




# 使用 Object Pascal 开始编程

Free Pascal/Lazarus Book

(苏丹) 穆塔兹·阿卜杜勒·阿齐姆 著  
(美) 帕特·安德森、杰森·哈克尼 编辑  
robsean、方 × × 译

2012 年 8 月 30 日 code.sd

翻译时间：2018 年 7 月 10 日—2018 年 9 月 14 日 [www.lazaruscn.top](http://www.lazaruscn.top)



# 介绍

这本书是为想学习 Object Pascal 语言的程序员编写的，也适合作为新学生或非程序员的第一本编程书。它除 Object Pascal 语言之外，还大体上说明编程技术。

## Object Pascal 语言

1983 年，苹果公司首次让 Pascal 语言支持面向对象编程。在那之后，Borland 将面向对象编程用于他们著名的 Turbo Pascal 产品线。

Object Pascal 是一个通用混合（结构化和面向对象编程）语言。它可以用于一个广阔的应用程序范围，如学习、游戏开发、商业应用程序、互联网应用程序、通信应用程序、工具开发，以及操作系统内核。

## Delphi

在 Turbo Pascal 成功之后，Borland 决定把它移植到 Windows，并且引进组件驱动技术。很快，Delphi 在那时成为最好的 RAD（Rapid Application Development，快速应用软件开发）工具。

Delphi 的第一版在 1995 年发布，它带有一套丰富的支持 Windows 和数据库应用程序的开发的组件和软件包。

## Free Pascal

在 Borland 放弃对 Turbo Pascal 的支持后，Free Pascal 开发组启动了一个开放源码项目：从零开始编写一个 Turbo Pascal 兼容编译器，然后使它兼容 Delphi。Free Pascal 也面向其他平台和操作系统，如 Windows、Linux、Mac、ARM、Win CE。

Free Pascal 编译器 1.0 版本于 2000 年 7 月发布。

## Lazarus

Free Pascal 只是一个编译器，而缺少类似于 Windows 下 Delphi IDE 的集成开发环境（integrated development environment，下文缩写为 IDE）。The Lazarus 项目随后开始，目标是为

Free Pascal 提供一个 IDE。它提供源文件编辑器、调试器，也包含很多类似于 Delphi IDE 的框架、软件包和组件库。

在 2012 年 8 月 Lazarus 1.0 版本发布，但也有很多用 Lazarus beta 版开发的应用程序。很多志愿者为 Lazarus 编写软件包和组件，Lazarus 社区也正在成长。

## Object Pascal 的特点

- 简单易懂，可读性强；
- 编译速度快；
- 产生应用程序可靠、快速，可以与 C/C++ 媲美；
- 使用 Lazarus 或 Delphi 等 IDE，可以写出健壮、大型的应用程序，也并不复杂。

## 作者：穆塔兹·阿卜杜勒·阿齐姆（Motaz Abdel Azeem）

我在 1999 年毕业于苏丹科技大学。我在 BASIC 之后，开始学习第二门编程语言——Pascal。从那时起我就一直使用它。尤其在我学习 C 和 C++ 以后，我发现它是一个简单、强大的工具。在这之后，我转移到 Delphi。从那时起，我的大多数应用程序都是用 Delphi 和 Lazarus 编写的。

我住在喀土穆。我当前的工作是软件开发。

## 第一编辑

帕特·安德森（Pat Anderson）在 1975 年毕业于西华盛顿州立学院，在 1968 年毕业于拉特格斯法学院。他是华盛顿州斯诺夸尔米市的市检察官。1982 年，安德森开始在一台无线电室 TRS-80 三号模型上用内置的 BASIC 解释器编程，不久后又发现了 Turbo Pascal。他购买了 Turbo Pascal 从 4.0 到 7.0 所有的版本，以及 Delphi 从 1.0 到 4.0 的每个版本。安德森从 1998 年到 2009 年中断了编程，Free Pascal/Lazarus 又重新激起了他的编程爱好。

## 第二编辑

杰森·哈克尼（Jason Hackney）是西密歇根大学航空学院的毕业生。他是密歇根东南部的电力公司的一位全职飞行员。哈克尼在 1984 年前后接触到 Commodore 64，之后成为一位

业余程序员。他在 1990 年首次简单接触 Turbo Pascal。最近，在发现 Linux、Lazarus 和 Free Pascal 后，他又重新点燃潜在的编程兴趣。

## 许可/协议

本书的许可/协议是 Creative Commons。

## 本书示例环境

本书所有示例使用 Lazarus 和 Free Pascal。Lazarus 集成开发环境（包括 Free Pascal 编译器）可以从这个站点下载：<http://lazarus.freepascal.org>。

在 Linux 下，也可以从软件库获取 Lazarus。在 Ubuntu 下，可以使用如下命令：

```
sudo apt-get install lazarus
```

在 Fedora 下，可以使用如下命令：

```
yum install lazarus
```

Lazarus 是一个自由和开放源码的应用程序。它可用于很多平台。用 Lazarus 编写的应用程序可以在其他平台上被重新编译，产生用于该平台的可执行文件。例如，在 Windows 中使用 Lazarus 编写一个应用程序后，如果需要产生 Linux 可执行文件，仅需复制你的源代码到在 Linux 下的 Lazarus，然后编译它。

Lazarus 产生每个操作系统下原生的（native）应用程序，不需要附加的库或虚拟机。因此，它易于部署和快速执行。

## 使用文本模式

本书第一章所有示例将是控制台应用程序（也称“文本模式应用程序”“命令行应用程序”），因为它们易于理解和规范化。图形用户界面应用程序将在后面的章节介绍。

# 目录

## 第一章 语言基础

1.1 第一个应用程序 .....	9
1.1.1 其他示例 .....	11
1.2 变量 .....	13
1.2.1 子类型 .....	17
1.3 条件结构 .....	18
1.3.1 if 语句 .....	18
空调程序-18 · 体重程序-20	
1.3.2 Case—of 语句 .....	22
餐馆程序-22 · 用 if 语句重写的餐馆程序-23 · 学生成绩程序-23 · 键盘程 序-24	
1.4 循环 .....	26
1.4.1 for 循环 .....	26
使用 for 循环的乘法表-27 · 阶乘程序-28	
1.4.2 repeat 循环 .....	29
使用 repeat 循环实现的餐馆程序-29	
1.4.3 While 循环 .....	31
使用 while 循环实现的阶乘程序-31	
1.5 字符串 .....	33
1.5.1 Copy 函数——字符串的复制 .....	36
1.5.2 Insert 过程——字符串的插入 .....	37
1.5.3 Delete 过程——字符串的删除 .....	37
1.5.4 Trim 函数——去除空格 .....	38
1.5.5 StringReplace 函数——查找与替换 .....	38
1.6 数组 .....	40
1.7 记录 .....	43

## 第二章 进阶应用

2.1 文件 .....	46
2.1.1 文本文件 .....	47
读取文本文件示例程序-47 · 创建和写入到文本文件-49 · 附加到文本文件-52	
2.1.2 任意存取文件 .....	53
2.1.3 有类型文件 .....	53
写入成绩的程序-53 · 读取成绩的程序-54 · 添加成绩的程序-55 · 创建或 添加成绩的程序-56 · 汽车数据库程序-57	
2.1.4 无类型文件 .....	59
逐字节显示文件内容-59	
2.1.5 文件的复制 .....	62
以字节文件实现复制-62 · 以无类型文件实现复制-63	
2.2 日期与时间 .....	65
2.2.1 日期与时间的比较 .....	67
新闻记录程序-68	
2.3 常量 .....	69
燃油消费程序-69	
2.4 序列类型 .....	71
2.5 集合 .....	72
2.6 异常处理 .....	74
2.6.1 Try—except 语句 .....	74
2.6.2 Try—finally .....	75
2.6.3 人为产生异常 .....	76

## 第三章 结构化编程

3.1 简介 .....	79
3.2 过程 .....	79
3.2.1 参数 .....	80
以过程重写的餐馆程序-81	
3.3 函数 .....	82
使用函数的餐馆程序-82	
3.4 局部变量 .....	84
新闻数据库程序-85	
3.5 以函数作为输入参数 .....	88
3.6 过程和函数输出参数 .....	89
3.6.1 传址调用 .....	90

3.7 单元 .....	92
3.7.1 Lazarus 和 Free Pascal 中的单元 .....	94
3.7.2 程序员编写的单元 .....	94
大流士火星历 - 95	
3.8 重载 .....	98
3.9 排序 .....	99
3.9.1 冒泡排序 .....	99
排序学生成绩 - 101	
3.9.2 选择排序 .....	102
3.9.3 堆排序 .....	104
3.9.4 希尔排序 .....	106
3.9.5 其他排序算法 .....	107
3.9.6 字符串排序 .....	107
学生姓名排序程序 - 108	
3.9.7 排序算法比较 .....	109

## 第四章 图形用户界面

4.1 介绍 .....	116
4.2 第一个图形界面程序 .....	116
4.3 第二个程序 .....	121
4.4 列表框程序 .....	122
4.5 文本编辑器程序 .....	123
4.6 新闻应用程序 .....	125
4.7 带有第二个窗口的应用程序 .....	127

## 第五章 面向对象编程

5.1 介绍 .....	129
5.2 第一个示例：日期和时间 .....	129
5.3 面向对象式的新闻应用程序 .....	134
5.4 队列 .....	140
5.5 面向对象的文件读写 .....	145
5.5.1 使用 TFileStream 复制文件 .....	145
5.6 类的继承 .....	147

# 第一章

## 语言基础



## 1.1 第一个应用程序

在安装和运行 Lazarus 后，我们可以点击 **工程 (P) — 新建工程...** — 程序，以新建一个新程序。

我们将在源代码编辑器窗口中获得以下代码。

```
program Project1;

{$mode objfpc}{$H+}

uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
    cthreads,
    {$ENDIF}{$ENDIF}
    Classes
    { you can add units after this };

{$IFDEF WINDOWS}{$R project1.rc}{$ENDIF}

begin
end.
```

我们可以通过点击主菜单的 **文件 — 保存** 保存这个程序，然后我们可以对它命名，如 **first.lpi**。

随后，我们可以在 **begin** 和 **end** 语句之间写入以下内容。

```
Writeln('This is Free Pascal and Lazarus');
Writeln('Press enter key to close');
Readln;
```

完整的源代码将如下所示。

```
program first;

{$mode objfpc}{$H+}

uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
    cthreads,
    {$ENDIF}{$ENDIF}
    Classes
    { you can add units after this };

{$IFDEF WINDOWS}{$R first.rc}{$ENDIF}

begin
    Writeln('This is Free Pascal and Lazarus');
    Writeln('Press enter key to close');
```

```
Readln;  
end.
```

**Writeln** 语句在屏幕（控制台窗体）显示文本。**Readln** 停止执行，让用户读取显示的文本，直到他/她按 **Enter** 键关闭并回到 Lazarus IDE 中。

按 **F9** 运行应用程序，或者单击按钮：



在运行完这个程序后，我们将获得这些输出文本：

```
This is Free Pascal and Lazarus  
Press enter key to close
```

在 Linux 中，当前目录下会新增名为 **first** 的文件，在 Windows 中则会是 **first.exe**。这些文件能通过鼠标双击直接执行，也可复制到其他电脑上运行，而不需要 Lazarus IDE。

注意：如果控制台应用程序窗口不出现，我们可以从 Lazarus 菜单禁用调试器：先选择 **工具 (T)**—**选项...**—**调试器**，然后在 **调试器类型和路径** 中选择 **(None)**。

### 1.1.1 其他示例

在先前的程序中先找到

```
Writeln('This is Free Pascal and Lazarus');
```

一行，再如下改动。

```
Writeln('This is a number: ', 15);
```

在重新编译后，按 F9 运行这个应用程序。程序将输出如下结果：

```
This is a number: 15
```

可以参照下面的例子分别更改相应行，并且编译、运行。

代码：

```
Writeln('This is a number: ', 3 + 2);
```

输出：

```
This is a number: 5
```

代码：

```
Writeln('5 * 2 = ', 5 * 2);
```

输出：

```
5 * 2 = 10
```

代码：

```
Writeln('This is real number: ', 7.2);
```

输出：

```
This is real number: 7.20000000000000E+0000
```

代码:

```
Writeln('One, Two, Three: ', 1, 2, 3);
```

输出:

```
One, Two, Three: 123
```

代码:

```
Writeln(10, ' * ', 3, ' = ', 10 * 3);
```

输出:

```
10 * 3 = 30
```

在 **Writeln** 语句中写入不同的值，并查看结果，可以帮助我们清晰地理解它。

## 1.2 变量

变量是数据的容器。例如若声明  $X = 5$ ，意味着  $X$  是一个变量，含有值 5。

Object Pascal 是强类型语言。“强类型”表示每个变量赋值前都需要确定其类型。例如，如果变量  $X$  被声明为整型，在整个应用程序的生命周期中就应该仅向  $X$  赋值整数。

下方的代码是声明并使用变量的示例。

```
program FirstVar;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var
  x: Integer;
begin
  x:= 5;
  Writeln(x * 2);
  Writeln('Press enter key to close');
  Readln;
end.
```

这个应用程序的输出将为 10。

注意：此程序使用了新的保留字 **Var**，表示它下方的行是变量声明。

```
x: Integer;
```

这一行有双重意义：

1. 声明的变量名为  $X$ ；
2. 该变量属于整型（Integer），即仅能容纳不带分数的整数，包括负值和正值。

语句

```
x:= 5;
```

意味着向变量  $X$  赋值 5。

下例加入了另一个变量，标志为  $Y$ 。

```
var
  x, y: Integer;
begin
```

```

x:= 5;
y:= 10;
Writeln(x * y);
Writeln('Press enter key to close');
Readln;
end.

```

相应的输出是：

```

50
Press enter key to close

```

其中，50 是公式  $x * y$  的结果。

下例展示了一种新类型，称为字符（character）。

```

var
  c: Char;
begin
  c:= 'M';
  Writeln('My first letter is: ', c);
  Writeln('Press enter key to close');
  Readln;
end.

```

这个类型仅能容纳单个字母，或者作为字符而非数值的单个数字。

下例展示了实数（Real）类型。该类型可以容纳一个分数部分。

```

var
  x: Single;
begin
  x:= 1.8;
  Writeln('My Car engine capacity is ', x, ' liters');
  Writeln('Press enter key to close');
  Readln;
end.

```

为编写更多交互式的和灵活的应用程序，我们需要接受来自用户的输入。例如，我们可以要求用户输入一个数字，并使用 **Readln** 语句（或称“过程”，procedure）获取该数字。

```

var
  x: Integer;
begin
  Write('Please input any number:');
  Readln(x);
  Writeln('You have entered: ', x);
  Writeln('Press enter key to close');
  Readln;
end.

```

本例中，X 的赋值是通过输入完成的，而非应用程序中的常量。

下例为一个乘法表程序，由用户输入相相应的数字。

```
program MultTable;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var
  x: Integer;
begin
  Write('Please input any number:');
  Readln(x);
  Writeln(x, ' * 1 = ', x * 1);
  Writeln(x, ' * 2 = ', x * 2);
  Writeln(x, ' * 3 = ', x * 3);
  Writeln(x, ' * 4 = ', x * 4);
  Writeln(x, ' * 5 = ', x * 5);
  Writeln(x, ' * 6 = ', x * 6);
  Writeln(x, ' * 7 = ', x * 7);
  Writeln(x, ' * 8 = ', x * 8);
  Writeln(x, ' * 9 = ', x * 9);
  Writeln(x, ' * 10 = ', x * 10);
  Writeln(x, ' * 11 = ', x * 11);
  Writeln(x, ' * 12 = ', x * 12);
  Writeln('Press enter key to close');
  Readln;
end.
```

注意，上例中所有在一对单引号中的内容会照原样输出到控制台。如程序中的

```
' * 1 = '
```

及类似内容。其他的变量和表达式则将先求值再输出。

试比较下面两条语句的结果不同：

```
Writeln('5 * 3');
Writeln(5 * 3);
```

第一条语句会将 '5 \* 3' 直接输出，

```
5 * 3
```

而第二条语句则先求值 ( $5 \times 3 = 15$ )，再输出。

下例对两个数字（x、y）进行除法运算，并将结果放置在第三个变量（Res）中。

```
var
  x, y: Integer;
  Res: Single;
begin
  Write('Input a number: ');
  Readln(x);
  Write('Input another number: ');
  Readln(y);
  Res:= x / y;
  Writeln(x, ' / ', y, ' = ', Res);
  Writeln('Press enter key to close');
  Readln;
end.
```

因为本例使用了除法运算（这意味着运算结果可能带分数部分），因此我们将结果变量 Res 声明为浮点数类型。Single 是以单精度（4 字节）浮点数存储的实数类型。



### 1.2.1 子类型

变量的类型也有子类型（subtype）。同一类型（如整型）的各个子类型在可表示的范围和存储空间大小上各不相同。

下表包含各种整型，显示表示范围和存储字节数。

名称	拼写	最小值	最大值	字节数
字节型	Byte	0	255	1
短整型	ShortInt	-128	127	1
小整型	SmallInt	-32768	32767	2
字型	Word	0	65535	2
整型	Integer	-2147483648	2147483647	4
长整型	LongInt	-2147483648	2147483647	4
序号型	Cardinal	0	4294967295	4
64 位整型	Int64	-9223372036854775808	9223372036854775807	8

如下例所示，我们可以分别使用函数 **Low**、**High**、**SizeOf** 得到最大和最小值以及字节大小。

```
program Types;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes;

begin
  Writeln('Byte: Size = ', SizeOf(Byte), ', Minimum value = ', Low(Byte),
    ', Maximum value = ', High(Byte));

  Writeln('Integer: Size = ', SizeOf(Integer), ', Minimum value = ',
    Low(Integer), ', Maximum value = ', High(Integer));

  Write('Press enter key to close');
  Readln;
end.
```

## 1.3 条件结构

智能设备（如计算机等可编程的设备）有一个重要特点：可以在不同条件下采取不同的动作。这种效果可以通过使用条件结构完成。例如，一些车当速度达到或超过 40 km/h 时锁门。在这个案例中该条件将是：

如果 速度  $\geq 40$  而门没有锁，则锁门。

汽车，洗衣机和其他小机器包含微控制器等的可编程电路，或 ARM 等小处理器。这些电路可以分别依照它们的架构，使用汇编、C 或 Free Pascal 编程。

### 1.3.1 if 语句

**if** 条件语句在 Pascal 语言中非常简单清楚。在下面的示例中，程序需要依据输入的房间温度决定空调的开关。

#### 空调程序

```
var
  Temp: Single;
begin
  Write('Please enter Temperature of this room:');
  Readln(Temp);
  if Temp > 22 then
    Writeln('Please turn on air-condition')
  else
    Writeln('Please turn off air-condition');

  Write('Press enter key to close');
  Readln;
end.
```

以上介绍的是 **if—then—else** 语句。在这个示例中，若温度大于 22，则显示第一条语句。

Please turn on air-conditioner

否则，如果条件不满足（小于或等于 22），则显示第二条。

Please turn off air-conditioner

我们也可以编写多个条件，如下所示。

```
var
  Temp: Single;
begin
```

```

Write('Please enter Temperature of this room:');
Readln(Temp);
if Temp > 22 then
    Writeln('Please turn on air-conditioner')
else
    if Temp < 18 then
        Writeln('Please turn off air-conditioner')
    else
        Writeln('Do nothing');

```

你可以用不同的温度值测试上面的示例来查看结果。

我们可以使条件更复杂更有用：

```

var
    Temp: Single;
    ACIsOn: Byte;
begin
    Write('Please enter Temperature of this room: ');
    Readln(Temp);
    Write('Is air conditioner on? if it is (On) write 1,',
        ' if it is (Off) write 0: ');
    Readln(ACIsOn);

    if (ACIsOn = 1) and (Temp > 22) then
        Writeln('Do nothing, we still need cooling')
    else
        if (ACIsOn = 1) and (Temp < 18) then
            Writeln('Please turn off air-conditioner')
        else
            if (ACIsOn = 0) and (Temp < 18) then
                Writeln('Please turn on air-conditioner')
            else
                Writeln('Please enter a valid values');

    Write('Press enter key to close');
    Readln;
end.

