

赋值（assignment）是把一个值放进变量。赋值运算符是**向左的箭头** `←`。

```
<identifier> ← <value>
```

`Counter ← 0` 读作”Counter **变为** 0” 或” 把 0 赋给 Counter”。先计算右边，再把结果存到左边。所以下面这一行是讲得通的：

```
DECLARE Counter : INTEGER
Counter ← 5
Counter ← Counter + 1
OUTPUT Counter
```

右边的 `Counter + 1` 用的是**旧**值 5，算出的结果 6 再替换掉旧值。

- 左边必须是**一个变量**（或数组的一个元素，或记录的一个字段）。
- 右边可以是任何**表达式**（expression），只要它算出的值与变量类型相同。
- 用键盘输入时，本站也接受 `<-`，并会把它自动换成 `←`。

常见错误

- 用 `Counter = 0` 来赋值。在伪代码中 `=` 用来**比较**两个值，从不用来存值。
- 写成 `0 ← Counter`，左右颠倒。被改变的变量永远在左边。

- 变量还没有值就使用它，例如在 `Total ← 0` 之前就往 `Total` 上加数。一定要先给出初始值，这叫作 **初始化**（initialising）变量。

2.3 ■ CONSTANT 常量

常量（constant）是一个有名字的值，在程序运行期间**永远不变**。

```
CONSTANT <identifier> = <value>
```

```
CONSTANT HourlyRate = 6.50
CONSTANT DefaultText = "N/A"
DECLARE Hours : INTEGER
Hours ← 8
OUTPUT "Pay: ", Hours * HourlyRate
OUTPUT "Comment: ", DefaultText
```

注意这里用的是**等号** `=`，而不是箭头。常量只定义一次，不是被赋值的。

2.3.1 为什么要用常量?

- `HourlyRate` 告诉读者 `6.50` 代表什么；代码中间孤零零的一个数字做不到这一点。
- 如果费率变了，你只需改**一行**，而不用改每一个出现这个数字的地方。

2.3.2 规则

- 把常量写在程序的**开头**。
- 值必须是**字面量**，例如 `6.50` 或 `"N/A"`。不能是变量、另一个常量或一个计算式。`CONSTANT Total = Price * 2` 是不允许的。
- 程序中绝不能给常量赋值。`HourlyRate ← 7.00` 是错误的。

2.3.3 练习

1. 为以下数据写出声明：一个人的年龄、以米为单位的身高、是否是会员、名字的首字母。

2. 某商店的税率固定为 0.08。写出这个常量，然后输出价格为 50 时的税额。

<details> <summary> 参考答案 </summary>

```
DECLARE Age : INTEGER
DECLARE Height : REAL
DECLARE IsMember : BOOLEAN
DECLARE FirstInitial : CHAR
Age ← 17
Height ← 1.72
IsMember ← TRUE
FirstInitial ← 'J'
OUTPUT Age, " ", Height, " ", IsMember, " ", FirstInitial
```

```
CONSTANT TaxRate = 0.08
DECLARE Price : REAL
Price ← 50.0
OUTPUT "Tax: ", Price * TaxRate
```

第二个程序输出 `Tax: 4.0`。

</details>

常见错误

- `CONSTANT TaxRate ← 0.08`。这是 **IGCSE** 的写法。在 9618 中，常量用 `=`。
- 在常量前面写 `DECLARE`。常量只用关键字 `CONSTANT`。

3 输入、输出与运算符

3.1 ■ INPUT 与 OUTPUT

3.1.1 INPUT

INPUT 从用户那里读取一个值，并把它存进一个变量。

```
INPUT <identifier>
```

标识符必须是一个变量，也可以是数组的一个元素或记录的一个字段。

```
DECLARE Age : INTEGER
OUTPUT "How old are you?"
INPUT Age
OUTPUT "Next year you will be ", Age + 1
```

点击“ 运行 ”时，页面会先让你输入数据。

- 先显示一条**提示信息**（prompt）：用一条 OUTPUT 告诉用户该输入什么。评分标准常常为此给分。
- 一条 INPUT 只读取**一个**变量。要读两个值，就写两行 INPUT。

3.1.2 OUTPUT

OUTPUT 在屏幕上显示一个或多个值。多个值之间用**逗号**分隔。

```
OUTPUT <value(s)>
```

```
DECLARE Lives : INTEGER
DECLARE Score : INTEGER
Lives ← 3
Score ← 1250
OUTPUT Score
OUTPUT "You have ", Lives, " lives left"
```

这些值在同一行上一个接一个地输出，中间不会自动加任何东西。所以 "You have " 的末尾留了一个空格。

常见错误

- INPUT "Enter age", Age。INPUT 不会显示消息，应该先写一行 OUTPUT。
- OUTPUT "Total is Total"。引号里的单词只是普通文字。要把变量写在引号外面：OUTPUT "Total is ", Total。
- 用 PRINT 代替 OUTPUT。PRINT 是 Python 的写法，不是剑桥伪代码。

3.2 ■ 算术运算、DIV 与 MOD

3.2.1 算术运算符

运算符	含义	示例	结果
+	加	7 + 2	9
-	减	7 - 2	5
*	乘	7 * 2	14
/	除	7 / 2	3.5
DIV	整数除法	7 DIV 2	3
MOD	除法后的余数	7 MOD 2	1

- / 的结果总是 REAL，即使两个数都是整数：10 / 2 等于 5.0。
- 和数学一样，先算 * 和 /，再算 + 和 -。用**括号**让运算顺序一目了然：(A + B) / 2。

```
DECLARE A : INTEGER
DECLARE B : INTEGER
A ← 10
B ← 4
OUTPUT A + B * 2
OUTPUT (A + B) * 2
OUTPUT A / B
```

输出依次是 18、28 和 2.5。

3.2.2 DIV 与 MOD

DIV 给出**商**（quotient）：一个数里面能完整地包含另一个数多少次。MOD 给出**余数**（remainder）：剩下的部分。在 9618 中，这两个运算符都写在两个数**中间**，就像 + 一样。

```
DECLARE Minutes : INTEGER
Minutes ← 135
OUTPUT Minutes DIV 60, " hours and ", Minutes MOD 60, " minutes"
```

135 分钟就是 2 小时（$2 \times 60 = 120$），还剩 15 分钟。

下面三种用法你会反复遇到：

- **判断奇偶**：偶数的 Number MOD 2 = 0 为 TRUE。
- **取最后一位数字**：对正整数，Number MOD 10 给出它的个位数。
- **去掉最后一位数字**：Number DIV 10。所以 472 DIV 10 等于 47。

```
DECLARE Number : INTEGER
DECLARE DigitSum : INTEGER
Number ← 472
DigitSum ← 0
WHILE Number > 0
   DigitSum ← DigitSum + (Number MOD 10)
   Number ← Number DIV 10
ENDWHILE
OUTPUT DigitSum
```

这段程序把各位数字相加：$4 + 7 + 2$，输出 13。

常见错误

- DIV(7, 2) 或 MOD(7, 2)。这是 **IGCSE** 的写法。在 9618 中要写 7 DIV 2。
- 需要整数时却用了 /，例如用作数组下标。7 / 2 等于 3.5，不是合法的下标；应该用 7 DIV 2。

3.3 ■ 关系运算符与逻辑运算符

3.3.1 关系运算符

关系运算符（relational operator）比较两个值，结果永远是 BOOLEAN：TRUE 或 FALSE。

运算符	含义
=	等于
<>	不等于
<	小于
>	大于
<=	小于或等于
>=	大于或等于

```
DECLARE Mark : INTEGER
Mark ← 65
OUTPUT Mark >= 50
OUTPUT Mark = 100
OUTPUT Mark <> 0
```

输出依次是 TRUE、FALSE 和 TRUE。

3.3.2 逻辑运算符

逻辑运算符（logic operators）AND、OR 和 NOT 用来连接或改变布尔值。

表达式	何时为 TRUE
A AND B	A 和 B **都**为 TRUE
A OR B	A 和 B **至少一个**为 TRUE
NOT A	A 为 FALSE

判断一个范围要用 AND 连接**两个**完整的比较：

```
DECLARE Age : INTEGER
DECLARE HasTicket : BOOLEAN
Age ← 15
HasTicket ← FALSE
OUTPUT Age >= 13 AND Age <= 19
OUTPUT Age < 12 OR HasTicket
OUTPUT NOT HasTicket
```

输出依次是 TRUE、FALSE 和 TRUE。

AND 和 OR 混在一起时，要加**括号**让意思清楚：(Day = "Sat" OR Day = "Sun") AND NOT Raining。

3.3.3 练习

1. 29 DIV 4 等于多少？29 MOD 4 等于多少？

2. 写出一个表达式，当 Temp 在 18 到 25 这个范围**之外**时为 TRUE。

<details> <summary> 参考答案 </summary>

1. 29 DIV 4 等于 7，29 MOD 4 等于 1，因为 $4 \times 7 = 28$，还剩 1。
2. Temp < 18 OR Temp > 25。(也可以写 NOT (Temp >= 18 AND Temp <= 25)。)

```
DECLARE Temp : INTEGER
Temp ← 27
OUTPUT 29 DIV 4, " ", 29 MOD 4
OUTPUT Temp < 18 OR Temp > 25
```

</details>

常见错误

- IF 13 <= Age <= 19。每个比较都要有自己的变量：Age >= 13 AND Age <= 19。
- Age >= 13 AND <= 19。后半部分缺了左边的值，要再写一次 Age。
- 用 != 或 == 表示" 不等于" 和" 等于"。那是 Python 和 C 的写法；伪代码用 <> 和 =。

4 选择结构

4.1 ■ IF、THEN、ELSE、ENDIF

选择（selection）是指程序根据一个**条件**（condition）来决定下一步做什么。条件是一个结果为 TRUE 或 FALSE 的表达式。

4.1.1 不带 ELSE 的 IF

```
IF <condition> THEN
   <statement(s)>
ENDIF
```

```
DECLARE Temperature : INTEGER
Temperature ← 31
IF Temperature > 30 THEN
   OUTPUT "Hot day"
   OUTPUT "Drink water"
ENDIF
OUTPUT "Have a good day"
```

如果条件为 TRUE，就执行 THEN 和 ENDIF 之间的语句；如果为 FALSE，就跳过它们。无论哪种情况，程序之后都会继续执行 ENDIF 后面的内容。

4.1.2 带 ELSE 的 IF

```
IF <condition> THEN
   <statement(s)>
ELSE
   <statement(s)>
ENDIF
```

```
DECLARE Mark : INTEGER
Mark ← 41
IF Mark >= 50 THEN
   OUTPUT "Pass"
ELSE
   OUTPUT "Fail"
ENDIF
```

两个**分支**（branches）中恰好执行**一个**：不会两个都执行，也不会两个都不执行。

4.1.3 9618 的排版

- THEN 写在 IF **这一行的末尾**。
- ELSE 和 ENDIF 与 IF 对齐。
- 每个分支里的语句都要缩进。

常见错误

- 忘记写 ENDIF。每个 IF 都需要一个。
- IF Mark ← 50 THEN。箭头用来存值；条件中的比较要用 =。
- 把 THEN 单独写在 IF 的下一行。那是 **IGCSE 0478** 的排版；在 9618 中它写在 IF 这一行上。

4.2 ■ 嵌套 IF

放在另一个 IF 里面的 IF 叫作**嵌套**（nested）IF。当结果多于两种，或者第二个问题只有在第一个问题之后才有意义时，就使用它。

```
DECLARE Mark : INTEGER
Mark ← 64
IF Mark >= 70 THEN
   OUTPUT "Distinction"
ELSE
   IF Mark >= 50 THEN
      OUTPUT "Merit"
   ELSE
      OUTPUT "Fail"
   ENDIF
ENDIF
```

- 内层 IF 位于外层的 ELSE 里面，并且有它**自己**的 ENDIF。
- 数一数：两个 IF 就需要两个 ENDIF。
- **先判断最高的分界线。**内层的 Mark >= 50 只有在分数已经低于 70 时才会执行，所以不需要再写 AND Mark < 70。

