

第1章 Raptor 编程简介

资料来源: Raptor官网: <http://raptor.martincarlisle.com/>

翻译\整理: 程向前 xqcheng@mail.xjtu.edu.cn

Raptor是一种基于流程图的可视化编程开发环境。流程图是一系列相互连接的图形符号的集合, 其中每个符号代表要执行的特定类型的指令。符号之间的连接决定了指令的执行顺序。一旦开始使用Raptor解决问题, 这样的理念将会变得更加清晰。

使用Raptor基于以下几个原因:

- Raptor开发环境, 在最大限度地减少语法要求的情形下, 帮助用户编写正确的程序指令。
- Raptor开发环境是可视化的。Raptor程序实际上是一种有向图, 可以一次执行一个图形符号, 以便帮助用户跟踪Raptor程序的指令流执行过程。
- Raptor是为易用性而设计的 (用户可用它与其他任何的编程开发环境进行复杂性比较)。
- Raptor所设计的报错消息更容易为初学者理解。
- 使用Raptor的目的是进行算法设计和运行验证, 不需要重量级编程语言, 如C++或Java。

1.1 Raptor 程序结构

Raptor程序是一组连接的符号, 表示要执行的一系列动作。符号间的连接箭头确定所有操作的执行顺序。Raptor程序执行时, 从开始 (Start) 符号起步, 并按照箭头所指方向执行程序。Raptor程序执行到的结束 (End) 符号时停止。最小的Raptor程序 (什么也不做), 如图1-1所示。在开始和结束的符号之间插入一系列Raptor语句/符号, 就可以创建有意义的Raptor程序。



图1-1 开始和结束符号

Raptor 基本符号

Raptor有六种基本符号, 每个符号代表一个独特的指令类型。基本符号如图1-2所示。首先介绍赋值 (assignment), 调用 (Call), 输入 (Input) 和输出 (Output) 四个类型的语句, 而选择 (Selection) 和循环 (Loop) 中, 将在稍后解释。

典型的计算机程序有三个基本组成部分:

- 输入 (Input) - 完成任务所需要的数据值。
- 加工 (Process) - 操作数据值来完成任务。
- 输出 (Output) - 显示 (或保存) 加工处理后的结果。

这三个组件与Raptor指令形成直接的关系如表1-1所示。

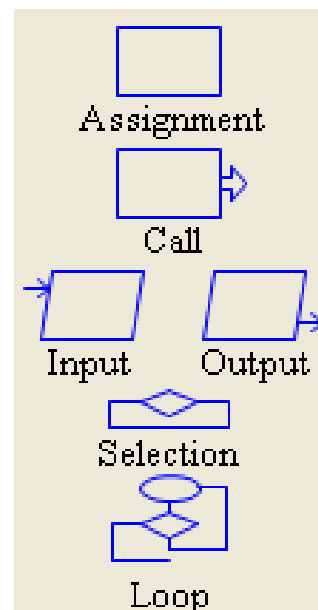
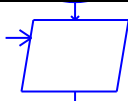

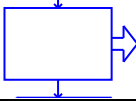
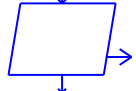


图1-2 Raptor六种基本符号

表 1-1 四种 Raptor 基本指令说明

目的	符号	名称	说明
输入		输入语句	允许用户输入的数据。每个数据值存储在一个 变量 中。
处理		赋值语句	使用某些类型的数学计算来更改的 变量 的值。
处理		过程调用	执行一组在命名过程中定义的指令。在某些情况下，过程中的指令将改变一些过程的参数（即 变量 ）。
输出		输出语句	显示 变量 的值（或保存到文件中）。

这四个指令之间的共同点是，它们都对变量进行某种形式的操作！要了解如何设计算法，进而使之成为可以运行的计算机程序，必须明白变量的概念。

变量

变量表示的是计算机内存中的位置，用于保存数据值。在任何时候，一个变量只能容纳一个值。然而，在程序执行过程中，变量的值可以改变。这就是为什么他们被称为“变量”的原因！作为一个例子，研究名为X的变量值的变化过程，如表1-2所示：

表 1-2 变量赋值过程

说明	X 的值	程序
当程序开始时，没有变量存在。Raptor 变量在某个语句中首次使用时自动创建。	未定义	<pre> graph TD Start([Start]) --> A[X ← 32] A --> B[X ← X + 1] B --> C[X ← X * 2] </pre>
第一个赋值语句， $X \leftarrow 32$ ，分配数据值 32 给变量 X	32	
下一个赋值语句， $X \leftarrow X + 1$ ，检索到当前 X 的值为 32，给它加 1，并把结果 33 给变量 X	33	
下一个赋值语句， $X \leftarrow X * 2$ ，检索到 X 当前值为 33，乘以 2，并把结果 66 给变量 X	66	

在上例程序的执行中，变量X存储过三个不同的值。请注意，在一个程序中的语句顺序是非常重要的。如果重新排列这三个赋值语句，存储在X中的值则会有所不同。

一个变量值的设置（或改变）可以采取以下三种方式之一：

- 用一个输入语句赋值。
- 通过赋值语句的中的公式计算赋值。
- 通过从一个过程调用的返回值赋值。

顾名思义，变量数据值的变化导致程序每次执行的结果可以不同。

程序员应给予所有的变量有意义的和具有描述性的名称。变量名应该与该变量在程序中的作用有关。变量名必须以字母开头，可以包含字母、数字、下划线（但不可以有空格或其他特殊字符）。如果一个变量名中包含多个单词，两个单词间用下划线字符分隔，这样变量名则更具有可读性。表1-3显示了一些好的、差的和非法的变量名的例子。

表1-3 变量名实例

好的变量名	差的变量名	非法的变量名
tax_rate sales_tax distance_in_miles mpg	a (没有描述) milesperhour(添加下划线) my4to (没有描述)	4sale (不可以字母开头) sales tax (包括空格) sales\$ (包括无效字符)

要点提示：如果给程序中的每个值一个有意义的、具有描述性的变量名，它会帮助用户更清楚思考需要解决的问题，并帮助寻找程序中的错误。

了解变量的方法之一，是将它们看成程序不同部分之间进行信息交流的一种手段。在你的程序的不同部分使用相同的变量名，用户使用的是存储在同一位置中值。把变量看作一个存储区域并在程序的计算过程中参与计算。

Raptor程序开始执行时，没有变量存在。当Raptor遇到一个新的变量名，它会自动创建一个新的内存位置并将该变量的名称与该位置相关联。在程序执行过程中，该变量将一直存在，直到程序终止。当一个新的变量创建时，其初始值将决定该变量将存储数值数据或文本数据。这就是所谓的变量的数据类型。一个变量的数据类型在程序执行期间不能更改的。总之，变量自动创建时，Raptor可以在其中保存：

- 数值 (Numbers) e.g., 12, 567, -4, 3.1415, 0.000371, 或
- 字符串¹ (Strings) e.g., “Hello, how are you?”, “James Bond”, “The value of x is ”

使用变量时常见的错误：

错误1：“变量____无值” (Error 1: "Variable ____ does not have a value")

此错误的常见原因有两个：1) 变量无值。 2) 变量名拼写错误。（参见图1-3）

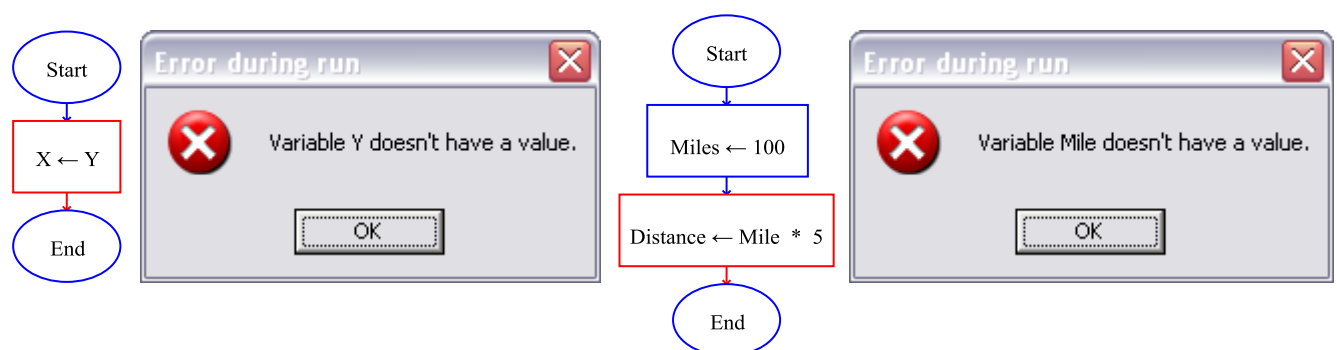


图 1-3 变量无值

错误2：“不能将字符串赋值给数值变量_____” (Error 2: "Can't assign string to numeric variable _____")；“不能将数值赋值给字符串变量_____” ("Can't assign numeric to string variable _____")，如果某个语句试图改变一个变量的数据类型，会发生此错误。（参见图1-4）

¹注：目前（raptor 2011）尚不能保存汉字，所有字符串仅限 ASCII 字符。

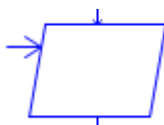


图 1-4 变量类型用错时的提示

1.2 Raptor 语句/符号

以下说明输入 (Input)、赋值 (Assignment)、调用 (Call) 和输出 (Output) 这四个基本语句。

输入(Input)语句/符号



输入语句/符号允许用户在程序执行过程中输入程序变量的数据值。这里最为重要的是，必须让用户明白这里程序需要什么类型的数据值。因此，当定义一个输入语句时，一定要在提示 (Prompt) 文本中说明所需要的输入。提示应尽可能明确，如果预期值需要单位或量纲 (如英尺，米，或英里)，应该在提示文本中说明 (参见图1-5)。

当你定义一个输入语句时，用户必须指定两件事：一是提示文本，二是变量名称，该变量的值将在程序运行时由用户输入。正如可以“Enter Input”对话框中看到的那样 (参见图1-5)。使用表达式提示 (Expression prompt) 可以将文本与变量进行组合成输入提示，如：**“Enter a number between ” + low + “ and ” + high + “: ”**。

输入语句在运行时，将显示一个输入对话框，如图1-6所示。

在用户输入一个值，并按下ENTER键 (或点击OK)，用户输入值由输入语句赋给变量。

请仔细思考“语句定义 (definition of a statement)”和“语句执行 (execution of a statement)”的区别。定义语句对话框与执行

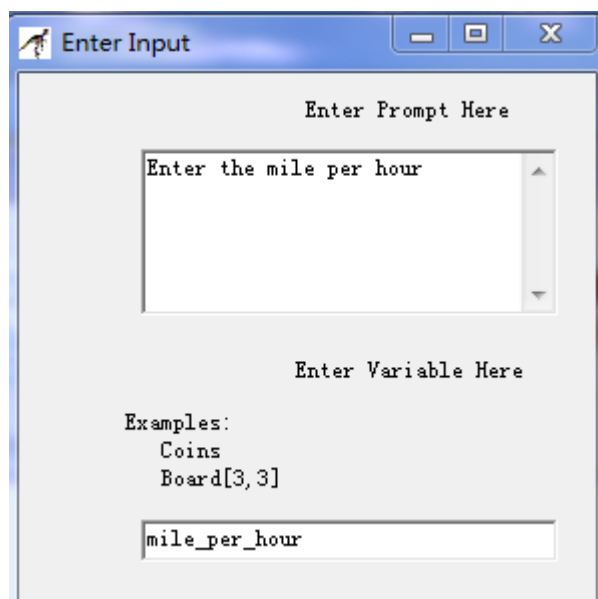


图 1-5 输入语句的编辑 (Edit) 对话框

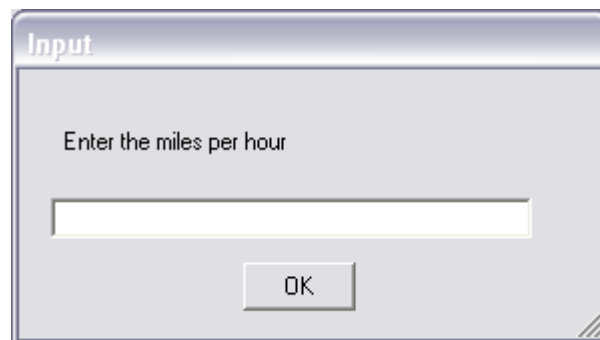


图 1-6 输入语句的运行时 (run-time) 对话框

程序时使用的对话框是完全不同的。

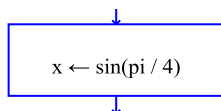
赋值语句/符号

赋值符号是用于执行计算，然后将其结果存储在变量中。赋值语句的定义是使用如图1-7显示的对话框。需要赋值的变量名须输入到“Set”字段，需要执行的计算输入到“to”字段。图1-7的示例将变量X的值赋为 0.7071。

Raptor使用的赋值语句在其符号中语法为：

变量 ← 表达式 (Variable ← Expression)

例如，图1-7对话框创建语句显示为：



一个赋值语句只能改变一个变量的值，也就是箭头左边所指的变量。如果这个变量在先前的语句中未曾出现过，则raptor会创建一个新的变量。如果这个变量在先前的语句已经出现，那么先前的值就将为目前所执行的计算所得的值所取代。而位于箭头右侧（即表达式）中的变量的值则不会被赋值语句改变。

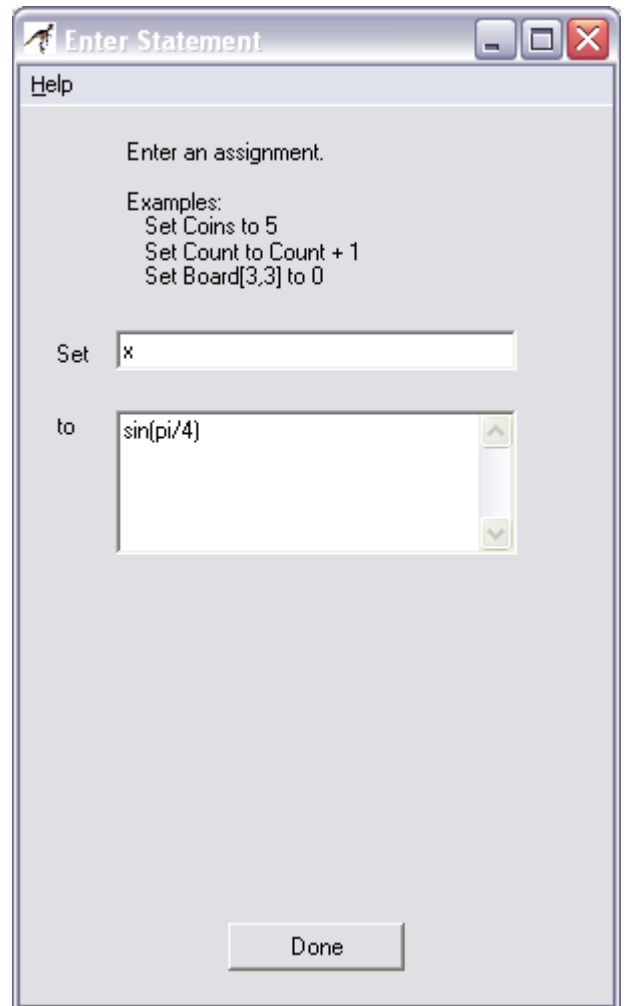


图 1-7 赋值语句的编辑对话框

表达式

一个赋值语句中的表达式（或计算）的可以是任何计算单个值的简单或复杂的公式。**表达式是值（无论是常量或变量）和运算符的组合。**请仔细研究以下构建有效表达式的规则。

一台计算机只能一次执行一个操作。当一个表达式进行计算时，方程的运算并不是象用户输入时那样，按从左到右的优先顺序进行。实际的运算的执行顺序，是按照预先定义“优先顺序”进行的。例如，考虑下面的两个例子：

$$(1) x \leftarrow (3+9)/3 \qquad (2) x \leftarrow 3+(9/3)$$

在第一种情况中，变量x被赋的值为4，而在第二种情况下，变量x被赋的值为6。从这些例子中可以看到，可以随时使用括号明确地控制值和运算符的分组，一般性的“优先顺序”为：

1. 计算的所有函数（function），然后

2. 计算括号中表达式，然后
3. 计算乘幂(^,**)，然后
4. 从左到右，计算乘法和除法，最后
5. 从左到右，计算加法和减法。

运算符或函数指示计算机对一些数据执行计算。运算符须放在操作数据之间（如 $X/3$ ），而函数使用括号来表示正在操作的数据（例如：SQRT（4.7））。在执行时，运算符和函数执行各自的计算，并返回其结果。表1-4概括了Raptor内置的运算符和函数。

表 1-4 内置的运算符和函数

基本数学运算(basic math) :	+, -, *, /, ^, **, rem, mod, sqrt, log, abs, ceiling, floor
三角函数 (trigonometry) :	sin, cos, tan, cot, arcsin, arcos, arctan, arccot
杂项 (miscellaneous) :	random, Length_of

表1-5简要介绍了这些内置的运算符和函数。关于这些运算符和函数的全部细节，可以查阅Raptor帮助文档。

表 1-5 运算符和函数说明

运算	说明	范例
+	加	$3+4 = 7$
-	减	$3-4 = -1$
-	负号	-3
*	乘	$3*4 = 12$
/	除	$3/4$ is 0.75
^ 或 **	幂运算	$3^4 = 3*3*3*3=81$ $3**4 = 81$
rem mod	求余数	$10 \text{ rem } 3 = 1$ $10 \text{ mod } 4 = 2$
sqrt	求平方根	$\text{sqrt}(4) = 2$
log	自然对数（以 e 为底）	$\log(e) = 1$
abs	绝对值	$\text{abs}(-9) = 9$
ceiling	向上取整	$\text{ceiling}(3.14159) = 4$
floor	向下取整	$\text{floor}(9.82) = 9$
sin	正弦(以弧度表示)	$\sin(\pi/6) = 0.5$
cos	余弦(以弧度表示)	$\cos(\pi/3)$ is 0.5
tan	正切(以弧度表示)	$\tan(\pi/4)$ is 1.0
cot	余切(以弧度表示)	$\cot(\pi/4)$ is 1
arcsin	反正弦(expression), 返回弧度	$\arcsin(0.5) = \pi/6$
arcos	反余弦(expression), 返回弧度	$\arccos(0.5) = \pi/3$
arctan	反正切(y,x), 返回弧度	$\arctan(10, 3) = 1.2793$
arccot	反余切(x,y), 返回弧度	$\text{arccot}(10, 3) = 0.2915$
random	生成一个范围在 (1.0, 0.0) 随机值	random * 100 = 0~99.9999
Length_of	字符数返回一个字符串变量	Example ← "Sell now" Length_of(Example) = 8

在赋值语句中的表达式的运行结果（result of evaluating）必须是一个数值或一个文本字符串。大部分表达式用于计算的数值，但也可以用加号（+）进行简单的文字处理，把两个或两个以上的文本字符串的合并成为单个字符串。用户还可以将字符串和数值变量

组合成一个单一的字符串。下面的例子显示赋值语句的字符串操作。

```
Full_name ← "Joe " + "Alexander " + "Smith"
Answer ← "The average is " + (Total / Number)
```

Raptor定义了几个符号表示常用的**常量**。当用户需要在计算其相应的值，应该使用这些常数的符号。

pi 定义为 3.1416.

e 定义为 2.7183

过程调用语句/符号



一个过程是一个编程语句的命名集合，用以完成某项任务。调用过程时，首先暂停当前程序的执行，然后执行过程中的程序指令，然后在先前暂停的程序的下一语句恢复执行原来的程序。要正确使用过程，用户需要知道两件事情：1) 过程的名称和 2) 完成任务所需要的数据值，也就是所谓的参数。

Raptor 设计中，为尽量减少用户的记忆负担，在过程调用的编辑对话框“Enter Call”中，会随用户的输入，按部分匹配原则，“Enter Call”对话框中按用户输入过程的名称进行提示，这对减少输入错误大有裨益。例如，输入字母“d”后，窗口的下部会列出所有以字母“D”开头的内置的过程。该列表还提醒每个过程所需的参数。在图 1-8 所列的对话框中，告诉用户，“Draw_Line”过程需要 5 个数据值：线段的起始位置的 X 和 Y 坐标，(X1, Y1) 和结束位置的 X 和 Y 坐标，(X2, Y2) 和线段的颜色。过程调用时的参数值的顺序必须与过程定义参数一致。例如，Draw_Line (blue, 3, 5, 100, 200) 将产生一个错误，因为该线段的颜色必须在参数列表中的最后一个参数值位置上。