```

在上面的示例中，我们使用了新的关键字 **and**。它的意义是，若第一个条件返回真（即 `ACIsOn = 1`），第二个条件也返回真（`Temp > 22`），则执行 **Writeln** 语句；否则，跳转到 **else** 部分。

如果空调连接到计算机（例如通过串行端口），那么我们可以通过应用程序控制开关它，使用串行端口相应的过程或组件。此时，我们需要为 **if** 条件添加额外的参数，如空调已使用的时间。如果它超过限定的时间（假定为 1 小时），那么无论温度如何它都该被关闭。另外，我们可以考虑冷气的流失——如果流失非常慢（例如在夜间），那么空调可以关闭较长时间。

## 体重程序

在这个示例中，我们要求用户输入他/她的身高（以米为单位）和体重（以千克为单位）。程序将依照输入的数据计算该人合适的体重，然后它将输出结果：

```
program Weight;

{$mode objfpc}{$H+}

uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
    cthreads,
    {$ENDIF}{$ENDIF}
    Classes, SysUtils
    { you can add units after this };

var
    Height: Double;
    Weight: Double;
    IdealWeight: Double;
begin
    Write('What is your height in meters (e.g. 1.8 meter): ');
    Readln(Height);
    Write('What is your weight in kilos: ');
    Readln(Weight);
    if Height >= 1.4 then
        IdealWeight:= (Height - 1) * 100
    else
        IdealWeight:= Height * 20;
    if (Height < 0.4) or (Height > 2.5) or (Weight < 3) or (Weight > 200) then
        begin
            Writeln('Invalid values');
            Writeln('Please enter proper values');
        end
    else
        if IdealWeight = Weight then
            Writeln('Your weight is suitable')
        else
            if IdealWeight > Weight then
                Writeln('You are under weight, you need more ',
                    Format('%.2f', [IdealWeight - Weight]), ' Kilos')
            else
                Writeln('You are over weight, you need to lose ',
                    Format('%.2f', [Weight - IdealWeight]), ' Kilos');
            Write('Press enter key to close');
            Readln;
        end.
end.
```

在这个示例中，出现了几个新的关键字。

1. **Double**: 类似于 **Single**, 这个关键字也表示浮点数。但是, **Double** 以双精度浮点数表示, 在内存中占 8 字节, 而 **Single** 仅占 4 字节。
2. **or**: 用于检查是否有满足的条件。如果两项中至少有一项满足, 则返回真。例如: 如果第一个语句 (**Height < 0.4**) 为真, 则下一行的 **Writeln('Invalid values')** 将被调用; 如果第一个语句返回假, 那么它将检查第二个, 以此类推。如果所有语句都返回假, 它将跳转到 **else** 部分。
3. **begin** 和 **end**: 和 **if** 一起使用。**if** 语句在一般情况下将执行一条语句, 但是 **begin—end** 组将多个语句结合为一个“语句块”, 然后通过 **if** 语句依次被执行。

例如, 以下两条语句

```
Writeln('Invalid values');  
Writeln('Please enter proper values');
```

在使用 **begin—end** 组之后, 则相当于一个语句, 就可以被 **if** 语句执行了。

```
if (Height < 0.4) or (Height > 2.5) or (Weight < 3)  
or (Weight > 200) then  
begin  
    Writeln('Invalid values');  
    Writeln('Please enter proper values');  
end
```

4. **Format** (格式化) 过程用于将数字显示为特定格式。在所给程序中, 我们仅需要在小数点后显示 2 个数字。使用该函数时需要在 **uses** 部分添加 **SysUtils** 单元。本例最终输出如下所示。

```
What is your height in meters (e.g. 1.8 meter): 1.8  
What is your weight in kilos: 60.2  
You are under weight, you need more 19.80 Kilos
```

注意: 本例可能并非完全正确。标准体重的详细算法可以从网上找到。该程序仅用于演示使用程序解决实际问题, 以及如何通过问题分析生成可靠的程序。

### 1.3.2 Case—of 语句

条件分支的另一个方法是 **case—of** 语句，依照 **case** 序号值下分支的指令执行。下述餐馆程序将说明 **case—of** 语句的使用。

#### 餐馆程序

```
var
  Meal: Byte;
begin

  Writeln('Welcome to Pascal Restaurant. Please select your order');
  Writeln('1 - Chicken      ($10)');
  Writeln('2 - Fish         ($7)');
  Writeln('3 - Meat            ($8)');
  Writeln('4 - Salad           ($2)');
  Writeln('5 - Orange Juice ($1)');
  Writeln('6 - Milk             ($1)');
  Writeln;
  Write('Please enter your selection: ');
  Readln(Meal);

  case Meal of
    1: Writeln('You have ordered Chicken, this will take 15 minutes');
    2: Writeln('You have ordered Fish, this will take 12 minutes');
    3: Writeln('You have ordered meat, this will take 18 minutes');
    4: Writeln('You have ordered Salad, this will take 5 minutes');
    5: Writeln('You have ordered Orange juice, this will take 2 minutes');
    6: Writeln('You have ordered Milk, this will take 1 minute');
  else
    Writeln('Wrong entry');
  end;
  Write('Press enter key to close');
  Readln;
end.
```

如果我们使用 **if** 条件语句编写相同的应用程序，它将变的更复杂，并且将包含不必要的重复。

## 用 if 语句重写的餐馆程序

```
var
    Meal: Byte;
begin
    Writeln('Welcome to Pascal restaurant, please select your meal');
    Writeln('1 - Chicken      ($10)');
    Writeln('2 - Fish         ($7)');
    Writeln('3 - Meat           ($8)');
    Writeln('4 - Salad           ($2)');
    Writeln('5 - Orange Juice ($1)');
    Writeln('6 - Milk            ($1)');
    Writeln;
    Write('Please enter your selection: ');
    Readln(Meal);

    if Meal = 1 then
        Writeln('You have ordered Chicken, this will take 15 minutes')
    else
        if Meal = 2 then
            Writeln('You have ordered Fish, this will take 12 minutes')
        else
            if Meal = 3 then
                Writeln('You have ordered meat, this will take 18 minutes')
            else
                if Meal = 4 then
                    Writeln('You have ordered Salad, this will take 5 minutes')
                else
                    if Meal = 5 then
                        Writeln('You have ordered Orange juice, this will take 2 minutes')
                    else
                        if Meal = 6 then
                            Writeln('You have ordered Milk, this will take 1 minute')
                        else
                            Writeln('Wrong entry');

    Write('Press enter key to close');
    Readln;
end.
```

在下一个示例中，应用程序评价学生的成绩，并转换为字母等级。<sup>1</sup>

## 学生成绩程序

```
var
    Mark: Integer;
begin
    Write('Please enter student mark: ');
    Readln(Mark);
```

---

<sup>1</sup>原作者为苏丹人，此处改为中国惯用的等级。

```

Writeln;

case Mark of
  0 .. 59: Writeln('Student grade is: F');
  60 .. 69: Writeln('Student grade is: D');
  70 .. 79: Writeln('Student grade is: C');
  80 .. 89: Writeln('Student grade is: B');
  90 .. 100: Writeln('Student grade is: A');
else
  Writeln('Wrong mark');
end;

Write('Press enter key to close');
Readln;
end.

```

在前一个示例中，我们使用 `0 .. 59` 这样的数字范围。这意味着当  $0 \leq \text{成绩} \leq 59$  时，判断条件为真。

注意：**case** 语句仅在有序类型（如整型 `Integer`、字符型 `char`）下工作，而不能在 **string**、`real` 等其他类型下工作。

## 键盘程序

在这个示例中，我们将从键盘中获取一个字母，并且该应用程序将告诉我们该输入的按键在键盘上的行数。

```

var
  Key: Char;
begin
  Write('Please enter any English letter: ');
  Readln(Key);
  Writeln;

  case Key of
    'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p':
      Writeln('This is in the second row in keyboard');

    'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l':
      Writeln('This is in the third row in keyboard');

    'z', 'x', 'c', 'v', 'b', 'n', 'm':
      Writeln('This is in the fourth row in keyboard');
  else
    Writeln('Unknown letter');
  end;

  Write('Press enter key to close');

```



```
Readln;  
end.
```

注意：我们在 **case** 条件中使用了新技巧，它是一系列的值.

```
'z', 'x', 'c', 'v', 'b', 'n', 'm':
```

这意味着如果按下的键是 Z、X、C、V、B、N、M 中的一个，则执行此条 case 分支语句。

我们也可以使用混合范围，如

```
'a' .. 'd', 'x', 'y', 'z':
```

这样。这意味着如果该值落在 a 和 d 之间，或等于 x, y 或 z，则执行语句。

## 1.4 循环

循环用于执行具体的次数，或直到一个条件满足的代码（语句）的某些部分。

### 1.4.1 for 循环

**for** 语句使用一个计数器，使代码循环执行一定的次数的，如下例所示。

```
var
  i: Integer;
  Count: Integer;
begin
  Write('How many times? ');
  Readln(Count);
  for i:= 1 to Count do
    Writeln('Hello there');

  Write('Press enter key to close');
  Readln;
end.
```

**for** 循环使用的变量应为 **Integer**、**Byte**、**Char** 等有序类型。这个变量称为循环变量（loop variable），或循环计数器（loop counter）。循环计数器的始、终值可以是任意的。例如，如果需要对计数器从 5 到 10，则写成这样。

```
for i:= 5 to 10 do
```

我们可以在每个循环周期中显示循环计数器的值，如下例所示。

```
var
  i: Integer;
  Count: Integer;
begin
  Write('How many times? ');
  Readln(Count);
  for i:= 1 to Count do
    begin
      Writeln('Cycle number: ', i);
      Writeln('Hello there');
    end;

  Write('Press enter key to close');
  Readln;
end.
```

注意：这次我们需要让两条语句循环，所以需要使用 **begin—end** 关键字进行组合。

## 使用 for 循环的乘法表

**for** 循环版本的乘法表更简单和更简洁：

```
program MultTableWithForLoop;

{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var
  x, i: Integer;
begin
  Write('Please input any number: ');
  Readln(x);
  for i:= 1 to 12 do
    Writeln(x, ' * ', i, ' = ', x * i);

  Writeln('Press enter key to close');
  Readln;
end.
```

在这个版本中，我们不需要写 12 条输出语句，而只需写 1 条循环 12 次的语句。

我们可以使用 **downto** 关键字代替 **to** 关键字，使循环计数器递减。

```
for i:= 12 downto 1 do
```

## 阶乘程序

一个整数的阶乘是从 1 到它本身的所有整数的积，如  $3! = 3 \times 2 \times 1 = 6$ 。

```
var
  Fac, Num, i: Integer;
begin
  Write('Please input any number: ');
  Readln(Num);
  Fac:= 1;
  for i:= Num downto 1 do
    Fac:= Fac * i;
  Writeln('Factorial of ', Num, ' is ', Fac);

  Writeln('Press enter key to close');
  Readln;
end.
```

## 1.4.2 repeat 循环

**repeat** 循环与 **for** 循环不同：与重复指定次数的 **for** 循环不同，**repeat** 循环没有计数器。这种循环持续到某一条件成立（返回真值）为止，在结束后跳转到下一条语句。

**repeat** 循环的示例如下所示。

```
var
  Num : Integer;
begin
  repeat
    Write('Please input a number: ');
    Readln(Num);
  until Num <= 0;
  Writeln('Finished, please press enter key to close');
  Readln;
end.
```

在上例中，程序进入循环，然后它要求用户输入一个数字。如果数字小于或等于 0，则退出循环；如果数字大于 0，则循环将继续。

### 使用 repeat 循环实现的餐馆程序

```
var
  Selection: Char;
  Price: Integer;
  Total: Integer;
begin
  Total:= 0;
  repeat
    Writeln('Welcome to Pascal Restaurant. Please select your order');
    Writeln('1 - Chicken      ($10)');
    Writeln('2 - Fish         ($7)');
    Writeln('3 - Meat           ($8)');
    Writeln('4 - Salad          ($2)');
    Writeln('5 - Orange Juice ($1)');
    Writeln('6 - Milk            ($1)');
    Writeln('X - nothing');
    Writeln;
    Write('Please enter your selection: ');
    Readln(Selection);

    case Selection of
      '1': begin
        Writeln('You have ordered Chicken, '
          'this will take 15 minutes');
        Price:= 10;
      end;
      '2': begin
```

```

        Writeln('You have ordered Fish, this will take 12 minutes');
        Price:= 7;
    end;
    '3': begin
        Writeln('You have ordered meat, this will take 18 minutes');
        Price:= 8;
    end;
    '4': begin
        Writeln('You have ordered Salad, this will take 5 minutes');
        Price:= 2;
    end;
    '5': begin
        Writeln('You have ordered Orange juice, '
                'this will take 2 minutes');
        Price:= 1;
    end;
    '6': begin
        Writeln('You have ordered Milk, this will take 1 minute');
        Price:= 1;
    end;
else
begin
    Writeln('Wrong entry');
    Price:= 0;
end;
end;

Total:= Total + Price;

until (Selection = 'x') or (Selection = 'X');
Writeln('Total price          = ', Total);
Write('Press enter key to close');
Readln;
end.

```

在上例中，我们使用了这些技巧：

1. 在 **case** 分支结构中添加 **begin—end**，以将多个语句视为一个语句。
2. 将 **Total** 初始化为 0，以累计订单的总价。然后，在每次循环中将所选价格添加到该变量：

```
Total:= Total + Price;
```

3. 我们给予两个选项来完成订单：X 或 x。在计算机中，这两个字符的存储是不同的。

注意：倒数第 5 行

```
until (Selection = 'x') or (Selection = 'X');
```

可由

```
until UpCase(Selection) = 'X';
```

替换。如果用户输入小写字母，则会先改为大写，因此两种输入（'x' 和 'X'）都返回真。

### 1.4.3 While 循环

**while** 循环类似于 **repeat** 循环，但是有两点不同：

1. 在 **while** 循环中，进入循环前会先检查条件，而 **repeat** 循环则是先进入再检查条件。这意味着 **repeat** 循环体至少被执行一次。但如果条件在循环起始时就返回假，**while** 循环会被阻止。
2. 如果在循环中有多条需要语句被执行，**while** 循环需要 **begin—end** 对。**repeat** 循环则不需要，它的循环体以 **repeat** 开始，到 **until** 为止。

**while** 循环的应用例如下所示。

```
var
  Num: Integer;
begin
  Write('Input a number: ');
  Readln(Num);
  while Num > 0 do
  begin
    Write('From inside loop: Input a number: ');
    Readln(Num);
  end;
  Write('Press enter key to close');
  Readln;
end.
```

### 使用 while 循环实现的阶乘程序

```
var
  Fac, Num, i: Integer;
begin
  Write('Please input any number: ');
  Readln(Num);
  Fac:= 1;
```

```
i:= Num;  
while i > 1 do  
begin  
    Fac:= Fac * i;  
    i:= i - 1;  
end;  
Writeln('Factorial of ', Num, ' is ', Fac);  
  
Writeln('Press enter key to close');  
Readln;  
end.
```

**while** 循环没有循环计数器。由于这个原因,我们需要使用变量 **i** 当作循环计数器工作。循环计数器的值通过输入数字来初始化,在每次循环中减少 1,当 **i** 达到 1 时循环停止。



## 1.5 字符串

字符串类型可以容纳一连串的字符，可以用于储存文本、人名、车牌号码等内容。

下例展示用字符串存储人名。

```
var
  Name: string;
begin
  Write('Please enter your name: ');
  Readln(Name);
  Writeln('Hello ', Name);

  Writeln('Press enter key to close');
  Readln;
end.
```

下例使用字符串存储用户信息。

```
var
  Name: string;
  Address: string;
  ID: string;
  DOB: string;
begin
  Write('Please enter your name: ');
  Readln(Name);
  Write('Please enter your address: ');
  Readln(Address);
  Write('Please enter your ID number: ');
  Readln(ID);
  Write('Please enter your date of birth: ');
  Readln(DOB);
  Writeln;
  Writeln('Card:');
  Writeln('-----');
  Writeln('| Name:      ', Name);
  Writeln('| Address:   ', Address);
  Writeln('| ID:        ', ID);
  Writeln('| D. O. B.: ', DOB);
  Writeln('-----');

  Writeln('Press enter key to close');
  Readln;
end.
```

字符串可以进行连接，以便产生更长的字符串。例如，我们可以将 `FirstName`、`MiddleName` 和 `FamilyName` 连接为 `FullName` 字符串，如下例所示。

```
var
```

```

FirstName: string;
MiddleName: string;
FamilyName: string;
FullName: string;
begin
  Write('Please enter your first name: ');
  Readln(FirstName);
  Write('Please enter your middle name: ');
  Readln(MiddleName);
  Write('Please enter your family name: ');
  Readln(FamilyName);
  FullName:= FirstName + ' ' + MiddleName + ' ' + FamilyName;
  Writeln('Your full name is: ', FullName);

  Writeln('Press enter key to close');
  Readln;
end.

```

注意: 在上例中, 我们通过在名称中间添加一个空格字符 (' '), 如 `FirstName + ' ' + MiddleName`), 以分隔名称的。空格也是一个字符 (character)。

我们可以对字符串做很多操作, 如搜索子串, 或复制一个字符串的内容到另一个字符串。这行将 `FullName` 字符串全部转换为大写。

```

FullName:= UpperCase(FullName);

```

这行将它全部小写化:

```

FullName:= LowerCase(FullName);

```

下例使用 **Pos** 函数在一个用户名称中搜索字母 **a**。**Pos** 函数返回在字符串中某个字符 (或子串) 第一次出现的索引 (index)。如果在所给字符串中这个字母或子串不存在, 返回 0。

```

var
  YourName: string;
begin
  Write('Please enter your name: ');
  Readln(YourName);
  If Pos('a', YourName) > 0 then
    Writeln('Your name contains a')
  else
    Writeln('Your name does not contain letter a');

  Writeln('Press enter key to close');
  Readln;
end.

```

**Pos** 函数不会找出名称中可能包含的大写 **A**。因为正如之前提到的, '**A**' 不同于 '**a**'。为解决这个问题, 我们可以将名称的所有字母转换为小写, 再进行搜索。

```
If Pos('a', LowerCase(YourName)) > 0 then
```

在代码的下一个修改版中，程序可以显示名称中字母 a 的位置。

```
var
  YourName: string;
begin
  Write('Please enter your name: ');
  Readln(YourName);
  If Pos('a', LowerCase(YourName)) > 0 then
  begin
    Writeln('Your name contains a');
    Writeln('a position in your name is: ',
      Pos('a', LowerCase(YourName)));
  end
  else
    Writeln('Your name does not contain a letter');

  Write('Press enter key to close');
  Readln;
end.
```

若名称包含多个字母 a，**Pos** 函数只会返回名称中第一次出现的索引。

**Length** 函数返回字符串中的字符总数。

```
Writeln('Your name length is ', Length(YourName), ' letters');
```

字符串可以使用索引（index number）获得其第一个字符、

```
Writeln('Your first letter is ', YourName[1]);
```

第二个字符、

```
Writeln('Your second letter is ', YourName[2]);
```

或最后一个字符。

```
Writeln('Your last letter is ', YourName[Length(YourName)]);
```

使用 **for** 循环，可以逐字符显示字符串变量的内容：

```
for i:= 1 to Length(YourName) do
  Writeln(YourName[i]);
```

### 1.5.1 Copy 函数——字符串的复制

我们可以使用函数 **Copy** 复制一个字符串的部分。例如，如果需要从字符串 `'hello world'` 中提取单词 `'world'`，且这一部分的位置已知，就可以如下例一样完成。

```
var
  Line: string;
  Part: string;
begin
  Line:= 'Hello world';

  Part:= Copy(Line, 7, 5);

  Writeln(Part);

  Writeln('Press enter key to close');
  Readln;
end.
```

**Copy** 函数有如下参数。

- Part 是字符串变量，用于存储截取的结果（子串 `'world'`）。
- Line 是源字符串，含有内容 `'Hello world'`。
- 7 是起始点，也是需要提取的子串的索引；在本例中，它表示是字符 w。
- 5 是提取部分的长度，在本例中为单词 `'world'` 的长度。

在下例中，要求用户输入一个月份名称（如 February）然后重新打出它的 3 字母简写（如 Feb）。<sup>2</sup>

```
var
  Month: string;
  ShortName: string;
begin
  Write('Please input full month name e.g. January: ');
  Readln(Month);

  ShortName:= Copy(Month, 1, 3);

  Writeln(Month, ' is abbreviated as: ', ShortName);

  Writeln('Press enter key to close');
  Readln;
end.
```

---

<sup>2</sup>事实上，英美国家对 6 月、7 月、9 月更倾向于使用 4 字母，即“June”、“July”、“Sept.”。

## 1.5.2 Insert 过程——字符串的插入

Insert 过程将一个子串插入到字符串中。与字符串连接运算符 (+) 仅能前后连接不同，**Insert** 可以在已有字符串当中添加子串。

例如，我们可以把单词 **Pascal** 插入到已有的字符串 **'Hello world'** 的第一个空格后，使结果为 **'Hello Pascal World'**，如下例所示。

```
var
  Line: string;
begin
  Line:= 'Hello world';

  Insert('Pascal ', Line, 7);

  Writeln(Line);

  Writeln('Press enter key to close');
  Readln;
end.
```

**Insert** 过程有如下参数。

- **'Pascal '** 是需要插入的子串。
- **Line** 是目标字符串，将包含运算的结果。
- **7** 是在目标字符串中开始插入的位置。本例中为第 7 个字符，即 **'Hello world'** 的第一个空格后。

## 1.5.3 Delete 过程——字符串的删除

该过程用于从字符串中删除字符或者子串。使用时，需要知道将被删除的子串的开始位置和长度。

例如，如果我们需要从字符串 **'Hello world'** 中删除两个字母 l，生成 **'Heo world'**，可以如下例所示。

```
var
  Line: string;
begin
  Line:= 'Hello world';
  Delete(Line, 3, 2);
  Writeln(Line);
  Writeln('Press enter key to close');
  Readln;
end.
```

## 1.5.4 Trim 函数——去除空格

这个函数用于从字符串首尾移除空格。如果我们有一个含有 ' Hello ' 的字符串，在使用这个函数后将变为 'Hello'。

下例中，为让空格更明显，我们在字符串前后加入额外的字符。

```
program TrimStr;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  Line: string;
begin
  Line:= ' Hello ';

  Writeln('<', Line, '>');

  Line:= Trim(Line);

  Writeln('<', Line, '>');

  Writeln('Press enter key to close');
  Readln;
end.
```

上例中使用了单元 SysUtils，它含有 Trim 函数。

也有另外两个仅移除字符串一侧的空格的函数：TrimRight 和 TrimLeft。

## 1.5.5 StringReplace 函数——查找与替换

StringReplace 函数替换字符、带有其他字符的子串、或字符串中的子串。

```
program StrReplace;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
```

```

Classes, SysUtils
{ you can add units after this };

var
  Line: string;
  Line2: string;
begin
  Line:= 'This is a test for string replacement';
  Line2:= StringReplace(Line, ' ', '-', [rfReplaceAll]);
  Writeln(Line);
  Writeln(Line2);
  Write('Press enter key to close');
  Readln;
end.

```

StringReplace 函数的参数如下。

1. Line 是需要修改的原始的字符串。
2. ' ' 是需要被替换的子串，本例中为空格。
3. '-' 是新子串，用于替换上一个子串。
4. [rfReplaceAll] 是替换类型。本例替换所有空格子串的出现。

我们可以仅使用一个字符串变量，并且丢弃变量 Line2，如下例所示。但是，这样做将失去原始的文本值。

```

var
  Line: string;
begin
  Line:= 'This is a test for string replacement';
  Writeln(Line);
  Line:= StringReplace(Line, ' ', '-', [rfReplaceAll]);
  Writeln(Line);
  Write('Press enter key to close');
  Readln;
end.

```

## 1.6 数组

数组（array）是相同类型的变量的一个链接（chain）。如果需要声明一个 10 个整型变量的数组，可以如下完成。

```
Numbers: array [1 .. 10] of Integer;
```

我们可以使用它的索引在数组中获取单个变量。例如，为数组第 1 个变量赋值，则如下完成。

```
Numbers[1] := 30;
```

若为第二个变量赋值，则把上例中的索引改为 2。

```
Numbers[2] := 315;
```

在下例中，我们将要求用户输入 10 位学生的分数，并且在一个数组中存储，最终得出该学生是否及格。

```
var
  Marks: array [1 .. 10] of Integer;
  i: Integer;
begin
  for i:= 1 to 10 do
  begin
    Write('Input student number ', i, ' mark: ');
    Readln(Marks[i]);
  end;

  for i:= 1 to 10 do
  begin
    Write('Student number ', i, ' mark is: ', Marks[i]);
    if Marks[i] >= 60 then
      Writeln(' Pass')
    else
      Writeln(' Fail');
    end;

    Writeln('Press enter key to close');
    Readln;
  end.
```

我们可以修改先前的代码，用于获取最高分、最低分。

```
var
  Marks: array [1 .. 10] of Integer;
  i: Integer;
  Max, Min: Integer;
```



```

begin
  for i:= 1 to 10 do
  begin
    Write('Input student number ', i, ' mark: ');
    Readln(Marks[i]);
  end;

  Max:= Marks[1];
  Min:= Marks[1];

  for i:= 1 to 10 do
  begin
    // 检查当前分数是否最大
    if Marks[i] > Max then
      Max:= Marks[i];

    // 检查当前分数是否最小
    if Marks[i] < Min then
      Min:= Marks[i];

    Write('Student number ', i, ' mark is: ', Marks[i]);
    if Marks[i] >= 60 then
      Writeln(' Pass')
    else
      Writeln(' Fail');
  end;

  Writeln('Max mark is: ', Max);
  Writeln('Min mark is: ', Min);
  Writeln('Press enter key to close');
  Readln;
end.

