Broadview
www.broadview.com.cn

博文视点原创精品大系

架构设计专家

UML+OOPC
嵌入式C语言开发精讲

高焕堂 著

電子工業出版社

序　言

近年来，C 语言类书籍的销售量扶摇直上，起因于在数码家电、手机、数字化汽车等产业中，嵌入式（Embedded）软件应用愈来愈广。而嵌入式软件开发所使用的语言中，C 语言仍约占 80%多。

由于嵌入式软件应用愈来愈广，软件质量决定了数码产品的稳定性和可靠度，因此，如何提升 C 程序的简洁性、易读性及重复使用性，乃是当今软件业的热门话题。例如，世界知名的麦肯锡（McKinsey）顾问公司，在 2006 年的报告（“Getting better software to manufactured products”）中，呼吁嵌入式软件业必须积极提升其系统分析及架构设计的技术能力，才能解决使用软件愈来愈多的数码产品的信赖度问题。

如何解决上述问题呢？其方向已经很清楚了，就是让 C 语言与面向对象程序设计（Object-Oriented Programming，简称 OOP）技术相结合。就像当今的其它主流计算机语言（如 VB.NET、C#、Java 等）一样。由于当今的世界标准系统分析与架构设计的建模语言——UML，也是基于面向对象技术而发展出来的，因此，一旦 C 语言与面向对象技术相结合了，也就是与 UML 结合了，便能逐渐提升系统分析与设计的质量。因此笔者在出版《精通 ANSI-C 语言》一书之后，继续编写本书，期望陪伴众多 C 程序员能更上层楼，强化系统分析及架构设计的能力，以适应日益热络的嵌入式系统开发市场的需要。

也许你会问：在 1986 年时，贝尔（Bell）实验室已经将 C 语言与面向对象技术结合成为 C++语言了，为何还需要 OOPC 呢？其答案是 C++语言有些贪心，将整套的面向对象技术涵括进去，导致 C++的效率远比单纯 C 语言慢了许多。由于嵌入式软件所能使用的硬件资源大都极为有限，对程序执行效率斤斤计较；所以在今天嵌入式软件开发上，使用最广的仍是 C 语言。

而本书所介绍的面向对象 C 语言并不是一种新的语言，它只运用单纯 C 语言的宏（Macro）技巧，实现了面向对象的基本技术，让系统分析与设计阶段的 UML 模型能与 C 程序紧密对应，以提升 C 程序的质量。此外，这些宏在编译阶段就被翻译为单纯 C 程序代码了，仍然保持其单纯 C 的高效率，符合嵌入式软件环境的需要。本书由浅入深分为 4 篇（共 26 章）：

第 1 篇——复习 ANSI-C 语言。

第 2 篇——学习面向对象技术和 OOPC。详细说明了 ANSI-C 如何与 OOP 技术相结合成为面向对象 C 语言（简称 OOPC）。

第 3 篇——基于面向对象技术而进入 UML 世界。

第 4 篇——活用 UML + OOPC。从实例演练中，运用 UML+OOPC 开发流程，做好系统分析和架构设计，实现高质量的嵌入式软件系统。

第 5 篇——面向对象 Keil C51 语言：在单片机（SOC）的应用。

愿本书能陪伴你驰聘于嵌入式领域中，并鸿图大展。

高焕堂

谨识于 2008 年元月

目　　录

第 1 章　嵌入式时代最划算策略

——替 ANSI-C 黄袍加身，让他飞上枝头变凤凰

1.1 迎接高质量的嵌入式软件时代

——以 0.5 KB 代价换得优雅架构
——基于 Win32 执行平台的测量

OOPC 是“Object-Oriented Programming in C”的简写，而 LW_OOPC 是轻量（Light-weight）级 OOPC 的意思。这是针对嵌入式及数字产品开发而设计的轻薄短小型的 OOPC 语言。只要你的 C 程序允许增加 400B（Byte）的大小，就能运用面向对象机制撰写美好的软件架构。所以，从今天开始，只要包含 lw_oopc.h 头文件，就能开始编写面向对象 C 程序代码了。

而本书所介绍的面向对象 C 语言并不是一个新的语言，它只是单纯地运用 C 语言的宏（Macro）技巧，实现了面向对象的基本技术，让系统分析与设计阶段的 UML 模型能与 C 程序紧密对应，以提升 C 程序的质量。此外，这些宏在预编译阶段就被翻译为单纯 C 程序代码了，它仍然保持单纯 C 的高效率，符合嵌入式软件环境的需要。

嵌入式系统的需求多样，由于硬件资源有限，所以必须斤斤计较、精准计算。如果能搭配上弹性的选择，就能顺畅地决定出最佳的解决方案。俗语说，“天下没有白吃的午餐”。要在 Turbo C、VC/C++、Dev-C/C++等加上美好的面向对象机制，需要付出多少代价呢？以下就以 Windows 平台上的 Turbo C 2.0 和 VC++环境来进行评估。

1.2 基于 Turbo C 2.0 环境的评估

在 Turbo C 2.0 环境里，C 程序加上美好的面向对象机制，需要付出多少代价呢？答案是一个类增加 0.5KB（即 400 Bytes）左右，即使你的程序含有 10 个类（这已经是大系统了），也不过增加 5KB 左右。以下就以 Turbo C 2.0 IDE 的评估来说明这是一项很划算的策略。

1.2.1 加入第 1 个类

在 C 程序里加入第 1 个类时，你的程序大小会增加约 0.5KB。如果继续加入，每增加一个类，会增加约 0.3KB ~ 0.5KB。一般而言，如果系统环境里搭配有 RTOS（Real-Time Operating System），通常不会斤斤计较 500B 的空间，本书就是针对这类环境而写的。现在先测量一个没有类的程序，代码如下所示：

```
/*  CX01-01.C  */
#include "stdio.h"
static void turnOn() {  printf("Light is ON\n");  }
static void turnOff()
{  printf("Light is OFF\n");  }
/*---------------------------------*/
int main()
{
  turnOn();   turnOff();
  getchar();   return 0;
}
```

这是一个传统的 Turbo C 程序，并没有搭配 LW_OOPC 机制，此程序生成一个 CX01-AP1.EXE 的可执行文件，其大小如图 1-1 所示。

图 1-1

这个程序在 Turbo C 2.0 IDE 上编译及连接之后，可执行文件的大小为 10.4KB，我们将以这个范例来作为评估的基准，来算出增加第 1 个类必须付出的代价。现在就来配上 LW_OOPC 并规划一个 Light 类，代码如下所示。

定义 Light 类

```
/*   light.h   */
#include "lw_oopc.h"

CLASS(Light)
{
  void (*turnOn)();
  void (*turnOff)();
};
```

实现 Light 类

```
/* light.c   */
#include "stdio.h"
#include "light.h"
static void turnOn()
{  printf("Light is ON\n");  }
static void turnOff()
{  printf("Light is OFF\n");  }
CTOR(Light)
  FUNCTION_SETTING(turnOn, turnOn)
  FUNCTION_SETTING(turnOff, turnOff)
END_CTOR
```

编写 main()主函数

```
/* tc_ex01.c   */
#include "stdio.h"
#include "lw_oopc.h"
#include "light.h"

extern void* LightNew();
```

```
void main()
{
  Light* light = (Light*)LightNew();
  light->turnOn();
  light->turnOff();
  getchar();
  return;
}
```

建立一个 Turbo C 的 TC_EX01.Prj 文件

```
light.c
tc_ex01.c
```

此程序生成一个 TC_EX01.EXE 可执行文件，其大小如图 1-2 所示。

图 1-2

这个程序在 Turbo C 2.0 IDE 上编译及连接之后，可执行文件的大小为 10.9KB。

与前面 CX01-AP1.EXE 相比，增加了 0.5KB。这是加入第 1 个类（即 Light 类）所付出的代价。当你看到这个 OOPC 程序时，可能对 CLASS、CTOR 等宏很好奇。不过，在本章里，我们只把焦点放在 OOPC 机制对程序大小的影响上，后续各章会对这些宏逐一说明。

1.2.2 加入第 2 个类

为了测量第 2 个类对程序大小的影响，我们把上述范例加以扩充，增加了三个一般函数，先测量此程序的大小，然后将此函数纳入第 2 个类里，最后再测量程序的大小，进行比较。现在就来加入三个函数，代码如下所示。

定义及实现三个函数

```
/* control.c  */
#include "stdio.h"
#include "light.h"
static Light *pl;
void init()
{  pl = (Light*)LightNew();    }
void turnOn()
{  pl->turnOn();  }
void turnOff()
```

```
{  pl->turnOff();  }
```

编写 main()主函数

```
/* cx01-01.c  */
#include "stdio.h"
#include "lw_oopc.h"
extern void init();
extern void turnOn();
extern void turnOff();
void main()
{
  init();
  turnOn();   turnOff();
  getchar();    return;
}
```

建立一个 Turbo C 的 TC_EX02.Prj 文件

```
       light.c
       control.c
       tc_ex02.c
```

此程序生成一个 TC_EX02.EXE 可执行文件，其大小如图 1-3 所示。

图 1-3

此程序大小为 11.2KB，比上一个程序增加了 0.3KB，这不是由增加类而引起的，而是由增加 init()等三个函数所引起的。我们先以此程序大小为准，然后将这三个函数纳入类里，再测量其大小，就可以看出新增类得付出多少代价了。现在就将其纳入新类（称为 CTRL）里，代码如下所示。

定义 CTRL 类

```
/*  ctrl.h   */
#include "lw_oopc.h"
#include "light.h"

CLASS(CTRL)
{
  void (*init)(CTRL*);
  void (*turnOn)(CTRL*);
  void (*turnOff)(CTRL*);
```

```
  Light *pl;
};
```

实现 CTRL 类

```
/* ctrl.c   */
#include "stdio.h"
#include "ctrl.h"

extern void* LightNew();
static void init(CTRL *t)
{  t->pl = (Light*)LightNew();    }
static void turnOn(CTRL *t)
{  t->pl->turnOn(t->pl);  }
static void turnOff(CTRL *t)
{  t->pl->turnOff(t->pl);  }

CTOR(CTRL)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(turnOn, turnOn)
  FUNCTION_SETTING(turnOff, turnOff)
END_CTOR
```

编写 main()主函数

```
/* tc_ex02.c   */
#include "stdio.h"
#include "lw_oopc.h"
#include "ctrl.h"

extern void* CTRLNew();
void main()
{
  CTRL* ctrl = (CTRL*)CTRLNew();
  ctrl->init(ctrl);
  ctrl->turnOn(ctrl);
  ctrl->turnOff(ctrl);
  getchar();    return;
}
```


建立一个 Turbo C 的 TC_EX03.Prj 文件

```
       light.c
       ctrl.c
       tc_ex03.c
```

此程序生成一个 TC_EX03.EXE 可执行文件，其大小如图 1-4 所示。

图 1-4

此程序大小为 11.9KB，与前面 TC_EX02.EXE 相比，增加了 0.7KB。这是加入第 2 个类（即 CTRL 类）的代价。

1.2.3 加入第 3 个类

为了测量第 3 个类对程序大小的影响，我们把上述范例加以扩充，加了两个函数，先测量此程序的大小；之后将此函数纳入第 3 个类里，再测量程序的大小，进行比较。现在就来加入两个函数，代码如下所示。

定义及实现两个函数

```
/* sw.c   */
#include "stdio.h"
void swDown()
{  printf("SW is Down\n");  }
void swUp()
{  printf("SW is Up\n");  }
```

撰写 main()主函数

```
/* tc_ex04.c   */
#include "stdio.h"
#include "lw_oopc.h"
#include "ctrl.h"
extern void swDown();
extern void swUp();
void main()
{
  CTRL* ctrl = (CTRL*)CTRLNew();
  ctrl->init(ctrl);
  ctrl->turnOn(ctrl);    swDown();
  ctrl->turnOff(ctrl);    swUp();
  getchar();
  return;
}
```

建立一个 Turbo C 的 TC_EX04.Prj 文件

```
       light.c
       ctrl.c
       sw.c
       tc_ex04.c
```

此程序产生一个 TC_EX04.EXE 执行文件，其大小如图 1-5 所示。

图 1-5

此程序大小为 12.0KB。这比上一个程序增加了 0.1KB，这其实不是由增加类而引起的，而是由增加 swDown()等函数所影响的。我们先以此程序大小为准，然后将这两个函数纳入类里，再测量其大小，就可以看出新增类得付出多少代价了。现在就将其纳入新类（称为 DoorSwitch）里，代码如下所示。

定义 DoorSwitch 类

```
/*   DoorSwitch.h   */
#include "lw_oopc.h"

CLASS(DoorSwitch)
{
  void (*swDown)();
  void (*swUp)();
};
```

实现 DoorSwitch 类

```
/* DoorSwitch.c   */
#include "stdio.h"
#include "switch.h"

static void swDown()
{  printf("SW is Down\n");  }
static void swUp()
{  printf("SW is Up\n");  }
CTOR(DoorSwitch)
  FUNCTION_SETTING(swDown, swDown)
  FUNCTION_SETTING(swUp, swUp)
END_CTOR
```

撰写 main()主函数

```
/* tc_ex05.c   */
#include "stdio.h"
#include "lw_oopc.h"
#include "ctrl.h"
#include "switch.h"

void main()
{
  CTRL* ctrl = CTRLNew();
  DoorSwitch *psw = DoorSwitchNew();
  ctrl->init(ctrl);
```

```
  ctrl->turnOn(ctrl);
  psw->swDown();
  ctrl->turnOff(ctrl);
  78psw->swUp();
  getchar();    return;
}
```

建立一个 Turbo C 的 TC_EX05.Prj 文件

```
light.c
ctrl.c
switch.c
tc_ex05.c
```

此程序产生一个 TC_EX05.EXE 执行文件，如图 1-6 所示。

此程序大小为 12.4KB。与前面 TC_EX04.EXE 相比，增加了 0.4KB。这是加入第 3 个类（即 DoorSwitch 类）的代价。

图 1-6

1.2.4 加入第 4 个类

为了测量第 4 个类对程序大小的影响，我们把上述范例加以扩充，加了两个函数，先测量此程序的大小；之后将此函数纳入第 4 个类里，再测量程序的大小，进行比较。现在就来加入两个函数，代码如下所示。

撰写两个新函数及 main()主函数

```
/* tc_ex06.c   */
#include "stdio.h"
#include "lw_oopc.h"
#include "ctrl.h"
#include "switch.h"

double length, width;
void init()
{ length = 10.5;   width = 5.125;  }
void print_perimeter()
{  printf("P = %f\n", 2*(length + width));  }
```

```
void main()
{
  CTRL* ctrl = (CTRL*)CTRLNew();
  DoorSwitch *psw = (DoorSwitch*)DoorSwitchNew();
  ctrl->init(ctrl);
  ctrl->turnOn(ctrl);
  psw->swDown();
  ctrl->turnOff(ctrl);
  psw->swUp();
  init();     print_perimeter();
  getchar();  return;
}
```

建立一个 Turbo C 的 TC_EX06.Prj 文件

```
light.c
ctrl.c
switch.c
tc_ex06.c
```

此程序产生一个 TC_EX06.EXE 执行文件，如图 1-7 所示。

图 1-7

此程序大小为 26.5KB。这比上一个程序增加了 14.1KB，这不是由增加类而引起的，而是由增加 print_perimeter()等函数所影响的。我们先以此程序大小为准，然后将这两个函数纳入类里，再测量其大小，就可以看出新增类得付出多少代价了。现在就将其纳入新类（称为 Rectangle）里，代码如下所示。

撰写 Rectangle 新类及 main()主函数

```
/* tc_ex07.c   */
#include "stdio.h"
#include "lw_oopc.h"
#include "ctrl.h"
#include "switch.h"

CLASS(Rectangle)
{
  void (*init)(Rectangle*);
  void (*print_perimeter)(Rectangle*);
  double length, width;
};
static void init(Rectangle *t)
```

```
{
  t->length = 10.5;  t->width = 5.125;
}
static void pr_perimeter(Rectangle *t)
{
  printf("P = %f\n", 2*(t->length + t->width));
}
CTOR(Rectangle)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(print_perimeter, pr_perimeter)
END_CTOR
/* ------------------------------------------------------------------------ */
void main()
{
  CTRL* ctrl = (CTRL*)CTRLNew();
  DoorSwitch *psw = (DoorSwitch*)DoorSwitchNew();
  Rectangle *rect  = (Rectangle*)RectangleNew();
  ctrl->init(ctrl);
  ctrl->turnOn(ctrl);
  psw->swDown();
  ctrl->turnOff(ctrl);
  psw->swUp();
  rect->init(rect);
  rect->print_perimeter(rect);
  getchar();  return;
}
```

建立一个 Turbo C 的 TC_EX07.Prj 文件

```
light.c
ctrl.c
switch.c
tc_ex07.c
```

此程序产生一个 TC_EX07.EXE 执行文件，如图 1-8 所示。

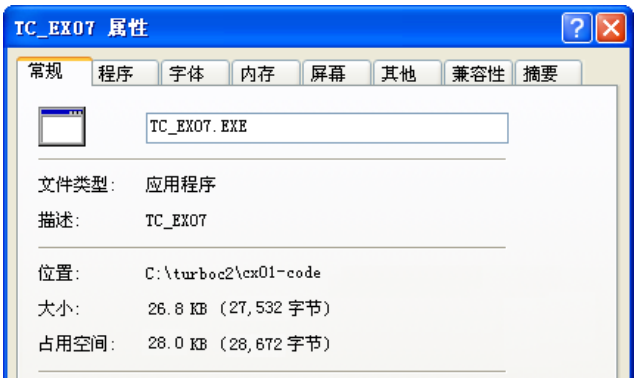


图 1-8

此程序大小为 26.8KB。与前面 TC_EX06.EXE 相比，增加了 0.3KB。这是加入第 4 个类（即 Rectangle 类）的代价。

1.2.5 评估统计图

以上总共加入了四个类，如果你继续加入更多类时，会发现每一个类大约都增加 0.3KB~0.5KB 左右。如图 1-9 所示。

综合上述的评估，兹说明如下：

- 由于 LW_OOPC 是由 C 语言的宏（Macro）所建立的，在编译之前就已经全部被转为一般非对象化的 C 程序了。所以，类能与类外的一般函数并存于你的程序里。这意味着，你可以视你的内存资源的宽裕情形而决定写入多少个类，并非要全部类化（即对象化）不可。因此，程序大小与执行速度并不是决定要不要采用 LW_OOPC 类机制的关键因素。反而，如何藉由 LW_OOPC 改善嵌入式程序架构，以提升系统的稳定性和可靠性才是采用 LW_OOPC 的最主要诱因。因为程序员能自由地决定程序的形式，一方面满足硬件资源的限制，另一方面又能追求最顶级的美好结构。因此，对一个 C 程序员及其撰写的嵌入式软件而言，LW_OOPC 只是给他（或它）黄袍加身，并不会伤害到龙体本身。
- 从图 1-9 中可以看到，如果你的 TurboC 程序使用 LW_OOPC 提供的机制，而且写了 n 个类时，其程序大小的增加量大约为 n * 0.5KB。如果写了 10 个精简型的类（即 n 的值为 10），大约需付出 5KB 的代价。一般而言，C 程序能写到 10 个类已经算是复杂的了，只付出不到 5KB 代价，却能让复杂的程序得到美好结构和高度稳定性，真是非常划算的策略。

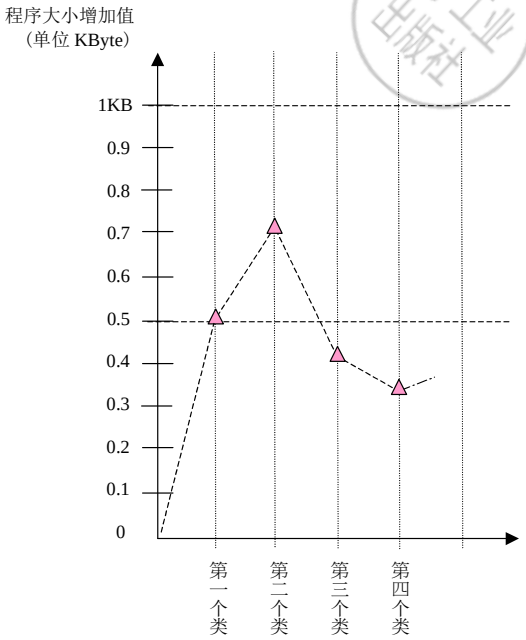


图 1-9

1.3 基于 Visual C++环境的评估

在 Visual C++ 环境里的 C 程序加上美好的面向对象机制，需要付出多少代价呢？答案是平均一个类增加 0.25KB（即 250 Bytes）左右，即使你的程序含有 10 个类（这已经是大系统了），也不过增加 2.5KB 左右。我们来看一个例子，如下代码所示。

```
/*  Win32_EX01_01.c   这是一个 ANSI-C 程序  */
#include "stdio.h"
double radius;
double length, width;

void init_circle()
{ radius = 10.5; }
void pr_area_circle()
{ printf("AC = %f\n", 3.14 * radius * radius); }
void pr_perimeter_circle()
{ printf("PC = %f\n", 2 * 3.14 * radius);  }
/* ------------------------------------------------- */
void init_rect()
{ length = 10.5; width = 5.125; }
void pr_area_rect()
{ printf("AR = %f\n", length * width); }
void pr_perimeter_rect()
{ printf("PR = %f\n", 2 * (length + width)); }

extern void* LightNew();
void main()
{
  init_circle();
  pr_area_circle();
  pr_perimeter_circle();
  init_rect();
  pr_area_rect();
  pr_perimeter_rect();
  getchar();
  return;
}
```

此程序产生一个 Win32_EX01_01.exe 执行文件，其大小如图 1-10 所示。

图 1-10

这个程序在 Visual C++ IDE 上编译及连接之后，执行文件的大小为 28.0KB，我们将以这个范例作为评估的基准，来算出增加两个类所必须付出的代价。现在就来配上 LW_OOPC 并规划两个类，其程序代码如下：

```
// Win32_EX01_01.c
#include "stdio.h"
#include "lw_oopc.h"
/* ---------------------------------- */
INTERFACE(IA)
{


   void (*init)(void*);
   void (*print_area)(void*);
};

CLASS(Circle)
{
  IMPLEMENTS(IA);
  void (*print_perimeter)(Circle*);
  double radius;
};
static void init_circle(Circle *t)
{
  t->radius = 10.5;
}
static void pr_area_circle(Circle *t)
{
  printf("AC = %f\n", 3.14 * t->radius * t->radius);
}
static void pr_perimeter_circle(Circle *t)
{
  printf("PC = %f\n", 2 * 3.14 * t->radius);
}
CTOR(Circle)
  FUNCTION_SETTING(IA.init, init_circle)
  FUNCTION_SETTING(IA.print_area, pr_area_circle)
  FUNCTION_SETTING(print_perimeter, pr_perimeter_circle)
END_CTOR

/* ---------------------------------- */
CLASS(Rectangle)
{
  IMPLEMENTS(IA);
  void (*print_perimeter)(Rectangle*);
  double length, width;
};
static void init_rect(Rectangle *t)
{
  t->length = 10.5;  t->width = 5.125;
}
static void pr_area_rect(Rectangle *t)
{
  printf("AR = %f\n", t->length * t->width);
}
```

```
static void pr_perimeter_rect(Rectangle *t)
{
  printf("P = %f\n", 2*(t->length + t->width));
}
CTOR(Rectangle)
  FUNCTION_SETTING(IA.init, init_rect)
  FUNCTION_SETTING(IA.print_area, pr_area_rect)
  FUNCTION_SETTING(print_perimeter, pr_perimeter_rect)
END_CTOR
/* ------------------------------------------------ */

extern void* LightNew();
void main()
{
  IA *pc = (IA*)CircleNew();
  IA *pr = (IA*)RectangleNew();
  pc->init(pc);
  pc->print_area(pc);
  pr->init(pr);
  pr->print_area(pr);
  getchar();    return;
}
```

此程序产生一个 Win32_EX01_01.exe 执行文件，其大小如图 1-11 所示。

图 1-11

此程序大小为 28.5KB，因为加入两个类而增加了 0.5KB，平均每个类增加 0.25KB。

1.4 LW_OOPC 与 C++ 的比较和评估

既然使用 Visual C++环境，顺便跟 C++程序比较一番。现在将刚才的 LW_OOPC 程序改写为 C++程序，程序代码如下：

```
// Win32_EX01_02.cpp   这是 C++程序
#include "stdafx.h"
class IA
{
public:
```

```
  virtual void init()=0;
  virtual void print_area()=0;
};
//-----------------------------------------------
class Circle: public IA {
private:
   virtual void init();
   virtual void print_area();
   void print_perimeter();
   double radius;
};

void Circle::init()
{ radius = 10.5; }
void Circle::print_area()
{ printf("AC = %f\n", 3.14 * radius * radius); }
void Circle::print_perimeter()
{ printf("PC = %f\n", 2 * 3.14 * radius);  }
//-------------------------------------------------------------------------
class Rectangle: public IA
{
private:
   virtual void init();
   virtual void print_area();
   void print_perimeter();
   double length, width;
}

void Rectangle::init()
{ length = 10.5;  width = 5.125; }
void Rectangle::print_area()
{ printf("AR = %f\n", length * width); }
void Rectangle::print_perimeter()
{ printf("P = %f\n", 2*(length + width));  }
//-----------------------------------------------------------
int _tmain(int argc, _TCHAR* argv[])
{
  IA *pc =  new Circle();
  pc->init();
  pc->print_area();
  IA *pr = new Rectangle();
  pr->init();
  pr->print_area();
  getchar();
  return 0;
}
```

此程序产生一个 Win32_EX01_02.EXE 执行文件，其大小如图 1-12 所示。

图 1-12

此程序大小为 30.5KB。C++版程序比 LW_OOPC 版增加了 2KB。

以上的评估说明了面向对象 ANSI-C 是一项很划算的策略。

第 2 章　C 程序的基本组成

2.1 认识 C 语言与 LW_OOPC 语言

2.1.1 C 语言的身世背景

20 世纪 70 年代，D. Richie 及 K. Thompson 任职于美国贝尔实验室（Bell Labs）。当时 Thompson 正在设计 Unix 操作系统，而 Richie 负责设计新语言来编写 Unix 程序。新语言继承了 Thompson 原来设计的 B 语言，因此，称之为 C 语言。后来，Unix 平步青云，成为标准操作系统，C 随即水涨船高，受人青睐。

Unix 以前的操作系统，皆以“汇编语言”（Assembly Language）编写。C 问世后，逐渐取代了汇编语言的角色。Unix、DOS、Windows、OS/2 等操作系统皆以 C 写成。到了 2008 年的今天，许多软件系统，如 Google 推出的 Android 手机平台，其内核模块仍是 C 语言的杰作呢!

C 语言能获得极大的成功，D. Richie 认为有五项主要因素：

1. C 语言既不够优雅，也不够正式；
2. C 语言符合程序员的需要，并且到处可见支持 C 语言的程序开发环境；
3. 众多人持续加以修改，与大环境保持交互；
4. 一开始把焦点设定在操作系统的开发是相当正确的，所谓好的开始是成功的一半；
5. 运气好。

2.1.2 C 影响 C++、Java、C#等语言文化

Java 是由 SUN 公司的 James Gosling 率领的小组开发出来的。本来该小组的任务是要开发可以控制小家电、交互电视等的编程语言，之后因缘际会，与当时逐渐流行的 Internet 技术结合，在 1995 年发布并取名为 Java。

Gosling 在学生时代就深受 C 语言的洗礼，他在 C.M.U.攻读博士学位的时候，曾经开发了一个 C 语言版本的 Emacs，因而在 Unix 领域享有盛名。

Java 从原本定位为家电控制语言，到后来成为 Internet 上热门的编程语言，在其发展过程中，从 C++那里借鉴了许多宝贵的经验。例如效仿 C++的标准程序库，推出一套完善的 Java 标准程序库；另外加上网络应用，以及最棘手的安全机制。至今，Java 仍是 Internet 应用上最流行的编程语言（见图 2-1）。

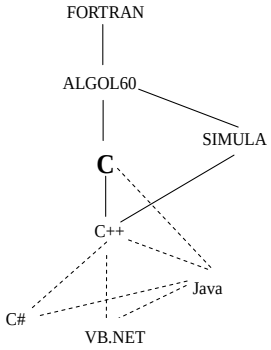


图 2-1

到了 2000 年以后，Microsoft 又从 C++派生出 C#作为与 Java 分庭抗礼的编程语言。之后连 VB.NET 都沿袭了 C++的编译引擎设计，从而大幅提升了 VB.NET 的执行效率，并提供完整的 OOP 功能。

2.1.3 用 C 语言编写面向对象（Object-Oriented）程序

何谓 OOPC，OOPC 是指 OOP（Object-Oriented Programming）与 C 语言结合，藉由 C 语言的 Macro 指令定义出 OOP 概念的关键字（Key Word）。这样，C 程序员就能运用这些关键字来表达 OOP 的概念，如类、对象、信息、继承、接口等。

虽然 OOPC 程序的语法不像 C++那么简洁，但是 OOPC 也有其亮丽的特色，就是编译后的程序所占的内存空间（Size）比 C++程序小，能够满足像 Embedded System 等的内存限制，程序员也能有效地调整程序的瓶颈，从而提升其执行效率。

C++的功能齐全，但是在一般的嵌入式系统（Embedded System）开发上，经常只用到其中的一小部分功能，而不需要用到全部的机制，例如多重继承（Multiple Inheritance）、运算符的重载（Overloading）等。此时，许多嵌入式系统的开发者就舍弃 C++的庞大身躯而回归到精简的 C 环境里。只是开发者舍弃 C++的同时也舍弃了珍贵的 OOP 能力，实在太可惜了。

OOPC 就是要弥补这个缺憾的。它是以 C 的宏写成的 Header 文件，可以任由 C 程序员对它瘦身美容，删去不需要的部分，挑出自己所需要的 OOP 特性，以小而美的身材满足嵌入式系统开发的需要。

以 LW_OOPC 为例

LW_OOPC 是一种既轻便又快速的面向对象的 C 语言。做嵌入式开发的程序员还是比较青睐 C 语言的，只是 C 语言没有对象、类等概念，程序很容易变成意大利面型的结构，维护上比较费力。在 1986 年 C++上市时，开发者希望大家改用 C++，但是 C++的效

率不如 C，因而不受嵌入式系统开发的程序员的喜爱。于是，MISOO 团队便设计了一个既轻便又高效的 OOPC 语言。轻便的意思是它只用了约 20 个 C 宏语句而已，简单易学。其宏如下：

```
/*  lw_oopc.h */  /* 这就是 MISOO 团队所设计的 C 宏 */
#include "malloc.h"
#ifndef LOOPC_H
#define LOOPC_H

#define CLASS(type)\
typedef struct type type; \
struct type

#define CTOR(type) \
void* type##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));
}
#define CTOR2(type, type2) \
void* type2##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));
}
#define END_CTOR return (void*)t;
#define FUNCTION_SETTING(f1, f2)  t->f1 = f2;
#define IMPLEMENTS(type) struct type type
#define INTERFACE(type) struct type
#endif
/*    end    */
```

高效的意思是它没提供类继承，内部没有虚函数表（Virtual Function Table），所以仍保持着原来 C 语言的高效率。除了没有继承机制之外，它提供了类、对象、信息传递、接口和接口多态等常用的机制。目前受到不少 C 程序员的喜爱。

2.1.4 面向对象概念让 UML 与 C 携手合作

简单示例：从 UML 到 OOPC 程序

UML 来自 OO（Object-Oriented）领域，通过 OO 概念，善加运用 UML，能让 C 程序员更上一层楼。由于 UML 也能创建硬件系统的模型，愈来愈多的嵌入式系统公司藉由 UML 来表达软件与硬件的分析与设计。

由于一般 C 语言并没有使用 OO 概念，因此将 UML 设计图实现为 C 程序的过程让人感到不顺畅。所以，MISOO 团队将 OO 概念添加到 C 语言，让人们既能以 OO 概念去分析和设计，也能以 OO 概念去编写 C 程序。这样一来，从分析、设计到程序编写的过程就非常直截了当了。如下述的步骤。

Step-1 分析出一个类叫 Light，它提供两项服务，以 UML 表达（见图 2-2）。

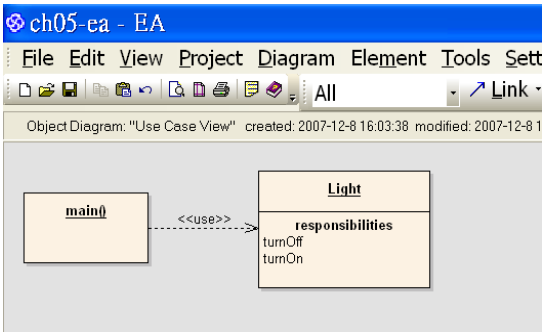


图 2-2

Step-2 实现为 LW_OOPC 程序。

基于上述的 lw_oopc.h 就可以定义出类了，例如定义一个 Light 类，其 light.h 的内容为：

```
/*   light.h   */
#include "lw_oopc.h"

CLASS(Light)
{
  void (*turnOn)();
  void (*turnOff)();
};
```

类里的函数定义格式为：

返回值的类型 (*函数名称)();

类定义好了，就开始编写函数的实现内容，代码如下：

```
/* light.c   */
#include "stdio.h"
#include "light.h"

static void turnOn()
{ printf("Light is ON\n");  }
static void turnOff()
{ printf("Light is OFF\n");  }

CTOR(Light)
  FUNCTION_SETTING(turnOn, turnOn)
  FUNCTION_SETTING(turnOff, turnOff)
END_CTOR
```

这个 FUNCTION_SETTING（turnOn, turnOn）宏的用意是：让类定义（.h 文件）函数名称能够与实现的函数名称不同。例如在 light.c 里可写为：

```
      static void TurnLightOn()
      { ….. }

      CTOR(Light)
```

```
    {
      FUNCTION_SETTING(turnOn, TurnLightOn);
      …..
    }
```

这是创造.c 文件自由替换的空间，是实现接口的重要基础。最后看看如何编写主程序：

```
/* tc_ex01.c   */
#include "stdio.h"
#include "lw_oopc.h"
#include "light.h"

extern void* LightNew();
void main()
{
  Light* light = (Light*)LightNew();
  light->turnOn();
  light->turnOff();
  getchar();
  return;
}
```

LightNew()是由 CTOR 宏所生成的类构造器（Constructor）。由于它定义于别的文件，所以必须加上 extern void* LightNew();指令。生成对象的基本格式为：

```
    类名称*  对象指针 = (类名称*)类名称New();
    示例: Light* light = (Light*)LightNew()
```

然后就能通过对象指针去调用成员函数了。

使用 Turbo C 2.0 开发环境

创建一个项目文件名叫 PRJ_EX01.PRJ，并输入内容，如图 2-3 所示。

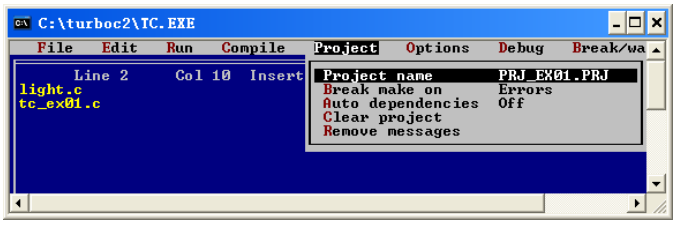


图 2-3

按 ALT+R 执行，则输出如图 2-4 所示的执行结果。

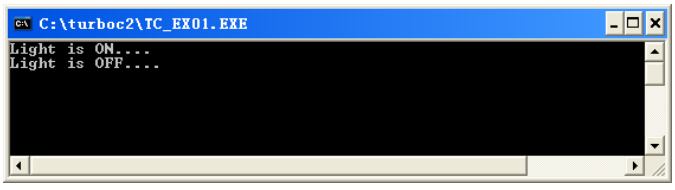


图 2-4

2.2 函数：C 程序的基本结构

2.2.1 指令、函数与程序

C 程序（Program）的基本结构是函数（Function），就是数学中的函数，如 sin()、cos()等。就像“砖块”是建造房子的基本模块（Building Block）一样，“函数”是构成 C 程序的基本模块。在数学课本里，常用图 2-5 表示函数的意义。

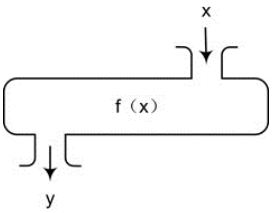


图 2-5

f 是名称，f()代表函数，括号内的 x 表示参数，y 表示函数值。假设 f(x)= x * x，则 y = f(5)= 5 * 5 = 25；此时 y 值是 25。函数的参数摆在括号内，其个数依函数的目的而定。例如：n(a, b)含两个参数 a, b。如果 n(a, b)= a + b，则 n(2, 6)= 2 + 6 = 8 = y。因为 n 之后有()，表示 n 为函数名字；a 和 b 摆在()内，表示 a 和 b 为参数。

2.2.2 函数间的调用（Call）

一方面，函数包含许多指令（Instruction）；另一方面，多个函数又可以组成一个程序。函数之间互相调用与沟通，协力达成程序的目的和任务。如同砖块借水泥来联结并堆成房子一样，函数之间则借“调用”（Call）来组成程序。

图 2-6 中的 main()将参数 x 传给 f()，f() 则传回其函数值 y，这就是 main()调用 f()。main()为主函数（Main-function），f() 为子函数（Sub-function）。另外，图 2-6 中的 main()将参数 a 和 b 传给 n()，而 n() 则传回函数值 y 给 main()，所以 n() 为 main()的子函数。C 程序为一个家族，其内的函数为家族的成员，main()为家族中的家长。拜访一个家族，常先拜访家长；同样，当执行 C 程序时，先执行 main()。

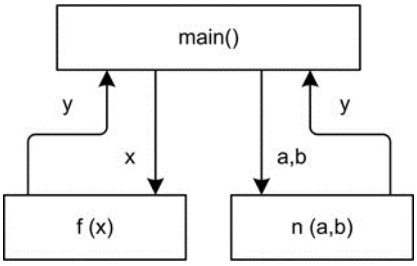


图 2-6

2.2.3 库函数

数学中的函数常表现为数学表达式；程序中的函数则是一段小程序，担任某项明确的任务，例如求算圆的面积、利息、平均成绩等。C 中有些现成函数，例如 sin()及 cos()等，您的程序可以调用它们做事，这种函数称为“库函数”（Library Function）。

然而，实际应用中库函数仍然不够，所以当编写程序时，经常需要创建新函数。它们之间可互相调用，也可以调用现成的库函数。例如：

```
#include <stdio.h>
void f1()
{  printf( "Hello" );   }
int main(void)
{
   f1();
   return 0;
}
```

其中，main()和 f1()是我们自创的函数，printf()是库函数。当 f1()调用 printf()时，传送字符串 "Hello" 给它。main()和 f1()函数各代表一段小程序，“{”表示函数的开始，而“}”表示函数的结束。

2.3 变量的概念

2.3.1 数据分类与变量

在计算机中，数据可分为数字数据和文字数据。其区别是数字数据表示事物的高低、大小、轻重等，且常做加减乘除等数学运算；而文字数据不做数学运算。例如，电话号码 5711438 是文字数据，原因是电话号码相加并无意义。

字符串（Character String）是最常见的文字数据，顾名思义，字符串，就是一连串的字符。键盘上的键就是最常见的字符。在程序中，常以双引号（" "）括住一串字符就是字符串了。例如：H、e、l、l 及 o，共 5 个字符组成字符串"Hello"。

数字数据分为整数（Integer）和浮点数（Floating-point）两大类。整数不含小数部分。例如：报摊已卖出 15 份报纸，其中 15 是整数。浮点数含有小数部分。例如，有一条鱼，质量为 1.25 千克，其中 1.25 是浮点数。

上述的字符串"Hello"、整数 15，以及浮点数 1.25 皆为常数（Constant）。由于它们代表着固定不变的值，因此称之为常数。常数的反义词是变量（Variable）。在数学上，变量代表的值常随运算而变化；在程序中，变量的值也常随程序的执行与运算而变化。譬如，sogo 是百货公司的名字，在程序中可用 sogo 来代表公司的每日销售金额，因此 sogo 所代表的值会天天变化。所以，我们称 sogo 为变量。

2.3.2 声明变量

在 C 程序里，常用数据来做运算，在运算之前这些数据先存于变量中。为了储存数据，计算机必须先分配主存储器区域给变量，才能把数据存于该区域中。如同您住进饭店之前，必须先订房间，才能住进去一样。饭店的“订房”就相当于变量的声明，声明时必须说明：

- 变量的种类；
- 变量的名字。

此时，计算机便安排空间给变量，准备储存数据。变量的生成过程，称为声明（Declaration）。变量的种类，又称为变量的类型（Type）。变量的类型说明它将储存何种类型的数据。例如：指令 int x 声明变量 x，其类型为 int（整数）表示 x 将储存整数数据，如 100、–3 等。于是，我们称 x 为整数变量（int Variable）。

计算机分配空间给 x 之后，数据就能存入 x 中了。C 的“=”符号表示“存入”操作。例如：指令 x=3 的意义是把 3 存入 x 变量中，得出 x 值为 3。

2.4 变量的声明格式

常见的变量声明格式为：

```
数据类型   变量名称；
--- --- --- ---   --- --- --- ---
```

例如：int day;

但也有其他各种声明格式。当编写程序时，常需要看看别人的程序或删改别人的程序，使之合乎您的需要。因此，我们有必要熟悉各种声明格式。欲声明多个同类型的变量，可写为：

```
数据类型   变量名称 1，变量名称 2,…，变量名称 n；
---------   ----------   ----------   ----------
```

例如：int score, charge;

此处声明两个 int 变量 score 和 charge。请看例子：

```
int x;  int y;
x=3;
y=5;
printf( "%d", x+y );
```

两个声明指令 int x 及 int y，可摆在同一行。然而，因 x 与 y 的类型皆为 int，可合并为 int x, y;，这是常见的写法。

2.5 如何输出数据

人们常借助显示器观察计算机的操作及运算结果，所以显示器又称为监视器（Monitor）。数据常显示于显示器上，也常印刷于报表纸上，这些数据包括常数及变量值。C 程序常利用 printf()函数来输出数据于显示器上，例如：

```
printf( "Hello, welcome you to C!" );
```

当 main()调用 printf()时，传送一个礼物——字符串"Hello, welcome you to C!" 给 printf()，这个礼物就是参数（Argument），放在小括号内。printf()接到礼物后，就将其显示在显示器上："Hello, welcome you to C!"

printf()既能输出字符串，也能输出其他类型的数据。不论整数、浮点数还是字符串，都能交由 printf()转变类型并重新组成字符串，再显示于显示器上（见图 2-7）。代码如下：

```
printf( "a=%d", 30 );
```

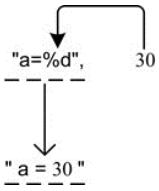


图 2-7

printf（"a=%d", 30）显示出"a=%d"字符串。其中，30 是来宾人数，%d 为输出格式，代表来宾的座位。printf()请来宾坐上位子，"a=%d"变成了"a=30"字符串。于是 printf()将结果显示于显示器上 a=30。此程序输出整数常数 30，同样地，也能显示变量的值。例如：

```
int x;
x=30;
printf( "a=%d", x );
```

指令 int x 声明 x 为 int 变量。x=30 是将 30 存入 x 中。printf（"a=%d", x）是把 x 值（30）摆于%d 位置上，"a=%d"就变成"a=30"，然后显示于显示器上—— a=30 了。此程序显示出 int 变量值，当然也能将浮点数显示于显示器上（见图 2-8），代码如下：

```
int x;    float y;
x=50;     y=1.25;
printf( "x=%d y=%f", x, y );
```

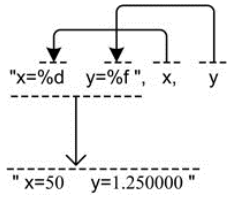


图 2-8

printf()欲显示出"x=%d y=%f"字符串，但含%d 及%f 格式，并非完整的字符串。%d 是来宾 x 的座位，%f 是来宾 y 的座位，于是 printf()把 x 值摆在%d 位置，同时把 y 值摆在%f 位置上。"%f"将 1.25 补足 6 位小数。于是，"x=%d y=%f" 字符串转变成"x=50 y=1.250000" 字符串，然后显示于显示器上。此程序输出整数和浮点数，当然也能显示两个字符串。例如：

```
printf("%s, %s", "Hello", "welcome you to C!");
```

"%s, %s"含%s 格式，但并非完整字符串。于是 printf()将第 1 位来宾"Hello" 摆在第 1 个%s 位置上，第 2 位来宾"welcome you to C!" 摆在第 2 个%s 位置上。"%s %s" 字符串就变为"Hello, welcome you to C!"再送往显示器：Hello, welcome you to C!。%d 用来显示整数值，%f 用来显示浮点数，而%s 用来显示字符串。

2.6 如何传递参数

C 程序是由多个函数组合而成的。当函数间互相调用（Call）时，主函数给予函数礼物（参数），子函数接到参数后，会做些运算工作，再把运算结果回赠（Return）给主函数。例如：

```
#include <stdio.h>
void sub( int x )
{  printf( "x=%d", x );   }

int main(void)
{ sub( 30 );   /*  CALL sub() and PASS 30 to x */
  return 0;
}
```

main()内的指令——sub(30)调用 sub() 函数且赠送礼物 30。sub() 接收到礼物后立刻存入 x 中，使 x 值为 30。如果 main()代表圣诞老人，30 代表圣诞礼物的个数，那么 x 就是装礼物的袜子。当此程序执行时，由 main()的“{”执行到 sub(30)指令，便将 30 传递给 x，同时转移到 sub()的“{”，开始执行 sub()内的指令，一直到 sub()的“}”止，再返回 main()，继续执行 sub(30)后的指令，直到 main()的“}”，就全部结束了。

main()送给 sub()的礼物是整数常数 30。sub() 必须以 int 变量接受它。主函数送给子函数一些礼物，子函数处理后，常会回赠给主函数。例如：

```
int sub()
{ return 8;  }

int main(void)
{
  printf( "%d", sub() );
  return 0;
}
```

sub() 的 return 指令将 8 传回给 main()。此时 sub() 值就是 8，亦即指令 printf（"%d", sub()，相当于 printf（"%d", 8）

请再看个例子：

```
int dog( int p )
{ return( p*2 );   }

int main(void)
{ int x;
  x = dog( 5 );
  printf( "x=%d", x );
  x = dog( 10 );
  printf( "x=%d", x );
  return 0;
}
```

当计算机执行 x=dog(5)指令时，先计算 dog(5)的值，于是执行 dog() 内的指令。因 p 值为 5，return 指令将 p*2 的值 10 传回 main()，所以 dog(5)的值为 10。归纳如下。

- 当计算机执行 dog(5)时，操作为：

  第 1 步——将 5 传给 dog()的 p 变量。

  第 2 步——跳到 dog()执行其内的指令。

- 当计算机执行 return(p*2)时，操作为：

  第 1 步——将 p*2 的值传回 main()，就成为 dog(5)的值。

  第 2 步——返回 main()，继续执行。

得到 dog(5)的值为 10，则指令 x=dog(5)相当于 x=10，因此 x 的值为 10 且由 printf()显示于显示器上。

接下来，x=dog(10)将 10 传给 dog()，得 p 值为 10。指令 return(p*2)将 20 传回 main(); 得 dog(10)值为 20，且存入 x，得 x 值为 20。最后，printf()将 20 显示于显示器上。

2.7 如何替函数和变量命名

函数和变量应有个好听且又有韵味的名字。为 C 函数与变量命名时，规则如下：

- 由英文字母（'A' ～'Z'，'a' ～'z'）、阿拉伯数字（'0' ～'9'）和下划线（'_'）所组成。
- 第 1 个字符必须是英文字母或下划线，不能是阿拉伯数字。

- 字母大小写有区别，大小写英文字母被视成不同的字符。
- 最长允许 32 个字符。
- 不能与关键字（Keyword）相同。

ANSI C 的标准关键字（即保留字）如表 2-1 所示。

表 2-1 C 的关键字

int	external	else
char	register	for
float	typedef	do
double	static	while
struct	goto	switch
union	return	case
long	sizeof	default
short	break	auto
unsigned	continue	if
void	typeof	enum
const	volatile	
signed	while	

此外，各种开发环境（例如 Turbo C、VC++、C++ Builder、Dev-C++等）通常会增订一些特定的关键字，只要查看其 Help 就会有详细的说明。例如，Turbo C 增加了 asm、_cs、_ds、_es、_ss、cdel、far、huge、near 等关键字。另外，请注意以下几点。

- 计算机会区分大小写字母，例如：hello、Hello 及 HELLO 分别代表不同的名称。（不过，有些开发环境里的编译及连接器，部分参数如\Gc 等会影响到大小写的区别，所以还是请您在开发之前参阅开发环境的 Help 说明。）
- 开头两字符不能是“__”（两个连续下划线）。例如：__dog 、__aBC 等应避免。这种名称是编译程序（Compiler）专用的。
- 开头字符若是“_”下划线时，第 2 个字符不能是大写字母。例如，_ABC、_X 等应避免。

例 1 下列名称对吗？

1. TIME　　2. dog3　　3. 3dog
4. do#g　　5. Dog_3　　6. _3416
7. A3-B3　　8. A$　　9. int
10. interest　　11. __pascal　　12. _USA

【答案】

1. TIME　　对。

2. dog3 对。

3. 3dog 不对。原因：勿以阿拉伯数字开头。

4. do#g 不对。原因：不应含特殊符号（?，%，# 等）。

5. Dog_3 对。

6. _3416 对。

7. A3-B3 不对。原因：不宜用减号“–”（请注意：减号与下划线不同）。

8. A$ 不对。原因：不应含特殊符号$。

9. int 不对。原因：int 为关键字，不能作为名称。

10. interest 对。虽然前三个字母是 int，与关键字 int 相同，但 int 与 interest 对计算机而言是不同的。

11. __pascal 不对。不能以“__”开头。

12. _USA 不对。开头为“_”，不能紧跟着大写字母。

例 2 下列变量名称哪些适合代表学生的年龄？

1. student_age 2. st_age 3. _stage

4. sTage 5. age_of_student 6. STUDENT_AGE

7. StudentAge

【答案】

皆为适当的名称，但能表达出其含义者为佳：

1. student_age 佳。

2. st_age 尚可。

3. stage 不佳，常误以为英文单词 stage。

4. sTage 尚可。

5. age_of_student 佳。

6. STUDENT_AGE 佳。

7. StudentAge 佳。

第 3 章　C 语言的数据类型

3.1 基本数据类型

计算机将数据分为三类：整数、字符及浮点数。

类型		例如
整数	←→	1, 2, 3, 0, –1
字符	←→	'a', '*', '9'
浮点数	←→	12.345, –0.0659

编写程序时，欲将数据储存于内存中，其简单方法是先声明变量，然后把数据存入变量中。变量的声明格式为：

```
类型      变量名称;
----      ----------
```

因此，可以使用三种基本类型来声明变量，让它储存上述三种数据。这三种类型是 int、char 和 float。例如：

```
#include <stdio.h>
int main(void)
{
  int x = 3200;
  char ck = '%';
  float score = 95.32;
  printf( "%d, %c, %f", x, ck, score );
  return 0;
}
```

x 为 int 类型，ck 为 char 类型，score 为 float 类型。整数数据常存于 int 变量中，字符数据常存于 char 变量中，浮点数常存于 float 变量中。例如，整数 3200 存于 x 中，字符'%'存于 ck 中，而 95.32 存于 score 中。

3.2 整数类型

整数不含小数，例如：5、25、–3600 等。它可分为正整数（Positive Integer）、0 及负整数（Negative Integer），例如：1 为正整数，–1 为负整数。通常，1 代表正 1，–1 代表负 1。根据所占内存空间的大小，整数又分为短整数（short int）与长整数（long int）两种。short int 占 2 Bytes（16 bits）空间，而 long int 占 4 Bytes。在 20 世纪 80 年代至 90 年代里，计算机的内存较贵，CPU 缓存器也比较小，主流配置为 16 bits，当时，C 的 int 类型就是指占 16 bits 的 short int 类型。

到了 21 世纪，CPU 缓存器的主流配置是 32 bits，所以目前 C 的 int 类型大多是指占 32 bits 的 long int 类型。当声明 int 变量时，您必须知道它占 4 Bytes 空间。如果不需要这么大的空间，就声明为 short int，既可节省内存空间，又可加快运算速度。

3.2.1 short int 类型

您已经了解到：“变量”代表一块内存空间，“类型”则决定空间的大小。short int 变量占 2 Bytes（16 bits）空间。由于空间的限制，short int 变量只能储存–32768～32767 范围内的值。这些值排成环状如图 3-1 所示。