当一个过程调用显示在 Raptor 程序中时，可以看到被调用的过程名称和参数值。

例如，图 1-9 所示的第一个过程调用执行时，它会画一条从点 (1,1) 到点 (100,200) 红线。第二个过程调用时，也将画一条线，但由于参数是变量，该线段的确切位置只有在程序执行到所有的参数变量有值后才能知道。

Raptor 定义了较多的内置过程，无法在此一一说明。在必要时，可以参考 Raptor 帮助中的所有内置过程的文档。

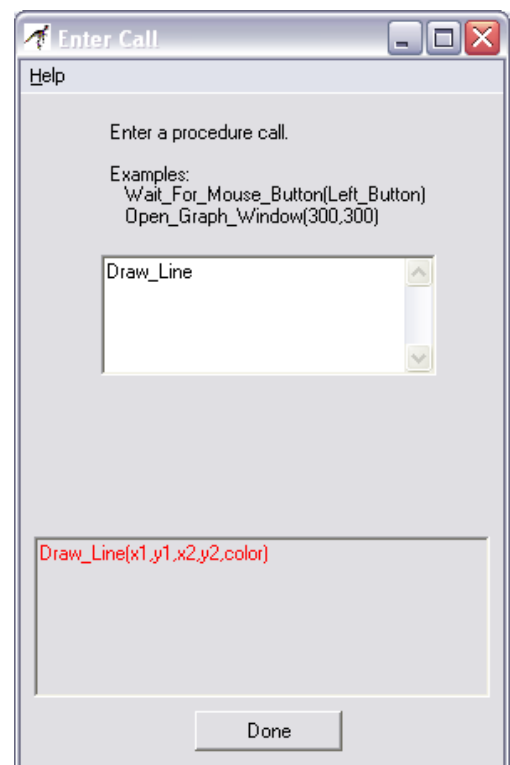


图 1-8 调用过程编辑对话框

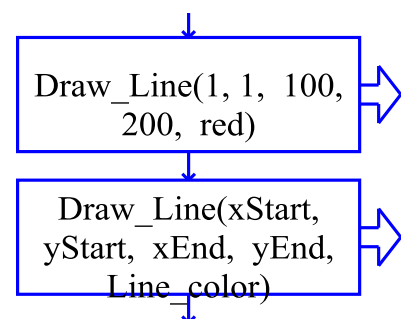
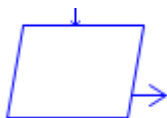


图1-9 过程调用的例子

输出语句/符号



Raptor 环境中，执行输出语句将导致程序执行时，在主控（Master Console）窗口显示输出结果。当定义一个输出语句，“Enter Output Here”对话框，要求用户指定两件事：

- 要如何显示什么样的文字或表达式结果；
- 是否需要在输出结束时输出一个换行符？

图 1-10 所示的例子输出语句将显示文本，输出窗口上的“The sale tax is”，并另起一行。这是由于“End Current line”被选中，以后的输出内容将从新的一行开始显示。

可以在“Enter Output Here”对话框中使用字符串加号（+）运算符将文本字符串与从两个或多个值构成一个单一的输出语句。必须将任何文本包含在引号（“）中以区分文本和计算值，在这种情况下，引号不会显示在输出窗口。例如，表达式：

```
"Active Point = (" + x +  
", " + y + ")"
```

如果 x 200 和 y 是 5，将显示以下结果：

Active Point = (200,5)

请注意，引号不在输出设备上显示。引号用于环绕任何文字，而不是表达式运算的组成部分。

老师常常会说：“请使用用户友好的方式显示的结果”。这意味着应该显示一些说明性文本解释任何输出在 Master Console 窗口的数字。图 1-11 分别展示了一个“非人性化的输出”和“用户友好的输出”的例子。

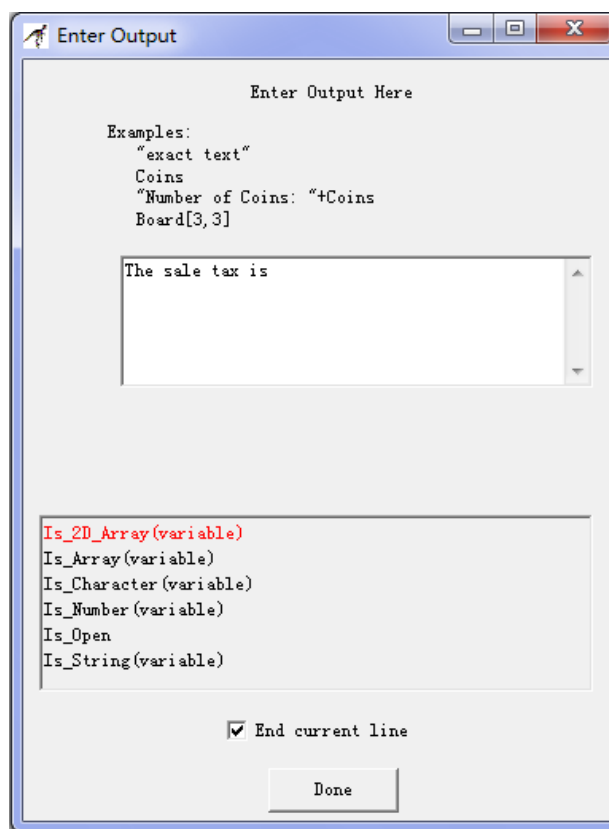
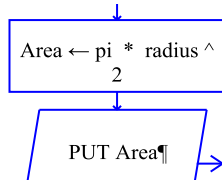


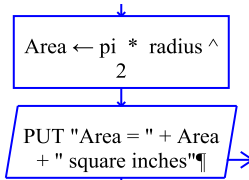
图 1-10 输出编辑对话框

非用户友好的输出



示例输出： 2.5678

用户友好的输出



示例输出： Area = 2.5678 square inches

图 1-11 两个输出语句的比较

Raptor 中的注释

Raptor 的开发环境，像其他许多编程语言一样，允许对程序进行注释。注释用来帮助他人理解程序，特别是程序代码比较复杂、很难理解情况下。注释本身对计算机毫无意义，并不会被执行。然而，如果注释得当，可以使程序更容易理解为他人理解。

要为某个语句中添加注释，用户鼠标右键单击相关的语句符号，然后选择“Enter Comment”。然后进入“Enter Comment”对话框，如图 1-12 所示。注释可以在 Raptor 窗口被移动，但建议不需要移动注释的默认位置，以防在需要更改时，引起错位和寻找的麻烦。

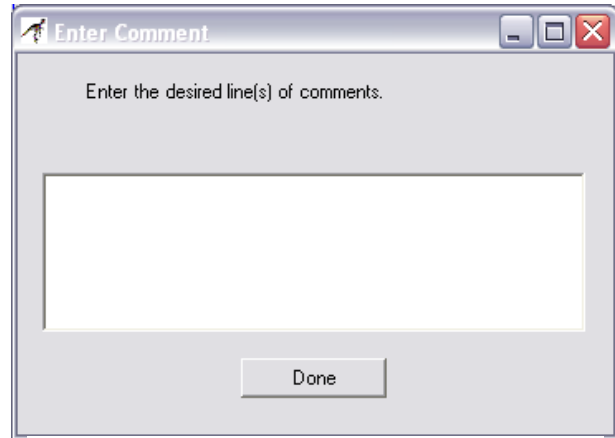


图 1-12 注释编辑对话框

注释一般有三种类型：

- 编程标题 - 谁是程序的作者和编写的时间，程序的目的等。（添加到“Start”符号中）
- 分节描述 - 用于标记程序，使程序员更容易理解的程序整体结构中的主要部分。
- 逻辑描述 - 解释非标准逻辑。

通常情况下，没有必要注释每一个程序语句。图 1-13 是一个示例程序，其中包括了注释。

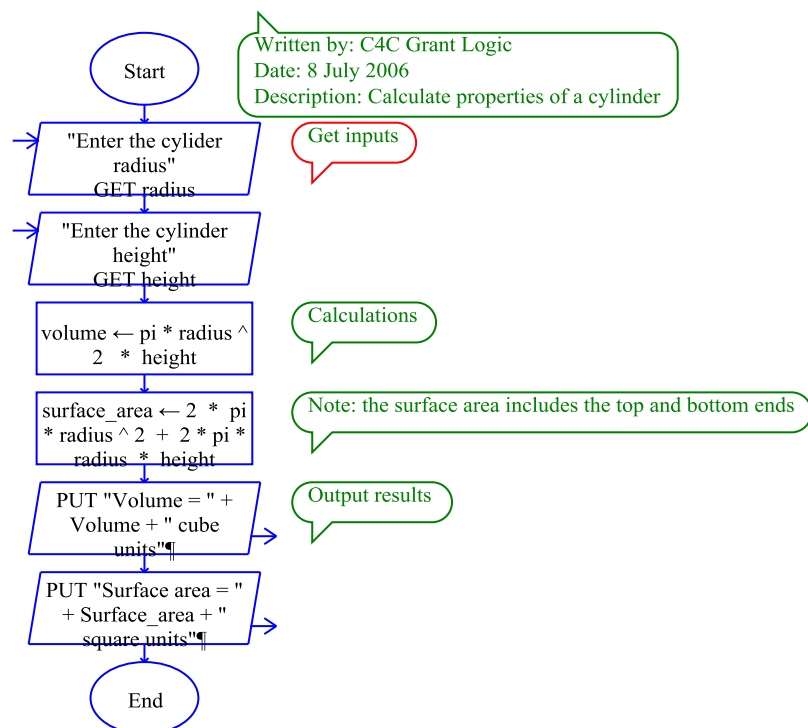


图 1-13 带注释的算法

1.3 Raptor 控制结构

编程的最重要工作之一是控制的语句的执行流程。控制结构/控制语句使程序员，可以确定程序语句的执行顺序。这些控制结构可以做两件事：

- 1) 跳过某些语句而执行的其他语句，
- 2) 条件为真时重复执行一条或多条语句。

Raptor程序使用的语句有六种基本类型，如图1-14所示。前文已经介绍了其中的四个。本节介绍选择（Selection）和循环（Loop）命令。

顺序控制

前文所述的大部分案例使用顺序控制。顺序逻辑是最简单的程序构造。本质上，就是把每个语句按顺序排列，程序执行时，从开始（Start）语句顺序执行到结束

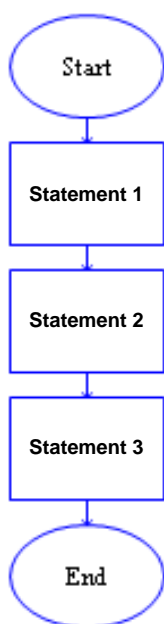


图 1-15 顺序控制结构

(End) 语句。正如图1-15的示例程序，箭头连接的语句描绘了执行流程。如果程序，包括20个基本命令，然后它会顺序执行这20个语句，然后退出。

程序员为解决问题，必须确定创建一个问题的解决方案需要那些语句，以及语句的执行顺序。编写正确的语句是一个任务，同样重要的是确定语句在程序的何处放置。例如，当要获取和处理来自用户的数据时，必须先取得数据，然后才可以使用。如果交换一下这些语句的顺序，则程序根本无法执行。

顺序控制是一种“默认”的控制，在这个意义上，流程图中的每个语句自动指向下一个。顺序控制是如此简单，除了把语句按顺序排列，不需要做任何额外的工作。然而，仅仅使用顺序控制，是无法开发真正针对现实世界的问题解决方案。真实的世界中的问题包括了各种“条件”，并以此来确定下一步应该怎样做。例如，“如果熄灯号响了，就必须把灯熄了”，是基于一天的某个时间点所做出的决定。这里的“条件”（即当前时间）确定的了某个行动（熄灯）是否应执行或不执行。这就是所谓的“选择控制”。

选择控制

一般情况下，程序需要根据数据的一些条件来决定是否应执行某些语句。例如，使用赋值语句计算的线段的斜率， $\text{slope} \leftarrow \text{DY} / \text{DX}$ ，就需要确保的DX值不为零（因为被零除没有数学定义，因此会产生一个运行时错误）。因此，需要做的决策是，“DX=0？”

选择控制语句可以使程序根据数据的当前状态，选择两种可选择的路径中的一条来执行下一条语句。如图1-16所示，Raptor的选择控制语句，呈现出一

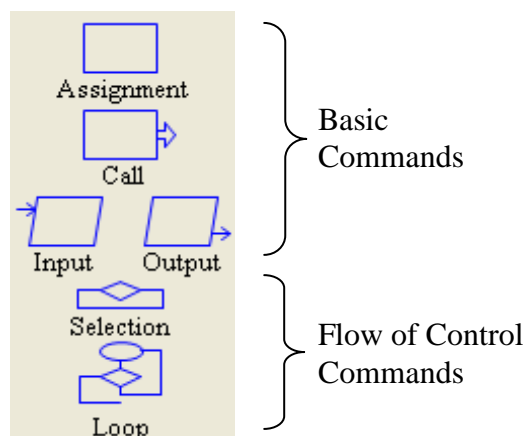


图 1-14 两类不同的基本命令

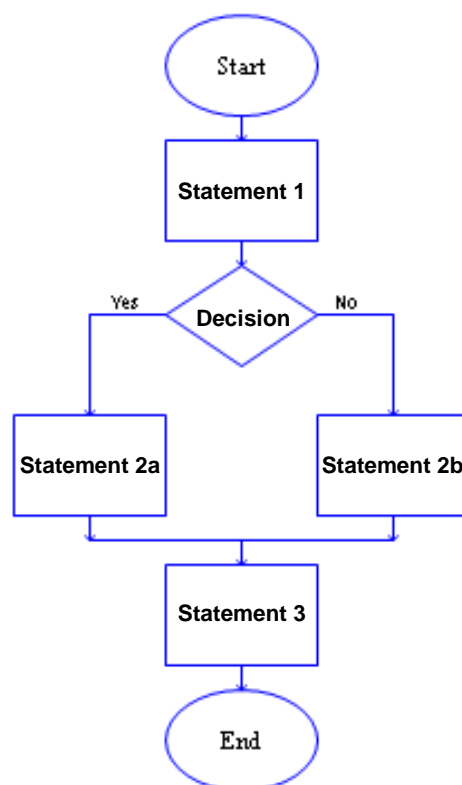
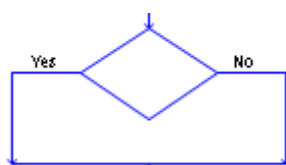


图 1-16 选择控制结构

个菱形的符号，用“**Yes/No**”表示对问题的决策结果以及决策后程序语句执行指向。当程序执行时，如果决策的结果是“**Yes**”（**True**），则执行左侧分支。如果结果是“**No**”（**False**），则执行右侧分支。在图1-16的示例中，Statement 2A或Statement 2B都有可能执行，但不会二者都被同时执行。请注意，该程序有两种可能的执行过程（表1-6）：

表 1-6 图 1-16 选择结构的可能执行过程

可能性 1	可能性 2
Statement 1	Statement 1
Statement 2a	Statement 2b
Statement 3	Statement 3

另外还要注意选择控制语句的两个路径之一可能是空的，或包含多条语句。如果两个路径同时为空或包含完全有相同的语句，则是不合适的。因为无论选择决策的结果如何，对程序的过程都没有影响。

决策表达式（Decision Expressions）

选择控制语句需要一个表达式来得到“**Yes/No**”（ **True/False**）的评估值。决策表达式是一组值（常量或变量）和运算符的结合。请仔细研究以下规则，以便构建有效的决策表达式。

在赋值语句表达式的讨论中曾经提到，计算机只能一次执行一个操作。决策表达式求值时，表达式的运算也不会按输入的顺序由左到右。相反，它是根据预定义的“**优先顺序**”的基础上执行运算。不同的运算执行顺序，可以产生完全不同的结果。通过括号，可以显式控制值和运算符的分组，从而控制运算的顺序。由于决策表达式可以包含在赋值语句中类似的计算，以下的“**优先顺序**”也必须包括赋值语句表达式中的运算符。决策表达式的运算“**优先顺序**”是：

- 1. 计算的所有函数，然后
- 2. 计算括号中的所有表达式，然后
- 3. 计算乘幂(^,**)，然后
- 4. 从左到右，计算乘法和除法，然后
- 5. 从左到右，计算加法和减法，然后
- 6. 从左到右，进行关系运算（= = / = <<=>> =）
- 7. 从左到右，进行not逻辑运算
- 8. 从左到右，进行and逻辑运算，
- 9. 从左到右，进行xor逻辑运算，最后
- 10. 从左到右，进行or逻辑运算。

在上面的列表中，关系和逻辑运算符都是新出现的，这些新的运算符解释如下。

表 1-6 决策表达式中的运算符说明

运算	说明	范例
=	等于（切勿与赋值符号混淆）	3 = 4 结果为 No(false)
!=	不等于	3 != 4 结果为 Yes(true)
/=		3 /= 4 结果为 Yes(true)
<	小于	3 < 4 结果为 Yes(true)

<=	小于或等于	3 <= 4 结果为 Yes(true)
>	大于	3 > 4 结果为 No(false)
>=	大于或等于	3 >= 4 结果为 No(false)
and	与	(3 < 4) and (10 < 20) 结果为 Yes(true)
or	或	(3 < 4) or (10 > 20) 结果为 Yes(true)
xor	异或	Yes xor No 结果为 Yes(true)
not	非	not (3 < 4) 结果为 No(false)

关系运算符(= / = <=> =)，必须针对两个相同的数据类型值（无论是数值，文本，或布尔值）比较。例如，3 = 4或"Wayne" = "Sam"是有效的比较，但3 = "Mike"则是无效的。

逻辑运算符(AND, OR, XOR)，必须结合两个布尔值进行运算，并得到布尔值的结果。逻辑运算符中的not(非运算)，必须与单个布尔值结合，并形成与原值相反的布尔值。一些有效和无效的决策表达式的示例子如表1-7所示：

表 1-7 决策表达式中逻辑运算符的应用案例

范例	有效或无效
(3<4) and (10<20)	有效
(flaps_angle < 30) and (air_speed < 120)	有效，假设 flaps_angle 和 air_speed 都包含数值数据。
5 and (10<20)	无效 - and 左侧是数值，不是一个布尔值。
5 <= x <= 7	无效 - 因为 5 <= x 的运算结果为布尔值，然后的 true/false <= 7 的关系运算是无效的。