```

在本例中，我们先假定第一个同学的分數（Marks[1]）为最大或最小，然后用其他分数比较。

```

Max:= Marks[1];
Min:= Marks[1];

```

然后，在循环内部，我们用每一个分数和 Max、Min 两个变量比较。找到一个大于 Max 的数字，就用它为 Max 赋值；找到一个小于 Min 的数字，就用它为 Min 赋值。

在前面的示例中，首次出现了注释，如下所示。

```

// 检查当前分数是否最大

```

我们在行首使用 //，表明该行是注释。注释不影响代码，也不参与编译，一般用于向其他程序员（或自己）解释代码。

//用于短注释。如果我们需要编写多行注释，我们可以使用 {} 或 (\*\*)，如下例。

```

for i:= 1 to 10 do
begin
    { 检查当前分数是否最大;
      如果大于 Max, 就向 Max 赋值 }
    if Marks[i] > Max then
        Max:= Marks[i];

    (* 检查当前分数是否最小
       若 Mark<Min, 则对 Mark 赋 Min 的值
    *)
    if Marks[i] < Min then
        Min:= Marks[i];

    Write('Student number ', i, ' mark is: ', Marks[i]);
    if Marks[i] >= 60 then
        Writeln(' Pass')
    else
        Writeln(' Fail');
end;

```

我们也可以通过注释暂时禁用部分我们代码。

```

Writeln('Max mark is: ', Max);
// Writeln('Min mark is: ', Min);
Writeln('Press enter key to close');
Readln;

```

在上面的代码中，我们禁用输出最低分的过程。

注意：我们也可以声明一个索引从 0 开始的数组，与 C 语言的数组相似。

```

Marks: array [0 .. 9] of Integer;

```

这个数组也可以容纳 10 个元素，但需要使用索引 0 获取第一项，

```

Numbers[0] := 30;

```

用 1 获取第二项，

```

Numbers[1] := 315;

```

用 9 获取最后一项。

```

Numbers[9] := 10;

```

上一行代码也可以写成如下形式。

```

Numbers[High(Numbers)] := 10;

```

## 1.7 记录

数组可以容纳多个相同类型的变量，记录（record）则可以容纳不同类型的变量，称为“字段”（fields）。

这种字段的组合可以当作单个单元或变量对待。我们可以使用记录用来储存属于相同对象的信息。例如，一辆车的信息可以由如下 3 个变量表示。

1. 车型：字符串
2. 引擎大小：实型
3. 出厂年份：整型

我们可以在一个记录中包括这些不同的类型。例如，在下例中，记录代表一辆车的信息。

```
program Cars;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

type
  TCar = record
    ModelName: string;
    Engine: Single;
    ModelYear: Integer;
  end;

var
  Car: TCar;
begin
  Write('Input car Model Name: ');
  Readln(Car.ModelName);
  Write('Input car Engine size: ');
  Readln(Car.Engine);
  Write('Input car Model year: ');
  Readln(Car.ModelYear);

  Writeln;
  Writeln('Car information: ');
  Writeln('Model Name: ', Car.ModelName);
  Writeln('Engine size: ', Car.Engine);
  Writeln('Model Year: ', Car.ModelYear);

  Write('Press enter key to close. ');
  Readln;
```

```
end.
```

在这个示例中，我们使用“**type**”关键字定义了一个新的记录类型。

```
type
  TCar = record
    ModelName: string;
    Engine: Single;
    ModelYear: Integer;
  end;
```

类型名以字母T开始,表示该关键字是类型(Type)而非变量。变量名应如同Car、Hour、UserName,而类型名称则如同 TCar、THouse 或 TUserName。这是 Pascal 语言的一个标准。

当我们需要使用这个新类型时,应该声明一个该类型的变量,如下所示。

```
var
  Car: TCar;
```

如果我们需要在它的字段中储存一个值,则应如下所示。

```
Car.ModelName
```

记录类型将被使用在下一章的“任意存取文件”部分。

## 第二章

# 进阶应用

## 2.1 文件

文件是操作系统和应用程序的重要元素。操作系统的组成是文件，而信息和数据也能存储为文件，如照片、书、应用程序，或简单的文本文件。

操作系统控制文件管理，包括读、写、编辑和删除等操作。

文件可以通过很多标准进行分类。例如，文件可以分为可执行（executable）文件和数据（data）文件。Lazarus 编译后的二进制应用程序是可执行文件，而 Pascal 源代码（.pas 后缀）是数据文件。PDF 格式的电子书、JPEG 图片也是数据文件。

依照它们内容的表现形式，又可以如下分类。

1. 纯文本文件（text files）。简单的文本文件，可以使用简单的工具读写。这些工具包括操作系统的命令行，如 Linux 中的 `cat`、`vi` 命令，或 Windows 命令行中的 `type`、`copy` 命令。
2. 二进制文件（binary data files）。这些文件更复杂，也需要特定应用程序来打开。例如，图片文件不能使用简单的命令行工具打开，而应该用 GIMP、Kolour Paint、Windows 的画图等应用程序打开。如果我们使用简单的命令行工具打开图片或音频文件，获取的内容将既无法识别又没有意义。数据库文件就是一种二进制文件，应该使用适当的应用程序打开。

依照存取（access）类型，则可以用另一种标准分类。

1. 顺序存取（sequential access）的文件。这种文件的例子是文本文件。它没有固定大小的记录。因为每一行的长度不确定，所以“第三行的开始”这类位置无法预先知道。因此，我们仅能以只读模式打开文件，编写新的文件，或在文件结尾添加内容。如果需要在文件中间增减内容，只能全部读取到内存中，进行修改，删除整个文件，然后用修改的文本覆盖它。
2. 任意存取（random access）的文件。这种类型的文件有固定大小的记录。记录指可以一次内读写的最小单元，可以是字节（Byte）、整型（Integer）、字符串（**string**），也可以是用户自定义的记录。这类文件可以同时进行读、写。例如，可以读取第 3 笔记录，并复制到第 10 笔。这是因为我们可以准确地知道文件中每笔记录的位置。修改记录是简单的。在随机存取文件中，我们可以修改一些记录，而不影响剩余部分。

### 2.1.1 文本文件

文本文件是最简单的文件。但是，我们应该仅仅以从前到后的方向写入它们——当读写文本文件时，我们不能退回。我们也应该在打开文件前选择操作模式：读、写或追加（在文件结尾后写入）。

在这个示例中，我们将显示一个通过用户选择的文本文件的内容。例如，文件可能在 Windows 中位于 `c:\test\first.pas`，或在 Linux 中位于 `/home/user/first.pas`。

#### 读取文本文件示例程序

```
program ReadFile;

{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, sysUtils
  { you can add units after this };

var
  FileName: string;
  F: TextFile;
  Line: string;
begin
  Write('Input a text file name: ');
  Readln(FileName);
  if FileExists(FileName) then
  begin
    // 使物理文件 (Filename) 与文件变量 (F) 连接
    AssignFile(F, FileName);

    Reset(F); // 文件存在时，设定为读取模式

    // 当文件未读完时
    while not Eof(F) do
    begin
      Readln(F, Line); // 从文件读取一行
      Writeln(Line);   // 在屏幕上显示这一行
    end;
    CloseFile(F); // 释放文件 (Filename) 与变量 (F) 的连接
  end
  else // 文件不存在
  begin
    Writeln('File does not exist');
    Write('Press enter key to close. ');
    Readln;
  end;
end.
```

在 Linux 中，我们可以这样输入名称：

```
/etc/resolv.conf
```

或者

```
/proc/meminfo  
/proc/cpuinfo
```

我们需要使用新的类型、函数和过程（procedures）来处理文本文件，如下所示。

1. **F: TextFile;**

**TextFile** 是一个用于声明一个文本文件变量的类型。这个变量可以链接到一个后来使用的文本文件。

2. **if FileExists(FileName) then**

**FileExists** 是一个包含在 **SysUtils** 单元中的函数。它检查一个文件的存在。如果文件在存储介质中存在，则返回 **True**。

3. **AssignFile(F, FileName);**

在已确定文件的存在以后，我们可以使用 **AssignFile** 过程，使物理文件与文件变量 **F** 连接。此后，对 **F** 变量的继续使用便代表该物理文件。

4. **Reset(F);** // 文件存在时，设定为读取模式

**Reset** 过程以只读方式打开文本文件，并将读取点（pointer）放置在文件头。如果当前用户对这个文件没有读取权限，一个错误信息将出现，例如“access denied”（拒绝访问）。

5. **ReadLn(F, Line);** // 从文件读取一行

**ReadLn** 过程被用于从文件中读取完整的一行，并放置它在变量 **Line** 中。在从文本文件中读取一行前，我们应该确保该文件没有到达结尾。这可以通过下一个函数 **Eof** 完成。

6. **while not Eof(F) do**

若文件到达结尾，**Eof** 函数就返回 **True**。它表示我们已经读取了该文件的全部内容，并不能再进行读取操作。

7. **CloseFile(F);** // 释放文件（Filename）与变量（F）的连接



在完成文件读写后，我们应该进行关闭，以释放文件。这是因为当文件打开时，**Reset**过程在操作系统中保留文件，阻止其他应用程序的读写。

**CloseFile** 仅能在文件成功打开时被使用。若打开失败（例如文件不存在，或正被其它的应用程序使用），则不能关闭该文件。

在下一个示例中，我们将创建一个新的文本文件并且在其中给予一些文本。

## 创建和写入到文本文件

```
var
  FileName: string;
  F: TextFile;
  Line: string;
  ReadyToCreate: Boolean;
  Ans: Char;
  i: Integer;
begin
  Write('Input a new file name: ');
  Readln(FileName);

  // 检查文件是否存在，若存在则警告
  if FileExists(FileName) then
    begin
      Write('File already exist, did you want to overwrite it? (y/n)');
      Readln(Ans);
      if upcase(Ans) = 'Y' then
        ReadyToCreate:= True
      else
        ReadyToCreate:= False;
    end
  else // 文件不存在
    ReadyToCreate:= True;

  if ReadyToCreate then
    begin
      // 使物理文件 (Filename) 与文件变量 (F) 连接
      AssignFile(F, FileName);

      Rewrite(F); // 创建或覆盖文件，以供写入

      Writeln('Please input file contents line by line, '
        , 'when you finish write % then press enter');
      i:= 1;
      repeat
        Write('Line # ', i, ':');
        Inc(i);
        Readln(Line);
        if Line <> '%' then
```

```

        Writeln(F, Line); // 向文件内写入一行
    until Line = '%';

    CloseFile(F); // 释放文件 (Filename) 与变量 (F) 的连接, 刷新缓冲区
end
else // 文件已存在, 且用户不想覆盖
    Writeln('Doing nothing');
Write('Press enter key to close. ');
Readln;
end.

```

本例中有如下要点。

#### 1. 布尔 (boolean) 类型。

```
ReadyToCreate: Boolean;
```

这种类型可以储存两个值中的一个: **True** (真) 或 **False** (假)。这些值可以在 **if** 条件、**repeat** 循环或 **while** 循环中直接使用。

在上例中, 我们像这样使用 if 条件。

```
if Marks[i] > Max then
```

它返回的也是 **True** 或 **False**。

#### 2. UpCase 函数。

```
if upcase(Ans) = 'Y' then
```

当文件存在时, 会执行这条语句。该程序将警告用户是否覆盖一个已存在的文件。如果用户希望继续, 则输入一个小写字母 **y** 或者大写字母 **Y**。若输入的是小写字母, **UpCase** 函数将其转换为大写字母。

#### 3. Rewrite 过程。

```
Rewrite(F); // 创建或覆盖文件, 以供写入
```

**Rewrite** 过程被用于创建一个新的空文件。如果该文件已经存在, 它将被擦除并且被覆盖。在文本文件情况下, 它也将文件以写入模式打开。

#### 4. Writeln(F, ...) 过程。

```
Writeln(F, Line); // 向文件内写入一行
```

这个过程用于在文本文件中写入字符串或变量, 并且在之后加入行尾字符。行尾字符是回车—换行 (CR/LF) 的组合, 即分别是编号为 13 和 10 的字符。

这些字符在控制台窗口中不能被显示, 但是它会使屏幕上的光标换到新的一行。

#### 5. Inc 过程。

```
Inc(i);
```

这个过程给一个整型变量加 1，等价于下面的语句：

```
i := i + 1;
```

## 6. CloseFile 过程。

```
CloseFile(F); // 释放文件 (Filename) 与变量 (F) 的连接，刷新缓冲区
```

正如我们先前提到的，CloseFile 过程在操作系统中释放一个文件。另外，当写入到一个文本文件时，它也会刷新写入缓冲区。

文本文件的缓冲区是使得处理文本文件更快的特征。程序并不直接将一行或字符直接写入到磁盘或任何其他介质（与写入内存相比，这是非常慢的），而是写入内存缓冲区。当缓冲区的大小充满时，相应内容将被刷新（即强制写入）到硬盘等永久性存储介质中。这个处理使写入更快，但在突然断电等特殊情况下，则将增加丢失（缓冲区）数据的风险。为尽量避免数据丢失，应该在完成写入后立即关闭文件，或者调用 **Flush** 过程以显式刷新缓冲区。

## 附加到文本文件

在这个示例中，程序打开已经存在的文件，并使用 **Append** 过程，在不移除原始内容的前提下向文件尾写入。

```
var
  FileName: string;
  F: TextFile;
  Line: string;
  i: Integer;
begin
  Write('Input an existed file name: ');
  Readln(FileName);
  if FileExists(FileName) then
  begin
    // 使物理文件 (Filename) 与文件变量 (F) 连接
    AssignFile(F, FileName);

    Append(F); // 以附加模式写入文件

    Writeln('Please input file contents line by line',
      'when you finish write % then press enter');
    i:= 1;
    repeat
      Write('Line # ', i, ' append :');
      Inc(i);
      Readln(Line);
      if Line <> '%' then
        Writeln(F, Line); // 向文件内写入一行
    until Line = '%';
    CloseFile(F); // 释放文件 (Filename) 与变量 (F) 的连接，刷新缓冲区
  end
  else
    Writeln('File does not exist');
  Write('Press enter key to close. ');
  Readln;
end.
```

在我们运行这个应用程序，并键入已存在的文件名后，我们可以使用命令或双击来查看附加数据。

## 2.1.2 任意存取文件

正如先前所说，以访问类型分类时，另一类文件为任意存取文件，又称直接存取文件。这类文件有固定大小的记录，所以我们能任意跳转到任意一笔记录进行读写。

随机存取文件有两种类别：有类型文件，无类型文件。

## 2.1.3 有类型文件

有类型文件的每笔记录大小、类型都相同。例如，如果一个文件包含单字节（Byte）类型的记录，则每笔记录大小都是 1 字节；如果文件包含单精度浮点数（Single）类型，则每笔记录大小都是 4 字节，以此类推。

下例演示如何以单字节文件存储数据。

### 写入成绩的程序

```
var
  F: file of Byte;
  Mark: Byte;
begin
  AssignFile(F, 'marks.dat');
  Rewrite(F); // 创建文件
  Writeln('Please input students marks, write 0 to exit');

  repeat
    Write('Input a mark: ');
    Readln(Mark);
    if Mark <> 0 then // 不写0值，因为表示结束
      Write(F, Mark);
  until Mark = 0;
  CloseFile(F);

  Write('Press enter key to close. ');
  Readln;
end.
```

在这个示例中，我们使用这个语句来定义一个有类型文件：

```
F: file of Byte;
```

这意味着：该文件包含单字节类型的记录，每笔可以容纳从 0 到 255 的值。

我们使用 **Rewrite** 过程创建文件，并且打开它用于写入：

```
Rewrite(F); // 创建文件
```

我们也使用 **Write** 而非 **Writeln** 在有类型文件中写入记录：

```
Write(F, Mark);
```

**Writeln** 不适合用于有类型文件，因为它会在写入文本的每一行末添加换行符；但是，**Write** 只写入数据本身，并不附加内容。在目前情况下，如果输入 10 笔记录，文件的大小也将是 10 字节。

下例演示如何读取已保存的文件的内容。

## 读取成绩的程序

```
program ReadMarks;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  F: file of Byte;
  Mark: Byte;
begin
  AssignFile(F, 'marks.dat');
  if FileExists('marks.dat') then
  begin
    Reset(F); // 打开文件
    while not Eof(F) do
    begin
      Read(F, Mark);
      Writeln('Mark: ', Mark);
    end;
    CloseFile(F);
  end
  else
    Writeln('File (marks.dat) not found');

  Write('Press enter key to close. ');
  Readln;
end.
```

下例演示在保留已有数据的同时添加新的记录。

## 添加成绩的程序

```
program AppendMarks;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  F: file of Byte;
  Mark: Byte;
begin
  AssignFile(F, 'marks.dat');
  if FileExists('marks.dat') then
  begin
    FileMode:= 2; // 指定为读写方式
    Reset(F); // 打开文件
    Seek(F, FileSize(F)); // 跳转到文件末尾
    Writeln('Please input students marks, write 0 to exit');

    repeat
      Write('Input a mark: ');
      Readln(Mark);
      if Mark <> 0 then // 不写0值，因为表示结束
        Write(F, Mark);
    until Mark = 0;
    CloseFile(F);
  end
  else
    Writeln('File marks.dat not found');

  Write('Press enter key to close. ');
  Readln;
end.
```

在运行这个程序，并且输入新的记录后，我们可以再次运行它来查看添加的数据。

注意，我们使用 **Reset** 代替 **Rewrite** 打开文件用于写入。**Rewrite** 过程删除同名的已存在文件，再创建空文件；**Reset** 只打开已存在文件，而不清除的内容。

我们也向 **Filemode** 赋值 2，表示以读写方式打开文件。该变量的值对应打开方式，0 表示只读，1 表示只写，2 表示读写。

```
FileMode:= 2; // 读写方式
Reset(F); // 打开文件
```

**Reset** 将读写点指向文件开始，因此直接进行写入将覆盖原来的记录。所以，我们使用 **Seek** 过程把读写点移动到文件尾。**Seek** 仅能用于随机存储文件。

如果试图用 **Seek** 访问文件中不存在的位置（如指定访问第 100 笔记录，而文件中只有 50 笔记录），将产生错误。

**FileSize** 函数返回文件中记录数。它与 **Seek** 联合使用，以跳到文件结尾：

```
Seek(F, FileSize(F)); // 跳转到文件末尾
```

注意，如果记录文件已经存在，那么可以使用本例；否则应使用本章第一个程序“写入学生数据”，因为 **Rewrite** 可以创建文件。在下例中，我们检查文件是否存在，然后选择两个方法中的一个。

## 创建或添加成绩的程序

```
program ReadWriteMarks;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  F: file of Byte;
  Mark: Byte;
begin
  AssignFile(F, 'marks.dat');
  if FileExists('marks.dat') then
  begin
    FileMode:= 2; // 指定为读写方式
    Reset(F); // 打开文件
    Writeln('File already exist, opened for append');
    // Display file records
    while not Eof(F) do
    begin
      Read(F, Mark);
      Writeln('Mark: ', Mark);
    end
  end
  else // 文件不存在，新建文件
  begin
    Rewrite(F);
    Writeln('File does not exist, now it is created');
  end;
end;
```



```

Writeln('Please input students marks, write 0 to exit');
Writeln('File pointer position at record # ', FilePos(f));
repeat
  Write('Input a mark: ');
  Readln(Mark);
  if Mark <> 0 then // 不写0值，因为表示结束
    Write(F, Mark);
until Mark = 0;
CloseFile(F);

Write('Press enter key to close. ');
Readln;
end.

```

注意：本例不使用 **Seek**，而是读取全部文件内容。这项操作也能把读写点移到文件的结尾。

在下例中，使用记录类型的文件来存储汽车信息。

## 汽车数据库程序

```

program CarRecords;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

type
  TCar = record
    ModelName: string[20];
    Engine: Single;
    ModelYear: Integer;
  end;

var
  F: file of TCar;
  Car: TCar;
begin
  AssignFile(F, 'cars.dat');
  if FileExists('cars.dat') then
  begin
    FileMode:= 2; // 指定为读写方式
    Reset(F); // 打开文件
    Writeln('File already exist, opened for append');
    // 显示文件内容
    while not Eof(F) do

```

```

begin
    Read(F, Car);
    Writeln;
    Writeln('Car # ', FilePos(F), ' -----');
    Writeln('Model : ', Car.ModelName);
    Writeln('Year : ', Car.ModelYear);
    Writeln('Engine: ', Car.Engine);
end
end
else // 文件不存在，新建文件
begin
    Rewrite(F);
    Writeln('File does not exist, created');
end;

Writeln('Please input car informaion, ',
        'write x in model name to exit');
Writeln('File pointer position at record # ', FilePos(f));

repeat
    Writeln('-----');
    Write('Input car Model Name : ');
    Readln(car.ModelName);
    if Car.ModelName <> 'x' then
    begin
        Write('Input car Model Year : ');
        Readln(car.ModelYear);
        Write('Input car Engine size: ');
        Readln(car.Engine);
        Write(F, Car);
    end;
until Car.ModelName = 'x';
CloseFile(F);

Write('Press enter key to close. ');
Readln;
end.

```

在上例中，我们声明 TCar 类型来定义汽车信息。第一个字段（ModelName）是一个字符串变量，但是我们限制它的最大长度为 20：

```
ModelName: string[20];
```

我们应使用这种方式定义字符串以用于文件。内存中默认的 ANSI 字符串以不同方法储存，对长度没有限制；但在有类型文件中，数据所占大小需要预先定义。

## 2.1.4 无类型文件

无类型文件的数据没有固定的类型，而是逐字节存储。

在下例中，我们将一个文件逐字节显示，正如存储格式一样。

### 逐字节显示文件内容

```
program ReadContents;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  FileName: string;
  F: file;
  Block: array [0 .. 1023] of Byte;
  i, NumRead: Integer;
begin
  Write('Input source file name: ');
  Readln(FileName);

  if FileExists(FileName) then
    begin
      AssignFile(F, FileName);

      FileMode:= 0; // 设置为只读方式
      Reset(F, 1);

      while not Eof(F) do
        begin
          BlockRead(F, Block, SizeOf(Block), NumRead);
          // 在屏幕上显示字符
          for i:= 0 to NumRead - 1 do
            Writeln(Block[i], ': ', Chr(Block[i]));
          end;
          CloseFile(F);
        end
      else // 文件不存在
        Writeln('Source File does not exist');

      Write('press enter key to close. ');
      Readln;
```

```
end.
```

在上例代码中，有以下新的用法。

1. 声明文件的类型：

```
F: file;
```

2. 读写缓冲器：

```
Block: array [0 .. 1023] of Byte;
```

我们使用 1024 项的字节数组（大小 1 KiB）来进行分段读取及复制。

3. 当打开一个无类型文件时，需要一个附加参数：

```
Reset(F, 1);
```

附加参数表示同一时间内读取的最小单元，以字节表示。因为我们需要读取任意文件，这个参数应该取为 1 字节。

在下一节写入文件时，也有类似的附加参数

4. **BlockRead** 过程：

```
BlockRead(F, Block, SizeOf(Block), NumRead);
```

**BlockRead** 用于无类型文件，每次读取一个分块（block）的数据。

在本例中，过程参数是：

- **SourceF** 指向需要复制的源代码变量。
- **Block** 数组存储正在读写的内容。
- **SizeOf(Block)** 项表示单次需要的记录数。例如，输入 100 表示需要读 100 笔记录（此情况下为字节）。使用 **SizeOf** 函数说明需要读的记录数等于存储空间大小。
- **NumRead** 存储实际读出的记录数。**BlockRead** 函数有时能成功读取相应数量的记录，有时则只能读取部分。在文件的最后部分，读取的记录数量常少于需要读取的数量。例如，若文件长 1034 字节，则第一次循环中可以读满 1024 字节，但第二次则只能读出最后 10 字节，且 **Eof** 函数将返回真。因此，**NumRead** 值在使用 **BlockWrite** 过程时是必要的。

在使用这个文件显示文件时，我们将首次见到行末的换行符（13 号字符与 10 号字符），因为我们显示每个字节的 ASCII 码。在 Linux 中，我们将仅找到 LF 字符。这两种情形都是文本文件中的行分隔符。

我们可以其他类型的文件，如图片或可执行文件，以观察它们如何存储。

在这个示例中，我们使用 **Chr** 函数来获取字符的数字值。比如，字母 a 占一个字节，被存储为 97，而相应的大写字母 A 被存储为 65。

## 2.1.5 文件的复制

所有的文件类型，不论文本文件或二进制文件，都基于字节存储；字节是计算机内存和磁盘中最小的数据单位。每个文件应该包含整数个字节（包括空文件——大小为 0 字节）。每个字节能容纳一个范围在 0 至 255 的整数，或一个字符。我们可以使用字节文件或字符文件来复制文件。复制结果是一个新文件，与源代码文件的内容完全相同。

### 以字节文件实现复制

```
program FilesCopy;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  SourceName, DestName: string;
  SourceF, DestF: file of Byte;
  Block: Byte;
begin
  Writeln('Files copy');
  Write('Input source file name: ');
  Readln(SourceName);

  Write('Input destination file name: ');
  Readln(DestName);
  if FileExists(SourceName) then
  begin
    AssignFile(SourceF, SourceName);
    AssignFile(DestF, DestName);

    FileMode:= 0; // 设置为只读方式
    Reset(SourceF); // 打开文件
    Rewrite(DestF); // 创建目标文件

    // 开始复制
    Writeln('Copying..');
    while not Eof(SourceF) do
    begin
      Read(SourceF, Block); // 读取1个字节
      Write(DestF, Block); // 写入新文件
    end;
    CloseFile(SourceF);
    CloseFile(DestF);
```

```

end
else // 文件不存在
    Writeln('Source File does not exist');

    Write('Copy file is finished, press enter key to close. ');
    Readln;
end.