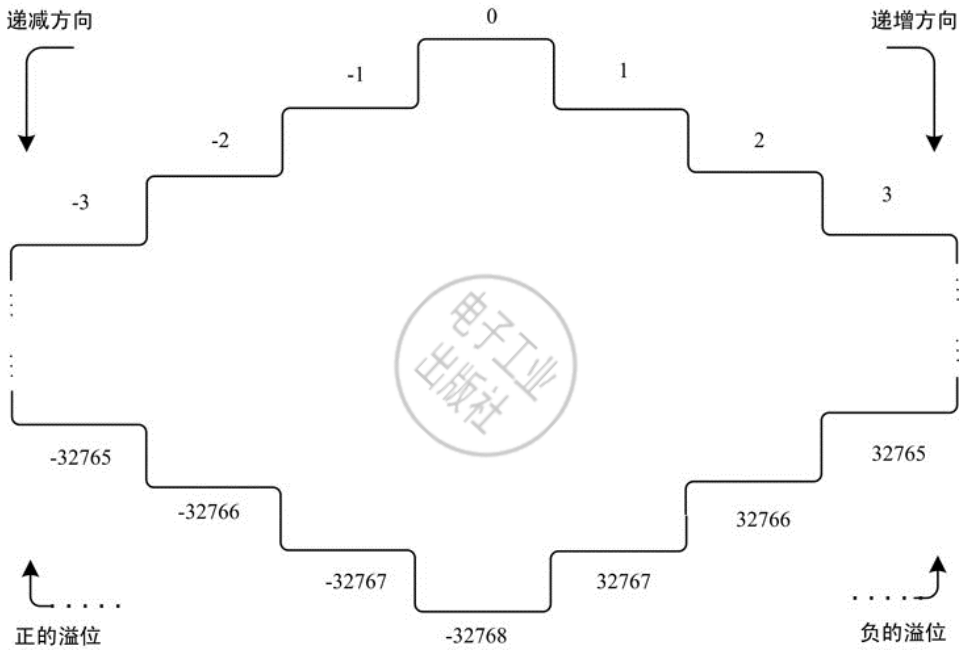


图 3-1

大于 32767 或小于–32768 的整数已超过 16 bits 所能储存的范围，若欲存于 short int 变量中，将发生“溢位”（Overflow），例如：

```
short int ik = 32767;
ik = ik + 1;
```

32767 存入 ik 变量中，于是 ik 值为 32767。指令 ik=ik+1，令 ik 值加 1，于是 ik 值超过了 32767，发生正溢位。如图 3-1 所示，比 32767 大 1 的正溢位值是–32768。因此，ik 值为–32768，而非 32768。小于–32768 的整数存于 int 变量中，将产生负溢位。例如：

```
short int egg = -32769;
```

–32769 小于–32768，存于 egg 中，产生了负溢位。依图 3-1 所示，比–32768 小 1 的负溢

位值是 32767。因此，egg 值为 32767，而非–32769。

产生环状溢位的原因是 short int 变量占 2 Bytes，共 16 bits，其格式如图 3-2 所示。

图 3-2

最左端的位代表正负号，若为 0，则表示正值；若为 1，则表示负值。计算机采用补码的概念来储存数值。例如，x 值如图 3-3 所示。

图 3-3

则此值为–1，何以得知呢？转成补码就知道了。转换过程如下：

Step 1 转为 1 的补码。

Step 2 加上 1。

x 为 2 的补码值为 1，其符号位为 1，表示负值，所以得到–1。

3.2.2 long int 类型

long int 类型，常简称为 int 类型。int 变量（即 long int 变量）占 4 Bytes（32 bits）空间。因空间较大，int 变量能储存–2147483648～2147483647 范围内的值。这些值排成环状，如图 3-4 所示。

大于 2147483647 或小于–2147483648 的整数已超过 32 bits 所能储存的范围，若欲存于 int 变量中，将产生“溢位”（Overflow）。例如：

```
int ik = 2147483647;
ik = ik + 1;
```

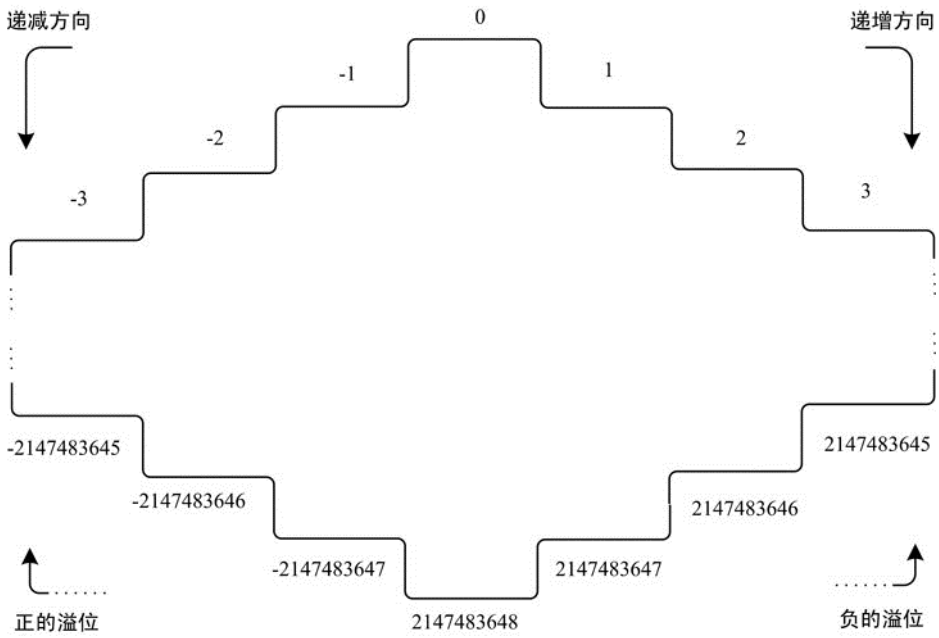


图 3-4

2147483647 存入 ik 变量中，于是 ik 值为 2147483647。指令 ik=ik+1，令 ik 值加 1，于是 ik 值超过了 2147483647，发生正溢位。如图 3-4 所示，比 2147483647 大 1 的正溢位值是–2147483648。因此，ik 值为–2147483648，而非 2147483648。发生环状溢位的原因是 int 变量占 4 Bytes，共 32 bits，其格式如图 3-5 所示。

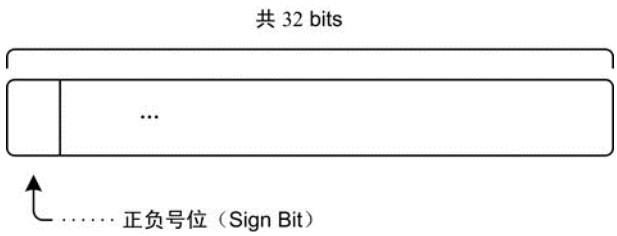


图 3-5

无论是 int 类型还是 short int 类型，其溢位的原理都是一样的。

3.3 无符号整数

在日常生活中，有些数据一直为正值，永不为负值。例如，年纪、人数等。它们可以存于 char、int 或 long 变量中，但会浪费一些空间。因此，衍生出新类型如下：

- unsigned char （无符号字符）

- unsigned short int （无符号短整数）
- unsigned int （无符号整数）

3.3.1 无符号字符（unsigned char）

介于–128 ~ 127 之间的整数，可存于 char 变量中。超出这个范围，宜采用 short int 变量储存。其声明格式为：

```
unsigned  char  变量名称;
--------  ----  ----------
```

凡是介于 0 ~ 255 之间的整数，皆可存于 unsigned char 变量中。例如：

```
unsigned char y;
y = 150;
```

y 占 1 Byte 空间，能储存 0 ~ 255 之间的整数。再如：

```
char x=253;
unsigned char y=253;
int px, py;
px = x;
py = y;
```

x 值扩充为 2 Bytes 整数值，再存入 px 中。x 为 char 类型，有正负之分，把 x 的符号位扩充到 px 的左 Byte 中。若符号位为 1，左 Byte 就补 1；若符号位为 0，左 Byte 就补 0。同理，y 值扩充为 2 Bytes 整数值，再存入 py 中。y 为 unsigned char 类型，无负数概念，左 Byte 就补 0。

3.3.2 无符号短整数 （unsigned short int）

介于 0 ~ 65535 之间的整数，可储存于 unsigned short int 变量中。其声明格式为：

```
unsigned  short  int  变量名称;
--------  -----  ---  ----------
```

或

```
unsigned  short  变量名称;
--------  -----  ----------
```

unsigned short int 可以简写为 unsigned short，皆占 2 Bytes 空间。例如：

```
short int x;
unsigned short y;
x = 32768;
y = 32768;
printf( "x=%d \ny=%u", x, y );
```

32768 转为二进制值，存于 x 及 y 变量中，如图 3-6 所示。

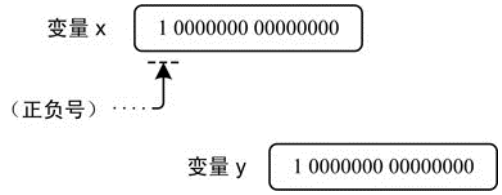


图 3-6

x 与 y 值相同，但 x 是 short int 类型，最左位的值为 1，表示负值。于是，"%d"格式将 x 值视为 2 的补码，就得到值−32768。y 变量为 unsigned short 类型，"%u"格式直接取得其值 32768。

3.3.3 无符号（长）整数（unsigned int）

介于 0 ~ 255 之间的整数应存于 unsigned char 变量中。

介于 0 ~ 65535 之间的整数应存于 unsigned short int 变量中。

介于 0 ~ 4294967295 之间的整数应存于 unsigned int 变量中。

unsigned int 与 int 变量皆占 4 Bytes 空间，但 int 数据的最左位为正负号，而 unsigned int 数据永远为正值，如图 3-7 所示。

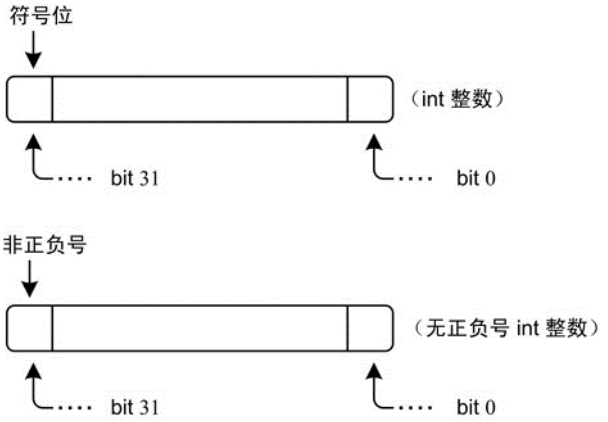
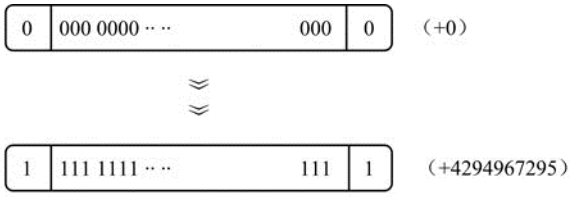


图 3-7

因此，unsigned int 的范围如图 3-8 所示。



如下所示

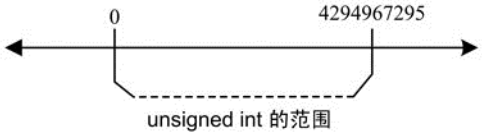


图 3-8

unsigned int 变量的声明格式为：

```
unsigned     int    变量名称;
---------- ------ ----------
```

或

```
unsigned  long int  变量名称;
--------- -------- ----------
```

或

```
unsigned  long   变量名称;
--------- ------ ----------
```

例如：

```
unsigned int salary;
salary = 80000;
printf( "%u", salary );
```

此程序输出值为 80000。printf()常用"%u"格式输出 unsigned int 的变量值。此外，"%x"、"%X" 及"%o" 格式也能输出 unsigned int 数据。80000 大于 65535，储存于 unsigned int 变量中是对的。然而，80000 也能存于 int 整数中，如：

```
int salary;
salary = 80000;
printf( "%d", salary );
```

这也是对的。习惯上，只要小于（或等于）2147483647（long 变量的最大范围）的值，

都可以储存于 int 变量中。

3.4 整数的输出格式

十进制整数是人们最习惯使用的，其次为十六进制、八进制和二进制。

printf()用格式（Format）来输出各种形式的数据，包括整数输出格式，如表 3-1 所示。

表 3-1 整数输出格式

输出格式	输出形式
%d	十进制整数
%i	十进制整数（与%d 相同）
%u	无正负的十进制整数
%x	十六进制（小写 a～f）
%X	十六进制（大写 A～F）
%o	八进制

"%d"（输出 Decimal Number）相当于"%i"（输出 Integer），皆输出含正负号的十进制数。如果符号位（Sign Bit）为 0，就输出正值；若符号位为 1，就输出负值。请看下面的例子。

```
short int k, n;
int a, b;
k = 32765;
n = k + 3;
printf( "k=%i, n=%d\n", k, n );
a=32765;
b = a + 3;
printf( "a=%i, b=%d", a, b );
```

此程序输出：

k = 32765，n = –32768

a = 32765，b = 32768

32765 介于–32768～32767 之间，能完整储存于 k 中。k 加上 3 之后存入 n 中。运算过程如图 3-9 所示。

k 值加上 3 后，超过了 32767，于是 n 的符号位值为 1。此时 k 和 n 都占用 2 Bytes 空间。指令：

```
printf( "k=%i, n=%d\n", k, n )
```

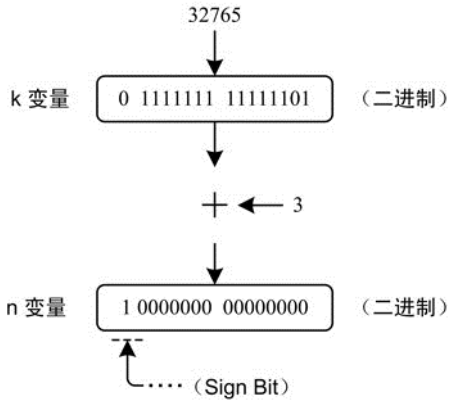


图 3-9

首先将 k 值扩大为 int 类型占 4 Bytes 空间，然后再输出。n 的符号位值为 1，将 n 值转为 2 的补码，输出负值−32768。"%i"及"%d"视最左位为符号位。"%u"视最左位为一般数据，无“负值”的概念。

k、a 和 b 的符号位值为 0，"%d"与"%u"的结果相同。n 的符号位值为 1，"%d"与"%u"的结果就不同了。"%d"输出十进制数据，"%x"输出十六进制数据，"%o"则输出八进制数据，例如：

```
Short int egg;
egg = 31;
printf( "%x", egg );
```

在 31 存入 egg 之前，先转为二进制，再存入 egg 中，如图 3-10 所示。

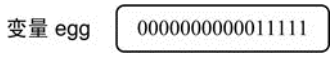


图 3-10

"%x"格式的“x”代表“十六进制”（Hexadecimal），printf()的 "%x" 将 egg 值转成十六进制数据，再显示出来。其转换过程如图 3-11 所示。

- 将 egg 值分成 4 组，每组含 4 bits。
- 将各组转为十六进制数字。

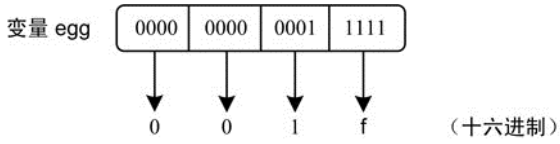


图 3-11

于是，得到 1f。若将"%x"改为"%X"，则输出大写的十六进制值，如 1F、BF900 等。"%x"格式输出符号 0～9、a～f，"%X"格式输出符号 0～9、A～F。"%o"格式的“o”代表“八进制”（Octal），printf()的 "%o"将 egg 值转成八进制数据，再显示出来。其转换过程如下。

- 每 3 bits 为一组。
- 将各组转为八进制数字。

3.5 字符类型

3.5.1 一般字符

字符概念让人联想到英文字母，其组成了文字数据。例如：

"I am object-oriented."

这是由'I'、'a'、'm' 等字符（字母）组成的字符串（文字）数据。此外，字符具有另一种意义：**“字符是一种特殊的整数数据”**。

字符与整数的区别为：

- 字符数据占 1 Byte，short int（短整数）数据占 2 Bytes，而 int（整数）数据则占 4 Bytes。
- 每个字符具有唯一的 ASCII 码。
- 字符变量能储存介于–128～127 之间的整数，而 short int 变量能储存–32768～32767 间的整数，int 变量则能储存–2147483648～2147483647 之间的整数。

字符（char）变量占 1 Byte 空间，格式如图 3-12 所示。

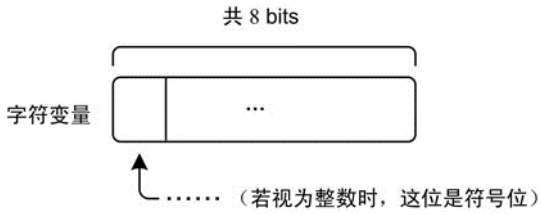


图 3-12

例如：

```
char cx;
cx = 65;
printf( "%c", cx );
```

cx 为字符变量，65 介于–128 至 127 之间，可完整存入 cx 中，如图 3-13 所示。

图 3-13

您能以两种方式来解释此值的意义。

（1）视为“整数”。

此时，最左位为符号位，于是 cx 值为正 65。

（2）视为“ASC II 码”。

01000001 为字母 A 的 ASC II 码，所以 cx 值为'A'。

printf()的"%c"将 cx 值视为 ASC II 码，输出其代表的字母或特殊符号。当然，字符能直接存入字符变量中，例如：

```
char asc;
asc = 'B';
printf( "%d", asc );
```

此程序输出 66。

'B' 字符代表字母 B 的 ASC II 码。指令 asc='B' 将 ASC II 值 01000010 存入 asc 中，则 asc 的内容如图 3-14 所示。

图 3-14

printf()的"%d"将 asc 值视为整数。最左位为 0，表示正值，于是 asc 值为正 66。同样地，阿拉伯数字也须区别，例如：**0 是整数，而'0' 是字符，代表阿拉伯数字 0 的 ASCII 码。**

请看如下例子：

```
char cx, cy;
cx = 8;
cy = '8';
printf( "%d, %d", cx, cy );
```

此程序输出 8，56。

指令 cx=8 将十进制值 8（即二进制的 1000）存入 cx 中，如图 3-15 所示。

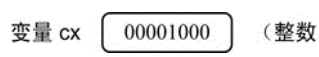


图 3-15

指令 cy='8' 将 ASC II 码 00111000 存入 cy 中，如图 3-16 所示。

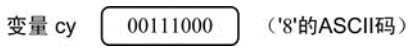


图 3-16

最后，printf() 将 cx 和 cy 值视为整数，并显示出来。

3.5.2 控制字符

在 ASCII 表中，有些码值能控制屏幕的显示，有些码值能控制打印机的动作，例如：

'\n'字符念作“Backslash N”，代表 ASC II 码——0xa。其中 n 表示“新行”（New Line），所以'\n'的作用是使光标换行。除了'\n'字符外，还有其他控制输出的特殊字符，如表 3-2 所示。

表 3-2 特殊的控制字符

字符符号	（十六进制）	意义
\n	0x0a	换行（newline）
\r	0x0d	回车（enter 或 return）
\t	0x09	跳栏 （tab）
\a	0x07	“哔”一声（bell）
\b	0x08	左移一格（backspace）
\f	0x0c	换页（formfeed）
\\	0x5c	反斜线（backslash）
\"	0x22	双引号（quotation mark）
\'	0x27	单引号（apostrophe）
\0	0x00	空字符（null）
\?	0x3f	问号（Question Mark）
\ooo	ooo	以八进制表示的字符（o 代表 0~7 阿拉伯数字）
\xhh	hh	以十六进制表示的字符（h 代表 0~9、a~f 或 A~F ）

'\ooo'相当于 0ooo，例如：

$$\text{'\123'} \equiv 0123$$

同样地，'\xhh' 相当于 0xhh，例如：

$$\text{'\x4a'} \equiv \text{'\x04a'} \equiv 0x4a$$

但它们之间有如下区别：

- '\123' 和'\x4a'为 char 数据，各占 1 Byte 空间。
- 0123 和 0x4a 为 int 数据，各占 2 Bytes 空间。

其中，'\0'是个特殊字符，每 bit 皆为 0，在 C/C++ 中扮演重要角色，即'\0'为字符串（String)的结尾字符。

3.6 浮点数类型

3.6.1 float 类型

浮点数（Floating-point Number）含有小数，例如：

1.25
0.0678
−3.141592678
−0.0000000000005
.....

当此种数据存入 int 或 char 变量时，会删掉小数部分，只存入整数部分。例如：

```
#include <stdio.h>
int main(void)
{
  int x;
  x = 3.14;
  printf( "%d", x );
  return 0;
}
```

指令 x=3.14 将 3.14 转换为 3（即删去 .14）再存入 x 中。于是，x 值为 3。欲将小数完整地储存于变量中，须使用浮点变量。其声明格式为：

```
float    变量名称;
-----    ----------
```

例如：

```
#include <stdio.h>
int main(void)
{
  int x;
  float f;
  x = 3.25;
  f = 3.25;
  x = x * 10;
  f = f * 10;
  printf( "%d, %f ", x, f );
  return 0;
}
```

此程序输出 30，32.500000。x 无法储存小数部分，其值为 3。f 则能储存小数部分，其值为 3.25。x * 10 相当于 3*10，其值为 30，于是 x 值为 30。f * 10 相当于 3.25*10 得 32.5，于是 f 值为 32.5。

printf()的"%d"格式以十进制方式输出 x 整数值，"%f"格式以十进制方式输出 f 浮点数

值。通常，"%f"格式在输出整数部分的同时输出 6 位小数（见图 3-17）。

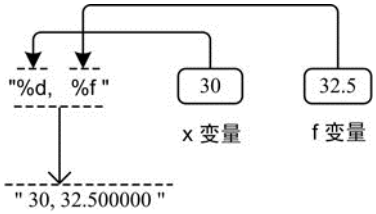


图 3-17

人们常把 32.5 表示成为 3.25*10 或 0.325*10^2 等，并称为“科学计数法”（Scientific Notation）。在 C 语言里，3.25*10 表示为 3.25e1 或 3.25E1，而 0.325*10^2 表示为 0.325e2 或 0.325E2。3.25e1 等于 0.325e2，皆表示 32.5，只是表示方式不同。其中，e 和 E 表示“指数”（Exponent）。例如：

```
#include <stdio.h>
int main(void)
{
  float fx, fy;
  fx = 7.47e1;
  fy = 5.5E-1;
  fx = fx + fy;
  printf( "fx=%f", fx );
  return 0;
}
```


此程序输出 fx=75.250000。

因为 7.47e1 ≡ 7.47 * 10^1 ≡ 74.7

5.5E-1 ≡ 5.5e-1 ≡ 5.5 * 10 ^ -1

所以 fx = fx + fy

相当于 fx = 74.7 + 0.55

得出 fx 值为 75.25。

printf() 的"%f"将 fx 值显示出来，如图 3-18 所示。

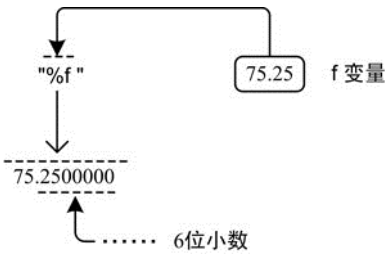


图 3-18

3.6.2 double 类型

我们前面谈过，float 数据常会四舍五入（或二进制的 0 舍 1 入），且具有 6（或 7）位有效数字。在精密的科学计算中，常要求更精确的结果，因此必须使用 double 变量来储存更多的有效数字。当程序含有浮点数数据时，应关心其精确到什么程度。例如：

```
float x;
x = 2.0 / 3.0;
printf( "%f", x );
```

此程序输出 0.666667。2.0 / 3.0 等于 0.6666666……，为循环小数，做四舍五入后存于 x 中，由 printf()负责输出。浮点数常为近似值，其计算结果也常为近似值，可能包含误差。误差愈小就愈精确，愈可靠，例如：

```
float pi, area;
pi = 3.141592678;
area = 100 * 100 * pi;
printf( "%f", area );
```

此程序输出圆面积 31415.927734。pi 只储存 6 或 7 位有效数字，四舍五入后，pi 值为 3.141593，与真正圆周率有些误差，因此圆面积也有误差。

为了提升可靠性、精确性，C 语言提供“双精度”（Double Precision）浮点类型，储存更精确的浮点数据。双精度浮点变量的声明格式为：

```
double   float   变量名称;
--------  -----  ------------
```

或

```
double   变量名称;
--------  ------------
```

这种变量占 8 Bytes 空间，储存 15 位有效数字，精确度较高，运算结果较可靠。其表示范围是：

–1.7e308 ~ –1.7e-308

及

1.7e–308 ~ 1.7e308

如图 3-19 所示。

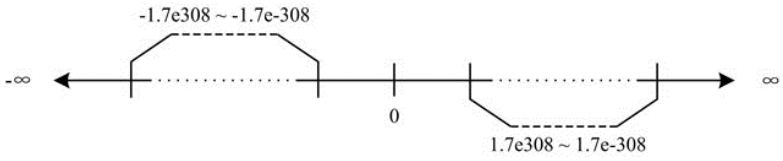


图 3-19

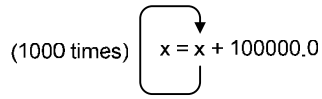
指数最大值是 308，为 15 位有效数字。上述程序可改写为：

```
double pi, area;
pi = 3.141592678;
area = 100 * 100 * pi;
printf( "%f", area );
```

此程序输出较准确的圆面积 31415.926780。pi 占 8 Bytes 空间，足够储存 3.141592678，不必四舍五入，其计算结果较接近真实面积。对浮点数进行运算时，应留意其可能的误差。例如：

float x=0.0；

x 初值为 0.0。若执行 1000 次如下指令：



则 x 值加上了 1000 个 100000.0。从理论上讲，以此 x 除以 1000.0，结果如下：

x = x / 1000.0；

应得到 x 值为 100000.0。但 C 程序输出 100000.03，有 0.03 的误差。若声明 x 为 double 变量：

double x=0.0；

其误差就小多了。"%f"能输出 double 和 float 数据。此外，"%e"、"%E"、"%g"及"%G"也能输出 double 数据，例如：

```
float x=1000.555;
double y=1000.555;
printf( "FLOAT X=%f \nDOUBLE Y=%f", x, y );
```

此"%f"输出 6 位小数：

FLOAT X=1000.554993

DOUBLE Y=1000.555000

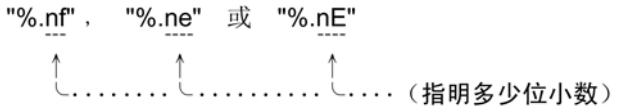
x 占 4 Bytes 空间，y 占 8 Bytes 空间。x 和 y 皆可能是 1000.555 的近似值，但 y 比 x 更接近 1000.555，请留意此区别。下面再看个例子：

```
float hen=4096.88888
double egg=4096.88888
printf( "%f \t%g\n", hen, hen );
printf( "%f \t%g\n", egg, egg );
```

此程序输出：4096.879883 4.09688e+003

4096.880000 4.09688e+003

egg 比 hen 更精确些。通常，"%f"输出 6 位小数，"%e"输出 5 位小数。欲输出较多或较少小数字时，可用如下格式：



例如：

```
float height=1000.555;
double depth=2500.325;
printf( "%f\t %.0f\t\t %.2f\n",
             height, height, height );
printf( "%f\t %.1f\t\t %.10f\n",
             depth, depth, depth );
```

此程序输出：

1000.554993 1001 1000.55

2500.325000 2500.3 2500.3250000000

"%.10f"表示输出 10 个小数字。double 类型的变量值能精确到 15 位有效数字，这对于一般数学计算已足够了。然而，当 double 变量仍无法满足您的要求时，有些 C 语言开发环境还提供了长双精度浮点数（long double）类型，可以精确到 17 位有效数字。

第 4 章　C 的数据运算

4.1 基本运算符号

运算（Operation）代表数据的处理操作。算术四则运算“加、减、乘、除”是计算机最基本的处理操作。“+”、“–”、“*”和“/”符号分别代表“加、减、乘、除”运算，称为“运算符号”（Operator）。参与运算的数据对象，称为“操作数”（Operand）。例如：5 + 3，指令含有两个操作数 5 和 3，含有 1 个运算符号+，代表“加”的操作。整个式子 5 + 3，通称为“表达式”（Expression），其值为 8。C 语言的基本运算符号有：

- 算术运算（Arithmetic Operators），为一般代数计算。
- 赋值运算（Assignment Operators），将数据存入变量中。
- 关系运算（Relational Operators），比较数据的大小，提供决策的依据。
- 逻辑运算（Logical Operators），做复杂的决策。
- 算术赋值运算（Arithmetic Assignment Operators），结合算术与赋值运算，提高数据的处理速度。
- 加 1 与减 1 运算（Increment and Decrement Operators），因为加 1 与减 1 的操作频繁，这两种运算可提高程序效率。
- 取地址运算（Address Operator)，为指针（Pointer）概念的根源，能灵活地存取数据。
- 按位运算（Bitwise Operators），用此运算能对最小数据——位（Bit）控制自如。
- 类型转换运算（Cast Operator），随时转换数据类型，增加运算的精确度。

数据运算（或计算）是计算机的本能，传统 C 的魅力即来自于强大的运算能力，善于利用各运算符号是程序设计的基础。

4.2 算术及赋值运算

算术运算符号（Arithmetic Operators）包括：

符号	意义
+	加
–	减
*	乘
/	除
%	求余数

赋值运算符号（Assignment Operator）为：

符号	意义
=	赋值

请回忆算术运算规则：

先“乘除”后“加减”

C 语言沿袭了此规则，例如：

```
int x;
x = 1 + 2 * 3;
```

先做“=”右边的表达式 1+2*3，它包含两个运算符号（+ 和*）及三个操作数（1、2 和 3）。先做 2*3 得到 6，再拿 6 与 1 相加，得到 7 并存入 x 中。先执行“*”运算，再执行“+”运算，亦即“*”比“+”优先。也可以说，“*”的优先级（Priority）高于“+”的优先级。必要时，可用小括号改变上述优先级，如下：

```
int a,b;
a=( b=(1+2)*3 ) - 1;
```


该表达式含两层括号，按照算术规则：

内层小括号最优先。

其执行顺序为（如图 4-1 所示）。

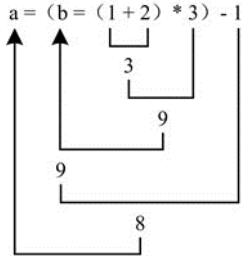


图 4-1

“–”符号除了代表“减”运算外，还有个用途，即代表“负值”运算，它只有一个操作数，称为“单元运算符号”（Unary Operator)。负号——“–”应摆在其操作数之前。例如：

```
int x, y;
x = 3 * -1;
```

```
y = -x;
printf( "%d", y );
```

–1 的“–”符号代表“负值”运算，令其操作数 1 变为负值——–1。

“–”（负值）的优先级高于“*”及“/”，所以先执行“–”运算，再执行“*”，最后执行“=”运算。算术与赋值运算的优先级归纳如下所示：

运算符号	优先性
()	高
– （负号）	↑
* / %	|
+ –	↓
=	低

上述是一般的优先级，能用小括号改变这种顺序。再看看算术运算吧！对整数做“+”、“–”及“*”等运算，其结果是整数，然而，整数的“/”运算是否为整数呢？想一想，数学上 1/4 等于 0.25，可是对计算机而言呢？请看个例子：

```
int x;
x = 1 / 4;
```

为何不是 0.25 呢？您可能解释为 1 除以 4 得 0.25，由于 x 是整数，无法储存小数。所以存入 x 时，删除掉小数“.25”，因而 x 值为 0。这个解释并不完全正确，因为计算机做“1 除以 4”时，就已删去小数了，所以 1/4 的值是 0，而非 0.25。再进一步分析，表达式 x=1/4 是子表达式（Sub Expression），“/”的操作数皆为整数，其结果是整数（删去小数了），然后再存入 x 中。只要表达式中含有“=”，就应了解“=”右边表达式的结果为何种类型，例如：

```
float y;
y = 1 / 4;
```

先计算 1/4，结果为 0，再存入 y 中，于是 y 值为 0.0。如果希望答案为 0.25，可写为：

```
float y;
y = 1.0 / 4;
printf( "%f", y );
```

由于“/”的操作数 1.0 是个浮点数，所以结果为浮点数 0.25，y 值为 0.25。像 1.0 / 4 这样的式子中含不同类型的操作数，通常称为“混合型表达式”。处理这种表达式时，C 语言将类型分为如下等级：

类型	等级
double	↑ 高阶
float	
unsigned	
int	
unsigned short	
short int	
char(unsigned char)	↓ 低阶

运算时，低阶类型逐渐提升至高阶类型，所以表达式结果的类型决定于最高阶操作数的类型。例如：

```
double x, y=4.0;
x = 5/y;
```

先计算“=”右边的式子 5/y，“/”有两个操作数 5 及 y。y 为 double 类型，5 为 int 类型，double 较 int 高阶。于是低阶的 5 向 y 看齐，把 5 提升为 double 类型的 5.0。5 是 int 类型，占 4 Bytes 空间，提升后的 5.0 占 8 Bytes 空间。5 / y 运算值为 1.25，其为 double 类型，存入 x 中，得出 x 值为 1.25。

了解了 int 提升为 double 类型的过程后，其观念也能用于别的类型上。现将“类型提升”（Type Promotion）的规则表示为如图 4-2 所示。

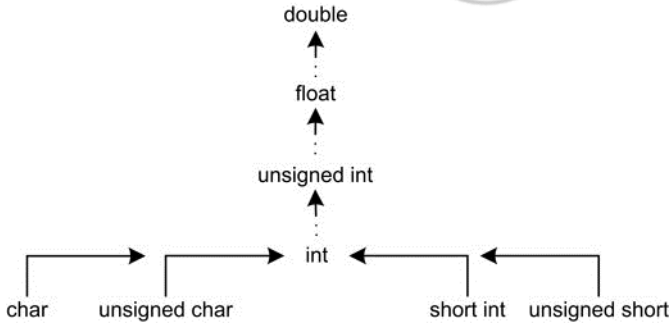


图 4-2

说明如下：

- char、unsigned char、short int 和 unsigned short 类型的操作数，无条件提升为 int 类型。
- float 类型的操作数无条件提升为 double 类型。此时，操作数的类型不外乎 int、unsigned int 或 double 三种。
- 按照运算符号的优先级，将其操作数提升为一致的类型，方法是依照 double > float >

unsigned int > int > unsigned short >short 的等级，低阶操作数向高阶操作数看齐。

4.3 关系运算

关系运算（Relational Operations）又称为“比较”运算，比较两项数据，看看相等或不相等，若不相等，还可比较其大小。比较结果为 True 或 False，这个结果称为“逻辑数据”（Logical Data）。

逻辑数据只有 True 及 False 两个值，计算机利用 1 及 0 表示，其中 1 代表 True，而 0 代表 False。这两个值可存于 int 变量中，例如：8>5 表示“8 大于 5”吗？答案是 True。计算机做运算，得到的结果为 1，表示 True。于是我们说，关系表达式 8>5 的值为 1。

C 的关系运算符号及意义如下所示：

关系表达式	意义
a < b	a 小于 b 吗？
a > b	a 大于 b 吗？
a == b	a 等于 b 吗？
a != b	a 不等于 b 吗？
a <= b	a 小于或等于 b 吗？
a >= b	a 大于或等于 b 吗？

上述 a 及 b 的值可为字符、整数或浮点数，例如：

```
int y;
y = 5 > 4 + 3.0;
```

表达式 5 > 4 + 3.0 含“+”及“>”两个符号，根据规则“算术运算优先于关系运算”，于是先做“+”运算，然后做“>”运算。当做“+”运算时，将 4 提升为 4.0，使其与 3.0 的类型一致，得结果为 7.0。当做“>”运算时，将 5 提升为 5.0，使其与 7.0 的类型一致。因 5.0 小于 7.0，所以所得结果为 0。虽然关系运算的优先性低于算术运算，但可用小括号()改变。例如：

```
int y;
y = (5 > 4) + 3.0;
```

小括号令“>”优先于“+”，就先做“>”，再做“+”运算。请注意“赋值”运算符号——“=”与“等于”运算符号——“==”的区别。简单规则是：单等号（“=”）代表“赋值”运算，双等号（“==”）代表“等于”运算。

4.4 逻辑运算符号

什么是好学生？有人回答“品学兼优”，意思是品行好“而且”功课好。其中“品行好”是条件，“功课好”也是条件，“而且”是逻辑运算，条件是它的操作数。“条件”（Condition）常是“关系表达式”，所以逻辑运算符号常连接两个关系表达式，并且组成更大的关系表达式。例如：

```
int x=5;
if( x < 6 && 2 == 3 )  printf( "12345" );
```

“&&”代表“而且”，连接两个小条件（关系表达式）x<6 和 2==3，并组成复合的大条件 x < 6 && 2==3。x 值为 5，得出 x<6 值为 True（1）。2==3 值为 False（0）。接着拿两个条件值做“&&”运算，于是大条件相当于 1 && 0，其值为 0。由于大条件 x<6 && 2==3 值为 0，所以不执行 printf()，屏幕不显示任何数据。C 语言的基本逻辑运算符号为：

运算符号	意义
&&	AND（且）
||	OR （或）
!	NOT（非）

“&&”及“||”皆需两个操作数，其操作数应为逻辑数据（0 或 1）。“!”只需一个操作数，其操作数也应为逻辑数据。

- “&&”运算

&&	1	0	（右操作数）
1	1	0	
0	0	0	（结果）
（左操作数）			

1 && 1　等于 1，表示若左右两个条件皆成立，则总条件就成立。

1 && 0　等于 0，

0 && 1　等于 0，表示若有一个条件不成立，则总条件就不成立。

0 && 0　等于 0

例如，如果好学生的条件是品学兼优，那么品行好但不用功的学生，其“品学兼优”条件则不成立，就不是好学生。

- “||”运算

||	1	0
1	1	1
0	1	0

1 || 1 等于 1，

1 || 0 等于 1，表示若有一个条件成立，则总条件就成立。

0 || 1 等于 1

0 && 0 等于 0，表示若左右两条件皆不成立，则总条件就不成立。

例如：

```
char ch;
ch = 'A';
if( ch == 'B'-1 || 2>3 )
   printf( "LION" );
```

大条件 ch == 'B'–1 || 2>3，包含三种运算：

- “–”为算术运算。
- “==”和“>”为关系运算。
- “||”为逻辑运算。

其优先级是：算术运算优先，关系运算其次，逻辑运算最后。

运算种类	优先等级
算术运算（如“+” ）	↑高
关系运算（如“==”）	|
逻辑运算的“&&”和“||”	↓低

因此，先做“–”再做“==”，最后做“||”运算。由于大条件值为 1，就执行 printf()，输出结果 LION。

- “!”运算

!	1	0 （操作数）
	0	1 （结果）

!1 等于 0，原来的 1 代表条件成立；!1 值为 0，代表条件不成立。

!0 等于 1，原来的 0 代表条件不成立；!0 值为 1，代表条件成立。

其中，“!”是“单元运算符号”（Unary Operator），其优先性高于二元运算，如下所示。

种类	运算符号	优先等级
括号	()	↑高
单元运算	! – （负号）	|
算术运算	+ – * / %	|
关系运算	< > <= >= == !=	|
逻辑运算	&& ||	|
赋值运算	=	↓低

例如：

```
if( !0 > 3 * -1 )     printf( "567" );
if( !( 0 > 3 * -1 ))  printf( "TOM" );
```

此程序输出 567。条件!0 > 3*–1，含四个运算符号“!”、“>”、“*”及“–”。由于“!”及“–”是单元运算符号，优先于“>”和“*”。“*”是算术运算，优先于关系运算“>”，如下所示：

运算符号	优先等级
“!”、“–”	↑高
“*”	|
“>”	↓低

其中，“!”与“–”属于同等级，其顺序是：由左至右。于是，此条件值为 1，就显示出 567。另一条件!（0 > 3 * –1），小括号改变了优先级。括号内的运算优先于括号外的运算，则!（0 > 3 * –1）条件值为 0，于是不显示出 TOM。请注意，“&&”优先于“||”。

4.5 算术赋值运算符号

C 语言的目标之一是追求执行速度的极限。从这个角度来看，必须善用各种运算符号。算术赋值运算符号（Arithmetic Assignment Operator）即为此而设。它是由算术运算符号（+、–、*等）与赋值符号（=）组成的复合运算符号，包括：

算术赋值运算符号	意义
+=	相加并存入
-=	相减并存入
*=	相乘并存入
/=	相除并存入
%=	求余数并存入

例如：

```
char p;
p = 'A';
p += 1;
printf( "%c", p );
```

同理，“*=”代表“乘上”的意思。例如，表达式 x* = 5 – 1 包含两个运算符号“*=”及“–”，哪个先做呢？由于算术赋值运算的优先性很低，与赋值符号“=”同级。按照规则，“–”优先于“=”，而“=”与“*=”同等级；所以“–”优先于“*=”。请注意：

- “!=”符号不是“!”与“=”的复合运算。

  原因是：“=”不能和逻辑运算（“&&”、“||”、“!”）合并。

  “!=”永远代表“不等于”之意。

- “>=”符号不是“>”与“=”的复合运算。

  原因是：“=”不能和关系运算合并。“>=”永远代表“大于等于”之意。

4.6 加 1 及减 1 运算符号

计算机常做加 1（Increment）与减 1（Decrement）的操作。为了加快执行速度，C 语言提供“++”运算符号做加 1 运算，提供“—”运算符号做减 1 运算。若想令变量 x 加 1，可写为 x=x+1；或 x+=1；或 ++x;。上述三者的运算结果相同，但“++”运算更加快速、简洁。例如：

```
int x=6;
x += 1;
printf( "%d", x );
```

此输出结果 7。上述代码相当于：

```
int x=6;
++x;
printf( "%d", x );
```

“++”是“+=1”的简洁写法，为一元运算符号，其操作数必须是变量。

按照规则，一元运算符号（像–、!等）优先于二元运算符号，例如：

```
int x, y;
x = 5;
y = ++x;
printf( "%d, %d", x, y );
```

此输出结果为 6, 6。指令 y = ++x 的“++”优先于“=”，于是先做“++”运算，再做“=”运算。先做“++”运算，x 值加上 1；再做“=”，将 x 值 6 存入 y 中。与“++”对应的是“--”运算。“--”就是“–=1”（减 1）的简洁写法，能加快程序的速度。它与“++”皆具有下述特性：

- 优先性很高；
- 操作数必须是变量；
- 必须紧接着操作数。

例如：

```
int x=5;
if( --x-5 )
   printf( "%d", x );
```

条件--x-5 含两个运算符号“--”及“-”。这个“-”符号前后各有一个操作数，所以是二元运算的“减法”符号，并非“负值”符号。“--”优先于“-”运算，于是先做“--”，得出 x 值为 4，再将 x 值 4 减去 5，结果为–1。

4.7 取地址运算符号

写程序时，常将数据存入变量中，例如：

```
short int  x=2;
int  y=3;
```

将 2 存入 x 变量中。x 代表一块内存空间，占 2 Bytes。y 也代表一块内存空间，占 4 Bytes。有时候，您会问：x 变量所代表的空间，到底是哪两个 Bytes？就像许多人知道台北有栋大楼叫“老爷大酒店”，但不知道坐落于何处一样。此时，可打电话去问，便可得知“老爷大酒店”的地址了——中山北路 2 段 37 号。同理，欲知 x 变量的位置，可向计算机询问 x 变量的地址（Address）在哪里？

计算机内存被划分成许多个小区域，每个区域的大小为 1 Byte，恰能存一个字符。相连的 2 Bytes 恰能存一个 short int 整数。相连的 4 Bytes 恰能存一个 int 整数。计算机对各 Byte 逐一编号，每个 Byte 有唯一编号，这就是该 Byte 的“地址”。这些地址是按 0、1、2 …… 依序编下去的。

声明指令 int x=2 要求分配 2 Bytes 空间给 x 变量，并将 2 存入 x 所代表的空间内。分配空间给变量时，计算机根据当时主存储器的使用情况，将可利用的空位分配给它。因此，您不能指定 x 变量的储存位置，那是计算机的责任与权利。所以，通常您不知道变量的真正地址，但计算机则很清楚，可以问它。方法是使用“取地址运算符号”（Address Operator）——“&”。例如：

```
short int x=2;
printf( "%u", &x );
```

“&”是单元运算符号，对变量做运算，且“&”必须摆在变量之前。在此 printf()中的&x 可询问计算机 x 的地址在哪里？指令如下：

```
short int  x=2;
```

生成 x 变量，它占用 2 Bytes 空间（假设为 Byte #4854 及#4855），其中第 1 个 Byte 的地址为 4854，计算机便将 4854 视为 x 变量的地址。此程序输出 x 变量的地址为 4854。至于 float 类型的变量，则占 4 个 Bytes 空间，其中第 1 个 Byte 的地址即为此变量的地址。

编写 C 语言程序时经常会用到“取地址运算”。C 语言的原始目标之一是取代汇编语言，例如，以 C 语言来设计 Unix 操作系统。由于 C 语言沿袭了汇编语言的地址观念，若能充分运用取地址运算，将可充分发挥 C 语言的潜能！

4.8 按位运算符号

位（bit）是最小的数据单位，8 个连续位组成“字节”（Byte）。前面用过的运算符号，皆视变量内的位为整体，并做运算。本节将说明如何存取变量内的某位，并与另一位做运算。在 C 语言程序中，常见位与位之间的运算，例如：在显示器上画图，可用位值来控制像点。若位值为 1，其对应的像点就亮；反之，若位值为 0，其对应像点就变暗。再如，在网络通信上，位运算是不可或缺的。

“按位运算”（Bitwise Operation）的符号是“&”。它对各位逐一做“AND”运算。如：

```
char x=12, y=2;
if( x & y )
    printf( "BIT" );
```

这个“&”用 x 的位值与 y 的位值做“AND”运算，如图 4-3 所示。

```
  000001100
& 000000010
-----------
  000000000
```

图 4-3

例如，x 最右位是 0，与 y 最右位 1 做“AND”运算，结果为 0。其他位也逐一与对应

位做运算。

除“&”符号之外，还有多个按位运算符号。如下所示：

符号	意义
&	逐位 AND
|	逐位 OR
^	逐位 XOR（互斥性 OR ）
<<	逐位向左移位
>>	逐位向右移位
~	1 的补码（1 变 0，0 变 1 ）

这种逐位符号只针对字符类型（即 char 及 unsigned char）及整数类型的数据（含 unsigned、int 及 long 等）做运算，不能对 float 及 double 类型的数据做运算。

4.8.1 &、|、^ 及~ 运算符号

“按位”意味着对操作数中各相对位逐一做运算。两个相应位的运算情形，如表 4-1 所示。

表 4-1 按位操作数

位值	运算结果				
左操作数	右操作数	&	|	^	~
0	0	0	0	0	
1	0	0	1	1	
0	1	0	1	1	
1	1	1	1	0	
0	1				
1	0				

做“&”运算时，若两个位皆为 1，则结果位值为 1；若有任何位为 0，则其结果就是 0。做“|”运算时，若任何位为 1，其结果就是 1；只有当两位皆为 0 时，才得到 0。做“^”运算时，若两个位值不相同，结果为 1；反之，两位值相同（皆为 0 或皆为 1），得出结果为 0（同值相斥，所以称为互斥性 OR）。欲将某位值设为 0，可用“&”运算来担任，例如：

```
char x = 0xfa;
x = x & '\x07';
printf( "%#X", x );
```

此输出结果为 0x2。x 的内容为二进制的 11111010。欲将 x 的左边 5 个位值皆变为 0，就用 x 值与二进制的 00000111 做“&”运算，如图 4-4 所示。

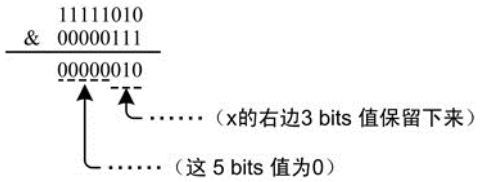


图 4-4

左边 5 位变成 0，且保留右边 3 位的值。写程序时，常需混合上述各运算符号，所以必须熟悉各符号的优先性。其优先性如下所示：

符号	优先性
~	↑高
&	|
^	|
|	↓低

例如：

```
if( ~1 | 'a' & 1 )    printf( "KING" );
if( ~(1 | 'a') & 1 )   printf( "QUEEN" );
```

此输出 KING，如图 4-5 所示。

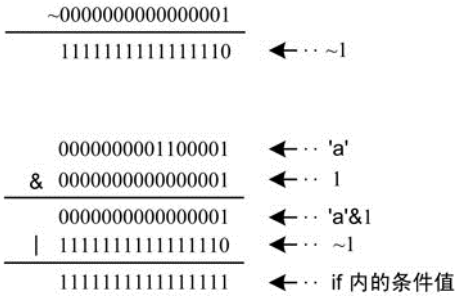


图 4-5

因此，表达式~1 | 'a' & 1 的值为 True（非 0）。表达式~ （1|'a'） & 1 含个小括号，须优先做“|”运算，如图 4-6 所示。

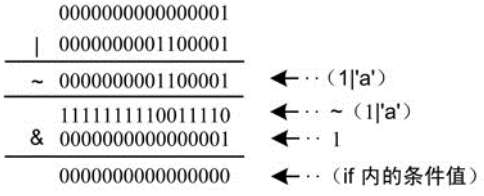


图 4-6

于是，条件值为 False（为 0）。

4.8.2 <<及>>运算符号

“<<”运算是将各位依序向左移位（Left-Shift）。“>>”运算是将各位依序向右移位（Right-Shift）。请看下面的例子：

```
char x=3, k, r;
k = x<<1;
printf( "%d\n", k );
r = k<<1;
printf( "%d\n", r );
```

x 是 char 变量，占 8 bits 空间，其内容为二进制的 00000011，如图 4-7 所示。

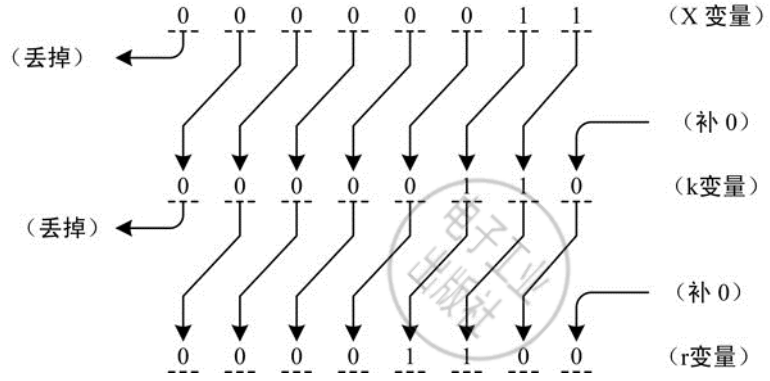


图 4-7

经过“<<”运算后，k 值为 6，r 值为 12。请注意，做“<<”运算时，最右位会补上 0，而最左位则被抛弃。

“>>”运算令位向右移动，但对于含正负号和无正负号的数据，其移动的方式有所不同。

- 当操作数为 char、int 及 long 类型时，则将正负号位值补充到最左边的位，例如：

```
Short int a=0x8008, b, c;
b = a >> 1;
c = b >> 2;
printf( "%#x\n%#x", b, c );
```

a >> 1 及 b >> 2 的运算过程如图 4-8 所示。

于是，此代码输出：

```
0xffffc004
0xfffff001
```

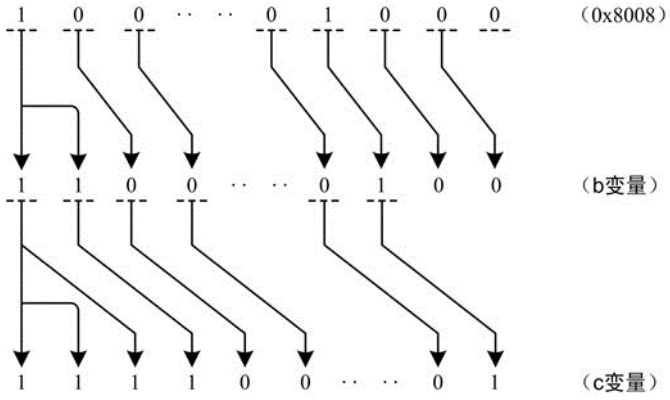


图 4-8

- 当操作数为 unsigned char、unsigned int 及 unsigned long 时，因没有正负号位，就自动补 0 了。

4.8.3 按位赋值运算符号

前面介绍过算术赋值运算，如+=、*=、/=等，合并了算术运算与赋值运算。同样地，也能将按位运算与赋值运算合并。

x &= y	相当于	x = x&y
x |= y	相当于	x = x|y
x ^= y	相当于	x = x^y
x <<= n	相当于	x = x<<n
x >>= n	相当于	x = x>>n