剑桥伪代码中**没有** ELSEIF 或 ELIF 关键字。要先写 ELSE，再在里面写一个新的 IF，就像上面那样。

4.2.1 在第一个问题里再问第二个问题

有时内层 IF 位于 THEN 分支里：

```
DECLARE ChallengerScore : INTEGER
DECLARE ChampionScore : INTEGER
DECLARE HighestScore : INTEGER
ChallengerScore ← 95
ChampionScore ← 80
HighestScore ← 90
IF ChallengerScore > ChampionScore THEN
   IF ChallengerScore > HighestScore THEN
      OUTPUT "New champion and new highest score"
   ELSE
      OUTPUT "New champion"
   ENDIF
ELSE
   OUTPUT "The champion stays"
ENDIF
```

常见错误

- 漏掉内层的 ENDIF。这样后面的每一行看起来都属于错误的代码块。
- 先判断最低的分界线：如果先写 IF Mark >= 50，85 分会输出 Merit，永远到不了 Distinction 的判断。
- 写了 ELSE IF，却只用一个 ENDIF 结束。每个 IF 都要分别结束。

4.3 ■ CASE OF 多分支

CASE OF 根据**一个变量的值**选择一个分支。当你反复判断同一个变量时，它比许多层嵌套的 IF 更清楚。

```
CASE OF <identifier>
   <value 1> : <statement(s)>
   <value 2> : <statement(s)>
   OTHERWISE : <statement(s)>
ENDCASE
```

```
DECLARE Move : CHAR
DECLARE Position : INTEGER
Position ← 50
Move ← 'A'
CASE OF Move
   'W' : Position ← Position - 10
   'S' : Position ← Position + 10
   'A' : Position ← Position - 1
   'D' : Position ← Position + 1
   OTHERWISE : OUTPUT "Unknown key"
ENDCASE
OUTPUT Position
```

- 每一行是：一个**值**、一个冒号 :、然后是要做的事。
- 各个情况**按顺序**判断。第一个匹配的情况被执行，然后整个 CASE 就结束了，其余情况不再判断。
- OTHERWISE 可以不写。如果写了，它必须放在**最后**，在没有任何值匹配时执行。在 9618 中它要带冒号：OTHERWISE :。

4.3.1 用 TO 表示范围

在 9618 中，一个情况可以是一个**范围**（range）：<value1> TO <value2>，两端都包括在内。

```
DECLARE Mark : INTEGER
Mark ← 64
CASE OF Mark
   70 TO 100 : OUTPUT "A"
   60 TO 69 : OUTPUT "B"
   50 TO 59 : OUTPUT "C"
   OTHERWISE : OUTPUT "U"
ENDCASE
```

4.3.2 一个情况里有多条语句

一个情况可以执行不止一条语句。把它们写在接下来的几行，并缩进：

```
DECLARE Choice : INTEGER
DECLARE Balance : REAL
Balance ← 100.0
Choice ← 2
CASE OF Choice
   1 : OUTPUT "Balance: ", Balance
   2 :
      Balance ← Balance - 20.0
      OUTPUT "Took out 20. Balance: ", Balance
   OTHERWISE : OUTPUT "Not a menu choice"
ENDCASE
```

4.3.3 用 IF 还是 CASE?

- 一个变量与**固定的值或范围**比较：CASE OF 往往更清楚。
- 条件涉及**多个变量**，或用到 AND / OR：必须用 IF。

4.3.4 练习

1. Day 存放 1 到 7 之间的一个数。用 CASE OF 实现：1 到 5 输出 "Weekday"，6 或 7 输出 "Weekend"，其他值输出 "Error"。

2. 用嵌套 IF 实现：0–12 岁输出 "Child"，13–19 岁输出 "Teen"，20 岁及以上输出 "Adult"。

<details> <summary> 参考答案 </summary>

```
DECLARE Day : INTEGER
Day ← 6
CASE OF Day
   1 TO 5 : OUTPUT "Weekday"
   6 TO 7 : OUTPUT "Weekend"
   OTHERWISE : OUTPUT "Error"
ENDCASE
```

```
DECLARE Age : INTEGER
Age ← 15
IF Age <= 12 THEN
   OUTPUT "Child"
ELSE
   IF Age <= 19 THEN
      OUTPUT "Teen"
   ELSE
      OUTPUT "Adult"
   ENDIF
ENDIF
```

</details>

常见错误

- 在 `CASE OF` 中写条件，例如 `Mark >= 50 :`。一个情况只能是一个值或一个范围，绝不能是比较式。
- 把 `OTHERWISE` 放在最后以外的位置。
- 忘记写 `ENDCASE`。
- `OTHERWISE` 后面不加冒号。那是 IGCSE 的写法；9618 写作 `OTHERWISE :`。

5 循环结构

5.1 ■ FOR 循环与 STEP

迭代（iteration）就是重复执行一组语句。负责重复的结构叫作**循环**（loop）。伪代码有三种循环，每一种适合不同的任务。

5.1.1 计数控制循环

如果在循环开始之前就知道**要重复多少次**，就用 FOR。它叫作**计数控制循环**（count-controlled loop）。

```
FOR <identifier> ← <value1> TO <value2>
   <statement(s)>
NEXT <identifier>
```

```
DECLARE Counter : INTEGER
FOR Counter ← 1 TO 5
   OUTPUT Counter, " x 3 = ", Counter * 3
NEXT Counter
```

- **循环计数器**（loop counter，这里是 Counter）必须是 INTEGER。
- 它依次取 value1 到 value2 之间的**每一个**值，**两端都包括**。所以 1 TO 5 执行五次。
- 如果 value1 = value2，循环体执行一次；如果 value1 > value2，循环体一次也不执行。
- 在 NEXT 后面写上计数器的名字是好习惯，循环套循环时尤其如此。
- 不要在循环里自己修改计数器，FOR 循环会负责这件事。

5.1.2 STEP

加上 STEP 可以按 1 以外的步长计数。步长可以是**负数**，用来倒数。

```
FOR <identifier> ← <value1> TO <value2> STEP <increment>
```

```
DECLARE i : INTEGER
FOR i ← 0 TO 20 STEP 5
   OUTPUT i
NEXT i
FOR i ← 3 TO 1 STEP -1
   OUTPUT "Countdown ", i
NEXT i
```

一旦下一个值会**超过** value2，循环就停止。所以 FOR i ← 1 TO 10 STEP 4 只给出 1、5 和 9。

5.1.3 循环中的循环

一个循环可以放在另一个循环里面。外层循环每执行一遍，内层循环就**完整地**执行一次。

```
DECLARE Row : INTEGER
DECLARE Column : INTEGER
DECLARE Line : STRING
FOR Row ← 1 TO 3
   Line ← ""
   FOR Column ← 1 TO 4
      Line ← Line & NUM_TO_STR(Row * Column) & " "
   NEXT Column
   OUTPUT Line
NEXT Row
```

（NUM_TO_STR 把数字变成字符串，这样才能用 & 连接；见第 7 章。）外层循环执行 3 次，每次内层循环执行 4 次，一共 12 遍。

常见错误

- FOR Counter = 1 TO 5。FOR 这一行要用箭头 ←。
- 倒数时没写 STEP -1。FOR i ← 10 TO 1 一次也不会执行。
- 内外层循环结束的顺序写反。**最后**打开的循环必须**最先**结束。

5.2 ■ WHILE 前置条件循环

如果**不知道**要重复多少次，而且循环可能需要执行**零**次，就用 WHILE。

```
WHILE <condition>
   <statement(s)>
ENDWHILE
```

```
DECLARE Number : INTEGER
Number ← 40
WHILE Number > 9
   Number ← Number - 9
ENDWHILE
OUTPUT Number
```

- 每一遍开始**之前**先检查条件。所以它叫作**前置条件循环**（pre-condition loop）。
- 条件为 TRUE **时**就继续重复，条件变为 FALSE 时停止。
- 如果第一次检查时条件就是 FALSE，循环体**一次也不执行**。

5.2.1 循环里必须有东西在变化

循环体必须改变条件所检查的内容。否则条件永远是 TRUE，就会形成**无限循环**(infinite loop)。

```
DECLARE Balance : REAL
DECLARE Years : INTEGER
Balance ← 1000.0
Years ← 0
WHILE Balance < 2000.0
   Balance ← Balance * 1.1
   Years ← Years + 1
ENDWHILE
OUTPUT "Doubled after ", Years, " years"
```

这里 Balance 每一遍都在增长，所以循环一定会结束。

常见错误

- 忘记写 ENDWHILE。
- 循环之前没有给变量赋值。条件是先检查的，所以变量必须已经有值。
- 无限循环：循环体里没有任何东西改变条件。

5.3 ■ REPEAT 后置条件循环

如果循环体必须**至少执行一次**，就用 REPEAT，例如先让用户输入，再检查输入是否正确。

```
REPEAT
   <statement(s)>
UNTIL <condition>
```

```
DECLARE Mark : INTEGER
REPEAT
   OUTPUT "Enter a mark from 0 to 100"
   INPUT Mark
UNTIL Mark >= 0 AND Mark <= 100
OUTPUT "Mark saved: ", Mark
```

- 每一遍执行**之后**才检查条件：这是**后置条件循环**（post-condition loop）。
- 循环一直重复，**直到**条件为 TRUE。条件一变成 TRUE，循环就停止。
- 没有 ENDREPEAT：UNTIL 这一行就是循环的结束。

注意两种条件的意思正好相反：WHILE 在条件为 TRUE 时继续；REPEAT 在条件为 TRUE 时停止。

```
DECLARE Total : INTEGER
DECLARE Count : INTEGER
Total ← 0
Count ← 0
REPEAT
   Count ← Count + 1
   Total ← Total + Count
UNTIL Total > 20
OUTPUT "Added 1 to ", Count, " to pass 20: total ", Total
```

常见错误

- 条件写反。”年龄不对就一直问”要写成 UNTIL 年龄**正确**。
- 循环体可能需要执行零次时却用了 REPEAT。这时应该用 WHILE。

5.4 ■ 如何选择循环

循环	何时使用	何时检查	至少执行
FOR	开始之前就知道要执行几遍	（按计数）	0 次
WHILE	不知道次数，而且可能需要执行**零**次	每遍**之前**	0 次
REPEAT	不知道次数，而且至少需要执行**一**次	每遍**之后**	1 次

考试中的典型情况：

- 遍历数组的每个元素：FOR。
- 读文件直到结束：WHILE NOT EOF(...)，因为文件可能是空的。
- 反复要求输入，直到输入有效：REPEAT ... UNTIL。
- 在数组中查找，找到后提前停止：带标志的 WHILE（或 REPEAT）。

5.4.1 练习

1. 用一个 FOR 循环输出 2 到 20 之间的所有偶数。

2. 从 1 开始，不断把一个数翻倍。用 WHILE 循环输出要翻倍多少次才能超过 1000。

3. 把下面的 WHILE 循环改写成输出相同的 REPEAT 循环：

```
Count ← 1
WHILE Count <= 3
   OUTPUT Count
   Count ← Count + 1
ENDWHILE
```

<details> <summary> 参考答案 </summary>

```
DECLARE Number : INTEGER
FOR Number ← 2 TO 20 STEP 2
   OUTPUT Number
NEXT Number
```

```
DECLARE Value : INTEGER
DECLARE Doublings : INTEGER
Value ← 1
Doublings ← 0
WHILE Value <= 1000
   Value ← Value * 2
   Doublings ← Doublings + 1
ENDWHILE
OUTPUT Doublings
```

输出 10，因为 2 的 10 次方是 1024。

```
DECLARE Count : INTEGER
Count ← 1
REPEAT
   OUTPUT Count
   Count ← Count + 1
UNTIL Count > 3
```

WHILE 的条件 `Count <= 3` 变成了它的反面 `Count > 3`。两个循环之所以输出相同，只是因为这里的 WHILE 循环体至少会执行一次。

</details>

6 数组

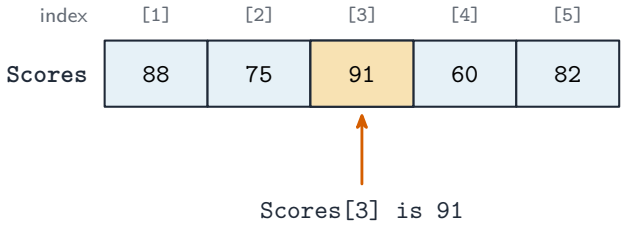
6.1 ■ 一维数组

数组(array)用一个名字存放许多个**同一数据类型**的值。每个值是一个**元素**(element),通过它的 **索引**（index），也就是它的位置编号，来访问。数组的大小是固定的，不会改变。

```
DECLARE <identifier> : ARRAY[<lower>:<upper>] OF <data type>
```

```
DECLARE Scores : ARRAY[1:5] OF INTEGER
Scores[1] ← 88
Scores[2] ← 75
Scores[3] ← 91
Scores[4] ← 60
Scores[5] ← 82
OUTPUT "The third score is ", Scores[3]
```