选择控制范例

为了帮助理解选择控制语句，请研究下面的例子。图1-17所示的例子，如果一个学生成绩上了优秀生榜单，然后将显示祝贺- 否则没有显示（因为“No”分支是空的）。

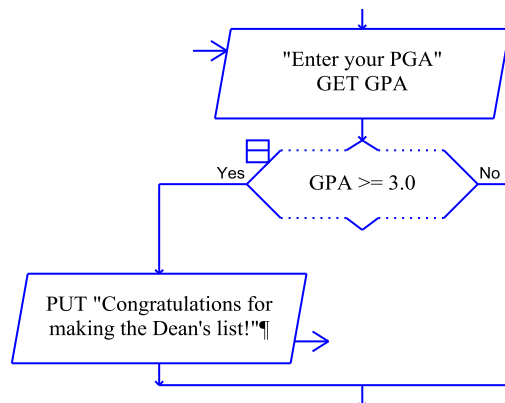


图 1-17 选择控制案例之一

在接下来的例子（参见图1-18），该选择控制案例总归要显示一行文字，并由GPA变量来决定显示哪一行。

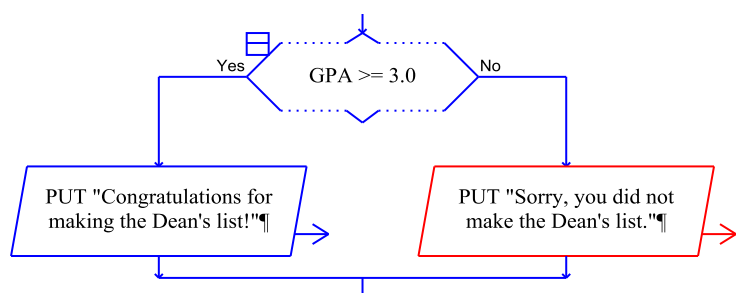


图 1-18 选择控制案例之二

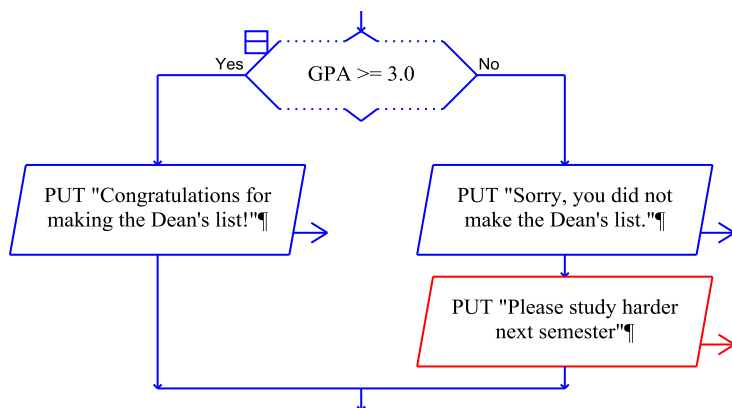


图 1-19 选择控制案例之三

在图1-20所示的例子中，决策表达的逻辑被倒置了。只要确保倒置后的决策可以涵盖所有可能的情况下，这样做完全没有问题。请注意，把“大于或等于”倒置后的逻辑运算，就是简单的“小于（<）”。

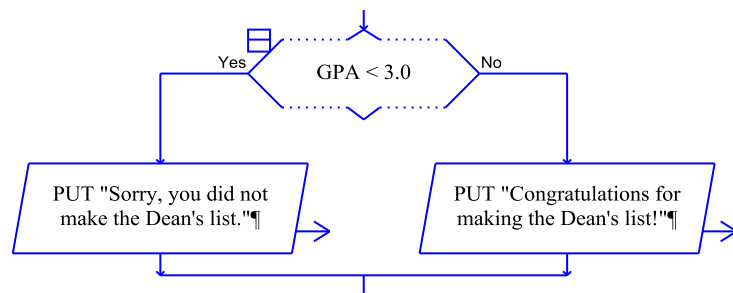


图 1-20 选择控制案例之四

级联选择控制

单一的选择控制语句可以在一个或两个选择之间选择。如果需要做出的决策，涉及两个以上的选择，则需要有多个选择控制语句。例如，如果在数字评分的基础上换算字母（A，B，C，D，或F）等级，就需要在五个选项中选择，如图1-21所示。这有时被称为“级联选择控制”，犹如溪水中的一系列级联的瀑布。

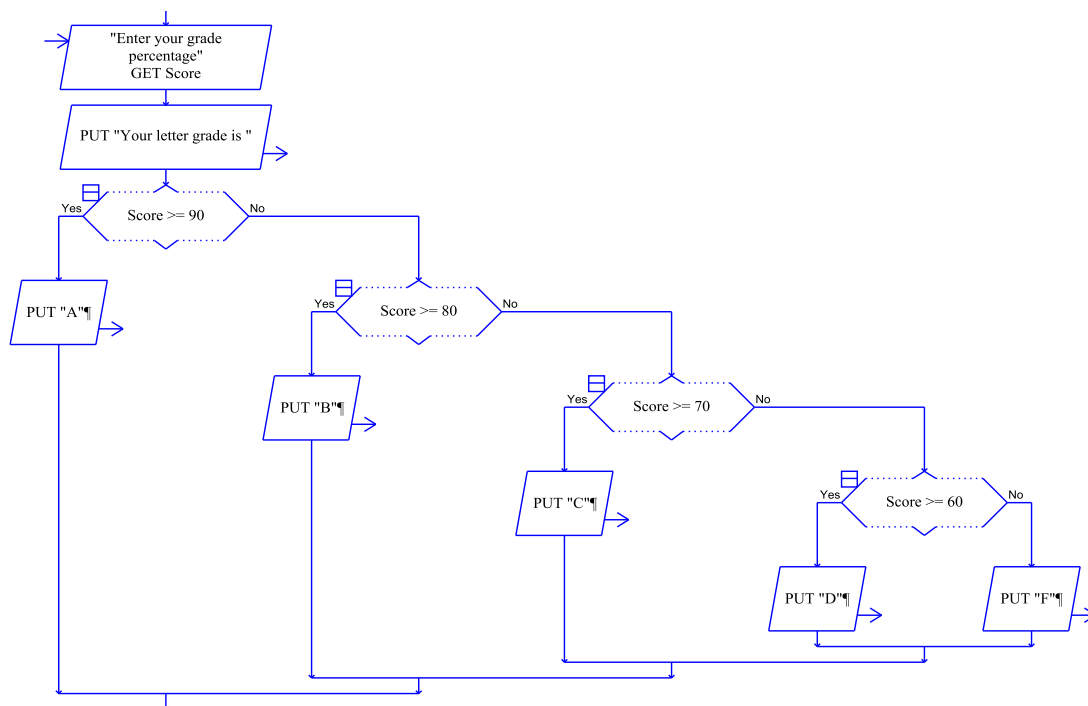
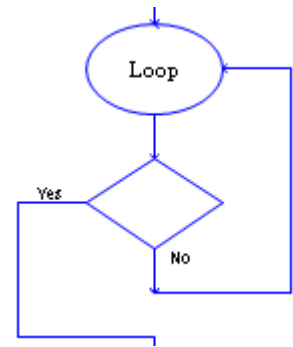


图1-21 级联选择控制

循环（迭代）控制

一个循环（迭代）控制语句允许重复执行一个或多个语句，直到某些条件变为True。这种类型的控制语句使得计算机真正的价值所在，因为计算机可以重复执行无数相同的而不会厌烦。

在Raptor中一个椭圆和一个菱形符号被用来表示一个循环的。循环执行的次数，由菱形符号中的表达式来控制。在执行过程中，菱形符号中的表达式结果为“No”，则执行“No”的分支，这将导致循环语句和重复。要重复执行的语句可以放在菱形符号上方或下方。



为了准确地了解一个循环语句，请参考图1-22的例子，并注意以下情况：

- Statement 1 在循环开始之前执行。
- Statement 2 至少被执行一次，因为该语句处在决策语句之前。
- 如果决策表达式的计算结果为“yes”，则循环终止和控制传递给 Statement 4。
- 如果决策表达式计算结果为“No”，然后控制传递 Statement 3 和 Statement 3 依次执行后控制返回到 Loop 语句并重新开始循环。

请注意，Statement2 至少保证执行一次。而 Statement3 可能是永远不会执行。

这个例子程序有太多的执行路线可能性，以致无法在此一一列出，表 1-8 中列出少数几种可能性是。读者可以在第四列填写某种正确的模式的。

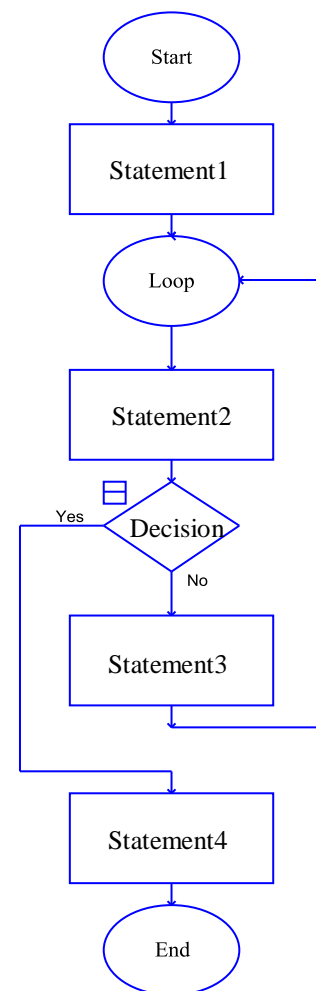


图 1-22 循环控制结构

表 1-8 图 1-22 循环控制结构的执行路线

执行路线 1	执行路线 2	执行路线 3	执行路线 4
Statement 1 Statement 2 Decision ("yes") Statement 4	Statement 1 Statement 2 Decision ("no") Statement 3 Statement 2 Decision ("yes") Statement 4	Statement 1 Statement 2 Decision ("no") Statement 3 Statement 2 Decision ("no") Statement 3 Statement 2 Decision ("yes") Statement 4	(do you see the pattern?)

图1-22的Raptor例子中，“Statement2”是可以去掉的，这意味着循环的第一条语句将是“Decision”（决策）语句。或者，“Statement2”也可以是有多个语句形成的区块。在这两种情况下，循环以同样的方式执行。同样，“Statement3”也可以删除或由多个语句取代。此外，在“Decision”上方或下方，可以是另一个循环语句！如果一个循环语句在一个循环内出现，被称为“嵌套循环”。

“Decision”的语句有可能无法运算出为“YES”，在这种情况下，就会出现永远不停止的“无限循环”。（一旦发生这种情况，用户只能选择Raptor工具栏上的“停止”图标，手动停止程序），当然谁也不期望出现无限循环，因此，在循环中的语句必须变更出现在“Decision”中的变量，使之最后可以运算得到“YES”。

输入验证循环（Input Validation Loops）

循环常见的用途之一用来是验证用户输入。如果用户输入的数据需要满足一定的约束，如输入人的年龄，或输入一个介于1和10之间的数字，程序要验证用户的输入，确保满足约束条件后，才可以将变量值用到程序的某个地方。能在运行时验证用户输入和进行其他错误检查的程序，被称为具有健壮性的程序。

初学者常犯的错误之一是使用选择语句来验证用户输入。这可能无法检测到错误的输入的数据，因为用户可能在第二次尝试输入仍然输入无效数据，因此，将已使用一个循环来验证用户输入。

图1-23显示了Raptor验证用户输入两个方案，希望读者关注此类的格局。几乎每一个验证循环将包括输入提示，决策并输出错误信息。

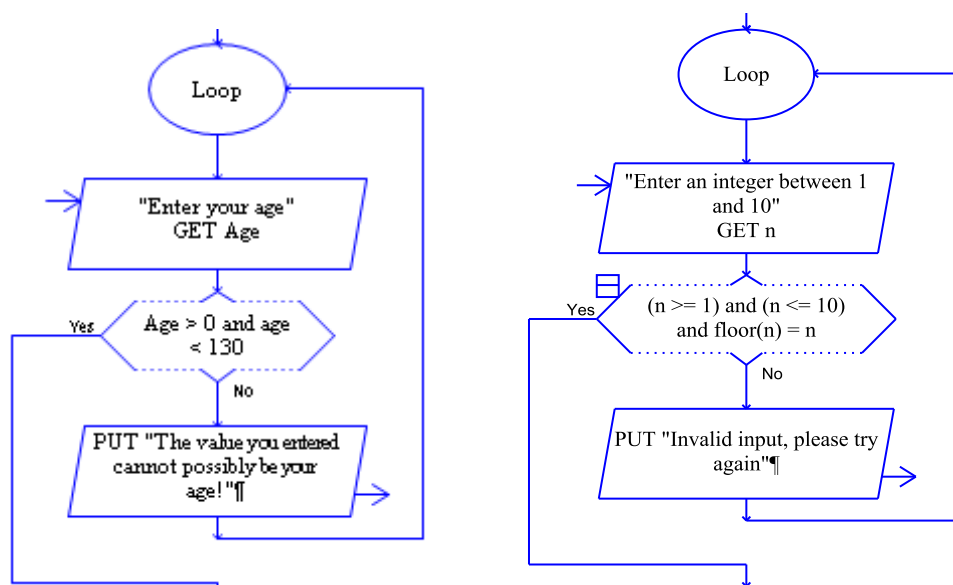


图 1-23 输入验证的两种格局

计数循环 (Counting Loops)

循环常见的模式之一是执行某个代码块某个特定的次数。这种类型的循环称为计数器控制的循环，因为它需要一个计数器变量，在每次循环的执行“计数器加一（或加X，X被称为步长）”。因此，除了loop语句，一个计数器控制的循环需要“计数器”的变量，其基本的操作包括：

- 循环开始前进行初始化，
- 在循环过程内部修改，并且，
- 使用在决策表达式中，用于停止循环。

一个著名的缩写 I.T.E.M (Initialize, Test, Execute, Modify, 初始化，测试，执行，和修改)表示可以用来检查一个循环计数器变量使用是否正确的基本过程。

图1-24显示可计数控制循环执行100次。当观察这个例子，请注意以下要点：

- 在图1-24中，计数器变量为“Count”。其实可以使用任何变量名，但尽量它与任务描述有关。
- 计数器变量在循环开始之前必须初始化为初值。常见的初值为1，但也可以有某个循环执行100次，从20开始计数至119为止。尝试使用适当的初值来解决问题。
- 决策表达式控制循环，通常应使用“大于或等于”测试。这比使用“等于”测试更安全。
- 计数器控制的循环通常每次循环执行一次计数器变量加一。当然可以增加一个其他值，但显然，这将改变循环次数。

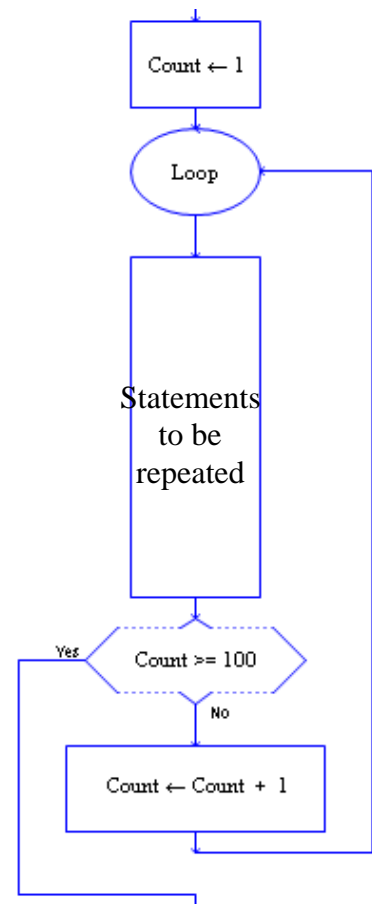


图 1-24 计数循环格局

图2-25显示了三个Raptor实例表明了实施循环时应避免的常见错误。看看你是否能

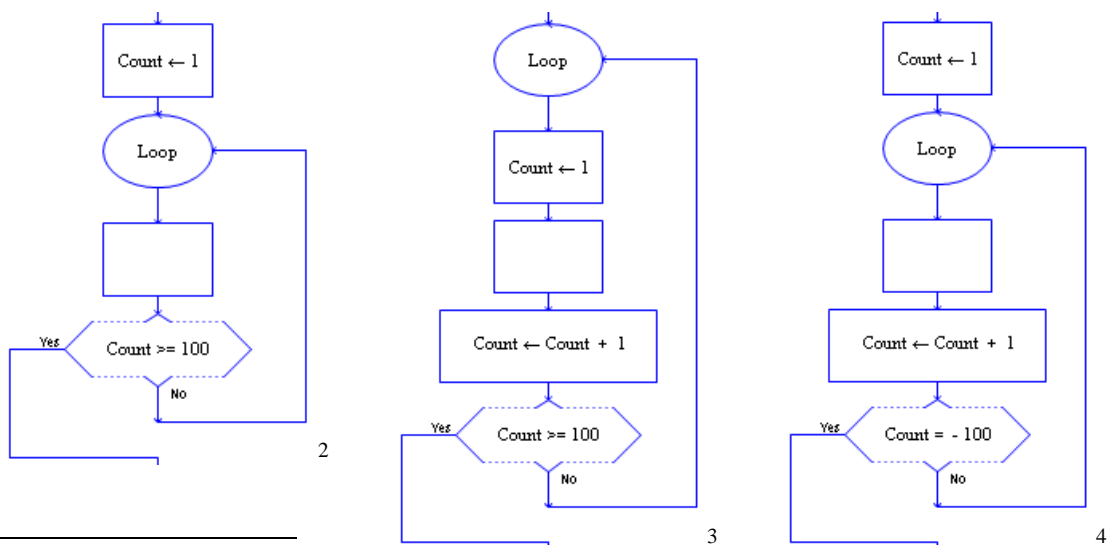


图 1-25 三个存在错误的循环控制结构

² Count 不会改变值，永远等于 1

³ Count 在循环开始后被复位，永远到不了 100.

⁴ 判断表达式结果永远不会是 "yes"

确定在每个程序中的错误。（如果不能发现其中错误，参见在页面底部脚注中的解释。）所有这些方案的问题是会出现无限循环 – 也就是说，一个永不停止的循环。为了避免编写无限循环，必须避免这些常见的错误。

图1-26的示例程序显示一个计数器控制的循环六种变型。它们都做同样的事情 – 执行空框表示的语句Limit（有限）次数。读者可以使用对自己最有感觉的变型。在每个例子中，请特别关注Count的初值和决策表达式。

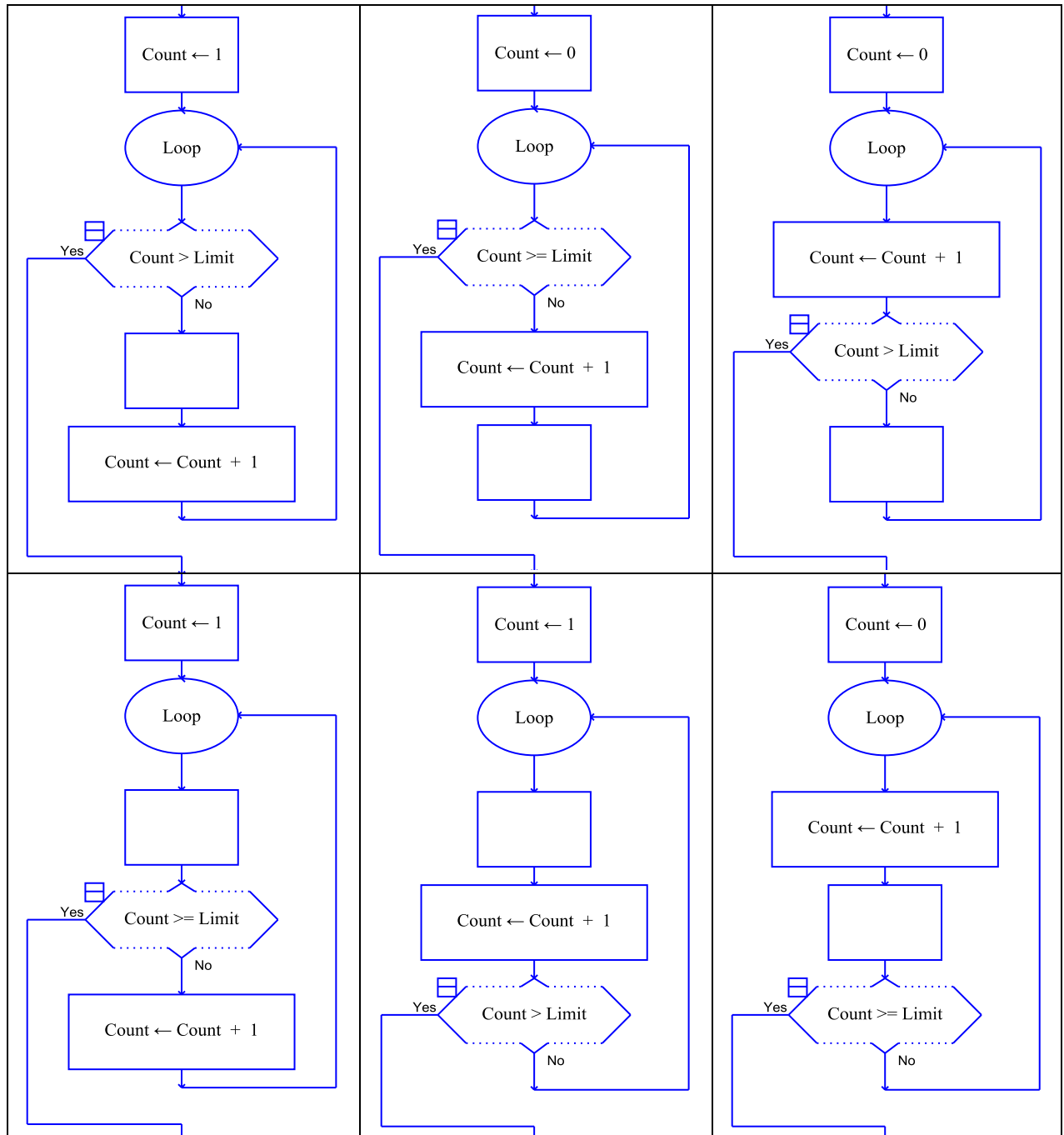


图 1-26 六个相同功能循环控制结构

输入循环（Input Loops）

有时程序需要用户输入一系列值，然后才能进行处理。一般有两种技术来完成这个任务。第一种方法是，让用户进入一个“特殊”的值，表示用户完成数据输入。第二种方法是事先询问用户，要输入多少个值，然后该值可用于实现一个计数器控制的循环。这两种方法都描绘在图 1-27 的样例程序中。图中的空框表示处理输入的数据的语句，不要担心如何处理数据，主要关注的是如何控制用户输入的数据。