```

在运行先前的示例后，我们应该输入两个地址：已存在的源文件地址、目标地址。在 Linux 中，我们可以输入这样的文件名称。

```

Input source file name: /home/motaz/quran/mishari/32.mp3
Input destination file name: /home/motaz/Alsajda.mp3

```

在 Windows 中，我们可以这样输入。

```

Input source file name: c:\photos\mypphoto.jpg
Input destination file name: c:\temp\copy.jpg

```

如果地址与程序目录相同，可以仅输入文件名，如下。

```

Input source file name: test.pas
Input destination file name: testcopy.pas

```

如果我们使用这个方法复制大文件，和操作系统的过程相比，它将花费很长时间。这表明操作系统使用其他方式复制文件。如果我们想复制 1 MiB 大小的文件，**while** 循环将进行约 100 万次，即进行读、写各 100 万次。如果我们使用“字”（word，2 字节单元）文件而非字节文件，则只用进行约 50 万次循环，但是仅对于偶数字节大小的文件适用：比如，这样的程序对于 1420 字节的文件将复制成功，但是对于 1423 字节的文件则将失败。

如果要想复制文件更快，则应使用无类型文件。

## 以无类型文件实现复制

```

program FilesCopy2;

{$mode objfpc}{$H+}

uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
    cthreads,
    {$ENDIF}{$ENDIF}
    Classes, SysUtils
    { you can add units after this };

var
    SourceName, DestName: string;
    SourceF, DestF: file;

```

```

Block: array [0 .. 1023] of Byte;
NumRead: Integer;
begin
  Writeln('Files copy');
  Write('Input source file name: ');
  Readln(SourceName);

  Write('Input destination file name: ');
  Readln(DestName);

  if FileExists(SourceName) then
  begin
    AssignFile(SourceF, SourceName);
    AssignFile(DestF, DestName);

    FileMode:= 0; // 设置为只读方式
    Reset(SourceF, 1); // 打开文件
    Rewrite(DestF, 1); // 创建目标文件

    // 开始复制
    Writeln('Copying..');
    while not Eof(SourceF) do
    begin
      // 读取多个字节
      BlockRead(SourceF, Block, SizeOf(Block), NumRead);
      // 写入新文件
      BlockWrite(DestF, Block, NumRead);
    end;
    CloseFile(SourceF);
    CloseFile(DestF);

  end
  else // 文件不存在
    Writeln('Source File does not exist');

  Write('Copy file is finished, press enter key to close. ');
  Readln;
end.

```

在上例代码中，我们使用 **BlockWrite** 函数写入无类型文件：

```
BlockWrite(DestF, Block, NumRead);
```

这是一个写入过程，并且与 **BlockRead** 相似，但也有不同：一、使用 **NumRead** 代替 **SizeOf** 函数，因为 **NumRead** 包含实际的读取的语句块大小；二、第四个参数 **NumWritten**（本例中未使用）并不重要，因为我们在磁盘未满时总能按照需求获取相应记录。

在我们运行这个应用程序后，注意读写大型文件的速度更快。如果文件大小约 1 MiB，则仅需约 1000 次循环来复制输入文件。



## 2.2 日期与时间

日期与时间在编程中也是十分重要的内容。对于交易、购物、支付等操作，存储日期和时间是重要的。这是因为在操作后，程序可以按时间找出某一类事件，例如所有“上月”或“本月”发生的交易。

应用程序日志是依赖于记录日期与时间的操作之一。我们需要知道应用程序启动、停止、出错或崩溃的时间。

**TDateTime** 是 Object Pascal 中的一种可以用于存储特定时刻的类型。它是双精度浮点数，在内存中占用 8 字节。它的小数部分代表时间，而整数部分代表自 1899 年 12 月 30 日以来已经过去的天数。

在下一个示例中，我们将显示如何使用 **Now** 函数显示当前的日期与时间。

```
program DateTime;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes , SysUtils
  { you can add units after this };

begin
  Writeln('Current date and time: ', DateTimeToStr(Now));
  Write('Press enter key to close');
  Readln;
end.
```

我们使用 SysUtils 单元中的 **DateTimeToStr** 函数，将 **Now** 函数的 TDateTime 类型返回值转换为可读的字符串表示。

如果不进行转换，则将获取作为浮点数显示的时刻。

```
Writeln('Current date and time: ', Now);
```

另有两个转换函数，它们仅显示日期部分或仅显示时间部分。

```
Writeln('Current date is ', DateToStr(Now));
Writeln('Current time is ', TimeToStr(Now));
```

此外，**Date** 函数仅返回今天的日期，**Time** 函数仅返回当前时间。

```
Writeln('Current date is ', DateToStr(Date));
Writeln('Current time is ', TimeToStr(Time));
```

这两个函数在不使用的地方置 0: **Date** 函数返回当前日期, 并且在时间部分置 0 (相当于午夜 0 点); **Time** 函数返回当前系统时间, 并且在日期部分置 0 (相当于 1899 年 12 月 30 日)。我们可以使用 **DateTimeToStr** 函数验证以上事实。

```
Writeln('Current date is ', DateTimeToStr(Date));  
Writeln('Current time is ', DateTimeToStr(Time));
```

**DateTimeToStr** 依照计算机配置显示日期与时间, 其结果可能在两个计算机系统中不同。**FormatDateTime** 函数则不管计算机的配置, 而照程序员编写的格式显示日期和时间。

```
Writeln('Current date is ',  
    FormatDateTime('yyyy-mm-dd hh:nn:ss', Date));  
Writeln('Current time is ',  
    FormatDateTime('yyyy-mm-dd hh:nn:ss', Time));
```

在下一个示例中, 我们将像浮点数一样对待日期-时间项, 并且做加减。

```
begin  
    Writeln('Current date and time is ',  
        FormatDateTime('yyyy-mm-dd hh:nn:ss', Now));  
    Writeln('Yesterday time is ',  
        FormatDateTime('yyyy-mm-dd hh:nn:ss', Now - 1));  
    Writeln('Tomorrow time is ',  
        FormatDateTime('yyyy-mm-dd hh:nn:ss', Now + 1));  
    Writeln('Today + 12 hours is ',  
        FormatDateTime('yyyy-mm-dd hh:nn:ss', Now + 1/2));  
    Writeln('Today + 6 hours is ',  
        FormatDateTime('yyyy-mm-dd hh:nn:ss', Now + 1/4));  
    Write('Press enter key to close');  
    Readln;  
end.
```

如果我们从一个日期中加 1 或减 1, 它将加或减一整天。如果我们加上二分之一 (或 0.5), 相当于加半天 (或 12 小时)。

在下例中, 我们将使用一个指定的年, 月和日, 来生产一个日期值。

```
var  
    ADate: TDateTime;  
  
begin  
    ADate:= EncodeDate(1975, 11, 16);  
    Writeln('My date of birth is: ', FormatDateTime('yyyy-mm-dd', ADate));  
    Write('Press enter key to close');  
    Readln;  
end.
```

**EncodeDate** 函数接受年、月、日, 返回含有相应日期的 TDateTime 类变量。

**EncodeTime** 函数接受时、分、秒、毫秒, 返回含有相应时间的 TDateTime 变量。

```

var
  ATime: TDateTime;
begin
  ATime:= EncodeTime(19, 30, 0, 0);
  Writeln('Time of weather report is: ',
    FormatDateTime('hh:nn:ss', ATime));
  Write('Press enter key to close');
  Readln;
end.

```

### 2.2.1 日期与时间的比较

可以像比较浮点数一样比较两个时刻变量。例如，在浮点数中 9.3 大于 5.1，则对于 TDateTime 也相同：**Now + 1**（明天）大于 **Now**（今天）；**Now + 1/24**（1 小时后）大于 **Now - 2/24**（2 小时前）。

在下一个示例中，我们将 2012 年 1 月 1 日放置在一个变量中，与当前日期比较，并检查是否这一日期是否已过去。

```

var
  Year2012: TDateTime;
begin
  Year2012:= EncodeDate(2012, 1, 1);

  if Now < Year2012 then
    Writeln('Year 2012 is not coming yet')
  else
    Writeln('Year 2012 is already passed');
  Write('Press enter key to close');
  Readln;
end.

```

我们可以添加新的函数到这个示例中，用于显示从这天起剩余的天数或过去的天数：

```

var
  Year2012: TDateTime;
  Diff: Double;
begin
  Year2012:= EncodeDate(2012, 1, 1);
  Diff:= Abs(Now - Year2012);

  if Now < Year2012 then
    Writeln('Year 2012 is not coming yet, there are ',
      Format('%0.2f', [Diff]), ' days Remaining ')
  else
    Writeln('First day of January 2012 is passed by ',
      Format('%0.2f', [Diff]), ' Days');

```

```
Write('Press enter key to close');
Readln;
end.
```

Diff 是浮点数变量, 容纳当前日期和 2012 年元旦的差。**Abs** 函数返回一个数的绝对值。

## 新闻记录程序

在这个示例中, 我们将使用文本文件来存储新闻标题, 同时也存储日期和时间。

在再次关闭和打开应用程序后。它将显示先前输入的新闻和它的日期与时间:

```
program news;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  Title: string;
  F: TextFile;
begin
  AssignFile(F, 'news.txt');
  if FileExists('news.txt') then

    begin          // 显示已有新闻
      Reset(F);
      while not Eof(F) do
        begin
          Readln(F, Title);
          Writeln(Title);
        end;
      CloseFile(F); // 读取已经完毕
      Append(F);    // 重新以附加模式打开
    end
  else
    Rewrite(F);

  Write('Input current hour news title: ');
  Readln(Title);
  Writeln(F, DateTimeToStr(Now), ', ', Title);
  CloseFile(F);

  Write('Press enter to close');
  Readln;
end.
```

## 2.3 常量

常量（const）与变量一样有自己的名称且容纳特定值；但是与变量不同，常量的值应在编译时确定，且在程序运行中不能修改。在此之前，我们已经使用了常量——它们是整型或字符串的字面值（literal value），没有命名。如下例：

```
Writeln(5);  
Writeln('Hello');
```

5 是整型常量，在运行时不再更改。'Hello' 是字符串常量。在应用程序的 **uses** 语句后，我们可以使用 **const** 关键字定义常量，如下例所示。

### 燃油消费程序

```
program FuelConsumption;  
  
{$mode objfpc}{$H+}  
  
uses  
    {$IFDEF UNIX}{$IFDEF UseCThreads}  
    cthreads,  
    {$ENDIF}{$ENDIF}  
    Classes, SysUtils  
    { you can add units after this };  
  
const Price = 7.5;  
  
var  
    Payment: Integer;  
    Consumption: Integer;  
    Kilos: Single;  
begin  
    Write('How much did you pay for your car's fuel: ');  
    Readln(Payment);  
    Write('What is the consumption of your car? (Liter per 100 km): ');  
    Readln(Consumption);  
  
    Kilos:= (Payment / Price) / Consumption * 100;  
  
    Writeln('This fuel will keep your car running for : ',  
        Format('%0.1f', [Kilos]), ' Kilometers');  
    Write('Press enter');  
    Readln;  
end.
```

在上例中，依照以下事实，我们将计算汽车用它当前的燃油可以运行的公里数：

1. 油耗: **Consumption** 变量存储当前汽车的油耗, 以每 100 公里的升数表示。<sup>1</sup>
2. 油费: **Payment** 变量存储我们为现在的汽油支付了多少钱。
3. 油价: **Price** 常量存储当前每升<sup>2</sup>汽油的价格。这个值不能被用户输入, 而应该被程序员定义。

在应用程序中, 当多次使用相同的值时, 应使用常量。因为如果有更改的需要, 只需在代码头部更改一次。

---

<sup>1</sup>原文单位为“千克/加仑”, 此处按照中国习惯更改。

<sup>2</sup>原文为每加仑

## 2.4 序列类型

序列 (ordinal) 类型是整数值，它使用文字指示代替数字。例如，如果我们定义语言变量 (阿拉伯语、英语、法语)，我们可以使用 1 代表汉语，2 代表英语，3 代表法语。但是，其他程序员在不看注释的情况下，并不知道 1、2、3 代表的值。如果我们使用序列类型，它将更可读，如下例所示。

```
program OrdinalTypes;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

type
  TLanguageType = (ltChinese, ltEnglish);

var
  Lang: TLanguageType;
  AName: string;
  Selection: Byte;
begin
  Write('Please select Language: 1 (Chinese), 2 (English)');
  Readln(Selection);

  if Selection = 1 then
    Lang:= ltChinese
  else
    if selection = 2 then
      Lang:= ltEnglish
    else
      Writeln('Wrong entry');

  if Lang = ltChinese then
    Write('请输入您的姓名: ')
  else
    if Lang = ltEnglish then
      Write('What is your name: ');

  Readln(AName);

  if Lang = ltChinese then
  begin
    Writeln('您好, ', AName);
    Write('请按回车键关闭程序');
```

```

end
else
if Lang = ltEnglish then
begin
  Writeln('Hello ', AName);
  Write('Please press enter key to close');
end;
Readln;
end.

```

整型、字符、布尔型都是有序类型，而浮点型、字符串不是。

## 2.5 集合

集合（set）类型可以在一个变量中容纳多个属性或特性。集合仅可用于有序类型。例如，如果我们需要定义操作系统对应用程序的支持，我们可以这样完成：

1. 定义序数类型 TApplicationEnv，它代表操作系统：

```
TApplicationEnv = (aeLinux, aeMac, aeWindows);
```

2. 应用程序定义为 TApplicationEnv 的集合，例如：

```
FireFox: set of TApplicationEnv;
```

3. 将操作系统值放在应用程序变量中：

```
FireFox:= [aeLinux, aeWindows];
```

```

program Sets;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

type
  TApplicationEnv = (aeLinux, aeMac, aeWindows);

var

```



```

Firefox: set of TApplicationEnv;
SuperTux: set of TApplicationEnv;
Delphi: set of TApplicationEnv;
Lazarus: set of TApplicationEnv;
begin
  Firefox:= [aeLinux, aeWindows];
  SuperTux:= [aeLinux];
  Delphi:= [aeWindows];
  Lazarus:= [aeLinux, aeMac, aeWindows];

  if aeLinux in Lazarus then
    Writeln('There is a version for Lazarus under Linux')
  else
    Writeln('There is no version of Lazarus under linux');

  if aeLinux in SuperTux then
    Writeln('There is a version for SuperTux under Linux')
  else
    Writeln('There is no version of SuperTux under linux');

  if aeMac in SuperTux then
    Writeln('There is a version for SuperTux under Mac')
  else
    Writeln('There is no version of SuperTux under Mac');

  Readln;
end.

```

我们也可以对其他序数类型使用集合语法，如整型数：

```

if Month in [1, 3, 5, 7, 8, 10, 12] then
  Writeln('This month contains 31 days');

```

或字符：

```

if Char in ['a', 'A'] then
  Writeln('This letter is A ');

```

## 2.6 异常处理

编程中有两类错误。第一类称为编译错误，如使用未定义的变量，或者编写带有错误语法的语句。这类错误会阻止程序编译，编译器显示错误内容，并指向包含错误的行。

另一类是运行错误（run-time error）。这类错误在运行程序时才发生，如在某行中将整数除以 0。

```
x:= y / z;
```

这一行在语法上正确，但运行时变量 *z* 可能等于 0。这时，程序会崩溃，并显示错误信息“division by zero”（除以 0）。

试图打开不存在的文件会产生运行错误“File not found”（文件未找到）；试图在只读目录中创建文件也会产生运行错误。

部分错误编译器不能发现，因为它们仅在运行程序时发生。

为编写出可靠、在运行时不崩溃的程序，应该使用异常处理。

Object Pascal 中有几种异常处理方法，见下。

### 2.6.1 Try—except 语句

语法：

```
try
  // 保护部分开始
  CallProc1;
  CallProc2;
  // 保护部分结束
except
on e: exception do // 异常处理
begin
  Writeln('Error: ' + e.message);
end;
end;
```

除数为 0 的示例如下。

```
program ExceptionHandling;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
```

```

Classes, sysutils
{ you can add units after this };

var
  x, y: Integer;
  Res: Double;
begin
  try
    Write('Input x: ');
    Readln(x);
    Write('Input y: ');
    Readln(y);
    Res:= x / y;
    Writeln('x / y = ', Res);

  except
  on e: exception do
  begin
    Writeln('An error occurred: ', e.message);
  end;
  end;
  Write('Press enter key to close');
  Readln;
end.

```

try 语句分为两部分。第一部分是受保护的语句块，在 **try** 和 **except** 之间；另一部分部分在 **except** 和 **end** 之间。如果 **try** 部分运行出错，程序将转到 **except** 部分。此时，程序不会崩溃，而是显示特定的错误信息并继续执行。

若 y 的值是 0，异常会在这一行产生。

```
Res:= x / y;
```

而如果没有异常，**except** 到 **end** 部分不被执行。

## 2.6.2 Try—finally

语法:

```

try
  // 保护部分开始
  CallProc1;
  CallProc2;
  // 保护部分结束

finally
  Writeln('This line will be printed in screen for sure');
end;

```

使用 try finally 方法实现上述程序。

```
program ExceptionHandling;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var
  x, y: Integer;
  Res: Double;
begin
  try
    Write('Input x: ');
    Readln(x);
    Write('Input y: ');
    Readln(y);
    Res:= x / y;
    Writeln('x / y = ', Res);

  finally
    Write('Press enter key to close');
    Readln;
  end;
end.
```

此时，不论异常是否出现，**finally** 至 **end** 部分都将被执行。

### 2.6.3 人为产生异常

有时，我们需要产生（raise）一个异常以阻止逻辑错误。例如，如果用户对月份变量输入 13，我们可以产生异常来告诉用户“该输入违背月份的范围”。

示例:

```
program RaiseExcept;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
```

```

Classes, SysUtils
{ you can add units after this };

var
  x: Integer;
begin
  Write('Please input a number from 1 to 10: ');
  Readln(X);
  try

    if (x < 1) or (x > 10) then // 产生异常
      raise exception.Create('X is out of range');
    Writeln(' X * 10 = ', x * 10);

  except
    on e: exception do // 捕捉异常
    begin
      Writeln('Error: ' + e.Message);
    end;
  end;
  Write('Press enter to close');
  Readln;
end.

```

如果用户输入的值在 1~10 范围外，将生成异常 “X is out of range”；没有异常处理的话程序将崩溃。因为我们在包含 **raise** 关键字的代码外编写 **try except**，程序不会崩溃而显示错误信息。

# 第三章

## 结构化编程

## 3.1 简介

在程序扩展时，需要使用结构化编程（structured programming），否则大型程序将既不可读又不可维护。

在结构化编程中，将一个程序的源代码分割为小的部分，称为过程（procedure）或函数（functions）。我们也可以将同一主题的过程和函数独立为另一个代码文件，称为单元（unit）。

结构化编程有如下优点：

1. 将程序代码分块（partition）为可读的模块和过程。
2. 代码有可重复使用（reusable）：过程和函数可在代码中多次调用，不需要重写代码。
3. 程序员可以在同一时间分享（share）、参与（participate）同一工程。程序员可以在不同单元中各自写过程与函数，再结合为一个工程。
4. 简化程序的维护（maintenance）和扩展（enhancement）：我们可以容易地在过程中找到错误，或改进程序中已有的过程和函数、写新的过程和函数、增加新单元等。
5. 引入模块（module）和层（layer）：可以照逻辑将程序划分为不同的层和模块。例如，第一层包含读写文件数据的过程，第二层包括验证规则与确认，第三层为用户界面。

## 3.2 过程

我们已经在前两章中用过一些过程，如 **Writeln**、**Readln**、**Reset** 等。现在，我们需要写新的过程，可以通过程序使用。

下例写了两个过程，SayHello 和 SayGoodbye。

```
program Structured;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

procedure SayHello;
begin
  Writeln('Hello there');
end;
```

```

procedure SayGoodbye;
begin
    Writeln('Good bye');
end;

begin // Here main application starts
    Writeln('This is the main application started');
    SayHello;
    Writeln('This is structured application');
    SayGoodbye;
    Write('Press enter key to close');
    Readln;
end.

```

我们看到，过程本身如同小程序，含有自己的 **begin—end** 对，也可以从主程序的代码中调用。

### 3.2.1 参数

下例引入了参数（parameters）。在调用过程时，参数是传递到其中的变量。

```

procedure WriteSum(x, y: Integer);
begin
    Writeln('The summation of ', x, ' + ', y, ' = ', x + y)
end;

begin
    WriteSum(2, 7);
    Write('Press enter key to close');
    Readln;
end.

```

在主应用程序中，我们调用 WriteSum 过程，并向它传递值 2、7。该过程将接受它们为整数变量 x、y，并输出它们的和。

下例中，使用相应的过程重写餐馆程序：



## 以过程重写的餐馆程序

```
procedure Menu;
begin
  Writeln('Welcome to Pascal Restaurant. Please select your order');
  Writeln('1 - Chicken      ($10)');
  Writeln('2 - Fish         ($7)');
  Writeln('3 - Meat           ($8)');
  Writeln('4 - Salad           ($2)');
  Writeln('5 - Orange Juice ($1)');
  Writeln('6 - Milk            ($1)');
  Writeln;
end;

procedure GetOrder(AName: string; Minutes: Integer);
begin
  Writeln('You have ordered: ', AName, ', this will take ',
    Minutes, ' minutes');
end;

// 主程序

var
  Meal: Byte;
begin
  Menu;
  Write('Please enter your selection: ');
  Readln(Meal);

  case Meal of
    1: GetOrder('Chicken', 15);
    2: GetOrder('Fish', 12);
    3: GetOrder('Meat', 18);
    4: GetOrder('Salad', 5);
    5: GetOrder('Orange juice', 2);
    6: GetOrder('Milk', 1);
  else
    Writeln('Wrong entry');
  end;
  Write('Press enter key to close');
  Readln;
end.
```

现在，主变得更小，也更可读。其它部分的细节则独立为过程，如 `GetOrder` 处理订单输入，`DisplayMenu` 处理输出菜单。

### 3.3 函数

函数类似于过程，但有一个不同：它有返回值。之前我们也用过 **UpperCase** 函数，转换文本为大写并返回；**Abs** 函数，返回数的绝对值。

下例编写了 **GetSum** 函数，接受两个整数值并且返回它们的和。

```
function GetSum(x, y: Integer): Integer;
begin
    Result:= x + y;
end;

var
    Sum: Integer;
begin
    Sum:= GetSum(2, 7);
    Writeln('Summation of 2 + 7 = ', Sum);
    Write('Press enter key to close');
    Readln;
end.
```

注意：我们声明函数（的返回值）为整型，并使用 **Result** 关键字来代表函数的返回值。

在主应用程序中，我们使用变量 **Sum**，并在其中接收函数结果。事实上，我们也可以取消这个中间变量，并在 **Writeln** 过程内部中调用这个函数。这是函数过程的一个不同。我们可以在其它过程或函数中调用函数作为输入参数，却不能调用过程作为其它函数和过程的参数。

```
function GetSumm(x, y: Integer): Integer;
begin
    Result:= x + y;
end;

begin
    Writeln('Summation of 2 + 7 = ', GetSumm(2, 7));
    Write('Press enter key to close');
    Readln;
end.
```

下例使用函数重写餐馆程序：

#### 使用函数的餐馆程序

```
procedure Menu;
begin
    Writeln('Welcome to Pascal Restaurant. Please select your order');
    Writeln('1 - Chicken      ($10)');
    Writeln('2 - Fish          ($7)');
```

```

Writeln('3 - Meat      ($8)');
Writeln('4 - Salad      ($2)');
Writeln('5 - Orange Juice ($1)');
Writeln('6 - Milk       ($1)');
Writeln('X - nothing');
Writeln;
end;

function GetOrder(AName: string; Minutes, Price: Integer): Integer;
begin
    Writeln('You have ordered: ', AName, ' this will take ',
        Minutes, ' minutes');
    Result:= Price;
end;

var
    Selection: Char;
    Price: Integer;
    Total: Integer;
begin
    Total:= 0;
    repeat
        Menu;
        Write('Please enter your selection: ');
        Readln(Selection);

        case Selection of
            '1': Price:= GetOrder('Chicken', 15, 10);
            '2': Price:= GetOrder('Fish', 12, 7);
            '3': Price:= GetOrder('Meat', 18, 8);
            '4': Price:= GetOrder('Salad', 5, 2);
            '5': Price:= GetOrder('Orange juice', 2, 1);
            '6': Price:= GetOrder('Milk', 1, 1);
            'x', 'X': Writeln('Thanks');
            else
                begin
                    Writeln('Wrong entry');
                    Price:= 0;
                end;
        end;
        Total:= Total + Price;

    until (Selection = 'x') or (Selection = 'X');
    Writeln('Total price      = ', Total);
    Write('Press enter key to close');
    Readln;
end.