- `[1:5]` 给出**下界**（lower bound）1 和**上界**（upper bound）5，所以这个数组有 5 个元素。
- 剑桥通常从 **1** 开始。一定要写出下界，因为有些语言从 0 开始。
- 索引写在**方括号**里：`Scores[3]`。
- 索引可以是任何结果为整数的表达式：`Scores[i]`、`Scores[n + 1]`。



Scores[1:5]：五个格子，一个名字，每个格子通过索引访问

6.1.1 用循环访问每个元素

用 FOR 循环的计数器作索引，就能依次访问每个元素。这是 Paper 2 中最常见的写法。

```
DECLARE Names : ARRAY[1:4] OF STRING
DECLARE i : INTEGER
Names[1] ← "Ali"
Names[2] ← "Bo"
Names[3] ← "Chen"
Names[4] ← "Dara"
FOR i ← 1 TO 4
   OUTPUT i, ": ", Names[i]
NEXT i
```

你**不能**用一条语句给一组元素赋值。`Names[1 TO 4] ← ""` 是不允许的，要用循环：

```
DECLARE Names : ARRAY[1:30] OF STRING
DECLARE Index : INTEGER
FOR Index ← 1 TO 30
   Names[Index] ← ""
NEXT Index
OUTPUT "All 30 names cleared"
```

但你**可以**用一行把整个数组复制给另一个大小和类型都相同的数组：`SavedGame ← Board`。

常见错误

- 索引超出范围，例如对 `ARRAY[1:5]` 使用 `Scores[6]` 或 `Scores[0]`。这是运行时错误。
- 用圆括号：`Scores(3)`。数组要用方括号。
- 忘记写数据类型：每个数组都需要 `OF <data type>`。

6.2 ■ 二维数组

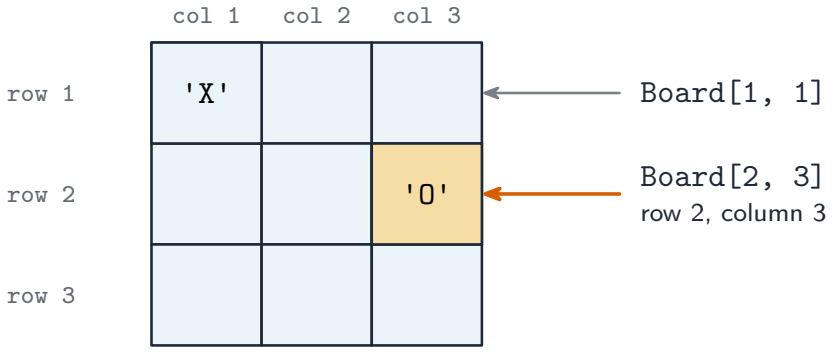
二维（two-dimensional）数组就像一张表格，有**行**和**列**。访问一个元素需要两个索引。

```
DECLARE <identifier> : ARRAY[<lower1>:<upper1>, <lower2>:<upper2>] OF
↪  <data type>
```

```
DECLARE Board : ARRAY[1:3, 1:3] OF CHAR
Board[1, 1] ← 'X'
Board[2, 3] ← 'O'
OUTPUT "Row 2, column 3 holds ", Board[2, 3]
```

- 按惯例，第一个索引是**行**，第二个索引是**列**。

- 两个索引写在**同一对**方括号里，用逗号隔开：`Board[2, 3]`，而不是 `Board[2][3]`。



Board[1:3, 1:3]：第一个索引选行，第二个索引选列

6.2.1 用嵌套循环处理二维数组

要访问每个元素，用一个循环处理行，在它里面再用一个循环处理列。

```
DECLARE Amount : ARRAY[1:2, 1:3] OF INTEGER
DECLARE Row : INTEGER
DECLARE Column : INTEGER
DECLARE RowTotal : INTEGER
Amount[1, 1] ← 4
Amount[1, 2] ← 7
Amount[1, 3] ← 1
Amount[2, 1] ← 9
Amount[2, 2] ← 2
Amount[2, 3] ← 5
FOR Row ← 1 TO 2
   RowTotal ← 0
   FOR Column ← 1 TO 3
      RowTotal ← RowTotal + Amount[Row, Column]
   NEXT Column
   OUTPUT "Total for row ", Row, " is ", RowTotal
NEXT Row
```

`RowTotal ← 0` 写在外层循环**里面**，所以每一行都从零开始重新累加。

常见错误

- 把两个索引写反：数组按”先行后列”建立，却写成 `Board[Column, Row]`。
- 把 `RowTotal ← 0` 写在外层循环**之前**。这样第二行的总和里也包含了第一行。

6.3 ■ 求和、计数、最大值与最小值

大多数数组题目考的都是下面几种固定写法。把它们当作“ 模板 ”来记。

6.3.1 求和与平均值

```
DECLARE Marks : ARRAY[1:5] OF INTEGER
DECLARE Total : INTEGER
DECLARE i : INTEGER
Marks[1] ← 56
Marks[2] ← 71
Marks[3] ← 48
Marks[4] ← 90
Marks[5] ← 65
Total ← 0
FOR i ← 1 TO 5
   Total ← Total + Marks[i]
NEXT i
OUTPUT "Total: ", Total
OUTPUT "Average: ", Total / 5
```

循环**之前**把**总和**（total）设为 0，在循环里累加，循环**之后**再做除法。

6.3.2 统计满足条件的元素个数

```
DECLARE Marks : ARRAY[1:5] OF INTEGER
DECLARE PassCount : INTEGER
DECLARE i : INTEGER
Marks[1] ← 56
Marks[2] ← 71
Marks[3] ← 48
Marks[4] ← 90
Marks[5] ← 65
PassCount ← 0
FOR i ← 1 TO 5
   IF Marks[i] >= 60 THEN
      PassCount ← PassCount + 1
   ENDIF
NEXT i
OUTPUT PassCount, " marks are 60 or more"
```

6.3.3 最大值与最小值

从**第一个元素**开始，而不是从 0 开始。然后把其余每个元素都与它比较。

```
DECLARE Marks : ARRAY[1:5] OF INTEGER
DECLARE Highest : INTEGER
DECLARE Lowest : INTEGER
DECLARE i : INTEGER
Marks[1] ← 56
Marks[2] ← 71
Marks[3] ← 48
Marks[4] ← 90
Marks[5] ← 65
Highest ← Marks[1]
Lowest ← Marks[1]
FOR i ← 2 TO 5
   IF Marks[i] > Highest THEN
      Highest ← Marks[i]
   ENDIF
   IF Marks[i] < Lowest THEN
      Lowest ← Marks[i]
   ENDIF
NEXT i
OUTPUT "Highest ", Highest, ", lowest ", Lowest
```

为什么 `Lowest` 不从 0 开始？因为没有分数低于 0，`Lowest` 就会一直是 0，结果是错的。从 `Marks[1]` 开始永远正确，即使数组里有负数也一样。

6.3.4 练习

1. 数组 `Temps[1:7]` 存放一周的气温。输出气温最高的那一天的编号（不是气温本身）。

<details> <summary> 参考答案 </summary>

```
DECLARE Temps : ARRAY[1:7] OF INTEGER
DECLARE HottestDay : INTEGER
DECLARE Day : INTEGER
Temps[1] ← 21
Temps[2] ← 25
Temps[3] ← 19
Temps[4] ← 28
Temps[5] ← 24
Temps[6] ← 28
Temps[7] ← 22
HottestDay ← 1
FOR Day ← 2 TO 7
   IF Temps[Day] > Temps[HottestDay] THEN
      HottestDay ← Day
   ENDIF
NEXT Day
OUTPUT "Hottest day: ", HottestDay
```

输出 4。要记录目前最佳元素的**索引**，而不是它的值。第 6 天也是 28 度，但 `>` 不会用相等的值去替换，所以保留的是第一个最热的日子。

</details>

常见错误

- 把 `Total ← 0` 写在循环里面。这样每一遍都会重新清零。
- 求平均值时除错了数。要除以实际累加的值的个数。
- 题目要的是**第一个**最大值时却用了 `>=`。遇到相等的值时 `>=` 会再次替换，结果得到的是**最后一个**最大值。

7 字符串与内置函数

7.1 ■ 用 & 连接字符串

连接（concatenate）是指把字符串首尾相接。在 9618 中，连接运算符是 `&`。

```
DECLARE FirstName : STRING
DECLARE LastName : STRING
DECLARE FullName : STRING
FirstName ← "Summer"
LastName ← "Pudding"
FullName ← FirstName & " " & LastName
OUTPUT FullName
OUTPUT LENGTH(FullName)
```

- `&` 生成**一个新的字符串**。你可以把它存起来、传给函数，或者输出它。
- 各部分之间不会自动加任何东西。要空格，就自己连接一个 `" "`。
- `&` 用来连接字符串和字符。要连接数字，先用 `NUM_TO_STR` 把它变成字符串（见 7.4 节）。

7.1.1 在循环中拼出一个字符串

从空字符串开始，每一遍往后面加一点：

```
DECLARE Stars : STRING
DECLARE i : INTEGER
Stars ← ""
FOR i ← 1 TO 5
   Stars ← Stars & "*"
NEXT i
OUTPUT Stars
```

7.1.2 用 & 还是逗号?

- 在 `OUTPUT` 中，逗号把各个值依次输出：`OUTPUT "Age: ", Age`。这些值可以是任何类型。
- `&` 生成一个新的**字符串值**，例如用来存进变量或写入文件。

常见错误

- `"Total: " & 25`。25 是数字；应写 `"Total: " & NUM_TO_STR(25)`。
- 像 Python 那样用 `+` 连接字符串。伪代码用 `&`。

7.2 ■ LEFT、RIGHT、MID 与 LENGTH

考试中，你可以使用的字符串函数**总会印在随卷附带的插页（insert）里**。但你仍然要知道它们的用法，因为考试时必须又快又准地使用它们。字符串中的位置从 **1** 开始。

函数	返回	示例	结果
`LENGTH(S)`	S 中的字符个数	`LENGTH("Happy Days")`	`10`
`LEFT(S, x)`	最左边的 x 个字符	`LEFT("ABCDEFGH", 3)`	`"ABC"`
`RIGHT(S, x)`	最右边的 x 个字符	`RIGHT("ABCDEFGH", 3)`	`"FGH"`
`MID(S, x, y)`	从第 x 个位置开始的 y 个字符	`MID("ABCDEFGH", 2, 3)`	`"BCD"`

- `LENGTH` 也会计算空格和标点。
- `MID` 的第三个值是**长度**，不是结束位置。`MID("ABCDEFGH", 2, 3)` 从 B 开始，取 3 个字符。
- **子串**（substring）是字符串的一部分。`LEFT`、`RIGHT` 和 `MID` 返回的都是子串。



MID(“ABCDEFGH”, 2, 3)：从第 *2* 个位置开始，取 *3* 个字符

7.2.1 逐个访问字符

MID(S, i, 1) 就是第 i 个位置上的字符。用一个从 1 到 LENGTH(S) 的循环，就能访问所有字符。几乎每道字符串题都会用到这种写法。

```
DECLARE Word : STRING
DECLARE i : INTEGER
DECLARE Vowels : INTEGER
DECLARE Letter : CHAR
Word ← "EDUCATION"
Vowels ← 0
FOR i ← 1 TO LENGTH(Word)
   Letter ← MID(Word, i, 1)
   IF Letter = 'A' OR Letter = 'E' OR Letter = 'I' OR Letter = 'O' OR
   ↪  Letter = 'U' THEN
      Vowels ← Vowels + 1
   ENDIF
NEXT i
OUTPUT Word, " has ", Vowels, " vowels"
```

7.2.2 把字符串倒过来

```
DECLARE Word : STRING
DECLARE Reversed : STRING
DECLARE i : INTEGER
Word ← "STRESSED"
Reversed ← ""
FOR i ← LENGTH(Word) TO 1 STEP -1
   Reversed ← Reversed & MID(Word, i, 1)
NEXT i
OUTPUT Reversed
```