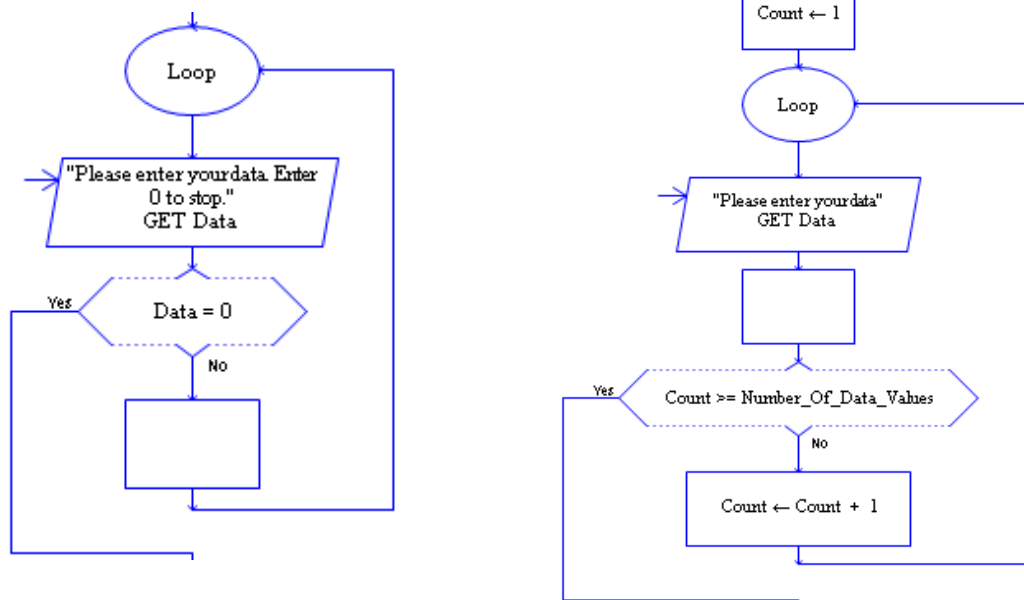


图 1-27 两种输入循环控制方法

“累计”循环 (“Running Total” Loops)

循环的另一个常见用途是累计一系列的数据值，图 1-28 为累计循环的例子。

要累计，必须在循环中添加两个额外的语句：

- 初始化语句，在循环开始之前，累计变量设为零 (0)。
- 赋值语句，在循环内，将某个值加到“累计”变量中。

请确保了解的赋值语句， $\text{Sum} \leftarrow \text{Sum} + \text{Value}$ 的意思。它表示，计算箭头右侧的 $\text{Sum} + \text{Value}$ 的和，然后把结果赋给箭头左侧的 Sum 。变量名使用 Sum ，并没有其他的特殊含义，可以用于任何变量名，如 Total 或 Running_Sum 来替代。

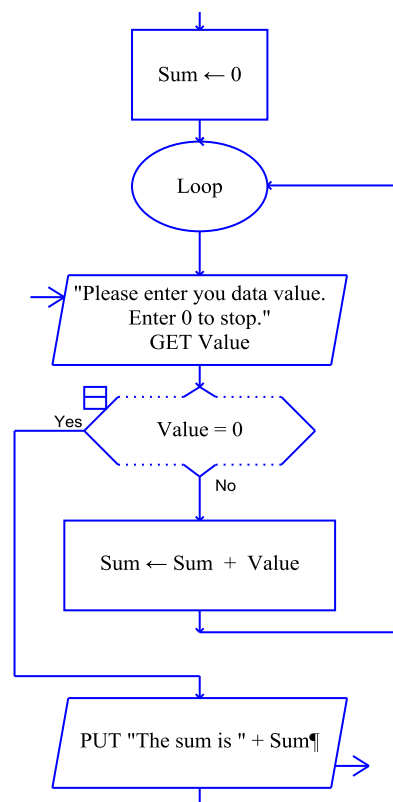


图 1-28 累计循环控制方法

“计数”循环 (“Counting”

另一种常见的用途是对循环本身进行计数。如图 1-29。注意，图 1-28 与图 1-29 有相似之处。

最后两个例子演示了如何编程语句相同的模式，可用于解决各种不同的问题。通过学习和理解简单的例子基础上，将能够用它们解决更复杂的问题。

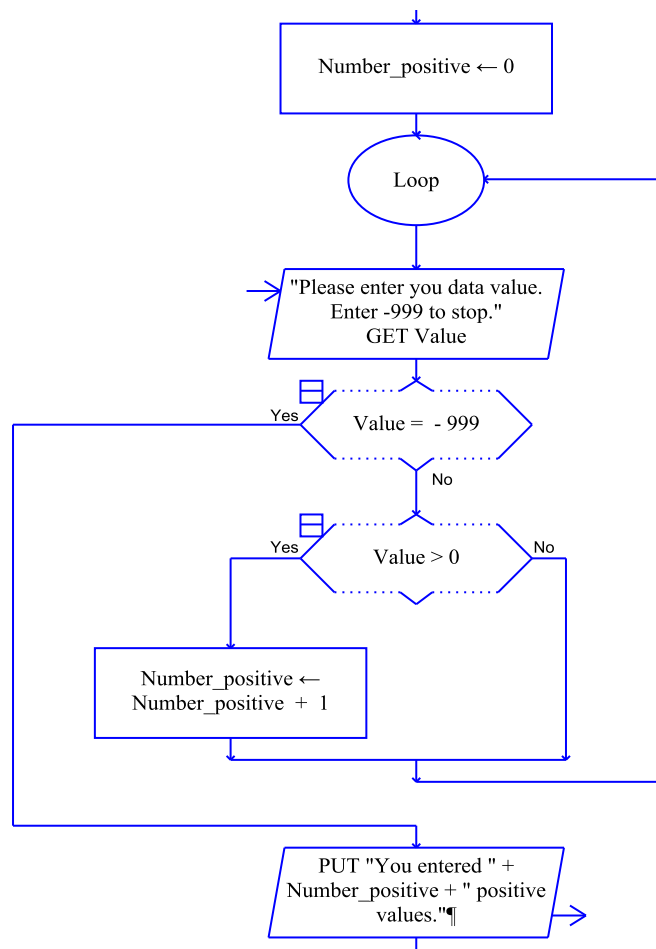


图 1-29 “计数”循环

1.4 数组变量

让我们来考虑一个现实世界的问题 - 确定飞机的重力中心。下面的算法是由无数的飞行员每天使用的例行程序：

1) 对于每个飞机上的组件：

A. 取得该物体的重量，并乘以该物体的力臂获得该物体力矩。

B. 将该物体的力矩加入该到该飞机的总力矩。

C. 加入该物体的重量加入到飞机的总重量。

2) 将总重量除以总力矩获得飞机的重心。

首先，解释一下上述算法中使用的术语：

一个“物体”是任何对象，它是整体飞机的一个组部分。它包括燃料，货物，机员和乘客，空飞机本身等等，每个对象都必须考虑到。每个物体都有与它相关的两个数值 - 它的重量和力臂。物体的“重量”是以磅度量。

一个物体的“力臂”是指物体位于飞机内的位置。就像一个足球在球场上的位置通常是用离某个球队的门线的距离来表示一样，一个物体在飞机上的位置是以距基准线后多少英寸表示。（或者是距离飞机机位的“基准”的距离，指某个飞机的基准参考点）。大部分情况下，单引擎飞机的基准点是安装在发动机上的防火墙。因此，位于防火墙后面的（这是绝大多数）的物体将有正的力臂，而位于防火墙前面的物体将有负的力臂。

现试验一个简单案例，只有三个物体 - 空飞机，燃料，飞行员。

物体 1：1500 磅位于 34.3 英寸。

物体 2：125 磅位于 29.7 英寸。

物体 2：185 磅位于 37.2 英寸。

以下是这架飞机的重心的人工计算结果：

物体 1: $1500 \text{ 磅} * 34.3 = 51450.0 \text{ 吋磅}$
 总力矩: 51450.0 吋磅 总重量: 1500 磅
 物体 2: $125 \text{ 磅} * 29.7 = 3712.5 \text{ 吋磅}$
 总力矩: $51450.0 \text{ 吋磅} + 3712.5 \text{ 吋磅} = 55162.5 \text{ 吋磅}$
 总重量: $1500 \text{ 磅} + 125 \text{ 磅} = 1625 \text{ 英磅}$
 物体 3: $185 \text{ 磅} * 37.2 = 6882.0 \text{ 吋磅}$ 总力矩: $55162.5 \text{ 吋磅} + 6882.0 \text{ 吋磅} = 62044.5 \text{ 吋磅}$
 总重量: $1625 \text{ 磅} + 185 \text{ 磅} = 1810 \text{ 磅}$

重心: $62044.5 \text{ 磅吋} / 1810 \text{ 磅} = 34.3 \text{ 英寸 (基准线后方)}。$

使用数组的动机 - 简单的变量有时使用并不简单。

正如我们已经看到, 在计算机程序中的一个“变量”是内存的一个位置, 可以存储单个数据。事实上, 我们可以在以后检索和/或更改这个数据是计算机和计算机程序的核心价值所在。

我们为了使用 Raptor 实现“飞机重力中心”的算法更加简单, 并减少计算错误的机会。最初, 只考虑三个物体, 就像前面的实验性问题, 程序要求输入每三个物体的信息, 然后报告的最后的总重量和重力中心。图 1-30 中的 Raptor 程序仅使用了以前讨论过的变量类型。

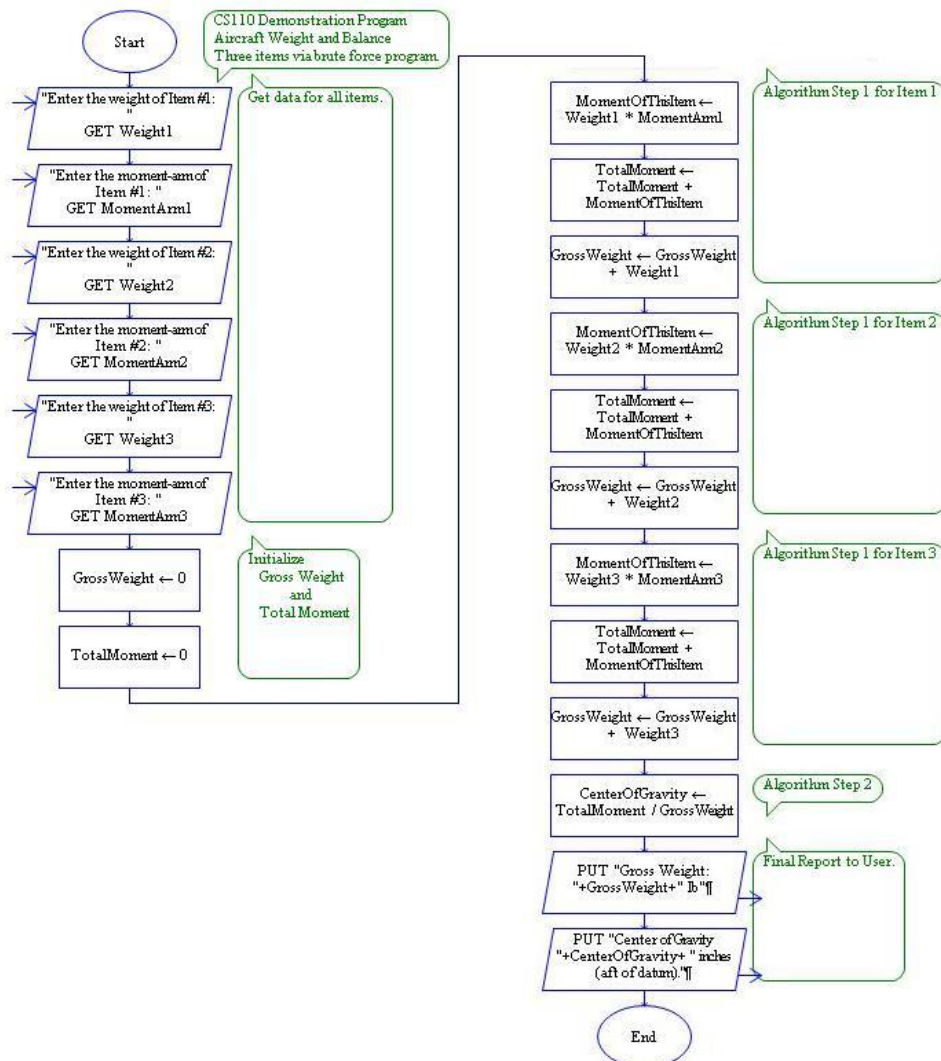


图1-30 蛮力型raptor程序来执行的重量和平衡计算

这个程序是可以工作，但有重大缺陷。其中最重要的是，如果想要三个以上物体参与运算，必须重写程序。其次，仔细检查在此流程图的符号数量可以看出，实施的算法性质很简单，但程序却显得很大。最后，如果我们希望它参与计算的飞机上由数百个物体（如大型货物或客机），这一程序将完全没有实际的使用价值，特别是在程序设计时，还不知道要考虑有多少个物体将要参与计算。

让我们来仔细看看这个程序，看到有什么新的功能能添加到 Raptor 来克服这些缺点。考虑一下获取所有物体的数据的程序部分。这包括了共 6 个符号，但要注意，首对符号与第二对和第三对只有略微不同。事实上，我们已经知道循环，我们可以使用图 1-31 所示的流程图片段的程序完成这部分的预期目标。

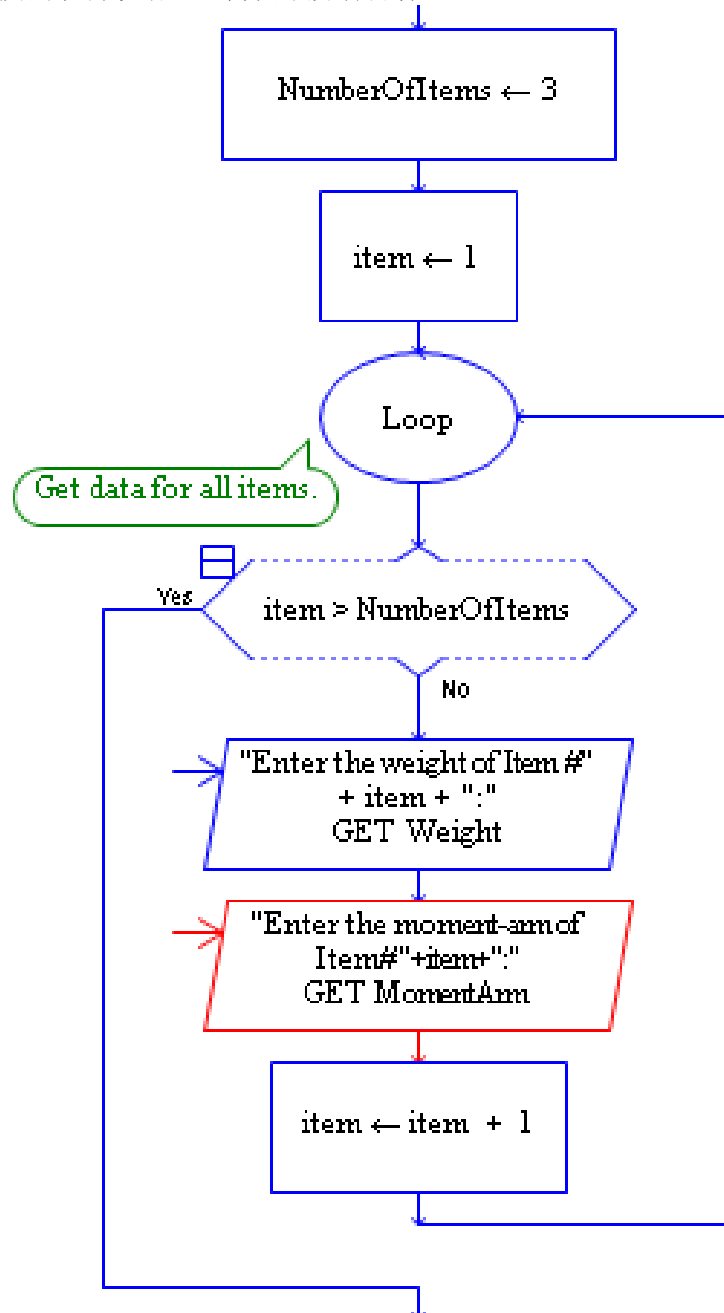


图 1-31 首次尝试使用循环来输入数据

请注意，图1-31的例子不仅要求用户三个重量和三个力臂，它甚至会与原来的程序发出包括物体编号在内的相同的提示。这个循环的唯一问题（也是一个大问题），就是只使

用两个变量，**Weight**（重量）和**MomentArm**（力臂），而实际上需要有一个六个变量（保存六个用户的输入值）。在图1-31的这个循环中，后面每执行一次循环将覆盖以前输入的值，因此，当循环结束后，我们将只保留最后输入的重量和力臂值。

什么是数组变量和数组符号

考虑以下三个简单的变量：

```
Weight1
Weight2
Weight3
```

就Raptor而言，这些都是完全不同的三个变量，每个都能够保存有一个值。每当我们想让其参与操作，必须明确指出该变量的名字。

现在，让我们扩展命名约定，重新命名这些变量如下：

```
Weight[1]
Weight[2]
Weight[3]
```

我们所做出的命名改变约定如下：一个变量名用方括号中的数字（大于零的整数）结尾。每个这样的变量仍然在程序中具有唯一性 – 如同Weight1和Weight2是不同的变量，持有不同的值，Weight[1]和 Weight[2] 也是一样。括号中的数字被称为这一特定的变量的“索引（index）”。

这种变量命名方式通常被称为“数组表示法”。在上面的例子中，我们有所谓的“Weight”数组，具有三个变量。这只是意味着，我们使用数组表示法创建了三个共享相同的根名称（root name）“Weight”的独特变量。另一个数组通常使用的术语是“元素”。我们的数组由三个元素组成；而索引值可以告诉我们，那个元素在某个给定的点上被访问。

除了数组的概念，至少在表面上我们真的有没有改变任何东西。为了表明这种情况，我们可以采取前述的蛮力程序的例子，将变量名称由Weight1改变 成 Weight[1]（当重命名一个变量时需要小心，必须在该变量所有被用到的地方做出相同的改变）而让Weight2和Weight3依旧。同样，也可以把所有的MomentArm2改成MomentArm[2]。该程序将仍然可以运行。

有一个微妙的区别，预先说明一下比较好。如果我们使用简单的变量命名Weight1和Weight2，则没有任何理由不可以使用Weight作为独立变量。但是，当我们使用数组表示法，我们失去了这种任意性。如果我们有一个变量名为Fred[52]（或任何其他索引值），那么我们就不能单独使用一个变量名Fred。然而，比较我们所能获得的极大便利，这只能算一个极小的代价。

如何使用数组变量

数组变量的好处来自数组符号允许Raptor在方括号内的执行数学计算。换句话说，Raptor可以计算数组的索引值。因此，表达式计算所得相同的索引值，均指向相同的变量，例如：

```
Weight[2]
Weight[1 + 1]
```