例如：

```
unsigned char value = 0x71;
value <<= 2;
value >>= 1;
printf( "%#x", value );
```

此输出结果为 0x62。指令 value <<= 2 相当于 value = value << 2。其运算过程如图 4-9 所示。

此程序将 value 左边第 1 位清为 0 了。

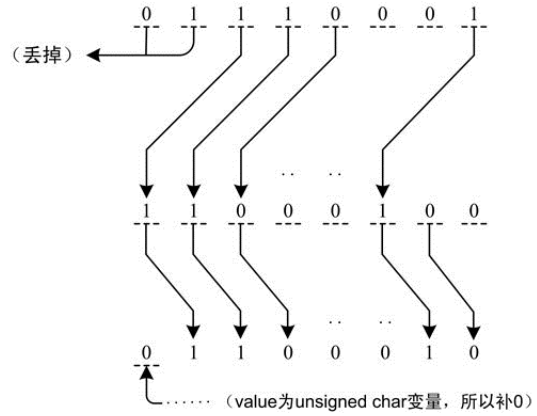


图 4-9

4.9 类型转换运算符号

声明变量时，必须确定其类型。例如，指令 int y 生成 y 变量，且应该说明其类型为 int 的整数。至于常数的类型可通过外表即可知道。例如，0x2f 是十六进制的整数，35U 是 unsigned 类型的，而 5.3F 是 float 类型的。从这些常数或变量值可得出其他类型的值，例如：

```
char  x='A';
int   y = x;
```

此时，y=x 指令先将 x 值扩大为 2 Bytes 的整数，然后存入 y 变量中。这个扩大的过程就是类型转换（Cast），即取出 x 的值，将之转换为 int 类型。这段程序可表示为：

```
(int) x
```

再如：

```
char x = 2;
printf( "%f", (float)x );
```

这里（float）x 的意思是：取出 x 值，将其转换为浮点数，如图 4-10 所示。

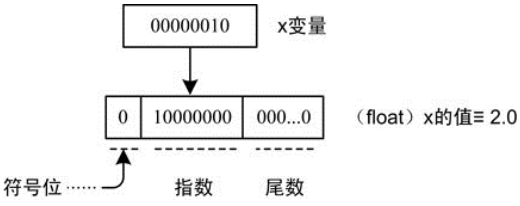


图 4-10

转换后的值为 2.0，其内部表示法也转为浮点数表示法，但并未改变 x 的内部情形。因此，类型转换运算的格式为：

（新类型）数据

↑
└‥‥（包括变量、常数及表达式的值）

例如，2 是 int 类型，而（float）2 就是 float 类型，不再是 int 类型了。亦即（float）2 的值为 2.0。此外，介绍另一个重要的概念：return 指令如何返回浮点数呢？请先看个重要的规则——“函数须声明其类型；这个类型告诉 return 应返回何种数据”。

例如：

```
int sub( double x, long y )
{
  return (int)(x + y);
}

int main(void)
{
  double a;
  a = (double)sub( a, (long)300 );
  printf( "%f", a );
  return 0;
}
```

sub() 函数的类型是 int，表示 sub() 内的 return 指令将返回 int 数据。但是（x+y）值的类型为 double，此时应加上（int）将（x+y）值转为 int 类型就一致了。main()里面的指令 sub（a,（long）300），从 sub()的类型可知道它将返回 int 值。于是加上（double）将它转为 double 类型再指定给 a 变量。

现在，您已知道函数类型的意义了。其目的在于想告诉其他函数：“我将返回该类型的值，请您接待它。”也许您会问：“如果函数并不返回任何值，应该如何声明呢？”很简单，若函数不返回任何值，就声明其为 void 类型。C 语言运算符号的优先级如下所示。

运算类	符号	优先性
括号	( )	↑高
单元	! ~ ++ -- - （负号）& （位）(type)	|
算术	* / % + -	|
关系	<< >> < <= > >= == !=	|
逐位	& ^ |	|
逻辑	&& ||	|
指定	= += -= *= /= <<= >>= &= ^= |=	↓低

括号的优先级最高，所以它常被用来改变整个表达式的运算顺序。

第 5 章　决策与循环

5.1 逻辑运算与决策

关系运算的结果为逻辑数据 True 或 False，可作为决策判断的依据。例如，人们根据红绿灯决定前进或停止，流程图如图 5-1 所示。

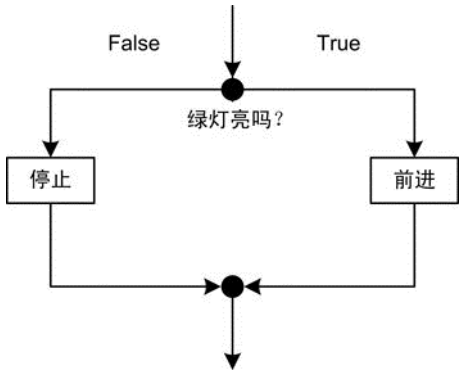


图 5-1

“红绿灯状况”是判断的依据，称为“条件”（Condition）。在程序中，关系表达式的值为条件；若其值为 True，表示此条件成立；若其值为 False，表示条件不成立。例如下述程序：

```
int x, y;
x = 3;
y = 5;
if( x < y )
    printf( "*" );
else
    printf( "#" );
```

x<y 为 if 指令的条件，如果条件成立（即 x<y 值为 True），则执行 printf("*")。如果条件不成立（即 x<y 值为 False ），就执行 printf("#")。如图 5-2 所示。

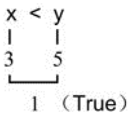


图 5-2

x<y 相当于 3<5，其值为 1，表示条件成立，就执行 printf("*")，表示输出“*”。条件表达式有时只含操作数而无操作符，例如：

```
int x;
x = 4 > 0;
if( x )
   printf( "ABC" );
```

x 位于 if 的括号内，成为判断的依据——条件，计算机就视其为逻辑数据。由于 4>0 的值为 1，表示 True，条件成立，于是执行 printf("ABC")，输出结果 ABC。此程序相当于：

```
int x;
x = 1;
if( x )
   printf( "ABC" );
```

x 位于 if 的括号内，成为逻辑数据。因 x 值为 1，表示 True，就执行 printf(“ABC”)。当 int 变量位于条件的位置上时，就成为逻辑数据。若其值为 1，就表示条件成立；若为 0，表示条件不成立。然而，若其值既非 1 也非 0，又如何呢？简单规则：

“若变量值为 0，代表 False；若不为 0，就代表 True”。

例如：

```
int x;
x = 2+5;
if( x )
   {
       printf( "pqr " );
       printf( "XYZ" );
   }
```

x 值为 7，不是 0，条件成立就输出结果 pqr XYZ 字符串。

请注意：因为 if 与 else 是控制指令，并非一般运算指令，所以 if（条件）与 else 之后不用加分号。有时候，当条件为 True，应执行的指令不只一个时，则须用大括号将这些指令括起来。同样地，当条件为 False，应执行的指令不只一个时，也必须用大括号将它们括起来。在 if 部分或 else 部分中，若只含一个指令，则其大括号可省略。例如：输入一个字符，检查看是否为阿拉伯数字，代码如下。

```
char digit;
digit = getchar();
if( digit >= '0' && digit <= '9' )
  { printf( "DIGIT" ); }
else
   printf( "Not digit" );
```

这里 if 部分和 else 部分各含一个指令。if 部分的大括号说明只有一个指令。else 部分没有大括号，意味着 else 部分只含一个指令。因此，只含一个指令时，可省略大括号。

5.2 嵌套的 if 指令

如果 if 部分或 else 部分内含其他的 if 指令，则称为“嵌套的”（Nested）if 指令。请看下面的程序——输入字符，并检查是否为大写字母。

```
char y;
printf( "Type in a character:\n" );
scanf( "%c", &y );
if( y <= 'Z' )
   if( y >= 'A' )
      printf( "[%c]", y );
```

当您阅读程序时，请先弄清 if 及 else 部分的范围。

例如，此程序含两个 if 指令，都没有 else 部分，且未使用大括号。因此，if 部分包括一个指令，其范围是到其后第一个分号“;”为止。

内层 if 指令较单纯，其范围包括 printf() 指令。外层 if 指令的范围包括内层 if 指令。必要时，可加上大括号来让其更清楚些。

因此，找出 if 部分的简单方法为：“有大括号，看大括号；无大括号就看第一个分号。”

执行 printf() 时，当碰到第一个分号时，就结束内层 if 部分，也结束外层 if 部分。有些情况，else 部分也含别的 if 指令。

5.3 多选 1 的抉择

最常见的选择动作是“2 选 1”，它含有两个选择性的“操作”，让计算机挑选其中之一。此外，还常有“3 选 1”的情形。这可利用“嵌套的 if”来表示这种情况，例如：

```
if(k==1)  行为 1;
 else if(k==2) 行为 2;
    else if(k==3)  行为 3;
```

这是 3 选 1 抉择，根据三个条件来抉择。除了“嵌套的 if”可以表达“多选 1”的抉择之外，还有一种常用 switch...case 指令，能更清楚地表达 3 选 1 的抉择。

```
switch(k)
  {
    case 1:
           行为 1;
           break;
    case 2:
           行为 2;
           break;
    case 3:
           行为 3;
           break;
  }
```

各 case 的末尾含有 break 指令。此指令具有重要用途，请小心掌握。其简单规则：除非有特别目的，每个选择行动之后宜加上 break 指令，这是个好习惯。什么特殊目的呢？稍后再说明。请先看个例子：由键盘输入整数，其值为 1、2、3 或 4；让计算机打印出相对应的

“星期几”名称。这是 4 选 1 的情形，其程序可写为：

```
int k;
scanf( "%d", &k );
switch( k )
   {
    case 1:
        printf( "Monday" );
        break;
    case 2:
        printf( "Tuesday" );
        break;
    case 3:
        printf( "Wednesday" );
        break;
    case 4:
        printf( "Thursday" );
        break;
   }
```

switch()指令使用大括号将各 case 括起来。其中，在 switch 的小括号内，必须是整数或字符。而且 case 与“冒号”(“:”) 之间的数字，也必须是整数或字符。此程序让您输入整数，假若输入整数 2，则 k 值为 2。switch(k) 的意思是：用 k 值 2 跟各 case 后面的整数比较。计算机依序比较各 case，先用 k 值 2 与第一个 case 的整数值 1 比较，若它们不相等，就跳过第一个 case 部分。继续与第二个 case 的整数值 2 比较，若其值相等，就执行此 case 部分—— printf("Tuesday")及其后的 break 指令。switch 指令通常含几个 case 部分，一旦碰到 case 内的 break 指令，就立即跳到 switch...case 指令的尾巴，亦即 switch 指令的右大括号（")"）。如图 5-3 所示。

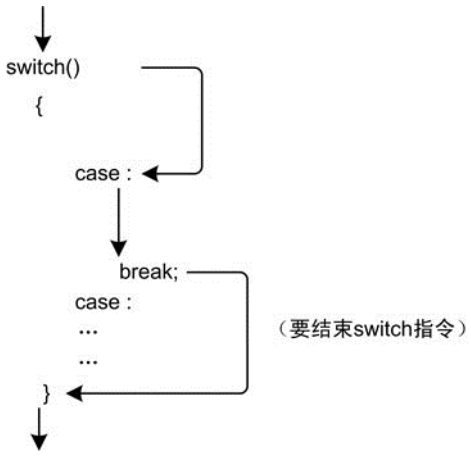


图 5-3

一旦执行到 switch 指令的结尾大括号，就完成 switch 指令。

5.4 while 循环

人与计算机的区别是人喜欢思考但讨厌重复性的单调工作，而计算机则擅长这种工作，不厌其烦！while 循环（while-loop）可以让计算机重复地工作。其格式如下：

```
while （条件）
  {
    重复性工作
  }
```

先看条件是否成立，若条件为 True，就做工作；做完工作再回头看条件，若条件仍然成立，再做工作；做完了再回头看条件，一直循环到条件为 False 时才离开这个 while 循环。

在数学计算上常见到循环。例如，计算某数 n 的阶乘 n! = n * (n-1) * (n-2) *... * 1。此计算可实现为如下 C 程序：

```
int factor=1, n=5;
while( n > 0 )
   factor *= n--;
printf( "factor=%d", factor );
```

求 5!的值，就是 5*4*3*2*1=120。

如果您常用阶乘运算，可将上述 while 循环摆在专门做阶乘运算的函数 factor()中。例如，求 3!+5! 的值，可写程序如下：

```
int factor( int n )
{
  int fac=1;
  while( n>0 )
        fac *= n--;
  return( fac );
}

int main(void)
{
  int result;
  result = factor( 3 ) + factor( 5 );
  printf( "3!+5!=%d", result );
  return 0;
}
```

此程序输出 3!+5!=126。

创建自己的函数，是 C 程序员的职责。此程序的 factor() 是自创的函数。程序共有两个函数 main()及 factor()。计算机执行 main() 的指令—— result=factor(3)+facotr(5)。因“+”的优先级高于“=”，所以先处理“+”的表达式 factor(3)+factor(5)。

5.5 for 循环

5.5.1 基本格式

这是最常用的循环，比 while 更简洁、清楚。尤其在明确知道重复次数时，更加好用。如果未知重复次数，仍可用 for 循环，但较常用 while 来表示。例如：重复做 100 次“加”法运算。在 C 语言中，则表达为：

```
int i, sum=0;
for( i=1; i<=100; i++ )
    sum = sum + i;
printf("SUM=%d\n", sum);
```

此程序输出 sum=5050。这是最简单的 for 循环格式。for 循环与 while 一样，当条件成立时，才能执行内部的重复性动作。for 循环也用大括号来括住其范围。若范围只含一个指令，可省略大括号。

for 的小括号内由两个“分号”(;) 来区分为三部分。中间是条件部分，左边是第一次进入循环的初始操作，常用来设定变量的初值。右边为第 2 、3 、……、n 次进入循环的准备动作，常令索引变量增加 1 或减 1 操作。for 循环可包含 if 指令，例如，求 0+2+4+6++8+10 的和，可写程序如下：

```
int sum, i;
for( sum=0, i=0; i<11; i++)
  if( !(i & 1) )
     sum += i;
printf( "sum=%d", sum );
```

这段代码输出 sum=30。表达式 i&1 相当于 i%2，其目的是判断 i 值为偶数或奇数。if 指令未使用大括号，其范围只含一个指令 sum+=i。同时，for 循环也未使用大括号，其范围也只含一个指令——if 指令，如图 5-4 所示。

图 5-4

进入 for 循环时，先执行小括号内的左边部分 sum=0，i=0，即令 sum 及 i 值都为 0，再执行中间的条件 i<11。如果条件为 True，就执行 for 循环内的指令 if。计算 if 条件! (i & 1)，若条件值为 1，就执行 if 内的操作 sum+=i。于是 sum 值加上 i，得 sum 值为 0。然后返回 for 括号中的右边部分 i++，令 i 加上 1，计算 for 的条件，这样循环下去，直到 for 的条件为 false 时才结束 for 循环。

通常，for 及 while 循环的出口是在条件的位置上。然而，必要时也能在其操作中以 break

指令跳出循环。例如：

```
for( int inning=1; inning<=6; inning++ )
  {
   比赛一局;
   if（分数相差太多）
     break;
  }
```

此 break 指令表示立即离开循环。

5.5.2 各式各样的 for 循环

上节已说明了 for 循环的基本动作。基于这些基本概念，可派生出各式各样的变化格式，使得 for 循环的威力大增。本节将介绍几种主要的 for 循环种类。

1. 索引变量的递增量不限定为 1

for 循环的递增量可大于 1。例如，有一棵树目前高度为 30 厘米，每周长高 5 厘米，则何时会长到 1 米以上呢？其间各周的高度是多少？可写程序代码如下：

```
for( int h = 30;  h<100;  h+=5 )
   printf( "Height=%6.2f\n", h);
```

进入 for 循环时，立即声明 h 且给予初值 h。h 表示 n 周后的高度，h += 5 表示每次循环增量为 5 厘米。

2. 浮点索引变量

for 循环的索引变量不限定为整数，也可为浮点数。例如：

```
for( double h = 0.3;  h<1.0;  h += 0.05 )
   printf( "Height=%6.2f\n", h);
```

其中，h 为树的高度，以米为单位。目前树高 0.3 米，每周长 0.05 米。

3. 索引变量值递减

for 循环的索引变量值还可为负值，例如：

```
for( int x=5; x>0; x-- )
   printf( "x=%6.2f\n", x);
```

每执行一次循环，x 值便减去 1 ，一直到 x>0 可得到的逻辑值为 false 为止，亦即直到 x 小于或等于 0 为止。

4. 索引变量的变化，不限于递加或递减

例如，有一支甘蔗长 195 厘米，若每天切掉一半，则多少天后其长度将小于 20 厘米呢？

```
int day = 0;
for( double p=195.0;  p>20.0;  p=p*0.5 )
   day++;
printf( "days = %d\n", day);
```

指令 p = p * 0.5 每次减少一半。

5. 设定数个变量的初值

例如：

```
int day;
for( double p=195.0, day=0  ; p<20.0; p=p*0.5 )
   day++;
printf( "days = %d\n", day);
```

6. 对数个变量做递增或递减运算

例如：

```
int day;
for( double p=195.0, day=0; p<20.0; p=p*0.5, day++ )
   { }
printf( "days = %d\n", day);
```

因{ } 内并无指令了，干脆也省略掉{ } 。

7. 递增或递减部分留白

for （初始设置；逻辑数据；___ ）

例如：

```
int day;
for( double p=195.0,  day=0;  p<20.0;  )
   {
    p = p * 0.5;
    day++;
   }
printf( "days = %d\n", day);
```

8. 初始设置部分留白

for（ ___；逻辑数据；递增部分）

例如：

```
int day=0;
double p = 195.0;
for(  ; p<20.0;  )
   {
     p = p * 0.5;
     day++;
   }
printf( "days = %d\n", day);
```

5.6 do 循环

C 的循环三剑客是 while 、for 及 do 循环。其中，for 最常见，while 次之，do 最少。然而，do 循环也有其适用的场合，请看它的用法吧！当 for 及 while 循环的条件为 true 时，才做重复性的动作；而 do 循环有个特色：先做一次动作，再查看其条件。如果条件为 true，则做一次循环，一直到条件为 false 为止。其格式为：

```
do
   {  重复性的工作  }
while(条件) ;
```

例如：棒球比赛，先赛 6 局，若不分胜负，则加赛，直到分出胜负为止。若以 do 及 while 循环表示，可写为：

```
do
   {
     比赛一局；
   }
while( 未赛完 6 局) ；
while( 平分秋色)
     {
       加赛一局；
     }
```

因此，do 循环先做事后比较，而 while 循环则先比较后做事。

第 6 章　C 语言的指针

6.1 认识指针

6.1.1 指针是什么

指针是一种特殊的变量，它的内容是变量的地址。因此，指针用来指向别的变量。例如：

```
int x;
x = 10;
int *p;
p = &x;
```

声明指令 int x ，说明 x 是 int 类型的变量。而另一个声明指令 int * p; 它说明了 p 为指针，且用来指向 int 类型的变量。于是通称 p 为“int 指针”。由于 x 为 int 类型的变量，p 为 int 类型的指针，恰好是一对。因此，可写指令：

```
p = &x;
```

p 就指向 x 了，如图 6-1 所示。

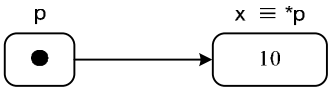


图 6-1

此时，有个重要概念：一旦 p 指向 x，x 就相当于*p。也就是说，x 与 *p 是等效的，代表同一个值。

有人说 C 语言并不好学，何以致此呢？大概是因为 C 程序中充满着“指针”（Pointer）的概念，而许多人认为指针很难缠。其实指针观念并不如想象中的麻烦，只是有点不习惯罢了。

上述的 p 和 x 都是变量，但其类型并不一样。y 是一般变量，类型可为 int （或 float、char 等）。x 变量的类型是指针（Pointer），这种指针变量专门用来指向别的变量。

6.1.2 指针的声明

指针变量的声明格式为：

```
类型 *   变量名称
------ ---  ----------
```

例如：int * x;

这个声明指令说明：x 是个指针（又称指针变量），其类型为“整数指针”int *。例如：

```
int y=50;
int *x;
x = &y;
*x = *x + y;
printf( "*x==%d\ny==%d\n", *x,y);
```

看到指令 x = &y，可联想到如图 6-2 所示的情况。

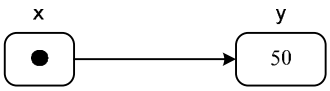


图 6-2

还可以也联想到如图 6-3 所示的情况。

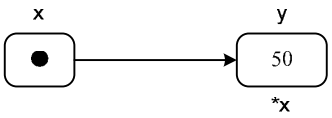


图 6-3

这表示 y 与*x 是等效的，代表同一块区域，它们是同一个变量的两个不同表示方法，它们的值相同。如果想存取此区域的值，可用 y 或*x 来存取它。指针除了能指向 int 外，也能指向 char、long 及 double 等变量。例如：

```
long *np, a=4;
double *dp, d=21.3;
np = &a;
dp = &d;
*dp += (*np)++;
printf( "*np= %ld,  *dp= %.1f", *np, *dp );
```

np 指向变量 a，可知*np 相当于 a。而 dp 指向变量 d，所以*dp 相当于 d。因此，指令 *dp += (*np)++运算后，d 变量的值为 25.3，即*dp 的值为 25.3；同时 a 变量的值为 5，即*np 的值为 5。

6.1.3 指针的指针

刚才介绍了指针可以指向 char、int 等变量，例如：

```
char k, ch, *pc;
ch = 'A';
pc=&ch;
k = *pc;
```

ch 是字符变量；pc 是指向字符的指针，*pc 相当于 ch。由于指针也是一种变量，所以能定义指针来指向指针。例如：

```
char k, ch, *pc, **ppc;
ch = 'A';
pc=&ch;
ppc = &pc;
k = *(*ppc);
```

ppc 是指向字符指针的指针。*ppc 相当于 pc，而*（*ppc）相当于*pc，也相当于 ch。所以 k 值等于 ch 值。这种指针的指针与数组有密切关系，将在第 9 章中详细介绍。

6.2 传递指针参数

主函数能将指针值传给子函数，子函数也能将指针传回主函数。这是 C 程序中常用的技巧，但对初学者而言，是较难的部分。例如：

```
void sub ( int *pb )
{ *pb = 88;  }

int main(void)
{
  int x=25;
  int *pa;
  pa = &x;
  sub( pa );
  printf( "x=%d", x );
  return 0;
}
```

指令 sub(pa) 将 pa 的值传给 pb，则 pa 等于 pb 了。亦即 pa 和 pb 皆指向 x 变量。这样子函数指针可指向主函数变量，如图 6-4 所示。

因 pb 正指向 x 变量，指令*pb=88 就将 88 存入 x，得 x 值为 88，如图 6-5 所示。

现在您已看到子函数 sub() 将 88 传回主函数了。在 C 程序中，将数据传回主函数的方法有以下两种：

- 使用传统的 return 指令。
- 使用刚才介绍的指针方法，这是高效的方法，盼您善用。

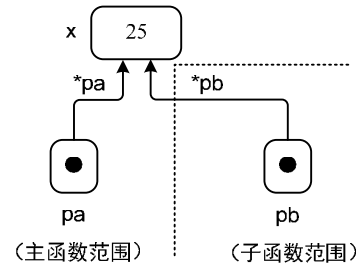


图 6-4

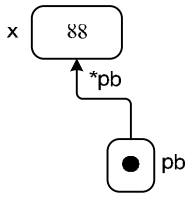


图 6-5

6.3 函数回传指针

子函数里的 return 指令能把整数、浮点数等数据传回给主函数。同样，return 指令也传回指针值。这是极常见的方法。例如：

```
int* sub( void *t )
{
  int *pi;
  pi = (int*)t;
  pi += 26;
  return (int*)t;
}

int main(void)
{
  int x=25;
  int *pi = sub( (void*)&x );
  printf( "x=%d", *pi );
  return 0;
}
```

sub()传回 int 指针。传回的指针也可以转换类型，例如上述程序相当于：

```
void* sub( void *t )
{
  int *pi;
  pi = (int*)t;
  pi += 26;
  return t;
}
```

```
int main(void)
{
  int x=25;
  int *pi = (int*)sub( (void*)&x );
  printf( "x=%d", *pi );
  return 0;
}
```

sub()传回 void 指针。传回到 main()时再将其转为 int 指针。

6.4 函数指针

在 C 语言里，指向函数的指针可能是最难懂的一环，但有时却很有用途。所谓函数指针，就是可以指向函数的指针。它的内容就是函数在内存里的地址（Address）。在计算机里，可以执行的函数程序代码必须放在内存的某个区域才能执行，此区域的起始地址就是该函数的地址。

例如，当 a()函数叫 b()函数去执行 c()函数时，就得把 c()函数的地址传给 b()函数。例如：

```
void f1( int k, int (*pf) () )
{
  (*pf)( 50+k );
  pf(60+k);
}
void call_back(int value)
{ printf( "%d", value);  }

int main()
{
  f1(88, &call_back);
  f1(100, call_back);
}
```

这里 main()调用 f1()并且把 call_back()的地址传过去，f1()的 pf 是指向 call_back()函数的指针，而在 f1()里又通过 pf 反过来调用 call_back()函数。

声明函数指针如下：

```
   int (*pc) ();
```

这说明 pd 将指向一个函数，而(*pf)()就是相当于该函数，它将返回一个 int 值。如果定义一个函数，代码如下：

```
      int c()
      {  return 10.5;   }
```

就可以写成如下指令：

```
pc = &c;
int k = (*pc)();
int m = pc();
```

其中，(*pc)() 和 pc()的意义是一样的。再看一个例子，如果有一个函数为：

```
double circle_area(double radius)
{
   return 3.1416*radius*radius;
}
double square_area(double length)
{
   return length * length;
}
```

如果 cal_area 是一个指针，它将指向 circle_area()函数，则此定义指令就可写为 double (*pca)(double radius);

编写程序如下：

```
int main()
{
  pca = circle_area;
  double a = pca(100);
  printf()
  pca = &square_area;
  double a = pca(66);
  printf()

}
```


此 main()里的 pca 指针可随时变更所指的函数，使这些函数可以被替换。如果拿汽车来打比方，main()是车体，而 circle_area()和 square_area() 则是两个轮胎；pca 指针让车体随时能轻易地更换轮胎，这是很有用的做法。

第 7 章　结构（struct）及动态内存分配

7.1 C 语言的结构（struct）

C 语言的结构可以含有不同类型的数据。例如，冰果室的价目表如下：

名称	简称	大小碗	价格
绵绵冰	M	S	15.0
爱玉冰	A	S	35.0
泡泡冰	P	S	10.5

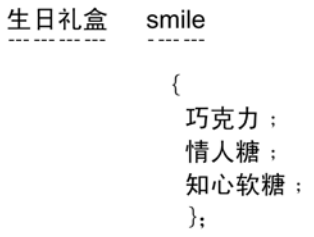
欲储存绵绵冰的“简称”、“大小碗”及“价格”，程序可写为：

```
#include "stdio.h"
struct Ice
{
  char sna;
  char size;
  float price;
};

int main(void)
{
  struct Ice x;
  x.sna = 'M';
  x.size = 'B';
  x.price = 20.5;
  printf( "%c, %c, %2.1f", x.sna, x.size, x.price );
  return 0;
}
```

此程序输出 M、B、20.5。C 结构的声明过程如下。

Step-1 定义结构类型。商品有牌子，定义类型就像描述一件商品的品牌。

例如：



这说明了 smile 生日礼盒内装有三种糖果，以 C 语言表达如下：

```
struct smile
{
   char sna;
   char size;
```

```
    float price;
};
```

这说明了 smile 结构内含三项数据：两项字符数据，一项浮点数数据。

Step-2 根据所定义的结构来声明结构变量。声明结构变量就像“订”礼盒。

例如：

生日礼盒 Smile x, y;

这里订购了两个 Smile 生日礼盒，一个给 x，另一个给 y。接下来，请看如何向计算机“订”结构变量（Structure Variable）。

```
struct smile x, y;
```

这里共声明了三个结构变量：x 和 y 为 smile 结构的变量，而 z 是 kiki 结构的变量。亦即 x 和 y 变量的类型是 struct smile；而 z 变量的类型是 struct kiki，如图 7-1 所示。

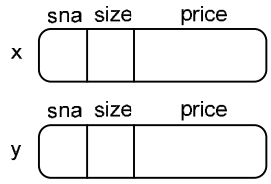


图 7-1

此时，x 变量占 6 Bytes （san 占 1 Byte，size 占 1 Byte， price 占 4 Bytes），y 变量占 6 Bytes （y 与 x 类型相同，大小也相同）。

请注意，上面所说的占用空间是指实际使用（即实际需要）的空间，例如 sna 和 size 的类型是 char，各需要 1 Byte，而 price 的类型是 float，需要 4 Bytes 空间，实际共需 6 Bytes。或者说实际使用空间为 6 Bytes。然而，计算机在分配内存空间时，因为 float 长度为 4 Bytes，会分配在 4 的倍数的地址上，所以实际分派的空间为 8 Bytes，如图 7-2 所示。

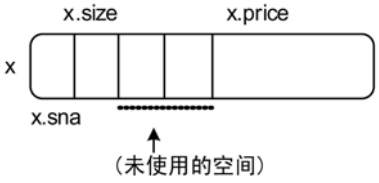


图 7-2

其中有 2 Bytes 是没有使用的。在程序上，仍然常说 x 变量占用 6 Bytes 空间，也常说计算机分配 6 Bytes 空间给 x 变量。在图形上，通常会省略没有使用的空间，所以常常将图 7-2

画为如图 7-3 所示。

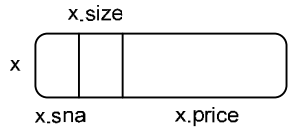


图 7-3

这两个图所表达的意义是一样的。

7.2 结构指针

整数指针能指向整数变量。例如：

```
int x, *px;
px = &x;
```

px 指向 x 变量了。同样地，也能声明结构指针来指向结构变量。例如：

```
#include "stdio.h"
struct Rose
{
  char na;
  float price;
};

int main(void)
{
  struct Rose x;
  struct Rose *pr;
  pr = &x;
  pr->na = 'R';
  pr->price = 26.8;
  printf( "Name=[%c], Price=%.1f", x.na, x.price );
  return 0;
}
```

px 是 struct smile 类型的指针，x 是 struct smile 类型的变量，pr 可能指向 x 变量。“&”运算能把 x 变量的地址存入 pr 中，使得 pr 指向 x 变量，如图 7-4 所示。

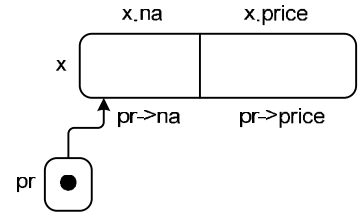


图 7-4

一旦 pr 指向 x，则 pr->na 相当于 x.na，且 pr->price 相当于 x.price，亦即 pr->sna 及 x.sna 皆代表 x 结构的 na 元素，而 pr->price 及 x.price 皆代表 x 结构的 price 元素。这时，x 的内容如图 7-5 所示。

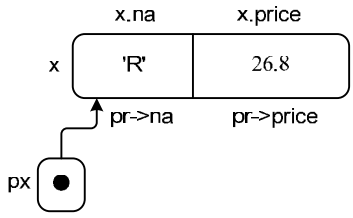


图 7-5

此程序输出 Name=[R]，Price=26.8。

7.3 传递结构参数

结构变量（Structure Variable）跟一般变量一样可当作参数来传递，但必须满足参数传递规则——对应参数的类型必须一致。例如：

```
#include "stdio.h"
struct Cash
{
  int x,y,sum;
};

struct Cash add( struct Cash v)
{
  v.sum = v.x + v.y;
  return v;
}

int main(void)
{
  struct Cash val, val2;
  val.x=10;
  val.y=5;
  val2 = add( val );
  printf( "SUM=%d", val2.sum );
  return 0;
}
```

此程序输出 sum=15。val 变量与 add()内的 v 变量具有同样的类型，满足了参数的传递规则。main()把 val 的内容传给 v，且 v 和 lop 是独立的自动变量，并非完全等价。add() 内的 return 指令将 v 变量值传回给 main()，然后存入 val2 变量中。

除了能传递结构变量给子函数外，也能将结构的指针传给子函数，并用它传递结构变量值。例如，上述程序相当于：

```
#include "stdio.h"
struct Cash
{
  int x, y, sum;
};

void add( struct Cash *pv )
{  pv->sum = pv->x + pv->y;       }

int main(void)
{
  struct Cash val;
  val.x=10;
  val.y=5;
  add( &val );
  printf( "SUM=%d", val.sum );
  return 0;
}
```

指令 add(&val) 把 val 的地址传给 add() 内的 pv 指针，于是 pv 指向 val 变量，如图 7-6 所示。

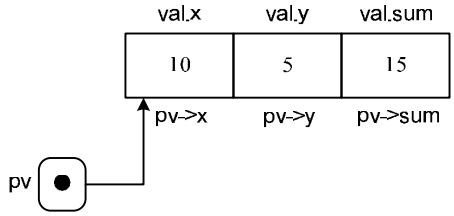


图 7-6

此时， pv->x ≡ val.x

pv->y ≡ val.y

pv->sum ≡ val.sum

因此，指令 pv->sum = pv->x + pv->y;

相当于 val.sum = val.x + val.y;

此时，val.sum 值为 15。

7.4 结构内的函数指针

7.4.1 先介绍 typedef 指令

typedef 指令能为类型取一个传神的名字。例如：

```
#include "stdio.h"
typedef unsigned char BYTE;
```

```
typedef unsigned short int TWOBYTE;

int main(void)
{
  BYTE x;
  TWOBYTE y,*py;
  x=50;
  y=1000;
  py = &y;
  printf( "x=%d,y=%d", x, *py );
  return 0;
}
```

此程序输出 x=50, y=1000。typedef 指令定义了两个类型 BYTE 和 TWOBYTE。BYTE 为 unsigned char 的别名，而 TWOBYTE 为 unsigned short int 的别名。此时：

声明指令 BYTE x;

相当于 unsigned char x;

且指令 TWOBYTE y, *py;

相当于 unsigned short int y, *py。

7.4.2 复习函数指针

我们在第 6.4 节里介绍过函数指针，它就是可以指向函数的指针。亦即它的内容是函数在内存里的地址（Address）。例如：

```
#include "stdio.h"
double cal_circle_area(double r)
{
  return 3.1416 * r * r;
}

int main()
{
  double a, b;
  double (*cal_area)(double);
  cal_area = cal_circle_area;   /*  设定指针内容   */
  a = (*cal_area)( 10 );    /*  调用函数  */
  b = cal_area(10);
  printf("%5.1f,  %5.1f\n", a, b);
  getchar();
  return 0;
}
```

此 call_area 就是指向 cal_circle_area()函数的指针了。指令 cal_area(10)调用 cal_circle_area()函数，并将参数 10 传递给它。 (*cal_area)(10)和 cal_area(10)的意义是一样的，因此 a 和 b 的值都是 314。

7.4.3 把函数指针放入结构里

在本书所介绍的 OOPC 语言里，就是利用结构来实现类（Class）的，而结构的变量就实现了类的对象。所以，本节和下一节所介绍的概念是 OOPC 语言的基础。在上一节里，我们已经定义过 cal_area 函数指针，并让它指向一个 cal_circle_area()函数。

现在将它放入结构里，让结构里含有该函数，并让它对应到 OOP 里的类概念。请先看程序代码：

```
#include "stdio.h"
struct Circle
{
  double (*cal_area)(double);
};

double cal_circle_area(double radius)
{
  return 3.1416*radius*radius;
}
int main()
{
  double a, b;
  struct Circle cir, *pc;
  pc = &cir;
  pc->cal_area = cal_circle_area;
  a = cir.cal_area(100);
  b = pc->cal_area(100);
  printf("%5.1f,  %5.1f\n", a, b);
  getchar();
  return 0;
}
```

很简单吧？Circle 结构里就有函数了。其 cir.cal_area(100)与 pc->cal_area()的意义是相同的，所以 a 和 b 的值都是 31416.0。

7.4.4 让函数存取结构里的数据细项

在刚才的程序代码里，有一点美中不足的地方就是：在 cal_circle_area()函数里无法存取结构里的数据细项值，所以必须由 main()传递 radius 值给它。请你想一想，如果把 radius 变量移到 struct Circle 结构里面，程序的执行将会如何呢？那么又该如何解决呢？办法是将程序代码改为：

```
#include "stdio.h"
typedef struct Circle Circle;

struct Circle
{
  double radius;
  double (*cal_area)(Circle*);
};
```

```
double cal_circle_area(Circle* t)
{
  Circle* cthis = (Circle*)t;
  return 3.1416 * cthis->radius * cthis->radius;
}

int main()
{
  double a;
  Circle cir;
  cir.radius = 10;
  cir.cal_area = cal_circle_area;
  a = cir.cal_area(&cir);
  printf("area = %6.2f\n", a);
  getchar();
  return 0;
}
```

很简单吧？main()函数将整个结构变量传递过去就可以了。其中的指令 Circle cir 生成了一个结构变量。此程序输出值为 314.16。如果与 malloc()库函数搭配使用，就能够动态地生成结构变量，并返回结构指针值。在下一节里，我们将详细介绍 malloc()库函数的意义和用法。不过，你可以先睹为快，上述程序代码相当于：

```
#include "stdio.h"
typedef struct Circle Circle;

struct Circle
{
  double radius;
  double (*cal_area)(Circle*);
};

double cal_circle_area(Circle* t)
{
  Circle* cthis = (Circle*)t;
  return 3.1416 * cthis->radius * cthis->radius;
}

int main()
{
  double a;
  Circle* pc;
  pc =  (Circle *)malloc(sizeof(Circle));
  pc->radius = 10.0;
  pc->cal_area = cal_circle_area;
  a = pc->cal_area(pc);
  printf("area = %6.2f\n", a);
  getchar();
  return 0;
}
```


此 malloc()动态生成了一个 Circle 结构变量，并让 pc 指向它。main()函数将结构变量指针传递过去。接下来，继续美化它，把 malloc()指令从 main()里独立出来，成为另一个 CircleNew()函数。那么，上述程序相当于如下代码：

```
#include "stdio.h"
typedef struct Circle Circle;
struct Circle
{
  double radius;
  void (*init)(Circle*, double);
  double (*cal_area)(Circle*);
};

void initialize(Circle* t, double r)
{
  Circle* cthis = (Circle*)t;
  cthis->radius = r;
}

double cal_circle_area(Circle* t)
{
  Circle* cthis = (Circle*)t;
  return 3.1416 * cthis->radius * cthis->radius;
}

void* CircleNew()
{
  Circle* p = (Circle*)malloc(sizeof(Circle));
  p->init = initialize;
  p->cal_area = cal_circle_area;
  return p;
}

int main()
{
  double a;
  Circle* pc = CircleNew();
  pc->init(pc, 10.0);
  a = pc->cal_area(pc);
  printf("area = %6.2f\n", a);
  getchar();
  return 0;
}
```

此程序添增了一个 init()函数，负责设定 radius 的初值。同时也将 malloc()指令移入 CircleNew()里，让 main()更加简洁明了。我们从第 11 章起介绍对象、类等 OOP 概念时，你就会发现这个 CircleNew()函数就是 OOP 里的对象构造器（Constructor），专门负责生成对象的函数，如图 7-7 所示。

图 7-7

7.5 动态内存分配

“动态”（Dynamic）的意思是：待程序执行后（Run-Time）再告诉计算机共需要多少内存空间，计算机依照需要立即分配空间，使它储存数据。这种空间，又称为“动态数组”（Dynamic Array）。传统数组的优点是简单易用，其弱点是缺乏弹性。而动态内存分配的方法恰恰可以弥补这项弱点。也就是说，传统数组的缺点是无法等到执行时才决定数组的大小。而动态内存分配方法，是在执行程序时，根据数据的多少来决定空间的大小。

在 C 语言里，提供 malloc()和 calloc()函数来动态地取得内存空间。

7.5.1 malloc()及 free()函数

malloc()和 free()是最常用的动态内存分配函数。如果在执行时需要空间来储存数据，则适合使用 malloc()函数，用完则用 free()释放该空间。malloc()的格式为：

```
指针 = malloc( 空间大小 )
                ↑
                ‥（共需多少 Bytes ）
```

例如：ptr = malloc(100);

这个指令要求计算机分配 100 Bytes 空间。malloc()函数会返回该空间的地址，且存入 ptr 内，于是 ptr 指向该空间，如图 7-8 所示。

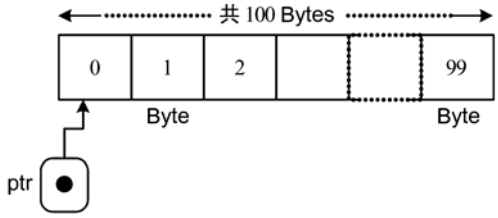


图 7-8

这就是用 ptr 指针来存取此空间的数据了。例如：

```
char *p;
p = (char*)malloc(8);
*p = 'A';
*(p+1) = 'L';
*(p+2) = 'L';
*(p+3) = '\0';
puts(p);
free(p);
```

malloc()定义于 malloc.h 头文件中，所以程序应。指令 p=(char *)malloc(8) 取得 8 Bytes

空间，且返回第 0 Byte 地址给 p 指针。于是 p 指向此区域的 Byte #0，如图 7-9 所示。

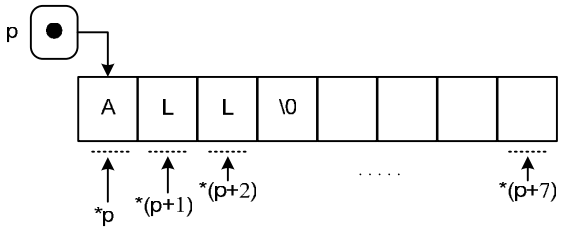


图 7-9

因此，*p 就是此区域的第 0 Byte，*(p+1)就是第 1 Byte，*(p+2)就是第 2 Byte，依序下去，就可用 p 来存取此区域内的数据了。于是把“ALL”字符串存入该区域中，并由 puts()显示出字符串“ALL”。

最后，free()函数归还于 malloc() 所申请的空间。其格式为：

```
free( 指针 )
       ↑
       └..（把此指针所指的区域归还掉）
```

例如，指令 free(p)释放掉 p 所指的空间。编写程序时请注意，使用完毕应立即调用 free()来释放空间；不然，计算机中可用的空间会愈来愈少。还记得吗？*(p+n)永远相当于 p[n]。因此，上述程序相当于：

```
char *p;
int size=8;
p = (char*)malloc(size);
p[0] = 'A';
p[1] = 'L';
p[2] = 'L';
p[3] = '\0';
puts( p );
free( p );
```

此程序输出字符串“ALL”。整数、常数、变量或表达式，皆可作为 malloc()的参数。

```
malloc( 空间大小 )
        ↑
        └....┤常数
             ├变量
             └表达式
```

malloc()取得的空间也能视为数组。除了存放字符串外，malloc()也可取得空间来储存整数数据。例如：

```
int *px, i, sum;
px = (int *) malloc (sizeof(int)*4);
*px = 10;
px[1] = 5;
*(px+2) = 2;
px[3] = 20;
for( i=sum=0; i<4; i++)
     sum += *(px+i);
printf( "Sum=%d", sum );
free( px );
```

此程序输出 sum=37。malloc()取得空间来储存 4 个整数数据。所以索取 sizeof(int)*4=8 Bytes 空间。由于 malloc()总传回第 0 Byte 的地址，且返回值必定是 char *类型。于是 malloc()返回的指针类型转为 int * 类型，然后存入 px 中，如图 7-10 所示。

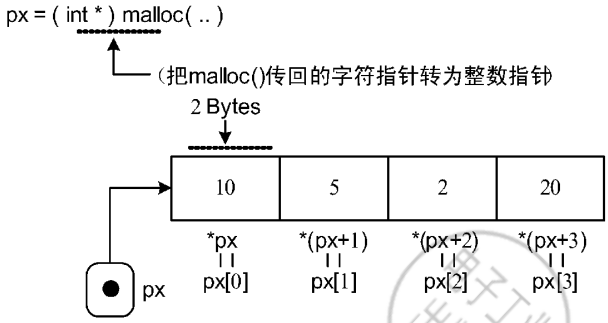


图 7-10

px 指向这块空间了。malloc()取得的空间还能储存更复杂的数据。例如：

```
struct kiki
{ char na[10];
  short int age;
};
typedef struct kiki NODE;
int main(void)
{ NODE *pn;
  pn = (NODE *) malloc (sizeof(NODE));
  if( pn==NULL )
  {     printf("malloc() failed\n");
        exit(0);
  }
  strcpy( pn->na,"Alvin");
  pn->age = 28;
  printf("AGE=%d", pn->age);
  free(pn);
  return 0;
}
```

此程序输出 AGE=28。typedef 指令定义的新类型——NODE 是 struct kiki 的别名。sizeof(NODE)值为 16，malloc()就索取 16 Bytes 的空间（因为 10 不是 4 的倍数，而 age 必须从第 12 Bytes 开始），并令 pn 指向此区域，如图 7-11 所示。

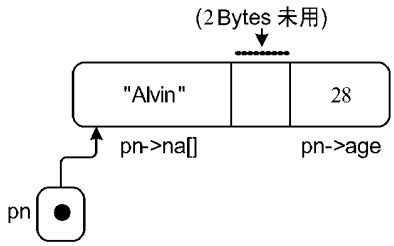


图 7-11

当 malloc()要求空间时，万一计算机无法提供足够空间，就返回 NULL 指针。所以，如果 pn 等于 NULL，就表示已无足够空间了，无法满足 malloc()的要求。编写程序时，应养成好习惯——检查 malloc()返回的指针值，看看是否为 NULL。

7.5.2 calloc()及 realloc()函数

calloc()函数与 malloc()的用途相同，能在程序执行时，视数据量的多少随时申请足够的空间。calloc()函数的格式为：

```
指针 = calloc( n, size )
          ↑   ↑
          |   └..（每项需 size 个 bytes ）
          └..（共有 n 项数据）
```

这相当于 `指针 = malloc( n*size )`

共分配了 n*size 个 Bytes 空间。通常，当数据分为许多项时，calloc()函数较方便。例如，想从数据文件内读取 10 个记录（Record），且储存于内存中，就能使用 calloc()函数了：

```
指针 = calloc(10, 记录长度)
```

这样便有足够的空间储存这 10 个记录了。此外，calloc()函数在获得空间后，会立即把空间清除为 0 （每 bits 皆清为 0 ），而 malloc()并不做这项清除工作。realloc()函数能扩大或缩小已取得的区域。数据常随着时间而增减，其空间也应随数据而增减，此时 realloc()具有极大弹性。其格式为：

```
指针 = realloc( 原先的指针，现需空间的大小 )
```

请注意，计算机可能会另找一块足够大的空间来取代原先分配的空间。因此，新返回的指针值可能与原先的指针值不同。

第 8 章　外部变量与静态函数

8.1 变量的储存种类

在大家庭中，每人都有自己的财产，也有属于家庭的公共财产。在 C 程序（犹如大家庭）中，每个函数都有自己的变量，也有属于整个程序的公共变量。私有财产属于个人，别人无权使用；同样地，私有变量属于函数，别的函数也无权存取。

私有变量的概念有助于“分工”；公共变量的概念有助于“合作”。分工合作可汇集所有人的力量，同心协力完成庞大的软件。写小程序时，这些概念并不很重要，一旦想写大程序，这种分工合作就重要无比了。例如：

```
int add( int x, int y )
{ x = x+y;
  return (x);
}

int mul( int x, int y )
{ x = x*y;
  return( x );
}

int main(void)
{
  int x=3, y=2;
  printf( "SUM=%d\n", add(x, y) );
  printf( "MUL=%d\n", mul(x, y) );
  return 0;
}
```

main()的 x 及 y 变量是 main()的私有变量，其内容不受其他函数的影响。add() 的 x 及 y 变量是 add()的私有变量，也不受 main()及 mul()的影响。同理，mul()的私有变量 x 及 y，也不受其他函数影响。因此，当编写 add() 函数时，不必担心别的函数会干扰 x 及 y 变量的值。因为 main()、add() 及 mul() 各有独立的天地，分工起来格外容易。例如，add()及 mul()由不同人来编写，在不同地点编写，在不同时间编写，益处多多！

私有变量的使用限于函数之内，通称为“内部”（Internal 或 Local）变量。公共变量则可让各个函数共享，通称为“外部”（External 或 Global）变量。根据使用权限，变量可分为三类，如表 8-1 所示。

表 8-1

种类	使用权限范围
自动（Automatic） 变量	隶属于一个函数，别的函数无权使用
外部（External）变量	程序（常由数个源代码文件所组成）由各函数公用
外部静态（External Static）变量	一个源代码文件（*.c），该文件内的各函数公用，但其他文件内的函数无权使用

以上说明计算机根据使用权限、生命期及储存位置来对变量进行分类，目的是追求团队合

作，高效使用内存空间及提升执行速度。这种分类通称为变量的“储存种类”（Storage Classes）。

8.2 自动变量

自动变量（Automatic Variable）是内部变量，必须声明于函数之内。

其声明格式为：

auto 数据类型 变量名称

↑
└··（这里 auto 可省略）

您已使用过许多自动变量了。请看下面的例子：

```
int main(void)
{
  int x, y;
  float score;
  x = y = score = 12.5;
  printf( "%d, %d, %.2f", x, y, score );
  return 0;
}
```

x、y 和 score 变量声明于 main()的大括号——{}内，隶属于 main()的自动变量。

8.3 外部变量

函数内的变量为私有变量，别的函数无法取得该变量的值。如果数个函数要共享同一变量的值，就必须声明于函数外了。如果变量声明于函数之外，就可供各个函数共享。这种共享的变量通称为外部变量（External Variable）。请看下面的例子：

```
float rate=0.06;

float saving( int money )
{ return (money * rate);  }

int main(void)
{
  float interest;
  interest = saving( 10000 );
  printf( "Income = %.2f With rate = %.2f\n",
                          interest, rate );
  interest = saving( 22000 );
  printf( "Income = %.2f With rate = %.2f\n",
          nterest, rate );
  return 0;
}
```

interest 变量声明于 main()内，是 main()的自动变量；money 声明于 saving()内，是 saving()

的自动变量。rate 变量声明于函数之外，是外部变量，各函数可共享。外部变量是程序开始执行时就生成了的，一直到程序结束时才消失。此程序一开始，就生成外部变量 rate，且给予初值 0.06。接着，进入 main()函数，并立即产生自动变量——第 1 次调用 saving()函数时，一进入 saving()就生成自动变量 money，且把 main()传来的值 10000 存入 money 变量中。

此时正在执行 saving()函数，main()的 interest 变量正在冬眠中，所以 saving()不可以使用 interest，但可使用 money 及 rate。saving()计算利息所得 600.0，且将之送回主函数。执行完 saving()后，自动变量 money 就消失了。printf()输出 interest 及 rate 的值 Income=600.00 With rate=0.06。

第 2 次调用 saving()时，再度生成 money 变量，且把 main()传来的值 22000 存入 money 变量。此时，interest 变量再度冬眠了，其值 600.0 仍在。接着，saving()计算所得——22000 * 0.06 = 1320.0，且把 1320.0 送回 main()。此时，money 变量不见了，而 interest 值改变了。

接着 printf()输出 interest 及 rate 的值 Income = 1320.00 With rate = 0.06。此刻，main()已执行完毕，立即除去自动变量 interest，离开了 main()。除去外部变量 rate，就返回 DOS（或 Windows）中去了。归纳上述的说明：

- 外部变量的生命期和程序一样长。
- 外部变量不会冬眠，各函数随时可使用。
- 自动变量在需要时才生成，只供暂时储存数据，用完即丢。

8.4 外部静态变量

外部变量犹如家庭的公用“钱包”，内部变量犹如个人的“私有”钱包，如图 8-1 所示。

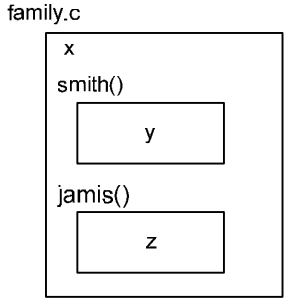


图 8-1

family.c 如同一个家庭，smith()及 jamis()是家人。自动变量 y 是 smith()的私人钱包，自动变量 z 是 jamis()的私人钱包，外部变量 x 是他们两人的公用钱包。家庭要成长为大家庭，常须设立大家庭的公用钱包。这个大家庭就是“程序”（Program），如图 8-2 所示。