常见错误

- 从位置 0 开始。第一个字符在位置 **1**。
- 把 MID(S, 2, 5) 理解为” 从 2 到 5”。它的意思是” 从位置 2 开始取 5 个字符”。
- 写 SUBSTRING(S, 2, 3)。那是 IGCSE 的函数名；9618 中的函数是 MID。
- 要取的字符比实际的多：LEFT("Hi", 5) 会出错。

7.3 ■ 大小写与字符编码

7.3.1 大写与小写

函数	返回	示例	结果
`TO_UPPER(x)`	把 x 中的字母全部变成大写；x 可以是 STRING 或 CHAR	`TO_UPPER("Hello")`	`"HELLO"`
`TO_LOWER(x)`	把 x 中的字母全部变成小写	`TO_LOWER("Hello")`	`"hello"`
`UCASE(c)`	把一个 CHAR 变成大写	`UCASE('h')`	`'H'`
`LCASE(c)`	把一个 CHAR 变成小写	`LCASE('W')`	`'w'`

数字、空格等非字母字符不会改变。Paper 2 插页里印的是 `TO_UPPER` 和 `TO_LOWER`，它们可以处理整个字符串。

一个常见用法：**不区分大小写**地比较两个字符串。

```
DECLARE Answer : STRING
Answer ← "yEs"
IF TO_UPPER(Answer) = "YES" THEN
   OUTPUT "You said yes"
ENDIF
```

7.3.2 字符编码：ASC 与 CHR

每个字符都以一个数字的形式存储，这个数字就是它的**字符编码**（character code）。在 ASCII 中，`'A'` 是 65，`'B'` 是 66，……`'Z'` 是 90，`'a'` 是 97。

函数	返回	示例	结果
`ASC(c)`	字符 c 的编码	`ASC('A')`	65
`CHR(x)`	编码为 x 的字符	`CHR(66)`	`'B'`

```
DECLARE Letter : CHAR
DECLARE Code : INTEGER
Letter ← 'C'
Code ← ASC(Letter)
OUTPUT Letter, " has code ", Code
OUTPUT "The next letter is ", CHR(Code + 1)
```

有了编码，下面两件事就很容易：

- **是不是大写字母？** `Letter >= 'A' AND Letter <= 'Z'` 能这样判断，是因为编码是按顺序排列的。
- **字母移位**（如凯撒密码）：`CHR(ASC(Letter) + 3)`。

常见错误

- 在 9618 答案中对整个字符串用 `UCASE("hello")`。`UCASE` 只处理一个字符；字符串要用 `TO_UPPER`。

- 忘记 'a' 和 'A' 的编码不同，所以 'a' = 'A' 为 FALSE。

7.4 ■ 数字与字符串的转换

7.4.1 互相转换

函数	返回	示例	结果
NUM_TO_STR(x)	把数字 x 变成字符串	NUM_TO_STR(87.5)	"87.5"
STR_TO_NUM(x)	把字符串 x 变成数字	STR_TO_NUM("23")	23
IS_NUM(S)	如果 S 是合法的数字，返回 TRUE	IS_NUM("12a")	FALSE

从文件读入或由用户输入的数据，往往是字符串。转换之前先用 IS_NUM 检查，否则程序可能出错。

```
DECLARE Entry : STRING
DECLARE Value : INTEGER
Entry ← "42"
IF IS_NUM(Entry) THEN
   Value ← STR_TO_NUM(Entry)
   OUTPUT "Double: ", Value * 2
ELSE
   OUTPUT "Not a number"
ENDIF
OUTPUT "Code " & NUM_TO_STR(7) & "B"
```

7.4.2 INT 与 RAND

函数	返回	示例	结果
INT(x)	x 的整数部分；小数部分被**直接去掉**	INT(27.9)	27
RAND(x)	一个从 0 到 x 的随机 REAL，**不包括** x	RAND(87)	例如 35.43

INT **不是**四舍五入：INT(27.9) 是 27，不是 28。要四舍五入到整数，先加 0.5：INT(X + 0.5)。

要得到一个 1 到 6 之间的随机整数，就像掷骰子：

```
DECLARE Dice : INTEGER
Dice ← INT(RAND(6)) + 1
OUTPUT "You rolled ", Dice
```

RAND(6) 给出一个从 0 到 6（不包括 6）的值。INT 把它变成 0 到 5，再 + 1 就得到 1 到 6。

7.4.3 日期

DATE 类型的值在插页中有自己的函数，例如 DAY(ThisDate)、MONTH(ThisDate)、YEAR(ThisDate) 和 SETDATE(Day, Month, Year)。具体有哪些，请看你那份试卷的插页，上面对每个函数都有说明。

7.4.4 练习

1. 像 "AB1234" 这样的产品编号由两个字母和四个数字组成。写出伪代码：如果最后四个字符是数字，就输出 "Valid"，否则输出 "Invalid"。

2. 输出 "Grace Brewster Hopper" 的首字母缩写，也就是每个单词的第一个字母。

<details> <summary> 参考答案 </summary>

```
DECLARE Code : STRING
Code ← "AB1234"
IF LENGTH(Code) = 6 AND IS_NUM(RIGHT(Code, 4)) THEN
   OUTPUT "Valid"
ELSE
   OUTPUT "Invalid"
ENDIF
```

```
DECLARE FullName : STRING
DECLARE Initials : STRING
DECLARE i : INTEGER
FullName ← "Grace Brewster Hopper"
Initials ← LEFT(FullName, 1)
FOR i ← 2 TO LENGTH(FullName)
   IF MID(FullName, i - 1, 1) = " " THEN
      Initials ← Initials & MID(FullName, i, 1)
   ENDIF
NEXT i
OUTPUT Initials
```

输出 GBH。如果一个字母**前面**的字符是空格，它就是一个新单词的开头。

</details>

常见错误

- 对字符串做算术："5" + 1。要先用 STR_TO_NUM 转换。
- 以为 INT 会四舍五入。它总是直接去掉小数部分。
- 把 RAND(6) 当骰子用。它会给出 3.81 这样的小数，还可能给出 0；应该用 INT(RAND(6)) + 1。

8 过程与函数

8.1 ■ PROCEDURE 与 CALL

子程序（subroutine）是一段有名字、完成一项任务的代码。写一次，就可以按需要多次使用。伪代码有两种子程序：过程和函数。

过程（procedure）用来完成一项任务，它不返回值。

```
PROCEDURE <identifier>()
   <statement(s)>
ENDPROCEDURE

PROCEDURE <identifier>(<param1> : <data type>, <param2> : <data type>
↪  ...)
   <statement(s)>
ENDPROCEDURE
```

用 CALL 来运行过程：

```
CALL <identifier>()
CALL <identifier>(<value1>, <value2> ...)
```

```
PROCEDURE DrawLine(LineLength : INTEGER, Symbol : CHAR)
   DECLARE Line : STRING
   DECLARE i : INTEGER
   Line ← ""
   FOR i ← 1 TO LineLength
      Line ← Line & Symbol
   NEXT i
   OUTPUT Line
ENDPROCEDURE

PROCEDURE ShowTitle()
   CALL DrawLine(12, '=')
   OUTPUT "  MAIN MENU"
   CALL DrawLine(12, '=')
ENDPROCEDURE

CALL ShowTitle()
CALL DrawLine(5, '*')
```

- **参数**（parameter）是写在过程头部、用来接收值的变量。LineLength 和 Symbol 就是 DrawLine 的参数。
- 在 CALL 中传进去的值叫作**实参**（argument）。实参的**顺序**和**类型**必须与参数一致。
- 没有参数的过程也要写空括号：ShowTitle()。

- 一条 CALL 本身就是一条完整的语句，单独占一行。

8.1.1 为什么要用子程序?

- 同样的代码不必一遍又一遍地写。
- 每个子程序都可以单独编写和测试。
- 主程序变得简短、易读。

常见错误

- 调用过程时不写 CALL：应写 `CALL DrawLine(5, '*')`，而不是只写 `DrawLine(5, '*')`。
- 在过程头部漏写参数的数据类型。
- 忘记写 ENDPROCEDURE。

8.2 ■ FUNCTION 与 RETURN

函数（function）和过程很像，但它会把**一个值返回**到调用它的地方。函数头部要写明返回值的数据类型。

```
FUNCTION <identifier>(<param1> : <data type> ...) RETURNS <data type>
   <statement(s)>
ENDFUNCTION
```

```
FUNCTION Max(Number1 : INTEGER, Number2 : INTEGER) RETURNS INTEGER
   IF Number1 > Number2 THEN
      RETURN Number1
   ELSE
      RETURN Number2
   ENDIF
ENDFUNCTION

DECLARE Distance : INTEGER
Distance ← 8
OUTPUT "Penalty fine = ", Max(10, Distance * 2)
```

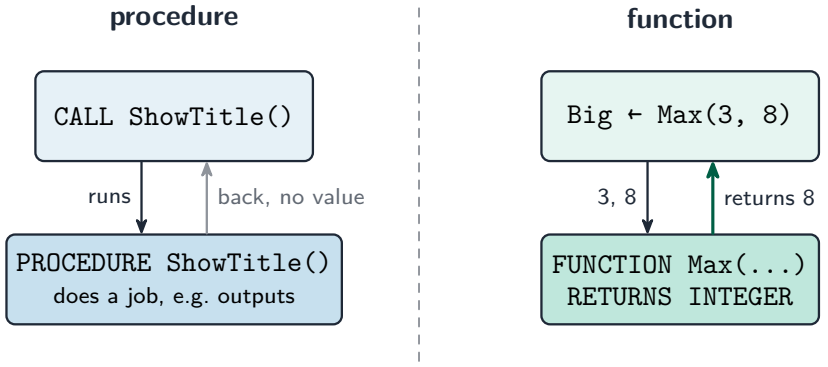
- RETURN 把值送回去。RETURN 一执行，函数就**立刻结束**，后面的语句都会被跳过。
- 函数调用本身**不是**一条语句，要把它用在**表达式里**：赋值、OUTPUT、条件，或者作为另一个调用的实参。
- 调用函数时**绝不能**用 CALL。

```
FUNCTION IsEven(Number : INTEGER) RETURNS BOOLEAN
   RETURN Number MOD 2 = 0
ENDFUNCTION

DECLARE Value : INTEGER
FOR Value ← 1 TO 4
   IF IsEven(Value) THEN
      OUTPUT Value, " is even"
   ENDIF
NEXT Value
```

8.2.1 用过程还是函数?

- 子程序要算出调用者需要的**一个值**：用**函数**。
- 子程序是去**做**一件事（输出、修改数组、写文件），没有唯一的答案：用**过程**。



过程被调用去完成一项任务；函数返回一个值供使用

常见错误

- `CALL Max(3, 4)`。函数要用在表达式里：`Biggest ← Max(3, 4)`。
- 函数头部漏写 `RETURNS <data type>`，或者函数体里漏写 `RETURN` 语句。
- 题目要求函数**返回**一个值，你却在函数里把答案输出了。

8.3 ■ BYVAL 与 BYREF

把值传给参数有两种方式。

- **按值传递**（by value，`BYVAL`）：子程序得到值的一个**副本**。子程序修改参数时，调用者的变量**不会**改变。
- **按引用传递**（by reference，`BYREF`）：子程序得到的是调用者的**那个变量本身**。子程序里的修改，调用结束后调用者**能**看到。

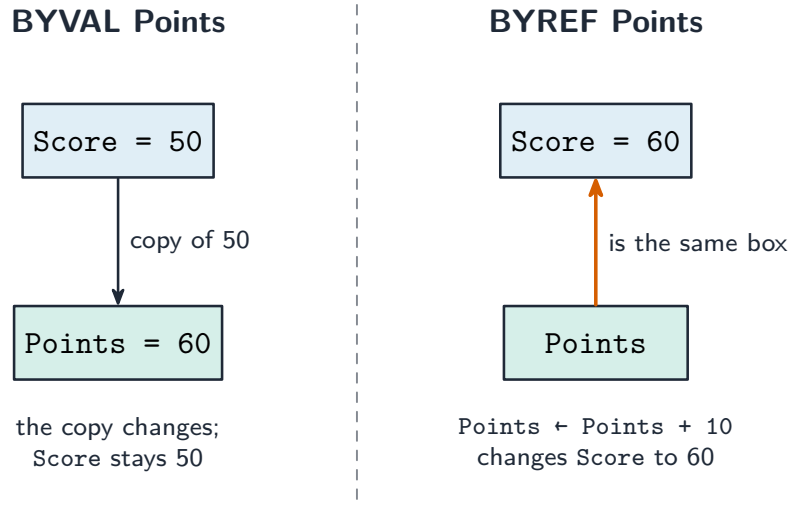
把关键字写在头部的参数前面。如果两个都不写，参数就**按值**传递。

```
PROCEDURE AddBonus(BYVAL Points : INTEGER)
   Points ← Points + 10
ENDPROCEDURE

PROCEDURE AddBonusRef(BYREF Points : INTEGER)
   Points ← Points + 10
ENDPROCEDURE

DECLARE Score : INTEGER
Score ← 50
CALL AddBonus(Score)
OUTPUT "After BYVAL: ", Score
CALL AddBonusRef(Score)
OUTPUT "After BYREF: ", Score
```