Weight[23 - 21]
Weight[(5 - 14)*2 + 4*7 - 2*(7 - 3)]

在Raptor数组应用是有一些限制的，在方括号内的表达式可以是产生一个**正整数**的任何合法的表达式，在涉及数组变量时，Raptor 会重新计算索引值的表达式。这才是数组变量和数组表示法的力量所在。

如图1_33所示的代码片段，考虑利用数组后稍加修改所得。唯一的变化是使用Weight和MomentArm数组。但是真正深刻的变化，是使用了表达式，也就是把“item”变量作为数组的索引值，因为每次循环，itemd的值会发生变化。这就实现我们的目标，在不同的提示出现是，可以指向不同的变量。

如图1-34所示Raptor程序执行起来与图1-30中的原始版本完全一样。乍一看这两个方案显得相比符号总数相当，而且有循环出现，基于数组的方案看起来更加复杂。这些看法确实是对的，但是再从一个稍微不同的角度考虑的情况。如果要求创建一个程序，输入四个物体组成的飞机数据？哪一个程序，你更愿意修改？改第一个程序，就必须添加5个新的符号。在第二程序中，你只需要改变开始符号后的第一个赋值符号，使存储在变量“NumberOfItems”的值成为4而不是3。要要命的是，考虑修改第一个程序，以便它可以处理一百余项物品试试！

到了这里，我们已经解决了一些原程序的缺点 - 至少现在我们稍微改变程序，就可以处理不同数量的物体，尽管我们还每次修改程序得修改原程序。但对于程序员来说，在第一个程序中想要处理未知数量的物体是不可能的。但对第二个程序做修改则很简单，从用户那里获得飞机上物件的数量，只需要改变变量“NumberOfItems”的值即可。

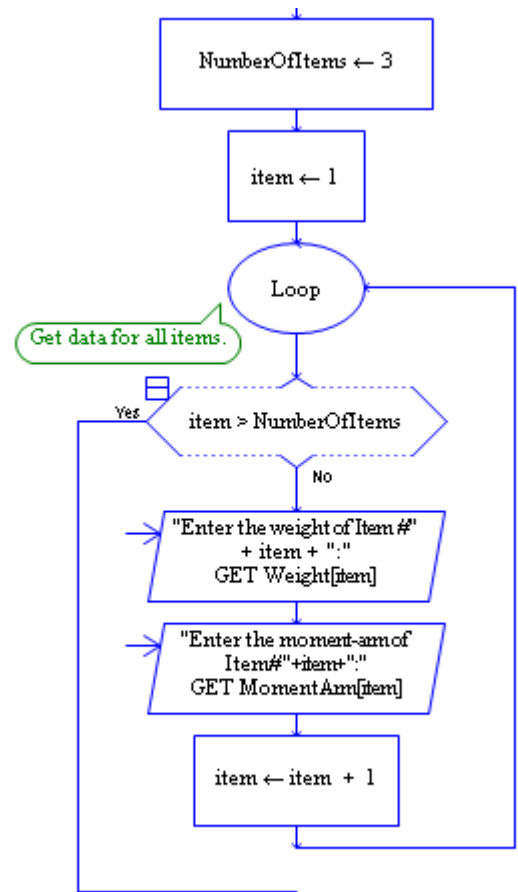


图1-32 Raptor代码使用数组来存储数据

如何在Raptor中创建数组变量

就像Raptor的简单变量，一个数组变量是第一次使用时自动创建的，它是用来存储Raptor中的值。例如，在图4程序中的第一个循环，一旦把某个值保存到Weight [2]，随后即可使用。

Raptor的数组创建也有意思，。如果我们存储一个值，比如说，Fred[100]，然后Raptor将自动创建Fred[1]~ Fred[99]连续99个尚未被创建的变量并在其中存储零。

什么是平行的数组

如果要查询#2物体的重量和力臂的数据，我们会自然而然想到存储在变Weight[2]和MomentArm [2]的值。对不对？

然而，“Weight”和“MomentArm”是两个不同的数组，两者之间并不存在任何必然的联

系。但是，我们必须选择用这样的方式，使它们产生关联。具体来说，为同一个物体中的相关数据存储在各数组相同的索引元素中。采用这种方式存储数据的数组被称为“并行数组”。这个简单的发明希望是一种最自然的、而且能够解决问题的手段。

在编程时有许多情况需要大量的相关数据值，仅使用简单的变量，将会使程序变得非常繁琐或不切实际。数组允许我们使用一个单一的基本名称，结合一个索引值，来访问许多不同的变量。因为每次遇到一个数组变量的引用时，数组索引值会进行表达式计算，这在大量的循环过程中，每次都会同过索引变量，每次循环都会针对各个不同的变量进行操作。

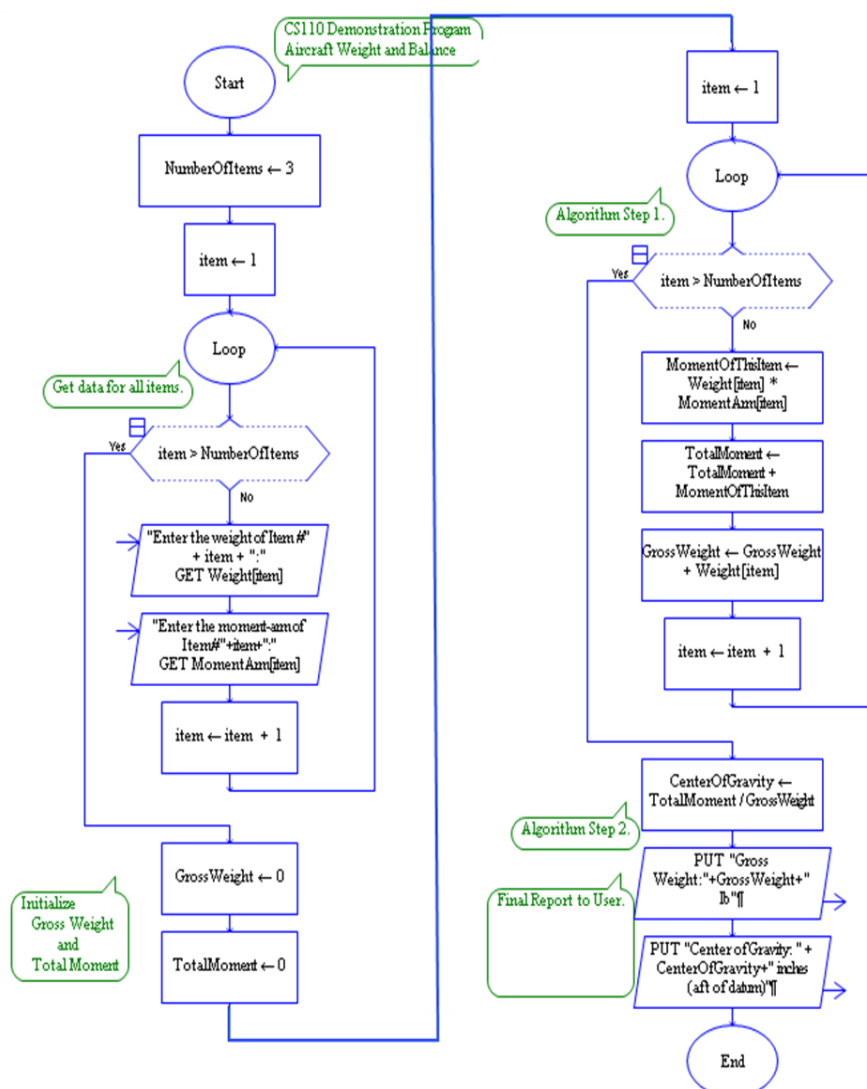


图 1-34 使用数组的重量和平衡程序

1.5 过程调用与图形编程

一个抽象的概念，是一个没有细节的想法。为了加强抽象的概念，考虑下面的例子。技术文件的“抽象”可以解释某个文档的整体内容，但所有细节仍在纸面上。典型的“抽象艺术”绘画是一堆斑驳的色彩，凭籍着一些想象力，可以将其解释为一些可识别的、现实世界中的对象。你的想象力加入所需的详细信息才能将一堆斑驳的色彩转换成可辨认的东西。观察一下图1-35的两个图像，这两者是隐形战斗机的图像。一个包含了精细的细节而另一个则是抽象的。



图1-35 隐形飞机的精细和抽象表达

在计算机科学中，抽象是解决问题的关键要素之一。我们知道，从人的生理研究，人类的大脑平均只能同时积极思考约7件事情。为了解决复杂的问题，必须能够研究问题的“主要方面（big issues）”，而不是所有方面的所有细节。计算机程序设计中，通过组合一系列相关指令，组成分立和离散的过程，就可以抽象所有的细节。

为了介绍过程的概念，本节引入一套过程，允许用用户在计算机屏幕上绘制简单的图形对象。在计算机屏幕上绘制的图形对象，其实需要做大量的工作。但是，做这项工作的算法已在过去开发出来，也没有必要再推倒重来（reinvent the wheel）。因此，如果想在屏幕上画一条线，只需要调用一个包含了所有的画线详细说明的过程。这将解放人的思想，使人们可以专注于一个更高层面上的问题。“抽象”掉画线的细节，可以考虑一下解决当前任务或问题在“更高层次的抽象。”需要强调的是，这是解决问题的一个关键要素。学习开发产生图形输出的简单程序，也是对“过程抽象”的一个认识过程。

RAPTOR图形编程

Raptor图形是一组预先定义好的过程，用于在计算机屏幕上绘制图形对象。所有Raptor图形命令是一个特殊的图形窗口，这是在图1-36所示的例子。可以绘制成图形窗口的各种规格和颜色的线条，矩形，圆，弧和椭圆，也可以在图形窗口中显示文本。

可以通过确定在图形窗口中鼠标的位置，并确定鼠标按钮或键盘键是否点击，与一个图形程序交互。并且，通过在图形窗口中通过多次清屏，并每次重新绘制在稍有不同的位置上，可以在图形窗口中创建动画。

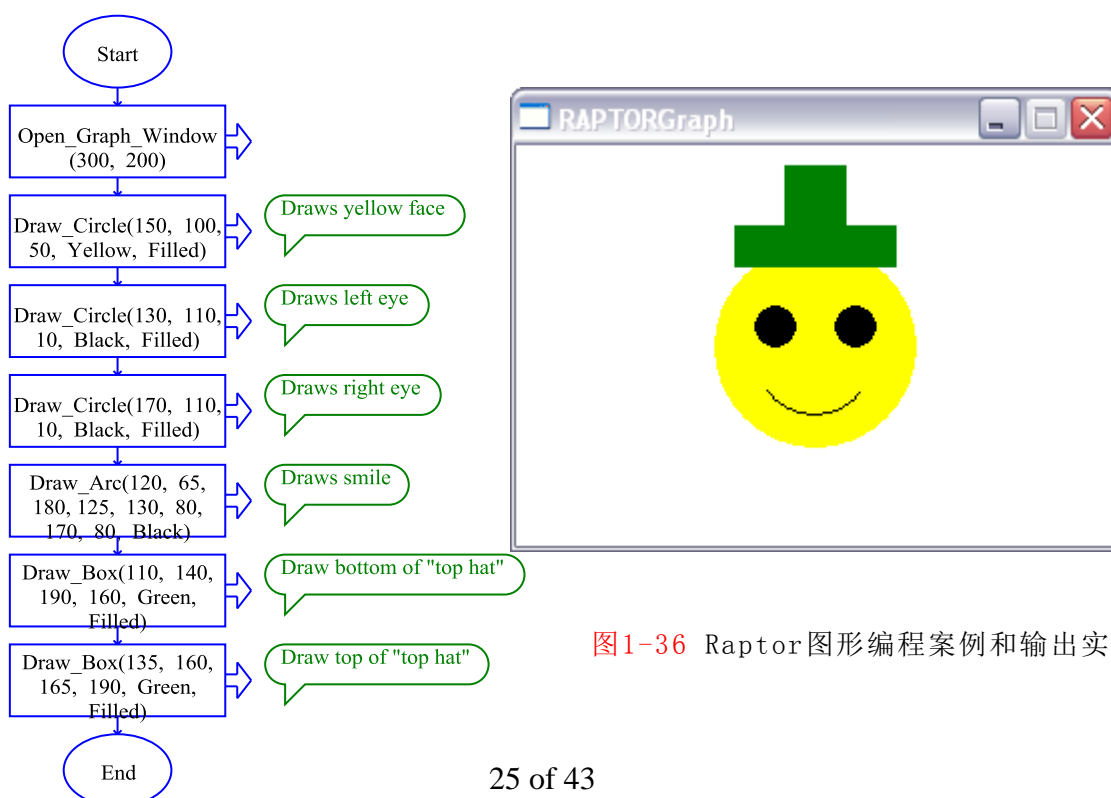


图1-36 Raptor 图形编程案例和输出实例

图形窗口

要使用Raptor图形，必须打开一个图形窗口。调用任何其他Raptor图形过程或函数之前，必须创建此图形窗口。

`Open_graph_Window (X_Size, Y_Size)`

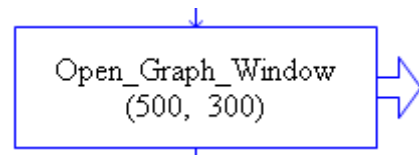


图 1-37 打开图形窗口指令

如果使用图1-37程序中的过程调用，将创建一个宽度为500像素高度300像素的图形窗口。如图1-38所示。请注意，四个角上的坐标位置表达。

图形窗口总是以白色背景。图形窗口 (X, Y) 坐标系的原点在窗口的左下角。X轴由1开始从左到右。Y轴由1开始自底向上。

当程序完成所有图形命令的执行后，应该调用图形窗口删除过程（参见图1-39）关闭图形窗口：

`Close_graph_Window`

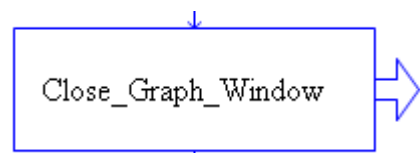


图 1-39 关闭图形窗口指令

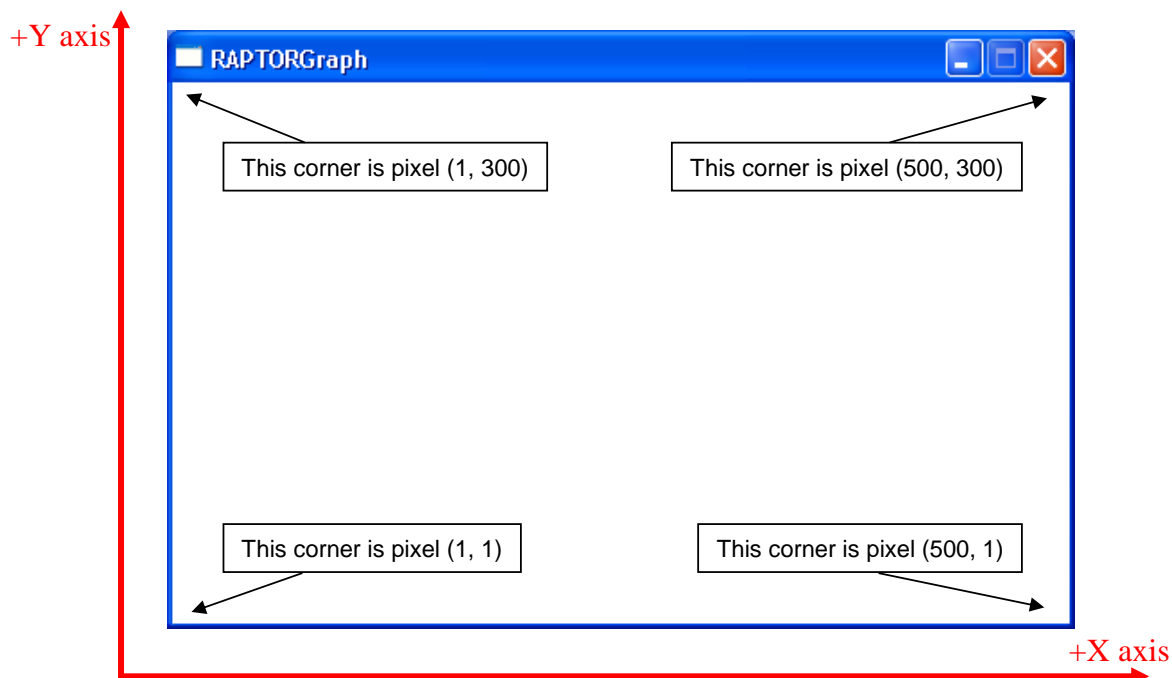


图 1-38 Raptor 图形窗口和坐标表示

例如图1-40的程序所示，打开和关闭通常是图形程序的第一和最后一个命令。如果程序中没有Wait_For_Key命令，则如窗口会在打开后很快关闭，你会看到只有一个简单的实例窗口一闪而过。

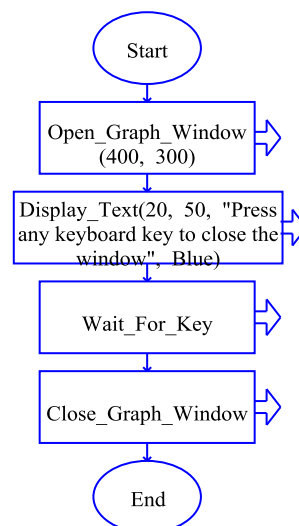


图 1-40 开关图形窗口的实例

绘图命令

Raptor图形有9个绘图过程，用于在图形窗口中绘制形状。这些都是在表1-9中有概括性地说明。所有图形命令绘制以前绘制的图形之上。因此，在绘制图形的顺序是很重要的。所有的图形程序需要设置的参数指定要绘制的形状，大小，颜色，而且，如果它覆盖了一个区域，则需说明是一个轮廓或实体。

表1-9 绘图命令与说明

形状	过程调用和描述
单个像素	Put_Pixel(X,Y,Color) 设置单个像素为特定的颜色
线段	Draw_Line(X1,Y1,X2,Y2,Color) 在(X1,Y1) 和(X2,Y2)之间画出特定颜色的线段
矩形	Draw_Box(X1,Y1,X2,Y2,Color,Filled/Unfilled) 以(X1,Y1)和(X2,Y2)为对角，画出一个矩形。
圆	Draw_Circle(X,Y,Radius,Color,Filled/Unfilled) 以(X,Y)为圆心，以radius为半径，画圆。
椭圆	Draw_Ellipse(X1,Y1,X2,Y2,Color,Filled/Unfilled) 在以(X1,Y1)和(X2,Y2)为对角的矩形范围内画椭圆
弧	Draw_Arc(X1,Y1,X2,Y2,Startx,Starty,Endx,Endy,Color) 在以(X1,Y1)和(X2,Y2)为对角的矩形范围内画出椭圆的一部分
为一个封闭区域填色	Flood_Fill(X,Y,Color) 在一个包含(X,Y)坐标的封闭区域内填色（如果该区域没有封闭，则整个窗口全部被填色）
绘制文本	Display_Text(X,Y,Text,Color) 在(X,Y)位置上，落下首先绘制的文字串，绘制方式从左到右，水平伸展。
绘制数字	Display_Number(X,Y,Number,Color) 在(X,Y)位置上，落下首先绘制的数值，绘制方式从左到右，水平伸展。

绘制的图形对象，可以采用表1-10中的参数表示色彩。

表1-10 图形对象的色彩参数与说明

色彩参数	色彩描述	色彩参数	色彩描述
White	白色	Brown	棕色
Black	黑色	Light_Gray	浅灰色
Red	红色	Dark_Gray	深灰色
Blue	蓝色	Light_Blue	浅蓝色
Green	绿色	Light_Green	浅绿色
Cyan	青色	Light_Cyan	浅青色
Magenta	品红	Light_Red	浅红色
Yellow	黄色	Light_Magenta	浅品红色

另外两个过程修改图形窗口中的某个绘图区域。它们是（表1-11）：

表1-11 两个修改图形窗口的过程

效果	过程调用和描述
清除窗口	Clear_Window(Color) 使用指定的色彩，清除（擦除）整个窗口。
绘制图像	Draw_Bitmap(Bitmap, X, Y, Width, Height) 绘制图像（通过 Load_Bitmap 调用载入），(X,Y)定义左上角的坐标，Width和 Height 定义图像绘制的区域。