```

## 3.4 局部变量

可以在过程或函数内部定义局部变量，该变量仅用于其代码内部。这类变量不能从主程序、其它过程或函数访问。

局部变量的示例如下。

```
procedure Loop(Counter: Integer);
var
  i: Integer;
  Sum: Integer;
begin
  Sum:= 0;
  for i:= 1 to Counter do
    Sum:= Sum + i;
  Writeln('Summation of ', Counter, ' numbers is: ', Sum);
end;

begin // 主程序
  Loop;
  Write('Press enter key to close');
  Readln;
end.
```

在 Loop 过程中，有两个局部变量：Sum 和 I。局部变量储存在栈存储器中。栈存储器用于临时分配变量，直到过程或函数的执行完成。所以，局部变量不可在主程序中访问，但当程序执行到这行时可以被覆盖：

```
Write('Press enter key to close');
```

全局变量可以从主程序、其他过程或函数访问。它们可以在程序关闭前保留值，但是这会破坏程序的结构，并使跟踪错误变得困难。这是因为一些过程可以更改全局变量值，所以可能会导致未知的变量值，或在未初始化时出现未知行为。

局部变量的定义确保它们的私有性，这有助于过程或函数的移植或任意调用，而不用担心全局变量值。

## 新闻数据库程序

本例中有 3 个过程和 1 个函数：AddTitle（写入新闻标题）、ReadAllNews（读取并输出已有新闻）、Search（查找新闻）、Menu（显示菜单并让用户选择）。

```
program news;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

type
  TNews = record
    ATime: TDateTime;
    Title: string[100];
  end;

procedure AddTitle;
var
  F: file of TNews;
  News: TNews;
begin
  AssignFile(F, 'news.dat');
  Write('Input current news title: ');
  Readln(News.Title);
  News.ATime:= Now;
  if FileExists('news.dat') then
  begin
    FileMode:= 2; // 进行读写
    Reset(F);
    Seek(F, System.FileSize(F)); // 跳转到文件末
  end
  else
    Rewrite(F);
  Write(F, News);
  CloseFile(F);
end;

procedure ReadAllNews;
var
  F: file of TNews;
  News: TNews;
begin
  AssignFile(F, 'news.dat');
  if FileExists('news.dat') then
  begin
```

```

    Reset(F);
    while not Eof(F) do
    begin
        Read(F, News);
        Writeln('-----');
        Writeln('Title: ', News.Title);
        Writeln('Time : ', DateTimeToStr(News.ATime));
    end;
    CloseFile(F);
end
else
    Writeln('Empty database');
end;

procedure Search;
var
    F: file of TNews;
    News: TNews;
    Keyword: string;
    Found: Boolean;
begin
    AssignFile(F, 'news.dat');
    Write('Input keyword to search for: ');
    Readln(Keyword);
    Found:= False;
    if FileExists('news.dat') then
    begin
        Reset(F);
        while not Eof(F) do
        begin
            Read(F, News);
            if Pos(LowerCase(Keyword), LowerCase(News.Title)) > 0 then
            begin
                Found:= True;
                Writeln('-----');
                Writeln('Title: ', News.Title);
                Writeln('Time : ', DateTimeToStr(News.ATime));
            end;
        end;
        CloseFile(F);
        if not Found then
            Writeln(Keyword, ' Not found');
    end
    else
        Writeln('Empty database');
end;

function Menu: char;
begin
    Writeln;
    Writeln('.....News database.....');

```

```

Writeln('1. Add news title');
Writeln('2. Display all news');
Writeln('3. Search');
Writeln('x. Exit');
Write('Please input a selection : ');
Readln(Result);
end;

// 主程序

var
  Selection: Char;

begin
  repeat
    Selection:= Menu;
    case Selection of
      '1': AddTitle;
      '2': ReadAllNews;
      '3': Search;
    end;
  until LowerCase(Selection) = 'x';
end.

```

本程序可读性强，主程序部分代码也简洁清晰。我们可以向任何过程中添加新的特征，而不影响或修改其它部分。

## 3.5 以函数作为输入参数

如前所述，我们可以调用函数作为过程或其他函数的输入参数，因为我们可以像数值一样对待它。

以下是一个以函数作为输入参数的例子。

```
function DoubleNumber(x: Integer): Integer;
begin
    Result:= x * 2;
end;

// 主程序

begin
    Writeln('The double of 5 is : ', DoubleNumber(5));
    Readln;
end.
```

下例作出了修改：使用中间变量来存储函数的结果，然后输入到 **Writeln** 过程。

```
unction DoubleNumber(x: Integer): Integer;
begin
    Result:= x * 2;
end;

// 主程序

var
    MyNum: Integer;
begin
    MyNum:= DoubleNumber(5);
    Writeln('The double of 5 is : ', MyNum);
    Readln;
end.
```

我们也可以在 **if** 条件和循环状态中调用函数。

```
unction DoubleNumber(x: Integer): Integer;
begin
    Result:= x * 2;
end;

// 主程序

begin
    if DoubleNumber(5) > 10 then
        Writeln('This number is larger than 10')
    else
        Writeln('This number is equal or less than 10');
```



```
Readln;  
end.
```

## 3.6 过程和函数输出参数

在先前的函数用法中，仅能返回一个值——函数的结果。如何才能在函数或过程中返回多个值？

以下是一个实验。

```
procedure DoSomething(x: Integer);  
begin  
    x:= x * 2;  
    Writeln('From inside procedure: x = ', x);  
end;  
  
// 主程序  
  
var  
    MyNumber: Integer;  
begin  
    MyNumber:= 5;  
  
    DoSomething(MyNumber);  
    Writeln('From main program, MyNumber = ', MyNumber);  
    Writeln('Press enter key to close');  
    Readln;  
end.
```

本例中的 doSomething 过程以整型接收 x，乘以 2，然后显示。

在主程序中，我们声明变量 MyNumber 为整型并赋值 5，然后把这个值作为参数传递到 DoSomething 过程中。在这种情况下，MyNumber 的值 5 被复制到 x 中。

在调用函数后，x 等于 10，但是 MyNumber 的显示表明它依然为 5。这表明 MyNumber 和 x 的存储地址不同，这是正常情况。

这种参数传递类型称为传值调用，不影响原始参数。我们也可以使用常量来传递值，如：

```
DoSomething(5);
```

### 3.6.1 传址调用

若向 DoSomething 的 x 参数声明中添加 **var** 关键词，事情将发生变化。

```
procedure DoSomething(var x: Integer);
begin
    x:= x * 2;
    Writeln('From inside procedure: x = ', x);
end;

// 主程序

var
    MyNumber: Integer;
begin
    MyNumber:= 5;

    DoSomething(MyNumber);
    Writeln('From main program, MyNumber = ', MyNumber);
    Writeln('Press enter key to close');
    Readln;
end.
```

此时，MyNumber 的值将随 x 更改，表明他们共享相同的存储器位置。

传送到过程中的变量（而非常量）此时应当与声明相同：如果参数被声明为 Byte，那么 MyNumber 也应该被声明为 Byte；如果参数是 Integer，那么 MyNumber 也应该是 Integer。

下例中的调用将产生错误，因为参数需要为变量。

```
DoSomething(5);
```

在传值调用时，过程仅关心参数的值，因此上述代码能正常执行；但在传址调用时，参数应该为变量，且过程修改该变量的值。

下例中传递了两个变量，并由过程交换它们的值。

```
procedure SwapNumbers(var x, y: Integer);
var
    Temp: Integer;
begin
    Temp:= x;
    x:= y;
    y:= Temp;
end;

// 主程序

var
```

```
A, B: Integer;  
begin  
  
    Write('Please input value for A: ');  
    Readln(A);  
  
    Write('Please input value for B: ');  
    Readln(B);  
  
    SwapNumbers(A, B);  
    Writeln('A = ', A, ', and B = ', B);  
    Writeln('Press enter key to close');  
    Readln;  
end.
```

## 3.7 单元

Pascal 中，单元（unit）是包含过程、函数、常量和自定义类型的库。同一个单元可以在很多程序中使用。

单元有以下用途：

1. 在外部的单元中积累频繁使用的过程和函数，使相应代码可复用。
2. 结合同一实体中用于执行某些任务的过程的函数。将程序分为不同的逻辑模块好于在主程序源代码中加入无关的过程和函数。

创建新单元时，在 Lazarus 菜单中选择 [文件—新建单元](#)，得到如下模板。

```
unit Unit1;  
  
{$mode objfpc}{$H+}  
  
interface  
  
uses  
    Classes, SysUtils;  
  
implementation  
  
end.
```

在创建新单元后可以命名保存，如 **Test**。它将被保存在一个命名为 **Test.pas** 的文件中。但是在应用程序中，单元名仍然为 **Test**。

在此之后，就可以在开始在单元中写过程、函数，或其他可重复使用的代码。

```
unit Test;  
  
{$mode objfpc}{$H+}  
  
interface  
  
uses  
    Classes, SysUtils;  
  
const  
    Price = 7.5;  
  
    function GetKilometers(Payment, Consumption: Integer): Single;  
  
implementation
```

```
function GetKilometers(Payment, Consumption: Integer): Single;
begin
    Result:= (Payment / Price) / Consumption * 100;
end;

end.
```

常量 `Price` 和函数 `GetKilometers` 可以从任意的外部程序调用。

函数声明放在单元的 **interface**（接口）部分，以使它从外部可访问。外部程序仅可访问该部分声明的内容。

为使用这个单元，我们在该单元（`Test.pas`）所在目录下创建新程序，并在 **uses** 分句中添加该单元：

```
program PetrolConsumption;

{$mode objfpc}{$H+}

uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
    cthreads,
    {$ENDIF}{$ENDIF}
    Classes, SysUtils
    { you can add units after this }, Test;

var
    Payment: Integer;
    Consumption: Integer;
    Kilos: Single;
begin
    Write('How much did you pay for your car's petrol: ');
    Readln(Payment);
    Write('What is the consumption of your car (Liter per 100 km) ');
    Readln(Consumption);

    Kilos:= GetKilometers(Payment, Consumption);

    Writeln('This petrol will keep your car running for: ',
        Format('%0.1f', [Kilos]), ' Kilometers');
    Write('Press enter to close');
    Readln;
end.
```

如果要转到 `GetKilometers` 函数的代码，可以按 `Ctrl` 键，然后点击名称；Lazarus（或 Delphi）将立即打开 `Test` 单元，显示该函数。

我们也可以在编辑器中移动光标到单元的名称上，并按 `Alt + Enter`。

我们可以在这些状态下从程序中访问单元：

1. 单元文件保存在程序的同一目录下，如前例。
2. 打开单元文件，然后单击 [源文件—添加单元到 Uses 部分](#)。
3. 在 [工程—工程选项/编译器选项/路径/其他单元文件](#) 中添加该单元的路径。

### 3.7.1 Lazarus 和 Free Pascal 中的单元

在 Lazarus 和 Free Pascal 中，单元是程序最重要的构建单位。大多数的过程、函数、类型、类和组件都写在单元中。

Free Pascal 的库中有非常丰富的单元，它们功能可被广泛地用于各类程序开发。因此，Lazarus 和 Free Pascal 可以快速、简单地开发应用程序。

Free Pascal、Lazarus 库中的单元，如 `SysUtils` 或 `Classes`，包含常规过程和函数。

### 3.7.2 程序员编写的单元

在编程库的单元之外，程序员也可以写自己的单元，以解决他们的特殊需要。例如，如果一个开发者为汽车车库写软件，就可以制作一个包括添加新车数据、搜索车牌号等过程的单元。

将过程和函数放置在单元内使应用程序更可读，也更易于协作开发，因为每个人只需专心于少数模块（单元），并独立测试，最后再结合为一个工程。

## 大流士火星历

本代码描述了一种未来人类开拓者可能在火星上使用的历法。这种历法是基于托马斯·纲吉尔于 1985 年的草案编制而成的。这种火星历法基于以下四条事实。

1. 火星历 0 年元旦等于公历 1609 年 3 月 12 日；
2. 火星历的一年有 668.5910 火星日；
3. 火星历的一年分为 24 个月，除 6、12、28、24 月外，每月固定为 28 天；
4. 1 个火星日等于 1.02749125 地球日。

以下是火星单元的示例代码：

```
{
*****

  DarianUtils: 用于Object Pascal的火星历法转换器
  作者:       Fang
  协议:       LGPL
  编写日期:   2018-12-30

*****
}

unit DarianUtils;

{$IFDEF FPC}
{$mode objfpc}{$H+}
{$ENDIF}

interface

uses
  Classes, SysUtils;

const
  DarianMonths: array [1 .. 24] of string = ('Sagittarius', 'Dhanus',
    'Capricornus', 'Makara', 'Aquarius', 'Kumbha', 'Pisces', 'Mina',
    'Aries', 'Mesha', 'Taurus', 'Rishabha', 'Gemini', 'Mithuna', 'Cancer',
    'Karka', 'Leo', 'Simha', 'Virgo', 'Kanya', 'Libra', 'Tula',
    'Scorpius', 'Vrishika');

  DarianYearSols = 668.5910;
  DarianQuarterSols = 167.14775;
  DarianSolDays = 1.02749125;

  procedure DateToDarian(ADateTime: TDateTime;
    var Year, Month, Sol: Word);
  function DarianToSols(Year, Month, Sol: Word): Double;
  function DarianToDate(Year, Month, Sol: Word): TDateTime;
```

```

function DarianDifference(Year1, Month1, Sol1, Year2, Month2,
    Sol2: Word): Cardinal;

implementation

var
    DarianStart : TDateTime;

procedure DateToDarian(ATime: TDateTime; var Year, Month, Sol: Word);
var
    Sols: Double;
    Quarter: Word;
begin
    Sols:= (ATime - DarianStart - 1) / DarianSolDays;
    Year:= trunc(Sols / DarianYearSols);
    Sols:= Sols - Year * DarianYearSols;
    Quarter:= trunc(Sols / DarianQuarterSols);
    Sols:= Sols - Quarter * DarianQuarterSols;
    Month:= Quarter * 4 + trunc(Sols / 28) + 1;
    Sol:= Trunc(Sols) mod 28 + 1;
end;

function DarianToSols(Year, Month, Sol: Word): Double;
begin
    Result:= Year * DarianYearSols + (Month div 6) * DarianQuarterSols +
        (Month mod 6) * 28 + Sol;
end;

function DarianToDate(Year, Month, Sol: Word): TDateTime;
begin
    Result:= DarianToSols(Year, Month, Sol) * DarianSolDays + DarianStart;
end;

function DarianDifference(Year1, Month1, Sol1, Year2, Month2,
    Sol2: Word): Cardinal;
begin
    Result:= trunc(Abs(DarianToSols(Year1, Month1, Sol1) -
        DarianToSols(Year2, Month2, Sol2)));
end;

initialization

    DarianStart:= EncodeDate(1609, 3, 12);

end.

```

上述单元包含如下过程和函数：

1. DateToDarian 过程转换公历日期为到火星历，如下例。



```

program Project1;

{$mode objfpc}{$H+}

uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
    cthreads,
    {$ENDIF}{$ENDIF}
    Classes, DarianUtils, SysUtils
    { you can add units after this };

var
    Year, Month, Day: Word;
begin
    DateToDarian(Now, Year, Month, Day);
    Writeln('Today in Darian: ', Year, '-', DarianMonths[Month],
        '-', Day);
    Readln;
end.

```

2. DarianToSols 计算火星历日期到历元的差，以火星日表示。
3. DarianToDate 函数转换火星历日期到公历的 TDateTime 值。
4. DarianDifference 过程计算两个火星历日期之间的差，以火星日表示。

## 3.8 重载

“重载”指编写多个名称相同，但参数不同的过程或函数。参数的“不同”可以指类型或数量。例如，我们可能编写两个版本的 Sum 函数，第一个的参数和返回值是整型，第二个是浮点型：

```
program sum;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

function Sum(x, y: Integer): Integer; overload;
begin
  Result:= x + y;
end;

function Sum(x, y: Double): Double; overload;
begin
  Result:= x + y;
end;

var
  j, i: Integer;
  h, g: Double;
begin
  j:= 15;
  i:= 20;
  Writeln(J, ' + ', I, ' = ', Sum(j, i));

  h:= 2.5;
  g:= 7.12;
  Writeln(H, ' + ', G, ' = ', Sum(h, g));

  Write('Press enter key to close');
  Readln;
end.
```

关键字 `overload` 表示该函数被重载。

## 3.9 排序

数据排序 (sorting) 属于数据结构 (data structure) 主题，我们在这里介绍它，因为它需要过程或函数来实施。排序经常与数组或列表一起使用。

现在假设有一个整型数组，需要以升序或降序排序，则可以使用以下介绍的不同方法完成。本节中所有的算法都以排升序为假定，排降序时可相应调整。

### 3.9.1 冒泡排序

冒泡排序是最简单的排序算法的一种。在该算法中，程序先把数组第 2 项与第 1 项比较，如第 1 项大于第 2 项，则进行交换；之后，再把第 3 项与第 2 项比较，以此类推，直到数组结尾。在此之后，程序重复这个操作，直到整个数组已经被排序。

假设，我们在一个数组中有如下 6 项：

```
7, 10, 2, 5, 6, 3
```

不难推出，在第一次操作之后，数组内容如下：

```
7, 2, 5, 6, 3, 10
```

第二轮操作后：

```
7, 2, 5, 6, 3, 10
```

第三轮：

```
2, 5, 3, 6, 7, 10
```

第四轮：

```
2, 3, 5, 6, 7, 10
```

可以看到，第  $n$  轮排序之后至少已经完成了最后  $n$  项的排序。因为第二项排好时第 1 项也自动排好，因此 6 项排序不超过 5 轮（本例中为 4 轮，程序实际需要循环 5 次，第 5 次用于检查）。

“冒泡”指在操作中较小的项像气泡一样向上浮动，而最大的值则“沉”到底。

依据上面的例子，可以写出如下的程序。

```
program BubbleSortProj;  
  
{$mode objfpc}{$H+}  
  
uses
```

```

{$IFDEF UNIX}{$IFDEF UseCThreads}
cthreads,
{$ENDIF}{$ENDIF}
Classes;

procedure BubbleSort(var X: array of Integer);
var
    Temp: Integer;
    round: Integer;
    i: Integer;
    Changed: Boolean;
begin
    round:= 1;
    repeat // 外循环
        Changed:= False;
        for i:= 0 to High(X) - round do // 内循环
            if X[i] > X[i + 1] then
                begin
                    Temp:= X[i];
                    X[i]:= X[i + 1];
                    X[i + 1]:= Temp;
                    Changed:= True;
                end;
            round:= round + 1;
        until not Changed or (High(X) = round + 1);
    end;

var
    Numbers: array [0 .. 9] of Integer;
    i: Integer;
begin
    Writeln('Please input 10 random numbers');
    for i:= 0 to High(Numbers) do
        begin
            Write('#', i + 1, ': ');
            Readln(Numbers[i]);
        end;

    BubbleSort(Numbers);
    Writeln;
    Writeln('Numbers after sort: ');

    for i:= 0 to High(Numbers) do
        begin
            Writeln(Numbers[i]);
        end;
    Write('Press enter key to close');
    Readln;
end.

```

若将条件中的大于号改为小于号，则可以排成降序，如下一行所示。

```
if X[i] < X[i + 1] then
```

下例中展示以降序排序学生成绩。

## 排序学生成绩

```
program smSort;    // sm为 student mark

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

type
  TStudent = record
    StudentName: string;
    Mark: Byte;
  end;

procedure BubbleSort(var X: array of TStudent);
var
  Temp: TStudent;
  round: Integer;
  i: Integer;
  Changed: Boolean;
begin
  round:= 1;
  repeat
    Changed:= False;
    for i:= 0 to High(X) - 1 do
      if X[i].Mark < X[i + 1].Mark then
        begin
          Temp:= X[i];
          X[i]:= X[i + 1];
          X[i + 1]:= Temp;
          Changed:= True;
        end;
    until not Changed or (High(X) = round + 1);
end;

var
  Students: array [0 .. 9] of TStudent;
  i: Integer;

begin
```

```

Writeln('Please input 10 student names and marks');
for i:= 0 to High(Students) do
begin
Write('Student #', i + 1, ' name : ');
Readln(Students[i].StudentName);

    Write('Student #', i + 1, ' mark : ');
    Readln(Students[i].Mark);
    Writeln;
end;

BubbleSort(Students);
Writeln;
Writeln('Marks after Bubble sort: ');
Writeln('-----');

for i:= 0 to High(Students) do
begin
    Writeln('# ', i + 1, ' ', Students[i].StudentName,
        ' with mark (', Students[i].Mark, ')');
end;

Write('Press enter key to close');
Readln;
end.

```

冒泡排序算法十分简单，便于记忆。但是，它仅适用于少量或略有序的数据；对于大量的无序数据，它会耗费很多时间。

### 3.9.2 选择排序

这个算法的原理是每轮中选出最小值排在第 1 项，再对第 2 项以后的部分选出第 2 项，以此类推，直到只有两项。

```

program SelectionSort;

{$mode objfpc}{$H+}

uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
    cthreads,
    {$ENDIF}{$ENDIF}
    Classes
    { you can add units after this };

procedure SelectSort(var X: array of Integer);
var
    i: Integer;

```

```

j: Integer;
SmallPos: Integer;
Smallest: Integer;
begin
  for i:= 0 to High(X) -1 do // 外循环
  begin
    SmallPos:= i;
    Smallest:= X[SmallPos];
    for j:= i + 1 to High(X) do // 内循环
      if X[j] < Smallest then
      begin
        SmallPos:= j;
        Smallest:= X[SmallPos];
      end;
    X[SmallPos]:= X[i];
    X[i]:= Smallest;
  end;
end;

// 主程序

var
  Numbers: array [0 .. 9] of Integer;
  i: Integer;

begin
  Writeln('Please input 10 random numbers');
  for i:= 0 to High(Numbers) do
  begin
    Write('#', i + 1, ': ');
    Readln(Numbers[i]);
  end;

  SelectSort(Numbers);
  Writeln;
  Writeln('Numbers after Selection sort: ');

  for i:= 0 to High(Numbers) do
  begin
    Writeln(Numbers[i]);
  end;
  Write('Press enter key to close');
  Readln;
end.