第一个 `OUTPUT` 显示 50（只有副本变了），第二个显示 60。



BYVAL 给过程一个副本；*BYREF* 给过程调用者自己的变量

8.3.1 一个关键字管多个参数

如果连续几个参数用同一种方式传递，关键字只写一次。下面 X 和 Y **都**是 BYREF：

```
PROCEDURE Swap(BYREF X : INTEGER, Y : INTEGER)
   DECLARE Temp : INTEGER
   Temp ← X
   X ← Y
   Y ← Temp
ENDPROCEDURE

DECLARE A : INTEGER
DECLARE B : INTEGER
A ← 3
B ← 8
CALL Swap(A, B)
OUTPUT "A = ", A, ", B = ", B
```

8.3.2 什么时候用 BYREF

- 过程需要送回**不止一个**结果。
- 过程需要修改调用者的变量，就像 Swap 那样。

不要在**函数**中按引用传递参数。函数用 RETURN 送回它的那一个值。

常见错误

- 以为修改普通（BYVAL）参数后，外面也能看到变化。它只是一个副本。
- 在 FUNCTION 头部使用 BYREF。
- 在 IGCSE 0478 的答案里写 BYREF。那个考纲里没有它。

8.4 ■ 局部变量与全局变量

变量的**作用域**（scope）是程序中可以使用这个变量的范围。

- **局部变量**（local variable）声明在子程序**内部**。它只在这个子程序运行期间存在，也只有这个子程序能用它。参数也是局部的。
- **全局变量**（global variable）声明在**主程序**中，在所有子程序之外。程序的任何部分都能使用它。

```
DECLARE Total : INTEGER

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

Total ← 0
CALL AddToTotal(5)
CALL AddToTotal(3)
OUTPUT "Total: ", Total
```

- `Total` 是**全局**的：过程可以修改它，主程序能看到结果 16。
- `Doubled` 和 `Amount` 是 `AddToTotal` 的**局部**变量，主程序不能使用它们。

8.4.1 尽量使用局部变量

- 只使用参数和局部变量的子程序，可以单独测试、重复使用。
- 两个子程序都可以有一个叫 `i` 的局部变量，互不冲突。
- 如果许多子程序都去修改同一个全局变量，出了错就很难找。

所以：用**参数**把数据传进去，用 `RETURN`（或 `BYREF`）把结果送回来，全局变量只留给真正属于整个程序的少数几个值。

8.4.2 练习

1. 写一个函数 `Area(Width : REAL, Height : REAL) RETURNS REAL`，然后用它输出一个 3.5×2.0 的长方形的面积。

2. 写一个过程 `MinMax`，接收三个整数，通过两个 `BYREF` 参数送回最小值和最大值。

<details> <summary> 参考答案 </summary>

```
FUNCTION Area(Width : REAL, Height : REAL) RETURNS REAL
   RETURN Width * Height
ENDFUNCTION

OUTPUT "Area: ", Area(3.5, 2.0)
```

```
PROCEDURE MinMax(A : INTEGER, B : INTEGER, C : INTEGER, BYREF Smallest :
↪  INTEGER, Largest : INTEGER)
   Smallest ← A
   Largest ← A
   IF B < Smallest THEN
      Smallest ← B
   ENDIF
   IF C < Smallest THEN
      Smallest ← C
   ENDIF
   IF B > Largest THEN
      Largest ← B
   ENDIF
   IF C > Largest THEN
      Largest ← C
   ENDIF
ENDPROCEDURE

DECLARE Low : INTEGER
DECLARE High : INTEGER
CALL MinMax(7, 2, 9, Low, High)
OUTPUT "Smallest ", Low, ", largest ", High
```

A、B 和 C 按值传递（默认方式）。Smallest 前面的 BYREF 同样适用于 Largest，因为关键字没有重复书写。

</details>

9 记录与用户自定义类型

9.1 ■ 定义记录类型

数组存放许多个**同一种**类型的值。但我们常常需要把**不同**类型的数据放在一起：一个学生的姓名（STRING）、分数（INTEGER）和班级（CHAR）。**记录**（record）就能做到这一点。记录里的每一项叫作一个**字段**（field）。

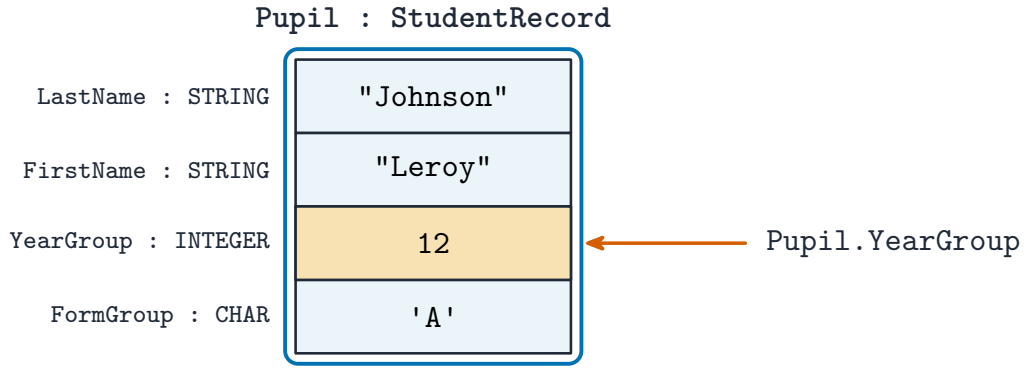
记录是一种**用户自定义数据类型**（user-defined data type）：你先设计好它，然后像使用 INTEGER 一样，声明这种类型的变量。

```
TYPE <identifier>
   DECLARE <field1> : <data type>
   DECLARE <field2> : <data type>
   ...
ENDTYPE
```

```
TYPE StudentRecord
   DECLARE LastName : STRING
   DECLARE FirstName : STRING
   DECLARE YearGroup : INTEGER
   DECLARE FormGroup : CHAR
ENDTYPE

DECLARE Pupil : StudentRecord
Pupil.LastName ← "Johnson"
Pupil.FirstName ← "Leroy"
Pupil.YearGroup ← 12
Pupil.FormGroup ← 'A'
OUTPUT Pupil.FirstName, " ", Pupil.LastName, " is in ", Pupil.YearGroup,
 ↪  Pupil.FormGroup
```

- TYPE … ENDTYPE 只是**描述**记录的样子，并不会创建变量。
- DECLARE Pupil : StudentRecord 才创建一个这种新类型的变量。
- 用**点号表示法**（dot notation）访问某个字段：<variable>.<field>，例如 Pupil.YearGroup。
- 字段可以用在普通变量能用的任何地方：INPUT、OUTPUT、条件和赋值。



记录把不同类型的字段组合在一起；用点号访问每个字段

9.1.1 复制整条记录

同一记录类型的两个变量可以用一行来赋值，每个字段都会被复制。

```
TYPE Point
   DECLARE X : INTEGER
   DECLARE Y : INTEGER
ENDTYPE

DECLARE Start : Point
DECLARE Copy : Point
Start.X ← 3
Start.Y ← 7
Copy ← Start
Copy.X ← 10
OUTPUT "Start is (", Start.X, ", ", Start.Y, ")"
OUTPUT "Copy is (", Copy.X, ", ", Copy.Y, ")"
```

之后修改 Copy.X 不会改变 Start：它们是两条独立的记录。

常见错误

- 用**类型**名代替变量名：StudentRecord.LastName ← "Lee"。应写 Pupil.LastName。
- 用 OUTPUT Pupil 显示整条记录。要分别输出你需要的每个字段。
- 在 TYPE … ENDTYPE 里漏写每个字段前面的 DECLARE。

9.2 ■ 记录数组

元素是记录的数组是一种非常常见的数据结构：它就像一张表格，每一行是一条记录，每一列是一个字段。

```
TYPE Book
   DECLARE Title : STRING
   DECLARE Pages : INTEGER
   DECLARE OnLoan : BOOLEAN
ENDTYPE

DECLARE Library : ARRAY[1:3] OF Book
DECLARE i : INTEGER
Library[1].Title ← "Dune"
Library[1].Pages ← 412
Library[1].OnLoan ← FALSE
Library[2].Title ← "Emma"
Library[2].Pages ← 474
Library[2].OnLoan ← TRUE
Library[3].Title ← "Holes"
Library[3].Pages ← 233
Library[3].OnLoan ← FALSE

FOR i ← 1 TO 3
   IF Library[i].OnLoan = FALSE THEN
      OUTPUT Library[i].Title, " (", Library[i].Pages, " pages) is
      ↪  available"
   ENDIF
NEXT i
```

- 先写索引，再写字段：`Library[i].Title`。
- 从左往右读：“ 在 `Library` 中，第 `i` 个元素的 `Title` 字段 ”。

9.2.1 用记录代替并行数组

如果没有记录，就需要三个共用同一个索引的独立数组（`Titles`、`Pages`、`OnLoan`），这叫作**并行数组**（parallel arrays）。它们也能用，但很容易只给一个数组排了序，却忘了其他数组。记录数组则让每本书的数据始终在一起。

常见错误

- 写成 `Library.Title[i]`。索引属于数组：应写 `Library[i].Title`。
- 在定义 `TYPE` 之前就声明使用它的数组。要先定义类型。

9.3 ■ 其他用户自定义类型

9618 指南还介绍了其他几种用户自定义类型。它们属于 **A-Level（Paper 3）** 的内容，但出现在同一份指南里，认识它们会有帮助。这些例子在本页上不能运行。

9.3.1 枚举类型

枚举类型（enumerated type）列出这种类型的变量可以取的全部值。

```
TYPE Season = (Spring, Summer, Autumn, Winter)
DECLARE ThisSeason : Season
ThisSeason ← Spring
```

这些值是名字，**不是**字符串：`Spring` 两边没有引号。

9.3.2 指针与集合

- **指针**（pointer）类型存放一个内存地址。`TYPE TIntPointer = ^INTEGER` 声明这种类型；`^` 写在变量前面表示取它的地址，写在指针后面表示取它所指向的值。
- **集合**（set）类型存放一组没有顺序的值：`TYPE LetterSet = SET OF CHAR`。

AS 阶段请把重点放在**记录**和**记录数组**上：它们在 Paper 2 的题目中反复出现。

9.3.3 练习

1. 定义一个记录类型 `Car`，字段为 `Registration`（字符串）、`Year`（整数）和 `Price`（实数）。声明一个含 3 辆车的数组并填入数据，然后输出所有价格低于 5000 的车的车牌号。

<details> <summary> 参考答案 </summary>

```
TYPE Car
   DECLARE Registration : STRING
   DECLARE Year : INTEGER
   DECLARE Price : REAL
ENDTYPE

DECLARE Cars : ARRAY[1:3] OF Car
DECLARE i : INTEGER
Cars[1].Registration ← "AB12 CDE"
Cars[1].Year ← 2015
Cars[1].Price ← 4500.0
Cars[2].Registration ← "FG34 HIJ"
Cars[2].Year ← 2019
Cars[2].Price ← 9800.0
Cars[3].Registration ← "KL56 MNO"
Cars[3].Year ← 2012
Cars[3].Price ← 2950.0
FOR i ← 1 TO 3
   IF Cars[i].Price < 5000.0 THEN
      OUTPUT Cars[i].Registration
   ENDIF
NEXT i
```

输出 AB12 CDE 和 KL56 MNO。

</details>

10 文本文件

10.1 ■ 打开与关闭文件

程序结束后，变量里的值就消失了。**文本文件**（text file）把数据保存在存储设备上，下次运行时还能再读出来。文本文件是由若干**行**组成的列表，每一行是一个字符串。各行按**顺序**一行接一行地读或写。