绘图过程的参数必须指定在定义它们的顺序。此外，参数可以是以下三种情况之一：

- 一个数值或字符串常量，
- 一个变量，它包含一个适当的值，或
- 一个公式计算出一个适当的值。

以下，分别使用案例加以说明。

常量参数

图1-41中例子显示了使用数字常量指定中心点和三个圆圈半径。请注意的绘图顺序（从大到小）是非常重要的。如果画的顺序是按其他顺序，则得不到这样的效果。

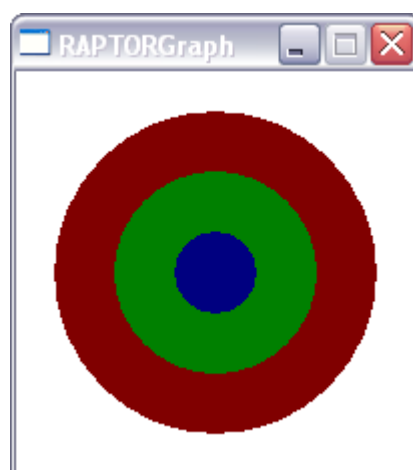
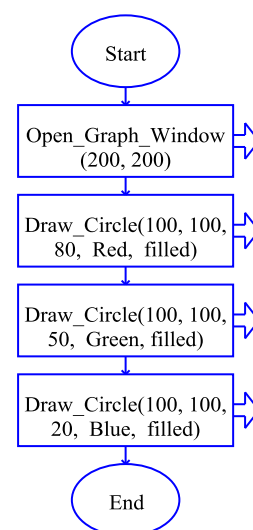


图 1-41 数值常量作为图形命令参数



变量参数

图1-42的例子显示使用可变参数指定了三个圆圈的中心位置。该图所示的图片绘制时，由用户输入 xCenter 值为80，yCenter 值为120。

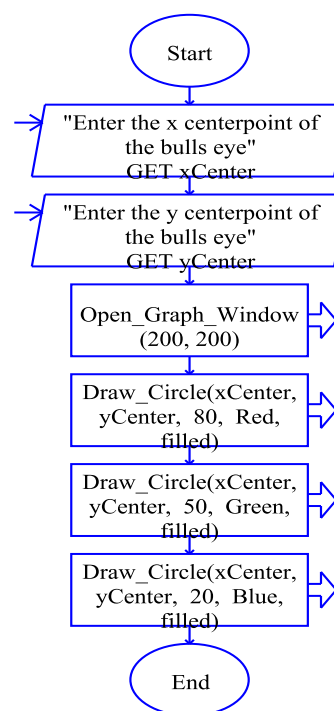
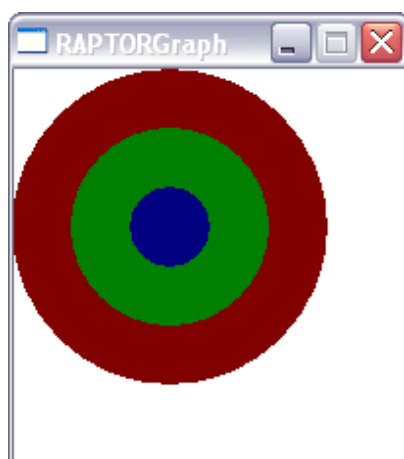


图 1-42 数值变量作为图形命令参数

公式参数

图1-43的例子显示使用指定的三个圆圈半径的方程。该图绘制后，用户输入一个Initial_radius（最大圆的半径）的值为90，Radius_ratio（后两个圆圈的比例）值为0.8。

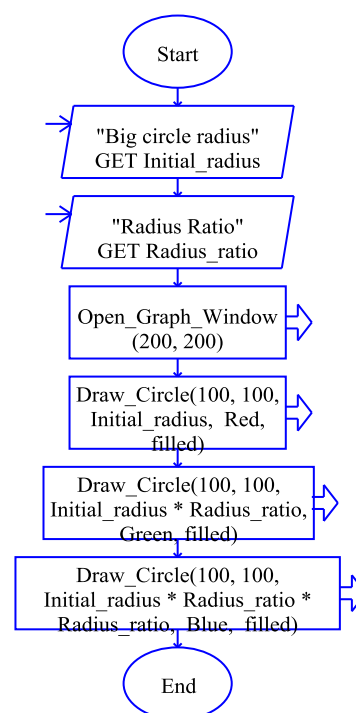
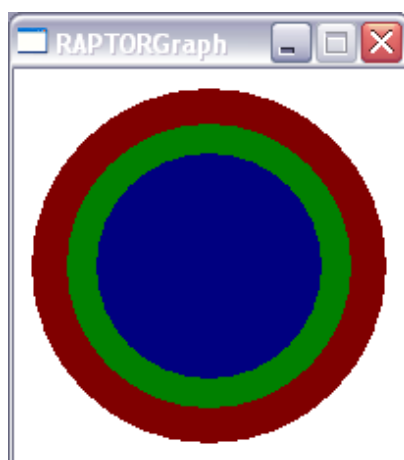


图 1-43 使用公式作为绘图参数

还有三个绘图命令Draw_arc、Flood_fill和Draw_text需要进一步的解释。

Draw_Arc(X1, Y1, X2, Y2, Startx, Starty, Endx, Endy, Color)

Draw_Arc过程绘制出一个椭圆形的一部分。用户必须指定一个矩形来定义整个椭圆形的大小。弧线的起点在从椭圆中心开始一条线的椭圆的 (startx, Starty) 上的交

点。弧线的结束点从椭圆中心点出发的一条线与椭圆在 (Endx, Endy) 上的交点。弧线始终按逆反时针方向绘制。只有在图1-44的例子中的一个Draw_Arc命令绘制了红色弧线。

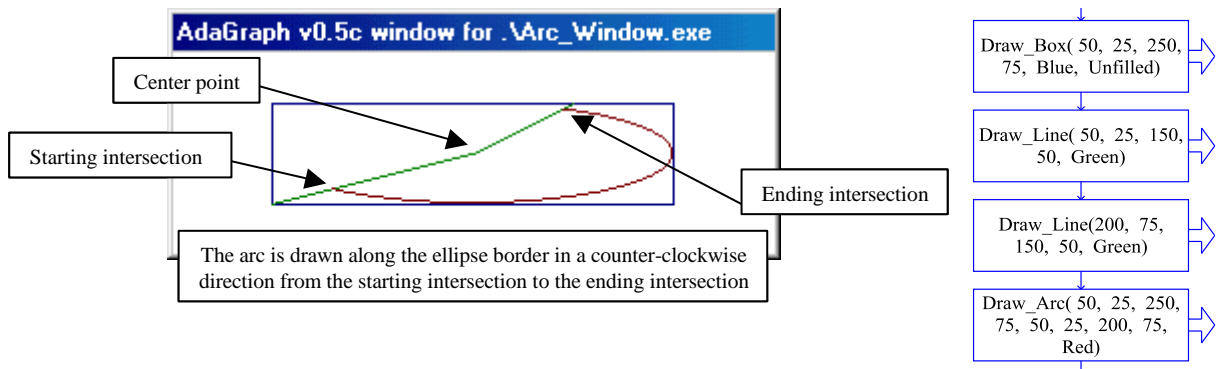


图 1-44 绘制弧线的案例

Flood_Fill(X, Y, Color)

如果需要绘制一个“非标准”的形状，那么你应该使用一系列Draw_Line命令来绘制形状，并完全封闭该区域，然后再用Flood_Fill填充封闭区域所需的颜色。用于绘制边界的颜色，必须不同于Flood_Fill的颜色。必须小心使用**Flood_Fill**命令，因为，如果该地区没有完全封闭，填充色将“泄漏”出来，并可能填满整个图形窗口。图1-45的示例代码，创建下面的黄色三角形。

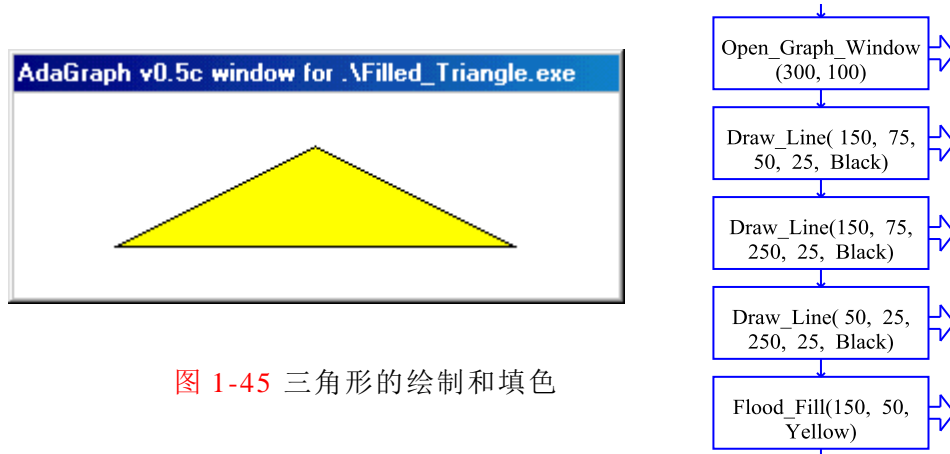


图 1-45 三角形的绘制和填色

Display_Text(X, Y, Text, Color)

图形窗口中绘制文本是相当简单的。但经常需要画一个文本字符串，其中包含程序变量的值。结合常数的文字和变量值使用加号 (+) 运算符，如下面的例子所示，可以建立一个文本字符串。

```
Display_Text(10, 20, "The answer is " + Answer, Black)
```

```
Display_Text(1, 5, "Pt (" + X + ", " + Y + ")", Black)
```

可以使用Set_Font_Size(Height_in_pixels)命令更改绘制文本的大小。默认的文本高度是8像素高。一般在两行文本行之间的垂直方向默认间距的约12像素。

用户交互

如果想与图形程序互动（使用户能够驱动在窗口中发生的事件），您将需要使用一个或多个下列的“输入”函数或过程。一些输入命令将暂停程序运行，直到用户输入为止（称为阻塞型输入）。其它输入命令可以得到有关鼠标或键盘的当前信息，但不暂停执行中的程序。以下说明将输入命令分为鼠标和键盘函数和命令是否暂停程序的执行。

键盘命令

阻塞型输入	过程、函数调用和说明
等待击键	<code>Wait_For_Key</code> 只需等待，直到一个键盘键被按下的过程。
取得用户输入的字符键值	<code>Character_variable ← Get_Key</code> 返回用户输入的字符键。如果没有字符输入， <code>Get_Key</code> 处于等待状态，直到用户击键。
取得用户输入的键值，即使是特殊键如 'home'	<code>String_variable ← Get_Key_String</code> 返回用户输入的串（string）。如果没有输入， <code>Get_Key_String</code> 处于等待状态，直到用户击键。（详情请参阅 Raptor 帮助屏幕。）
非阻塞型输入	过程、函数调用和说明
观察用户是否有击键	<code>Key_Hit</code> 自上次调用 <code>Get_Key</code> 后，如果有键按下，函数返回布尔值 <code>true</code> 。由于 <code>Key_Hit</code> 是一个函数调用，它只能用于赋值语句的“to”部分或在决策语句中。

鼠标输入

阻塞型输入	过程、函数调用和说明
等待按下鼠标按钮	<code>Wait_For_Mouse_Button(Which_Button)</code> 等待、直到指定的鼠标按钮（ <code>Left_Button</code> 或 <code>Right_Button</code> ）按下的过程。
等待按下鼠标按钮并返回鼠标的坐标	<code>Get_Mouse_Button(Which_Button, X, Y)</code> 等待、直到指定的鼠标按钮（ <code>Left_Button</code> 或 <code>Right_Button</code> ）按下，并返回鼠标的坐标位置。例如， <code>Get_Mouse_Button(Right_Button, My_X, My_Y)</code> 等待点击鼠标右键，然后将点击位置赋给进入变量 <code>My_X</code> 和 <code>My_Y</code> 。
非阻塞型输入	过程、函数调用和说明
获得鼠标光标位置的 X 坐标值	<code>x ← Get_Mouse_X</code> 返回当前鼠标位置的 X 坐标的一个函数。它通常用在赋值构造并保存 X 位置到一个变量，以供稍后使用。
获得鼠标光标位置的 Y 坐标值	<code>y ← Get_Mouse_Y</code> 返回当前鼠标位置的 Y 坐标的一个函数。它通常用在赋值构造并保存 Y 位置到一个变量，以供稍后使用。
是否有一个鼠标按钮按下？	<code>Mouse_Button_Down(Which_Button)</code> 如果鼠标按钮现在处于按下位置，函数返回 <code>true</code> 。
是否有一个鼠标按钮按下过？	<code>Mouse_Button_Pressed(Which_Button)</code> 如果鼠标按钮自上次调用 <code>Get_Mouse_Button</code> 或 <code>Wait_For_Mouse_Button</code> 后按下过，函数返回 <code>true</code> 。这通常是用测试是否调用过 <code>Get_Mouse_Button</code> 。
是否有一个鼠标按钮被释放？	<code>Mouse_Button_Released(Which_Button)</code> 如果鼠标按钮从上次调用 <code>Get_Mouse_Button</code> 或 <code>Wait_For_Mouse_Button</code> 后，鼠标按钮被释放，返回 <code>true</code> 的一个函数。

Raptor图形输入的例程

要了解如何使用这些Raptor图形输入例程，请仔细阅读图1-46的Raptor程序。程序首先打开一个图形窗口，并显示一个提示用户按键。然后等待，直到用户按键后继续。一旦用户点击了一个键盘键，程序清除图形窗口，并用黄色背景填充。然后程序等待用户来完成两次鼠标左键点击。它捕获并保存这两个左击中（X，Y）的坐标位置，并保存早（My_X1，My_Y1）和（My_X2，My_Y2）变量。然后，它使用两个左击位置绘制一个蓝色方块。最后，程序等待一个右击，然后关闭图形窗口。

本节内容，并不包括Raptor图形命令的所有细节。更多有关Raptor图形的细节，请参阅帮助屏幕。

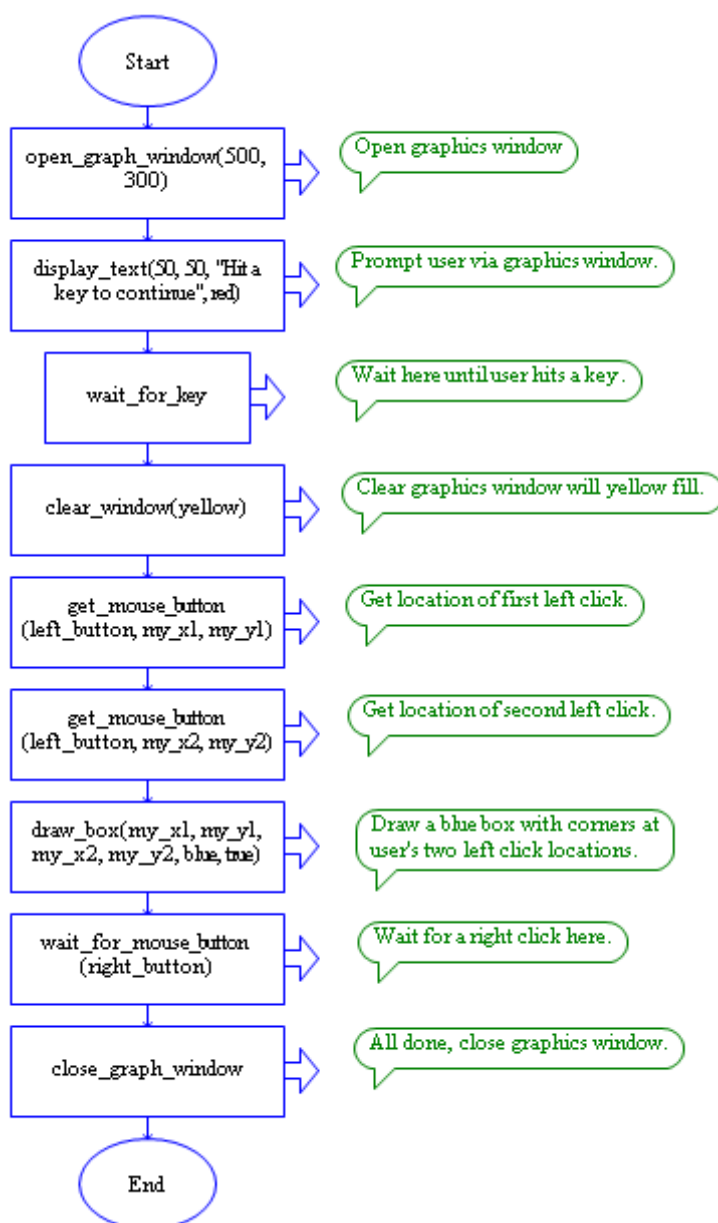


图 1-46 图形程序案例

1.6 功能分解和过程设计

当开发一个由计算机执行的算法时，需要写出在计算机上可以完成的一系列步骤。这些步骤通常是比较简单的指令，而一台计算机能够执行的指令数量有限。这使得完成复杂的任务算法变得很长和晦涩难懂，如果发生错误，也很难修改。

所以，我们能做些什么来处理这种复杂性？嗯，如果电脑执行的算法足够“更加聪明”，甚至能够在一步就解决一个复杂的操作就好了。好了，实际在概念上的计算机可以这样！我们可以将一组指令组合在一起做成一个过程（Process），这些指令可以完成某项任务并具有一个独特的名字。然后，当需要执行这种特定任务时，直接调用这个过程。从概念上，它作为算法的一个步骤来执行。这与上一节中在RAPTOR图形例程的思路是一样的。例如，调用Draw_Circle过程可以画一个圆圈。在Draw_Circle过程内不，它实际上做了很多的各个步骤来创建的一个圆圈，但用户并不需要考虑其中的细节。我们关心的是，应该做的事情已经做了。而其中的细节已被“抽象”掉了，这对一般用户来说很好，因为他

们还有其他的事情需要操心。我们调用这个“过程抽象”，是因为我们定义一个新过程和其中的细节抽象到了过程名字的背后。

如前所述，过程抽象是在处理复杂性时具有巨大的帮助，但它还可以节省大量时间。再次考虑Draw_Circle过程。有人勤奋地编写该过程的各个步骤，来完成相应的任务，他们只需要做一次，所有RAPTOR的所有用户就可以简单地调用这个过程来画圆圈。如果没有有人创建Draw_Circle过程，每次需要绘制一个圆形的时候，每个人都需要去写画圆的所有步骤。这肯定会减缓工作的速度！

RAPTOR的编程环境提供了两种机制来实施过程抽象：子图（subcharts）和子程序（procedures）。一般情况下，编程新手可以比较轻松地创建和理解子图。另一方面，子程序则是一种“增强”型的子图，对新的程序员来说较难理解。子程序允许不同的值在每次调用过程中被“传递”和改变。这些被“传递”的值被称为参数（parameters）。还记得Draw_Circle调用过程中的参数，例如

Draw_Circle(x, y, radius, color, filled)

子程序更容易改变过程调用中的行为，因为每次调用子程序可以发送不同的初始值。

比方说，我们要绘制公牛眼。RAPTOR不包括绘制公牛眼的命令，但我们可以用猛禽的Draw_Circle过程做此事。我们可以调用Draw_Circle若干次来建立我们自己的过程--先画一个有填充色的大圆，然后绘制较小的和更小的。每个圆的中心完全相同，但半径不同。可以把所有这些Draw_Circle调用放在一个叫做过程Draw_Bulls_Eye之中。然后，当需要绘制公牛的眼睛的时候，就只需要调用Draw_Bulls_Eye过程。这件事情的奇妙之处是，我们永远不会再需要操心Draw_Bulls_Eye如何完成这件事情。

子图

最简单的方法来实现RaptorDraw_Bulls_Eye的是使用子图。

要创建一个子图，鼠标光标定位在“main”子图标签（图1_47中红圈所示），点击并按住鼠标按钮。会出现一个弹出菜单。

如果Raptor“模式（mode）”设置为“Novice(新手)”，那么只有“add subchart”的选项将显示在弹出式菜单。如果Raptor“模式”设置为“Intermediate(中级)”，还可以看到一个“add procedure”选项。