```

一般而言，选择排序稍快于冒泡排序；但是在数据本身较为有序时，冒泡排序因为可以跳出循环，反而更快。

### 3.9.3 堆排序

堆排序是以“堆”（heap）这种数据类型为基础的。堆是一种特殊的二叉树，需要保证每个有子节点的节点上的数都比子节点大。容易看出，节点的顶层就是堆中最大的值，可以直接放到已经排好的更大值前。程序再如此运行，最终将数组按照升序排序。

在将堆用一维线性表示时，将下标为  $n$ （以 0 开始）的两个子节点分别设为  $2n+1$  和  $2n+2$ ，然后通过比较它们的大小进行堆的整理，再进行排序。

```
program HeapSort;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils;

procedure SiftDown(var X: array of Integer; offset, max: Integer);
// 进行一层整理
var
  tmp, tmpOffset, child1, child2: Integer;
begin
  child1:= 2 * offset + 1;
  child2:= 2 * offset + 2;

  if child2 <= max then // 两个子节点
  begin
    if (X[offset] < X[child1]) or (X[offset] < X[child2]) then // 需要交换
    begin
      if X[child1] > X[child2] then // 左子节点最大
        tmpOffset:= child1
      else // 右子节点最大
        tmpOffset:= child2;
      // 进行交换
      tmp:= X[tmpOffset];
      X[tmpOffset]:= X[offset];
      X[offset]:= tmp;
      // 递归向下
      SiftDown(X, child1, max);
      SiftDown(X, child2, max);
    end;
  end else if child1 <= max then // 一个子节点
  begin
    if X[offset] < X[child1] then // 需要交换
    begin
      tmp:= X[child1];
      X[child1]:= X[offset];

```



```

        X[offset] := tmp;
    end;
    // 单节点不再递归
end;
// 没有子节点时已经不需要交换
end;

procedure Heapify(var X: array of Integer);
var
    max, offset, i: Integer;
begin
    max := High(X);

    // 计算最大一层的起始
    offset := 0;
    while offset < max do
        offset := 2 * offset + 1;
    offset := offset div 2;

    // 分层进行整理
    repeat
        for i := offset to offset * 2 do
            SiftDown(X, i, max);
        offset := offset div 2;
    until offset = 0;
    SiftDown(X, 0, max);
end;

procedure HeapSort(var X: array of Integer);
var
    max, tmp: Integer;
begin
    max := High(X);
    Heapify(X); // 第一次整理
    repeat
        tmp := X[max]; // 交换
        X[max] := X[0];
        X[0] := tmp;
        max := max - 1; // 移出最大一项
        SiftDown(X, 0, max); // 整理余下各项
    until max = 0;
end;

var
    Numbers: array [0 .. 9] of Integer;
    i: Integer;

begin
    Writeln('Please input 10 random numbers');
    for i := 0 to High(Numbers) do
        begin

```

```

    Write('#', i + 1, ': ');
    Readln(Numbers[i]);
end;

HeapSort(Numbers);
Writeln;
Writeln('Numbers after Heap sort: ');

for i:= 0 to High(Numbers) do
begin
    Writeln(Numbers[i]);
end;
Write('Press enter key to close');
Readln;
end.

```

### 3.9.4 希尔排序

该排序方法由发明者唐纳德·希尔（Donald Shell）命名。它在数据量巨大时非常快速。在数据半排序时，它的表现如同冒泡排序，但更复杂。

```

program ShellSort;

{$mode objfpc}{$H+}

uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
    cthreads,
    {$ENDIF}{$ENDIF}
    Classes;

procedure ShellS(var X: array of Integer);
var
    Done: Boolean;
    Jump, j, i: Integer;
    Temp: Integer;
begin
    Jump:= High(X);
    while (Jump > 0) do // 外循环
    begin
        Jump:= Jump div 2;
        repeat // 中循环
            Done:= True;
            for j:= 0 to High(X) - Jump do // 内循环
            begin
                i:= j + Jump;
                if X[j] > X[i] then // 交换
                begin

```

```

        Temp:= X[i];
        X[i]:= X[j];
        X[j]:= Temp;
        Done:= False;
    end;

    end; // 内循环结束
until Done; // 中循环结束
end; // 外循环结束
end;

var
    Numbers: array [0 .. 9] of Integer;
    i: Integer;

begin
    Writeln('Please input 10 random numbers');
    for i:= 0 to High(Numbers) do
    begin
        Write('#', i + 1, ': ');
        Readln(Numbers[i]);
    end;

    Shells(Numbers);
    Writeln;
    Writeln('Numbers after Shell sort: ');

    for i:= 0 to High(Numbers) do
    begin
        Writeln(Numbers[i]);
    end;
    Write('Press enter key to close');
    Readln;
end.

```

### 3.9.5 其他排序算法

归并排序和快速排序也是两种效率比较高的排序算法。但是，因为它们需要先进行计算输入数组的长度，再进行排序，因此不适用于 Pascal 语言。

### 3.9.6 字符串排序

字符串的大小比较是依据自左向右的原则。例如，因为 A 在 B 前，所以 'Alice' 小于 'Bob'。当两个字符串的首字母相同时，则比较第二字母，如 'Alice' 小于 'Anna'（因为 l 在 n 前），以此类推。

## 学生姓名排序程序

```
program smSort;    // Students mark sort by name

{$mode objfpc}{$H+}

uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
    cthreads,
    {$ENDIF}{$ENDIF}
    Classes;

type
    TStudent = record
        StudentName: string;
        Mark: Byte;
    end;

procedure BubbleSort(var X: array of TStudent);
var
    Temp: TStudent;
    i: Integer;
    Changed: Boolean;
begin
    repeat
        Changed:= False;
        for i:= 0 to High(X) - 1 do
            if X[i].StudentName > X[i + 1].StudentName then
                begin
                    Temp:= X[i];
                    X[i]:= X[i + 1];
                    X[i + 1]:= Temp;
                    Changed:= True;
                end;
        until not Changed;
end;

var
    Students: array [0 .. 9] of TStudent;
    i: Integer;

begin
    Writeln('Please input 10 student names and marks');
    for i:= 0 to High(Students) do
        begin
            Write('Student #', i + 1, ' name : ');
            Readln(Students[i].StudentName);

            Write('Student #', i + 1, ' mark : ');
```

```

    Readln(Students[i].Mark);
    Writeln;
end;

BubbleSort(Students);
Writeln;
Writeln('Marks after Bubble sort: ');
Writeln('-----');

for i:= 0 to High(Students) do
begin
    Writeln('# ', i + 1, ' ', Students[i].StudentName,
        ' with mark (', Students[i].Mark, ')');
end;
Write('Press enterkey to close');
Readln;
end.

```

### 3.9.7 排序算法比较

本例使用一个较大的整型数组的排序比较本章中的四种排序算法。算法效率由时间表示，用时越短效率越高。

```

program SortComparison;

{$mode objfpc}{$H+}

uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
    cthreads,
    {$ENDIF}{$ENDIF}
    Classes, SysUtils;

//冒泡排序
procedure BubbleSort(var X: array of TStudent);
var
    Temp: TStudent;
    round: Integer;
    i: Integer;
    Changed: Boolean;
begin
    round:= 1;
    repeat
        Changed:= False;
        for i:= 0 to High(X) - 1 do
            if X[i].Mark < X[i + 1].Mark then
                begin
                    Temp:= X[i];
                    X[i]:= X[i + 1];
                    X[i + 1]:= Temp;
                end;
        until Changed;
        round:= round + 1;
    until round = High(X) + 1;
end;

```

```

        Changed:= True;
    end;
    until not Changed or (High(X) = round + 1);
end;

// 选择排序
procedure SelectionSort(var X: array of Integer);
var
    i: Integer;
    j: Integer;
    SmallPos: Integer;
    Smallest: Integer;
begin
    for i:= 0 to High(X) -1 do // 外循环
    begin
        SmallPos:= i;
        Smallest:= X[SmallPos];
        for j:= i + 1 to High(X) do // 内循环
            if X[j] < Smallest then
                begin
                    SmallPos:= j;
                    Smallest:= X[SmallPos];
                end;
            X[SmallPos]:= X[i];
            X[i]:= Smallest;
        end;
    end;
end;

// 堆排序
procedure SiftDown(var X: array of Integer; offset, max: Integer);
var
    tmp, tmpOffset, child1, child2: Integer;
begin
    child1:= 2 * offset + 1;
    child2:= 2 * offset + 2;

    if child2 <= max then // 两个子节点
    begin
        if (X[offset] < X[child1]) or (X[offset] < X[child2]) then
            begin
                if X[child1] > X[child2] then
                    tmpOffset:= child1
                else
                    tmpOffset:= child2;
                // 进行交换
                tmp:= X[tmpOffset];
                X[tmpOffset]:= X[offset];
                X[offset]:= tmp;
                // 递归向下
            end
        end
    end

```

```

        SiftDown(X, child1, max);
        SiftDown(X, child2, max);
    end;
end else if child1 <= max then // 一个子节点
begin
    if X[offset] < X[child1] then
    begin
        tmp:= X[child1];
        X[child1]:= X[offset];
        X[offset]:= tmp;
    end;
end;
end;

procedure Heapify(var X: array of Integer);
var
    max, offset, i: Integer;
begin
    max:= High(X);

    offset:= 0;
    while offset < max do
        offset:= 2 * offset + 1;
    offset:= offset div 2;

    repeat
        for i:= offset to offset * 2 do
            SiftDown(X, i, max);
        offset:= offset div 2;
    until offset = 0;
    SiftDown(X, 0, max);
end;

procedure HeapSort(var X: array of Integer);
var
    max, tmp: Integer;
begin
    max:= High(X);
    Heapify(X);
    repeat
        tmp:= X[max];
        X[max]:= X[0];
        X[0]:= tmp;
        max:= max - 1
        SiftDown(X, 0, max);
    until max = 0;
end;

// 希尔排序
procedure ShellSort(var X: array of Integer);

```

```

var
  Done: Boolean;
  Jump, j, i: Integer;
  Temp: Integer;
begin
  Jump:= High(X);
  while (Jump > 0) do // 外循环
  begin
    Jump:= Jump div 2;
    repeat // 中循环
      Done:= True;
      for j:= 0 to High(X) - Jump do // 内循环
      begin
        i:= j + Jump;
        if X[j] > X[i] then // 交换
        begin
          Temp:= X[i];
          X[i]:= X[j];
          X[j]:= Temp;
          Done:= False;
        end;

      end; // 内循环结束
    until Done; // 中循环结束
  end; // 外循环结束
end;

//打乱数据
procedure RandomizeData(var X: array of Integer);
var
  i: Integer;
begin
  Randomize;
  for i:= 0 to High(X) do
    X[i]:= Random(100000000);
end;

var
  Numbers: array [0 .. 59999] of Integer;
  StartTime: TDateTime;

begin
  Writeln('----- Full random data');
  RandomizeData(Numbers);
  StartTime:= Now;
  Writeln('Sorting.. Bubble');
  BubbleSort(Numbers);
  Writeln('Bubble sort tooks (mm:ss): ',
    FormatDateTime('nn:ss', Now - StartTime));
  Writeln;

```



```

RandomizeData(Numbers);
Writeln('Sorting.. Selection');
StartTime:= Now;
SelectionSort(Numbers);
Writeln('Selection sort tooks (mm:ss): ',
    FormatDateTime('nn:ss', Now - StartTime));
Writeln;

RandomizeData(Numbers);
Writeln('Sorting.. Heap');
StartTime:= Now;
HeapSort(Numbers);
Writeln('Heap sort tooks (mm:ss): ',
    FormatDateTime('nn:ss', Now - StartTime));
Writeln;

RandomizeData(Numbers);
Writeln('Sorting.. Shell');
StartTime:= Now;
ShellSort(Numbers);
Writeln('Shell sort tooks (mm:ss): ',
    FormatDateTime('nn:ss', Now - StartTime));

Writeln;
Writeln('----- Nearly sorted data');
Numbers[0]:= Random(10000);
Numbers[High(Numbers)]:= Random(10000);
StartTime:= Now;
Writeln('Sorting.. Bubble');
BubbleSort(Numbers);
Writeln('Bubble sort tooks (mm:ss): ',
    FormatDateTime('nn:ss', Now - StartTime));
Writeln;

Numbers[0]:= Random(10000);
Numbers[High(Numbers)]:= Random(10000);
Writeln('Sorting.. Selection');
StartTime:= Now;
SelectionSort(Numbers);
Writeln('Selection sort tooks (mm:ss): ',
    FormatDateTime('nn:ss', Now - StartTime));
Writeln;

Numbers[0]:= Random(10000);
Numbers[High(Numbers)]:= Random(10000);
Writeln('Sorting.. Selection');
StartTime:= Now;
HeapSort(Numbers);
Writeln('Heap sort tooks (mm:ss): ',
    FormatDateTime('nn:ss', Now - StartTime));

```

```
Writeln;  
  
Numbers[0]:= Random(10000);  
Numbers[High(Numbers)]:= Random(10000);  
Writeln('Sorting.. Shell');  
StartTime:= Now;  
ShellSort(Numbers);  
Writeln('Shell sort tooks (mm:ss): ',  
    FormatDateTime('nn:ss', Now - StartTime));  
  
Write('Press enter key to close');  
Readln;  
end.
```

# 第四章

## 图形用户界面

## 4.1 介绍

图形用户界面 (Graphical User Interface, GUI) 相比于控制台界面, 是一个新的选择。它包含窗体 (form), 按钮 (button), 消息框 (message box), 菜单 (menu), 复选框 (check box) 等图形化组件。用户使用图形界面的应用程序比控制台应用程序更容易。

图形界面的应用十分广泛, 包括商业程序、系统程序、游戏、Lazarus 等开发工具等。

## 4.2 第一个图形界面程序

创建图形界面的程序时, 需要在菜单点击

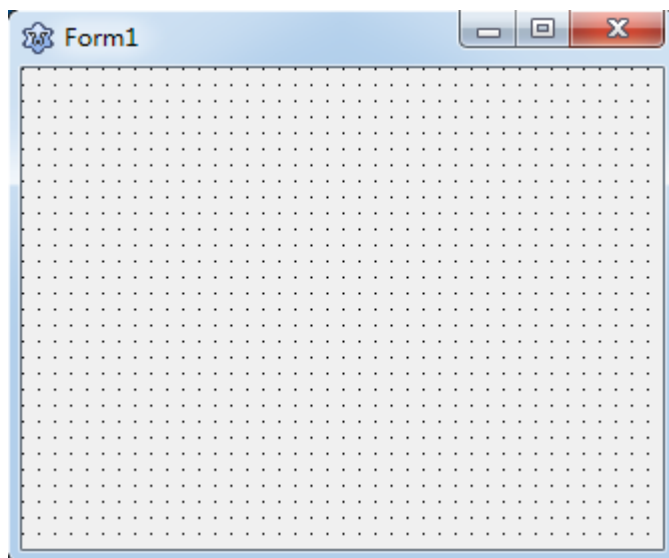
工程 (P)—新建工程...—应用程序

项。保存程序时, 则需要点击

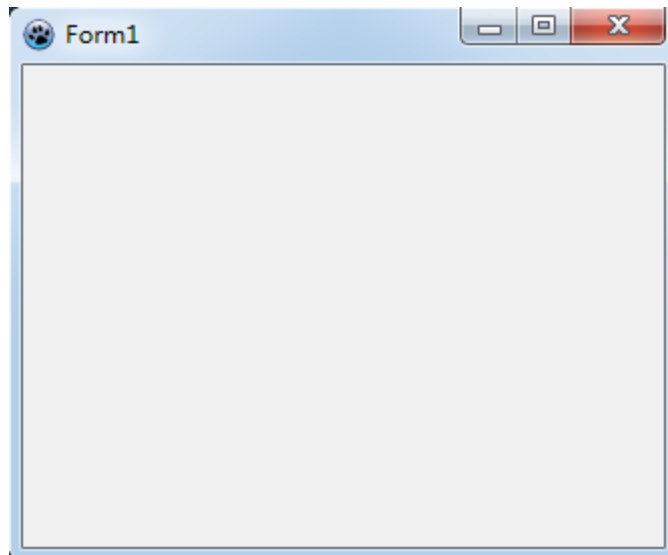
文件—保存所有

项。

我们可以创建一个新的文件夹, 例如命名为 `firstgui`, 并在其中来保存我们的工程文件。之后保存主单元, 如命名为 `main.pas`。最后, 我们将选择工程名, 如 `firstgui.lpi`。在主单元中可以按 `F12` 键查看与之相关联的窗体, 如图所示。

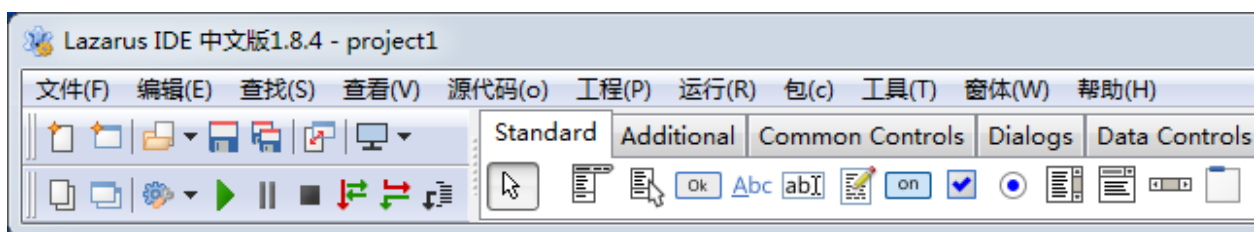


在编译、运行之后, 窗口则如下图所示。

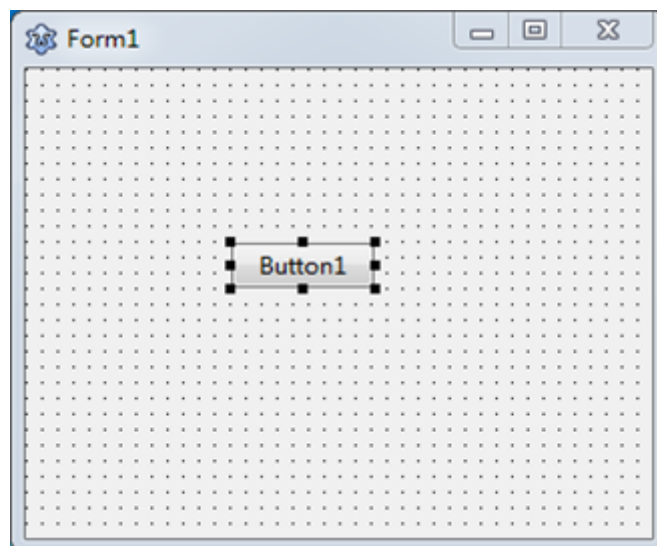


然后，我们可以关闭这个应用程序来返回设计器。

在这以后，我们从“standard”组件页拖拽一个按钮到窗体上。



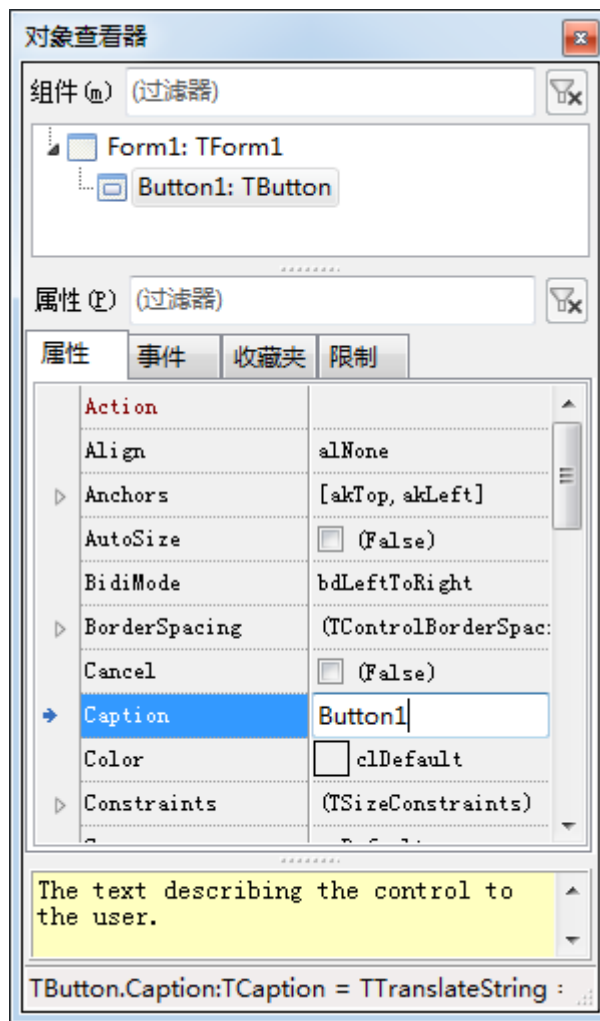
然后在该窗体的任意位置放置这个按钮，如图所示。



我们可以在“对象查看器”窗口中显示该按钮的属性（如标题名称、宽等）。如果它没有出现在 Lazarus 的窗口中。那么我们可以通过从主菜单点击

[窗体/对象查看器](#)

来显示它，或按 F11 键，则将获得这样的窗口。



在我们进行任何修改前，必须确保所选择的是按钮而非窗体。对象查看器窗口分为标签页，如第一项是属性（properties），第二项是事件（events）；每一个标签都有单独的页面。

先更改 Caption 属性，来变更按钮中的文本。然后，选择对象查看器的“事件”标签切换页面，并且双击 OnClick 事件，则将创建这样的代码模板。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
end;
```

然后加入相应代码。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage('Hello world, this is my first GUI application');
end;
```

因此，在应用程序中，点击按钮时将显示相应的对话框窗口。

在 Linux 下，我们将在工程目录下找到可执行的 `firstgui`，在 Windows 下则为 `firstgui.exe`。这些文件可以复制到没有 Lazarus 的其他计算机运行。

先前的示例展示了如下几个要点。

1. 主程序文件：本例中为 `firstgui.lpi`。我们可以通过点击 [工程/查看工程源代码](#) 以显示该文件。我们将获得如下的代码。

```
program firstgui;

{$mode objfpc}{$H+}

uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
    cthreads,
    {$ENDIF}{$ENDIF}
    Interfaces, // this includes the LCL widgetset
    Forms
    { you can add units after this }, main;

begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.
```

在某些特定情况下，我们可能需要修改这些代码；但一般情况下可以不作修改，并让 Lazarus 自动管理。

2. 主单元：此单元包含当我们运行应用程序时窗体自动的出现的定义。

下面是在上面已经完成 `OnClick` 事件代码的主单元代码。

```
unit main;

{$mode objfpc}{$H+}

interface

uses
    Classes, SysUtils, FileUtil, LResources, Forms, Controls,
    Graphics, Dialogs, StdCtrls;

type
    { TForm1 }

    TForm1 = class(TForm)
        Button1: TButton;
        procedure Button1Click(Sender: TObject);
    private
        { private declarations }
    end;

end.
```

```

public
    { public declarations }
end;

var
    Form1: TForm1;

implementation

{ TForm1 }

procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage('Hello world, this is my first GUI application');
end;

initialization
    {$I main.lrs}

end.