10.1.1 OPENFILE

使用文件之前，必须用某一种**文件模式**（file mode）把它打开：

```
OPENFILE <file identifier> FOR <file mode>
```

模式	作用
READ	从文件中读取各行，从第一行开始
WRITE	新建一个**空**文件并写入；文件中原有的数据会**丢失**
APPEND	在文件**末尾**添加新行；原有数据保留

文件标识符是文件名，可以是 "Scores.txt" 这样的字符串，也可以是存放文件名的 STRING 变量。

10.1.2 CLOSEFILE

```
CLOSEFILE <file identifier>
```

用完每个文件后都要关闭它。一个文件**同一时间只能用一种模式**打开：要读取刚写完的文件，先关闭它，再用 FOR READ 重新打开。

```
DECLARE FileName : STRING
FileName ← "Log.txt"
OPENFILE FileName FOR WRITE
WRITEFILE FileName, "Program started"
CLOSEFILE FileName
OUTPUT "Written to ", FileName
```

常见错误

- 没有先打开文件就使用它。
- 忘记写 CLOSEFILE，在 Paper 2 中这常常值一分。
- 想保留原有的行，却用 WRITE 打开。WRITE 会清空文件；应该用 APPEND。

10.2 ■ 写入与追加

WRITEFILE 向一个以 WRITE 或 APPEND 模式打开的文件写入**一行**。

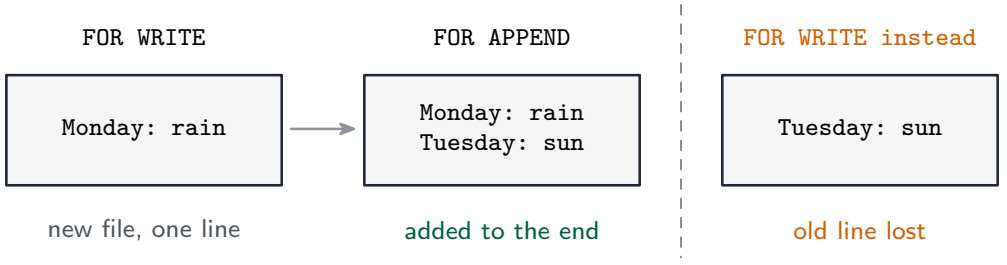
```
WRITEFILE <file identifier>, <data>
```

```
DECLARE Line : STRING
OPENFILE "Diary.txt" FOR WRITE
WRITEFILE "Diary.txt", "Monday: rain"
CLOSEFILE "Diary.txt"

OPENFILE "Diary.txt" FOR APPEND
WRITEFILE "Diary.txt", "Tuesday: sun"
CLOSEFILE "Diary.txt"

OPENFILE "Diary.txt" FOR READ
WHILE NOT EOF("Diary.txt")
   READFILE "Diary.txt", Line
   OUTPUT Line
ENDWHILE
CLOSEFILE "Diary.txt"
```

两行都会输出，因为 APPEND 保留了第一行。把 APPEND 改成 WRITE 再运行一次：只剩下星期二那一行。



WRITE 从空文件开始；*APPEND* 在末尾添加

10.2.1 一行存放多个值

一行就是一个字符串。要存放多个值，就用**分隔符**（separator），例如逗号，把它们连接成一个字符串：

```
DECLARE Name : STRING
DECLARE Score : INTEGER
Name ← "Ravi"
Score ← 87
OPENFILE "Results.txt" FOR WRITE
WRITEFILE "Results.txt", Name & "," & NUM_TO_STR(Score)
CLOSEFILE "Results.txt"
OUTPUT "Saved: ", Name & "," & NUM_TO_STR(Score)
```

10.2.2 保存整个数组

在循环中每行写入一个元素：

```
DECLARE Colours : ARRAY[1:3] OF STRING
DECLARE i : INTEGER
Colours[1] ← "red"
Colours[2] ← "green"
Colours[3] ← "blue"
OPENFILE "Colours.txt" FOR WRITE
FOR i ← 1 TO 3
   WRITEFILE "Colours.txt", Colours[i]
NEXT i
CLOSEFILE "Colours.txt"
OUTPUT "3 lines written"
```

把 OPENFILE 和 CLOSEFILE 放在循环**外面**。如果在循环里用 FOR WRITE 打开文件，每一遍都会把文件清空。

10.3 ■ 读取文件到末尾

READFILE 从一个以 FOR READ 模式打开的文件中读取**下一行**，并把它存进一个变量。这个变量应该是 STRING 类型。

```
READFILE <file identifier>, <variable>
```

函数 EOF 告诉你什么时候已经没有内容可读了。EOF 是**文件结束**（end of file）的缩写：

```
EOF(<file identifier>)
```

没有剩余的行时（包括文件为空时），EOF 返回 TRUE，否则返回 FALSE。

10.3.1 标准的读文件循环

你通常不知道文件有多少行，而且它可能是空的。所以要用 WHILE 循环，在每次读取**之前**检查 EOF：

```
DECLARE Line : STRING
DECLARE Count : INTEGER
OPENFILE "Names.txt" FOR WRITE
WRITEFILE "Names.txt", "Amir"
WRITEFILE "Names.txt", ""
WRITEFILE "Names.txt", "Bea"
CLOSEFILE "Names.txt"

Count ← 0
OPENFILE "Names.txt" FOR READ
WHILE NOT EOF("Names.txt")
   READFILE "Names.txt", Line
   IF Line <> "" THEN
      Count ← Count + 1
      OUTPUT Line
   ENDIF
ENDWHILE
CLOSEFILE "Names.txt"
OUTPUT Count, " names read"
```

第一部分只是为了让例子能运行而创建文件。第二部分才是要记住的模板：**打开，当未到文件末尾时循环，读取，处理，关闭**。

10.3.2 把一个文件复制到另一个文件

这是指南中的原例：复制每一行，但把空行换成一串短横线。

```
DECLARE LineOfText : STRING
OPENFILE "FileA.txt" FOR READ
OPENFILE "FileB.txt" FOR WRITE
WHILE NOT EOF("FileA.txt")
   READFILE "FileA.txt", LineOfText
   IF LineOfText = "" THEN
      WRITEFILE "FileB.txt", "-------------"
   ELSE
      WRITEFILE "FileB.txt", LineOfText
   ENDIF
ENDWHILE
CLOSEFILE "FileA.txt"
CLOSEFILE "FileB.txt"
```

两个文件可以同时打开，各用各的模式。

10.3.3 练习

1. 文件 Marks.txt 每行存放一个整数分数。写出伪代码，读取这个文件并输出平均分。

<details> <summary> 参考答案 </summary>

```
DECLARE Line : STRING
DECLARE Total : INTEGER
DECLARE Count : INTEGER
// Create a sample file so that the program can run
OPENFILE "Marks.txt" FOR WRITE
WRITEFILE "Marks.txt", "60"
WRITEFILE "Marks.txt", "75"
WRITEFILE "Marks.txt", "90"
CLOSEFILE "Marks.txt"

Total ← 0
Count ← 0
OPENFILE "Marks.txt" FOR READ
WHILE NOT EOF("Marks.txt")
   READFILE "Marks.txt", Line
   Total ← Total + STR_TO_NUM(Line)
   Count ← Count + 1
ENDWHILE
CLOSEFILE "Marks.txt"
IF Count > 0 THEN
   OUTPUT "Average: ", Total / Count
ELSE
   OUTPUT "The file is empty"
ENDIF
```

每一行都是字符串，所以要用 STR_TO_NUM 把它变成数字。IF Count > 0 可以防止文件为空时出现除以零。

</details>

常见错误

- 用 REPEAT … UNTIL EOF(…)。它在检查**之前**就先读一次，遇到空文件会出错。
- 用 FOR 循环，并猜测文件的行数。
- 对读入的一行直接做算术，没有先转换。
- 读取以 FOR WRITE 打开的文件，或者向以 FOR READ 打开的文件写入。

11 标准算法

11.1 ■ 线性查找

线性查找（linear search）从第一个元素开始，**逐个**检查数组中的元素，直到找到目标或到达末尾为止。无论数组是否排过序，它都适用。

11.1.1 一找到就停止

好的查找会提前停止。用一个 WHILE 循环，加上一个找到目标时变为 TRUE 的**标志**：

```
DECLARE Names : ARRAY[1:6] OF STRING
DECLARE SearchName : STRING
DECLARE Index : INTEGER
DECLARE Found : BOOLEAN
Names[1] ← "Ola"
Names[2] ← "Raj"
Names[3] ← "Kim"
Names[4] ← "Tom"
Names[5] ← "Eva"
Names[6] ← "Ivy"
SearchName ← "Tom"

Found ← FALSE
Index ← 1
WHILE Found = FALSE AND Index <= 6
   IF Names[Index] = SearchName THEN
      Found ← TRUE
   ELSE
      Index ← Index + 1
   ENDIF
ENDWHILE

IF Found THEN
   OUTPUT SearchName, " found at position ", Index
ELSE
   OUTPUT SearchName, " not found"
ENDIF
```

- 找到目标**或者**没有剩余元素时，循环就停止。
- 只有在**没有**找到时才增加 Index，所以循环结束后它仍然指向目标。
- 把 SearchName 改成 "Zoe" 再运行一次，看看”没找到”的情况。

11.1.2 写成函数

在 Paper 2 中，查找常常写成一个函数：返回目标的位置，找不到时返回 -1 这样的特殊值：

```
DECLARE Data : ARRAY[1:5] OF INTEGER

FUNCTION FindValue(Target : INTEGER) RETURNS INTEGER
   DECLARE i : INTEGER
   FOR i ← 1 TO 5
      IF Data[i] = Target THEN
         RETURN i
      ENDIF
   NEXT i
   RETURN -1
ENDFUNCTION

Data[1] ← 14
Data[2] ← 3
Data[3] ← 27
Data[4] ← 8
Data[5] ← 19
OUTPUT FindValue(27)
OUTPUT FindValue(5)
```

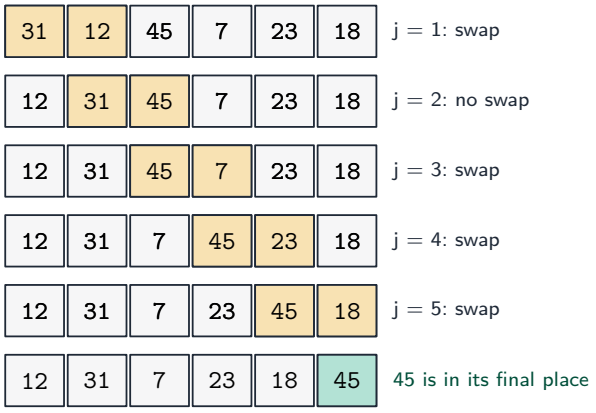
`RETURN i` 会立刻离开函数，所以 `FOR` 循环不会走完。只有在什么都没匹配上时，才会执行到 `RETURN -1`。

常见错误

- 在循环**里面**写 `ELSE OUTPUT "Not found"`。这样每遇到一个不匹配的元素就输出一次"Not found"。“ 没找到 ” 只能在循环**结束之后**判断。
- 循环条件永远不会停止：一定要加上 `Index <= <` 上界 `>` 的判断。

11.2 ■ 冒泡排序

冒泡排序（bubble sort）把数组元素按顺序排好。它从头到尾比较每一对**相邻的**元素，如果顺序不对就交换。完整地走完一**遍**（pass）后，最大的值就像气泡一样 “ 冒 ” 到了末尾。每走一遍，就有一个值被放到它最终的位置。



冒泡排序的一遍：比较相邻元素，顺序不对就交换

11.2.1 改进版

最好的写法是：一旦某一遍**没有发生交换**，就立即停止，因为这时数组已经排好了。它还会每一遍少检查一个元素。

```
DECLARE Numbers : ARRAY[1:6] OF INTEGER
DECLARE Top : INTEGER
DECLARE j : INTEGER
DECLARE Temp : INTEGER
DECLARE Swapped : BOOLEAN
Numbers[1] ← 31
Numbers[2] ← 12
Numbers[3] ← 45
Numbers[4] ← 7
Numbers[5] ← 23
Numbers[6] ← 18

Top ← 5
REPEAT
   Swapped ← FALSE
   FOR j ← 1 TO Top
      IF Numbers[j] > Numbers[j + 1] THEN
         Temp ← Numbers[j]
         Numbers[j] ← Numbers[j + 1]
         Numbers[j + 1] ← Temp
         Swapped ← TRUE
      ENDIF
   NEXT j
   Top ← Top - 1
UNTIL Swapped = FALSE OR Top = 0

FOR j ← 1 TO 6
   OUTPUT Numbers[j]
NEXT j
```