选中菜单中的“add subchart”选项，并释放鼠标按钮，将会出现一个新的对话框，提示为子图命名。所有子图和子程序都必须具有一个唯一性的名称。

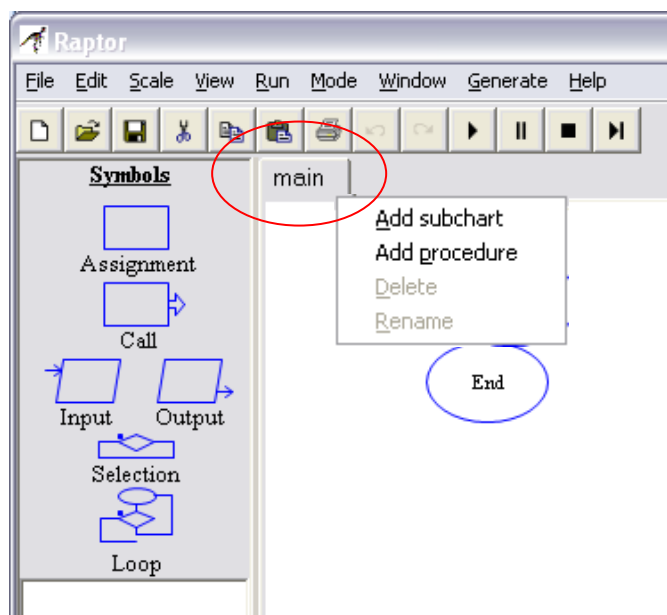


图 1-47 子图和子程序添加菜单

每个新创建的子图将有一个“main”子图选项卡右侧出现一个新的“标签”。图 1-48 中显示了一个方案：把一个程序划分为四个子图，分别为 main，Initialize_screen，raw_characters和Animate。要编辑各个子图，必须单击与及其相关的标签。一次只能查看或编辑一个子图。

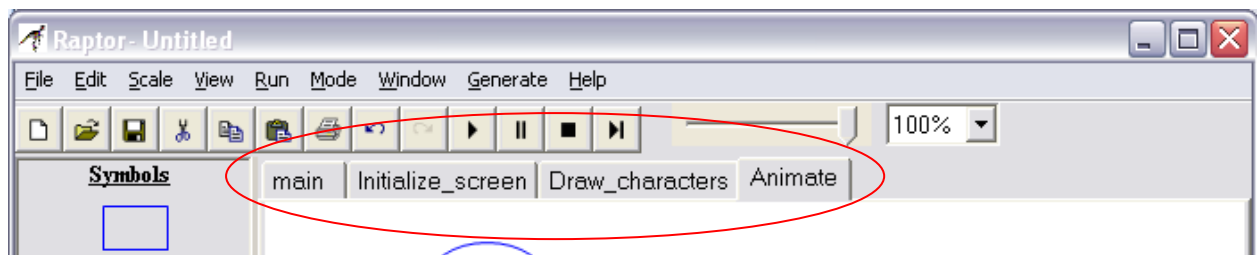


图1-48 具有四个子图的程序

当一个程序开始执行时，它总是首先从“main”子图的“start”符号开始。其它子图都只有当被调用时才执行。图1_49的Raptor说明了这一点。“main”子图显示在最左侧。在其执行过程中，调用了名为“Draw_bulls_eye”的子图（中间）。如果执行这个程序，它会产生显示在右侧所示的图形输出。

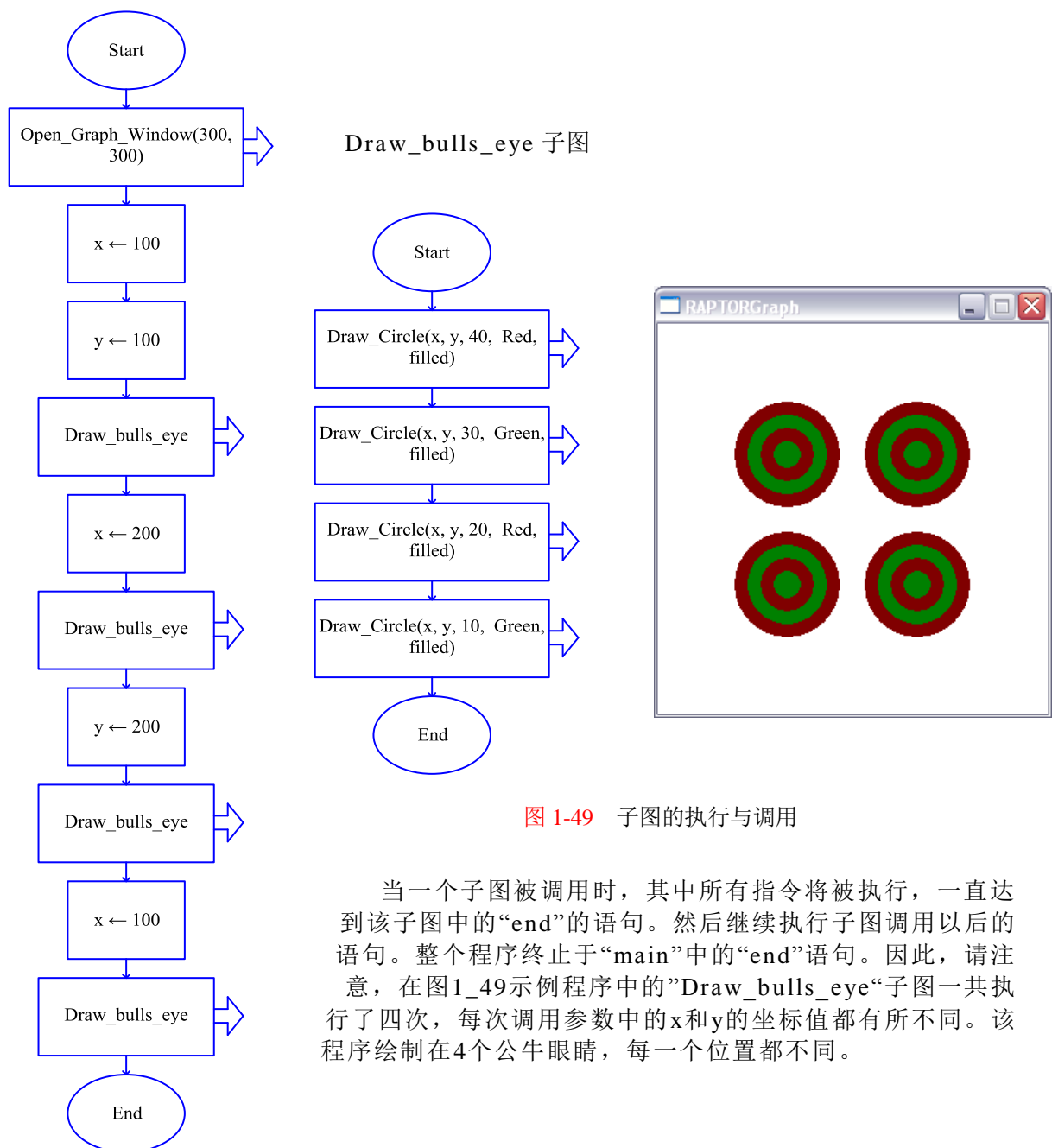


图 1-49 子图的执行与调用

当一个子图被调用时，其中所有指令将被执行，一直达到该子图中的“end”的语句。然后继续执行子图调用以后的语句。整个程序终止于“main”中的“end”语句。因此，请注意，在图1_49示例程序中的”Draw_bulls_eye“子图一共执行了四次，每次调用参数中的x和y的坐标值都有所不同。该程序绘制在4个公牛眼睛，每一个位置都不同。

子图的主要特色是所有子图使用相同的变量。一个变量的值可以在一个子图中改变和然后再在其他子图中使用。在图1_43右边的变量显示区中的X,Y的值的改变可以直观地看到这一事实。请注意，这里只有一个名为X的变量和一个名为Y的变量。

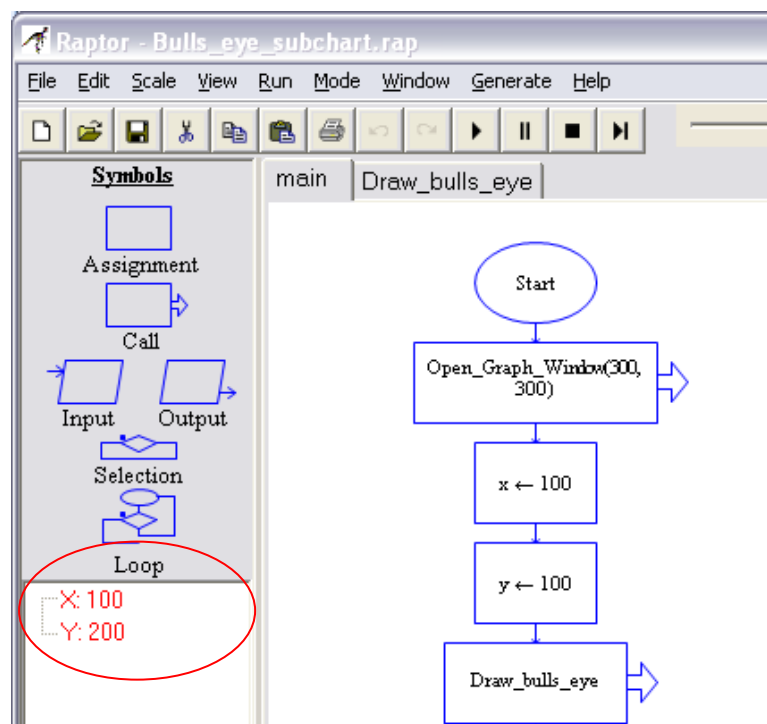


图 1-50 子图中公用的变量

总之，子图可以用来将程序划分成不同的指令集合。这使得一个复杂的程序可以划分为一套更小、更简单的小程序。子图的额外优势之一，就是消除程序中的重复代码。由此可以看出在牛眼绘制程序中，绘制代码只写一次，而不是分别写四次。如果子图划分适当，程序通常会更短，更容易开发，比没有子图的等效程序更容易调试。

子程序

正如上一节所见，使用子图的程序一定更短，更容易理解。然而，调用Draw_Bulls_Eye之前，需要使用单独的赋值符号为X和Y赋值则是很繁琐的。如果能像调用Draw_Circle那样，在调用的过程中传递X和Y的值肯定程序可以更加简化。

那么，好消息是我们可以这样做，但不是使用子图，而是使用“子过程（procedure）”。一个子过程与子图相似的地方在于，因为它是一套独立的，可以通过名字调用执行的指令集合。然而，子过程更加灵活，因为每次被调用时，可以提供不同的初始值。这是通过使用参数（parameter）实现的。一个子程序的参数是在其被调用时收到一个变量初始值，该之在子程序终止时可以发生变化。当一个子程序创建时，必须包含一个子程序的名称及参数定义。为了说明这些想法，请研究图1-51的Raptor程序例子。它产生与图1-49示例程序相同的输出效果。当研究这个例子时，仔细注意以下几点：

- 注意Draw_bulls_eye子程序的Start语句中的文本“(in x, in y)”。这意味着该过程有两个输入参数，X和Y，在子程序被调用时，它们必须给定初始值。
- 请注意，现在每个在“main”子图中出现的Draw_bulls_eye子程序的每一次调用包含一个括号中包含了一个值的列表，例如，（100,200）。第一个值定义的是X的初始值，而第二个值定义是Y的初始值，所以，在子程序调用过程中，参数的顺序是很重要的。在调用子程序语句中，参数初始值的顺序必须始终匹配子程序定义中的参数顺序。

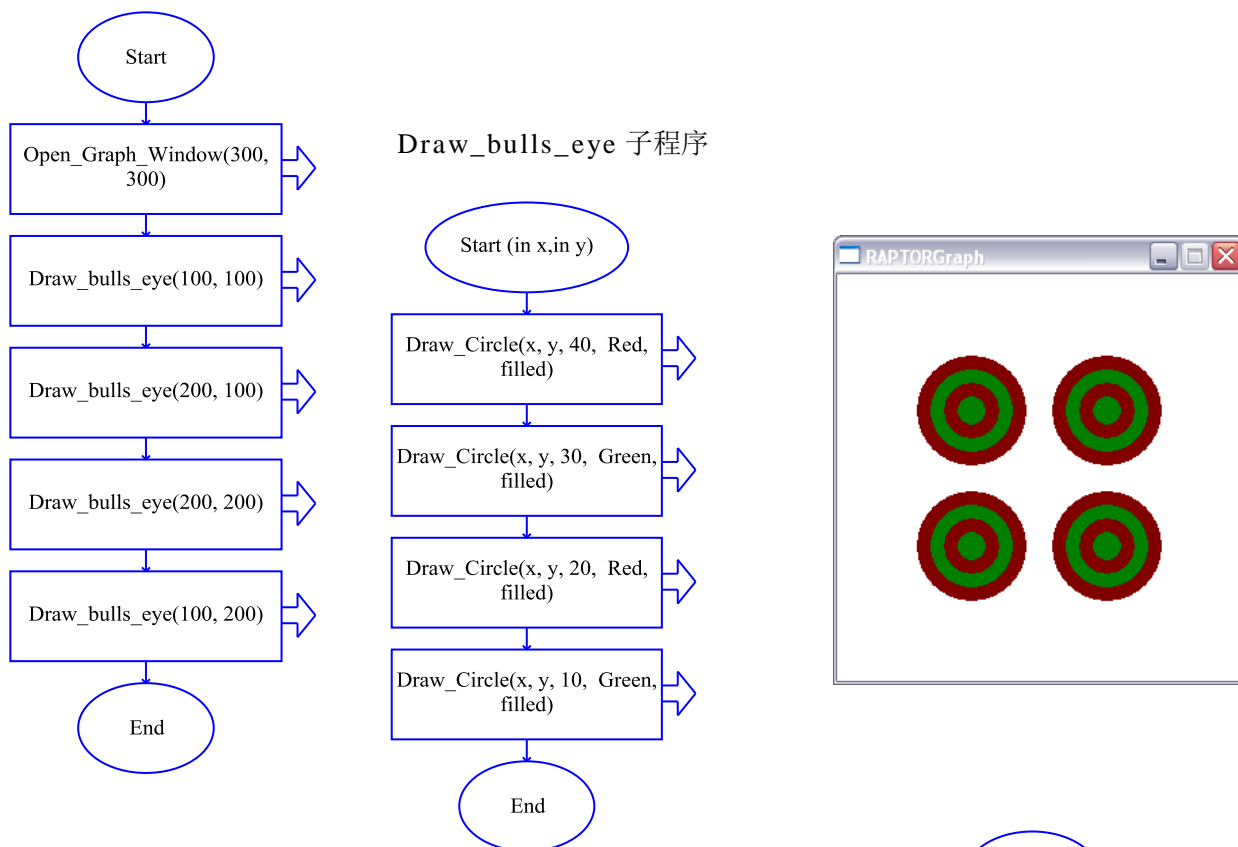
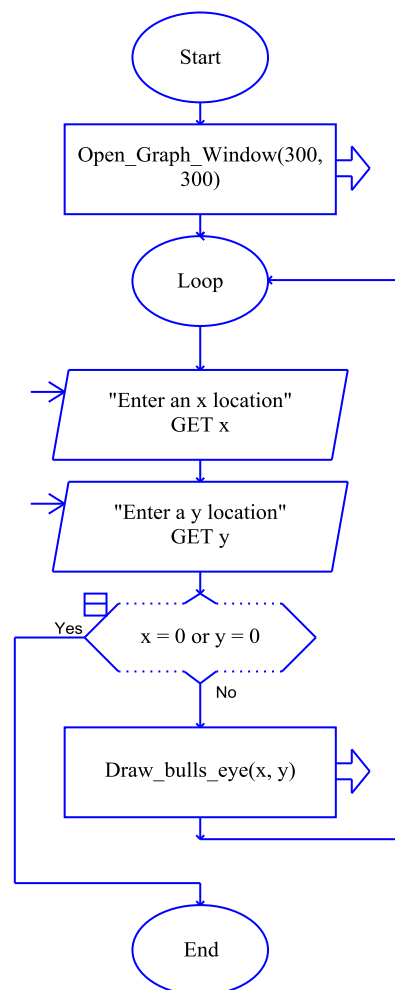


图 1-51 子程序的调用和执行

子程序的主要特点是，各自有其自己的一套独特的变量。如果这两个子程序都包含一个名为X的变量，则各自变量X都会有自己不同的内存位置，一个子程序中的X值的变化将不会影响其他子程序中的X变量值。为了说明这一点，请关注图 1_52中的Raptor程序。这个程序得到用户输入指定的位置（X，Y）上画出牛眼，然后将这个位置发送给Draw_bulls_eye子程序。现在的在“main”子图中有了名为X和Y的变量，而 Draw_bulls_eye 子程序也具有名为X和Y的变量，如果能关注到这些都是独立的变量，那么在使用子程序时，你不会有困难。



如果执行一个包含子程序的 raptor 程序，可以看到新的变量出现在 Raptor 的变量显示区域。这些新的变量将被列在它们所属的子程序的名称下（参见图 1-53）。计算机不会有任何困难将变量与相应的子程序相关联的。希望你也不会遇到困难！

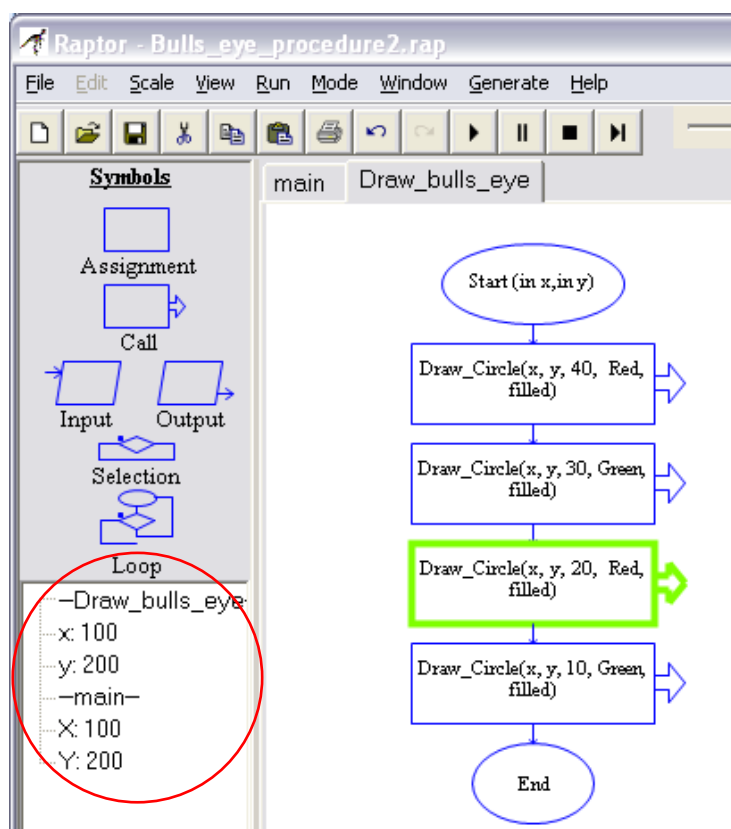


图1-53 子程序中独立的变量环境

过程参数

现在我们已经了解了子程序是如何工作的，下面将解释有关参数的细节。请记住，参数仅仅是一个具有特殊功能的子过程的局部变量。

有三种类型的参数：

- 输入参数 - 它值当程序被调用时被初始化（从调用程序中输入）。
- 输出参数 - 其最终值被复制到调用子程序的程序（子图或子程序）中的变量（返回给调用程序）。
- 输入/输出参数 - 它的值在子程序被调用时初始化，其最终值复制回调用程序的变量（从调用程序输入后，在子程序中处理，然后再返回给调用程序）。

当最初创建一个子程序时，可以指定任意的参数名称和它们的类型。如果一开始不确定需要什么参数，回头可以随时右击子程序选项卡，并选择““Modify procedure””选项来增减和修改参数。