```

主单元头部有 TForm1 的声明。该类型与记录类似，但是是一个类。类将在下一章“面向对象编程”中讨论。Button1 是在 TForm1 类中声明的。

我们将通过按 Ctrl—F12 后选择 main 单元来显示本单元的源代码。

3. 对象查看器—属性：在“对象查看器”页面可以查看和修改任何组件的属性，包括按钮的标题或位置，窗体颜色，或标签字体等。这些组件可以类比为记录格式，其属性类似于字段。
4. 对象查看器—事件：对象查看器的这页面包含一个组件的我们可以接受的事件，如按钮的“点击”事件、编辑框的“按键”，或标签的“双击”等。当我们选择一个事件时，Lazarus 为我们创建一个新的过程模板，用于输入该事件发生时将被调用的代码，如下方为“点击”事件的示例。

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage('Hello world, this is my first GUI application');
end;

```

当用户点击此按钮时，该过程将调用事件处理程序。



## 4.3 第二个程序

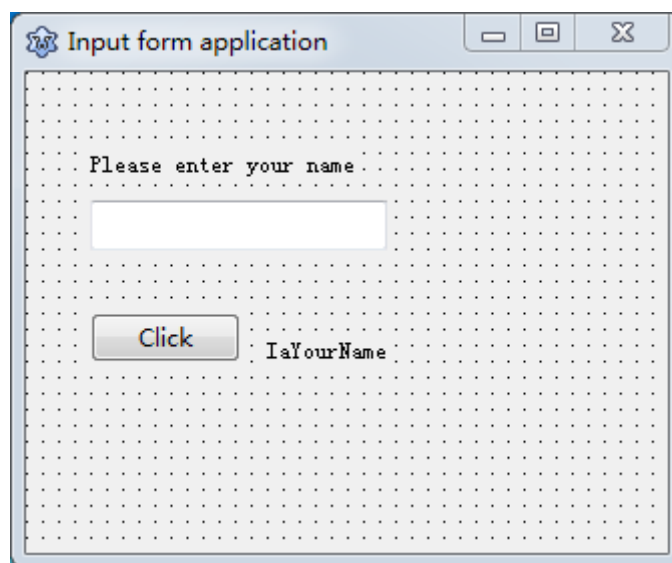
在本例中，我们将让用户在编辑框中输入姓名，并在点击按钮时在标签中显示消息。  
该程序可通过如下步骤创建。

- 创建一个新的应用程序，并保存为 `inputform`。保存主单元为 `main.pas`，然后从“标准”组件标签页上拖拽如下组件到窗体上。
  - 两个标签
  - 编辑框
  - 按钮

如下所示修改组件属性。

```
[Form1]   Name: fmMain  
          Caption: Input form application  
  
[Label1]  Caption: Please enter your name  
  
[Label2]  Name: laYourName  
  
[Edit1]   Name: edName  
          Text:  
  
[Button1] Caption: Click
```

如图在窗体上放置组件。



- 在 `OnClick` 事件中加入这些代码。

```
procedure TfmMain.Button1Click(Sender: TObject);
begin
    laYourName.Caption:= 'Hello ' + edName.Text;
end;
```

然后，可以运行应用程序，在编辑框中输入名字，并击按钮。

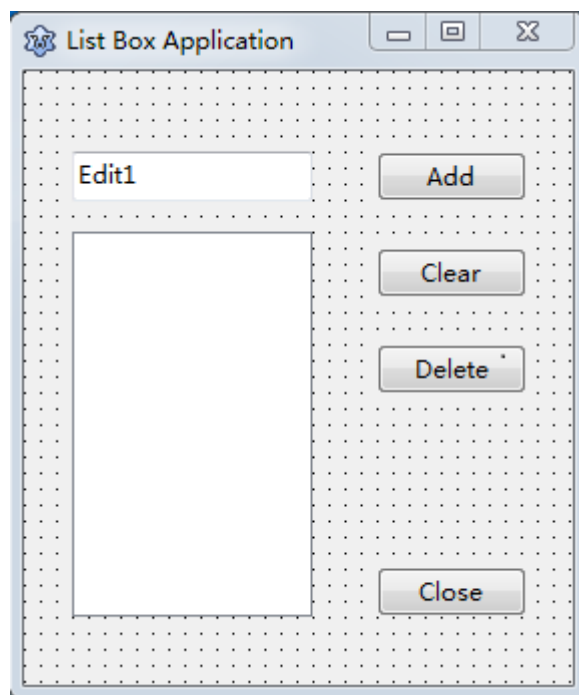
在上例中，我们在编辑框 **edName** 中使用字段 **Text** 读取用户的键入。这是命令行程序中 **Readln** 过程在图形界面下的替代办法。

我们也使用标签 **laYourName** 的 **Caption** 属性显示消息。这是图形界面下 **Writeln** 过程的替代。

## 4.4 列表框程序

在本例中，需要在列表框中添加文本，删除文本，和清除列表。该程序可通过如下步骤创建。

- 创建一个新的应用程序和在它的主窗体上放置四个按钮，一个编辑框，和一个列表框。
- 更改按钮的名称为 **btAdd**、**btClear**、**btDelete**、**btClose**。
- 依照下图更改按钮的 **Caption** 字段。



- 为按钮的 **OnClick** 事件写这些如下程序。

```

procedure TForm1.btAddClick(Sender: TObject);
begin
    ListBox1.Items.Add(Edit1.Text);
end;

procedure TForm1.btClearClick(Sender: TObject);
begin
    ListBox1.Clear;
end;

procedure TForm1.btDeleteClick(Sender: TObject);
var
    Index: Integer;
begin
    Index:= ListBox1.ItemIndex;
    if Index <> -1 then
        ListBox1.Items.Delete(Index);
end;

procedure TForm1.btCloseClick(Sender: TObject);
begin
    Close;
end;

```

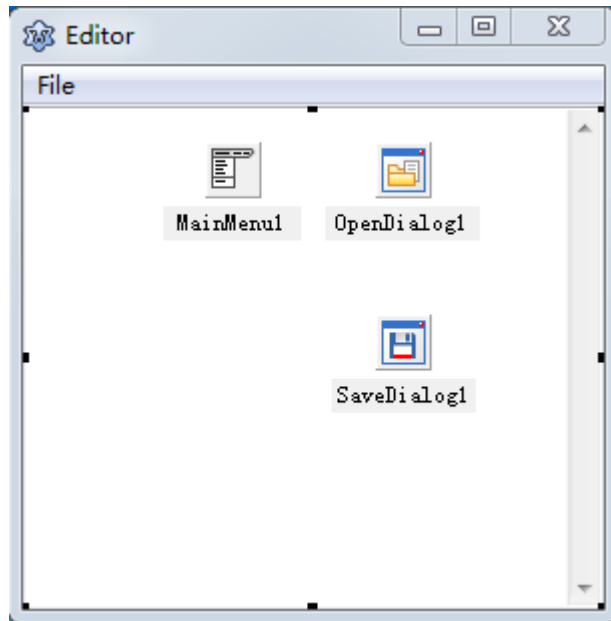
Add 按钮向列表框插入编辑框的文本；**Delete** 按钮删除当前选择的列表项目；Clear 按钮清除列表；Close 按钮关闭应用程序。

## 4.5 文本编辑器程序

在本例中，我们将创建一个简单的文本编辑器程序。程序的编写依照如下步骤：

- 创建应用程序，并在主窗体中放置如下组件：
  - 主菜单 (TMainMenu)
  - 多行文本区 (TMemo)：更改 align 属性为 alClient, ScrollBars 为 ssBoth。
  - 来自组件面板的“对话框” (dialogs) 页面的 TOpenDialog 和 TSaveDialog。
- 双击 MainMenu1 组件，添加 **File** 菜单，以及包含 Open **File**、Save **File**、Close 项的子菜单。

此时，窗体如同这样。



- 对于 **Open File** 项的 OnClick 事件，写这些代码：

```
if OpenDialog1.Execute then  
    Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
```

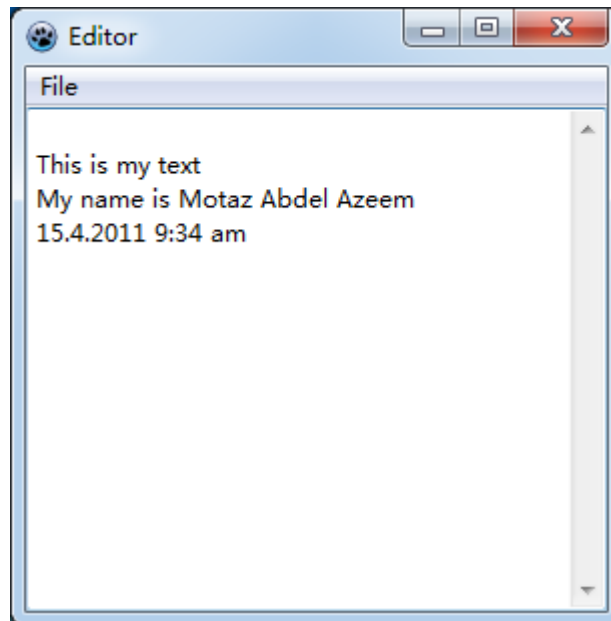
- 对于 **Save File** 项目，写这些代码：

```
if SaveDialog1.Execute then  
    Memo1.Lines.SaveToFile(SaveDialog1.FileName);
```

- 对于 **Close** 项，写：

```
Close;
```

在运行此程序后，我们可以编写文本并保存它到磁盘中，也可以打开已有的文本文件，如下图所示。



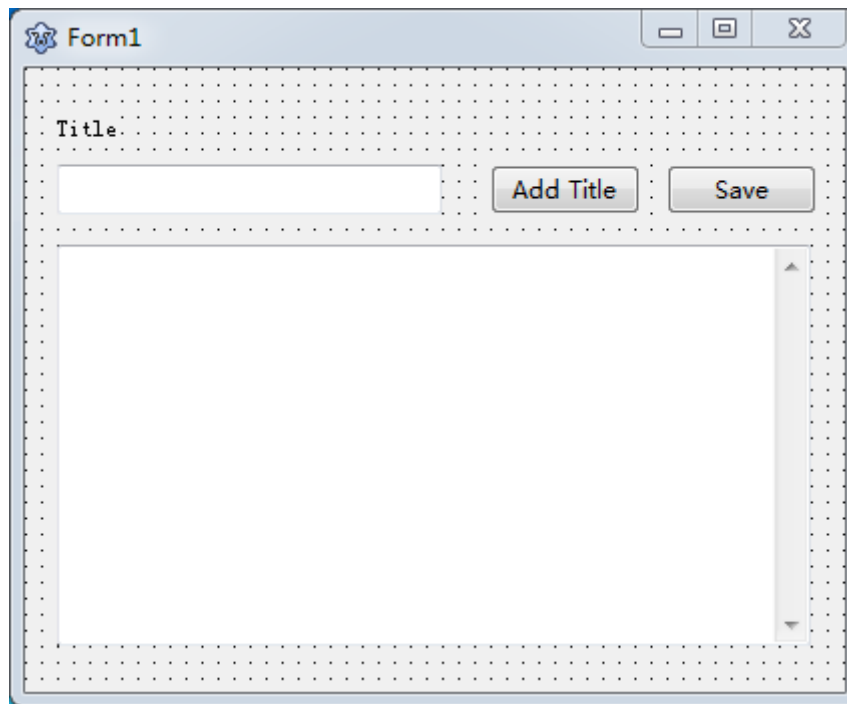
## 4.6 新闻应用程序

本例通过以下步骤完成一个存储新闻标题的程序。

- 创建一个新程序，命名为 gnews
- 添加两个按钮、一个文本框、一个多行文本区。
- 依照下面的值更改组件的属性：

```
[Button1] Caption: Add title  
  
[Button2] Caption: Save  
           Anchors: Left=false, right=true  
  
[Edit1]   Text=  
[Memo1]   ScrollBars: ssBoth  
           ReadOnly: True  
           Anchors: Top=True, Left=True, Right=True, Bottom=True
```

然后，我们将获得这样的窗口。



- 对于 Add Title 按钮的 OnClick 事件，输入：

```
Memo1.Lines.Insert(0,  
    FormatDateTime('yyyy-mm-dd hh:nn', Now) + ': ' + Edit1.Text);
```

- 对于 Save 按钮的 OnClick 事件，输入：

```
Memo1.Lines.SaveToFile('news.txt');
```

- 对于主窗体的 OnClose 事件，输入下列代码来保存输入的新闻：

```
Memo1.Lines.SaveToFile('news.txt');
```

- 对于主窗体的 OnCreate 事件，输入下列代码，来加载先前保存的新闻标题：

```
if FileExists('news.txt') then  
    Memo1.Lines.LoadFromFile('news.txt');
```

## 4.7 带有第二个窗口的应用程序

在先前的 GUI 应用程序中，我们仅使用 1 个窗体，但是在真正的应用程序中，我们有时需要多个窗体。

为写一个多窗体 GUI 应用程序，完成下面的步骤：

1. 创建一个新的应用程序并保存它在一个新的称为 `secondform` 的文件夹中。
2. 保存主单元为 `main.pas`，并命名窗体组件为 `fmMain`。保存该工程为 `secondform.lpi`。
3. 通过点击

文件/新建窗体

添加新窗体。保存此单元为 `second.pas`，并命名它的窗体组件为 `fmSecond`。

4. 在第二个窗体中添加一个标签，并在 `Caption` 属性中写入 `'Second Form'`。通过此标签的 `Font.Size` 属性改变字体大小。
5. 跳回到主窗体并在其上放置一个按钮
6. 在主单元的 **implementation** 部分后添加这行：

```
uses second;
```

7. 对于按钮的 `OnClick` 事件，写这些代码：

```
fmSecond.Show;
```

然后运行应用程序并点击按钮来显示第二个窗体。

# 第五章

## 面向对象编程



## 5.1 介绍

在面向对象编程中，我们使用对象（object）描述应用程序中的实体。例如，我们可以将汽车信息表示为一个含有型号名称、型号年份、价格的对象。对象也有存储为文件中的数据的功能。

对象有如下方面：

1. 属性 以变量存储该对象的状态和信息。
2. 方法 包括过程和函数，表示在该对象中可完成的动作。
3. 事件（events） 可以被对象接收的事件，如鼠标的经过和点击等。
4. 事件处理程序（event handler） 在相应事件发生时被执行的代码。

相互关联的属性和方法可以以一个对象而被代表：

对象 = 代码 + 数据。

一个面向对象编程的示例是已经在上一章使用的图形界面。该章中使用了很多对象，如按钮、标签、窗体。每个对象各自有 `Caption`、`Width` 等属性，`Show`、`Hide`、`Close` 等方法，以及 `OnClick`、`OnCreate`、`OnClose` 等事件。程序员编写的是响应特定的事件的代码，如响应 `OnClick` 事件的程序。

## 5.2 第一个示例：日期和时间

我们在本例中写了一个存储日期和时间，并对它们进行操作的类。

首先，创建一个新单元，称为 `DateTimeUnit`，并在其中创建类，命名为 `TmyDateTime`。

类（class）指对象的类型。在使用一个类时，应声明它的实例，称为对象（object）正如整型或字符串类型。在程序中，则声明类的实例作为变量。

本例中的单元代码如下：

```
unit DateTimeUnit;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;
```

```

type
    { TMyDateTime }

    TMyDateTime = class
    private
        fDateTime: TDateTime;
    public
        function GetDateTime: TDateTime;
        procedure SetDateTime(ADateTime: TDateTime);
        procedure AddDays(Days: Integer);
        procedure AddHours(Hours: Single);
        function GetDateTimeAsString: string;
        function GetTimeAsString: string;
        function GetDateAsString: string;
        constructor Create(ADateTime: TDateTime);
        destructor Destroy; override;
    end;

implementation

{ TMyDateTime }

function TMyDateTime.GetDateTime: TDateTime;
begin
    Result:= fDateTime;
end;

procedure TMyDateTime.SetDateTime(ADateTime: TDateTime);
begin
    fDateTime:= ADateTime;
end;

procedure TMyDateTime.AddDays(Days: Integer);
begin
    fDateTime:= fDateTime + Days;
end;

procedure TMyDateTime.AddHours(Hours: Single);
begin
    fDateTime:= fDateTime + Hours / 24;
end;

function TMyDateTime.GetDateTimeAsString: string;
begin
    Result:= DateTimeToStr(fDateTime);
end;

function TMyDateTime.GetTimeAsString: string;
begin
    Result:= TimeToStr(fDateTime);
end;

```

```

function TMyDateTime.GetDateAsString: string;
begin
    Result:= DateToStr(fDateTime);
end;

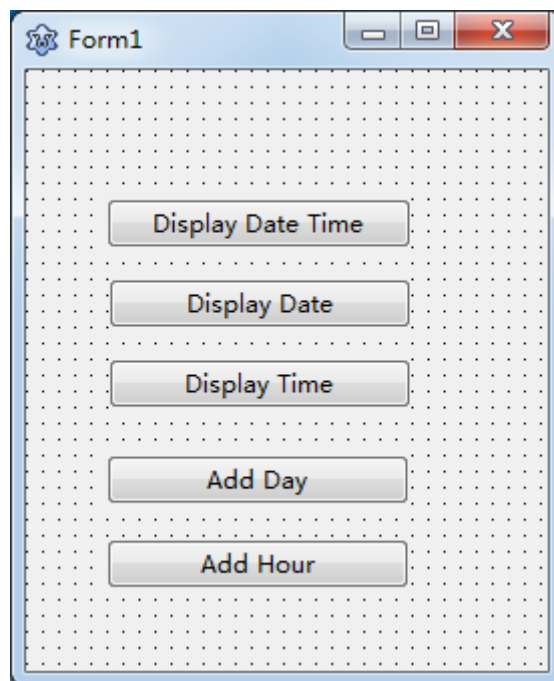
constructor TMyDateTime.Create(ADateTime: TDateTime);
begin
    fDateTime:= ADateTime;
end;

destructor TMyDateTime.Destroy;
begin
    inherited Destroy;
end;

end.

```

之后，如图在窗体上放置 5 个按钮。



在每个按钮的 Onclick 事件下加入如下代码。

```

unit main;

{$mode objfpc}{$H+}

interface

uses
    Classes, SysUtils, FileUtil, LResources, Forms, Controls,
    Graphics, Dialogs, StdCtrls, DateTimeUnit;

```

## type

```
{ TForm1 }
```

```
TForm1 = class(TForm)
```

```
    Button1: TButton;
```

```
    Button2: TButton;
```

```
    Button3: TButton;
```

```
    Button4: TButton;
```

```
    Button5: TButton;
```

```
    procedure Button1Click(Sender: TObject);
```

```
    procedure Button2Click(Sender: TObject);
```

```
    procedure Button3Click(Sender: TObject);
```

```
    procedure Button4Click(Sender: TObject);
```

```
    procedure Button5Click(Sender: TObject);
```

```
    procedure FormCreate(Sender: TObject);
```

```
    private
```

```
        { private declarations }
```

```
    public
```

```
        MyDT: TMyDateTime;
```

```
        { public declarations }
```

```
    end;
```

## var

```
    Form1: TForm1;
```

## implementation

```
{ TForm1 }
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
    MyDT:= TMyDateTime.Create(Now);
```

```
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
```

```
    ShowMessage(MyDT.GetDateTimeAsString);
```

```
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
```

```
begin
```

```
    ShowMessage(MyDT.GetDateAsString);
```

```
end;
```

```
procedure TForm1.Button3Click(Sender: TObject);
```

```
begin
```

```
    ShowMessage(MyDT.GetTimeAsString);
```

```

end;

procedure TForm1.Button4Click(Sender: TObject);
begin
    MyDT.AddHours(1);
end;

procedure TForm1.Button5Click(Sender: TObject);
begin
    MyDT.AddDays(1);
end;

initialization
    {$I main.lrs}

end.

```

本例中有以下要点。

### DateTimeUnit 单元

1. 定义 TMyDateTime 为 **class**——这是定义新的类的关键字。
2. 构造方法 Create: 这是一个特定的过程，用于在内存中创建对象并初始化。
3. 析构方法 Destroy: 这是一个特定的过程，用于处理不再使用的对象的内存。
4. 一个类分为两个部分。私密（private）部分包含不能被外部的类单元访问属性和方法，公开（public）部分包含可以被外部的单元访问属性和方法。如果一个类没有公开部分，则它不能被使用。

### 主单元

1. 在 **Uses** 语句添加 DateTimeUnit，以访问相应的类。
2. 在单元内部声明 MyDT 对象。

```
MyDT: TMyDateTime;
```

3. 在主窗体的 OnCreate 事件中创建对象并初始化。

```
MyDT:= TMyDateTime.Create(Now);
```

这是在 Object Pascal 语言中创建对象的方法。

## 5.3 面向对象式的新闻应用程序

在本例中，我们使用面向对象的思想重写新闻应用程序，并将新闻分类到独立的文件中。

首先，创建一个新的图形界面应用程序，并命名为 oonews。

下例是一个新单元，包含有新闻功能的 TNews 类。

```
unit news;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

type
  TNewsRec = record
    ATime: TDateTime;
    Title: string[100];
  end;

  { TNews }

  TNews = class
  private
    F: file of TNewsRec;
    fFileName: string;
  public
    constructor Create(FileName: string);
    destructor Destroy; override;
    procedure Add(ATitle: string);
    procedure ReadAll(var NewsList: TStringList);
    function Find(Keyword: string;
                  var ResultList: TStringList): Boolean;
  end;

implementation

{ TNews }

constructor TNews.Create(FileName: string);
begin
  fFileName:= FileName;
end;

destructor TNews.Destroy;
begin
  inherited Destroy;
end;
```

```

procedure TNews.Add(ATitle: string);
var
    Rec: TNewsRec;
begin
    AssignFile(F, fFileName);
    if FileExists(fFileName) then
        begin
            FileMode:= 2; // Read/write access
            Reset(F);
            Seek(F, FileSize(F));
        end
    else
        Rewrite(F);