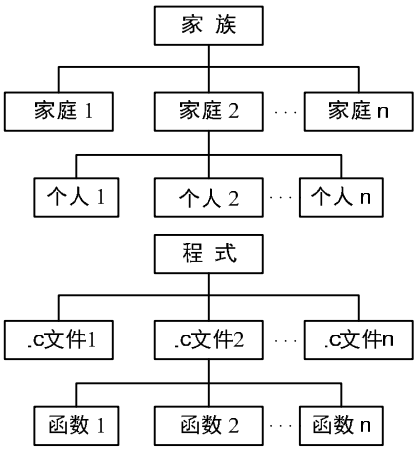


图 8-2

例如 project1.exe，如图 8-3 所示。

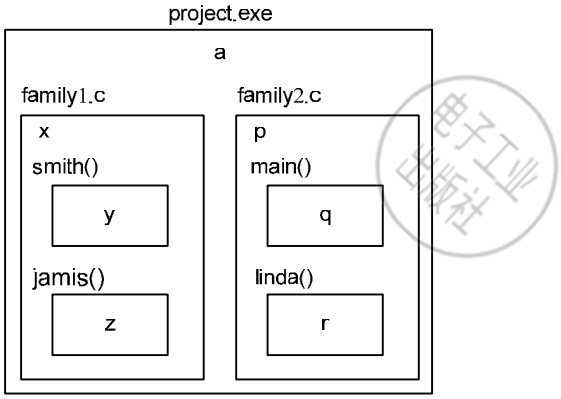


图 8-3

变量 a 是家族的公用变量，大家族中每个人皆可共享。变量 x 是 family1 家庭的公用变量，只有此家庭的成员 smith()及 jamis()才可共享，不与其他家庭公用。同样地，变量 p 是 family2 家庭的公用变量，只有 main()及 linda()可享用。

变量 a 、x 及 p 声明于函数之外，它们皆是外部变量；可是使用权限范围却有所不同。外部变量分为两类：

- 外部变量（External）是大家族的公用变量。
- 外部静态变量（External Static）——是家庭内的公用变量，只供同一源代码文件内的函数共享，其他文件内的函数不得存取。

创建多文件的程序是容易的事。其过程为：

Step-1 创建一个 C 程序项目(Project),例如创建一个 VC++或 TurboC 的项目:Prj-cx08.prj。

Step-2 为此项目创建并编写各源代码文件,例如有两个 C 原始文件(cx08-fa1.c 和 cx08-fa2.c)。内容分别如下:

```
/* cx08-fa1.c  */
#include <stdio.h>

static int numb1;

void smith();
int  jamis();

void linda()
{  printf( "SUM=%d", numb1 );  }

int main(void)
{
  smith();
  numb1 = jamis();
  linda();
  fflush(stdin);
  getchar();
  return 0;
}
```

以及:

```
/*  cx08-fa2.c  */
#include <stdio.h>

static int numb1, numb2;

void smith()
{
  puts( "Type In Two Numbers:" );
  scanf( "%d,%d", &numb1, &numb2 );
}

int jamis()
{
  int sum = numb1 + numb2;
  return sum;
}
```

Step-3 进行“编译”、“连接”,就可产生可执行程序 Proj-cx08.exe。其执行结果如下所示。

```
Type In Two Numbers:
760,120 ←┘
SUM=880
```

请看看上述两个源代码文件吧!cx08-fa2.c 含两个外部静态变量 numb1 及 numb2。其声明格式与内部静态变量完全一样,只是声明位置不同而已。cx08-fa1.c 含一个外部静态变量

numb1，它是 cx08-fa1.c 的私有变量，与外界无关联，不会与 cx08-fa2.cpp 内的 numb1 变量冲突。

现在来比较一下自动变量与外部静态变量吧！自动变量局限于（Local to）函数内，不受别的函数影响，也不影响别的函数的变量。至于外部静态变量则局限于（Local to）程序文件，不受别的程序文件内的函数影响，也不影响其他原始文件内的变量。

请想一想，假设您写了上述 cx08-fa2.c 程序文件，且赠送给了您的朋友；而他写了 cx08-fa2.c 程序文件，其 main()调用您的 smith()及 jamis()。由于您不允许他的函数破坏您的 numb1 及 numb2 变量，就将 numb1 和 numb2 声明为外部静态变量了。numb1 及 numb2 只是当做 smith()与 jamis()沟通的媒介而已。请注意，static 关键字的使用规则如下：

- 若想延长内部变量的生命期，就在内部变量前加上 static。
- 若想把外部变量的高效范围局限于所属的原始文件内，就在外部变量前加上 static。

8.5 extern 种类

当一个程序文件欲使用别的程序文件的外部变量时，应使用 extern 关键字。

现在来改写上一节里的 Prj-cx08.prj 项目内容，成为一个新项目 P2-cx08.prj。其中原来的 cx08-fa1.c 改写为 cx08-pa1.c：

```
/* cx08-pa1.c */
/* External storage class */
#include <stdio.h>

extern int sum;

void smith();
void jamis();

void linda()
{  printf( "SUM=%d", sum );  }

int main(void)
{
  smith();
  jamis();
  linda();

  fflush(stdin);
  getchar();
  return 0;
}
```

并且把原来的 cx08-fa2.c 改写为 cx08-pa2.c：

```
/* cx08-pa2.c */
/* External storage class */
#include <stdio.h>
int sum;
static int numb1, numb2;

void smith()
{
  puts( "Type In Two Numbers:" );
  scanf( "%d,%d", &numb1, &numb2 );
}

void jamis()
{
  sum = numb1 + numb2;
}
```

这个程序文件将借外部变量 sum 来传递数据给主函数。因 sum 是公共变量，所以其他的文件可以共享。也就是说，cx08-pa1.c 可共享 sum 的值。

“extern”是 foreign（外来）之意。其格式为：

```
extern  类型  变量名称
  ↑
  └‥‥（意味：去共享别的文件的外部变量）
```

经编译之后，在连接时，计算机把两文件的 sum 看成同一变量，所以可执行程序 P2-cx08.exe 中只含一个 sum 变量。cx08-pa1.c 的 sum 就是 cx08-pa2.c 的 sum。于是，各函数可借 sum 沟通了。程序的结果为：

```
Type In Two Numbers:
760,120 ←┘
SUM=880
```

请再看个例子，项目 P3-cx08.prj 含三个源代码文件：

第一个源代码文件

```
/* cx08-ta1.c */
/* External storage class */
#include <stdio.h>
#include <string.h>

void update();
void count();
extern int upr, lwr;
char *ps = "Cute";

int main(void)
{
  update();
  count();
```

```
  printf( "upr=%d, lwr=%d", upr, lwr );
  fflush(stdin);
  getchar();
  return 0;
}
```

第二个源代码文件

```
/* cx08-pa2.c */
/* External storage class */
#include <stdio.h>
#include <string.h>

extern char *ps;
void update()
{
  char bb[10];
  bb[0] = ps[3];
  bb[1] = ps[2];
  bb[2] = ps[1];
  bb[3] = ps[0];
  bb[4] = '\0';
  puts(bb);
}
```

第三个源代码文件

```
*  cx08-ta3.cpp  */
/*  External Variable  */

#include <stdio.h>
#include <string.h>
#include <ctype.h>

int upr, lwr;
extern char *ps;

void count()
{
  char *pk;
  upr = lwr = 0;
  pk = ps;
  while( *pk != '\0' )    {
        if( islower( *pk++ ) )  lwr++;
        else  upr++;
 }
}
```


此程序输出 etuC，如下：

```
upr=1, lwr=3
```

此程序把字符串“Cute”转为反序字符串“etuC”，且计算大写字母的个数，也计算小写字母的个数。计算机知道 cx08-ta1.c 欲共享 cx08-ta3.c 中的 upr 及 lwr 变量。也知道 cx08-ta2.c 及 cx08-ta3.c 欲共享 cx08-ta1.c 中的 ps 指针变量。这样，即可沟通无阻了。

8.6 静态函数

不让其他文件共享的变量叫“外部静态”。不让其他文件来调用的函数称为“静态函数”（Static Function）。其声明格式为：

```
static  类型  函数名称
                {
                  .....
                }
```

请看下面例子，P4-cx08.prj 项目含有三个原始文件，如下：

第一个原始文件

```
/* cx08-ca1.c */
#include <stdio.h>

float cylinder( float h, float r );
float area( float x );
float volume( float h, float x );

int main(void)
{
  printf( "Area=%.2f\n", area( 10.0 ) );
  printf( "Volume=%.2f\n", volume(4, 20.0) );
  printf( "Cylinder=%.2f\n", cylinder(5.0, 10.0) );

  fflush(stdin);
  getchar();
  return 0;
}
```

第二个原始文件

```
//* cx08-ca2.c */
/* Static functions */
#include <stdio.h>
#define  PI  3.1416

static float area( float r )
{  return (PI*r*r);  }

float cylinder( float h, float r)
{  return (h * area( r ) );  }
```

area()计算圆的面积，为该原始文件的专用函数，此文件内的函数能调用它，其他文件的函数则不能调用它。cylinder()函数未加上“static”字眼，表示程序中的各个函数皆可调用它。

第三个原始文件

```
/* cx08-ca3.c */
/* static function */
#include <stdio.h>
```

```
float area( float x )
{ return (x * x);   }

float volume( float h, float x )
{ return (h * area( x ) ); }
```

这个 area()计算正方形的面积，volume()计算正立方体的面积。此程序含两个 area()函数，请观察计算机如何挑选它们。计算机执行 main()函数前，先调用 area()函数。cx08-ca2.c 内的 area()附有 static 字眼，main()无权使用它，于是 main()调用 cx08-ca3.c 内的 area()函数。同理，volume()也调用 cx08-ca3.c 内的 area()函数。

cylinder()会调用哪个 area()函数呢？当公用函数与私有函数名称相同时，私有函数优先，于是 cylinder()调用 cx08-ca2.c 的 area()函数。经编译和连接得到 P4-cx08.exe 程序就可以执行了，其结果为：

```
Area=100.00
Volume=1600.00
Cylinder=1570.80
```

第 9 章　数组与字符串

9.1 数组的意义

开学时，常填写课程表；到邮局划拨订杂志时，常填写划拨单。无论是填写课程表还是填写划拨单，皆把数据填入“表格”（Table）中。人们使用表格来记载数据，计算机则常使用“数组”（Array） 储存数据。

课程表是表格，垂直方向分为数“列”（Row），水平方向分为数“行”（Column）。在日常生活中，常见各式各样的表格，例如制作一份月历，如图 9-1 所示。

十 月

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

图 9-1

这是表格，依照垂直及水平方向分为“列”及“行”。此为 2 维空间表示法。在计算机中，用来储存这种数据的数组，称为“2 维数组”（2-Dimensional Array）。那么，如何将整个月历存入计算机呢？答案是：须使用“3 维数组”（3-Dimensional Array）来储存。除了表格外，数学上常用“矩阵”（Matrix）表示有次序的数据，例如：

$$A = \begin{pmatrix} 3 & -2 \\ 7 & 8 \end{pmatrix}$$

$$B = (1\,3\,5,\ -6,\ 5.6)$$

计算机的 2 维数组能存储矩阵 A 的数据，1 维数组则能存储向量 B 的数据。除了上述的 1 维、2 维及 3 维数组外，还有 4，5，…，N 维数组，应有尽有。写 C 程序时，应善用数组来储存有组织有顺序的数据。

9.2 1 维数组

声明变量时，计算机分配空间给变量，这样变量就能储存数据了。一个变量只能储存一项数据（整数、浮点数或字符等）。“数组”（Array） 也必须声明要求计算机分配空间，这样

数组就能储存表格或向量数据了。例如，欲将 Art 及 Science 两个科目的成绩存入计算机中，并求平均分数。可写程序代码如下：

```
float course[2];
course[0] = 98.5;
course[1] = 88.0;
printf( "AVERAGE=%.1f", (course[0] + course[1])/2 );
```

course[]是数组，此时，course[]数组拥有 8 个 Bytes 空间，如图 9-2 所示。

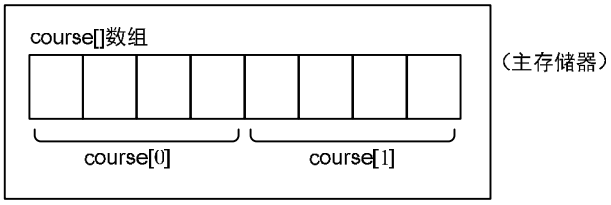


图 9-2

其中，couese[0]代表左边 4 Bytes 空间，course[1]代表右边 4 Bytes 空间。course[0] 及 course[1]是 course[]数组的“元素”（Element）。course 数组为 float 类型，意味着 course[0]及 course[1]两个元素皆为 float 类型，所以 course[0]及 course[1]各占 4 Bytes 空间。而且 course[0]与 course[1]的空间紧密相连在一起。因此，数组数据井然有序地储存于计算机中。此时 course[]值为（如图 9-3 所示）：

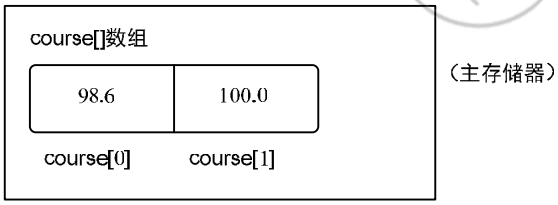


图 9-3

最后，course[0]加上 course[1]并计算平均数。

但是，请留意数组有两项特性：

- 每一元素的类型均相同。
- 借“索引”（Subscript）来区别元素，且最小索引值为 0。

9.3 1 维数组与指针

数组是由一串变量（元素）所组成的。因此，指针能指向数组内的任一元素。即，可借指针来存取数组内的元素。所以，除了传统的数组存取方法之外，还可利用更高效的指针处

理方法。如何令指针指向数组的元素呢？请看下面的例子。

```
int data[3];
int *px, *py;
data[0] = 5;
data[1] = 28;
data[2] = 78;
px = &data[0];
py = &data[1];
data[2] += *px + (*py)++;
printf("%d, %d, %d\n", data[0], data[1], data[2]);
```

data[]为整数数组，px 及 py 为整数指针。指令 px = &data[0]是表示 px 指向元素 data[0]。既然 px 指向 data[0]，就可用*px 来代替 data[0]，亦即可用*px 来存取 data[0] 的内容了。

同理，指令 py = &data[1]令 py 指向 data[1]元素，此时可用*py 代替 data[1]，亦即可用*py 存取 data[1]的内容了，如图 9-4 所示。

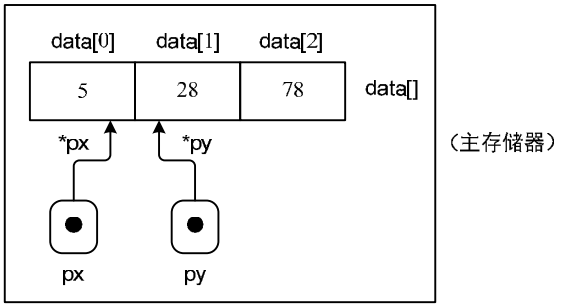


图 9-4

接下来，指令 data[2] += *px + (*py)++运算之后，data[]数组的内容变为如图 9-5 所示：

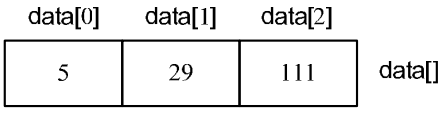


图 9-5

数组含有很多个元素，这些元素储存于内存中，且相连在一起。px 指针能指向 data[0]，也可指向下一个元素 data[1]。令 px 指向 data[1] 的方法有以下两种。

- 使用指令—— px = &data[1]
  这是您所熟悉的方式。

- 使用指令——
  px = &data[0]
  px++

  px++令 px 指向下一个元素。借此方法，就可令 px 依序指向数组的任一元素了。

指针的移动方法是基本技巧，愈熟练便愈能写出高效率的程序。请您看下面一个数组的例子，其内容如图 9-6 所示。

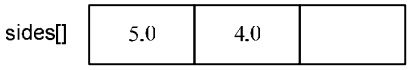


图 9-6

这是直角三角形的边长：斜边长为 5.0，另一边边长为 4.0，第三边长度未知。可写一个程序来计算第三边的长度，如下：

```
double sides[3];
double *k1, *k2, *k3;

sides[0] = 5.0;
sides[1] = 4.0;
k1 = &sides[0];
k2 = k1 + 1;
k3 = k1 + 2;
*k3 = sqrt( pow( *k1, 2.0 ) - pow( *k2, 2.0 ));
printf( "the 3rd side is: %.1f", *k3 );
```

指 k1=&sides[0]让 k1 指向 sides[0]。指令 k2=k1+1 让 k2 指向 k1 所指元素的下一个元素，即 k2 指向 sides[1]。指令 k3=k1+2 让 k3 指向 k1 所指元素的下两个元素，即 k3 指向 sides[2]，如图 9-7 所示。

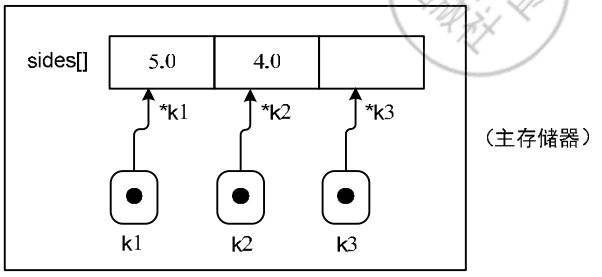


图 9-7

pow()函数用来计算次方值，例如：要计算 2 的 3 次方，写为 pow(2.0, 3.0)就可求出其值 8.03。sqrt()函数用来计算平方根，例如：要计算 2 的平方根，可写为 sqrt(2.0)就可算出其值 1.414。

目前，*k1 的值为 5.0，*k2 的值为 4.0。

所以，指令 *k3 = sqrt(pow(*k1, 2.0)-pow(*k2, 2.0))

相当于 *k3 = sqrt(pow(5.0, 2.0)-pow(4.0, 2.0))

sqrt()求得平方根 3.0 存入*k3 的位置中，使得 sides[]数组的内容如图 9-8 所示。

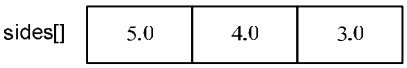


图 9-8

printf()将*k3 值显示出来 the 3rd side is: 3.0。请注意：sqrt()及 pow() 函数皆定义于 math.h 头文件，所以程序应加上指示：

```
#include <math.h>
```

这个程序使用了 k1、k2 及 k3 共三个指针，实在太浪费了。如何才能较为经济呢？想一想，当计算机执行到指令 k2 = k1 + 1 时，k2 值等于 k1+1 的值，亦即 k2 指向 sides[1]，同时 k1+1 也指向 sides[1]，如图 9-9 所示。

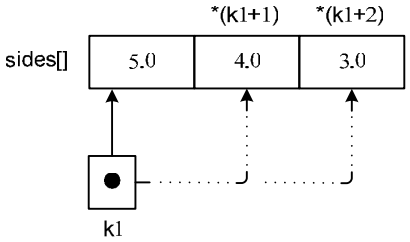


图 9-9

因此， *k2 ≡ *(k1+1) ≡ sides[1]

同理， *k3 ≡ *(k1+2) ≡ sides[2]

*(k1+1)代表着*k1 之后的第一个元素

*(k1+2)代表着*k1 之后的第二个元素

⋮

*(k1+n)代表着*k1 之后的第 n 个元素

例如，当 p 指向 x[]时，x[i] ≡ *(p+i)，如图 9-10 所示。

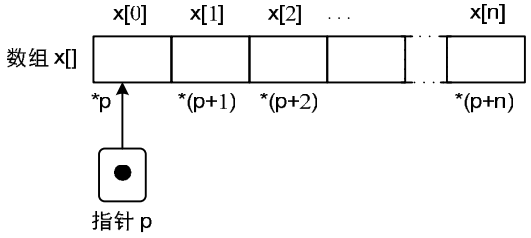


图 9-10

在 C 程序中，数组名有其特殊又重要的用法。对于初学者，刚开始会感到有点莫名其妙。

请回忆下列指令：

```
......
char a[4], *p;
p = &a[0];
......
```

这些指令让指针 p 指向 a[] 数组的开头，就能使用*(p+1)代替 a[1]、以*(p+2)代替 a[2]，等等，如图 9-11 所示。

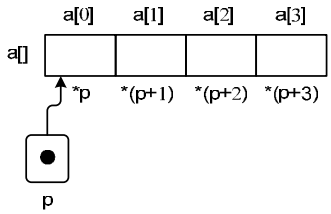


图 9-11

重要的观念是，数组的名称 a 永远代表着&a[0] 的值。

因此，指令　　p = &a[0];

相当于　　p = a;

由于 a 值等于 p 值，可推知，如图 9-12 所示。

*(p+0) 相等于 *(a+0)

*(p+1) 相等于 *(a+1)

*(p+2) 相等于 *(a+2)

…

*(p+n) 相等于 *(a+n)

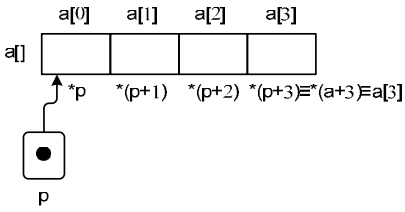


图 9-12

在 C 程序中，下列规则永远成立：

&a[0] ≡ a

指令　　p = a;

相当于　　p = &a[0];

令 p 指向 a[] 数组，即让 p 指向 a[0]元素。接下来，while 循环显示出 a[]数组 JCN。虽然，指令：

```
p = a;
```

令 p 值等于 a 值，但 p 与 a 有个重大区别：“p 是指针变量；但 a 是指针常数”

由于 a 是常数，其值永远固定不变，所以 a[i]永远相当于*(a+i)。然而，p 是变量，其值会改变，因而 *(p+i) 未必等于 a[i]。请特别留心，在什么情况下，*(p+i) 才会等于 a[i]呢？答案是：如果 p 正指向 a[0]，则*(p+i)相当于 a[i]。

所以，归纳以下简单规则：

- 如果 a 是常数，则 a[i]永远等于*(a+i)。
- 如果 p 是指针（变量），则 a[i]未必等于*(p+i)；仅当 p 正指向 a[0]时，a[i]才等于*(p+i)。

9.4 2 维数组与多维数组

1 维数组使用一个索引，2 维数组使用两个索引。2 维数组常表示为 n[][]。例如，声明 n[][]数组如下：

```
int n[2][12];
```

这个指令说明了此数组为 int 类型，即每个元素皆为 int 类型。

如果 2005 年及 2006 年的前 3 个月销售量为（如图 9-13 所示）：

	一	二	三	（月份）
2005	50	58	45	
2006	70	86	85	

图 9-13

将表内数据存入[][]中，就能计算销售量的增加额，如下所示：

```
int n[2][12],i,k;
n[0][0]=50;
n[0][1]=58;
n[0][2]=45;
n[1][0]=70;
n[1][1]=86;
n[1][2]=85;
for(i=0; i<3; i++)
   {
    k = n[1][i] - n[0][i];
    printf( "Month No. %d is: %d\n", i+1, k );
   }
```

此程序的 for 循环共重复三次。第一次的 i 值为 0，表达式 k= n[1][i] - n[0][i] 得 k 值为 20，表示 2006 年 1 月份的销量比 2005 年 1 月份增加 20 台。同样，第二次及第三次循环，分别算出 2 月份及 3 月份的增加数量。

以上介绍了 2 维数组，就可以类推到多维数组了。多维数组包括 3 维数组、4 维数组……及多维数组。如果您将 2 维数组视为 1 维数组的组合，就可类推得知 3 维数组就是为 2 维数组的组合，更可类推到 4 维、5 维等多维数组。

9.5 2 维数组与指针

x[i]是传统数组表示法，x 是数组名，i 是索引（Index）。编译时，传统表示法皆转换为指针表示法。例如：x[i]转换为*(x+i)。在此，请复习指针和数组的恒等式：

```
x[i]  ≡ *(x+i)
```

欲存取 x 数组的第 i 个变量（元素），可写为 x[i]或*(x+i)，两者皆可。通常后者的执行速度快，而前者令程序平易近人。同时，再复习另一个重要观念：

“若 p 是指针，p = &x[0] 令 p 指向 x[0]，就称 p 指向 x 数组”

因为&x[0] ≡ x

所以 p=&x[0] 相当于 p=x。

现在，您已经熟悉指针与数组的恒等式：

```
p[i]  ≡ *(p+i)
```

依此类推，2 维数组与指针的恒等式：

```
p[i][j]  ≡  *(p[i]+j)  ≡  *(*(p+i)+j)
```

这是 1 维数组的延伸。请看传统 2 维数组的格式：

```
void proc( char p[][5] )
{ putchar( p[0][2] );
  putchar( '\n' );
  putchar( p[1][3] );
}

int main(void)
{
  char x[2][5];
  strcpy( x[0],  "BEAR" );
  strcpy( x[1],  "LION" );
  proc( x );
  return 0;
}
```

x[][]是 2 维数组，x[0]和 x[1]为 1 维数组。strcpy()分别将“BEAR”和“LION”字符串存入 x[0]和 x[1]数组中。proc(x)使得 p 和 x 完全等价，如图 9-14 所示。

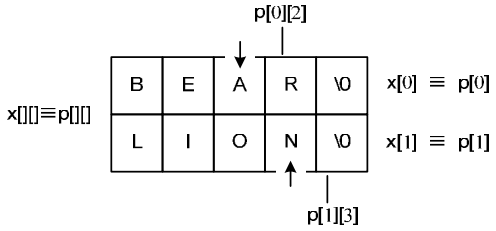


图 9-14

由于 p[0][2] ≡ *(p[0]+2) ≡ *(*(p+0)+2)

而且 p[1][3] ≡ *((p+1)+3)

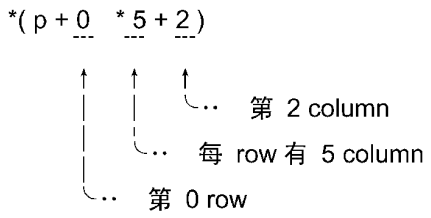
则上述程序相当于：

```
void proc( char p[][5] )
{
  putchar( *(*(p+0)+2) );
  putchar( '\n' );
  putchar( *(*(p+1)+3) );
}

int main(void)
{
  char x[2][5];
  strcpy( x[0],  "BEAR" );
  strcpy( x[1],  "LION" );
  proc( x );
  return 0;
}
```

请再看个例子：

```
void proc( char *p )
{
  putchar( *(p+0*5+2) );
  putchar( '\n' );
  putchar( *(p+1*5+3) );
}

int main(void)
{
  char x[2][5];
  strcpy( x[0], "BEAR" );
  strcpy( x[1], "LION" );
  proc( &x[0][0] );
  return 0;
}
```

指令 proc(&x[0][0])表示 p 指向 x[][] 数组。表达式：



因此，*(p+0*5+2) ≡ x[0][2]，而*(p+1*5+3) ≡ x[1][3]。

9.6 数组参数

9.6.1 1 维数组参数

主函数可以将数组元素逐一传给子函数，例如：

```
long toto( long x[] )
{ x[0] = x[0] + x[1];
  return (x[0]);
}

int main(void)
{
  long data[2], sum;
  data[0] = 68000;
  data[1] = 5000;

  sum = toto( data );
  printf( "data[0] = %ld\n", data[0] );
  printf( "data[1] = %ld\n", data[1] );
  printf( "sum = %ld\n", sum );
  return 0;
}
```


data[]对应参数 x[]，类型皆为 long 数组，满足了“对应参数的类型必须一致”的规则。子函数的 long x[]，其[]内是空的，表示 x[]数组的大小与 data[]一样。更重要的是：它们不仅大小一样，而且 x[]与 data[]就是同一数组，只是名称不同而已，主函数称它为 data[]，子函数则称它为 x[]。data[0] 与 x[0]代表同一空间，所以子函数内更改了 x[0]，也等于更改了主函数的 data[0]值，这是数组参数的重要特性，希望您能充分了解它。计算机执行 sum=toto(dist)指令之前，data[]的内容如图 9-15 所示。

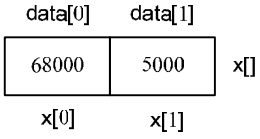


图 9-15

调用 toto()时，您可能认为 main()将 data[]内的各元素逐一传给了 toto()的 x[] 数组。其实

计算机偷懒了，根本没花时间做这事，因为 data[] 与 x[]代表同一块区域，何必再传递呢？

下面归纳了传递变量与传递数组的区别。

- 变量：真正传递变量的值，得花费时间，但能使主、子函数之间充分独立，有益于软件设计的分工合作。
- 数组：利用参数占同一空间的特性，节省传递各元素值的时间，可以加快程序速度，但相对应的参数会互相影响。

何以如此对待数组呢？原因是避免因数组元素多而浪费传递数据的时间。请想想，return 指令每次只送一个值回主函数，如果子函数欲将整个数组送回主函数时，应如何处理呢？此乃重要概念。

9.6.2 2 维数组参数

主函数能将整个 2 维数组传递给子函数，此时子函数也得声明对应的 2 维数组参数才行。例如：欲计算 2005 年及 2006 年前 3 个月的平均销售量，则程序可写为：

```
#define Row    2
#define Column  12

void average( int k[Row][Column] )
{
  int r, c, sum;
  for( r=0; r<Row; r++ )
      {
         for( c=sum=0; c<3; c++ )
             sum += k[r][c];
             printf( "%d ==> %.1f\n", 2005+r, sum / 3.0 );
      }
}

int main(void)
{
  int n[Row][Column];
  n[0][0]=50;
  n[0][1]=58;
  n[0][2]=45;
  n[1][0]=70;
  n[1][1]=86;
  n[1][2]=85;
  average( n );
  return 0;
}
```

main()内的 n 是 int 类型的 2 维数组。average() 内的对应参数 k 也是 int 类型的 2 维数组，它们的类型一致。

由于 n[][]与 k[][]为对应数组参数，所以 n[][]与 k[][]代表同一数组，其大小当然相同。例如，k 与 n 皆有 Row 行和 Column 列。然而，子函数的 2 维数组参数，可省略左边维度 Row，

亦即参数声明 int k[Row][Column]，可写为 int k[][Column]。但是不可省略 Column。

9.7 为数组赋初值

数组生成时能赋初值，例如：

```
int price[3]={ 15, 20, 31 };
```

初值摆在大括号{}内，且用逗号隔开各初值，如图 9-16 所示。

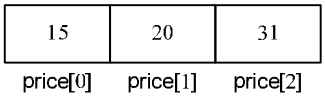


图 9-16

若只赋给部分初值，则未赋值的元素计算机会自动赋 0（即每 bit 皆为 0）。万一赋给了太多值，又如何呢？例如：

```
int  z[3] = { 8, 6, 45, 3};
```

计算机会输出错误信息 Too many initializers。原因是 z[]只含三个元素，但却赋给了四个值。此时，有个方便的方法，如下：

```
int z[]={ 8, 6, 45, 3 };
```

未说明 z[]数组到底含有多少元素，计算机会自动依大括号{}内的初值个数决定。{8,6,45,3}含四个值，就自动把 int z[]视为 int z[4]。于是 z[]含四个元素。

除了 1 维数组外，也能赋给 2 维数组初值；只要把 2 维数组看成 1 维数组的组合，就知道如何赋初值了。例如：

```
float np[2][3]= { {56.1, 25.2, 80.3},
                  {105.5, 75.25, 31.88}};
```

np[][]数组含 np[0][]及 np[1][]两个 1 维数组。初值取出两个一维的初值，分别赋给 np[0][]及 np[1][]，如图 9-17 所示。

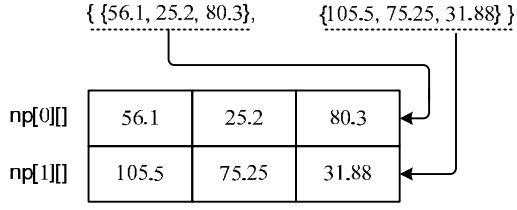


图 9-17

9.8 使用字符串

9.8.1 何谓字符串

“字符串”是一串有序的字符。例如："New-Year"，"Sea"，"BEAUTIFUL" 等皆是字符串。“有序”意指如果字符串内的字符顺序改变了，就变成另一字符串。例如："ARE" 和"EAR" 是不同的字符串。在 C 程序中，必须用双引号——" " 把字符串括起来。当储存字符串时，应该把字符串内的字符逐一储存起来。C 语言的字符串必须储存于“字符数组”中。例如：

```
char s[4];
strcpy( s, "Sea" );
printf( s );
```

strcpy()将"Sea"字符串存入 s 字符数组中。s[]的内容就是"Sea"字符串。printf(s)则输出 s[]的内容。strcpy()定义于 string.h 头文件中，所以当使用 strcpy()时，必须加上#include <string.h>才行。

9.8.2 给予字符串初值

生成字符数组时，可立即把字符串存入数组中，称为字符串的初值设定（Initializing String）。例如：

```
char line[] = "HI!";
puts( line );
```

这是最常用的字符串初值设定方法。当生成 line[]时，顺便把"HI!"字符串存入 line[]中，如图 9-18 所示。

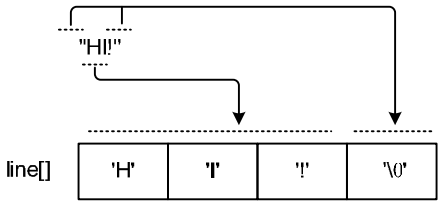


图 9-18

您也可设定 2 维字符数组的初值。由于 2 维数组就是 1 维数组的组合，依此类推就能设定其初值了。例如：

```
char str[4][5] = { "DOG", "Y\nZ", "LION", "" };
puts( str[0] );
puts( str[1] );
puts( str[2] );
puts( str[3] );
```

str[][]数组的内容如图 9-19 所示。

""表示空字符串（Null String），意思是此字符串只含字符串尾部——'\0'字符。

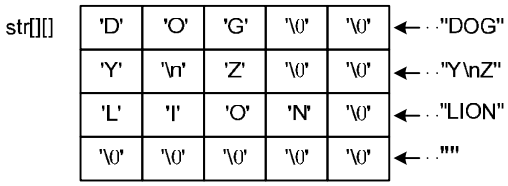


图 9-19

9.9 库字符串函数

上节的 strcpy()和 puts()函数，为 C 提供了字符串函数。此外，它还提供了许多字符串函数，大多定义于 stdio.h 及 string.h 头文件内，所以程序应加上：

```
#include <stdio.h>
#include <string.h>
```

常用字符串函数包括，如表 9-1 所示。

表 9-1

函数	用途
strcpy(s1,s2)	将字符串 s2 拷贝（Copy）到 s1 字符串
puts(s)	把字符串 s 显示出来
strlen(s)	计算字符串 s 的长度
scanf("%s", s)	由键盘输入一字符串
gets(s)	由键盘输入一字符串
strcmp(s1, s2)	比较二字符串，在“小于”、“等于”及“大于”情形下，分别传回-1、0 及 1
strcat(s1, s2)	把 s2 连接于 s1 的尾部
strupr(s)	把字符串 s 内的小写字符转换为大写字符
strlwr(s)	把字符串 s 内的大写字符转换为小写字符
sscanf()	从一字符串（字符数组）做格式化输入
sprintf()	把格式化的数据（字符串）存入字符数组中

另外，stdlib.h 头文件也定义了一些常用字符串函数，如表 9-2 所示。

表 9-2

函数	用途
atoi()	将一字符串转换为整数
atol()	将一字符串转换为长整数
atof()	将一字符串转换为浮点数
itoa()	将一整数转换为字符串
ltoa()	将一长整数转换为字符串

说明如下：

strlen()——算出字符串的长度。例如：

```
int k;
char y[] = "Orange";
k = strlen( y );
printf( "%d  ", k );
```

strlen()计算出字符串内含多少个字符；但不包括字尾的'\0'字符。

生成字符数组 y[]时，立即把"Orange"字符串存入 y[]。strlen(y)计算 y[]内的字符串长度（共含多少字符）。得 k 值为 6。

strcmp() ——比较两字符串的大小，也能比较两字符串是否相等。

例如，空字符串的长度为 0，您刚才借字符串长度判断字符串是否为空字符串。

strcmp()函数也能担任这项判断。strcmp()是比较字符的 ASC II 值，例如'A' 的 ASC II 值是 65，而'B'的 ASC II 值是 66，所以'B'大于'A'。依此类推，可知：

```
'\0' < 空格符 < 0 < 1 < …… < 9 < A < B < ……
< Z < a < b < …… < z < 特殊符号（如'*', '#）
```

strcmp()将传回
- 1 ……若字符串 1 大于字符串 2
- 0 ……若字符串 1 等于字符串 2
- -1 ……若字符串 1 小于字符串 2

strcat()——连接两个字符串。

请先回忆 strcpy()函数，把一个字符串存入数组中。例如：

```
char sx[10];
char sy[10];
strcpy( sx, "APPLE" );
strcpy( sy, "IBM" );
strcat( sx, sy );
```

于是，sx[]的内容变成"APPLEIBM"字符串，这是由"APPLE"与"IBM"连接而成的新字符串。

sprintf() ——把数个变量值组成字符串。您熟悉的 printf()将组成的字符串送往显示器，而 sprintf()把组成的字符串储存到字符数组中。例如：

```
float sum  = 198.6;
char ss[10];
sprintf( ss, "sum=%.1f", sum );
```

sum 值为 198.6。sprintf()把组合而成的字符串"sum=198.6" 存入 ss[]中，得 ss[]内容如图 9-20 所示。

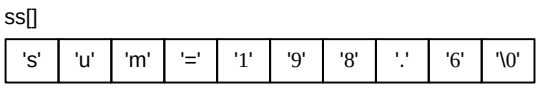


图 9-20

9.10 传递字符串参数

C 并没有提供“字符串变量”，因而字符串必须存于数组中。当字符串"ABC"存于计算机时，其'A' 、'B' 、'C' 及'\0' 4 等个字符分别存于相连的 Bytes 里。因此，可把"ABC" 字符串看成一个数组。前面，已介绍过如何传递数组给子函数。字符串就存于数组中，将字符串传递给子函数的过程与传递数组是一样的。请看下面的例子：

```
void sub( char data[] )
{
  puts(data);
}

int main(void)
{
  sub( "Horse" );
  return 0;
}
```

应用上，可认为 sub("Horse")指令将"Horse"字符串传给 data[]数组，所以 data[]的内容是"Horse" 字符串。同时，可进一步了解：是计算机让"Horse"字符串（本质上是数组）和 data[]重合在一起的，如图 9-21 所示。

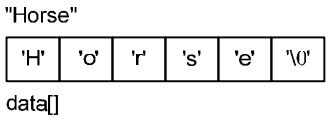


图 9-21

于是将字符串传递给 data[]了。

第 10 章　预处理程序

10.1 预处理程序的工作

预处理程序（Preprocessor）又称为预处理器，它的任务是在编译（Compile）之前把源代码先整合，再交给编译器（Compiler）处理。由于整合工作是在编译之前做，所以称为预处理。预处理的用途有以下三种。

- 处理宏常量（Macro Constant）及宏函数（Macro Function）。

“宏”是原始文件中某一小段指令的名称。当编写程序时，只需写上名称，预处理程序便会以真实指令代换宏名称，使编程与修改工作事半功倍。

- 合并（Include）源文件。

写文章时，可引用别人的文章，也可引用自己写过的文章。编写程序时，也能并入其他源代码文件，共享其程序。

- 条件性编译（Conditional Compilation）。

条件性编译能提高程序的可移植性，使程序适用于不同的计算机系统；但不能使目的程序变大（占较多内存空间）。

10.2 使用宏

人有正名（本名），也有别名，通常别名较亲切。程序内的常数或变量也可有含义深刻的别名。令程序更有人情味，更亲切。

有了别名，就可使用别名，也可用本名。预处理器（Preprocessor）将别名转成正名后，才交由编译器（Compiler）编译。例如，C 程序员常用 PI 作为圆周率 3.14159 的别名，于是可计算面积：

```
area = PI * r * r;
```

其中，r 及 area 是变量，分别代表半径及面积。PI 是常数 3.14159 的别名。PI 比 3.14159 有味道多了，不是吗？

常数的别名如 PI 等，称为“宏常数”；若别名中含有参数（Argument），则类似一般函数，称为“宏函数”。

10.2.1 宏常数

创造宏和使用宏是 C 程序的重要技巧。善于利用宏，可增进程序的韵味、亲切感，大家也乐于阅读您的程序。请看个例子，三个圆的半径分别为 10、15 及 20。用下面这段程序可求各个圆的面积。

```
#include "stdio.h"
#define PI 3.14

int main(void)
{
  float a1, a2, a3;
  a1 = PI * 10 * 10;
  a2 = PI * 15 * 15;
  a3 = PI * 20 * 20;
  printf( "a1=%.2f, a2=%.2f, a3=%.2f", a1, a2, a3 );
  getchar();
  return 0;
}
```

为 3.14 取别名叫 PI，此 PI 是常数而非变量。处理别名是预处理器的工作，因此须利用“指示”（Directive）来说明 PI 是 3.14 的别名。#define 为常用的“指示”（指导预处理器如何做事）。

#define 指示的格式为：

```
#define   别名   常数或符号
----------  ------  --------------
```

习惯上，宏常数皆采用大写字母，使之与变量及函数名称区别。然而，这并非严格规定，只是风俗习惯罢了。此外，#define 指示常位于程序开头位置，如下：

```
#include "stdio.h"
#define Real float
#define real float

int main(void)
{
  Real x=34.5;
  real y=23.7;
  printf( "Real x=%5.2f, real y=%5.2f", x, y );
  getchar();
  return 0;
}
```

这样，就能以 Real 代替 float 了。因 float 有两个别名：Real 及 real，所以 real 也能代替 float。预处理器能把 Real 还原为 float，也能把 real 还原为 float，结果如下：

```
      float x=34.5;
      float y=23.7;
      printf( "Real x=%f, real y=%f", x, y );
      getchar();
      return 0;
```

请注意，双引号（" "）内的 Real 及 real 为数据而非宏，所以不会被取代。必要时，也能为表达式取个别名，但别忘了用小括号把表达式括起来。例如：

```
#include "stdio.h"
#define  MINUS_ONE   (-1)
```

```
#define  Base_Offset (10+5)

int main(void)
{
  int x,y;
  x = - MINUS_ONE;
  y = Base_Offset * 10;
  printf( "%d, %d", x, y );
  getchar();
  return 0;
}
```

–1 是常数，也是表达式，含“–”运算符号和操作数 1，于是必须加上小括号才行。非把表达式括起来不可吗？

请看指令：

```
x = - MINUS_ONE;
```

预处理器将其还原为：

```
x = - (-1);
```

这是对的。假如去掉小括号，如下：

```
#define MINUS_ONE -1
```

则指令 x = - MINUS_ONE 被还原成：

```
x = - -1;
```

这是不正确的。

10.2.2 #define 与 typedef 的区别

typedef 用来替某一变量类型（variable type）取一个新的名称，它是一般指令，而不是宏定义。#define 与 typedef 是不一样的，两者通常不能互换，例如：

```
typedef char* STRING
STRING pc1, pc2;
```

这意味着 pc1 和 pc2 都是 STRING 类型的变量。相当于：

```
char* pc1;
char* pc2;
```

再如：

```
#define STRING char*
```

```
STRING  pc1, pc2;
```

预处理之后，变成：

```
char* pc1, pc2;
```

相当于：

```
char *pc1, pc2;
```

这意味着*pc1 和 pc2 都是 char 类型的变量。

10.2.3 带参数的宏

宏可含参数（Argument），这增加了宏的用途。

文字取代

1. 一般用法

参数被包含于宏名称后的小括号内，但是宏名称与小括号要紧密连接在一起。例如：

```
#define STRUCTURE(type) struct type
```

定义好了，就可写成如下指令：

```
STRUCTURE(Rectangle)
{   double length;
    double width;
}
```

预处理之后变成为：

```
struct Rectangle
{   double length;
    double width;
}
```

2. 跨行的宏

使用“反斜线+ENTER”可将宏定义切分为数行。例如：

```
#define STRUCTURE(type) \
struct type \
{

#define END \
};
```

定义好了，就可写指令如下：

```
STRUCTURE(Rectangle)
  double length;
  double width;
END
```

3. ##的用法

假设有一个宏定义：

```
#define FUNC(type) \
 type * type_function() \
   {
```

如果写为：

```
FUNC(float)
```

预处理之后，变成：

```
float* type_function()
{
```

Type_function 是一个完整的名称，此 type 并不是参数，所以不会被取代。反之将宏改为：

```
#define FUNC(type) \
 type * type##_function() \
   {
```

如果写为：

```
FUNC(int)
```

预处理之后，变成：

```
int* int_function()
{
```

type##表明了此 type 是参数，所以会被取代。

定义数学表达式

定义数学表达式宏时，别忘了用小括号把表达式括起来。例如：

```
#include "stdio.h"
#define fcn(x) (2*(x)*(x)+3*(x)-4)

int main(void)
{
  int x, y;
  x = 3;
  y = fcn( x );
  printf("The value of fcn(x) is %d\n", y);
  y = fcn( x+y );
```

```
  printf("The value of fcn(x) is %d\n", y);
  getchar();
  return 0;
}
```

在宏中，表达式 2*x*x+3*x-4 必须用小括号括住，此外，也必须将 x 参数括起来。因 fcn(x) 是宏函数，程序中的 fcn(x)被还原了。但 printf()的双引号内的 fcn(x) 仍原封未动。原因是计算机不会取代双引号内的字符串，宏函数能含几个参数，而且定义宏函数时，可引用已定义的宏函数，进而创造出复杂的宏函数，例如：

```
#include "stdio.h"
#define max(x,y)     ((x)>(y) ? (x):(y))
#define max3(x,y,z)   max(max(x,y),(z))

int main(void)
{
  int x;
  x = max3(10, 100, 50);
  printf( "max3(10, 100, 50) = %d", x );
  getchar();
  return 0;
}
```

定义 max(x,y) 宏后，max3(x,y,z) 宏能引用它。编写 C 程序时，常会用到下列宏函数：

```
#define  max(x, y)    ((x) > (y) ? (x) : (y))
#define  min(x, y)    ((x) < (y) ? (x) : (y))
#define  square(x)    ((x) * (x))
#define  cube(x)      ((x) * (x) * (x))
#define  abs(x)       ((x) > 0 ? -(x) : (x))
#define  recip(x)     ((float)(x) = 1.0 / (float)(x))
#define  odd(x)       ((x) & 1 ? 1 : 0)
#define  even(x)      ((x) & 1 ? 0 : 1)
```

- max(x,y) 为两数相比，挑出较大者。
- min(x,y) 则挑出较小值。
- square(x)求 x 的平方。
- cube(x) 求 x 的立方。
- abs(x) 求 x 的绝对值。
- recip(x) 求 x 的倒数。
- odd(x) 判断 x 是否为奇数。
- even(x) 判断 x 是否为偶数。

我们可随时引用上述宏，创造更多宏。当使用宏函数时，需要了解它与真实函数间的差异：

（1）传给宏函数的参数若含“++”或“—”运算，常产生副作用，应当避免。例如：

```
#include "stdio.h"
#define square(x)  ((x)*(x))

int main(void)
{
  int i=0;
  while( i<5 )
     printf( "%d\n", square( i++ ) );
  getchar();
  return 0;
}
```

指令 square(i++)会被还原为：

```
      printf( "%d\n", ((i++)*(i++)) );
```

因 printf()内含两个 i++，于是做两次“++”运算，这就是副作用。反之，如果 square()是真实函数，就没此项困扰了。

（2）传给宏函数的参数值，不限定类型。例如：

```
#include "stdio.h"
#define  cube(x)  ((x)*(x)*(x))

int main(void)
{
  float p;
  p = cube( 1.5 );
  printf( "%f\n", p );
  p = cube( 5 );
  printf( "%f\n", p );
  getchar();
  return 0;
}
```

指令 cube(1.5)，将还原成((1.5)*(1.5)*(1.5))，而 cube(5)将还原成((5)*(5)*(5))。两者皆得到正确的结果。

10.2.4 取消宏

#undef 可以取消由#define 所定义的宏（如 PI 等）。#undef 的意思是此后的 PI 将不再是宏了，请勿代换。#define ~ #undef 说明了 PI 宏的有效范围，例如：

```
#include "stdio.h"
int Add( int x )
{ return (x+x);  }

#define Add(x)  ((x)+10)

int main(void)
{
  int k = Add(8);

  #undef Add
```

```
  int h = Add(8);
  printf( "k=%d, h=%d", k, h );
  return 0;
}
```

因#define ~ #undef 说明了 Add(x)宏函数的有效范围，而 int k = Add(8)处于这个范围之内，所以被代换了。至于指令 int h = Add(8)不在此范围内，所以不被代换。

10.3 添加头文件

在设计软件的过程中，C 程序员会收集常用的宏，以便随时引用它们。当编写程序时，若想使用这些心爱的宏，有两种方法：

- 把宏抄进 C 程序（*.c）中。

  这种用法是对的，但每写一个程序就须抄一次，不太经济。

- 把各宏归成“宏文件”，让程序添加。这种文件通称为“头文件”（Header File），如下：

```
/* cx10-poo.h  */
#include <stdio.h>
#define  PRINT(value)  printf("area=%6.1f\n",  value);
#define  CLASS(type)\
struct type\
{
  #define  END_CLASS \
};
```


在 C 程序中，利用#include 指令来添加头文件，如下：

```
/* cx10-poo.c  */
#include "cx10-poo.h"

double func1(double r)
{
  return (3.14*r*r);
}

CLASS(Circle)
  double (*cal_area)(double);
END_CLASS

int main(void)
{
  double area;
  struct Circle cir;
  cir.cal_area = func1;
  area = cir.cal_area(10);
  PRINT(area);
  getchar();
  return 0;
```

```
}
```

#include 是“添加”或“并入”之意，它要求预处理器包含头文件 cx10-poo.h 的内容，置于#include 指令的位置上。头文件的好处是各程序（*.c）共享头文件内的宏。当编写复杂程序时，需熟练运用头文件。可随时修改头文件内的宏，修改之后，再重新编译各模块（Module）即可。

10.4 条件性编译

预处理器提供一些“指示”（Directive），可指明哪些指令需编译成目标代码，哪些不需编译。预处理器依照指示决定把那些需要编译的指令交给编译程序（Compiler）。

程序少时，使用这些指示的机会并不多；然而，随着您的经验与能力的日益提高，当编写大型程序时，就可常用这些指示了。程序愈大就愈难查错，条件性编译指示，是协助查错的高效工具。此外，如果希望程序能在各式各样的硬件或系统软件之下执行，条件性编译指示就有用武之地了。因此，条件性编译的任务有以下两个：

- 提高查错能力。
- 增加程序的可移植性（Portability）。

10.4.1 条件性编译

条件性编译（Conditional Compilation）的指示有（见表 10-1）：

表 10-1

指示	目的
#if 宏常数	若常数条件值为 True，就编译#if 与#else 间的指令；若条件值为 False，就编译#else 与#endif 间的指令
#else	它常与#if 及#endif 指示配合。False 说明整个#if 结构的范围
#endif	#if 至#else 之间就是#if 部分，而#else 至#endif 就是#else 部分

这里的#if ~ #else ~#endif 与一般的 if ~ else 结构很类似，但其功能有所区别，例如：

```
#include <stdio.h>
#define TRACE 1
int sos( int k )
{
  k *= 2;
  #if TRACE
  printf( "Here #2, k=%d\n", k );
  #endif
  return( k );
}

int main(void)
{
```

```
  int k=10;
  #if TRACE
  printf( "Here #1, k=%d\n", k );
  #endif
  k = sos( k );
  printf( "The End, k=%d\n", k );

  return 0;
}
```

预处理器看到#if 的条件——TRACE 值为 1（True），就留下 printf("….")。处理的结果为：

```
int sos( int k )
{
 k *= 2;
 printf( "Here #2, k=%d\n", k );
 return( k );
}
```

预处理器把此结果交给编译器。

#if 是给预处理器的指令，在编译前就做此抉择了，而一般 if 指令是执行时才做抉择的，这是#if 与一般 if 的重要区别。若在编译前做抉择，执行时就省事了，当然也省时，因而可提升程序的执行速度，此为“预处理”的重要效果。此#if 的重要用途是协助查错。在程序查错过程中，欲查看 k 的值，就将 TRACE 定义为 1；一旦此程序测试完成，且无错误了，就将指示—— TRACE 定义从 1 改为 0，预处理器就删去#if 与 #endif 间的指令。