分四部分来读：

1. **比较相邻元素**：`Numbers[j]` 与 `Numbers[j + 1]`。循环只走到 `Top`，所以 `j + 1` 永远不会越界。
2. 借助一个临时变量来**交换**。没有 `Temp`，第一个值就会丢失。
3. **记住**发生过交换：`Swapped ← TRUE`。
4. 某一遍没有交换时就**停止**。`Top ← Top - 1` 跳过已经到位的值。

要按**降序**（从大到小）排序，把 `>` 改成 `<` 即可。

11.2.2 对记录排序

对记录数组排序时，比较的是某一个**字段**，交换的却是**整条记录**：

```
IF Students[j].Mark > Students[j + 1].Mark THEN
   Temp ← Students[j]
   Students[j] ← Students[j + 1]
   Students[j + 1] ← Temp
ENDIF
```

这里 `Temp` 必须声明为该记录类型。

常见错误

- 对 6 个元素的数组写 `FOR j ← 1 TO 6`：这样 `Numbers[j + 1]` 会变成不存在的 `Numbers[7]`。内层循环要比数组大小少一。
- 不用临时变量就交换：先写 `Numbers[j] ← Numbers[j + 1]` 再写 `Numbers[j + 1] ← Numbers[j]`，两个位置会变成同一个值。
- 把 `Swapped ← FALSE` 写在 `REPEAT` **外面**。每一遍开始时都必须重新设置它。

11.3 ■ 输入验证

验证（validation）是在程序使用输入之前，自动检查这个输入是否**合理**。它不能证明数据是正确的，只能说明数据是合理的。

检查	它要问的问题	示例
范围检查（range check）	值是否在上下限之间？	分数在 0 到 100 之间
长度检查（length check）	字符个数是否在允许的范围内？	密码至少 8 个字符
类型检查（type check）	数据的种类对不对？	年龄是一个数字
存在检查（presence check）	是否输入了内容？	姓名不为空
格式检查（format check）	是否符合规定的模式？	编号是 2 个字母加 4 个数字

11.3.1 验证循环

一遍又一遍地询问，直到输入通过检查。REPEAT … UNTIL 很合适，因为至少要问一次：

```
DECLARE Entry : STRING
DECLARE Mark : INTEGER
DECLARE Valid : BOOLEAN
REPEAT
   OUTPUT "Enter a mark from 0 to 100"
   INPUT Entry
   Valid ← FALSE
   IF IS_NUM(Entry) THEN
      Mark ← STR_TO_NUM(Entry)
      IF Mark >= 0 AND Mark <= 100 THEN
         Valid ← TRUE
      ENDIF
   ENDIF
   IF Valid = FALSE THEN
      OUTPUT "Invalid. Try again."
   ENDIF
UNTIL Valid
OUTPUT "Mark accepted: ", Mark
```

这段程序先做**类型检查**（IS_NUM），再做**范围检查**。类型检查必须放在前面，因为对 "abc" 这样的文字做范围检查毫无意义。

11.3.2 验证函数

规则变长时，就把它们放进一个返回 TRUE 或 FALSE 的函数：

```
FUNCTION ValidPassword(Password : STRING) RETURNS BOOLEAN
   DECLARE i : INTEGER
   DECLARE HasDigit : BOOLEAN
   DECLARE Ch : CHAR
   IF LENGTH(Password) < 8 THEN
      RETURN FALSE
   ENDIF
   HasDigit ← FALSE
   FOR i ← 1 TO LENGTH(Password)
      Ch ← MID(Password, i, 1)
      IF Ch >= '0' AND Ch <= '9' THEN
         HasDigit ← TRUE
      ENDIF
   NEXT i
   RETURN HasDigit
ENDFUNCTION

OUTPUT ValidPassword("sunshine")
OUTPUT ValidPassword("sunshine7")
OUTPUT ValidPassword("ab1")
```

输出依次是 FALSE（没有数字）、TRUE 和 FALSE（太短）。

11.3.3 练习

1. 修改上面的冒泡排序，让它把数字从大到小排列。

2. 写一个函数 `CountOf(Target : INTEGER) RETURNS INTEGER`，返回 `Target` 在全局数组 `Data[1:8]` 中出现的次数。

<details> <summary> 参考答案 </summary>

1. 把比较改成 `IF Numbers[j] < Numbers[j + 1] THEN`，其他都不用改。

```
DECLARE Data : ARRAY[1:8] OF INTEGER

FUNCTION CountOf(Target : INTEGER) RETURNS INTEGER
   DECLARE i : INTEGER
   DECLARE Count : INTEGER
   Count ← 0
   FOR i ← 1 TO 8
      IF Data[i] = Target THEN
         Count ← Count + 1
      ENDIF
   NEXT i
   RETURN Count
ENDFUNCTION

Data[1] ← 4
Data[2] ← 7
Data[3] ← 4
Data[4] ← 1
Data[5] ← 9
Data[6] ← 4
Data[7] ← 2
Data[8] ← 7
OUTPUT CountOf(4)
OUTPUT CountOf(7)
```

输出 3 和 2。这里循环**不能**提前停止：每个元素都要数到，所以用 FOR 循环是对的。
</details>

常见错误

- 只用一个 IF 来验证：用户只多得到一次机会，之后程序就带着错误数据继续运行。要用循环。
- 把验证和**核实**（verification）混为一谈。核实检查数据是否被**正确地复制**（两次输入、目视检查）；验证检查数据是否**合理**。

12 栈、队列与链表

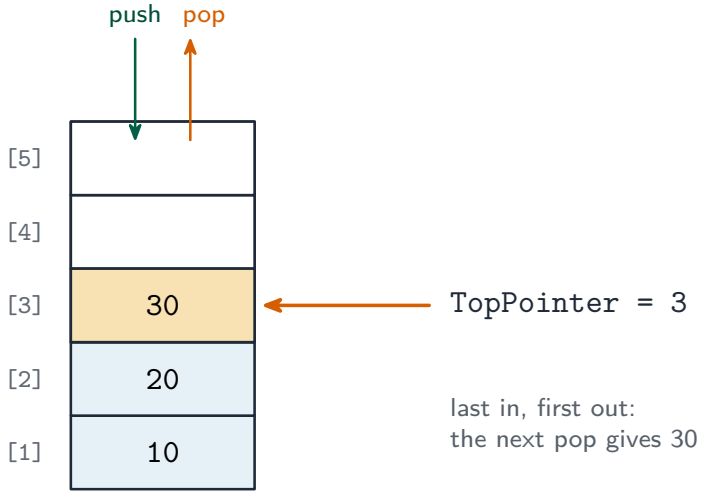
12.1 ■ 用数组实现栈

抽象数据类型（abstract data type，ADT）是一组数据，加上允许对这些数据进行的操作。使用 ADT 的人只需要知道各个操作**做什么**，不需要知道它们内部是怎么实现的。在 AS 阶段，你要用**数组**搭建三种 ADT：栈、队列和链表。

栈（stack）是**后进先出**（LIFO）的：最后放进去的项目最先被取出，就像一摞盘子。

- **压入**（push）：把一个项目放到顶部。
- **弹出**（pop）：取走顶部的项目。

数组存放各个项目，变量 `TopPointer` 存放顶部项目的索引。栈为空时，`TopPointer` 为 0。



用数组实现的栈：*TopPointer* 指向最后压入的项目

```
CONSTANT MaxSize = 5
DECLARE Stack : ARRAY[1:5] OF INTEGER
DECLARE TopPointer : INTEGER

PROCEDURE Push(Item : INTEGER)
   IF TopPointer = MaxSize THEN
      OUTPUT "Stack full: cannot push ", Item
   ELSE
      TopPointer ← TopPointer + 1
      Stack[TopPointer] ← Item
   ENDIF
ENDPROCEDURE

FUNCTION Pop() RETURNS INTEGER
   DECLARE Item : INTEGER
   IF TopPointer = 0 THEN
      OUTPUT "Stack empty"
      RETURN -1
   ELSE
      Item ← Stack[TopPointer]
      TopPointer ← TopPointer - 1
      RETURN Item
   ENDIF
ENDFUNCTION

TopPointer ← 0
CALL Push(10)
CALL Push(20)
CALL Push(30)
OUTPUT "Popped ", Pop()
OUTPUT "Popped ", Pop()
CALL Push(40)
OUTPUT "Popped ", Pop()
```

输出依次是 30、20、40：取出的永远是还留在栈里的**最新**项目。

- Push 先检查栈是否**已满**；Pop 先检查栈是否**为空**。评分标准会看这些检查。
- Pop 不需要清除数组里的旧值。把 TopPointer 往下移就够了：那个位置会被下一次 Push 覆盖。

常见错误

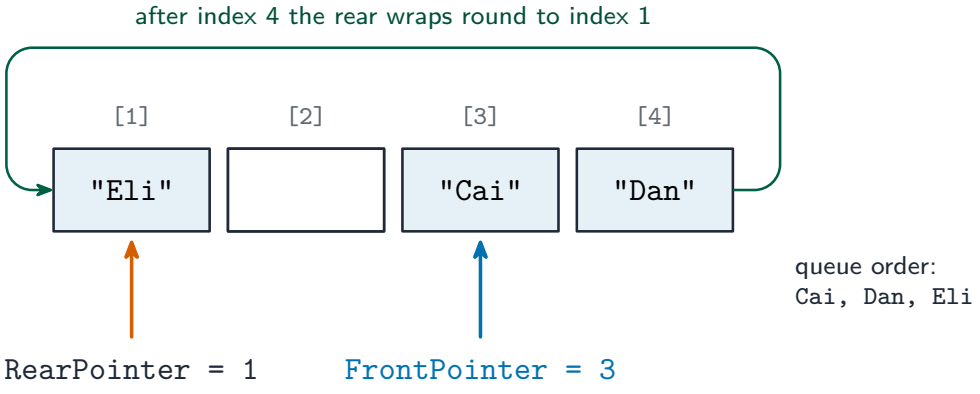
- 修改指针的顺序错了。在 Push 中，先增加 TopPointer **再**存值；在 Pop 中，先取出项目**再**减小 TopPointer。
- 忘记做“ 已满 ”或“ 为空 ”的检查。

12.2 ■ 用数组实现循环队列

队列（queue）是**先进先出**（FIFO）的：项目按到达的顺序离开，就像排队的人。

- **入队**（enqueue）：在**队尾**添加一个项目。
- **出队**（dequeue）：从**队头**取走一个项目。

如果队列只是在数组中一直往前移动，即使数组开头已经空出位置，它也很快会跑出数组末尾。**循环队列**（circular queue）解决了这个问题：指针越过最后一个索引时，会**绕回**索引 1。



循环队列：指针越过最后一个索引后回到 *1*

这个队列使用三个变量：

- `FrontPointer`：队头项目的索引；
- `RearPointer`：队尾项目的索引；
- `NumberInQueue`：项目的个数，有了它，判断”已满”和”为空”都很容易。

```
CONSTANT MaxSize = 4
DECLARE Queue : ARRAY[1:4] OF STRING
DECLARE FrontPointer : INTEGER
DECLARE RearPointer : INTEGER
DECLARE NumberInQueue : INTEGER

PROCEDURE Enqueue(Item : STRING)
   IF NumberInQueue = MaxSize THEN
      OUTPUT "Queue full"
   ELSE
      RearPointer ← RearPointer + 1
      IF RearPointer > MaxSize THEN
         RearPointer ← 1
      ENDIF
      Queue[RearPointer] ← Item
      NumberInQueue ← NumberInQueue + 1
```

```
   ENDIF
ENDPROCEDURE

FUNCTION Dequeue() RETURNS STRING
   DECLARE Item : STRING
   IF NumberInQueue = 0 THEN
      RETURN "EMPTY"
   ELSE
      Item ← Queue[FrontPointer]
      FrontPointer ← FrontPointer + 1
      IF FrontPointer > MaxSize THEN
         FrontPointer ← 1
      ENDIF
      NumberInQueue ← NumberInQueue - 1
      RETURN Item
   ENDIF
ENDFUNCTION

FrontPointer ← 1
RearPointer ← 0
NumberInQueue ← 0
CALL Enqueue("Ann")
CALL Enqueue("Ben")
CALL Enqueue("Cai")
OUTPUT Dequeue()
OUTPUT Dequeue()
CALL Enqueue("Dan")
CALL Enqueue("Eli")
CALL Enqueue("Fay")
OUTPUT Dequeue(), " ", Dequeue(), " ", Dequeue(), " ", Dequeue()
```