    Rec.ATime:= Now;
    Rec.Title:= ATitle;
    Write(F, Rec);
    CloseFile(F);
end;

procedure TNews.ReadAll(var NewsList: TStringList);
var
    Rec: TNewsRec;
begin
    NewsList.Clear;
    AssignFile(F, fFileName);
    if FileExists(fFileName) then
        begin
            Reset(F);
            while not Eof(F) do
                begin
                    Read(F, Rec);
                    NewsList.Add(DateTimeToStr(Rec.ATime) + ' : ' + Rec.Title);
                end;
            CloseFile(F);
        end;
end;

function TNews.Find(Keyword: string;
                    var ResultList: TStringList): Boolean;
var
    Rec: TNewsRec;
begin
    ResultList.Clear;
    Result:= False;
    AssignFile(F, fFileName);
    if FileExists(fFileName) then
        begin
            Reset(F);
            while not Eof(F) do
                begin

```

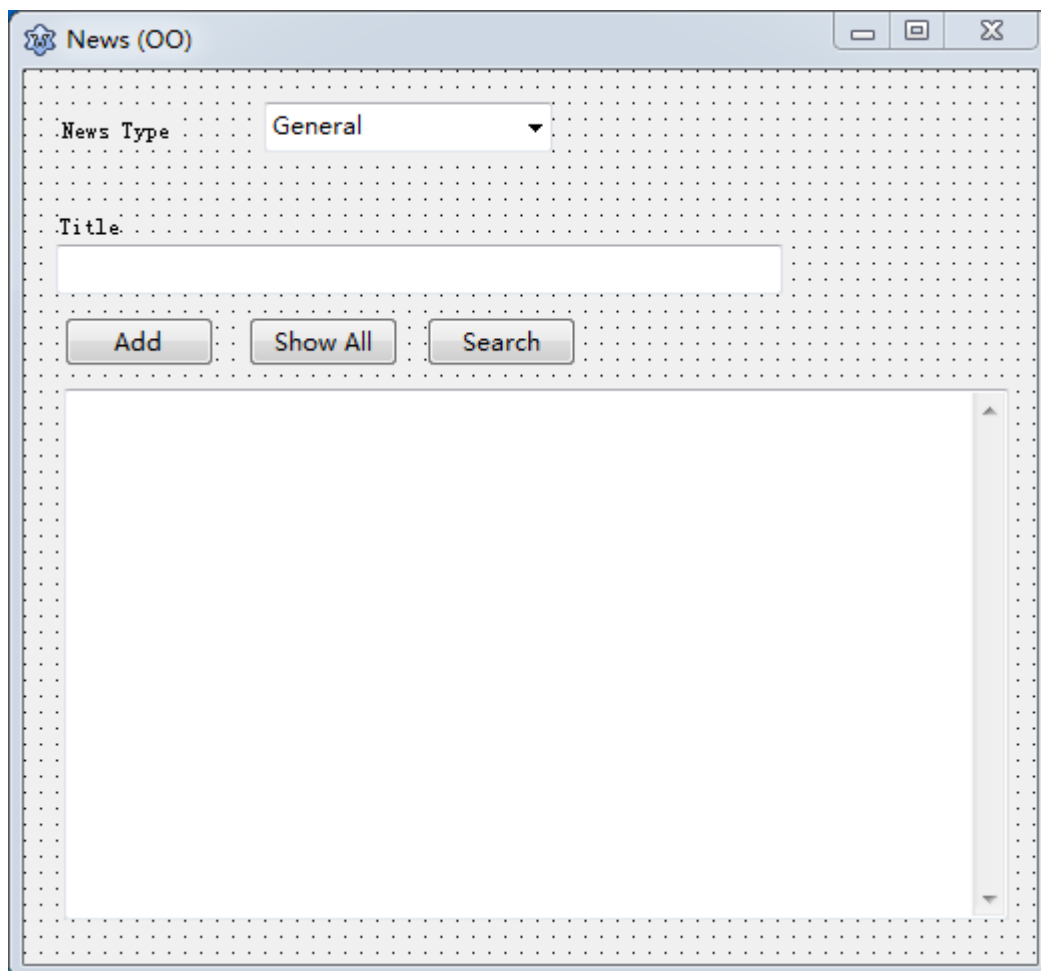
```

Read(F, Rec);

if Pos(LowerCase(Keyword), LowerCase(Rec.Title)) > 0 then
begin
    ResultList.Add(DateTimeToStr(Rec.ATime) + ' : ' + Rec.Title);
    Result:= True;
end;
end;
CloseFile(F);
end;
end;
end.

```

在主窗体中，添加文本框、下拉列表、3 个按钮、多行文本框、2 个标签组件，如图所示。



在主单元中，编写下列代码。

```

unit main;

{$mode objfpc}{$H+}

interface

```



**uses**

Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics,  
Dialogs, News, StdCtrls;

**type**

*{ TForm1 }*

TForm1 = **class**(TForm)

btAdd: TButton;

btShowAll: TButton;

btSearch: TButton;

cbType: TComboBox;

edTitle: TEdit;

Label1: TLabel;

Label2: TLabel;

Mem1: TMemo;

**procedure** btAddClick(Sender: TObject);

**procedure** btSearchClick(Sender: TObject);

**procedure** btShowAllClick(Sender: TObject);

**procedure** FormCreate(Sender: TObject);

**private**

*{ private declarations }*

**public**

NewsObj: **array of** TNews;

*{ public declarations }*

**end;**

**var**

Form1: TForm1;

**implementation**

*{ TForm1 }*

**procedure** TForm1.FormCreate(Sender: TObject);

**var**

i: Integer;

**begin**

SetLength(NewsObj, cbType.Items.Count);

**for** i:= 0 **to** High(NewsObj) **do**

NewsObj[i]:= TNews.Create(cbType.Items[i] + **'.news'**);

**end;**

**procedure** TForm1.btAddClick(Sender: TObject);

**begin**

NewsObj[cbType.ItemIndex].Add(edTitle.Text);

**end;**

```

procedure TForm1.btSearchClick(Sender: TObject);
var
    SearchStr: string;
    ResultList: TStringList;
begin
    ResultList:= TStringList.Create;
    if InputQuery('Search News', 'Please input keyword', SearchStr) then
        if NewsObj[cbType.ItemIndex].Find(SearchStr, ResultList) then
            begin
                Memo1.Lines.Clear;
                Memo1.Lines.Add(cbType.Text + ' News');
                Memo1.Lines.Add('-----');
                Memo1.Lines.Add(ResultList.Text);
            end
        else
            Memo1.Lines.Text:= SearchStr + ' not found in ' +
                cbType.Text + ' news';
            ResultList.Free;
end;

procedure TForm1.btShowAllClick(Sender: TObject);
var
    List: TStringList;
begin
    List:= TStringList.Create;
    NewsObj[cbType.ItemIndex].ReadAll(List);
    Memo1.Lines.Clear;
    Memo1.Lines.Add(cbType.Text + ' News');
    Memo1.Lines.Add('-----');
    Memo1.Lines.Add(List.Text);
    List.Free;
end;

procedure TForm1.FormClose(Sender: TObject;
                                var CloseAction: TCloseAction);
var
    i: Integer;
begin
    for i:= 0 to High(NewsObj) do
        NewsObj[i].Free;

    NewsObj:= nil;
end;

initialization
    {$I main.lrs}

end.

```

本例中有以下要点。

1. 本例中出现了动态数组。这种数组可以在运行时进行分配 (allocation)、扩大 (expansion)、缩小 (reduction) 或丢弃 (disposition)。动态数组以如下方式声明。

```
NewsObj: array of TNews;
```

而在使用该数组之前，应当使用 `SetLength` 过程进行初始化。

```
SetLength(NewsObj, 10);
```

该行表示给该数组分配 10 个元素。这两行的综合效果等于如下的普通数组声明：

```
NewsObj: array [0 .. 9] of TNews;
```

在程序的执行过程中，一般的数组大小固定不变，而动态数组的大小却可以增加、减小。本例中，数组通过下拉框中已有的分类进行初始化。

```
SetLength(NewsObj, cbType.Items.Count);
```

如果我们在下拉框中添加新分类，动态数组的大小也将相应地增加。

2. `TNews` 是一个类，不能直接使用，而必须声明一个对象，如 `NewsObj`，作为实例。
3. 在应用程序的结尾，先释放对象，再释放动态数组。

```
for i:= 0 to High(NewsObj) do  
    NewsObj[i].Free;  
  
NewsObj:= nil;
```

## 5.4 队列

队列（queue）是数据结构类型的一个范例，用于有序地插入、存储元素，并依据插入顺序删除元素。这种规则称为“先进先出”（first in first out，简写 FIFO）。

本例中完成了 Queue 单元，它包含 TQueue 类。TQueue 可以用于存储名称等数据，并按顺序读取。从队列中读取数据时会删除数据：如果队列原本包括 10 项，已经读取了 3 项，则队列中只有 7 项。

Queue 单元的实现如下所示。

```
unit queue;
// 此单元的 TQueue 类适合储存字符串队列

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

type

  { TQueue }

  TQueue = class
  private
    fArray: array of string;
    fTop: Integer;
  public
    constructor Create;
    destructor Destroy; override;
    function Put(AValue: string): Integer;
    function Get(var AValue: string): Boolean;
    function Count: Integer;
    function ReOrganize: Boolean;
  end;

implementation

{ TQueue }

constructor TQueue.create;
begin
  fTop := 0;
end;

destructor TQueue.destroy;
begin
  SetLength(fArray, 0); // 清除队列内容
  inherited destroy;
```

```

end;

function TQueue.Put(AValue: string): Integer;
begin
    if fTop >= 100 then
        ReOrganize;

        SetLength(fArray, Length(fArray) + 1);
        fArray[High(fArray)] := AValue;
        Result := High(fArray) - fTop;
    end;

function TQueue.Get(var AValue: string): Boolean;
begin
    AValue := '';
    if fTop <= High(fArray) then
        begin
            AValue := fArray[fTop];
            Inc(fTop);
            Result := True;
        end
    else // 已经为空
        begin
            Result := False;

            // 清除数组
            SetLength(fArray, 0);
            fTop := 0;
        end;
    end;

function TQueue.Count: Integer;
begin
    Result := Length(fArray) - fTop;
end;

function TQueue.ReOrganize: Boolean;
var
    i: Integer;
    PCount: Integer;
begin
    if fTop > 0 then
        begin
            PCount := Count;
            for i := fTop to High(fArray) do
                fArray[i - fTop] := fArray[i];

            // 去除未使用数据
            setLength(fArray, PCount);
            fTop := 0;
        end;
    end;
end;

```

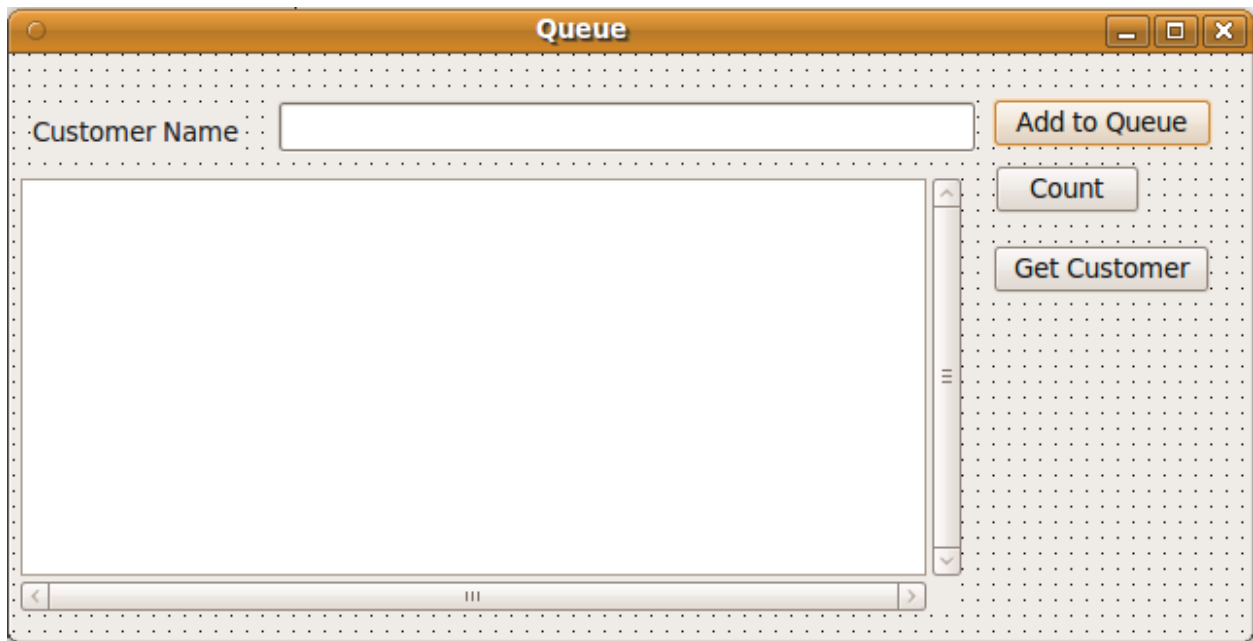
```

    Result:= True; // 已经完成整理
end
else
    Result:= False; // 未整理
end;
end;

end.

```

主窗体如图所示。



主单元代码如下。

```

unit main;

{$mode objfpc}{$H+}

interface

uses
    Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics,
    Dialogs, Queue, StdCtrls;

type
    { TfmMain }

    TfmMain = class(TForm)
        bbAdd: TButton;
        bbCount: TButton;
        bbGet: TButton;
        edCustomer: TEdit;
    end;

```

```

    Label1: TLabel;
    Memo1: TMemo;
    procedure bbAddClick(Sender: TObject);
    procedure bbCountClick(Sender: TObject);
    procedure bbGetClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var CloseAction: TCloseAction);
    procedure FormCreate(Sender: TObject);

    private
        { private declarations }
    public
        MyQueue: TQueue;
        { public declarations }
    end;

var
    fmMain: TfmMain;

implementation

{ TfmMain }

procedure TfmMain.FormCreate(Sender: TObject);
begin
    MyQueue := TQueue.Create;
end;

procedure TfmMain.FormClose(Sender: TObject;
                               var CloseAction: TCloseAction);
begin
    MyQueue.Free;
end;

procedure TfmMain.bbCountClick(Sender: TObject);
begin
    Memo1.Lines.Add('Queue length is: ' + IntToStr(MyQueue.Count));
end;

procedure TfmMain.bbAddClick(Sender: TObject);
var
    APosition: Integer;
begin
    APosition := MyQueue.Put(edCustomer.Text);
    Memo1.Lines.Add(edCustomer.Text + ' has been added as # ' +
                    IntToStr(APosition + 1));
end;

procedure TfmMain.bbGetClick(Sender: TObject);
var
    ACustomer: string;

```

```

begin
  if MyQueue.Get(ACustomer) then
  begin
    Memo1.Lines.Add('Got: ' + ACustomer + ' from the queue');
  end
  else
    Memo1.Lines.Add('Queue is empty');
  end;

initialization
  {$I main.lrs}

end.

```

TQueue 类使用 Put 方法扩大动态数组以插入新项目,并放置在数组最新的元素中。Get 方法移除队列顶部的项目,并将 fTop 下移。

当移除队列中的一项时, fTop 指针进行移动,但项目本身却仍占据内存空间。因此,如果已删除的项目达到 100 项,则会自动调用 ReOrganize 以移动元素的位置,并删除未使用的元素,如下列代码所示。

```

for i:= fTop to High(fArray) do
  fArray[i - fTop]:= fArray[i];

```

以下代码则用于去除未使用的数据。

```

// 去除未使用数据
setLength(fArray, PCount);
fTop:= 0;

```

本例体现了面向对象编程的又一特点:信息隐藏。对敏感数据,如变量本身,的访问将被拒绝。但是,可以使用不会使对象出现奇怪的运转状态的特定方法。

敏感的数据方法位于声明的 **private** (私密) 部分,如下例所示。

```

private
  fArray: array of string;
  fTop: Integer;

```

程序员在使用这个类时不能直接访问这些变量。否则,程序员可能不经意地修改 fTop 或 fArray,造成对队列的破坏。例如,若 fTop 被改为 1000,队列中却仅含有 10 个元素,则会造成访问异常。

在程序中不直接使用变量,而使用 Put、Get 方法,用于在数组中安全地添加和移除项目。这两个方法如同类的出入口,对内部元素进行控制。这个特点称为封装(encapsulation)。



## 5.5 面向对象的文件读写

第一章介绍了两种文件类型，并使用结构化编程思想操作；而本例则使用对象访问文件。

Object Pascal 的文件类之一是 `TFileStream`，它包含操作文件的方法和属性。

对程序员来说，使用这套方法使事情更标准，也更可预见。例如，文件的打开和初始化使用构造函数 `Create`，与其他对象相同；在结构化编程中，则需要 **AssignFile**、**Rewrite**、**Reset**、**Append** 等过程用于初始化文件。

### 5.5.1 使用 `TFileStream` 复制文件

本例中使用 `TFileStream` 类复制文件。

创建新应用程序后，在主窗体上放置按钮、“打开”对话框、“保存”对话框。

对于按钮的 `OnClick` 事件，加入如下代码。

```
procedure TfmMain.Button1Click(Sender: TObject);
var
  SourceF, DestF: TFileStream;
  Buf: array [0 .. 1023] of Byte;
  NumRead: Integer;
begin
  if OpenFileDialog1.Execute and SaveDialog1.Execute then
  begin
    SourceF:= TFileStream.Create(OpenDialog1.FileName, fmOpenRead);
    DestF:= TFileStream.Create(SaveDialog1.FileName, fmCreate);
    while SourceF.Position < SourceF.Size do
    begin
      NumRead:= SourceF.Read(Buf, SizeOf(Buf));
      DestF.Write(Buf, NumRead);
    end;
    SourceF.Free;
    DestF.Free;
    ShowMessage('Copy finished');
  end;
end;
```

此方法与使用无类型文件十分相似。

此外，我们也可以使用另一个更为简单的复制文件方法。

```
procedure TfmMain.Button1Click(Sender: TObject);
var
  SourceF, DestF: TFileStream;
begin
  if OpenFileDialog1.Execute and SaveDialog1.Execute then
  begin
    SourceF:= TFileStream.Create(OpenDialog1.FileName, fmOpenRead);
```

```
    DestF:= TFileStream.Create(SaveDialog1.FileName, fmCreate);  
    DestF.CopyFrom(SourceF, SourceF.Size);  
    SourceF.Free;  
    DestF.Free;  
    ShowMessage('Copy finished');  
end;  
end;
```

## 5.6 类的继承

在面向对象编程中，类的“继承”指从已有的类中创建新的类，并继承其方法和属性。此后，也可以添加新的方法和属性到其中。

以下是一个继承的范例，从已经写出的字符串队列类中创建新的整型 queue 队列类。这样做的好处是不必重新构建类的全部内容。

为实现类的继承，先创建一个新单元 `IntQueue`，再在 **uses** 分句中放置父类（字符串队列类）所在的单元。然后，如下例声明新的整型队列类。

```
TIntQueue = class(TQueue)
```

该类加入了新的方法，`PutInt` 和 `GetInt`。

`IntQueue` 类的完整代码如下。

```
unit IntQueue;

// 此单元包括 TQueue 的子类 TIntQueue。该子类增加了 PutInt、GetInt 方法，
// 用于整型队列

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, Queue;

type

  { TIntQueue }

  TIntQueue = class(TQueue)

  public
    function PutInt(AValue: Integer): Integer;
    function GetInt(var AValue: Integer): Boolean;

  end;

implementation

{ TIntQueue }

function TIntQueue.PutInt(AValue: Integer): Integer;
begin
  Result:= Put(IntToStr(AValue));
end;

function TIntQueue.GetInt(var AValue: Integer): Boolean;
```

```

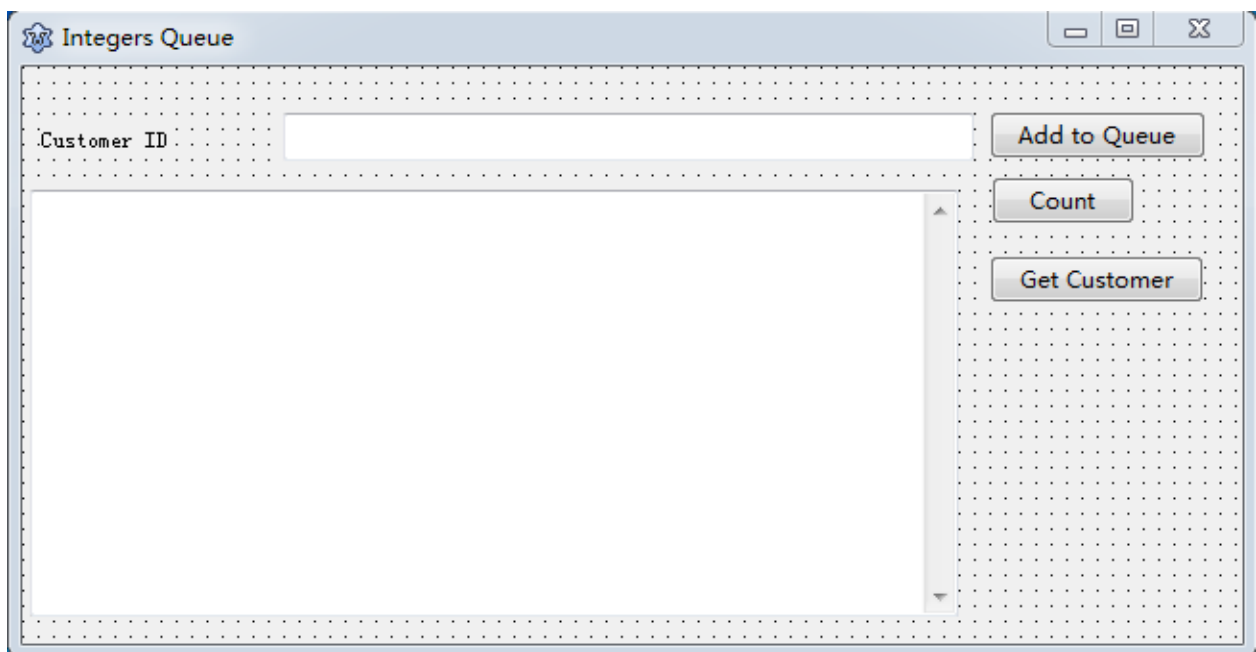
var
  StrValue: string;
begin
  Result:= Get(StrValue);
  if Result then
    AValue:= StrToInt(StrValue);

end;

end.

```

注意：Create、Destroy、Count 三个方法不需要重写，因为在父类中已经存在。  
为使用这个新类，我们创建一个新窗体，并如图加入组件。



主单元的代码如下所示。

```

unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics,
  Dialogs, IntQueue, StdCtrls;

type

  { TfmMain }

  TfmMain = class(TForm)
    bbAdd: TButton;

```

```

    bbCount: TButton;
    bbGet: TButton;
    edCustomerID: TEdit;
    Label1: TLabel;
    Memo1: TMemo;
    procedure bbAddClick(Sender: TObject);
    procedure bbCountClick(Sender: TObject);
    procedure bbGetClick(Sender: TObject);
    procedure FormClose(Sender: TObject;
        var CloseAction: TCloseAction);
    procedure FormCreate(Sender: TObject);
private
    { private declarations }
public
    MyQueue: TIntQueue;
    { public declarations }
end;

var
    fmMain: TfmMain;

implementation

{ TfmMain }

procedure TfmMain.FormCreate(Sender: TObject);
begin
    MyQueue:= TIntQueue.Create;
end;

procedure TfmMain.FormClose(Sender: TObject;
    var CloseAction: TCloseAction);
begin
    MyQueue.Free;
end;

procedure TfmMain.bbCountClick(Sender: TObject);
begin
    Memo1.Lines.Add('Queue length is: ' + IntToStr(MyQueue.Count));
end;

procedure TfmMain.bbAddClick(Sender: TObject);
var
    APosition: Integer;
begin
    APosition:= MyQueue.PutInt(StrToInt(edCustomerID.Text));
    Memo1.Lines.Add(edCustomerID.Text + ' has been added as # '
        + IntToStr(APosition + 1));
end;

```

```

procedure TfmMain.bbGetClick(Sender: TObject);
var
    ACustomerID: Integer;
begin
    if MyQueue.GetInt(ACustomerID) then
        begin
            Memo1.Lines.Add('Got: Customer ID : ' + IntToStr(ACustomerID) +
                            ' from the queue');
        end
    else
        Memo1.Lines.Add('Queue is empty');
    end;

initialization
    {$I main.lrs}

end.

```

注意，程序除使用 TIntQueue 的属性和方法外，也使用了继承的 TQueue 的属性和方法。

我们称原始类 TQueue 为基类，称新类为派生类。

若不创建新类，我们也可以修改字符串队列单元，添加 IntPut 和 IntGet 以处理整型。但是，本例通过 TIntQueue 说明类的继承。对类进行继承还可以有其他原因：例如，在没有 TQueue 的源代码，而仅有编译后的单元文件（Lazarus 中是 .ppu，Delphi 中是 .dcu）。此时，不可能对源代码进行修改。继承将是对类添加更多功能的唯一方法。

## 全 书 完

code.sd  
www.lazaruscn.top