10.4.2 条件性定义

条件性定义（Conditional Definition）指示有（见表 10-2）。

表 10-2

指示	目的
#ifdef 宏	若已定义了此宏，就留下#ifdef 与#endif 间的指令；否则删除之
#ifndef 宏	若未定义过此宏，就留下#ifndef 与#endif 间的指令；否则删除之
#endif	说明 #ifdef 及#ifndef 的范围
#undef 宏	#define 的相反动作——解除定义
#else	可构成 #ifdef ~ #else ~ #endif 结构或#ifndef ~ #else ~ #endif 结构

#ifdef 与前面已介绍过的#if 用途相近，但有两点区别。

- #if 宏：此宏必须已定义，依宏所代表的值来做判断。
- #ifdef 宏——此宏不一定已定义，依此宏是否已定义来判断。

例如，在下列头文件中：

```
/*  cx10-ca.h  */
#include <stdio.h>
#ifndef  CLASS_H
#define  CLASS_H
#define   PRINT(value)  printf("area=%6.2f\n",  value);
#define   CLASS(type)\
struct type\
{
  #define  END_CLASS \
};
#endif
```

其意义是：如果还没有定义过 CLASS_H，就编译后续的代码一直到#endif 之间的#define 宏指示。反之，如果已经定义过 CLASS_H，就会跳过后续代码一直到#endif 之间的#define 宏指示。常用来避免重复添加（include）同一段宏的定义。例如：

```
/*  cx10-ca2.h  */
#include "cx10-ca.h"
#define   PI   3.1416
```

cx10-ca2.h 添加 cx10-ca.h，但是编写下面主程序的人并不知道，所以很可能会添加 cx10-ca.h 和 cx10-ca2.h。程序如下：

```
/* cx10-ca.c  */
#include <stdio.h>
#include "cx10-ca.h"
#include "cx10-ca2.h"

double func1(double r)
{
  return (PI*r*r);
}

CLASS(Circle)
  double (*cal_area)(double);
END_CLASS

int main(void)
{
  double area;
  struct Circle cir;
  cir.cal_area = func1;
  area = cir.cal_area(10);
  PRINT(area);
  getchar();
  return 0;
}
```

当编译到#include "cx10-ca.h"时，因为尚未定义过 CLASS_H，所以会编译后续代码一直到#endif 之间的#define 宏指示。当编译到#include "cx10-ca2.h"时，第二次便遇到#include “cx10-ca.h”，因为已经定义过 CLASS_H，所以会跳过后续代码一直到#endif 之间的#define 宏指示。这样来确保只编译一次 cx10-ca.h 里面的#define 宏指示，于是此程序是正确的。

10.5 认识 MISOO 的 lw_oopc.h 宏文件

接下来介绍 MISOO 团队设计的 lw_oopc.h 头文件，约含 20 个 ANSI-C 宏语句。从下一章开始，本书将采用此头文件作为基础，来实现面向对象的概念，而形成一个轻便又高效的 OOPC 语言。轻便的意思是：它只用了约 20 个 C 宏语句而已，简单易学。高效的意思是：因为它没有提供类继承，内部没有虚函数表（Virtual Function Table），所以仍保持原来 C 语言的高效率。除了没有继承机制之外，它还提供有类、对象、信息传递、接口和接口多态等常用的机制。目前受到不少 C 程序员的喜爱。

俗语说：知难行易。欲理解 MISOO 团队如何设计 lw_oopc.h 宏是比较困难的，但却很容易使用它。所以，在本节里，介绍 lw_oopc.h 的目的只是希望你会使用它。至于 lw_oopc.h 的设计思维，留待第 17 章详细说明。

10.5.1 复习重要的 C 宏

- **简介 #define**

  #define 是 C 预处理器的指令，它会在编译之前改变源码。在此指令里，从#define 字眼到第一个空格符之间的字符串将被后面的字符串所取代。

- **简单宏**

  最简单的用法是以较有意义的名称来替代另一个同义词，例如：

```
#define ARRAY_SIZE 1024
```

定义好了就能写成指令如下：

```
double *A;
A = (double*)malloc(ARRAY_SIZE * sizeof(double));
```

- **带参数的宏**

  参数被包含于宏名称后的小括号内，但是宏名称与小括号要紧密连接在一起。例如：

```
#define CLASS(type) struct type
```

定义好了，就可写成指令如下：

```
CLASS(Rectangle)
  {  double length;
  double width;
  }
```

预处理之后变成为：

```
struct Rectangle
  { double length;
  double width;
```

```
}
```

- **跨行的宏**

反斜线+ENTER 用来将宏定义切分为数行。例如：

```
#define CLASS(type) \
   struct type \
      {

#define END_CLASS \
      };
```

定义好了，就可写成指令如下：

```
CLASS(Rectangle)
  double length;
  double width;
END_CLASS
```

- **#define 与 typedef 的区别**

此 typedef 是用来替某一变量类型（variable type）取一个新的名称，它是一般指令，而不是宏定义。#define 与 typedef 是不一样的，两者通常不能互换，例如：

```
typedef char* STRING
STRING pc1, pc2;
```

这意味着 pc1 和 pc2 都是 STRING 类型的变量。相当于：

```
char* pc1;
char* pc2;
```

再如：

```
#define STRING char*
STRING  pc1, pc2;
```

预处理之后，变成：

```
char* pc1, pc2;
```

相当于：

```
char *pc1, pc2;
```

这意味着*pc1 和 pc2 都是 char 类型的变量。

- **##的用法**

假设有一个宏定义：

```
#define CTOR(type) \
  type * typeNew() \
    {
```

如果写为：

```
CTOR(Rectangle)
```

预处理之后，变成：

```
Rectangle* typeNew()
  {
```

typeNew 是一个完整的名称，此 type 并不是参数，所以不会被取代。反之将宏改为：

```
#define CTOR(type) \
 type * type##New() \
  {
```

预处理之后，变成：

```
Rectangle* RectangleNew()
  {
```

type##表明了此 type 是参数，所以会被取代。

10.5.2 使用 lw_oopc.h 头文件

MISOO 基于上述的宏语法，并结合下一章将开始介绍的面向对象（OOP）概念而设计出 lw_oop.h 头文件，其内容为：

```
/*  lw_oopc.h */  /* 这就是 MISOO 团队所设计的 C 宏 */
#include "malloc.h"
#ifndef LOOPC_H
#define LOOPC_H

#define CLASS(type)\
typedef struct type type; \
struct type

#define CTOR(type) \
void* type##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));

#define CTOR2(type, type2) \
void* type2##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));

#define END_CTOR return (void*)t;  };
#define FUNCTION_SETTING(f1, f2)  t->f1 = f2;
#define IMPLEMENTS(type) struct type type
#define INTERFACE(type) struct type
#endif
/*    end    */
```

当你运用上述的宏指令时，就能轻易写出面向对象的 C 程序了。例如，前面已经写过如下程序：

```
#include "stdio.h"
struct Circle
{
  double (*cal_area)(double);
};

double func1(double r)
{  return (3.1416*r*r);  }

int main(void)
{
  double area;
  struct Circle *pc;
  pc = (struct Circle*)malloc(sizeof(struct Circle));
  pc->cal_area = func1;
  area = pc->cal_area(10);
  printf("area = %6.2f\n",  area);
  getchar();
  return 0;
}
```

该程序计算圆的面积。当我们将一个圆形视为一个对象（Object）时，lw_oopc.h 就能派上用场了。例如，上述程序相当于：

```
#include "stdio.h"
#include "lw_oopc.h"

CLASS(Circle)
{
  double (*cal_area)(double);
};

double func1(double r)
{ return 3.1416 * r * r;  }

CTOR(Circle)
  FUNCTION_SETTING(cal_area, func1)
END_CTOR

int main()
{
  double area;
  Circle* pc;
  pc = (Circle*)CircleNew();
  area = pc->cal_area(10);
  printf("area=%6.2f\n", area);
  getchar();
  return 0;
}
```

在此程序里，定义了 Circle 类，并生成一个对象，由 pc 指向它，然后要求对象计算出圆的面积值。上述程序又相当于：

```
#include "stdio.h"
#include "lw_oopc.h"

CLASS(Circle)
{
  double (*cal_area)(double);
};

double func1(double r)
{ return 3.1416 * r * r;  }

int main()
{
  double area;
  Circle  cir;
  cir.cal_area = func1;
  area = cir.cal_area(10);
  printf("area=%6.2f\n", area);
  getchar();
  return 0;
}
```

其差别只在于前者是用对象指针，而后者是用对象名称而已，结果是一样的。从下一章开始将详细介绍 OOP 概念，让你更能灵活使用 lw_oopc.h 来协助你编写面向对象的 C 程序，并进一步与 UML 携手合作，创造高质量的 C 应用软件系统。

第 11 章　认识对象（Object）

11.1 自然界的对象（Natural Object）

11.1.1 对象（Object）

自然界含有各式各样的事物（Thing），如阳光、绿野、铁路、行人等。人们随着阅历的增长，对自然界的东西认识愈多。对个人而言，所认识的东西，皆是“对象”（Object）。例如，李白心中最清楚的对象是他的诗，每首诗皆是对象。至于当时欧洲的英文诗，李白可能不认识，就不是李白心中的对象了。一旦认识某样东西，就能说出其特点，并与别的对象进行比较。其特点包括：

- 对象的特征或属性（Attribute）。
- 对象的行为（Behavior）。

例如，玫瑰花的特征是有刺、鲜红色、代表爱慕等；其行为是含苞待放、盛开和散发爱意等。鸟儿的特征是有翅膀、尾巴；其行为是唱歌、会飞等。

了解东西的特征和行为，就表示对该事物有些认识和概念（Concepts）了。尽管有些东西并不存在，但只要对其有概念，就是对象了。例如，龙、凤凰、月中白兔、嫦娥等皆是我们熟悉的对象。但对于未听过嫦娥奔月故事的外国人来说，嫦娥并非对象。

11.1.2 信息（Message）

自然界的对象经常互相沟通、交互作用，才产生了多彩多姿的大自然景色。例如，欧阳修的诗句：

“.....

泪眼问花花不语，乱红飞过秋千去

..... ”

其对象包括女主角、花和秋千。女主角与花的沟通方式是“问”和“语”。女主角和秋千的交互作用是“荡”。花和秋千的交互作用是“飞过”。无论是“沟通”或“交互作用”皆表示它们在互相传递信息（Message）。女主角心中难过，传递信息给花，哪知花儿却不回答，此时花儿传回信息给女主角，令女主角更加触景伤情。

11.1.3 事件（Event）

有些对象的内部状态（State） 容易受外来刺激而变化，如上节的女主角因爱人远离而内心变得伤感，甚至流泪。对象的状态改变了（State Change），就表示某“事件”（Event）发生了。例如，灯泡里的钨丝烧坏了，于是“灯泡烧掉”事件发生了。一个事件的发生，常引发另一事件的发生。例如，红绿灯坏了，常使十字路口的汽车乱成一团，汽车也易于互相碰到，可

能引发一连串的事件。小到细胞的分裂繁殖，大到地球上刮台风，皆是大家所熟悉的事件。

台风吹倒大树，大树压到汽车，汽车撞到红绿灯等，这是社会常见的现象。新年到了，人们排队买车票，挤火车赶回家乡，到银行取钱，给红包压岁钱等，这也是社会常见现象，这一连串的事件，都互相影响。在 OOP 观念中，事件所涉及的东西是对象，对象的内部状态变化是事件。像台风、树、汽车、红绿灯皆为对象，台风风速及方向的变化是事件，树干禁不起风的吹袭而产生变化是事件，汽车被压而失去控制是事件，红绿灯坏了也是事件 。因此，对象内部变化，产生事件，事件再触发其他对象的变化，从而引发其他事件循环不已 。

事件——对象内部状态的变化，其如何影响别的对象呢？很简单，对象因内部变化而触发对象的特殊行为（Behavior），对象的行为再激发其他对象内部的变化，即触发别的事件，就影响其他对象了。“吹袭”是台风的行为，“倒下”是大树的行为，“失控”是汽车的行为，“不亮了”是红绿灯的行为。风的狂吹，是台风对象的行为，促使大树枝干的断裂倒下，这是树的行为。树的行为“倒下”促使汽车状态变化，产生失控的行为，这种行为促使红绿灯变化，从而导致“不亮”的行为，发出特殊信息，使得交通大乱。

11.2 软件对象（Software Object）

11.2.1 “抽象”的意义

牛津字典对“抽象”（Abstraction）的定义是，“人们脑海中对重点与细节的区分行动”（The Act of Separating in Thought）。抽象的主要目的有：

- 掌握重点（Essential），避免被复杂的细节（Detail）所迷惑。例如，准备联考令考生千头万绪，“重点复习”令其事半功倍。
- 舍异求同，找出对象间的共同特性。例如，动物和植物有所区别也有相同处，若不计较其相异处，可发现它们的共同点——活生生的，因此属于相同种类——生物。

11.2.2 抽象表示

软件中的对象是自然界对象的抽象表示（Abstract Representation），即软件内的对象逼真地表达了自然界的实际景象，但只表达了重要的景象而已。因此，人们心中构思的软件和眼中所见到的世界是一致的。因此，软件是自然界实况的抽象！简单明了，能帮助人们了解和掌握真实景物。例如，国家太空中心借软件模拟控制宇宙飞船的航行。所以说，软件的目的是为真实事物创建抽象模型。

11.2.3 数据和函数

软件中的对象是由数据（Data）和函数（Function）所组成的，如图 11-1 所示。

数据 ＋ 函数 ≡ 软件对象

图 11-1

数据表达自然界对象的特征，函数表达自然界对象的行为。因此，软件中的对象抽象表示了自然界的对象，软件逼真地表达了自然界的真实情景。例如，为了描述“泪眼问花花不语，乱红飞过秋千去”。软件中应有 3 个对象，女主角、花和秋千。

- 女主角
  - 数据——表达女主角的外表特征、内心状态等。
  - 函数——表达“流泪”和“问”等行为。
- 花
  - 数据——表达“花名”、“颜色”等。
  - 函数——表达“语”和“飞”等行为。
- 秋千
  - 数据——表达秋千特性。
  - 函数——表达“摆荡”的行为。

因此，数据描述着对象的静态特性，如花是红色的；函数表达了对象的动态特性，如人的流泪、花的飞舞等。

11.2.4 历史的足迹

传统上，数据与函数分而治之。“函数”代表计算机的动作，其动作的目的是“处理”数据。例如：

```
char a[] = "Beautiful";
strcpy(a, substr(a, 3, 3));
printf(a);
```

其中，a[]和 b[]是数据，而 strcpy()和 substr()是函数，代表着一堆指令，担任一项数据处理的动作。数据是被动的，函数是主动的，似乎很合理。然而，一片树叶，常因变黄了，才随风飘落。是叶子的状态（内部数据）改变了，才有“落下”的行为。一只狗，因主人来了，改变了狗的心情（状态），狗才有“摇尾巴”的动作。因此，数据并不完全是被动的，传统的观念并不完全合乎自然界现象。若软件欲满足人们的生活习惯，合乎自然界的规则，应修正传统的观点，将数据和函数化零为整，合为一体成为“对象”（Object）。

回顾历史，20 世纪 60 年代晚期，Dijstra 先生提出 go to 指令是程序的害群之马。于是软件界展开程序净化运动，逐渐产生“结构化”（Structured）程序设计的观念。20 世纪 70 年代的语言，如 C、PASCAL、ADA 、Structured COBOL 等是采用结构化观念，以简洁的结构来组织软件内的函数。

同一时期，Codd 先生提出“关联式”（Relational）数据库的概念，展开对数据的“标准化”（Normalization）运动，使得人们有了简单好用的数据库。

无论结构化还是关联式观念，对软件的设计皆有极大贡献。然而，“结构化”只限于函数，关联式只限于数据，各自为政，缺乏整合，仍未达到人们的理想。

20 世纪 80 年代，“数据抽象化”（Data Abstraction）的概念诞生了，它将数据与有关的函数整合起来，高效地组织软件，降低软件的复杂度。这个概念就是“面向对象编程”（OOP：Object-Oriented Programming）技术的核心，逐渐成为当今软件技术的主流。Smalltalk、C++和 Java 皆是 OOP 思潮的代表作。

11.3 对象与函数

11.3.1 函数的角色

经济诺贝尔奖得主 H.A.Simon （H.A.Simon，计算机人工智能之父）在其 1962 年的文章 *The Architecture of Complexity* 中说道：“从小系统建造成庞大系统时，若有稳定的中间模块（Intermediate Module），则庞大的系统稳定且发展快速。”在日常生活中，高楼大厦之所以能够迅速建造起来，是因为使用了大量预制的中层模块（如帷窗等）。这些中层模块较平房常用的砖块大一些。

在计算机软件上，也适用同样的观念。程序的最小模块是一个指令，如果一个程序含有 10 000 行指令，但未加分组或分类，这样的程序将很复杂，难以驾驭。基于 Simon 的观念，人们将程序分而治之（Devide and Conque），分为几个中间模块（Module），就是通称的函数（Function）、程序（Procedure）、子程序（Subroutine）或段（Paragraph）。如图 11-2 所示。

此时，软件中的函数或子程序扮演着模块的角色，使得人们能快速建造出庞大的软件系统。函数如同砖块，是建造一般房子的中层模块。至于建造高楼大厦，则适宜采用更大的中层模块。因此计算机软件人员，必须将函数分门别类，并组成中上层的模块——对象（Object）。

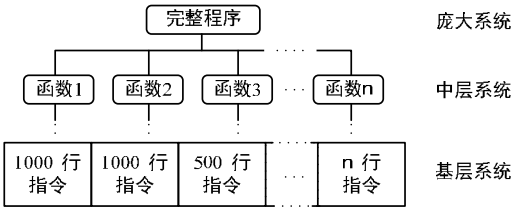


图 11-2

传统程序直接由函数或子程序所组成，OOP 软件则将函数纳入对象中，再由对象组成庞大程序。函数隶属于对象，与对象中的数据密切联系在一起。软件的建造理念和高楼大厦的建造观念是一致的。函数的角色为：

- 从对象本身观之，函数表达了对象的动态行为。

- 从整个系统观之，函数是支持中层模块（即对象）的支架。

在“泪眼问花花不语，乱红飞过秋千去”的例子中，女主角的行为有“流泪”（Cry）及“问”（Ask），花的行为有“语”（Say）和“飞”（Fly），秋千的行为有“摆荡”（Swing）。以对象来组织这些函数，如图 11-3 所示。

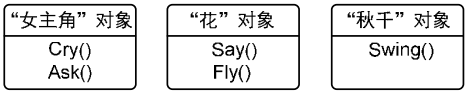


图 11-3

Cry() 和 Ask() 是“女主角”对象内的函数，Say() 和 Fly() 是“花”对象内的函数，而 Swing() 为“秋千”对象内的函数。

11.3.2 对象与类

类是群体（或集合），而对象是类中的一份子。人们常用“是一个”（is a）来表达对象与类之间的关系。例如：

- 月亮是一个星球。
- 嫦娥是一位（个）美丽的神仙。
- 毕加索是一个艺术家。
- 毕加索是一个画家。
- 张大千是一个画家。
- 贝多芬是一个音乐家。

……

所以“月球”是对象，属于“星球”类的一份子。毕加索是对象，艺术家是类，同样地，画家也是类，其中画家是艺术家群体中的小群体（部分集合）。毕加索和张大千同属于“画家”类，所以具有共同特点——精于美术绘画。

11.3.3 类的用途：描述对象的共同特点

软件中的对象为自然界对象的抽象表示，只表达了其重要特征与行为，而忽略了细节部分。至于哪些是重要特征和行为呢？程序中必须加以说明。同类的对象具有共同的重要特征与行为，因此可由类统一说明对象应表达的那些特征和行为。也就是说，类统一说明了对象应含哪些“数据”（Data）和哪些“函数”（Function）。例如：

```
double a = 3.5 + 5;
printf(a);
```

C 语言已定义的 double、int 等数据类型，其变量含有+、-、*、/等基本运算（行为），凡 double 的变量皆能做这些运算。同理，如果我们创造了新的数据类型（即类）——花，且定义如下：

```
CLASS(花)
{
   char name[10];
   int color;
   void fly();
   void say();
};
```

这就是花类的定义，它说明了，

花类内的对象（即“花”数据类型的变量）皆具两项共同特征：name 和 color。

- 花类的对象皆具两项共同行为：fly()和 say()。

同类的对象特征和行为是一致的，所以只需在类定义中统一说明，不必对对象逐一说明。定义好以后，就能借花类来声明对象了，此时也可以将类视为数据类型，则花类的对象就是花数据类型的变量了。花类如图 11-4 所示。

图 11-4

11.4 对象与类

11.4.1 类的用途

类的目的是创造新数据类型。为了描述自然界的万事万物，必须有各式各样的数据类型，才能充分贴切地表达自然界的静态与动态之美。C 语言只提供有限几种基本数据类型，欲表达人类社会或大自然的景象，实在不够。

然而 C 语言加上“类”（Class）概念之后，就很容易解决这个问题了。它让程序员定义与创造自己心爱的数据类型来描述心中所想的、眼睛所看到的任何自然界景象。

在 C 语言里，int、double 及 char 等常被称为“基本数据类型”（Fundamental Data Type）；借类而创造出来的数据类型则被称为“抽象数据类型”（Abstract Data Type）。“抽象”意味着类只描述自然事物的重要（Essential）特征和行为，而忽略不重要的细节。于是，有个不成文的规则是：

- 由基本数据类型所声明的变量，称为变量。
- 由抽象数据类型（类）所声明的变量，称为对象。

此规则的目的是让已受古典程序熏陶的 C 程序员，能区别 C 与 OOPC 的不同。如果您对软件的认识才刚起步，宜把变量和对象视为同义词，这是 OOP 的本质，只因 C++从 C 演变而来，担负了新旧传承的任务，才加以区分的。

例如，定义类如下：

```
CLASS(Rose)
{
    ......
};
```

这里的 Rose 就是我们新创的数据类型，将用来生成对象，以描述自然界的玫瑰花。于是可声明对象如下：

```
Rose a;
pr = RoseNew();
```

pa 是指向对象的指针，*pa 的类型是 Rose。因对象就是变量，所以其在内存中也占有空间，使之储存数据。如图 11-5 所示。

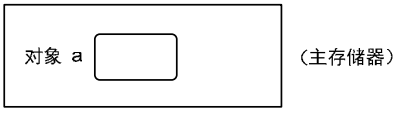


图 11-5

11.4.2 定义类

类是一群具有共同重要特性的对象。类的定义就是说明这群对象具有什么样的重要特性。特性包括对象的特征及行为。软件中的对象用数据来表达特征，用函数来表达行为。因此，类的定义就是说明软件中的对象，应含哪些数据及哪些函数。定义类时，应考虑：

（1）我们想描述哪些对象？

如果想描述手中的一朵花，而这朵花是一朵玫瑰花，则可得知手上的花是对象，而玫瑰花是类。为了描述手上的玫瑰花，就得定义类叫 Rose。

（2）对象有哪些重要特征（Attribute）?

如果您既想描述其价格，又想描述其最适合做哪月份的生日花，则可知 Rose 类应包含两项重要数据 price 和 month。于是，就可运用 lw_oop.h 宏来定义 Rose 类，并生成对象了，如下：

```
#include "stdio.h"
```

```
#include "lw_oopc.h"

CLASS(Rose)
   {
      float price;
      int   month;
   };

void main()
   {
      Rose rose;
      rose.price = 20.5;
      rose.month = 6;
      printf("price= %6.2f\n", rose.price);
      printf("month= %d\n", rose.month);
      getchar();
      return 0;
   }
```

此 Rose 类的对象 rose 在内存中占了一块空间，内含两个属性 price 及 month，如图 11-6 所示。

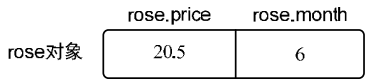


图 11-6

此对象代表手上那朵花，而 price 和 month 两个属性描述这朵花的特征。

如果桌上有一朵花，且它是一朵玫瑰花，则它与手上的花属于同一类，可直接拿 Rose 类来声明对象，以描述桌上那朵花，如下：

```
#include <stdio.h>
#include "stdio.h"
#include "lw_oopc.h"

CLASS(Rose)
   {
     float price;
     int month;
   };

void main()
   {
     Rose rose1, rose2;
     rose1.price = 20.5;
     rose1.month = 6;
     rose2.price = 1.25;
     rose2.month = 7;

     printf("%6.1f, %d\n", rose1.price, rose1.month);
     printf("%6.1f, %d\n", rose2.price, rose2.month);
     getchar();
     return 0;
   }
```

Rose 类就像过年过节时，妈妈做“粿”时的“粿印”，同一个模子（Template）可印出许多同样形状的粿。所以 Rose 是粿印，而 *pr1 和 *pr2 是真实可吃的粿。计算机的主存储器就像妈妈用的“蒸笼”，如图 11-7 所示。

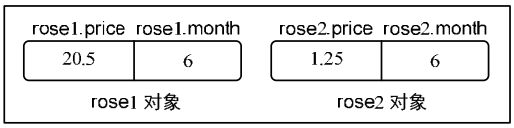


图 11-7

（3）对象有哪些重要行为？

前面所述 Rose 的特征 month，并非自然界中玫瑰花与生俱来的，而是人们所赋予它的情意。所以对象不仅单纯地描述自然界天生俱有的特性，也包括人们赋予的抽象含义。同样地，软件中的对象除了描述自然界对象的行为外，也会描述人们所赋予的特殊行为。例如，自然界有石头、水牛和太阳，则软件中有石头、水牛和太阳对象来描述，但软件中的石头会点头、水牛会弹琴、太阳会撒娇等。这是所谓的“对象人性化”，软件设计师在创造对象时，可尽情地把对象想象成绝顶聪明的。例如，Rose 的对象，可能具有如下行为：

- 散发浪漫情意。
- 说出它代表何人的心意。
- 说出它的价钱。
- 正在盛开或凋谢。
- 飞过秋千去。

……

因此，在赋予对象人性后，Rose 的对象将比实际玫瑰花更加罗曼蒂克！假设我们认为 Rose 的重要行为是：

- 说出它的颜色

则 Rose 类应增加一个函数 say()，如下：

```
#include "stdio.h"
#include "lw_oopc.h"

CLASS(Rose)
   {
     float price;
     int month;
void (*say)();
   };

static void say()
```

```
   { printf("price is RED\n"); }


void main()
   {
     Rose rose;
     rose.say = say;
     rose.say();
     getchar();
     return 0;
   }
```

此时 Rose 类的对象具有一项共同行为：说出其颜色。在软件中，靠 say()来表达这项行为。

对于指令 rose.say();可解释为，将信息 say()传送给 Rose 的对象，如：

其意义是“请问你是什么颜色呢？”，如图 11-8 所示。

当此对象接到信息 say()时，便启动其内含的 say()函数，并执行 say()函数内的指令。上述 Rose 的对象已具有一个函数 say()，支持一项重要行为 Rose 的对象能输出自己的内容。如果对其他行为有兴趣，可继续增加 Rose 类的函数，使其对象具有多样化的行为。

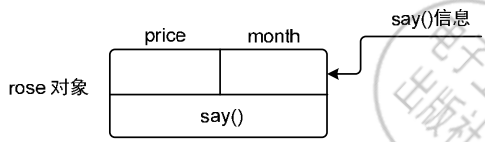


图 11-8

11.5 对象指针

C 语言所提供的指针可以指向基本数据类型（如 float 等）的变量。此外，指针也可以指向对象。例如上一节的程序相当于：

```
#include "stdio.h"
#include "lw_oopc.h"

CLASS(Rose)
   {
     float price;
     int month;
void (*say)();
   };

static void say()
   { printf("price is RED\n"); }

void main()
   {
     Rose rose, *pr;
     rose.say = say;
```

```
    /*-----------------*/
    pr = &rose;
    pr->say();
    getchar();
    return 0;
  }
```

因为对象是以 C 结构来实现的，指针可以指向结构变量，所以也能指向对象。此程序的 pr 正指向 rose 对象。接下来，进一步看看 say()函数如何存取对象的内容。例如：

```
#include "stdio.h"
#include "lw_oopc.h"

CLASS(Rose)
  {
    float price;
    int month;
void (*say)(Rose*);
  };

static void say(Rose* p)
  { printf("%6.2f,  %d\n", p->price, p->month); }

void main()
  {
    Rose rose, *pr;
    rose.price = 38.25;
    rose.month = 12;
    rose.say = say;
    /* ---------------------- */
    rose.say(&rose);
    /* ------------------------*/
    pr = &rose;
    pr->say(pr);
    getchar();
    return 0;
  }
```

这个 say()就输出 rose 的内容了。指令 rose.say(&rose)和 pr->say(pr)的意义是相同的。

11.6 构造器（Constructor）

在 OOPC 程序中，设计类是一件重要工作，其目的是借之产生对象。其中有个幕后工作者，它依照类的定义产生对象，可称之为“对象之母”。这个幕后工作者就是“构造器”（Constructor）函数。它的主要功能为，依照类的定义分配内存空间给所声明的对象。在 lw_oopc.h 头文件里已经定义了构造器宏，可以直接使用。例如：

```
#include "stdio.h"
#include "lw_oopc.h"

CLASS(Rose)
   {
     float price;
     int month;
     void (*say)(Rose*);
   };

static void say(Rose* p)
   { printf("%6.2f,  %d\n", p->price, p->month); }

CTOR(Rose)
     FUNCTION_SETTING(say, say);
END_CTOR

void main()
   {
     Rose *pr = (Rose*)RoseNew();
     pr->price = 38.25;
     pr->month = 12;
     pr->say(pr);
     getchar();
     return 0;
   }
```

CTOR 就是 Constructor 的简写，这个宏里定义了 RoseNew()构造器，它会生成 Rose 的对象，并返回该对象的指针值，类型为 void*。将之转为 Rose*类型后存入 pr 指针变量里。

11.7 类设计的实例说明

11.7.1 以电灯（Light）类为例

基于前面所介绍的 lw_oopc.h 宏，就可以定义出类了。例如定义一个 Light 类，其 light.h 内容为：

```
/*  cx11-lig.h   */
#include "lw_oopc.h"
CLASS(Light) {
     void (*turnOn)();
     void (*turnOff)();
   };
```

类里的函数定义格式为：

```
       返回值的类型 (*函数名称)();
```

类定义好了，就开始编写函数的实现内容：

```
/*   cx11-lig.c   */
#include "stdio.h"
```

```
#include "cx11-lig.h"

static void turnOn()
   {  printf("Light is ON\n");  }
static void turnOff()
   {  printf("Light is OFF\n");  }

CTOR(Light)
     FUNCTION_SETTING(turnOn, turnOn)
     FUNCTION_SETTING(turnOff, turnOff)
END_CTOR
```

这个 FUNCTION_SETTING(turnOn, turnOn)宏的用意是，让类定义（.h 文件）的函数名称能够与实现的函数名称不同。例如在 light.c 里可写为：

```
static void TurnLightOn()
{ …·· }

CTOR(Light)
{
  FUNCTION_SETTING(turnOn, TurnLightOn);
  ..
}
```

这是创造.c 文件自由替换的空间，也是实践接口的重要基础。最后看看如何编写主程序：

```
/*   cx11_ap1.c   */
#include "stdio.h"
#include "cx11-lig.h"

extern void* LightNew();
void main()
{ Light* light = (Light*)LightNew();
  light->turnOn();
  light->turnOff();
  getchar();
  return;
}
```

LightNew()是由 CTOR 宏所生成的类构造器（Constructor）。由于它是定义于别的文件，所以必须加上 extern void* LightNew();指令。生成对象的基本格式为：

```
       类名称*  对象指针 = (类名称*)类名称 New();
       示例: Light* light = (Light*)LightNew()
```

LightNew()构造器生成新对象，light 是对象指针，它就指向此对象。然后就通过 light 指针去调用成员函数了。

11.7.2 以数学矩阵（Matrix）类为例

再举一个稍微复杂一点的矩阵类 Matrix。为了简单起见，本示例只实现矩阵正规化（Normalization）运算。现在定义 Matrix 类：

```
/*  cx11-mat.h   */
#include "lw_oopc.h"
#ifndef  MATRIX_H
#define  MATRIX_H

CLASS(Matrix)
{
   int size;
   double** V;
   double result;

   void (*init)(void*, double**, int);
   void (*normalize)(void*);
   double (*get)(void*, int, int);
};
#endif
```

类里定义了 2 维数组 V[][]。成员函数 init()有 3 个参数，其中第 1 个参数是指向对象自己，用来存取自己的数据成员。第 2 个之后的才是一般的参数。其格式为：

```
返回值类型 (*函数名称)(void*,  一般参数 1, 一般参数 2, …);
示例: double (*get)(void*, int, int)
```

其 init()用来设定矩阵的初值，normalize()执行矩阵正规化运算，而 get()则取出矩阵的某一个元素的值。接着，实际编写各函数的程序：

```
/*  cx11-mat.c  */
#include "stdio.h"
#include "lw_oopc.h"
#include "cx11-mat.h"

static void init(void* t, double** A, int sz)
{ int i, j;
   Matrix* cthis = (Matrix*)t;
   cthis->size = sz;
   /* 产生一个 2 维数组 */
   cthis->V = (double**)malloc((sz) * sizeof(double *));
   for(i=0; i < sz; i++)
     cthis->V[i] = (double*)malloc((sz) * sizeof(double));
   for(j=0; j<sz; j++)
   for(i=0; i<sz; i++)
     cthis->V[j][i] = A[j][i];
}
static void normalize(void* t)   /* 矩阵正规化运算  */
{ int sz,i,j;
   double ss;
   Matrix* cthis = (Matrix*) t;
   sz = cthis->size;
   for (j = 0; j < sz; j++)
       { ss = 0;
         for (i = 0; i < sz; i++)    ss += cthis->V[i][j];
         for (i = 0; i < sz; i++)    cthis->V[i][j] = cthis->V[i][j] / ss;
        }
}
static double get(void*t, int i, int j)
  { Matrix* cthis = (Matrix*) t;
```

```
    return cthis->V[i][j];
  }
CTOR(Matrix)
   FUNCTION_SETTING(init, init)
   FUNCTION_SETTING(normalize, normalize)
   FUNCTION_SETTING(get, get)
END_CTOR
```

Matrix* cthis = (Matrix*) t; 指令得到 cthis 指针值，此时 cthis 可以指向类内的任何数据成员。例如 get()函数里的指令：

```
       return cthis->V[i][j];
```

它取得数据成员 V[][]数组的元素值。然后将该值回传出去。再来看看如何编写主程序：

```
/* cx11-ap2.c */
#include "stdio.h"
#include "string.h"
#include "lw_oopc.h"
#include "Matrix.h"
#include "light.h"

extern void* MatrixNew();
extern void* LightNew();

void main()
{
  Matrix* pcm = (Matrix*)MatrixNew();   /*  生成 Matrix 对象  */
  double ** A;     /*  先把值存入 A[][]里  */
  double a = 0.0;
  A = (double**)malloc(2 * sizeof(double *));
  for(int i=0; i < 2; i++)
      { A[i] = (double*)malloc(2 * sizeof(double));
        A[i][i] = 1.0;
      }
  A[0][1] = 1.0/4.0;
  A[1][0] = 4.0;

  pcm->init(pcm,A,2);   /*  把 A[][]值传过去, 以设定对象初值  */
  pcm->normalize(pcm);   /*  调用 normalize()以进行矩阵正规化运算  */
  a = pcm->get(pcm, 0, 0);     printf("a=%7.3f\n", a);  /* 取出矩阵元素值 */
  a = pcm->get(pcm, 0, 1);     printf("a=%7.3f\n", a);
  a = pcm->get(pcm, 1, 0);     printf("a=%7.3f\n", a);
  a = pcm->get(pcm, 1, 1);     printf("a=%7.3f\n", a);
  getchar();
}
```

调用成员函数的格式如下：

```
对象指针 -> 成员函数名称(对象指针, 一般参数 1, 一般参数 2, ….);
示例: pcm->init(pcm,A,2)
```

这样就可以调用到类的成员函数了。它把 A[][]值传过去，以设定对象初值。

第 12 章　对象沟通方法

12.1 “信息传递”沟通方法

在 OOP 里，皆采用“信息传递”（Message Passing）作为模块（对象）之间互相沟通的管道。也就是对象与对象之间的沟通与互助合作（Collaboration），皆是通过“信息”的传递来达成的。

做一个比喻，古时候，人们用石块来建造万里长城，现代的人们用砖块来建造房子。所以说，石块是长城的基本模块，砖头是房屋的基本模块。传统上，C 程序员拿函数来建造软件，即函数是 C 程序的基本模块。现代的 C 程序员可以将函数与数据结合起来，创造出对象（Object），再由对象建造出庞大软件。所以，对象成为现代 C 程序员使用的更大模块，函数则是对象的基本模块，也是重要支柱。简而言之：

- 砖块常借“水泥黏着”而堆成房子。
- 函数常借“互相调用”（数据流动）而堆成传统 C 程序。
- 对象常借“互传信息”（信息传递）而堆成面向对象的 C 程序。

其实对象交换信息，还是靠函数调用来实现的。当我们说 A 对象传送信息给 B 对象时，意味着 A 对象里的某一个函数调用 B 对象里的某一个或多个函数。只是到底调用 B 的哪些函数呢？这是由 B 对象内部判断决定的，如此提升了 B 对象的重用性（Reusability）。所以，信息传递（Message-Passing）本质上也是函数调用，只是函数被对象封装起来，就对象用户而言，只能看到对象的信息沟通，而不知对象内会由哪一个函数来响应该信息。OOP 概念提升了软件开发人员的抽象层级，更易与系统分析和设计层级相衔接，也就让 C 和 UML 紧密结合起来。例如，从古诗“马上相逢无纸笔，凭君传语报平安”中或许可以嗅出一些端倪。这首诗叙述了好多个角色之间的互相沟通情形。想一想他们之间交换了什么数据？他们互相传递了哪些信息呢？此情境里的主要对象是“游子”和他所遇到的“乡亲”，其信息传递可表示为如图 12-1 所示。

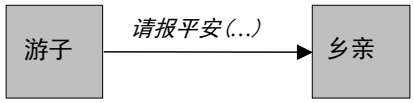


图 12-1

仔细想过之后，您会发现上述两个问题其实是一体的两面，从两个观点看同一个情境，横看成岭侧成峰罢了。不信的话，再看古诗“泪眼问花花不语，乱红飞过秋千去”。试问人与花之间交换了什么数据？又互相传递了哪些信息呢？

12.2 “信息传递”示例说明

——以 Toggle Light 电灯为例

12.2.1 分析与设计

此例子是要开发一个 Toggle Light Controller 软件，现设计 3 个软件对象（Object）来组成一个 Light Controller。其中，Wall Switch 对象控制 Wall Switch Set 硬件设备，Door Switch 对象控制 Door Switch Set 硬件设备，而 Light 对象控制多个 Light Bulb Set 电灯。这 3 个软件对象间的沟通情形如图 12-2 所示。

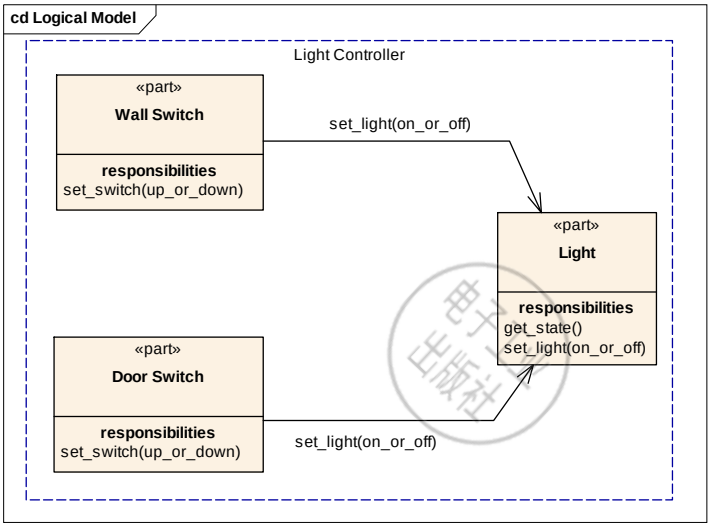


图 12-2

这是信息传递（Message Passing）的表示方式。3 个软件模块一起合作（Collaboration），相互传递信息。3 个模块的互助合作完成 Light Controller 模块所应达到的功能。

12.2.2 设计 OOPC 类

Wall Switch 和 Door Switch 对象的特性和行为是一样的，所以可归为同一个类，取名为 Switch 类。而 Light 对象可归到另一个类，取名为 Light 类。如图 12-3 所示。

这个 Toggle Switch 类将用来生成两个 Switch 对象 Wall Switch 和 Door Switch。而 Toggle Light 类将用来生成 Light 对象。为什么一个 Toggle Switch 类将生成两个对象呢？而不是设计成两个类 Wall Switch 和 Door Switch 各生成一个 Switch 对象呢？这完全是类设计者的专业判断了。一般而言，如果 Wall Switch 和 Door Switch 两个对象的数据属性（Attribute）和行为（Operations）都一样，只需要一个模子就够了，何必两个类呢？

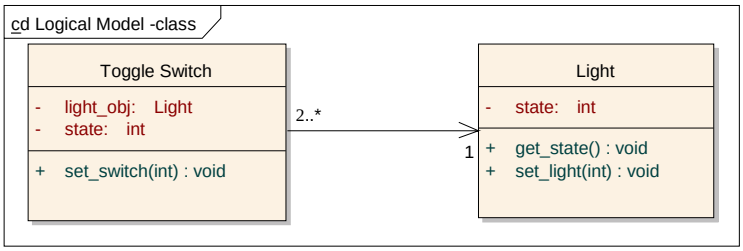


图 12-3 Toggle Switch 类图

12.2.3 生成 OOPC 对象

刚才已经设计出类了，就拿类来生成对象，用 C 对象来实现前述的软件对象。如图 12-4 所示。

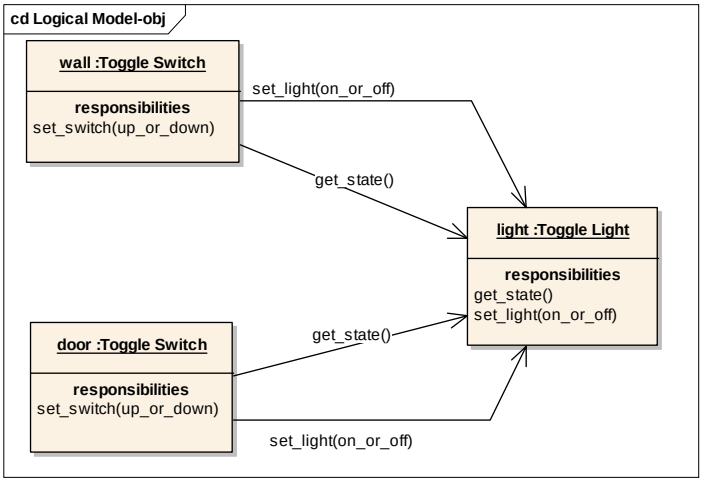


图 12-4 OOPC 对象合作图

这里仍然使用信息传递的沟通方式。

12.3 以 OOPC 实现：使用 Turbo C

现编写 OOPC 程序来实现上述的系统分析与设计，并在 Turbo C 环境里执行。其步骤如下。

Step-1 编写 Light 类。

小灯 Light 的定义文件

```
/* cx12-lig.h   */
#ifndef LIGHT_H
#define LIGHT_H
#include "lw_oopc.h"
```

```
CLASS(Light)
{
  int state;
  void (*init)(void*);
    int (*get_state)(void*);
    void (*set_light)(void*, int flag);
};
#endif
```

小灯 Light 的实现文件

```
/* cx12-lig.c  */
#include <stdio.h>
#include "cx12-lig.h"

static void init( void *t )
{
  Light* cthis = (Light*) t;
  cthis->state = 0;
}

static int get_state(void* t)
{
  Light* cthis = (Light*) t;
  return cthis->state;
}

static void  set_light(void* t, int flag)
{
  Light* cthis = (Light*) t;
  cthis->state = flag;
  if(cthis->state == 0)
    printf("LIGHT_OFF!\n");
  else
    printf("LIGHT_ON!\n");
}

CTOR(Light)
  FUNCTION_SETTING(init, init);
  FUNCTION_SETTING(get_state, get_state);
  FUNCTION_SETTING(set_light, set_light);
END_CTOR
```

Step-2 编写 Switch 类。

开关 Switch 的定义文件

```
/*   cx12-sw.h   */
#ifndef SWITCH_H
#define SWITCH_H
#include "lw_oopc.h"
#include "cx12-lig.h"

CLASS(Switch)
{
  int state;
  Light* light_obj;
  void (*init)(void*, Light*);
  int (*get_state)(void*);
  void  (*set_switch)(void*);
```

```
};
#endif
```

开关 Switch 的实现文件

```
/*  cx12-sw.c  */
#include <stdio.h>
#include "cx12-sw.h"

static void init(void *t, Light* light)
{
  Switch* cthis = (Switch*) t;
  cthis->light_obj = light;
  cthis->state = 0;
}

static int get_state(void* t)
{
  Switch* cthis = (Switch*) t;
  return cthis->state;
}

static void  set_switch(void* t)
{
  int st;
  Light* light;
  Switch* cthis = (Switch*) t;
  cthis->state = !(cthis->state);
  light = cthis->light_obj;
  st = light->get_state(light);
  if (st == 1)
      light->set_light(light, 0);
  else
      light->set_light(light, 1);
}

CTOR(Switch)
  FUNCTION_SETTING(init, init);
  FUNCTION_SETTING(get_state, get_state);
  FUNCTION_SETTING(set_switch, set_switch);
END_CTOR
```

Step-3 编写 main()函数如下。

main()函数

```
/*  cx12-a01.c   */
#include <stdio.h>
#include "cx12-lig.h"
#include "cx12-sw.h"

int main()
{
  Light* light = (Light*)LightNew();
  Switch* wall = (Switch*)SwitchNew();
  Switch* door = (Switch*)SwitchNew();
  light->init(light);
  wall->init(wall, light);
  door->init(door, light);
  /*  press PSW1  */
```

```
  wall->set_switch(wall);
  /*  press PSW2  */
  door->set_switch(door);
  /*  press PSW2  */
  door->set_switch(door);
  /*  press PSW1  */
  wall->set_switch(wall);
  getchar();
  return 0;
}
```

Step-4　编写 cx12-a01.prj 文件。

cx12-a01.prj 文件

```
        cx12-lig.c
        cx12-sw.c
        cx12-a01.c
```

Step-5　编译并执行，出现画面如图 12-5 所示。

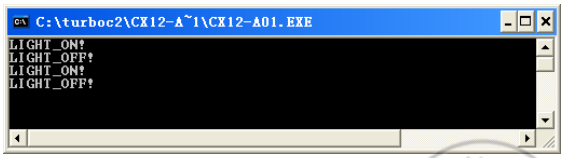


图 12-5

12.4　以 OOPC 实现：使用 VC++ 2005

上一节的示例是在 DOS 环境执行的，没有搭配 Windows 图形画面。由于上一节的 OOPC 程序码是属于标准的 ANSI-C 代码，除了 `main()`函数之外，其他的 `lw_oopc.h`、`Light.h`、`Light.c`、`Switch.h` 和 `Switch.c` 程序代码都可以原封不动地移入 VC++环境中编译和执行。

以下就将这个 VC++项目的程序代码全部列出来，让你理解 OOPC 代码是具有高度可移植性的。编写 VC++的 OOPC 程序的步骤如下：

Step-1　开启一个 win32 的 VC++项目，如图 12-6 所示。

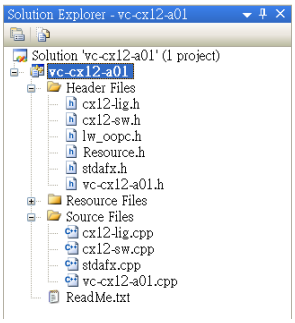


图 12-6

Step-2 加入 lw_oopc.h 宏文件。

lw_oopc.h 代码

```
/*  lw_oopc.h */
/* 这就是 MISOO 团队所设计的 C 宏*/
#include "malloc.h"
#ifndef LOOPC_H
#define LOOPC_H

#define CLASS(type)\
typedef struct type type; \
struct type

#define CTOR(type) \
void* type##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));

#define CTOR2(type, type2) \
void* type2##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));

#define END_CTOR return (void*)t;  };
#define FUNCTION_SETTING(f1, f2)  t->f1 = f2;
#define IMPLEMENTS(type) struct type type
#define INTERFACE(type) \
typedef struct type type; \
struct type
#endif
/*    end    */
```

Step-3 编写 Light 类。

小灯 Light 的定义文件

```
/* cx12-lig.h   */
#ifndef LIGHT_H
#define LIGHT_H
#include "lw_oopc.h"

CLASS(Light)
{
  int state;
  void (*init)(void*);
  int (*get_state)(void*);
  void (*set_light)(void*, int flag);
};
#endif
```

小灯 Light 的实现文件

```
/* cx12-lig.c  */
#include "StdAfx.h"
#include <stdio.h>
#include "cx12-lig.h"
```

```
extern void signal_to_light_UI(LPCWSTR);
static void init( void *t )
{
  Light* cthis = (Light*) t;
  cthis->state = 0;
}

static int get_state(void* t)
{
  Light* cthis = (Light*) t;
  return cthis->state;
}

static void  set_light(void* t, int flag)
{
  Light* cthis = (Light*) t;
  cthis->state = flag;
  if(cthis->state == 1)
     signal_to_light_UI(_T("ON"));
  else
     signal_to_light_UI(_T("OFF"));
}

CTOR(Light)
  FUNCTION_SETTING(init, init);
  FUNCTION_SETTING(get_state, get_state);
  FUNCTION_SETTING(set_light, set_light);
END_CTOR
```

Step-4 编写 Switch 类如下。

开关 Switch 的定义文件

```
/*   cx12-sw.h   */
#ifndef SWITCH_H
#define SWITCH_H
#include "lw_oopc.h"
#include "cx12-lig.h"

CLASS(Switch)
{
  int state;
  Light* light_obj;
  void (*init)(void*, Light*);
  int (*get_state)(void*);
  void  (*set_switch)(void*);
};
#endif
```

开关 Switch 的实现文件

```
/*  cx12-sw.c  */
#include "StdAfx.h"
#include <stdio.h>
#include "cx12-sw.h"

static void init(void *t, Light* light)
{
```

```
  Switch* cthis = (Switch*) t;
  cthis->light_obj = light;
  cthis->state = 0;
}

static int get_state(void* t)
{
  Switch* cthis = (Switch*) t;
  return cthis->state;
}

static void  set_switch(void* t)
{
  int st;
  Light* light;
  Switch* cthis = (Switch*) t;
  cthis->state = !(cthis->state);
  light = cthis->light_obj;
  st = light->get_state(light);
  if (st == 1)
      light->set_light(light, 0);
  else
      light->set_light(light, 1);
}

CTOR(Switch)
  FUNCTION_SETTING(init, init);
  FUNCTION_SETTING(get_state, get_state);
  FUNCTION_SETTING(set_switch, set_switch);
END_CTOR
```

Step-5 编写 Windows UI 程序代码如下。

以 Windows 画面上的 UI Control 来模拟墙上和门上的 Switch 及两个电灯。

在 Windows 环境中，实际执行以测验 3 个软件对象的行为及合作情形。这个步骤主要是测试 `Toggle Switch` 和 `Light` 两个类的正确性，借由 UI Control 发出事件或信息给对象。如图 12-7 所示。