`Eli` 又被放进了索引 1，因为 `RearPointer` 绕回来了。队列仍然按到达的顺序把名字送出来。

绕回也可以用 `MOD` 来写：`RearPointer ← (RearPointer MOD MaxSize) + 1`。两种写法都可以。

常见错误

- 忘记绕回，指针跑出了最后一个索引。
- 让 `RearPointer` 从 1 开始。在上面的写法中它从 0 开始，第一次 `Enqueue` 时才移到 1。

12.3 ■ 用数组实现链表

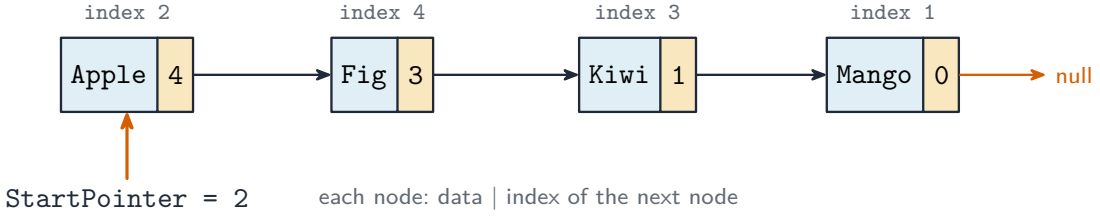
链表（linked list）是这样一种列表：每个项目叫作一个**节点**（node），它既存放数据，**也**存放一个指向下一个节点的**指针**（pointer）。各个项目不必按顺序存放：顺序由指针决定。

用数组实现链表需要：

- 一个存放数据的数组，以及一个同样大小的数组 `NextPointer`；
- `StartPointer`：**第一个**节点的索引；
- 一个**空指针**（null pointer），这里用 0，表示链表的结尾。

索引	Data	NextPointer
1	`"Mango"`	0
2	`"Apple"`	4
3	`"Kiwi"`	1
4	`"Fig"`	3

当 `StartPointer = 2` 时，这个链表读作 Apple → Fig → Kiwi → Mango，是按字母顺序排列的，尽管数组本身并不是。



从 *StartPointer* 出发沿着指针走，就得到链表的顺序

12.3.1 沿着指针走

要访问每个节点，从 StartPointer 开始，沿着每个 NextPointer 一直走，直到遇到空指针：

```
CONSTANT NullPointer = 0
DECLARE Data : ARRAY[1:4] OF STRING
DECLARE NextPointer : ARRAY[1:4] OF INTEGER
DECLARE StartPointer : INTEGER
DECLARE Current : INTEGER
Data[1] ← "Mango"
NextPointer[1] ← 0
Data[2] ← "Apple"
NextPointer[2] ← 4
Data[3] ← "Kiwi"
NextPointer[3] ← 1
Data[4] ← "Fig"
NextPointer[4] ← 3
StartPointer ← 2

Current ← StartPointer
WHILE Current <> NullPointer
   OUTPUT Data[Current]
   Current ← NextPointer[Current]
ENDWHILE
```

12.3.2 添加与删除节点

添加或删除节点时，**只改指针**，数据不用移动。

- **删除** “Kiwi”：让它前面的节点（“Fig”，索引 4）指向它后面的节点（“Mango”，索引 1），即 NextPointer[4] ← 1。
- **添加**新节点：把它的数据存进一个空闲元素，让它指向应该排在它后面的节点，再让它前面的节点指向这个新节点。

没有使用的元素通常放在第二个链表里，叫作**空闲链表**，它有自己的 FreeListPointer。带空闲链表的完整插入和删除算法属于 A-Level（Paper 4）的内容；在 AS 阶段，你应该能沿着指针走，并说出哪些地方发生了变化。

12.3.3 练习

1. 在上面的表格中，链表的**最后**一个节点在哪个索引？你是怎么知道的？

2. 使用上面的队列，"Fay" 入队之后，RearPointer 的值是多少？

<details> <summary> 参考答案 </summary>

1. 索引 1（"Mango"）。它的 NextPointer 是 0，也就是空指针，所以它后面没有节点了。
2. RearPointer 是 2。前三次入队把它移到 1、2、3；（两次出队之后）下一次移到 4；然后为 "Eli" 绕回到 1，再为 "Fay" 移到 2。

</details>

常见错误

- 按**索引顺序**（1、2、3、4）读链表，而不是沿着指针走。
- 把两种 ADT 弄混：栈从添加的**同一端**取出；队列从**另一端**取出。
- 用 `Current ← Current + 1` 在链表中前进。下一个节点是 `NextPointer[Current]`，它通常在别的位置。

13 速查表

13.1 ■ 关键字一览

用这一页快速核对某个**语法**（syntax）形式。章号告诉你它在哪里有详细讲解。

13.1.1 数据与赋值

形式	示例	章
DECLARE <id> : <type>	DECLARE Total : INTEGER	2
CONSTANT <id> = <value>	CONSTANT Rate = 0.08	2
<id> ← <value>	Total ← Total + 1	2
DECLARE <id> : ARRAY[<l>:<u>] OF <type>	DECLARE Names : ARRAY[1:30] OF STRING	6
DECLARE <id> : ARRAY[<l1>:<u1>, <l2>:<u2>] OF <type>	DECLARE Board : ARRAY[1:3, 1:3] OF CHAR	6
TYPE <id> … ENDTYPE	由若干 DECLARE 字段组成的记录	9

数据类型：INTEGER、REAL、CHAR、STRING、BOOLEAN、DATE。

13.1.2 运算符

种类	运算符
算术	+ - * / DIV MOD
关系	= <> < > <= >=
逻辑	AND OR NOT
字符串	&

13.1.3 输入与输出

形式	示例
INPUT <id>	INPUT Age
OUTPUT <value(s)>	OUTPUT "Total: ", Total

13.1.4 选择与循环

```
IF <condition> THEN
   <statement(s)>
ELSE
   <statement(s)>
ENDIF

CASE OF <identifier>
   <value> : <statement(s)>
   <value1> TO <value2> : <statement(s)>
   OTHERWISE : <statement(s)>
ENDCASE

FOR <id> ← <v1> TO <v2> STEP <n>
   <statement(s)>
   ↪ <statement(s)>
NEXT <id>

WHILE <condition>
   <statement(s)>
ENDWHILE

REPEAT

UNTIL
↪ <condition>
```

13.1.5 子程序

```
PROCEDURE <id>(BYREF <p> : <type>, …)
↪ …) RETURNS <type>
   <statement(s)>
ENDPROCEDURE

CALL <id>(<arguments>)
↪ <id>(<arguments>)

FUNCTION <id>(<p> : <type>,

   <statement(s)>
   RETURN <value>
ENDFUNCTION
<variable> ←
```

13.1.6 文件

形式	说明
`OPENFILE <file> FOR READ / WRITE / APPEND`	`WRITE` 会清空文件
`READFILE <file>, <variable>`	读取下一行
`WRITEFILE <file>, <data>`	写入一行
`EOF(<file>)`	没有剩余的行时为 `TRUE`
`CLOSEFILE <file>`	一定要关闭

13.1.7 内置函数（来自 Paper 2 插页）

函数	返回
LENGTH(S)	字符个数
LEFT(S, x) / RIGHT(S, x)	最左边 / 最右边的 x 个字符
MID(S, x, y)	从位置 x 开始的 y 个字符
TO_UPPER(x) / TO_LOWER(x)	把 x 变成大写 / 小写
NUM_TO_STR(x) / STR_TO_NUM(x)	数字转字符串 / 字符串转数字
IS_NUM(S)	S 是合法数字时为 TRUE
ASC(c) / CHR(x)	字符编码 / 字符
INT(x)	整数部分（直接去掉小数，从不四舍五入）
RAND(x)	从 0 到 x（不包括 x）的随机 REAL

注释以 // 开头。关键字全部大写；标识符以字母开头。

13.2 ■ 从 IGCSE 0478 过渡

很多同学先学的是 IGCSE 伪代码。剑桥的这两个考纲使用**不同的**规则，在一个考纲里养成的习惯，到另一个考纲里往往就是错误。下表列出你现在必须使用的 9618 写法。

	IGCSE 0478	AS/A Level 9618
THEN	单独一行，写在 IF 下面	写在 IF 这一行的末尾
常量	CONSTANT Pi ← 3.142	CONSTANT Pi = 3.142
DIV 和 MOD	函数：DIV(10, 3)	运算符：10 DIV 3
连接字符串	只能在 OUTPUT 中用逗号	& 运算符
CASE 中的 OTHERWISE	不带冒号	带冒号：OTHERWISE :
CASE 的值	单个值，每个值一条语句	可用 TO 表示范围，可有多条语句
参数	总是按值传递	BYVAL 或 BYREF
取字符串的一部分	SUBSTRING(S, 2, 3)	MID(S, 2, 3)，还有 LEFT 和 RIGHT
字符串转大写	UCASE(S)	TO_UPPER(S)
四舍五入、随机数	ROUND(X, 2)、RANDOM()	没有 ROUND；INT(X) 直接去掉小数；RAND(x)

	IGCSE 0478	AS/A Level 9618
标识符	以大写字母开头；只含字母和数字	可以任意字母开头；允许下划线
数据类型	五种	六种：增加了 DATE
记录	不使用	TYPE … ENDTYPE

下面是同一个小算法的 9618 写法，它可以在本页上运行：

```
CONSTANT Rate = 0.5
DECLARE Total : REAL
DECLARE Label : STRING
Total ← 10.0
IF 7 MOD 2 = 1 THEN
   Label ← "Odd, " & "half rate"
   OUTPUT Label, ": ", Total * Rate
ENDIF
```

IGCSE 的写法会把 `THEN` 单独放一行，写成 `CONSTANT Rate ← 0.5` 和 `MOD(7, 2)`，而且不能用 `&`。本站的 IGCSE 伪代码参考展示了那种写法。

13.3 ■ 如何写好 Paper 2 答案

Paper 2 要求你编写、补全、改正和跟踪伪代码。下面这些习惯能帮你拿分。

13.3.1 动笔之前

- **把整道题读完**，划出必须**输出**或**返回**的内容。
- 看清题目要的是**过程**还是**函数**；如果题目给出了头部，就原样照抄：名字、参数和返回类型。
- 看看**插页**：它列出了你可以使用的每个内置函数及其参数。

13.3.2 书写时

- **声明**变量，每行一个，类型要合理。
- 循环之前先**初始化**总和与计数器（`Total ← 0`）。
- 选择合适的循环（第 5 章）：已知次数用 `FOR`，”可能零次”用 `WHILE`，”至少一次”用 `REPEAT`。
- 每个代码块都要**缩进**，并且都要结束：`ENDIF`、`ENDWHILE`、`NEXT`、`ENDCASE`、`ENDPROCEDURE`、`ENDFUNCTION`。
- 数组、变量和子程序都使用题目中的**原名**。如果题目把数组叫作 `StockLevel`，就不要写成 `Stock`。
- 调用过程写 `CALL`，调用函数绝不写 `CALL`。
- 赋值用 `←`，`=` 只用于比较。

13.3.3 跟踪算法

跟踪表（trace table）记录每一步之后各个变量的值，让你看清一个算法在做什么。

- 每个变量占一列，输出也占一列。
- 只有当值**改变**时才写新值，每次往下写一行。
- 一行一行地执行。不要凭你觉得算法“ 应该 ”做什么来猜结果。

13.3.4 写完之后

- 像计算机那样读一遍你的答案：每个循环都会结束吗？每个变量在使用之前都有值吗？
- 在脑子里用一个简单的情况测试，再用一个**边界情况**（edge case）测试：空文件、恰好落在分界线上的值、找不到的值。

用跟踪表做一次**手工演算**（dry run），是在阅卷老师之前发现逻辑错误的最好方法。