任何问题求解任务的典型场景都是这样的：

- 1、获取一些数据。
- 2、利用这些数据来计算一些解决问题的“答案”。
- 3、发送“答案”进行展示。

此任务序列完全可以与一个子程序的输入和输出参数对应：

- 1、取得子程序的初始数据，建立相应的输入参数。
- 2、使用参数的数据（与其他可能参与的计算的值）来计算合适的“答案”。
- 3、确保包含了“答案”的变量成为子程序的输出参数。

为了帮助这些思路明确，请研究图1-54给出的Raptor程序。

问题：如何创建一个子程序，计算出两点之间的距离。

解决方案：参见图1-54对这个问题的解决方案，并注意以下几点：

- “Distance”的子程序中有4个输入参数，（X1，Y1）和（X2，Y2），如果要执行距离的计算，其中所有的值都是必须的。
- “Distance”的子程序中，有1个输出参数，length（长度），其中包含针对问题的“答案”。
- “Distance”的子程序中有局部变量：dx和dy，这些不是参数。为解决的它所设计来解决的问题，一个子程序尽管可以创建和使用任意多个必要的变量，但这些局部变量只存在于子程序执行之时。一次子程序执行时所得到的局部变量值并不能保留到下一次。
- 例如图1-54中，每次调用语句所使用的参数都不同于上一次。这就允许子程序被多次调用，每次调用的“答案”存储在一个单独的变量中。

参数定义

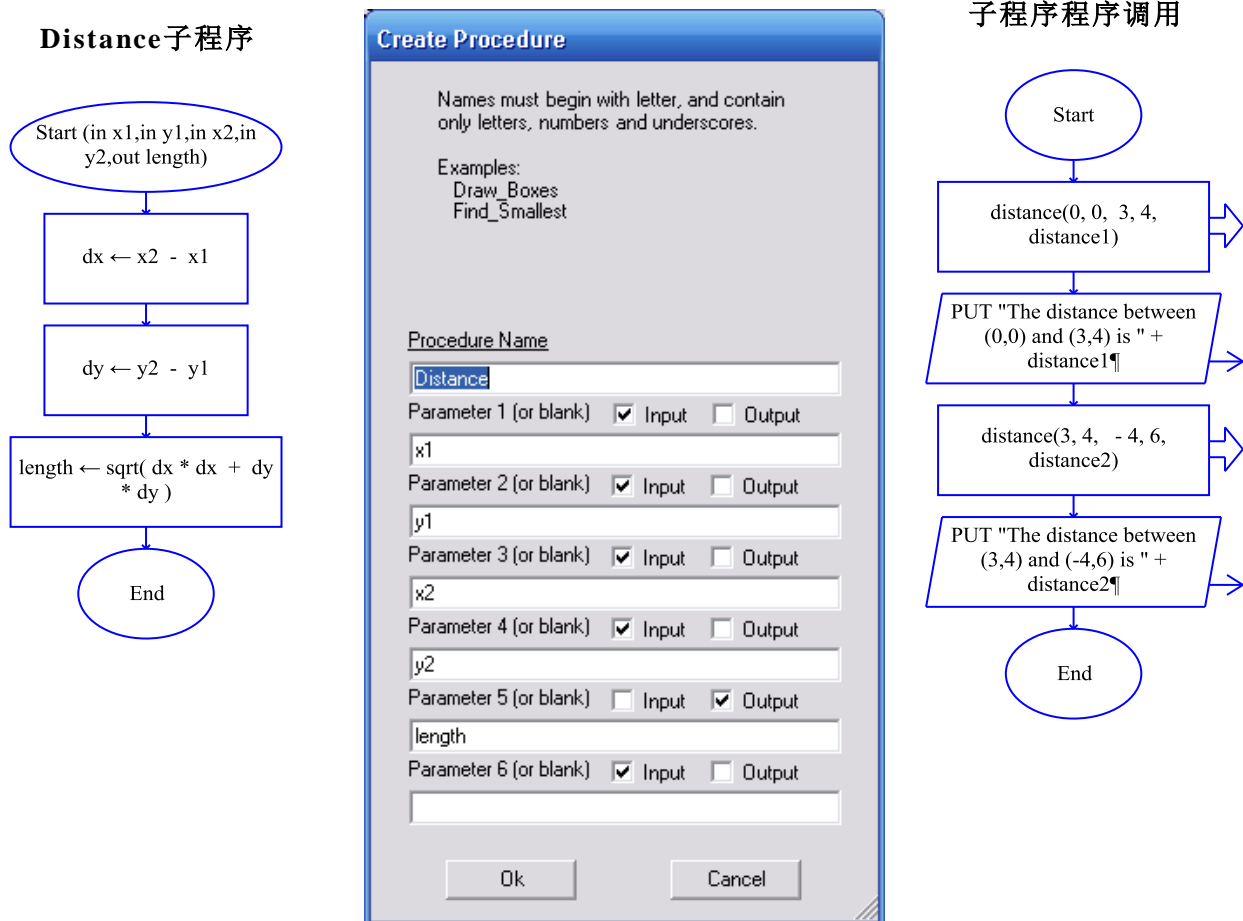


图1-54 子程序参数定义与调用

如前所述，在过程定义中的参数的排列顺序是非常重要的。当一个子程序被调用时，子程序的参数排列必须与定义中顺序相同。顺序确定与值相关的参数。当调用一个子程序时，需要记住两个简单的规则参数：

- 1、如果该过程需要输入参数，调用语句可以使用任何表达式只要能够产生适当的单

个值。

2、如果该子程序需要一个输出参数，调用语句必须使用一个单一的变量。

表1-9 中的例子调用先前定义的“Distance”的子程序，希望能帮助了解这些规则。

表1-9 对“Distance”子程序的调用案例

合法调用	说明
Distance(0,0, 5,-3, answer) Distance(x1,y1, x2,y2, dist1) Distance(x+3, y-7, x*2+5, (y-3)*4, distance2)	
非法调用	
Distance(0,0, 5,-3, 4) Distance(x1,y1, x2,y2, dist1+7)	最后一个参数必须是一个单一的变量，因为它是一个输出参数。

以上介绍了如何将复杂的问题划分成不同部分（子图或子程序）求解的编程方法。如果想让所有子部分使用相同的变量名称，可以使用子图来解决问题。如果想分隔变量，让所有子部分中的变量不与其他部分发生干扰，那么你必须使用子程序。可以使用子图编写较为简单的子程序，但更复杂的子程序，通常需要使用的子程序。

1.7 需求分析

关于流程图的主要优点就是，它们可以明确表示算法的逻辑和流程控制。而用口头和文字描述对程序设计所要解决的问题，则可能隐晦和含糊许多。这些问题描述，也就是对程序的需求，可能有遗漏和不一致的地方，也可能隐藏的一些问题的重要方面。因此，我们必须学会用批判的眼光解读问题所描述的要求，使之成为一个更加明确的要求格式。

本节会给你一些简单的方法和工具，以帮助识别和分析问题陈述中的要求。下面是一个简单的总结：

- 当阅读问题陈述时，写出需求的项目列表。
- 明确列出问题说明中包含的输入、输出和主要处理步骤。
- 当存在指定条件时，确保你知道如何处理被测试的变量每一种可能性
- 当条件涉及多个变量，请制作一个表格，一行代表一个变量，一列表示某（些）种条件。然后填写表应为每个组合条件进行什么样的行动。

在现实中，软件工程师做更多的工作，并使用更多的工具来分析系统的需求。然而，上述四种方法给了我们分析所有类型的需求、算法和流程一个良好的开端。这些方法无论是编写由计算机执行的程序，或由人类执行的过程，都是适用的。让我们来看看上述四种方法更加详细的用法。

列出需求

虽然“列出需求”听起来很琐碎，这是一个非常普遍的第一步。事实上，一些商业过程中过程的“必须”字样，表示所有的强制性要求。考虑下面的叙述，并列出在它的要求。

写一个程序，接受一个学生的GPA（平均成绩），MPA（音乐成绩）和PEA（体育成绩），然后计算并输出相关联的OPA（综合成绩）。OPA的计算公式为“GPA”乘以0.6加上MPA乘以0.3加上PEA乘以0.1。

提取的“需求清单”就变成了：

- 输入GPA
- 输入MPA
- 输入PEA

- 输出OPA
- 计算 $OPA = (GPA * 0.6) + (MPA * 0.3) + (PEA * 0.1)$

列出输入、输出和主要处理步骤

从上述有关OPA的计算例子，可以得到下面的输入，输出和处理步骤，可以描绘为（表1-10）：

表1-10 输入、输出、处理步骤的描述表格形式

输入	处理	输出
GPA	Compute OPA as $(GPA*0.6) + (MPA*0.3) + (PEA*0.1)$	OPA
MPA		
PEA		

确保所有可能的条件问题都已解决

当一个问题需求中涉及某个变量的条件，那么在所有值的可能范围内，一定要弄清楚会发生什么。例如，如果问题的需求说：“如果GPA是3.0或以上，须报告的军校学员是上荣誉榜”的。那么，“GPA”，如果不是3.0或以上又如何？或者，如果GPA不仅是3.0以上，而且是上了4.0又如何？如果历遍了所有可能的值，我们应该拿出像下面这样一张表格（表1-11）：

表1-11 GPA的输入与判断

$GPA < 0.0$	$0.0 \leq GPA < 2.0$	$2.0 \leq GPA < 3.0$	$3.0 \leq GPA \leq 4.0$	$GPA \geq 4.0$
报告出错	报告学生处	不报告	报告荣誉榜	报告出错

使用表格处理复杂条件

当条件复杂，涉及多个变量的当前值，那么好，请确保所有的组合都已经考虑到而且没有冲突。例如，如果一个问题的需求如下：

如果一个学生的GPA是3.0或以上，报告荣誉榜。如果学生的MPA是3.0或以上，报告艺术系。如果学生的PEA在3.0或以上，报告体育系。如果学生的GPA，MPA和PEA的都在3.0或以上，报告上优异生榜单。

这样一个条件清单可以令人眼花缭乱！为了帮助澄清一个问题的需求，制作表格往往是十分有益的。因为这个问题有三个变量，可以这样设计一张表，行地址处理一个变量和列地址处理其它两个。由于其他两个变量可以产生四个组合，所以表格需要需要四列。在表中的各个单元格显示在每种情况下应该是如何报告（表1-12）。

表1-12 复杂条件处理

	$MPA < 3.0$ and $PEA < 3.0$	$MPA \geq 3.0$ and $PEA < 3.0$	$MPA < 3.0$ and $PEA \geq 3.0$	$MPA \geq 3.0$ and $PEA \geq 3.0$
$GPA < 3.0$	不报告	报告艺术系	报告体育系	报告艺术系 报告体育系
$GPA \geq 3.0$	报告荣誉榜	报告荣誉榜 报告艺术系	报告荣誉榜 报告体育系	报告优异生榜单

作为一个练习，考虑应用此技术检测X，Y平面上的一个点，并确定它是否是在一，二，三，四象限。这是颇为简单的，只要检查X是否大于或小于0，并为Y做相同的检查。

但是，如果X和Y有一个或都等于零，又会如何？

让我们再看一个问题陈述的例子：

写一个程序，根据当前的温度和湿度的天气将划分为“好”或“不好”。如果温度是60至80度之间，报告“好”。如果温度是50和60之间，湿度低于30%，报告“好”。如果湿度在80%以上，报告“不好”。

表1-13 将温度的陈述转为表格表示

	湿度< 30%	湿度 30% ~ 80%	湿度 > 80%
温度 < 50			不好
温度 50 ~ 60	好		不好
温度 60 ~ 80	好	好	说法冲突
温度 > 80			不好

当我们这样做了以后，可以发现当温度是60和80之间，湿度在80%以上时，发生了表达上的冲突。同时也看到，有几种组合条件是没有指定动作的。还有一个问题，如果温度正好是60，哪一属于哪种情况？没有建立表格，发现这些含糊不清地方可能很难。

因此，一个求解问题描述底线会被触碰到在：不容易理解，不完整，不够明确等方面。通过列表和制作表格，可以明确将条件，填入表格，然后再努力解决这些问题。本节希望能够学到一些基本的工具来帮助你成为一个更好的解决问题的能手。此外，有关尚未表达要求，不完整的问题陈述，或含糊不清方向的问题不仅存在于计算机科学，在其他的学科中也同样会存在。

功能分解

在前面几节中，已经了解过程的抽象的概念，也就是一部分算法或计算机程序可以被分离出来，并给予一个名字代表它所执行的“过程”。要运行该过程，只需按名称调用即可，而过程运作的细节则都被“抽象”掉了。

从RAPTOR 图形例程，我们一直在使用一种过程抽象形式。当需要在图形窗口上绘制一个矩形时，只需调用Draw_Box绘制即可。而Raptor实际上是如何绘制的矩形的细节已经抽象化了。

通过抽象掉过程中的细节，我们自然就有了处理算法复杂性的工具。具体来说，我们可以把算法分解成不同的子过程，再分别设计各个子过程，需要时直接进行调用。这就像吃饭一样，要一口一口的吃。

这就使我们有了功能分解的概念。有了这个概念，可以把需要编写的程序看成是一个大过程，它可以完成一些功能。而设计这个过程就是写一个算法，它调用一系列子过程，它们各自完成整体功能的某些部分。此时，我们并不关心这些子过程如何完成自己的工作。相反，我们只是假设，它们能够做好各自的工作，稍后再考虑如何做。这样，我们可以将一个过程（或待解决的问题）分解成一系列更小的过程（或问题），这将更易于开发。然后，再以同样的方式细分。最后，成为我们可以直接用编程语言（例如“Raptor”）的实现的过程或函数。

让我们通过一个简单的例子说明如何进行功能分解。假设要求写一个程序，读入一堆分数，并打印出所有超过平均数的得分。在这个问题的最高层次上，只需要做三件事：读入分数，计算平均值，并打印每一个超过平均值的得分。因此在顶层出现了的三个过程，如图1-55所示。

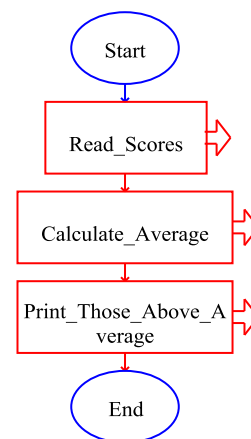


图1-54 顶层过程设计

然后，再分别开发这些子程序（或子图）的逻辑功能，让我们从Read_Scores开始，因为我们需要这个过程首先工作，才有可能做（或测试）的其他过程。

Read_Scores相当简单，可以直接实现，而无需再分解。图1-56是一个子图，可以为Score[]数组读入一个或多个值。

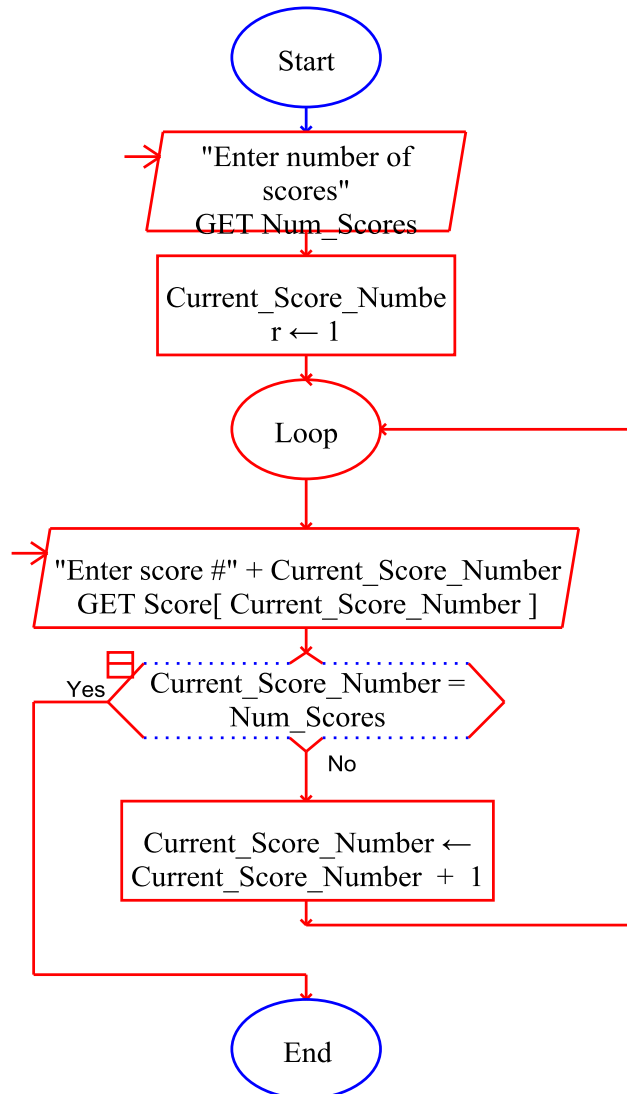


图 1-56 读入分数的子图

图1-56所示的子图不算复杂，总共只有六个符号。让我们再画出其他两个子过程（见下页图1-57，图1-58的两个子图）。这种功能分解是一种非常直观的概念，即使读者可能已经使用很长一段各种各样的形式化手段。这里最大的区别是，我们现在做得是非常正式和明确功能分解，特别是当我们有与其他人沟通的时候，更明确易懂，被误解的机会更少。

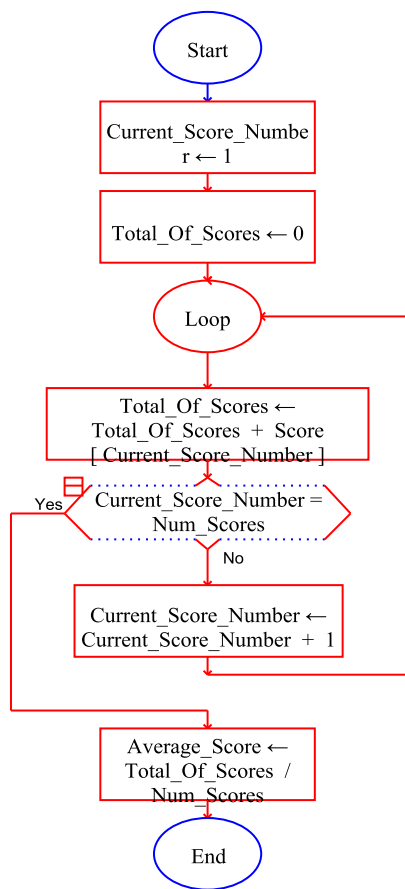


图 1-57 计算平均分

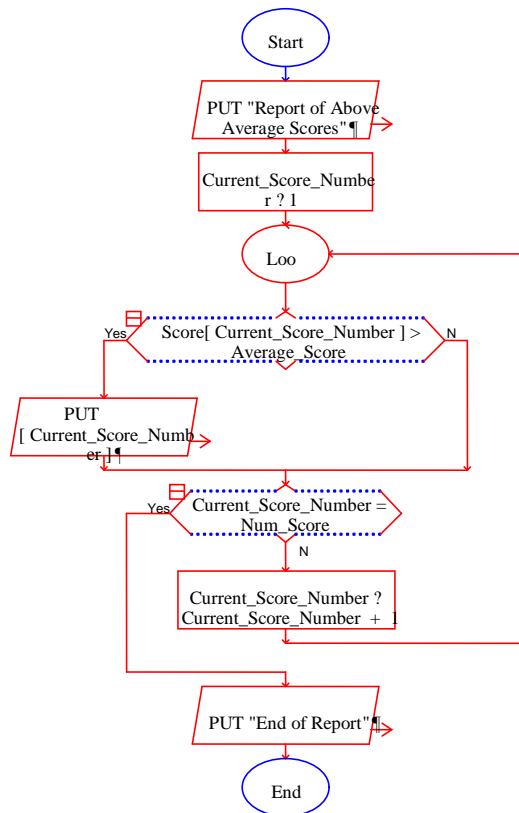


图1-58 打印平均分以上的分数

在最后，我们希望你已经看到了如何分解一个棘手的任务成更小的功能组件的过程，然后分别处理这些组件，可以帮助我们求解非常复杂的问题。事实上，使用功能分解来解决复杂问题的方法，不仅限于只是计算机编程中应用。此外，流程图（如Raptor中所建立的）是一个很好的方式来表达求解过程。事实上，它们正在成为许多新的编程环境，如IBM的WebSphere，用来描述计算过程的基础，因为它们很直观，易于理解。