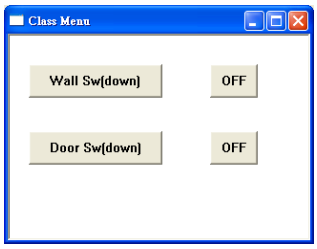


图 12-7

Win32 的 GUI 定义/实现文件

```
// vc-cx12-a01.cpp : Defines the entry point for the application.
#include "stdafx.h"
#include "vc-cx12-a01.h"
#include "cx12-lig.h"
```

```
#include "cx12-sw.h"

//-----------------------------------------
extern void* LightNew();
extern void* SwitchNew();
Light *light;
Switch *wall, *door;
//-----------------------------------------
#define MAX_LOADSTRING 100
HINSTANCE hInst;                          // current instance
TCHAR szTitle[MAX_LOADSTRING];            // The title bar text
TCHAR szWindowClass[MAX_LOADSTRING];      // the main window class name

// Forward declarations of functions included in this code module:
ATOM             MyRegisterClass(HINSTANCE hInstance);
BOOL             InitInstance(HINSTANCE, int);
LRESULT CALLBACK   WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK   About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY _tWinMain(HINSTANCE hInstance,   HINSTANCE hPrevInstance,
                       LPTSTR lpCmdLine, int nCmdShow)
{
   UNREFERENCED_PARAMETER(hPrevInstance);
   UNREFERENCED_PARAMETER(lpCmdLine);
   MSG msg;   HACCEL hAccelTable;
   // Initialize global strings
   LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
   LoadString(hInstance, IDC_VCCX12A01, szWindowClass, MAX_LOADSTRING);
   MyRegisterClass(hInstance);

   // Perform application initialization:
   if (!InitInstance (hInstance, nCmdShow))  return FALSE;
   hAccelTable = LoadAccelerators(hInstance,
     MAKEINTRESOURCE(IDC_VCCX12A01));
   // Main message loop:
   while (GetMessage(&msg, NULL, 0, 0))
   {  if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
      {   TranslateMessage(&msg);  DispatchMessage(&msg);  }
   }
   return (int) msg.wParam;
}

ATOM MyRegisterClass(HINSTANCE hInstance)
{
   WNDCLASSEX wcex;
   wcex.cbSize  = sizeof(WNDCLASSEX);
   wcex.style   = CS_HREDRAW | CS_VREDRAW;
   wcex.lpfnWndProc = WndProc;
   wcex.cbClsExtra  = 0;
   wcex.cbWndExtra  = 0;
   wcex.hInstance   = hInstance;
   wcex.hIcon       = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_VCCX12A01));
   wcex.hCursor     = LoadCursor(NULL, IDC_ARROW);
   wcex.hbrBackground    = (HBRUSH)(COLOR_WINDOW+1);
   wcex.lpszMenuName     = MAKEINTRESOURCE(IDC_VCCX12A01);
   wcex.lpszClassName    = szWindowClass;
   wcex.hIconSm = LoadIcon(wcex.hInstance,
     MAKEINTRESOURCE(IDI_SMALL));
   return RegisterClassEx(&wcex);
```

```
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{  HWND hWnd;
   hInst = hInstance; // Store instance handle in our global variable
   hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
          CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
   if (!hWnd)  return FALSE;
   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);
   return TRUE;
}

HWND hWndButton1, hWndButton2, hWndButton3, hWndButton4;
int st;  char ss[30];
void signal_to_light_UI(LPCWSTR ss)
{  SetWindowText(hWndButton3, ss); SetWindowText(hWndButton4, ss); }

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   int wmId, wmEvent;    PAINTSTRUCT ps;  HDC hdc;
   switch (message)
   { case WM_CREATE: {
          HINSTANCE hInstance = (HINSTANCE)GetWindowLong(hWnd,
                                GWL_HINSTANCE);
          light = (Light*)LightNew();     light->init(light);
          wall = (Switch*)SwitchNew();    wall->init(wall, light);
          door = (Switch*)SwitchNew();    door->init(door, light);
          //-------------------------------------
          hWndButton1= CreateWindowEx( 0, _T("BUTTON"), _T("Wall Sw(down)"),
          WS_VISIBLE | WS_CHILD, 20, 30, 140, 35,
          hWnd, (HMENU) IDB_BUTTON1, hInstance, NULL);

          hWndButton2= CreateWindowEx(0, _T("BUTTON"), _T("Door Sw(down)"),
          WS_VISIBLE | WS_CHILD, 20, 100, 140, 35,
          hWnd, (HMENU) IDB_BUTTON2, hInstance, NULL);

          hWndButton3= CreateWindowEx(0, _T("BUTTON"), _T("OFF"),
          WS_VISIBLE | WS_CHILD, 210, 30, 50, 35,
          hWnd, (HMENU) IDB_BUTTON3, hInstance, NULL);

          hWndButton4= CreateWindowEx(0, _T("BUTTON"), _T("OFF"),
          WS_VISIBLE | WS_CHILD, 210, 100, 50, 35,
          hWnd,(HMENU) IDB_BUTTON4, hInstance, NULL);
    }
   break;
   case WM_COMMAND:
      wmId    = LOWORD(wParam);
      wmEvent = HIWORD(wParam);
      // Parse the menu selections:
      switch (wmId)
      {
        case IDB_BUTTON1: {
           switch (HIWORD(wParam)){
             case BN_CLICKED:
                 wall->set_switch(wall);
              st = wall->get_state(wall);
             if (st == 1)
                    SetWindowText(hWndButton1, _T("Wall Sw(up)"));
```

```
                else
                   SetWindowText(hWndButton1, _T("Wall Sw(down)"));
                break;
             }
          }
          break;
          case IDB_BUTTON2: {
             switch (HIWORD(wParam)) {
               case BN_CLICKED:
                door->set_switch(door);
                st = door->get_state(door);
                if (st == 1)
                   SetWindowText(hWndButton2, _T("Door Sw(up)"));
                else
                   SetWindowText(hWndButton2, _T("Door Sw(down)"));
                break;
              break;
             }}
          break;
       case IDM_ABOUT:
          DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
          break;
       case IDM_EXIT:
          DestroyWindow(hWnd);
          break;
       default:
          return DefWindowProc(hWnd, message, wParam, lParam);
       }
       break;
   case WM_PAINT: hdc = BeginPaint(hWnd, &ps);   EndPaint(hWnd, &ps); break;
   case WM_DESTROY: PostQuitMessage(0);  break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
// Message handler for about box.
INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
   UNREFERENCED_PARAMETER(lParam);
   switch (message)
   {  case WM_INITDIALOG:  return (INT_PTR)TRUE;
   case WM_COMMAND:
      if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
      { EndDialog(hDlg, LOWORD(wParam));  return (INT_PTR)TRUE; }
      break;
   }
   return (INT_PTR)FALSE;
}
```

Step-6 执行。

写好了程序，就可执行该程序来检验各软件对象（类）的正确性和弹性。例如，在 Windows XP 上执行这个 OOPC 程序。一开始，呈现如图 12-8 所示画面，左上角的按钮代表墙壁上的开关，左下角的按钮代表门上的开关，右边的两个按钮各代表一盏灯。

按下墙壁上的开关，即<Wall Sw(down)>按钮，灯就亮了，如图 12-9 所示。

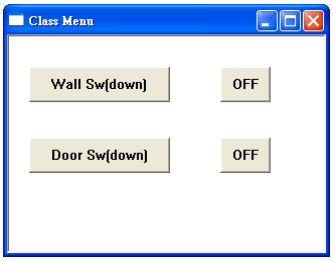


图 12-8

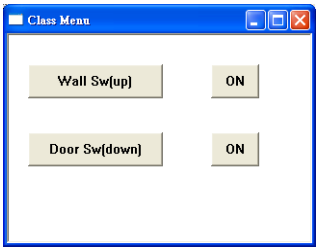


图 12-9

再按门上的开关，即<Door Sw(down)>按钮，灯就熄灭了，如图 12-10 所示。

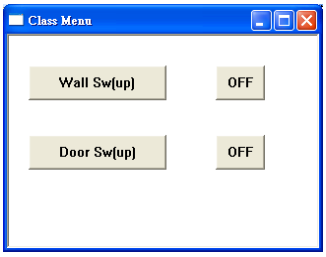


图 12-10

再按门上的开关，即<Door Sw(up)>按钮，灯又亮起来了，如图 12-11 所示。

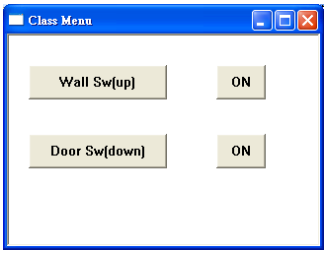


图 12-11

如此一直循环下去。此程序除了 main.c 与 Windows 平台有关之外，Switch 和 Light 两个类代码都是跨平台的。

第 13 章　对象沟通实例

——数学向量与矩阵对象

13.1　以向量类封装 1 维数组

13.2　以矩阵类封装 2 维数组

13.1 以向量类封装 1 维数组

13.1.1 定义 Vector 类

数学的向量概念可以使用 C 的数组来表示。例如，<3, 6, 9>是个向量，可用 1 维数组表示如下：

```
      int v[3] = {3, 6, 9};
      printf("%d\n", v[1]);
```

根据 OOP 的精神，比较好的做法应该是将 v[]数组视为向量对象的内部数据，并进而设计出向量类，如下：

```
/*  cx13-01.c    */
#include "stdio.h"
#include "lw_oopc.h"

CLASS(Vector)
{
  int v[3];
  void (*display)(void*, int);
};

static void print(void* t, int k)
{
  Vector* cthis = (Vector*)t;
  printf("%d\n", cthis->v[k]);
}

CTOR(Vector)
  FUNCTION_SETTING(display, print);
END_CTOR

int main()
{
  Vector *px = (Vector*)VectorNew();
  px->v[0] = 3;  px->v[1] = 6;  px->v[2] = 9;
  px->display = print;
  px->display(px, 1);
  getchar();
  return 0;
}
```

Vector 对象里含有 1 个 1 维数组，如图 13-1 所示。

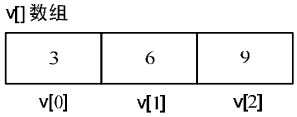


图 13-1

使用 OOPC 定义了 Vector 类之后，代码就更清晰易懂了。

13.1.2 运用 malloc()库函数

上述指令 int v[3]; 会分派内存空间来储存 3 个整数值。其实，它能用 malloc()库函数来取代，而且会具有更大的弹性。例如：

```
/*  cx13-02.c  */
#include <stdio.h>
#include "lw_oopc.h"

CLASS(Vector)
{
  int *pv;
  void (*init)(void*);
  void (*display)(void*, int);
};

static void init(void* t)
{
  Vector* cthis = (Vector*) t;
  cthis->pv = (int*)malloc(3 * sizeof(int));
  cthis->pv[0] =3;  cthis->pv[1] = 6;   cthis->pv[2] = 9;
}

static void print(void* t, int k)
{
  Vector* cthis = (Vector*)t;
  printf("%d\n", cthis->pv[k]);
}

CTOR(Vector)
  FUNCTION_SETTING(init, init);
  FUNCTION_SETTING(display, print);
END_CTOR

int main()
{
  Vector *px = (Vector*)VectorNew();
  px->init(px);
  px->display(px, 2);
  getchar();
  return 0;
}
```

Vector 对象里含有 1 个 1 维数组，如图 13-2 所示。

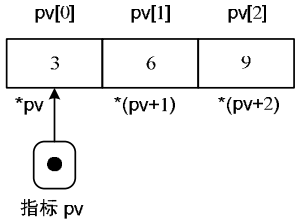


图 13-2

以上两种用法并没有好坏之分，完全视需要和你的习惯而定了。

13.1.3 运用#define 语句

上述 Vector 对象内只能存入整数向量值，如果运用#define 将能设计出更为通用的 Vector 对象。

```
/*  cx13-03.c  */
#include <stdio.h>
#include "lw_oopc.h"
#define VTYPE int

CLASS(Vector)
{
  VTYPE *pv;
  void (*init)(void*, VTYPE*, int);
  VTYPE (*get)(void*, int);
};

static void initialize(void* t, VTYPE* from, int n)
{
  int i;
  Vector* cthis = (Vector*) t;
  cthis->pv = (VTYPE*)malloc(n * sizeof(VTYPE));
  for(i=0; i<n; i++)
    cthis->pv[i] =from[i];
}

static VTYPE get_value(void* t, int k)
{
  Vector* cthis = (Vector*)t;
  return cthis->pv[k];
}

CTOR(Vector)
  FUNCTION_SETTING(init, initialize);
  FUNCTION_SETTING(get, get_value);
END_CTOR

int main()
{ Vector *px = (Vector*)VectorNew();
  int x[] = { 3, 4, 5, 6, 7};
  px->init(px, x, 5);
  printf("%d\n", px->get(px, 3));
  getchar();
  return 0;
}
```

只要改变 VTYPE 常数宏就能让 Vector 对象含有其他类型的向量值。例如：

```
/*  cx13-04.c  */
#include <stdio.h>
#include "lw_oopc.h"
#define VTYPE char

CLASS(Vector)
```

```
{
  VTYPE *pv;
  void (*init)(void*, VTYPE*, int);
  VTYPE (*get)(void*, int);
};

static void initialize(void* t, VTYPE* from, int n)
{
  int i;
  Vector* cthis = (Vector*) t;
  cthis->pv = (VTYPE*)malloc(n * sizeof(VTYPE));
  for(i=0; i<n; i++)
      cthis->pv[i] =from[i];
}

static VTYPE get_value(void* t, int k)
{
  Vector* cthis = (Vector*)t;
  return cthis->pv[k];
}

CTOR(Vector)
  FUNCTION_SETTING(init, initialize);
  FUNCTION_SETTING(get, get_value);
END_CTOR

int main()
{
  Vector *px = (Vector*)VectorNew();
  char z[]= "my dog";
  px->init(px, z, 7);
  printf("%c\n", px->get(px, 1));
  printf(px->pv);
  getchar();
  return 0;
}
```


此程序输出 y。

```
      my dog
```

以上说明了，当 VTYPE 代替 int 时，Vector 对象就能存放整数向量值。当 VTYPE 代替 char 时，Vector 对象就能存放字符串（即字符向量）了。

13.1.4 运用 void*指针

上述例子里，Vector 对象都含有向量元素的值。运用 void*指针可以让 Vector 对象内含有向量元素的指针值：

```
/*  cx13-05.c  */
#include <stdio.h>
#include "lw_oopc.h"

CLASS(Vector)
{
  void **pv;
```

```
  void (*init)(void*, int);
  void (*set)(void*, void*, int);
  void* (*get)(void*, int);
};

static void initialize(void* t, int n)
{
  Vector* cthis = (Vector*) t;
  cthis->pv = (void**)malloc(n * sizeof(void*));
}

static void set_value(void* t, void* from, int k)
{
  int i;
  Vector* cthis = (Vector*) t;
  cthis->pv[k] =from;
}

static void* get_value(void* t, int k)
{
  Vector* cthis = (Vector*)t;
  return cthis->pv[k];
}

CTOR(Vector)
  FUNCTION_SETTING(init, initialize);
  FUNCTION_SETTING(set, set_value);
  FUNCTION_SETTING(get, get_value);
END_CTOR

/*---------------------------------*/
CLASS(Rectangle)
{
  double width;
  double length;
};
CTOR(Rectangle)
END_CTOR

int main()
{
  Rectangle *pr, *obj;
  Vector *px = (Vector*)VectorNew();
  px->init(px, 3);

  pr = (Rectangle*)RectangleNew();
  pr->width = 3.5;
  pr->length =35.0;
  px->set(px, pr, 0);

  pr = (Rectangle*)RectangleNew();
  pr->width = 4.5;
  pr->length =45.0;
  px->set(px, pr, 1);

  pr = (Rectangle*)RectangleNew();
  pr->width = 5.5;
  pr->length =55.0;
  px->set(px, pr, 2);
```

```
  obj = (Rectangle*)px->get(px, 1);
  printf("%5.2f, %5.2f\n", obj->width, obj->length);
  obj = (Rectangle*)px->get(px, 2);
  printf("%5.2f, %5.2f\n", obj->width, obj->length);
  getchar();
  return 0;
}
```

输出：4.5 45.0

5.5 55.0

pv[] 为对象指针的数组，内含 3 个对象指针，分别指向 3 个 Rectangle 对象，如图 13-3 所示。

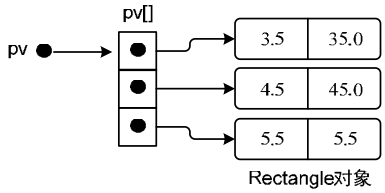


图 13-3

这个 pv[]是通用型的对象指针数组，可内含 n 个对象指针，也可以指向其他类的对象，例如：

```
/*  cx13-06.c  */
#include <stdio.h>
#include "lw_oopc.h"

CLASS(Vector)
{
  void **pv;
  void (*init)(void*, int);
  void (*set)(void*, void*, int);
  void* (*get)(void*, int);
};

static void initialize(void* t, int n)
{
  Vector* cthis = (Vector*) t;
  cthis->pv = (void**)malloc(n * sizeof(void*));
}

static void set_value(void* t, void* from, int k)
{
  int i;
  Vector* cthis = (Vector*) t;
  cthis->pv[k] =from;
}

static void* get_value(void* t, int k)
{
```

```
  Vector* cthis = (Vector*)t;
  return cthis->pv[k];
}

CTOR(Vector)
  FUNCTION_SETTING(init, initialize);
  FUNCTION_SETTING(set, set_value);
  FUNCTION_SETTING(get, get_value);
END_CTOR

/*---------------------------------*/
CLASS(Circle)
{
  double radius;
  double (*cal_area)(void*);
};

static double cal_area(void* t)
{
  Circle *cthis = (Circle*)t;
  return  3.1416 * cthis->radius * cthis->radius;
}

CTOR(Circle)
  FUNCTION_SETTING(cal_area, cal_area);
END_CTOR

int main()
{
  int i;
  Circle *pc, *obj;
  Vector *px = (Vector*)VectorNew();
  px->init(px, 3);

  pc = (Circle*)CircleNew();
  pc->radius = 3.5;
  px->set(px, pc, 0);

  pc = (Circle*)CircleNew();
  pc->radius = 4.5;
  px->set(px, pc, 1);

  pc = (Circle*)CircleNew();
  pc->radius = 10.0;
  px->set(px, pc, 2);

  for(i=0; i<3; i++)
     {
      obj = (Circle*)px->get(px, i);
      printf("%5.2f\n", obj->cal_area(obj));
     }
  getchar();
  return 0;
}
```

此 pv[]为对象指针的数组，内含 3 个对象指针，分别指向 3 个 Circle 的对象，如图 13-4 所示。

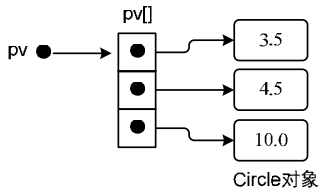


图 13-4

13.2 以矩阵类封装 2 维数组

13.2.1 定义 Matrix 类

数学的矩阵概念可以使用 C 的 2 维数组来表示。例如，<<1,4,7><2,5,8><3, 6, 9>>是矩阵，可用 2 维数组表示如下：

```
int v[3][3] = {{1,4,7},{2,5,8}{3, 6, 9}};
printf("%d\n", v[1][2]);
```

根据 OOP 的精神，比较好的做法应该是将 v[][]数组视为矩阵对象的内部数据，并进而设计出矩阵类，如下：


```
/*  cx13-07.c  */
#include <stdio.h>
#include "lw_oopc.h"

CLASS(Matrix)
{
  int m, n;
  double**v;
  void (*init)(void*, double**, int, int);
  double (*get)(void*, int, int);
};

static void init(void* t, double** A, int p, int q)
{
  int i, j;
  Matrix* cthis = (Matrix*)t;
  cthis->m = p;
  cthis->n = q;
  cthis->v = (double**)malloc(p * sizeof(double *));
  for(i=0; i < p; i++)
  cthis->v[i] = (double*)malloc(q * sizeof(double));
  for(i=0; i<p; i++)
     for(j=0; j<q; j++)
        cthis->v[i][j] = A[i][j];
}
static double get(void*t, int i, int j)
{ Matrix* cthis = (Matrix*) t;
  return cthis->v[i][j];
}
CTOR(Matrix)
```

```
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(get, get)
END_CTOR

void main()
{ Matrix* mat = (Matrix*)MatrixNew();
  double a = 0.0;
  double ** A;
  int i;
  A = (double**)malloc(2 * sizeof(double *));
  for(i=0; i < 2; i++)
      A[i] = (double*)malloc(2 * sizeof(double));
  A[0][0] = 1.0/9.0;
  A[1][1] = 9.0;
  A[0][1] = 1.0/4.0;
  A[1][0] = 4.0;

  mat->init(mat,A,2,2);
  a = mat->get(mat, 0, 1);    printf("a=%7.3f\n", a);
  a = mat->get(mat, 1, 0);    printf("a=%7.3f\n", a);
  getchar();
  return 0;
}
```

13.2.2 Matrix 对象包含 Vector 对象

数学的矩阵概念可以使用 C 的 2 维数组来表示。例如，<<1,4,7><2,5,8><3, 6, 9>>是矩阵，可用 2 维数组表示如下：

```
int v[3][3] = {{1,4,7},{2,5,8}{3, 6, 9}};
printf("%d\n", v[1][2]);
```

根据 OOP 的精神，比较好的做法应该是将 v[][]数组视为矩阵对象的内部数据，并进而设计出矩阵类，如下。

Step-1 设计 Vector 类。

编写 Vector 类定义文件：cx13-vec.h

```
/* cx13-vec.h */
#ifndef VEC_H
#define VEC_H
#include <stdio.h>
#include "lw_oopc.h"

CLASS(Vector)
{
  float *V;
  int size;
  void (*init)(void*);
  void (*init_fi)(void*, float*, int);
  void (*init_i)(void*, int);
  void (*destroy)(void*);
  float* (*get)(void*, int);
  float (*get_val)(void*, int);
```

```
  int   (*EQ)(void*, Vector*);
  void (*set)(void*, float*, int);
  Vector* (*assign)(void*, Vector*);
  void (*mul)(void*, float);
  void (*div)(void*, float);
  void (*PR)(void*);
  int (*indexOfMax)(void*);
  int (*includes)(void*, int);
  };
  #endif
```

编写 Vector 类实现文件：cx13-vec.c

```
/* cx13-vec.c  */
/*           */
#include <stdio.h>
#include "cx13-vec.h"

static void init(void* t)
{
  Vector* cthis = (Vector*)t;
  cthis->V = NULL;
  cthis->size = 0;
}
static void init_fi(void* t, float *from, int n )
{
  int i;
  Vector* cthis = (Vector*)t;
  cthis->V = (float*)malloc(n * sizeof(float));
  cthis->size=n;
  for( i=0;  i<cthis->size; i++ )
      cthis->V[i] = *from++;
}

static void init_i(void* t, int n )
{ int i;
  Vector* cthis = (Vector*)t;
  cthis->V = (float*)malloc(n * sizeof(float));
  cthis->size = n;
  for( i=0;  i<cthis->size; i++ )
      cthis->V[i] = 1.0;
}

static void destroy(void* t)
{
  Vector* cthis = (Vector*)t;
  if( cthis->V != NULL ) free(cthis->V);
}

static float* get(void* t, int i )
{
  Vector* cthis = (Vector*)t;
  return &cthis->V[i];
}
static float get_val(void* t, int i )
{
  Vector* cthis = (Vector*)t;
  return cthis->V[i];
}
```

```
static int EQ(void* t, Vector* from )
{ float k;  int i;
  Vector* cthis = (Vector*)t;
  if(cthis->size != from->size) return 0;
  for( i=0; i<cthis->size; i++ )
  {
     k = cthis->V[i] - from->get_val(from, i);
     if( k > 0.00001 || k < -0.00001 )  return 0;
  }
  return 1;
}
static void set(void* t, float *from, int n )
{
  int i;
  Vector* cthis = (Vector*)t;
  if( cthis->V != NULL )  free(cthis->V);
  cthis->V = (float*)malloc(n * sizeof(float));
  cthis->size = n;
  for(i=0;  i<cthis->size; i++ )
     cthis->V[i] = *from++;
}
static Vector* assign(void *t, Vector* from)
{ int i;
  Vector* cthis = (Vector*)t;
  if( cthis->V != NULL )   free(cthis->V);
  cthis->size = from->size;
  cthis->V = (float*)malloc(cthis->size * sizeof(float));
  for( i=0;  i<from->size; i++ )
     cthis->V[i] = from->get_val(from, i);
  return cthis;
 }
static void mul(void* t, float ratio )
{ int i;
  Vector* cthis = (Vector*)t;
  for( i=0;  i<cthis->size; i++ )
        cthis->V[i] =  cthis->V[i] * ratio;
}
static void div(void* t, float ratio )
{ int i;
  Vector* cthis = (Vector*)t;
  for( i=0;  i < cthis->size; i++ )
     cthis->V[i] /= ratio;
}
static void PR(void* t)
{ int i;
  Vector* cthis = (Vector*)t;
  for( i=0;  i<cthis->size; i++ )
     printf("%4.1f ", cthis->V[i]);
  printf("\n");
}
static int indexOfMax(void* t)
{ int i;
  Vector* cthis = (Vector*)t;
  float max = *(cthis->V);
  int k =0;
  for( i=0;  i<cthis->size; i++ )
  if( max < cthis->V[i] )
  {   max = cthis->V[i];
     k=i;
```

```
  }
  return k;
}
int includes(void* t,  int v )
{ int i;
  Vector* cthis = (Vector*)t;
  int k =0;
  for( i=0; i<cthis->size; i++ )
     if( v == (int)(cthis->V[i]) ) k = 1;
  return k;
}

CTOR(Vector)
  FUNCTION_SETTING(init, init);
  FUNCTION_SETTING(init_fi, init_fi);
  FUNCTION_SETTING(init_i, init_i);
  FUNCTION_SETTING(destroy, destroy);
  FUNCTION_SETTING(get, get);
  FUNCTION_SETTING(get_val, get_val);
  FUNCTION_SETTING(EQ, EQ);
  FUNCTION_SETTING(set, set);
  FUNCTION_SETTING(assign, assign);
  FUNCTION_SETTING(mul, mul);
  FUNCTION_SETTING(div, div);
  FUNCTION_SETTING(PR, PR);
  FUNCTION_SETTING(indexOfMax, indexOfMax);
  FUNCTION_SETTING(includes, includes);
END_CTOR
```

Step-2 设计 Matrix 类。

编写 Matrix 类定义文件：cx13-mat.h

```
/* cx13-mat.h  */
#ifndef MAT_H
#define MAT_H
#include <stdio.h>
#include "lw_oopc.h"
#include "cx13-vec.h"

CLASS(Matrix)
{
  int m, n;
  Vector **p;
  void (*initialize)(void*);
  void (*init_fii)(void*, float*, int, int);
  void (*assign)(void*, Matrix*);
  Vector* (*row)(void*, int);
  Vector* (*col)(void*, int);
  void (*destroy)(void*);
  void (*setRow)(void*, Vector*, int);
};
#endif
```

编写 Matrix 类实现文件：cx13-mat.c

```
/* cx13-mat.c  */
/*           */
#include <stdio.h>
```

```
#include "cx13-mat.h"

static void initialize(void* t)
{
  Matrix* cthis = (Matrix*)t;
  cthis->m = 0;
  cthis->n = 0;
}

static void init_fii(void* t, float* from, int row, int col )
{ Vector* pv;  int i;
  Matrix* cthis = (Matrix*)t;
  cthis->m = row+1;
  cthis->n = row+col+1;
  cthis->p = (Vector**)malloc(cthis->m * sizeof(Vector*));

  for( i=0; i<cthis->m; i++, from += cthis->n )
        {   pv = (Vector*)VectorNew();
            pv->init_fi(pv, from, cthis->n);
            cthis->p[i] = pv;
        }
}

static void destroy(void* t)
{ int i; Vector*pv;
  Matrix* cthis = (Matrix*)t;
  if(cthis->m)
  {
  for( i=0; i<cthis->m; i++)
     {   pv = cthis->p[i];
         pv->destroy(pv);
     }
     free(cthis->p);
  }
}

static void setRow(void* t, Vector* from, int row)
{
  Matrix* cthis = (Matrix*)t;
  Vector* pv;
  pv = cthis->p[row];
  pv->assign(pv, from);
}

static Vector* row(void* t, int i)
{
  Matrix* cthis = (Matrix*)t;
  return cthis->p[i];
}

static Vector* col(void* t, int i)
{
  int k;  Vector* pv2;
  Matrix* cthis = (Matrix*)t;
  Vector* pv = (Vector*)VectorNew();
  pv->init_i(pv, cthis->m);

  for(k=0; k<cthis->m; k++)
  {
```

```
      pv2 = cthis->p[k];
      *pv->get(pv,k) = pv2->get_val(pv2,i);
  }
  return pv;
}

static void assign(void* t, Matrix* px)
{
  Vector *pv, *pv2;  int i;
  Matrix* cthis = (Matrix*)t;
  cthis->m = px->m;
  cthis->n = px->n;
  cthis->p = (Vector**)malloc(cthis->m * sizeof(Vector*));

  for( i=0; i<cthis->m; i++)
  {
      pv2 = px->row(px, i);
      pv = (Vector*)VectorNew();
      pv->init(pv);
      pv->assign(pv, pv2);
      cthis->p[i] = pv;
  }
}

CTOR(Matrix)
  FUNCTION_SETTING(initialize, initialize);
  FUNCTION_SETTING(init_fii, init_fii);
  FUNCTION_SETTING(destroy, destroy);
  FUNCTION_SETTING(setRow, setRow);
  FUNCTION_SETTING(row, row);
  FUNCTION_SETTING(col, col);
  FUNCTION_SETTING(assign, assign);
END_CTOR
```

Step-3 编写主程序：cx13-app.c。

```
/* cx13-app.c  */
#include "cx13-vec.h"
#include "cx13-mat.h"

int main()
{ Matrix* py; Vector *pr, *pv;  int k;
  Matrix* px = (Matrix*)MatrixNew();
  float vv[28] = { 1.1, 2.2, 3.3, 4.4, 15.5, 6.5, 7.5,
          0.5, 1.5, 2.5, 13.5, 4.5, 5.5, 6.9,
          7.1, 7.2, 17.3, 7.4, 7.5, 7.6, 7.7,
          12.6, 13.7, 14.8, 25.9, 16.0, 17.1, 18.2  };
  px->init_fii(px, vv, 3, 3);
  py = (Matrix*)MatrixNew();
  py->assign(py, px);
  pr = px->row(px, 2);
  pr->PR(pr);
  k = pr->indexOfMax(pr);
  printf("indexOfMax = %d\n", k);
  /*-----------------------------*/
  py->setRow(py, pr, 1);
  pv = py->col(py, 1);
  pv->PR(pv);
  /*-----------------------------*/
```

```
  px->destroy(px);
  getchar();
  return 0;
}
```

Step-4 编写 TurboC cx13-app.prj 文件，其内容为：

```
cx13-vec.h
cx13-vec.c
cx13-mat.h
cx13-mat.c
cx13-app.c
```

Step-5 在 TurboC 环境里执行 cx13-app.prj 文件，其结果如图 13-5 所示。

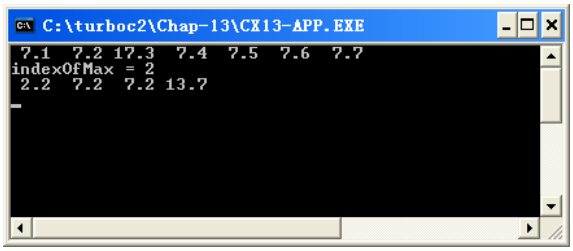


图 13-5

第 14 章　认识接口（Interface）

14.1 如何定义接口

外界环境快速变化，人们又生活在这种环境之中，并使用软件，所以软件的需求也是常常变化的。因此，为了让软件能长寿，其内部组织也必须呈现有机次序，就是将软件本身的组织依“稳定”到“善变”（有弹性）程度层次分明地区分出来，借稳定部分来支持有弹性部分的快速汰旧更新。并尽量避免波及稳定部分，软件就呈现出生机盎然的景象了。于是对象（object）概念就派上用场了。

对象用来划分出稳定与善变的界线，这个界线就通称为接口（Interface），就像树叶的叶柄部分，是叶与枝的界线。再如车轮的外胎是善变的部分，为了便于更换外胎，轮框与外胎含有个明确的界线（接口）。所以对象的主要用途是定义明确的接口，把软件里局部而善变的部分包含在接口之内。当有必要时，就把接口内的善变部分更换掉，让软件添增一份活力。接口规划良好，就易于更换；反之，接口不良，则常会牵一发而动全身，因失去弹性而僵化，就免不了被环境所淘汰。

设计对象的目的是要得到明确的接口；设计对象的方法是将善变与稳定部分并无直接关联的东西隐藏（Hide）在接口内，而接口上只留下与稳定部分有关的东西；这个过程就称为“抽象”（Abstraction），亦即从接口内的善变部分抽出与接口外有关联的部分，摆在接口上，而无关部分摆在接口内。稳定与善变部分划分得愈理想，就愈可得到简单的接口，对象就愈易于更换，软件就愈能蜕变成长。因此，软件设计的好坏就决定于程序员对稳定与善变的区分经验和能力。

理想的接口应是简单明确的，对象才容易更换。欲得到单纯的接口，最常见的方法是让对象所包装的细节部分具有高度紧密的相关性，也就是所谓的“凝聚性”（Cohesion）；并且让内部细节与外部的相关程度愈低愈好，就像陶渊明的世外桃源一样，除了少许信道之外，尽可能与世隔绝开来。由于接口所表达的是对象内部与外部的沟通关系（信道），所以内外愈分隔，愈互为独立，则信道就愈单纯，接口就愈单纯。在计算机程序中，主要包括两部分：

- 数据：即数据结构（Data Structure），又称为数据成员。
- 函数：即算法（Algorithm），又称为成员函数。

其中，数据是函数处理的对象，且函数之间会互相调用（call 或 invoke）。例如：

```
int total;
void add(int x)
   {  total = total + x;  }
```

函数 add()对 total 和 x 进行“加法”运算。再如：

```
void change()
{
   total = 0;
```

```
    add(100);
    total = total * 2;
}
```

change()函数调用了 add()函数。因此，创造强凝聚性的途径有以下两种：

（1）**以函数为中心**，将常互相调用的有关函数结合起来，构成较独立的单元，统称为模块（Module）。而不考虑数据的角色。

（2）**以数据为中心**，将直接处理到该数据的各有关函数结合起来，成为独立的单元，统称为对象（Object）。

对象将数据及有关的函数包装起来，但允许对象内的函数能调用其他对象内的函数。在 20 世纪 70 至 80 年代，人们采用第一种方法，称为结构化（Structured）方法，但并未成功。有关结构化的历史，在此不加详谈。逐渐地，人们改用第二种途径，称为面向对象（Object-Oriented）方法，到目前为止，已有些良好表现，但仍有待更加努力。

简而言之，对象的内部细节就包含两部分：

（1）**数据成员**，统称为对象的数据属性（Attribute）。

（2）**成员函数**，用来直接处理的程序，统称为对象的“方法”（Method），如图 14-1 所示。

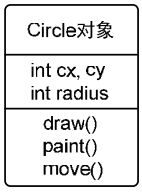


图 14-1

此对象以数据为核心，并含有相关的函数 `draw()`、`paint()`及 `move()`，它们皆能直接使用到内部的数据。对象的函数之间也可以互相调用，当然也可调用其他对象内的函数。一个理想的情形是：对象内的数据结构相当复杂，对象的函数互相调用情形相当复杂；但对象内函数与其他对象函数的互相调用情形，应尽量简单明确。这样，就可以得到单纯的对象接口（Object Interface）了，这更有利于对象的更换，软件也更易于维护和成长。于是对象接口的用途是：

让别的对象知道它们能调用此对象的哪些函数。

例如：

```
INTERFACE( ICircle )
{
    void (*draw)();
    void (*move)();
}
```

上面这段代码说明别的对象只能调用 Circle 对象的 draw()及 move()函数，而禁止调用其他函数如 paint()；更不能存取对象内的数据值。此对象内的函数能调用别的对象内的函数，这就是所谓的对象之间可以沟通了。不过，此对象能调用另一对象的哪些函数，就得依循对方对象接口的规定了。

由于沟通皆通过接口，遵循接口的规定，使得对象内部的细节与其他对象内部的细节之间呈现相当的独立性。当必须替换掉某对象时，不会造成牵一发而动全身的现象。既然对象易于更换，那么把软件中善变的部分包装隐藏于对象的内部，对象接口就成为善变与稳定部分的明确界限了。那么，软件就能如同有生命的树木一般，易于整合有机次序，能在快速变化的气候环境中历久常青。

在 OOP 程序中，通常借由下述两项定义来叙述对象：

（1）**类定义**（class definition）——叙述对象内部含有哪些数据成员，以及含哪些有关的函数。

（2）**接口定义**（interface definition）——叙述对象内的函数中，有哪些是允许别的对象经由此接口而调用的。如何让别的对象知道与此对象沟通。

例如，想描述“长方形”对象，可定义如下：

```
INTERFACE(IA)
{
  double (*cal_area)(void*);
  double (*cal_side)(void*);
};

CLASS(Rectangle)
{
  IMPLEMENTS(IA);
  void (*init)(void*, double, double);
  double length;
  double width;
};
```

类的定义说明了“长方形”对象内部含有两个数据成员及三个成员函数。接口的定义说明了其他对象能通过 IA 接口而调用“长方形”对象的 cal_area()及 cal_perimeter()函数，但不能调用 paint()函数，也不能直接存取 length 及 width 的数据值。此时也称 Rectangle 类实现（Implement）了 IA 接口，其表示于类定义里如下：

```
CLASS(Rectangle)
{
   IMPLEMENTS(IA);   /*  Rectangle 类实现(Implement)了 IA 接口  */
   ……
}
```

14.2 多个类实现同一接口

两个以上的类可以实现同一个接口，例如：

```
INTERFACE(IA)
{
  double (*cal_area)(void*);
  double (*cal_side)(void*);
};

CLASS(Circle)
{
  IMPLEMENTS(IA);
  void (*init)(void*, double);
  double radius;
};

CLASS(Square)
{
  IMPLEMENTS(IA);
  void (*init)(void*, double);
  double side;
};
```

由于 Circle 与 Square 类皆提供一样的接口 IA，所以能替换对象。也就是说，凡是支持同一接口的对象皆可以互换，例如有两个类：圆形（Circle）和正方形（Square），而且他们都支持 IA 接口的话，那么两者的对象就可以互换了。请看其 OOPC 代码。

编写 IA 接口代码

```
/* cx14-ia.h */
#ifndef IA_H
#define IA_H

INTERFACE(IA)
{
  void (*init)(void*, double);
  double (*cal_area)(void*);
  double (*cal_perimeter)(void*);
};
#endif
```

这个接口含有三个函数。

编写 Cicle 类

```
/* cx14-cir.c */
#include "lw_oopc.h"
#include "cx14-ia.h"

CLASS(Circle)
{
  IMPLEMENTS(IA);
  double radius;
```

```
};

static void init(void* t, double r)
{
  Circle* cthis = (Circle*)t;
  cthis->radius = r;
}

static double cal_area(void* t)
{
  Circle* cthis = (Circle*)t;
  return (3.1416 * cthis->radius * cthis->radius);
}

static double cal_perimeter(void* t)
{
  Circle* cthis = (Circle*)t;
  return (2 * 3.1416 * cthis->radius);
}

CTOR(Circle)
  FUNCTION_SETTING(IA.init, init)
  FUNCTION_SETTING(IA.cal_area, cal_area)
  FUNCTION_SETTING(IA.cal_perimeter, cal_perimeter)
END_CTOR
```

编写 Square 类

```
/* cx14-sq.c */
#include "lw_oopc.h"
#include "cx14-ia.h"

CLASS(Square)
{
  IMPLEMENTS(IA);
  double side;
};

static void init(void* t, double s)
{
  Square* cthis = (Square*)t;
  cthis->side = s;
}

static double cal_area(void* t)
{
  Square* cthis = (Square*)t;
  return (cthis->side * cthis->side);
}

static double cal_perimeter(void* t)
{
  Square* cthis = (Square*)t;
  return (4 * cthis->side);
}

CTOR(Square)
  FUNCTION_SETTING(IA.init, init)
  FUNCTION_SETTING(IA.cal_area, cal_area)
```

```
  FUNCTION_SETTING(IA.cal_perimeter, cal_perimeter);
END_CTOR
```

以上两个类都支持 IA 接口。

编写主程序：

```
/* cx14-ap1.c */
#include "stdio.h"
#include "lw_oopc.h"
#include "cx14-ia.h"

void print_area(IA* pi)
{
  printf("area=%6.2f\n", pi->cal_area(pi));
}

int main()
{ IA *pc, *ps;
  pc = (IA*)CircleNew();
  pc->init(pc, 10.0);
  print_area(pc);

  ps = (IA*)SquareNew();
  ps->init(ps, 10.0);
  print_area(ps);
  getchar();
  return 0;
}
```

这个程序复用（Reuse）了 print_area()函数，也就是说，print_area()可以跟任何支持 IA 接口的对象沟通。目前这个例子的 main()内容较多，而 print_area()内容较少，可能你看不出 print_area()的复用价值。当 print_area()内容较复杂或较专业时，其复用价值就迅速提高了。在下一节的售票机例子里，将展现出这种价值。

编写 TurboC Project：cx14-ap1.prj，其内容为：

```
       cx14-cir.c
       cx14-sq.c
       cx14-ap1.c
```

编译及执行，输出结果：

```
       area = 324.16
       area = 100.00
```

原来 pi 指向 Circle 的对象，将 pi 所指的对象换成 Square 的对象时，print_area()仍然可以执行。这 print_area()就像随身听音响，而 Circle 和 Square 就像两种不同品牌的电池，但是同一号（同样大小）的电池。同种类的电池与其随身听音响的接触面（即接口）是一样的，它们具有共同的接口，就能与随身听有完美的接触和结合了。再如，家里的灯泡坏掉了，在购买新灯泡前，一定会先看看旧灯泡的灯帽是几号的，只要购买同号灯帽的灯泡，就必定可装上旧的灯座。其原因是：灯帽是灯泡与灯座的接触面（接口），只要接口相符合，旧灯泡就必定能与旧灯座整合了。

14.3 以接口实现多态性（Polymorphism）

具有相同接口的一群对象，称之为多态对象。多态对象是可以互换的，如图 14-2 所示。

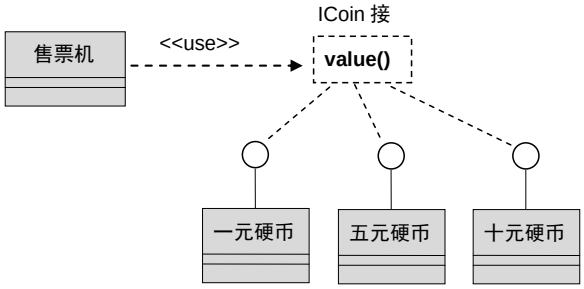


图 14-2

这两个类实现 ICoin 接口，都有 value()函数，而且各类实现的方式（即代码内容）都不相同，因而产生不同的行为。像 value()这种函数，称为“多态函数“（Polymorphic Function）。多态函数通常为多态对象皆能接受的信息。如设*d1、*d5、*d10 分别为这三类的对象指针，则*d1、*d5、*d10 皆能接受此 value()信息，例如指令：

```
d1->value();
d5->value();
d10->bonus();
```

但各调用不同的 value()实现代码。多态对象常存于一个数组（Array）中，如图 14-3 所示。

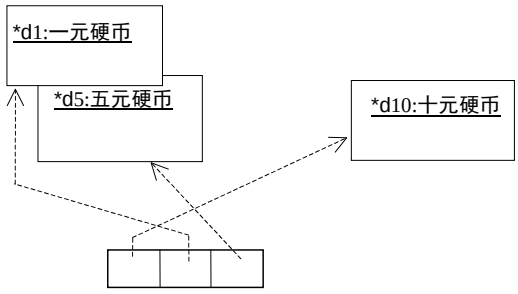


图 14-3

或者存于链表（List）里，如图 14-4 所示。

由此数组或链表，可知推销部门包括一个 1 元硬币、一个 5 元硬币和一个 10 元硬币。因他们为同部门的人员，故宜摆于同一数组或链表中。然而，他们却属于不同类。多态性能提供软件设计者极大的弹性与方便，因为计算机会根据对象的类而自动寻找适当的函数。例如：

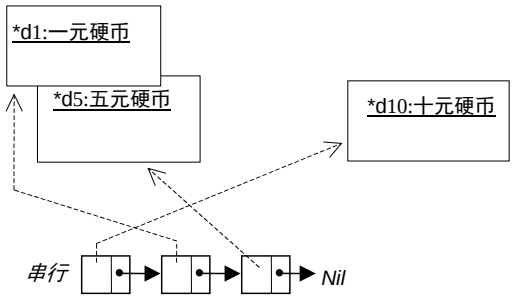


图 14-4

```
        ICoin *pi;
        pi = (ICoin*)one_dollarNew();
        pi->init();
        pi->value(pi);

        pi = (ICoin*)five_dollarNew();
        pi->init();
        pi->value(pi);
```

因 pi 指向 one_dollar 类的对象，计算机自动调用 one_dollar 类内的 value() 函数。当 pi 转而指向 five_dollar 类的对象时，计算机自动调用 five_dollar 类内的 value()函数。此时，可设计一个函数，让售票机能接受及分辨硬币体系的对象。例如：

```
        售票机.feedCoin( 1元硬币指针 )
        售票机.feedCoin( 5元硬币指针 )
        售票机.feedCoin( 10元硬币指针 )
```

feedCoin()为“售票机”类的函数，能接受硬币对象的指针，然后根据对象的类而自动寻找适当的函数。这不但给予软件设计者方便；更重要的是它也带给用户莫大的方便——无论用户拥有 1 元、5 元还是 10 元的硬币，皆可投入售票机。请看其 OOPC 代码：

Step-1 编写 ICoin 接口代码。

```
/*  c14-con.h  */
#ifndef COIN_H
#define COIN_H
#include "lw_oopc.h"

INTERFACE(ICoin)
{
    void (*init)();
    double (*value)();
};
#endif
```

Step-2 编写 1 元、5 元或 10 元的硬币类的代码。

one_dollar 类

```
/*  c14-one.c  */
#include <stdio.h>
#include "cx14-con.h"

CLASS(one_dollar)
{
    IMPLEMENTS(ICoin);
    int k;
};
static void init() {}
static double value()
    { return 1.0;   }

CTOR(one_dollar)
  FUNCTION_SETTING(ICoin.init, init)
  FUNCTION_SETTING(ICoin.value, value)
END_CTOR
```

five_dollar 类

```
/*  cx14-fiv.c  */
#include <stdio.h>
#include "cx14-con.h"

CLASS(five_dollar)
{
  IMPLEMENTS(ICoin);
};

static void init(){}
static double value()
{  return 5.0;   }

CTOR(five_dollar)
  FUNCTION_SETTING(ICoin.init, init)
  FUNCTION_SETTING(ICoin.value, value)
END_CTOR
```

ten_dollar 类

```
/*  cx14-ten.c  */
#include <stdio.h>
#include "cx14-con.h"

CLASS(ten_dollar)
{
  IMPLEMENTS(ICoin);
};
static void init(){}
static double value()
  {  return 10.0;   }

CTOR(ten_dollar)
  FUNCTION_SETTING(ICoin.init, init)
  FUNCTION_SETTING(ICoin.value, value)
END_CTOR
```

以上三个类都支持 ICoin 接口。现在准备将这三种不同的硬币对象投入售票机里，此时一般程序员会想到：售票机类里应该会用一些 if 指令来判断对象的类，才能正确计算出所投入的总金额。其实不然，这是多态性带来的好处，让售票机类既简洁又清晰。

Step-3 定义售票机类。

```
/*  cx14-vm.h  */
#include <stdio.h>
#include "cx14-con.h"

CLASS(VendingMachine)
{
  void  (*init)(void*);
  void (*feedCoin)(void*, ICoin*);
  double (*getTotal)(void*);
  int index;
  ICoin* array[10];
};
```

Step-4 编写售票机类。

```
/*  cx14-vm.c  */
#include <stdio.h>
#include "cx14-con.h"
#include "cx14-vm.h"

static void init(void*t)
{
  VendingMachine* cthis = (VendingMachine*)t;
  cthis->index = 0;
}

static void feedCoin(void*t, ICoin* c)
{ double v;
  VendingMachine* cthis = (VendingMachine*)t;
  cthis->array[cthis->index] = c;
  cthis->index++;
}
static double getTotal(void*t)
{
  int i, sum; ICoin *pc;
  VendingMachine* cthis = (VendingMachine*)t;
  sum = 0;
  for(i=0; i<cthis->index; i++)
  {
     pc = cthis->array[i];
     sum += pc->value(pc);
  }
return sum;
}
CTOR(VendingMachine)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(feedCoin, feedCoin)
  FUNCTION_SETTING(getTotal, getTotal)
END_CTOR
```

Step-5 编写主程序。

```
/*  cx14-ap2.c   */
#include "lw_oopc.h"
#include "cx14-con.h"
#include "cx14-vm.h"

int main()
{
  ICoin *c1, *c5, *c10;  double v;
  VendingMachine* vm = (VendingMachine*)VendingMachineNew();
  vm->init(vm);

  c1 = (ICoin*)one_dollarNew();
  c1->init();
  vm->feedCoin(vm, c1);

  c5 = (ICoin*)five_dollarNew();
  c5->init();
  vm->feedCoin(vm, c5);

  c10 = (ICoin*)ten_dollarNew();
  c10->init();
  vm->feedCoin(vm, c10);

  c5 = (ICoin*)five_dollarNew();
  c5->init();
  vm->feedCoin(vm, c5);

  vm->feedCoin(vm, c1);

  v = vm->getTotal(vm);
  printf("total=%6.2f\n", v);

  getchar();
  return 0;
}
```

编写 TurboC Project：cx14-ap2.prj，其内容为：

```
cx14-one.c
cx14-fiv.c
cx14-ten.c
cx14-vm.c
cx14-ap2.c
```

编译及执行，输出结果：

```
total = 22.0
```

14.4 一个类实现多个接口

一个类也可以实现两个以上的接口，也就是说，一个对象可以同时具有多个接口。就像麦当劳（McDonald's）餐厅，提供不同的接口——内食购买柜台及汽车外购窗口，来为不同的用户服务。如图 14-5 所示。

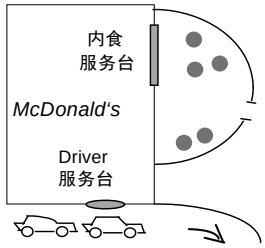


图 14-5

也常表示如图 14-6 所示。

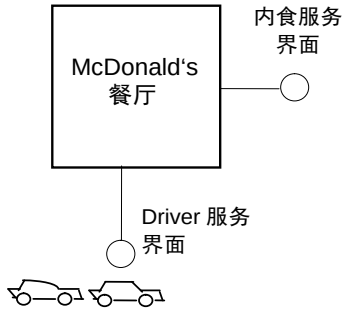


图 14-6

因此较新的计算机编程语言，如 VB、JAVA、C#等纷纷采用“类”与“接口”分开定义，让一个对象可提供多个不同的接口，呈现多面性（Multiple Interface）的对象。请看一个 OOPC 的例子。

编写 IB 接口代码

```
/*  cx14-ib.h  */
#ifndef IB_H
#define IB_H
INTERFACE(IB)
{
  void (*init)(void*, double, double);
  double (*cal_area)(void*);
};
#endif
```

编写 IC 接口代码

```
/*  cx14-ic.h  */
#ifndef IC_H
#define IC_H

INTERFACE(IC)
{
  void (*init)(void*, double, double);
  double (*cal_perimeter)(void*);
```

```
};
#endif
```

一个类可以支持上述 IB 和 IC 两个接口。为了让 IB 与 IC 两个接口互换，将定义一个 IALL 接口，它包含 IB 和 IC。如果 pb 正指向某个对象的 IB 接口时，你可以顺利地从 pb 值转到指向 IALL 的 pa 值，然后再从 pa 值转到指向 IC 的 pc 值。在本示例里，你将会看到这个精彩的转换。

编写 IALL 接口代码

```
/*  cx14-all.h  */
#ifndef IALL_H
#define IALL_H
#include "cx14-ib.h"
#include "cx14-ic.h"

INTERFACE(IALL)
{
  IMPLEMENTS(IB);
  IMPLEMENTS(IC);
};
#endif
```

接下来，就编写 Rect 类来支持 IB 和 IC 接口了。

编写 Rect 类代码

```
/*  cx14-rec.c  */
#include "stdio.h"
#include "lw_oopc.h"
#include "cx14-ib.h"
#include "cx14-ic.h"

CLASS(Rect)
{ IMPLEMENTS(IB);
  IMPLEMENTS(IC);
  double  length;
  double  width;
  void (*display)(char*, double);
};

void initialize(void*, double, double);
double calculate_area(void*);
double calculate_perimeter(void*);
void pr(char*, double);

CTOR(Rect)
  FUNCTION_SETTING(IB.init, initialize)
  FUNCTION_SETTING(IB.cal_area, calculate_area)
  FUNCTION_SETTING(IC.init, initialize)
  FUNCTION_SETTING(IC.cal_perimeter, calculate_perimeter)
  FUNCTION_SETTING(display, pr)
END_CTOR

static void initialize(void* t, double len, double wid)
{ Rect* cthis = (Rect*)t;
  cthis->length = len;
```

```
  cthis->width = wid;
}
static double calculate_area(void* t)
{ Rect* cthis = (Rect*)t;
  double v;
  v = cthis->length * cthis->width;
  cthis->display("area", v);
  return v;
}
static double calculate_perimeter(void* cthis)
{ Rect* t = (Rect*)cthis;
  double v = (t->length + t->width)*2;
  t->display("perimeter", v);
  return v;
}
static void pr(char* str, double v )
{
  printf("%s=%7.2f\n", str, v);
}
```

这里 Rect 类针对 IB 和 IC 接口去实现它。

编写主程序代码

```
/*  cx14-ap3.c  */
#include <stdio.h>
#include "lw_oopc.h"
#include "cx14-all.h"

void main()
{
  double a;
  IALL* pa;   IB* pb;   IC* pc;
  /* ------------------------- */
  pb = (IB*)RectNew();
  pb->init(pb, 10.0, 10.0);
  a = pb->cal_area(pb);
  /* ------------------------- */
  pa = (IALL*)pb;
  pc = &(pa->IC);
  a = pc->cal_perimeter(pa);
  getchar();
}
```


此 pb 指向该对象的 IB 接口，如果像要 pc 也指同一对象，但不同的接口，如图 14-7 所示。

通过 pb 和 pc 可以调用到对象的不同函数。例如，通过 pb 调用 cal_area()，也可以通过 pc 调用 cal_perimeter()函数。

编写 TurboC Project：cx14-ap3.prj，其内容为

```
      cx14-rec.c
      cx14-ap3.c
```

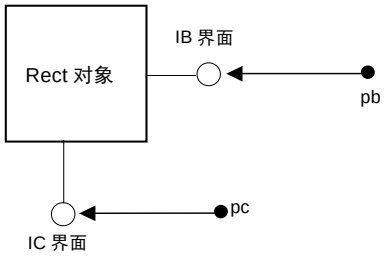


图 14-7

编译及执行，输出结果

```
area=100.0
perimeter=40.0
```

第 15 章　接口应用实例

——以串联电池为例

15.1 电池接口的用意

在日常生活中，大家都用过手电筒，也用过电池、灯泡等对象。手电筒是大对象，其内包含有电池、灯泡等小对象。在一般的手电筒里，常可看到串联的电池，如图 15-1 所示。

图 15-1

大家也都知道每块电池有正负两个接口，如图 15-2 所示。

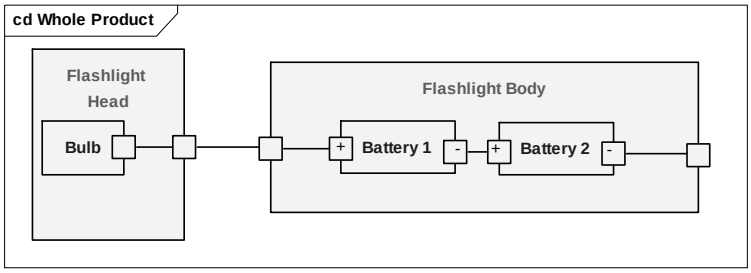


图 15-2

在本章里将不讨论灯泡（或灯头）部分，而把焦点放在电池接口的设计上，并说明如何基于接口而顺利将电池串联起来。

15.2 设计电池接口

从图 15-2 中可看出有两种主要对象：手电筒内壳（FlashLight Body，简称 FlashLight）类和电池（Battery Cell，简称 Cell）类。如图 15-3 所示。

将图 15-3 落实到 OOPC 程序上，FlashLight Body 将对应到 FlashLight 类的对象，而 Cell 将对应到 PanasonicCell 或 CatCell 类的对象。ICell 就是 PanasonicCell 或 CatCell 类必须支持的接口。而 FlashLight 对象则负责将不同类的 Cell 对象以串联方式组合起来。而图 15-3 中的手电筒将对应到 OOPC 程序里的 main()主程序，它负责生成 FlashLight 及 Cell 对象，并且将 Cell 对象放入 FlashLight 对象里面。如果想以标准图形表示，可绘制 UML 图

如下（见图 15-4）。

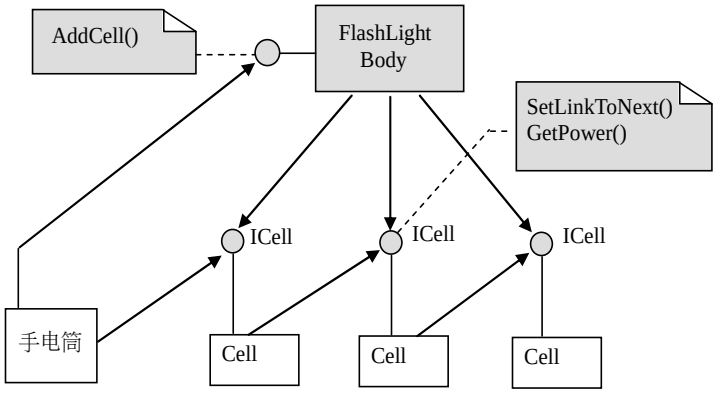


图 15-3

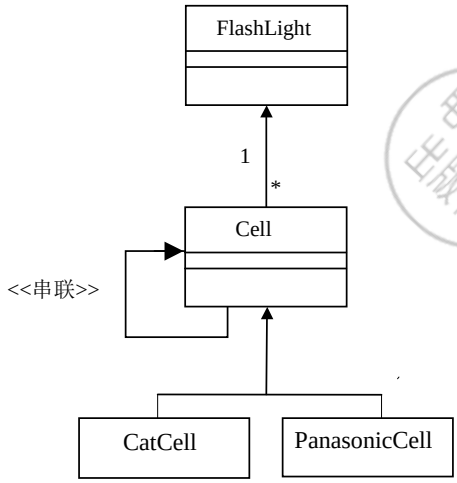


图 15-4

藉由精致的接口设计，就能随时组合出串联式手电筒了，如图 15-5 所示。

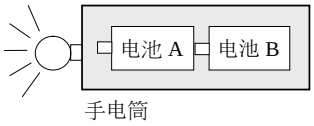


图 15-5

在此例子中，除了设计 ICell 接口之外，还会设计一个 ILight 接口，如图 15-6 所示。

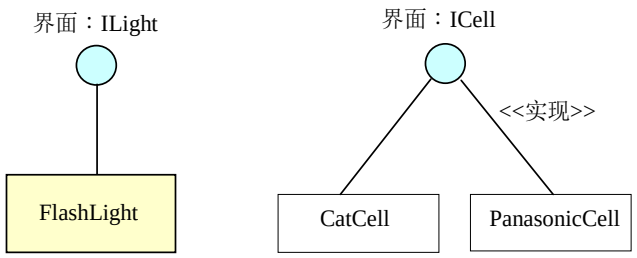


图 15-6

由于 CatCell 和 PanasonicCell 两个类都支持 ICell 接口，所以其对象具有多态性。这会让 FlashLight 的设计简单许多。

15.3 以 OOPC 实现接口设计

上一节的接口设计能顺畅地落实到 OOPC 程序里。其实现步骤如下：

Step-1 先定义 ICell 和 ILight 两个接口。

编写 ICell 接口代码

```
/* cx15-ic.h */
#ifndef IC_H
#define IC_H
INTERFACE(ICell)
{
  void (*init)(void*);
  void (*SetLinkToNext)(void*, ICell*);
  int (*GetPower)(void*) ;
};
#endif
```

编写 ILight 接口代码

```
/* cx15-il.h */
#ifndef IL_H
#define IL_H

INTERFACE(ILight)
{
  void (*init)(void*);
  void (*AddCell)(void*, void*);
  int (*Power)(void*);
};
#endif
```

Step-2 定义支持 ICell 接口的电池类。

编写 PanasonicCell 类代码

```
/*  cx15-pan.c   */
#include "lw_oopc.h"
#include "cx15-ic.h"

CLASS(PanasonicCell)
{
  IMPLEMENTS(ICell);
  int pw;
  ICell* next_cell;
};

static void init(void* t)
{
  PanasonicCell* cthis = (PanasonicCell*) t;
  cthis-> next_cell =NULL;
  cthis->pw = 10;
}

static void SetLink(void* t, ICell* nc)
{
  PanasonicCell* cthis = (PanasonicCell*)t;
  cthis->next_cell = nc;
}

static int GetPower(void* t)
{
  PanasonicCell* cthis = (PanasonicCell*) t;
  ICell *pc;
  if( cthis->next_cell == NULL)
  return cthis->pw;
  else
    {
       pc =  cthis->next_cell;
       return (cthis->pw +  pc->GetPower(pc));
    }
}
CTOR(PanasonicCell)
  FUNCTION_SETTING(ICell.init, init);
  FUNCTION_SETTING(ICell.SetLinkToNext, SetLink);
  FUNCTION_SETTING(ICell.GetPower, GetPower);
END_CTOR
```

编写 CatCell 类代码

```
/*  cx15-cat.c   */
#include "lw_oopc.h"
#include "cx15-ic.h"

CLASS(CatCell)
{
  IMPLEMENTS(ICell);
  int pw;
  ICell* next_cell;
};
```

```
static void init(void* t)
{
  CatCell* cthis = (CatCell*) t;
  cthis->next_cell = NULL;
  cthis->pw = 7;
}

static void SetLink(void* t, void* nc)
{
  CatCell* cthis = (CatCell*) t;
  cthis->next_cell = nc;
}

static int GetPower(void* t)
{
  CatCell* cthis = (CatCell*) t;
  ICell *pc;
  if( cthis->next_cell == NULL)
     return cthis->pw;
  else
    {
       pc =  cthis->next_cell;
       return (cthis->pw +  pc->GetPower(pc));
    }
}
CTOR(CatCell)
  FUNCTION_SETTING(ICell.init, init);
  FUNCTION_SETTING(ICell.SetLinkToNext, SetLink);
  FUNCTION_SETTING(ICell.GetPower, GetPower);
END_CTOR
```

Step-3 编写 FlashLight（手电筒）类代码。

```
/*  cx15-lig.c   */
#include "lw_oopc.h"
#include "cx15-ic.h"
#include "cx15-il.h"

CLASS(FlashLight)
{
  IMPLEMENTS(ILight);
  ICell *head, *tail;
};

static void init(void* t)
{
  FlashLight* cthis = (FlashLight*)t;
  cthis->head = NULL;
  cthis->tail = NULL;
}

static void AddCell(void*t, ICell* cell)
{ ICell *pc;
  FlashLight* cthis = (FlashLight*)t;
  if( cthis->head == NULL)
  {
     cthis->head = cell;
     cthis->tail =cthis-> head;
  }
  else
```

```
    {
        pc = cthis->tail;
        pc->SetLinkToNext(pc, cell);
        cthis->tail = cell;
    }
}
static int Power(void*t)
{
  FlashLight* cthis = (FlashLight*)t;
  ICell *pc = cthis->head;
  return pc->GetPower(pc);
}

CTOR(FlashLight)
  FUNCTION_SETTING(ILight.init, init);
  FUNCTION_SETTING(ILight.AddCell, AddCell);
  FUNCTION_SETTING(ILight.Power, Power);
END_CTOR
```

Step-4 编写 main()主程序。

```
/*  cx15-app.c   */
#include <stdio.h>
#include "lw_oopc.h"
#include "cx15-ic.h"
#include "cx15-il.h"

int main()
{
  ILight *light;
  ICell *pan1, *pan2, *cat1;
  int pow;

  light = (ILight*)FlashLightNew();
  light->init(light);

  pan1 = PanasonicCellNew();
  pan1->init(pan1);
  pan2 =PanasonicCellNew();
  pan2->init(pan2);
  cat1 = CatCellNew();
  cat1->init(cat1);

  light->AddCell(light, pan1);
  light->AddCell(light, pan2);
  light->AddCell(light, cat1);

  pow = light->Power(light);
  printf("Power = %d\n", pow);
  getchar();
  return 0;
}
```

Step-5 编写 TurboC 的 cx15-app.prj 文件，其内容为：

```
       cx15-pan.c
       cx15-cat.c
       cx15-lig.c
```

cx15-app.c

Step-6 在 TurboC 环境里执行 cx15-app.prj 文件。

此程序将两块松下牌电池装入手电筒，并继续将 1 块黑猫牌电池装入手电筒，手电筒计算出总电量，当您继续装入电池时，手电筒会累积而显示出总电量。

第 16 章　集合对象链表（Linked List）

16.1　认识集合对象

16.2　以 OOPC 实现 LList 集合类

16.3　应用实例说明

16.1 认识集合对象

在第 13 章里介绍了 Vector 类及其对象。当它能容纳多态性对象时，我们通常称之为集合（Collection）类或对象。本章将以 Vector 和 LList 为例说明如何以 OOPC 设计及实现好用的集合类。

俗语说：“宰相肚里能撑船”，意谓宰相肚量大，能容纳多态化的意见，且容量无限。反观计算机软件中常见的“数组”（Array），包含若干有序的变量。只是数组内的变量，其类型必须一致，是一元化的集合体。例如：

```
int  x[10];
int  *p[10];
```

前者表示 x[0]，x[1]，……，x[9]皆为 int 类型，即此数组只能包含 int 类型的数据，不能容纳别的类型的数据。后者表示 p[0]，p[1]，……，p[9]皆为 int 的指针，只能指向 int 类型的数据，不得指向别的类型的数据。请看看“结构”（Struct）吧！它比数组更具多态性，能包含多种类型的数据，例如：

```
struct  account
{char name;
 int  number;
 float amount;
};
```

包含了 char、int 及 float 3 种不同类型的数据，已具多态性的雏型了。不过，尚有美中不足之处：除了 char、int、float 之外，account 结构无法容纳其他类型的数据；而且肚量有限，不能延伸扩充。为了支持 OOP 的重要观念——多态性（Polymorphism），需要具备多态性且可伸展的集合对象（Container object）。

集合对象如“菜篮子”般，能装各种蔬菜水果。简而言之，集合对象是多态性的数据结构（Polymorphic data structure），像篮子一样，能包含多态性对象。还记得吗？凡是具有相同接口的对象皆为多态性对象；这群对象最适合放入集合对象里。在各种数据结构中，“数组”（Array）是最简单的。除了数组外，还有向量（Vector）、链表（Linked list）、树（Tree），等等。其用途是将相关的对象集合起来，表达对象间的关系，使得计算机软件易于掌握对象间的复杂关系。例如：一支棒球队，含各种成员，且各有不同的角色，像经理、教练及球员等各有各的职责。由于他们属于同一支球队，各自拥有重要角色，所以是息息相关的。就像蜜蜂一般，是具有多态性的族群（如工蜂、女王蜂等）。那么，多态性的数据结构——“集合对象”便表达这种复杂而多元的关系，担任起“牵红线”的任务。现在请看如何用一般数组表达上述棒球队的组织吧！其 OOPC 程序如下：

```
/*  cx16-ap1.c   */
#include "stdio.h"
#include "lw_oopc.h"
```

```
CLASS(BaseballTeam)
{
  void (*init)(void*);
  void (*setManager)(void*, char*);
  void (*setCoach)(void*, char*);
  void (*addPlayer)(void*, char*);
  void (*display)(void*);
  char *manager, *coach, *players[20];
  int i;
};

static void init(void* t)
   {
     BaseballTeam* cthis = (BaseballTeam*) t;
     cthis->i=0;
     cthis->manager = NULL;
     cthis->coach = NULL;
   }
static void setManager(void* t, char *manager_name)
   {
     BaseballTeam* cthis = (BaseballTeam*) t;
     cthis->manager = manager_name;
   }
static void setCoach(void*t, char *coach_name)
   {
     BaseballTeam* cthis = (BaseballTeam*) t;
     cthis->coach = coach_name;
   }
static void addPlayer(void* t, char *player_name )
   {
     BaseballTeam* cthis = (BaseballTeam*) t;
     cthis->players[cthis->i] = player_name;
     cthis->i++;
   }

void display(void*t)
   {
     int k;
     BaseballTeam* cthis = (BaseballTeam*) t;
     printf("%s\n", cthis->manager);
     printf("%s\n", cthis->coach);
     for(k=0; k < cthis->i; k++ )
        printf("%s\n", cthis->players[k]);
   }
CTOR(BaseballTeam)
  FUNCTION_SETTING(init, init);
  FUNCTION_SETTING(setManager, setManager);
  FUNCTION_SETTING(setCoach, setCoach);
  FUNCTION_SETTING(addPlayer, addPlayer);
  FUNCTION_SETTING(display, display);
END_CTOR

int main()
{
  BaseballTeam*  RedSock = BaseballTeamNew();
  RedSock->init(RedSock);
  RedSock->setManager(RedSock, "Jamis King");
  RedSock->setCoach(RedSock, "David Wang" );
```

```
  RedSock->addPlayer(RedSock, "Jim Lin" );
  RedSock->addPlayer(RedSock, "Alvin Kao" );
  RedSock->addPlayer(RedSock, "John Coppin" );
  RedSock->display(RedSock);
  getchar();
  return 0;
}
```

players[]含 20 个指针可指向字符串，但它至多可指向 20 个球员。add()令 players[]中的指针指向一个新球员。在 main()程序里，RedSock 是 BaseballTeam 类的对象，代表“红袜”队，目前队上有一位经理、一位教练及 3 位球员。此程序输出：

```
      Jamis King
      David Wang
      Jim Lin
      Alvin Kao
      John Coppin
```

此方法的缺点为：

（1）必须先确定球员人数。例如，*players[20]的 20 是如何决定的呢？万一有支球队拥有 22 位球员，又该如何呢？这就面临了“留太多浪费，太少又担心不够用”的两难局面。

（2）必须费神处理变量 i。i 是数组的下标（Subscript）。这里只含有一个数组 players[]，所以只需要一个 i 即可。然而，若当某支球队聘请多位经理及多位教练时，岂非要许多个像 i 这种下标呢？只是增加软件的复杂性，增加软件设计师负担罢了。

此时，若利用集合对象，上述问题便迎刃而解了。

16.2 以 OOPC 实现 LList 集合类

让我们以 OOPC 编写一个 LList 集合类，代替上述 players[]数组吧！LList 对象内含多态性链表（Linked List），能存不同类的对象，如果这些对象又具有多态性，就更能发挥巨大效益。现将上一节的棒球队示例里的 players[]数组改为更具有弹性的 LList 集合对象。其步骤如下。

Step-1 设计及实现 LList 集合类。

编写 IColl 接口代码

```
/*  llist.h   */
#include "lw_oopc.h"
#ifndef LLIST_H
#define LLIST_H

INTERFACE(IColl)
{
  void (*init)(void*);
  void (*add)(void*, void*);
```

```
  void (*top)(void*);
  void* (*next)(void*);
  void* (*get)(void*, int);
};
#endif
```

编写 LList 类代码

```
/* llist.c   */
#include "stdio.h"
#include "llist.h"

CLASS(ListNode)
{
  void* pItem;
  ListNode* next;
};
CONS(ListNode)
END_CONS
/* ------------------------------ */
CLASS(LList)
{
  IMPLEMENTS(IColl);
  ListNode *head, *tail, *current;
};

void init(void*);
void add(void*, void*);
void top(void*);
void* next(void*);
void* get(void*, int);

CONS(LList);
  FUNCTION_SETTING(IColl.init, init)
  FUNCTION_SETTING(IColl.add, add)
  FUNCTION_SETTING(IColl.top, top)
  FUNCTION_SETTING(IColl.next, next)
  FUNCTION_SETTING(IColl.get, get)
END_CONS

static void init(void* t)
{
  LList* cthis = (LList*)t;
  cthis->head = NULL;
  cthis->tail = NULL;
  cthis->current = NULL;
}

static void add(void* t, void* pi)
{
  LList* cthis = (LList*) t;
  ListNode* pn = (ListNode*)ListNodeNew();
  pn->next = NULL;
  pn->pItem = pi;
  if(cthis->head == NULL)
  {
     cthis->tail = pn;
     cthis->head = pn;
     cthis->current = pn;
```

```
  }
  else
  {
     cthis->tail->next = pn;
     cthis->tail = pn;
     cthis->current = pn;
  }
}

static void top(void* t)
{
  LList* cthis = (LList*) t;
  cthis->current = NULL;
}

static void* next(void* t)
{
  LList* cthis = (LList*) t;
  if(cthis->current == NULL)
  {
     if(cthis->head == NULL) return NULL;
     else
     {
        cthis->current = cthis->head;
        return cthis->current->pItem;
     }
  }
  else
  {
     cthis->current = cthis->current->next;
     if(cthis->current == NULL) return NULL;
     else return cthis->current->pItem;
  }
}

static void* get(void* t, int k)
{
  int i;
  LList* cthis = (LList*) t;
  ListNode* pn;
  if(cthis->head == NULL) return NULL;
  pn = cthis->head;
  for(i=0; i<k; i++)
  {
     pn = pn->next;
     if(pn == NULL) return NULL;
  }
  return pn->pItem;
 }
```

在 IColl 接口里提供了 5 个主要的函数：

（1）init()函数用来设定初始值。

（2）add()把一个对象指针加入 LList 对象里。

（3）top()让 current 指针值归零，如此确保下一个 next()指令能从 LList 里取出第 1 项对象。

（4）next()能依序取出下一项对象。

（5）get(n)取出第 n 项对象。

编写 BaseballTeam 类及 main()的代码

```
/*  cx16-ap2.c   */
#include "stdio.h"
#include "lw_oopc.h"
#include "llist.h"

CLASS(BaseballTeam)
{
  void (*init)(void*);
  void (*setManager)(void*, char*);
  void (*setCoach)(void*, char*);
  void (*addPlayer)(void*, char*);
  void (*display)(void*);
  char *manager, *coach;
  IColl *players;
};

static void init(void* t)
     {
       BaseballTeam* cthis = (BaseballTeam*) t;
       cthis->manager = NULL;
       cthis->coach = NULL;
       cthis->players = (IColl*)LListNew();
       cthis->players->init(cthis->players);
     }
static void setManager(void* t, char *manager_name)
     {
       BaseballTeam* cthis = (BaseballTeam*) t;
       cthis->manager = manager_name;
     }
static void setCoach(void*t, char *coach_name)
     {
        BaseballTeam* cthis = (BaseballTeam*) t;
        cthis->coach = coach_name;
     }
static void addPlayer(void* t, char *player_name )
{
  BaseballTeam* cthis = (BaseballTeam*) t;
  IColl *list = cthis->players;
  list->add(list, (void*)player_name);
}

void display(void*t)
     {
       IColl *list;
       char  *text;
       BaseballTeam* cthis = (BaseballTeam*) t;
       printf("%s\n", cthis->manager);
       printf("%s\n", cthis->coach);
```

```
        list = cthis->players;
        list->top(list);
        text = (char*)list->next(list);
        while(text != NULL )
            {
            printf("%s\n", text);
            text = (char*)list->next(list);
        }
      }
CTOR(BaseballTeam)
  FUNCTION_SETTING(init, init);
  FUNCTION_SETTING(setManager, setManager);
  FUNCTION_SETTING(setCoach, setCoach);
  FUNCTION_SETTING(addPlayer, addPlayer);
  FUNCTION_SETTING(display, display);
END_CTOR

int main()
{
  BaseballTeam*  RedSock = BaseballTeamNew();
  RedSock->init(RedSock);
  RedSock->setManager(RedSock, "Jamis King");
  RedSock->setCoach(RedSock, "David Wang" );
  RedSock->addPlayer(RedSock, "Jim Lin" );
  RedSock->addPlayer(RedSock, "Alvin Kao" );
  RedSock->addPlayer(RedSock, "John Coppin" );
  RedSock->display(RedSock);
  getchar();
  return 0;
}
```

players 的类型为 IColl*，表示 players 是 LList 集合对象的指针。指令 cthis->players = (IColl*)LListNew(); 生成 LList 的集合对象，并令 players 指向此对象，如图 16-1 所示。

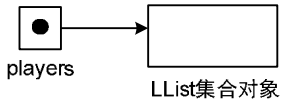


图 16-1

集合对象能容纳不同类的对象，而且不限个数，能视需要而不断地扩充。由于 LList 对象能自动扩大，因此您不必烦恼到底有多少球员。addPlayer()将球员数据统统存入集合对象中。此外，在 display()函数里，使用 while 循环能轻易地显示出集合对象的内容。此程序应输出：

```
       Jamis King
       David Wang
       Jim Lin,
       Alvin Kao,
       John Coppin
```

16.3 应用实例说明

——以并联电池为例

在上一章里，用 OOPC 编写了一个串联电池的例子。在本节里则将它改为用并联方式来组合电池。如图 16-2 所示。

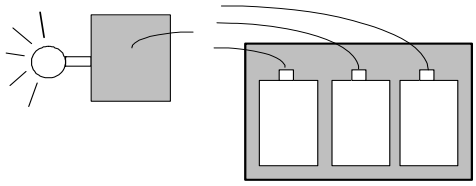


图 16-2

其中，将设计电池的共同接口，来发挥多态性效果，如图 16-3 所示。

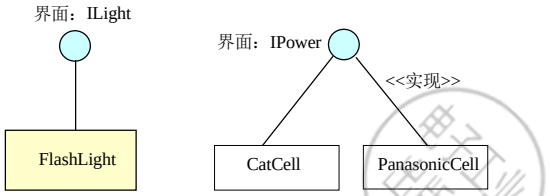


图 16-3

而且，设计一个 Vector 集合类来容纳这些多态性的电池对象。其实现步骤如下：

Step-1 设计 Vector 集合类。

定义 Vector 类

```
/*  cx16-vec.h  */
#include <stdio.h>
#include "lw_oopc.h"

CLASS(Vector)
{
  void **pv;
  int index;
  void (*init)(void*, int);
  void (*Add)(void*, void*);
  int (*GetSize)(void*);
  void* (*Get)(void*, int);
};
```

编写 Vector 类代码

```
/* cx16-vec.c  */
#include "cx16-vec.h"
static void initialize(void* t, int n)
```

```
{
  Vector* cthis = (Vector*) t;
  cthis->index = 0;
  cthis->pv = (void**)malloc(n * sizeof(void*));
}

static void Add(void* t, void* obj)
{
  Vector* cthis = (Vector*) t;
  cthis->pv[cthis->index] = obj;
  cthis->index++;
}

static int GetSize(void* t)
{
  Vector* cthis = (Vector*)t;
  return cthis->index;
}

static void* Get(void* t, int k)
{
  Vector* cthis = (Vector*)t;
  return cthis->pv[k];
}

CTOR(Vector)
  FUNCTION_SETTING(init, initialize);
  FUNCTION_SETTING(Add, Add)
  FUNCTION_SETTING(GetSize, GetSize);
  FUNCTION_SETTING(Get, Get);
END_CTOR
```

在 Vector 类里提供了 4 个主要的函数：

（1）init()函数用来设定初始值。

（2）Add()把一个对象指针加入 Vector 对象里。

（3）GetSize()得出目前含有多少个对象。

（4）Get(n)取出第 n 项对象。

Vector 类与前面 LList 的区别在于：LList 可随内部对象个数增加而不断扩大容量。但是 Vector 在生成时必须设定其容量，然而，Vector 的 Add()执行效率较佳。各有长处，视需要而决定采用哪个。

Step-2 设计 IPower 和 ILight 接口。

编写 IPower 接口代码

```
/* cx16-ip.h */
#ifndef IP_H
#define IP_H

INTERFACE(IPower)
{
```

```
  void (*init)(void*);
  int (*GetPower)(void*) ;
};
#endif
```

编写 ILight 接口代码

```
/* cx16-il.h */
#ifndef IL_H
#define IL_H

INTERFACE(ILight)
{
  void (*init)(void*);
  void (*AddCell)(void*, void*);
  int (*Power)(void*);
};
#endif
```

Step-3 定义合乎 IPower 接口的电池类。

编写 PanasonicCell 类代码

```
/*  cx16-pan.c   */
#include "lw_oopc.h"
#include "cx16-ip.h"

CLASS(PanasonicCell)
{
  IMPLEMENTS(IPower);
  int pw;
};

static void init(void* t)
{
  PanasonicCell* cthis = (PanasonicCell*)t;
  cthis->pw = 2;
}

static int GetPower(void* t)
{
  PanasonicCell* cthis = (PanasonicCell*)t;
  return cthis->pw;
}

CTOR(PanasonicCell)
  FUNCTION_SETTING(IPower.init, init);
  FUNCTION_SETTING(IPower.GetPower, GetPower);
END_CTOR
```

编写 CatCell 类代码

```
/*  cx16-cat.c   */
#include "lw_oopc.h"
#include "cx16-ip.h"

CLASS(CatCell)
{
  IMPLEMENTS(IPower);
```

```
  int watt;
};

static void initialize( void* t )
{
  CatCell* cthis = (CatCell*)t;
  cthis->watt = 3;
};

static int get_power(void* t)
{
  CatCell* cthis = (CatCell*)t;
  return cthis->watt;
}

CTOR(CatCell)
  FUNCTION_SETTING(IPower.init, initialize);
  FUNCTION_SETTING(IPower.GetPower, get_power);
END_CTOR
```

Step-4 编写 FlashLight（手电筒）类代码。

FlashLight（手电筒）类代码

```
/*  cx16-lig.c   */
#include "lw_oopc.h"
#include "cx16-ip.h"
#include "cx16-il.h"
#include "cx16-vec.h"

CLASS(FlashLight)
{
  IMPLEMENTS(ILight);
  Vector* cell_list;
};

static void init(void* t)
{
  FlashLight* cthis = (FlashLight*)t;
  Vector* pv = (Vector*)VectorNew();
  pv->init(pv, 10);
  cthis->cell_list = pv;
}

static void AddCell(void*t, void* cp)
{
  FlashLight* cthis = (FlashLight*)t;
  Vector* list = cthis->cell_list;
  list->Add(list, cp);
}
static int Power(void*t)
{
  FlashLight* cthis = (FlashLight*)t;
  IPower *po;
  int sum = 0;
  int i, n;
  Vector* list = cthis->cell_list;
  n = list->GetSize(list);
  for(i=0; i<n; i++)
```

```
    {
      po= (IPower*)list->Get(list, i);
      sum = sum + po->GetPower(po);
    }
  return sum;
}
CTOR(FlashLight)
  FUNCTION_SETTING(ILight.init, init);
  FUNCTION_SETTING(ILight.AddCell, AddCell);
  FUNCTION_SETTING(ILight.Power, Power);
END_CTOR
```

Step-5 编写 main()主程序。

main()主程序

```
/*  cx16-app.c   */
#include <stdio.h>
#include "lw_oopc.h"
#include "cx16-ip.h"
#include "cx16-il.h"

int main()
{
  ILight *light;
  IPower *pan1, *pan2, *cat1;
  int pow;

  light = (ILight*)FlashLightNew();
  light->init(light);

  pan1 = PanasonicCellNew();
  pan1->init(pan1);
  pan2 =PanasonicCellNew();
  pan2->init(pan2);
  cat1 = CatCellNew();
  cat1->init(cat1);

  light->AddCell(light, pan1);
  light->AddCell(light, pan2);
  light->AddCell(light, cat1);

  pow = light->Power(light);
  printf("Power = %d\n", pow);
  getchar();
  return 0;
  }
```

Step-6 编写 TurboC 的 cx16-app.prj 文件。

TurboC 的 cx16-app.prj 文件

```
      cx16-vec.c
      cx16-pan.c
      cx16-cat.c
      cx16-lig.c
      cx16-app.c
```

Step-7 在 TurboC 环境里执行 cx16-app.prj 文件。

本程序表示先装入两块松下牌（PanasonicCell）电池，再装入一块黑猫牌（CatCell）电池，手电筒计算出总电量为 7，并显示出来。当你继续装入电池时，手电筒会累积而计算出总电量。

第 17 章　LW_OOPC 宏的设计思维

——如果你只想使用 lw_oopc.h 头文件来写 OOPC 代码，可以略过本章

——如果你想探究 lw_oopc.h 内容或想修改它时，必须阅读本章

17.1 前言

30 多年来，许多专家都想去改进 C 程序的编写风格，因而有 OOPC 的名词出现，甚至鼎鼎大名的 Object-C 和 C++等新语言也纷纷上市。大家都努力将面向对象（Object-Oriented）观念与 ANSI-C 结合起来，提供较为美好的 C 程序编写风格。在结合的过程中，因为目的的不同，导致结合的结果也大异其趣。其中最大的区别在于：要不要支持类的继承（Inheritance）关系？若支持，其实现方法的效率怎样？例如 C++对继承有完全的支持，但其内部采取虚函数表（Virtual Function Table）机制，让执行效率微微下滑，也让程序稍稍变大。

这样的效率问题，在嵌入式系统开发上就有很大的影响，因为嵌入式系统资源都是斤斤计较的。所以 MISOO 团队就仔细针对嵌入式系统开发的需要，设计了 lw_oopc.h 头文件，确实达到了嵌入式系统开发需要的重要目标：

- 执行效率不能打折扣。
- 程序不能明显变大。
- 必须简单易学。
- 可以不支持类继承，但必须支持接口及多态性。

MISOO 团队经过 4 年多的使用，并仔细修正，而达到非常稳定、可靠的境界，这才开始编写此书来与读者分享。本章就由浅入深，逐步阐述 lw_oopc.h 头文件的深层考虑及其设计思维。当您有必要去修改 lw_oopc.h 头文件时，本章提供给您核心概念和技术基础。

17.2 从 ANSI-C 出发

——以计算长方形面积为例

以标准 ANSI-C 编写代码来计算长方形面积和周长，其代码如下：

```
/* cx17-ap1.c  */
#include "stdio.h"
double cal_area(double length, double width)
{ return length * width; }

double cal_perimeter(double length, double width)
{ return (length + width) * 2; }

/*----------------------------------------*/
int main()
{
  double v;
  v = cal_area(10.5, 20.5);
  printf("area=%7.3f\n", v);

  v = cal_perimeter(10.5, 20.5);
  printf("perimeter=%7.3f\n", v);
```

```
  getchar();
  return 0;
}
```

此 main()传递 length 及 width 两个参数给 cal_area()和 cal_perimeter()两个函数，这样就可以求算出长方形的面积和周长了。

17.3 运用 C 的结构

在 ANSI-C 里，其结构（struct）机制最适合用来支持面向对象的类概念。因此，如果使用 C 的结构（struct）将 length 及 width 封装在一起成为 struct Rectangle 结构，就可以传递结构，会显得更简洁，而且成为 OOPC 的基础，如下代码：

```
/* cx17-ap2.c  */
#include "stdio.h"
#include "malloc.h"

struct Rectangle
{
  double length;
  double width;
};

double cal_area(struct Rectangle* rec)
{
  return rec->length * rec->width;
}

double cal_perimeter(struct Rectangle* rec)
{
  return (rec->length + rec->width) * 2;
}

/*----------------------------------------*/
int main()
{
  struct Rectangle* pr;
  double v;
  pr = (struct Rectangle*)malloc(sizeof(struct Rectangle));
  pr->length = 10.5;
  pr->width = 20.5;

  v = cal_area(pr);
  printf("area=%7.3f\n", v);

  v = cal_perimeter(pr);
  printf("perimeter=%7.3f\n", v);

  getchar();
  return 0;
}
```

这个 struct Rectangle 结构里含有的属性（Attribute）数据，就如同类里的数据成员。

17.4 设计构造器

当你逐渐将 struct Rectangle 视为类时，上一节的示例代码：

```
    struct Rectangle* pr;
    pr = (struct Rectangle*)malloc(sizeof(struct Rectangle));
```

其负责分派内存空间给 pr 所指的对象，这就是 OOP 里的构造器（Constructor）的职责。可以将其独立出来，成为 RectangleNew()函数，如下代码：

```
/*   cx17-ap3.c   */
#include "stdio.h"
#include "malloc.h"

struct Rectangle
{
  double length;
  double width;
};

double cal_area(struct Rectangle* rec)
{
  return rec->length * rec->width;
}

double cal_perimeter(struct Rectangle* rec)
{
  return (rec->length + rec->width) * 2;
}

struct Rectangle* RectangleNew()
{
  struct Rectangle* pr;
  pr = (struct Rectangle*)malloc(sizeof(struct Rectangle));
  return pr;
}
void init(struct Rectangle* cthis, double len, double wid)
{
  cthis->length = len;
  cthis->width = wid;
}
/*----------------------------------------*/
int main()
{
  double v;
  struct Rectangle* pr;
  pr = RectangleNew();
  init(pr, 10.5, 20.5);

  v = cal_area(pr);
  printf("area=%7.2f\n", v);

  v = cal_perimeter(pr);
  printf("perimeter=%7.2f\n", v);

  getchar();
```

```
  return 0;
}
```

此程序输出：

```
       area = 215.25
       perimeter = 62.00
```

其中的 RrectangleNew()函数就可称为 struct Rectangle 对象的构造器了。

17.5 运用函数指针

在目前的结构里，只有属性（Attribute）数据而已，如果想要让结构更完美地对应到类概念，可以把函数指针放入结构里，并让其指向特定函数就行了。例如，可以在 struct Rectangle 结构里加入 3 个函数指针，分别指向 init()、cal_area()和 cal_perimeter() 3 个函数。这样结构就不仅仅有属性，而且也有行为能力，如此就更可将它视为对象了。其代码如下：

```
/*   cx17-ap4.c   */
#include "stdio.h"
#include "malloc.h"

struct Rectangle
{
  double length;
  double width;
  void (*init)(struct Rectangle*, double, double);
  double (*cal_area)(struct Rectangle*);
  double (*cal_perimeter)(struct Rectangle*);
};

void init(struct Rectangle* cthis, double l, double w)
{ cthis->length = l;
  cthis->width = w;
}

double cal_area(struct Rectangle* cthis)
{
  return cthis->length * cthis->width;
}

double cal_perimeter(struct Rectangle* cthis)
{
  return (cthis->length + cthis->width) * 2;
}

struct Rectangle* RectangleNew()
{
  struct Rectangle* t;
  t = (struct Rectangle*)malloc(sizeof(struct Rectangle));
  t->init = init;
  t->cal_area = cal_area;
  t->cal_perimeter = cal_perimeter;
```

```
  return t;
}
/*----------------------------------------*/
int main()
{ double v;
  struct Rectangle* pr;
  pr = RectangleNew();
  pr->init(pr, 10.5, 20.5);

  v = pr->cal_area(pr);
  printf("area=%7.2f\n", v);

  v = pr->cal_perimeter(pr);
  printf("perimeter=%7.2f\n", v);

  getchar();
  return 0;
}
```

此结构的定义：

```
struct Rectangle
{
  ……
  void (*init)(struct Rectangle*, double, double);
  ……
}
```

其中 init 是一个函数指针，用来指向一个函数，而且此函数必须有三个参数，并不返回任何值。由于一般函数的名称就是函数的地址，所以只要将其赋值（assign）给结构里的函数指针就可以让结构具有行为能力了。例如，在构造器里：

```
struct Rectangle* RectangleNew()
{
  ……
  t->init = init;
  t->cal_area = cal_area;
  t->cal_perimeter = cal_perimeter;
  ……
}
```

这就让此结构具有三个函数了，这样就和面向对象的类概念一致了。

17.6 运用 C 宏

前面已经活用结构来实现面向对象的类概念了。但是代码中一直出现 struct 字眼，如果能借宏将它换成 CLASS 将会更传神。于是宏就派上用场了。

17.6.1 定义宏：CLASS(类名称)

由于在目前的结构里，已经有数据属性（成员）和成员函数了，这样的结构定义可以视为类，于是定义出 CLASS 宏，其代码如下：

```
/*   cx17-ap5.c   */
#include "stdio.h"
#include "malloc.h"

#define CLASS(type)\
typedef struct type type; \
struct type

/* ------------------------------------------------ */
CLASS(Rectangle)
{
  void (*init)(struct Rectangle*, double, double);
  double (*cal_area)(struct Rectangle*);
  double (*cal_perimeter)(struct Rectangle*);
  double length;
  double width;
};

void init_imp(Rectangle* cthis, double len, double wid)
{
  cthis->length = len;
  cthis->width = wid;
}

double cal_area_imp(Rectangle* cthis)
{
  return cthis->length * cthis->width;
}

double cal_perimeter_imp(Rectangle* cthis)
{
  return (cthis->length + cthis->width) * 2;
}

struct Rectangle* RectangleNew()
{
  Rectangle* t;
  t = (Rectangle*)malloc(sizeof(Rectangle));
  t->init = init_imp;
  t->cal_area = cal_area_imp;
  t->cal_perimeter = cal_perimeter_imp;
  return t;
}
/*-----------------------------------------*/
int main()
{
  Rectangle* pr;   double v;
  pr = RectangleNew();
  pr->init(pr, 10.5, 20.5);

  v = pr->cal_area(pr);
  printf("area=%7.2f\n", v);
```

```
  v = pr->cal_perimeter(pr);
  printf("perimeter=%7.2f\n", v);

  getchar();
  return 0;
}
```

17.6.2 定义宏：CTOR（类名称）

刚才已经运用宏将 struct 字眼包装起来了。同理，也可以将构造器 RectangleNew()定义包装起来。也就是再度发挥 C 宏的威力，定义出构造器宏 CTOR（类），其代码如下：

```
/*  cx17-ap6.c   */
#include "stdio.h"
#include "malloc.h"

#define CLASS(type)\
typedef struct type type; \
struct type

#define CTOR(type) \
void* type##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));

#define FUNCTION_SETTING(f1, f2)  t->f1 = f2;
#define END_CTOR return (void*)t;  };

/*                          */
CLASS(Rectangle)
{
  void (*init)(struct Rectangle*, double, double);
  double (*cal_area)(struct Rectangle*);
  double (*cal_perimeter)(struct Rectangle*);
  double length;
  double width;
};

double cal_area_imp(Rectangle* cthis)
{
  return cthis->length * cthis->width;
}

double cal_perimeter_imp(Rectangle* cthis)
{
  return (cthis->length + cthis->width) * 2;
}

void init_imp(Rectangle* cthis, double len, double wid)
{
  cthis->length = len;
  cthis->width = wid;
}

CTOR(Rectangle)
```

```
  FUNCTION_SETTING(init, init_imp)
  FUNCTION_SETTING(cal_area,cal_area_imp)
  FUNCTION_SETTING(cal_perimeter,cal_perimeter_imp)
END_CTOR
/*----------------------------------------*/
int main()
{
  Rectangle* pr;   double v;
  pr = (Rectangle*)RectangleNew();
  pr->init(pr, 10.5, 20.5);

  v = pr->cal_area(pr);
  printf("area=%7.3f\n", v);

  v = pr->cal_perimeter(pr);
  printf("perimeter=%7.3f\n", v);

  getchar();
  return 0;
}
```

构造器 RectangleNew()的定义被隐藏于 CTOR(Rectangle)宏里，这让代码显得更简洁清晰。

17.7 定义接口（Interface）宏

自从 1996 年 Java 采取接口概念以来，现代的计算机语言几乎都提供接口机制了。在本书所提供的 OOPC 语言也不例外，必须提供接口机制。接口与类定义的差异在于：接口只含有成员函数，而无数据成员。所以仍可采用结构来实现接口机制。于是，可以从上述的结构中将成员函数抽离出来，成为一个独立的结构。其代码如下：

```
/*   cx17-ap7.c   */
#include "stdio.h"
#include "malloc.h"

#define CLASS(type)\
typedef struct type type; \
struct type

#define CTOR(type) \
void* type##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));

#define FUNCTION_SETTING(f1, f2)  t->f1 = f2;
#define END_CTOR return (void*)t;  };

/* ------------------------------------ */
struct IA
{
  void (*init)(void*, double, double);
  double (*cal_area)(void*);
```

```
  double (*cal_perimeter)(void*);
};
/* ------------------------------------ */

CLASS(Rectangle)
{
  struct IA IA;
  double length;
  double width;
};

double cal_area_imp(void* t)
{
  Rectangle* cthis = (Rectangle*)t;
  return cthis->length * cthis->width;
}

double cal_perimeter_imp(void* t)
{
  Rectangle* cthis = (Rectangle*)t;
  return (cthis->length + cthis->width) * 2;
}

void init_imp(void* t, double len, double wid)
{
  Rectangle* cthis = (Rectangle*)t;
  cthis->length = len;
  cthis->width = wid;
}

CTOR(Rectangle)
  FUNCTION_SETTING(IA.init, init_imp)
  FUNCTION_SETTING(IA.cal_area,cal_area_imp)
  FUNCTION_SETTING(IA.cal_perimeter,cal_perimeter_imp)
END_CTOR
/*----------------------------------------*/
int main()
{
  double v;
  struct IA* pr;
  pr = (struct IA*)RectangleNew();
  pr->init(pr, 10.5, 20.5);

  v = pr->cal_area(pr);
  printf("area=%7.2f\n", v);

  v = pr->cal_perimeter(pr);
  printf("perimeter=%7.2f\n", v);

  getchar();
  return 0;
}
```

这个程序使用 struct IA 来声明接口。可再运用 C 宏将它包装得更简洁一些，于是定义出 INTERFACE(IA)和 IMPLEMENTS(IA)两个宏，其代码如下：

```
/*  cx17-ap8.c   */
#include "stdio.h"
```

```
#include "malloc.h"

#define CLASS(type)\
typedef struct type type; \
struct type

#define CTOR(type) \
void* type##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));

#define FUNCTION_SETTING(f1, f2)  t->f1 = f2;
#define END_CTOR return (void*)t;  };
#define INTERFACE(type) \
typedef struct type type; \
struct type
#define IMPLEMENTS(type) \
struct type type

/* ------------------------------------ */
INTERFACE(IA)
{
  void (*init)(void*, double, double);
  double (*cal_area)(void*);
  double (*cal_perimeter)(void*);
};
/* ------------------------------------ */
CLASS(Rectangle)
{
  IMPLEMENTS(IA);
  double length;
  double width;
};

void init_imp(void* t, double len, double wid)
{
  Rectangle* cthis = (Rectangle*)t;
  cthis->length = len;
  cthis->width = wid;
}

double cal_area_imp(void* t)
{
  Rectangle* cthis = (Rectangle*)t;
  return cthis->length * cthis->width;
}

double cal_perimeter_imp(void* t)
{
  Rectangle* cthis = (Rectangle*)t;
  return (cthis->length + cthis->width) * 2;
}

CTOR(Rectangle)
  FUNCTION_SETTING(IA.init, init_imp)
  FUNCTION_SETTING(IA.cal_area,cal_area_imp)
  FUNCTION_SETTING(IA.cal_perimeter,cal_perimeter_imp)
END_CTOR
```

```
/*----------------------------------------*/
int main()
{
  IA* pr;  double v;
  pr = (IA*)RectangleNew();
  pr->init(pr, 10.5, 20.5);

  v = pr->cal_area(pr);
  printf("area=%7.2f\n", v);

  v = pr->cal_perimeter(pr);
  printf("perimeter=%7.2f\n", v);

  getchar();
  return 0;
}
```

17.8 定义 CTOR2()构造器宏

这是第二构造器宏，它的目的是要创造 Rectangle 类变动的空间和自由度。请你想象一下，如果 AA 团队负责 Rectangle 类的开发，而 main()函数由 BB 团队负责开发。如果有一天，AA 团队想把 Rectangle 字眼换成 NewRect 字眼时，显然 main()内的：

```
pr = (IA*)RectangleNew();
```

这个指令的 RectangleNew()也必须换成 NewRectNew()。但使用 CTOR2()就不必更改 NewRectNew()了。请看 CTOR2()的例子吧!

```
/*  lw_oopc.h   */
#include "stdio.h"
#include "malloc.h"

#define CLASS(type)\
typedef struct type type; \
struct type

#define CTOR(type) \
void* type##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));

#define CTOR2(type, type2) \
void* type2##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));

#define END_CTOR return (void*)t;  };
#define FUNCTION_SETTING(f1, f2)  t->f1 = f2;
#define IMPLEMENTS(type) struct type type
#define INTERFACE(type) \
typedef struct type type; \
struct type
/*--------------------------------------*/
```

```
INTERFACE(IA)
{
  void (*init)(void*, double, double);
  double (*cal_area)(void*);
  double (*cal_perimeter)(void*);
};
/* ------------------------------------ */
CLASS(NewRect)
{
  IMPLEMENTS(IA);
  double length;
  double width;
};

void init_imp(void* t, double len, double wid)
{
  NewRect* cthis = (NewRect*)t;
  cthis->length = len;
  cthis->width = wid;
}

double cal_area_imp(void* t)
{
  NewRect* cthis = (NewRect*)t;
  return cthis->length * cthis->width;
}

double cal_perimeter_imp(void* t)
{
  NewRect* cthis = (NewRect*)t;
  return (cthis->length + cthis->width) * 2;
}

CTOR2(NewRect, Rectangle)
  FUNCTION_SETTING(IA.init, init_imp)
  FUNCTION_SETTING(IA.cal_area,cal_area_imp)
  FUNCTION_SETTING(IA.cal_perimeter,cal_perimeter_imp)
END_CTOR
/*----------------------------------------*/
int main()
{
  IA* pr;  double v;
  pr = (IA*)RectangleNew();
  pr->init(pr, 10.5, 20.5);

  v = pr->cal_area(pr);
  printf("area=%7.2f\n", v);

  v = pr->cal_perimeter(pr);
  printf("perimeter=%7.2f\n", v);

  getchar();
  return 0;
}
```

宏 CTOR2（NewRect, Rectangle）说明了，它将转化出 RectangleNew()构造器，但是类名称改为 NewRect。如此，整个 main()的代码都不必修正。其中获得最大利益者就是负责 Rectangle 类开发的 BB 团队了。

17.9 将宏独立成 lw_oopc.h 头文件

最后，将刚才所设计的宏独立出来成为头文件，取名为 lw_oopc.h，这样任何.c 程序就能随时使用#include " lw_oopc.h"来添加它。此 lw_oopc.h 的内容为：

```
/*  lw_oopc.h */
/* 这就 MISOO 团队所设计的 C 宏*/
#include "malloc.h"
#ifndef LOOPC_H
#define LOOPC_H

#define CLASS(type)\
typedef struct type type; \
struct type

#define CTOR(type) \
void* type##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));

#define CTOR2(type, type2) \
void* type2##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));

#define END_CTOR return (void*)t;  };
#define FUNCTION_SETTING(f1, f2)  t->f1 = f2;
#define IMPLEMENTS(type) struct type type
#define INTERFACE(type) \
typedef struct type type; \
struct type
#endif
/*    end    */
```

接着定义 IA 接口

```
/*  cx17-ia.h   */
INTERFACE(IA)
{
  void (*init)(void*, double, double);
  double (*cal_area)(void*);
  double (*cal_perimeter)(void*);
};
```

编写 NewRect 类的代码

```
/*  cx17-rec.c   */
#include "stdio.h"
#include "lw_oopc.h"
#include "cx17-ia.h"

CLASS(NewRect)
{
  IMPLEMENTS(IA);
```

```
  double length;
  double width;
};

void init_imp(void* t, double len, double wid)
{
  NewRect* cthis = (NewRect*)t;
  cthis->length = len;
  cthis->width = wid;
}

double cal_area_imp(void* t)
{
  NewRect* cthis = (NewRect*)t;
  return cthis->length * cthis->width;
}

double cal_perimeter_imp(void* t)
{
  NewRect* cthis = (NewRect*)t;
  return (cthis->length + cthis->width) * 2;
}

CTOR2(NewRect, Rectangle)
  FUNCTION_SETTING(IA.init, init_imp)
  FUNCTION_SETTING(IA.cal_area,cal_area_imp)
  FUNCTION_SETTING(IA.cal_perimeter,cal_perimeter_imp)
END_CTOR
```

编写 main()主函数

```
/*  cx17-app.c   */
#include "stdio.h"
#include "lw_oopc.h"
#include "cx17-ia.h"
/*----------------------------------------*/
int main()
{
  IA* pr;  double v;
  pr = (IA*)RectangleNew();
  pr->init(pr, 10.0, 20.5);

  v = pr->cal_area(pr);
  printf("area=%7.2f\n", v);

  v = pr->cal_perimeter(pr);
  printf("perimeter=%7.2f\n", v);

  getchar();
  return 0;
}
```

最后，编写 TurboC 的 cx17-app.prj 文件，其内容为：

```
cx17-rec.c
cx17-app.c
```

输出结果为：

```
area = 205.00
perimeter= 61.00
```

以上就是 lw_oopc.h 的设计思维，希望你能深刻体会，必要时也可以修改它，而创造更新奇的 lw_oopc.h 来。但别忘了与笔者分享啊！

第 18 章　认识 UML

18.1 UML：世界标准对象模型语言

模型语言（Modeling Language 简称 ML）是一种设计语言（Design Language），人们藉由设计语言来创造设计品。模型语言是人们用来设计系统模型（Model）的语言，其设计品是系统的模型（或称为模型），也就是另一种设计品（即真实系统）的蓝图（Blueprint）。

在建筑界方面，建筑师 Christopher Alexander 在 20 世纪 70 年代就已提出模式语言（Pattern Language）的概念，模式语言含有建筑设计师与居住人的共同表达方式，并支持特殊的创意。因此，建筑师与居住者之间能互相了解与沟通，也让居住者能了解建筑师的创意。音乐家设计乐谱，它也是一种模型，如图 18-1 所示。

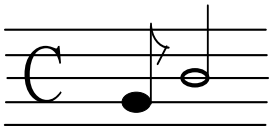


图 18-1

现在就来看看软件系统方面的模型语言——UML 吧！软件设计师藉由 UML 来创造软件系统的模型（蓝图），在创造的过程中设计师将他的心意、情感与经验融入模型中；程序员藉由 UML 来跟模型沟通与神往，体会与理解模型所表达出来的情感与经验，然后用 OOPC 程序语言表达成为真实的计算机软件。模型语言能促进设计师与用户之间的共识（Shared Understanding），由此观之，UML 模型语言就显得极有意义。

UML 模型语言是面向对象的软件模型语言，根据面向对象概念，这种模型语言比传统的更接近人类的日常思考方式，用户将更易于接受它。即用户将更易于借由 UML 模型语言来与软件沟通神往，也更易于表达其需求。UML 是普遍性的系统模型语言，除了软件之外，还适用于汽车、计算机、导弹等各领域的系统模型。UML 是软、硬整合设计的有力工具。

18.2 UML 的演化

1995 年 10 月，Grady Booch 和 James Rumbaugh 为了合并两大著名的对象方法——Booch’93 和 OMT-2 一同工作，合并后的新方法定为“整合方法”（Unified Method），即 UML 的前身。后来，在 1995 年秋天，Ivar Jacobson 也加入研究行列，动手合并另一个著名的对象方法——OOSE。在这之后的一年，他们才公布 UML，正式定位其为模型语言，并且舍弃了先前的整合方法。这令人精神一振，因为 UML 不再是一个老掉牙的对象方法了，它是一个新发明，至此 UML 真正诞生了！

请看图 18-2 的 UML 发展简图，图中展示了 3 位大师主导的 Booch’93、OMT-2 和 OOSE 是 UML 初期发展的核心。在 1996 年之后，才有更多的专家学者贡献己力，注入更广泛的见解。所以，UML 与其说是 3 位大师的发明，不如说是一件众人群力的智慧结晶。因而，在

1997 年 11 月时，OMG（Object Management Group）正式认定 UML 1.1 为其协会的标准，而后协会将竭尽心力负责改善 UML。如今，在协会的强力推广之下，全球的知识工作者用共同语言沟通的梦想，已然实现。

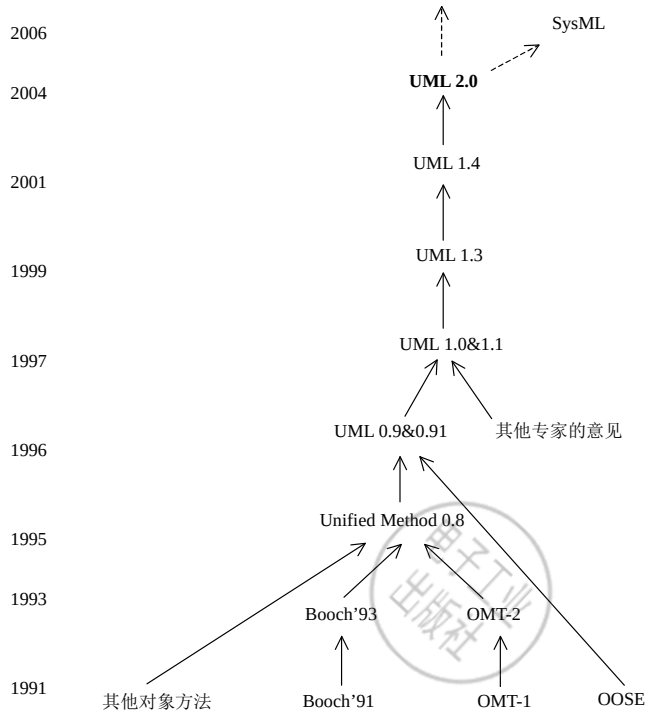


图 18-2

OMG 是一个国际性组织，目前约有 800 个会员，包括信息系统供货商、软件开发者和用户。协会成立于 1989 年，旨在推广对象技术于软件开发。为此，它制定了许多标准，UML 就是其标准之一，期望打造一个共通的应用架构，让基于此的软件在跨硬件平台和系统软件的异质环境中，保有可复用、可携带、可互相操作的特性。

在 OMG 决定采用 UML 作为其标准之一后，将负责后续的改版事宜。因此，从 1997 年的 1.1 版起，陆续推出 1999～2000 年 3 月的 1.3 版、2001 年 9 月的 1.4 版，最新的 2.0 版 beta 版已经制定完成。

此外，3 位大师放弃了 UML 的所有权，让它真正成为公众的智慧财富。如今，所有人都可以通过 OMG 的网站——*http://www.omg.org/uml/*了解 UML 的最新动态，以及提供己见，也可以登上它的专用网站 *http://www.uml.org/*。当然，若是通过 Yahoo!搜索，您将会发现 UML 是多么地 hot！UML 的注册商标如图 18-3 所示。

图 18-3

18.3 UML 的基本元素

使用中文时，我们利用有限的单字（Word）或词汇（Term），依语法规则做无限的组合，而创造出无限的句子。在 Alexander 的模式语言中，也是利用有限的“模式”（Pattern）来做各式各样的组合而成为多样化的建筑物的。其中，单字是中文的基本元素（Basic Element），而模式是模式语言中的基本元素。而且在中文中，我们也常随风俗习惯而将基本元素（字）组成较高阶层的元素——短语（Phrase）。像字和短语等皆是我们在创造句子时惯用的基本单位，通称为语言的元素（Element）。

当然 UML 也有基本元素，如类（Class）、对象、操作（Operation）、泛化（Generalization）及结合（Association）关系等。利用这些有限的元素，可组合成为各式各样的软件模型。在软件设计师的心中，也常有较高阶层的元素，如使用案例（Use Case）、软件设计模式（Design Pattern）及群组包裹（Package），等等。

UML 将软件模型（Software Model）中的元素看成像自然语言中的“词汇”一样，并将设计原则及经验法则视为语言中的“语法规则”一般，嵌含在词汇的含义之中。UML 会像自然语言一样，由许多小的词汇和简单的规则，逐渐从人们的经验中吸收而扩充其词汇和规则。同时词汇和规则也会汰旧换新呈现出现代感，表达人们新的思想和文化。于是，UML 成为放诸四海的共同语言了，只是它是活的，各领域中的人们可弹性地增加其领域专有的词汇、含义和规则，而成为领域中特殊文化下的特殊语言。软件人员使用这种模型语言来构思及创造软件，就如同我们利用中文来构思及说出一段话或写一篇文章一样，充分地再利用别人的创意、智慧，还可以表达自己的见解，这正合乎软硬件整合设计潮流的方向和目标。

18.4 UML 的图示

UML 包含一套符号，单一符号通常用处不大，但是几种符号组织成“图”（diagram）之后，用处就大了。原因很简单，因为“图”能用以表现某一角度下的细致构想。这好比只有一种音符是很单调的，有了几种音符才能表现贝多芬的命运交响曲，您赞同吧！

目前，UML2.0 共有 10 个图，分别为：组合结构图、用例图、类图、序列图、对象图、合作图、状态图、活动图、模块图和部署图，它们各用以表现不同的观点，如表 18-1 所示。

表 18-1

名称	观点	主要符号
1. 复合结构图（composite-structure diagram）	表现结构（架构）性需求，主要包括 Part、Port、接口和连接（Link）	Part、Port、接口、连接关系
2. 用例图（use case diagram）	表现功能性需求，主要包括用例和参与者	用例、参与者、结合关系
3. 类图（class diagram）	表现静态结构，主要包括一群类及其间的静态关系	类、结合关系、一般化关系
4. 序列图（sequence diagram）	表现一群对象依序传送信息的交互状况	对象、信息、活动期
5. 对象图（object diagram）	表现某时刻下的数据结构，主要包括一群对象及其间拥有的数据数值	对象、连接、信息
6. 合作图（collaboration diagram）	表现一群有连接的对象传送信息的交互状况	对象、连接
7. 状态图（statechart diagram）	表现某种对象的行为，主要呈现一堆状态因事件而转换的状况	状态、事件、转换、行动
8. 活动图（activity diagram）	表现一段自动转换的活动流程，主要包括一堆活动及其间的自动转换线	活动、转换、分叉、接合
9. 执行模块图（component diagram）	表现一群可执行模块及它们之间的依赖关系	模块、界面、相依关系、实现关系
10. 部署图（deployment diagram）	表现一堆设备及它们之间的依赖关系	节点、模块、相依关系

第 19 章　UML 类图

19.1 为什么需要面向对象思维

所谓面向对象（Object-Oriented）思维就是，人们心中有对象（Object），并且认为软件系统就是由一群有智慧、善传递信息的对象所组成的。目前的 C 程序员中还有许多人采取传统的面向功能（Function-Based）的概念，把软件系统视为是许多函数（Function）经由互相调用所组成的。无论是面向对象还是面向功能（又称为结构化）思维，本身都没有对或错之分，它们只是观点（View）的不同而已。

在本书前面各章里，已经说明了，当今市场急速扩大的嵌入式（Embedded）软件领域，亟需提升软件系统的系统分析及架构设计的技术能力，例如世界知名的麦肯锡（McKinsey）顾问公司，在 2006 年的报告（“Getting better software to manufactured products”）中呼吁嵌入式软件业必须积极提升其系统分析及架构设计的技术能力，才能解决软件含量愈来愈多的数码产品的信赖性问题。在面对当今的这个挑战时，我们必须重新选择一项比较有利的观点。

在计算机语言方面，到了 2000 年以后，几乎都采取面向对象概念了，例如 Java、C#、VB.net 等。此外，更重要的是，当今美国 OMG 所认定的标准系统分析与建模语言 UML 也采取面向对象概念。不论面向对象与面向功能两者之中哪一个表现较好还是较差，显然面向对象是当今的主流思维。如果 C 程序员维持传统的面向功能思维，则采用 UML 的系统分析员、架构设计师的思维与 C 程序员之间，便有了许多沟通上的困难，甚至误解对方的想法或需求。

反之，在面向对象思维下，分析员、设计师及程序员，大家皆使用一致的理念。例如，用户心中的对象，是分析员心中的对象，亦是程序员心中的对象，就容易达到心心相印的境界了。人人习惯于赋予对象人性，是面向对象思维的关键步骤。社会是由一群有智慧、能沟通且互助合作的个人所构成的，软件则由一群有智慧、善传递信息的对象所组成。我们脑海里所想象的对象，就如社会中的个人一般理性、善沟通且愿互助合作，此即所谓的对象“拟人化”（Anthropomorphize），或者俗称为“人格化”。例如，厨房里的冰箱，为聪明绝顶的对象，当您打开它的门时，它会打开小灯，让您看得见冰箱里的菜；当关起门时，会关闭小灯，以节省电费。您可尽情地去将它想象成神童，每天都会提醒您应买什么菜！再如，火车站里的月台知道即将驶入的火车种类、编号，并且立即告诉等候的旅客，月台不只了解它自己，也了解相关的对象——火车，使之为旅客提供完美服务。如果您看过《霹雳游侠》电视剧，便易于体会什么是有智慧的汽车了。拟人化之后的对象，就像霹雳车 KITT 一般聪明。

虽然冰箱、月台等并非人类，但赋予其人性之后，就会变成主动、积极且活生生的东西。在软件系统中，这些对象扮演着主动提供服务的角色（Role），担负着应有的责任（Responsibility），以支持整个软件系统的功能。这个角色和责任就是对象的行为（Behavior），了解和掌握对象行为是系统分析和架构设计的重要工作，诚如布朗大学的 M.Sharenberg 教授所说：对象高度拟人化是软件设计至高无上的方针。

当我们心中有对象（Thinking with objects），且将“苹果”拟人化，则苹果不再是消极被吃的东西，而是积极、主动、有智能且可沟通的对象了。它能担负适当的角色或责任。例如，回答下述问题：

- 我的价格为何？
- 我的重量为何？
- 我的颜色为何？
- 我是可吃的东西吗？
- 什么动物喜欢吃我？
- 我最喜欢生长于哪个国家？

对象行为常需其他对象行为的协助；例如，当苹果想回答问题——“什么动物喜欢吃我？”时，它会去询问有关的动物，取得动物数据，才能给您满意的答复。此时“动物”对象协助“苹果”对象来达成其任务。

19.2 为什么需要设计类

俗语说“物以类聚”，物就是对象，类就是类。在面向对象思维下，人们把软件视为一群对象的互相沟通合作（Collaboration），并且将属性及行为相似的对象归成同一类。因为同一类的对象其属性和行为相近，人们只要针对类去加以描述，就等于描述了所有的对象了，这让人们更容易去描述各对象的属性和行为。一旦软件人员描述好了类，计算机在执行期间，就把类视为模子（Template），依据模子而生成软件对象，并且安排对象之间的互相沟通与合作，那么，软件系统就能提供给用户各式各样的服务了。

简而言之，人们心存对象思维，将对象归类，然后在 C 程序里定义类以描述对象的属性和行为，到了程序执行时，计算机拿类模子去生成出软件对象来对应人们心中的对象。

人们的“对象概念”（Object concept）是有天赋的，从幼儿期就表现了认识对象的能力，即能区别含不同特征（Property）的东西。例如：您买个黄色皮球给 5 岁的阿皮，在他玩一阵子之后，您偷偷将皮球藏起来，则阿皮会去寻找那颗皮球，逐一比对眼前的东西，看看这些东西的特征是否跟脑海里的印象相同，一直至找到或疲倦了为止。

“特征”又称为“属性”（Attribute）。人类常根据对象的属性来区别不同的对象。例如：借耳朵长度、尾巴长度及眼睛颜色等特性来判断一只动物是否为兔子，同时也借这些属性值（Value）来区别各只兔子。每只兔子的耳朵长度、尾巴长度及眼睛颜色等皆不尽相同，即各只兔子各有其特征值。

就像各只兔子有其专有的特征值一样，各对象皆有其专有的特征值。特征值就是对象的

特有状态。通常，同一类（Class）的对象具有相同属性也具有相同的行为（Behavior），但各具有其不同的特征值，亦即各有自己的状态。如图 19-1 所示。

书号	书名	作者
0-201-54709-0	OOPC	Alvin Kao
0-800-23456-2	Java	Addison King
0-8306-3308-1	Google Android	Stone Bush

图 19-1

为了产生对象来储存书本的属性值，宜设计类（即模子）如下：

```
CLASS(Book)
{
 char book_no[14];
 char title[20];
 char author[15];
};
```

于是可拿 Book 来声明对象，每个对象皆含有 book_no、title 及 author3 个变量，以储存对象的状态。例如：

```
Book  book_obj1, book_obj2, book_obj3;
```

或

```
Book *book1, *book2, *book3;
```

由于各对象皆有自己的状态，所以 book1、book2 与 book3 都指向各自的空间（内存空间），以便储存其状态。通常，对象将其状态封装起来，避免外界的影响。所以，对象必须负责维护本身的状态，即必须提供函数来维护对象的状态。例如：

```
/* cx19-ap1.c  */
#include "stdio.h"
#include "lw_oopc.h"

CLASS(Book)
{
  void (*set)(void*, char*, char*, char*);
  char book_no[14];
  char title[20];
  char author[15];
};

static void set(void* t, char no[], char ti[], char a[] )
    {
      Book* cthis = (Book*)t;   strcpy(cthis->book_no, no);
      strcpy(cthis->title, ti);    strcpy(cthis->author, a);
    }
CTOR(Book)
  FUNCTION_SETTING(set, set)
END_CTOR
```

```
int main()
{
  Book *book = (Book*)BookNew();
  book->set(book, "0-201-54709-0", "OOPC", "Alvin Kao");
  printf("course title = %s\n", book->title);
  getchar();
  return 0;
}
```

set() 函数让 Book 的对象具有设定及更改其状态的行为。因此，用户可借 set() 函数改变对象的状态。例如指令：

```
Book *book = (Book*)BookNew();
book->set(book, "0-201-54709-0", "OOPC", "Alvin Kao");
```

此 BookNew()就拿类模子来生成 Book 的对象，而 set()指令设定该对象的状态值。有时候，对象也负责将状态传递出来，供用户来应用。例如，为 Book 类中加上 get_book_bo()函数等，其能取出对象的属性值，并且传递出来，让别的对象加以应用。

19.3 为什么要描述类间的关系

在面向对象思维下，人们把软件视为一群对象的互相沟通合作。在行为方面，对象会互相传递信息，例如银行“用户”对象会送信息“账户”，问他的银行存款余额。在结构方面，对象间会有其组合结构，例如一个用户能拥有多个账户，这意味着：一个用户对象可能会连接（Link）到多个账户对象。再如一个“车体”对象可能会连接到多个“轮胎”对象。

在开发软件时，不仅需要描述对象的属性（Data）和行为（Behavior），而且必须描述对象之间的沟通及其结构面的连接关系。在上一节里已经说明了，在 OOPC 程序里，必须定义类来叙述对象，所以在类定义里，不仅需要定义类的内部属性及函数，还得定义类间的连接关系。这种连接关系包括组合（Composition）关系及结合（Association）关系等。

19.3.1 类间的组合关系

上一节已经设计出 Book 类了。其中内含一个 author 字符串代表作者的姓名。当我们想要存入更多关于作者的数据时，通常会将作者视为对象，于是就有两个对象了，而且它们会互相传递信息。

```
/* cx19-ap2.c  */
#include "stdio.h"
#include "lw_oopc.h"

CLASS(Author)
   {
     void (*init)(void*, char*, char*);
     char* (*get_name)(void*);
     char* (*get_tel_no)(void*);
     char name[14];
```

```
      char telphone[20];
    };

static void init(void* t, char na[], char tel[] )
     {
       Author* cthis = (Author*)t;
       strcpy(cthis->name, na);
       strcpy(cthis->telphone, tel);
     }
static char* get_name(void* t)
     {
       Author* cthis = (Author*)t;
       return cthis->name;
     }
static char* get_tel_no(void* t)
     {
       Author* cthis = (Author*)t;
       return cthis->telphone;
     }
CTOR(Author)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(get_name, get_name)
  FUNCTION_SETTING(get_tel_no, get_tel_no)
END_CTOR

/*  ---------------------------  */
CLASS(Book)
   {
     void (*set)(void*, char*, char*, char*, char*);
     void (*print)(void*);
     char book_no[14];
     char title[20];
     Author* author;
   };

static void set(void* t, char no[], char ti[], char a[], char tel[] )
     {
       Book* cthis = (Book*)t;
       strcpy(cthis->book_no, no);
       strcpy(cthis->title, ti);
       cthis->author = (Author*)AuthorNew();
       cthis->author->init(cthis->author, a, tel);
     }
static void print(void* t)
{
  Book* cthis = (Book*)t;
  Author* pau;
  printf("book_no: %s,  title: %s\n",
  cthis->book_no, cthis->title);
  pau = cthis->author;
  printf("author_na: %s, author_tel: %s\n",
      pau->get_name(pau), pau->get_tel_no(pau));
}

CTOR(Book)
  FUNCTION_SETTING(set, set)
  FUNCTION_SETTING(print, print)
END_CTOR
```

```
int main()
{
  Book *book = (Book*)BookNew();
  book->set(book, "0-201-54709-0", "OOPC", "Alvin Kao", "2622-7777");
  book->print(book);
  getchar();
  return 0;
}
```

Book 对象内的 author 属性值为 Author 的对象。所以 Book 的对象内含一个 Author 的对象。如图 19-2 所示。

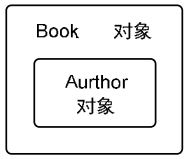


图 19-2

由于 author 是 Author 类的对象，它封装 name 及 telephone 两个特征值，使得 Book 的对象无法直接存取这些特性。不过，Author 类提供了 3 个函数——set()、get_name()及 get_tel_no()，让对象具有储存本身状态的行为。因此，Book 类的对象可委托 author 对象维护作者数据。例如，指令 author->set(na, tel)委托 author 保存 na 及 tel 两项数据。而指令 author->get_name()及 author->get_tel_no()则委托 author 取出 name 及 telphone 数据。

BookNew()函数生成 Book 类的对象，并由 book 指针指向它，也生成内含的 author 对象。因 book 对象内含另一对象，则称它为“复合对象”（Composite object）；而 author 对象被包含于别的对象内，称它为“组件对象”（Component object）。当 book 对象被删除时，其内含 author 对象也被删除了；所以 book 与 author 两对象共生共灭。即复合对象与其内含的组件对象的寿命一样长。这就是对象间的组合关系，创建此种关系的常见做法就是：在定义类时，将属性值的类型设定为类就行了。也就说，在 OOPC 程序里，就依赖这种机制来描绘与指示如何创建对象间的组合关系。

19.3.2 类间的结合关系

在上一节的组合关系里，author 对象被包含于 book 对象里，一旦 book 对象被删除了，则 author 对象也就被删除了。这样会呈现一些麻烦：如果一位作者写了许多本书时，作者数据就会重复出现于不同的 book 对象内。此时，最好改为结合关系。例如：

```
/* cx19-ap3.c  */
#include "stdio.h"
#include "lw_oopc.h"

CLASS(Author)
   {
     void (*init)(void*, char*, char*);
```

```
      char* (*get_name)(void*);
      char* (*get_tel_no)(void*);
      char name[14];
      char telphone[20];
    };

static void init(void* t, char na[], char tel[] )
      {
        Author* cthis = (Author*)t;
        strcpy(cthis->name, na);
        strcpy(cthis->telphone, tel);
      }
static char* get_name(void* t)
      {
        Author* cthis = (Author*)t;
        return cthis->name;
      }
static char* get_tel_no(void* t)
      {
        Author* cthis = (Author*)t;
        return cthis->telphone;
      }
CTOR(Author)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(get_name, get_name)
  FUNCTION_SETTING(get_tel_no, get_tel_no)
END_CTOR

/*  ---------------------------  */
CLASS(Book)
    {
      void (*set)(void*, char*, char*);
      void (*setAuthor)(void*, Author*);
      void (*print)(void*);
      char book_no[14];
      char title[20];
      Author* author;
    };
static void set(void* t, char no[], char ti[])
      {
        Book* cthis = (Book*)t;
        strcpy(cthis->book_no, no);
        strcpy(cthis->title, ti);
      }
static void setAuthor(void* t, Author* pau)
      {
        Book* cthis = (Book*)t;
        cthis->author = pau;
      }
static void print(void* t)
{
  Book* cthis = (Book*)t;
  Author* pau;
  printf("book_no: %s,  title: %s\n",
  cthis->book_no, cthis->title);
  pau = cthis->author;
  printf("author_na: %s, author_tel: %s\n",
      pau->get_name(pau), pau->get_tel_no(pau));
  }
```

```
CTOR(Book)
  FUNCTION_SETTING(set, set)
  FUNCTION_SETTING(setAuthor, setAuthor)
  FUNCTION_SETTING(print, print)
END_CTOR

int main()
{
  Author *mike = (Author*)AuthorNew();
  Book *book1 = (Book*)BookNew();
  Book *book2 = (Book*)BookNew();

  mike->init(mike, "Alvin Kao", "2622-7777");
  book1->set(book1, "0-201-54709-0", "OOPC");
  book2->set(book2, "0-183-156667", "Ajax");

  book1->setAuthor(book1, mike);
  book2->setAuthor(book2, mike);
  book1->print(book1);
  printf("\n");
  book2->print(book2);
  getchar();
  return 0;
}
```

Book 的对象不再负责生成 Book 的对象了，而是由 main()分别生成 Book 和 Author 的对象，并让两者“结合”起来，如图 19-3 所示。

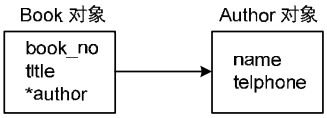


图 19-3

想打印出 Book 对象的内容，可由指针 author 找出其作者对象的内容。

19.4 为什么要绘制 UML 类图

前面说明了，类定义的主要工作是要描述下述各项：

- 对象的属性，这表达于类的数据成员。
- 对象的函数，这表达于类的成员函数。
- 对象与对象间的关系，这主要表达于类间的组合关系和结合关系。

在一个复杂一点的软件系统里，除了程序员使用上述的对象思维之外，系统分析员及架构设计师们也是采取一样的思维。程序员通常很熟悉计算机语言，能轻易地将心中的类和对象表现为 OOPC 的类和对象。然而。系统分析员和架构设计师们，通常并不熟悉 OOPC 等计算机语言，那么又该如何表达其心中的类和对象呢？为了解决此问题，UML 建模语言就应

运而生了。系统分析员及架构设计师们通常采用 UML 表达其心中的类及对象，以图形方式呈现出来，并且传达给程序员，再编写成为 OOPC 程序代码。

请看一个例子：我家的电冰箱里有个小灯，冰箱门打开时，它就亮起来；一旦关上门，它就熄灭了。系统分析及设计师，寻找到“冰箱”和“小灯”两个类。当冰箱门每次被打开时，冰箱就传送 turnOn()信息给小灯。小灯就亮起来了。当冰箱门每次被关起来时，冰箱就传送 turnOff()信息给小灯。小灯就熄灭了。

有人认为冰箱与小灯两个类应该是组合关系，另有些人会认为两者应该是结合关系。这两个观点没有对与错之分，如果认为是结合关系，就使用 UML 表达出来；若认为是组合关系，也可以使用 UML 表达出来。一旦大家愿意使用一致的对象思维，而且愿意使用 UML 表达出来，大家就容易互相了解，而逐渐取得共识了。一般而言，只要能取得共识，就会对复杂软件系统的开发提供极大的帮助，这就是为什么要绘制 UML 类图的基本理由：取得共识。我们以 UML 类图表示为例，如图 19-4 所示。

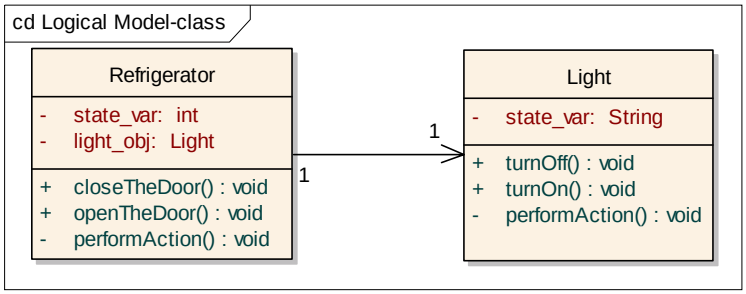


图 19-4

如果程序员也认同图 19-4，就能基于此类图而编写为 OOPC 的类代码了。根据图 19-4，我们必须以 OOPC 定义 Refrigerator 和 Light 两个类，并定义其结合关系，如下述的 OOPC 代码：

```
/* cx19-ap4.c   */
#include "stdio.h"
#include "lw_oopc.h"

CLASS(Light) {
  void (*init)(void*);
  void (*performAction)(void*);
  void (*turnOn)();
  void (*turnOff)();
  int state_var;
};
static void init(void* t)
{
  Light* cthis = (Light*) t;
  cthis->state_var = 0;
}
static void performAction(void* t)
```

```
{
  Light* cthis = (Light*) t;
  if( cthis->state_var == 1)
     printf("Light is ON\n");
  else
     printf("Light is OFF\n");
}
static void turnOn(void* t)
{
  Light* cthis = (Light*) t;
  cthis->state_var = 1;
  cthis->performAction(cthis);
}
static void turnOff(void* t)
{
  Light* cthis = (Light*) t;
  cthis->state_var = 0;
  cthis->performAction(cthis);
}

CTOR(Light)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(performAction, performAction)
  FUNCTION_SETTING(turnOn, turnOn)
  FUNCTION_SETTING(turnOff, turnOff)
END_CTOR
/*  ---------------------------  */
CLASS(Refrigerator)
{
  void (*set)(void*, Light*);
  void (*closeTheDoor)(void*);
  void (*openTheDoor)(void*);
  int state_var;
  Light* light_obj;
};

static void set(void* t, Light* light)
{
  Refrigerator* cthis = (Refrigerator*)t;
  cthis->state_var = 0;
  cthis->light_obj = light;
}
static void closeTheDoor(void* t)
{
  Refrigerator* cthis = (Refrigerator*)t;
  cthis->state_var = 0;
  cthis->light_obj->turnOff(cthis->light_obj);
}
static void openTheDoor(void* t)
{
  Refrigerator* cthis = (Refrigerator*)t;
  cthis->state_var = 1;
  cthis->light_obj->turnOn(cthis->light_obj);
}

CTOR(Refrigerator)
  FUNCTION_SETTING(set, set)
  FUNCTION_SETTING(closeTheDoor, closeTheDoor)
  FUNCTION_SETTING(openTheDoor, openTheDoor)
```

```
END_CTOR

int main()
{
  Light *light = (Light*)LightNew();
  Refrigerator *ref = (Refrigerator*)RefrigeratorNew();
  light->init(light);
  ref->set(ref, light);
  ref->openTheDoor(ref);
  ref->closeTheDoor(ref);
  ref->openTheDoor(ref);
  getchar();
  return 0;
}
```

ref->openTheDoor（ref）指令表示打开门（openTheDoor），冰箱转移到 1 状态冰箱对象产生一项行动——送出 turnOn()信息给小灯，于是小灯转移到 1 状态，灯就亮了。ref->closeTheDoor（ref）指令表示关起门（closeTheDoor），冰箱转移到 0 状态，冰箱产生一项行动——送出 turnOff()信息给小灯，于是小灯移到 0 状态，灯熄了。

19.5 如何绘制 UML 类图

——使用免费的 StarUML 及 JUDE 工具软件

——使用物美价廉的 Enterprise Architect 工具

UML 是世界标准的建模语言，目前有许多相关的工具可用，其中在需要付费工具方面，以 Enterprise Architect（简称为 EA）最流行，也很便宜。在开放源码方面以 StarUML 和 JUDE 名气最大。本节简介它们所绘出的类图，让你有些印象，至于如何熟练其使用方法，并非本书的范围，请你上网阅读它们的使用手册。

19.5.1 EA 的类图

EA 类图的画面为如图 19-5 所示。

左边是 UML 类图的基本图示，也就是俗称的工具箱（ToolBox），绘图时可以直接从工具箱把图示拉出来，摆入屏幕中央的绘图区里。例如，有个 Sensor 对象，它收集各个压力侦测器（Pressure Sensor）传来的压力数据，然后将之转送给已注册（Registered）的多个控制模块（Pressure Controller）。如果设计师心中认为 Sensor 对象与 Register 对象之间是结合关系，而且 Register 对象与 Controller 对象之间是组合关系的话，就可利用 EA 画出类图，如图 19-6 所示。

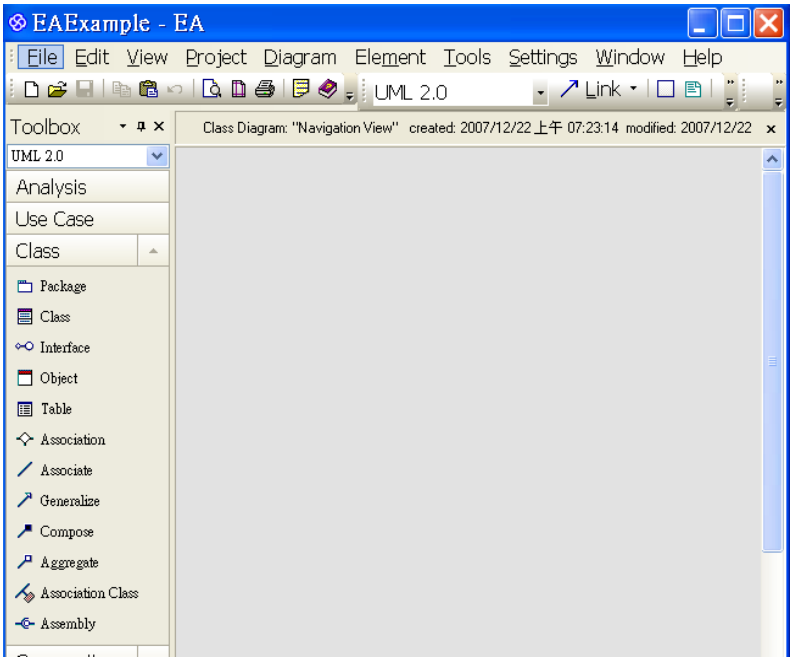


图 19-5

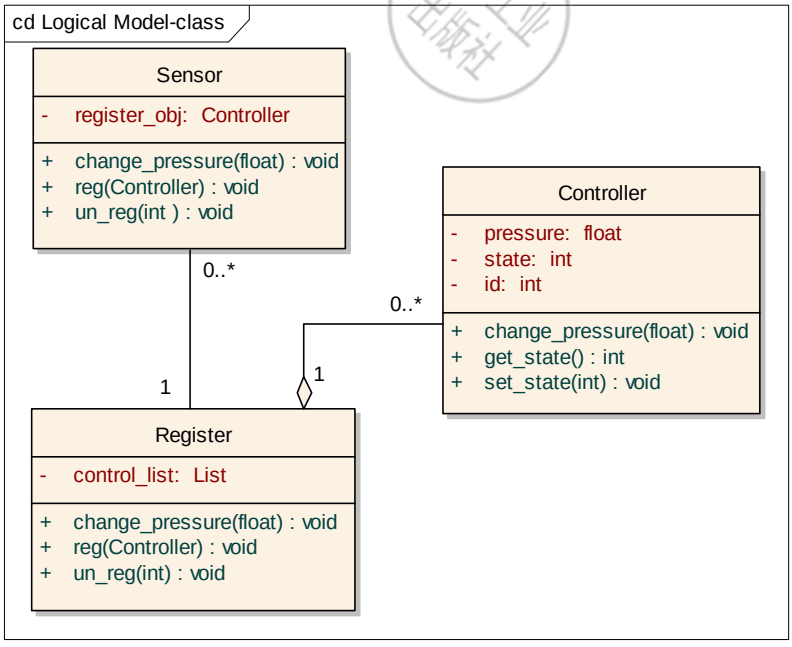


图 19-6

类图表达了类的内部结构，以及类间的关系。

19.5.2 StarUML 的类图

StarUML 类图的画面如图 19-7 所示。

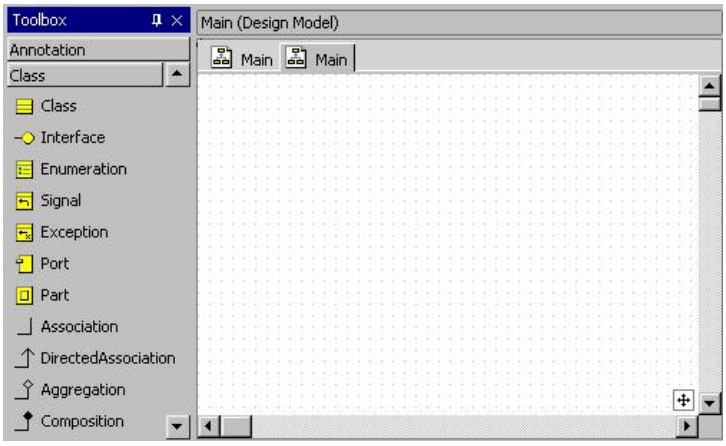


图 19-7

左边是 UML 类图的基本图示，也就是俗称的工具箱（ToolBox），绘图时可以直接从工具箱把图示拉出来，摆入屏幕中央的绘图区里。例如，有个冰箱温度监制系统，当人们启动节能调温功能，并设定温度范围，此系统指挥相关硬件进行温度侦测，如果温度超越预定范围，就进行减温或加温的修正，使其维持于预定温度范围之内，一直到人们关闭调温功能为止。此时可利用 StarUML 画出类图，如图 19-8 所示。

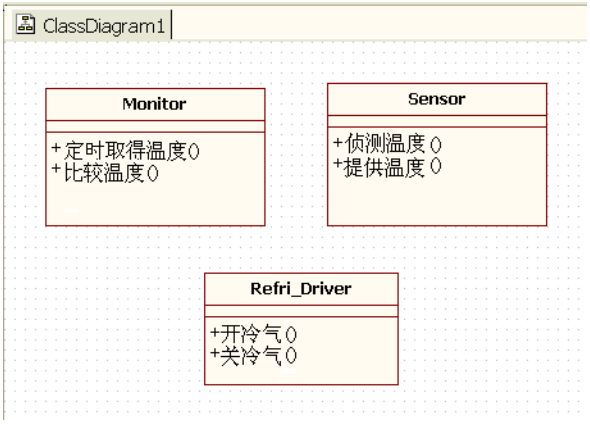


图 19-8

19.5.3 JUDE 的类图

JUDE 类图的画面如图 19-9 所示。

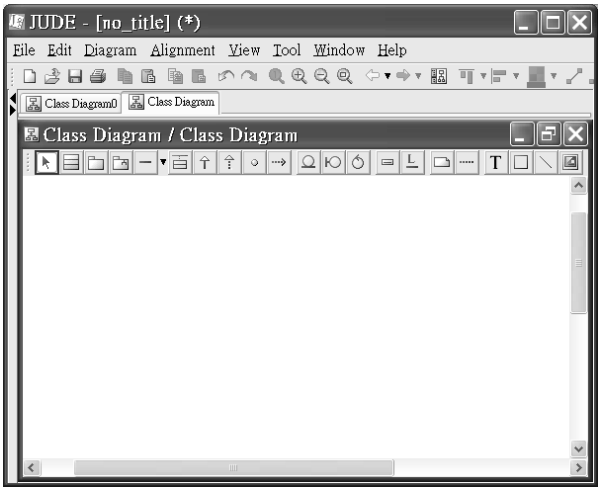


图 19-9

上边是 UML 类图的基本图示，也就是俗称的工具箱（ToolBox），绘图时可以直接从工具箱把图示拉出来，摆入屏幕中央的绘图区里。例如，在前面 19.3 节里，Book 对象内含一个 Author 对象。此时可利用 StarUML 画出类图，如图 19-10 所示。

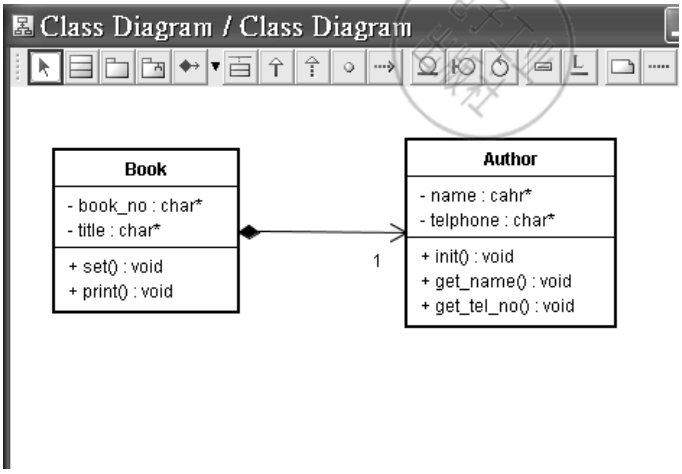
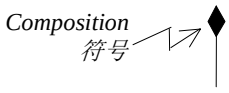


图 19-10

（1）组合/部分（Assembly-parts）关系

——使用 Composition 符号表示：



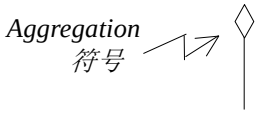
组合 / 部分关系，常称为 APO（A part of）关系；例如，汽车是“组合”，其内含各零件

是“部分”。门是房子的一部分，所以房子是“组合”，门是“部分”；此外，窗子也是房子的“部分”。在我们周围的产品结构中，处处可见组合／部分关系。例如：灯泡含灯芯、灯帽及玻璃球；书本含封面、目录及内容等部分；计算机系统含屏幕、键盘、主存储器及磁盘驱动器等。

上述关系，其整体与部分间有共生共灭的密切关系。例如：一个灯泡破了或者烧坏了，通常整个灯泡，包括其内部的灯帽、灯芯、玻璃球一齐都被丢弃了。于是，在软件系统中，这些部分对象（如灯芯）皆会随着整体对象（灯泡）的消失而一起消失。反之，司机对汽车而言是不可或缺的，但没有人认为司机是汽车的模块，原因是一部汽车报销了，司机可还存在呢！然而，司机仍是汽车的一部分，因为在空间上，汽车对象中，仍包含了司机对象。因此，汽车与司机之间仍是整体／部分关系。

（2）包含/内容（Container-contents）关系

——使用 Share Aggregation（简称 Aggregation）符号表示：



例如：笔芯是自动铅笔的一部分，但笔芯并不与铅笔共生共灭。同样地，干电池是手电筒的一部分，但两种对象并非共生共灭。这种整个/部分的关系，也让我们很容易找出相关的对象和类，同时也能利用这种关系来把软件中的对象组织起来。例如货柜对象，可将不同类的异质性对象整合起来。

19.6 如何得到类

19.6.1 从对象归类而得到类

前面我们一直强调心中要有对象，并将对象归纳为各式各样的类。着手设计一个系统或写一个程序时，第一件会出现在脑海中的问题是：对象那么多，哪个是跟此系统或程序有关的呢？例如：在设计一个销售系统时，“用户”是一重要对象，产品及订单也是重要对象；至于原料的产地及原料的供货商虽然是明显的对象，但不见得与销售系统有关。反之，若您所设计的是采购或生产系统时，原料、产地及其供货商就成为重要对象了。在寻找对象的过程中也会让您对所要设计的系统有更清楚的认识。现在就来介绍较实用的找对象方法。其中最常见的是，从有关文档着手。在文档里会发现下述线索，再进而找出对象。

（1）人（People）——人是任何系统的重要角色，通常是最容易找到的对象。例如：公司有 5 位销售员，各负责一个地区的任务，并与该区的用户联络。从这段叙述中，就可发现两种对象——销售员及用户，每一位销售员皆是对象；同样地，每一个用户皆是对象。

（2）**地方**（Sites）——地点是很容易发现的对象。例如：您从订单上看到产品将送达的目的地、用户的所在地。再拿旅行社行程来当例子，各旅行团将在不同的观光地区停留，各观光地点皆是对象。

（3）**事物**（Things）——在可摸到或看到的实物中，也容易找到与系统有关的对象，例如：产品是销售系统及生产系统的明显对象，原料细项是生产及库存系统的重要对象。就饭店管理系统而言，“房间”是重要对象。“书本”及“杂志”为图书馆或书店管理系统的明显对象。

（4）**事件**（Events）——在企业界最常见的事件是“交易”，当事件发生时，我们会去记录它发生的时刻、有关金额等。这种我们所关心的事件也常是重要的对象。值得注意的是，这些事件是已发生的，为一项行为或动作。所以在文档中，常是一个句子的动词。例如：今天共有 3 种原料已降至安全存量之下，所以共订购 3 种原料。这每一“订购”（Ordering）事件皆为对象。就飞机场的控制系统而言，每次飞机“起飞”或“降落”皆是重要对象。而“提款”及“转账”皆为银行系统的重要对象。“挂号”及“就诊”为医疗系统的对象。

（5）**构想**（Ideas）——与企业营运或机构管理有关的“构想”或各种“计划”或其他概念；这些无形的，但决定企业活动的构想，常是重要对象。例如：公司正拟定 3 种广告策略，其中每一种策略就是企业营销系统的重要对象。这家公司正透过 2 种渠道与小区居民沟通，这里的渠道也是概念性的对象。

（6）**外部系统或设备**（External Systems or Devices）——软件系统常会与其他系统沟通，互相交换信息。有时也由外部设备取得交易数据或把处理结果送往外部设备。这些外部系统或设备也是对象。例如：库存系统与采购系统会互相沟通，对库存系统而言，采购系统是对象；反之，对采购系统而言，库存系统则为对象。如果收款机直接把交易数据传送给销售系统，则收款机是此销售系统的对象。如果股票系统直接把数据传送到交易市场的电视显示屏上，则对股票系统而言，电视显示屏是对象。

（7）**组织单位**（Organization Units）——企业机构的部门或单位。例如，在学校管理系统中，教务处及训导处等单位皆是对象。

（8）**结构**（Structures）——有些对象会包含其他对象。所以在对象中常能找到其他对象。例如，在学校的组织单位——教务处，含有小对象如注册组及学籍组等。在汽车对象中可找到引擎、轮胎及座椅等对象。在“房屋”对象中，会发现到厨房、客厅、沙发等对象。

以上介绍的是常用的寻找对象方法，会寻找对象之后，就必须将对象分门别类，并且了解类之间的关系，以便把它们组织起来。例如，在公司的人事结构中，可发现人因扮演角色的不同而分为不同种类的对象，如推销员、司机、经理等等。汽车可分为跑车、巴士、旅行车等不同种类的对象。

19.6.2 从领域概念（Domain Concepts）找到类

这是比较高效的途径，也是面向对象分析（Object-Oriented Analysis, 简称 OOA）的基本技巧。OOA 的目的是要表达特定应用领域（Application Domain）里的专业知识（Domain Knowledge），然后将这专业知识转换为计算机可读的程序（Program），就能顺利存入计算机中，此时计算机就拥有专业知识了。执行程序时，计算机就能展现出专业的行为，提供专业的服务了。

知识的基本组成要素是“概念”（Concepts）。领域知识的组成要素是领域概念（Domain Concepts）。概念有它的属性（Attribute），概念之间有其关系（Relationship）。系统分析员就将分析到的概念对应到面向对象语言（如 OOPC、C++、Java 等）里的类（Class），如此就分析出类了。简而言之，面向对象分析（OOA）就是要分析领域知识里的概念，并以计算机语言的类机制来表示，然后将其存入计算机中，让计算机拥有专业知识，提供专业的服务。

概念的意义

什么是“概念”（Concept)呢？它是抽象的，代表一群实体，是沟通的重要媒介。例如：“请买杯咖啡”，咖啡是个概念，具有这种概念的人，都会了解这句话的意思。他会凭其经验而想到真实的咖啡。概念代表一个群体——“类”（Class），人们藉由天赋的能力运用经验去想到其所代表的实际东西——“对象”（Object）。例如您听到“买一只吉他”，这“吉他”概念让您想到经验中的吉他，而去乐器行买一只“真实的吉他”回家。一般而言，概念包括 3 个基本要素：

- 符号（Symbol）：代表它的文字、图形或影像。
- 含意叙述（Intension）：它的明确定义。
- 代表的东西（Extension）：它所代表的群体。

例如，Sale 是符号，大家都知道它的含意是：表示一个卖出商品的事件。大家也能想到实际的交易情况。所以 Sale 是一个概念。在不同的领域里，Sale 的定义可能不同，代表一群不同的事物，为不同的概念。像 Sale、Order、Purchase 等代表一项交易事件（Event），皆是重要的概念。至于像 Receipt、Invoice、Ticket 等皆为重要的概念。这些概念的符号，自然成为领域专家或知识工作者之间互相沟通的主要词汇（Vocabulary）。因此在叙述领域的相关文档里，常用词汇是此领域里的主要概念符号。

找出领域知识里的概念，就是找出软件系统的对象和类。找出软件系统的对象，就像寻找水的组成元素——H 与 O。例如麦当劳企业有汉堡、薯条、玩具、特餐、点餐、订购玩具、用户、员工、玩具商、分店等的概念，将它们对应到软件系统的类，所以在麦当劳的软件系统里就会有汉堡、薯条、玩具、特餐、点餐、订购玩具、用户、员工、玩具商、分店等的类。

因此在叙述业务的相关文档里，通俗而常见的词汇就是此业务领域里的主要概念。即使是一般的文档也处处可见领域概念。

举例：嫦娥的故事

“后羿从西王母处请来不死的药，嫦娥偷吃了这颗灵药，成仙了，身不由主飘飘然地飞往月宫之中，在那荒芜的月宫的中度着无边的寂寞岁月。”

虽然嫦娥可能是传说虚构的，并非事实（Reality），但是确确实实是我们心中的清晰概念，传说中的主角，所以是个重要的类，如图 19-11 所示。

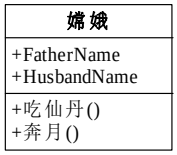


图 19-11

“嫦娥”的确是一个共通的概念和术语，只要有人说出这两个字时，别人几乎都知道其意义。一旦找到了核心的概念之后，就能继续发掘与它具有关联的其他概念了。例如，跟“嫦娥”具有密切关联的概念是：月亮和仙丹，这可以通过图 19-12 表示。

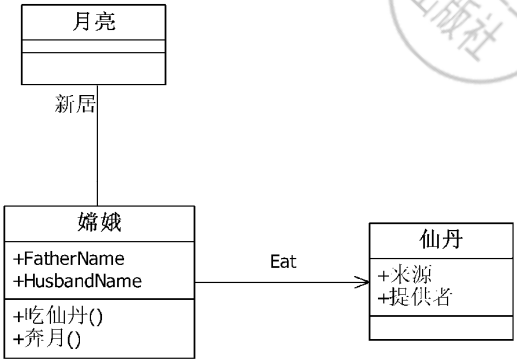


图 19-12

因为人们是藉由心中的概念来认知领域中的事物的，并传播给别人，与人分享；而 OOA 和 UML 模型则表达人们心中的概念，也是表达人们对领域中事物的认知（即 Through the concepts they acquire）。

只要更仔细观察领域专家们心中的更多主要术语或词汇，就能找到更多的概念了，然后使用世界标准的 SysML/UML 模型来表示。就能把知识传播到世界各地。不仅上述图 19-10 的名词概念而已，其相关的动作也常是重要概念，如图 19-13 所示。

动词常常代表一项事件（Event）的发生，而人们常从人、事、时、地、物等各层面去描

述一个事件的发生情境。譬如，吃仙丹就有动作（吃）的对象——仙丹，动作的主角——嫦娥，当然还有地点、时间，甚至仙丹来源等等。只要你懂得依循概念的关联来逐步找出相关的领域概念，系统分析就变成一种非常有趣的事情了。

专家们早已经对自然界或企业界定义了许多概念了，而且定义得一清二楚，连名称也都取好了，例如“汉堡”、“信用卡”、“饭店订房”、“大学”、“城市”、“CPU”、“结婚”、“开车”等，甚至数十年来都未曾更改过。

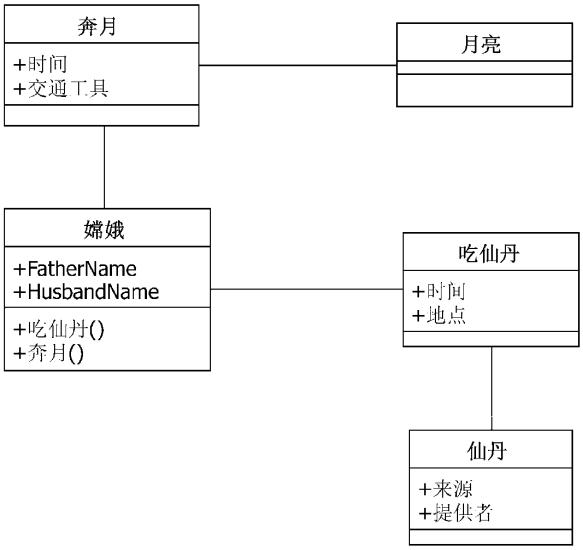


图 19-13

由于概念是非常稳定的。所以，OOA 鼓励大家以概念为中心，基于概念而得出类图，再以类为单元，将计算机程序里的函数和指令纳入类里。于是呈现出一个美好的景象：善变的指令被封装于稳定的类（概念）里。这也达到了 OOA 所追求的另一目标：封装（Encapsulation）。大家都知道封装良好的产品常常更令人喜爱。

此外，概念是共同的词汇，只要在同一领域里（例如餐饮业），概念是大家沟通的媒介。就如 James Martin 所说：

“概念是大家所共享的。概念是人们沟通的共同词汇。”

（Concepts are shared by others. Concepts provide the common vocalulary for communication.）

由于类的名称就是概念的名称，它是人人都理解的东西，因此由类所组成的计算机软件系统也就成为人人易于理解的东西了。总而言之，以领域知识概念为中心的系统分析模型，非常有助于开发出既稳定又易于理解和使用的系统。

第 20 章　UML 用例图

20.1 为什么需要用例图

用例图（Use Case Diagram）表达了系统提供用户哪些服务，这些是用户看得见的或感觉得到的，而且是用户觉得有意义的，愿意付钱去购买的服务。由于是用户付钱去买的系统服务，是系统需求的核心部分，所以它是 UML 想要表达的重点所在。

简而言之，用例在系统的众多功能当中，属于直接提供用户服务的部分。例如麦当劳餐厅可视为一个系统，它具有许多功能，像“卖汉堡”、“采购蔬菜”、“炸薯条”、“卖咖啡”、“提供厕所”、“提供小孩游戏”、“烘焙咖啡”，等等。其中有一小部分是用户能看得见、感觉得到又觉得有意义的服务，就是用例；而系统的其他功能就不称为用例，而只是一般的系统功能罢了。

于是，可区分出上述麦当劳系统功能当中，属于用例的为：“卖汉堡”、“卖咖啡”、“提供厕所”、“提供小孩游戏”等。

为什么要特别看待这些功能呢？因为这些是用户认为有价值的服务，也就是可以向用户收取费用的服务，属于“收益面”的功能。至于像“采购蔬菜”、“炸薯条”、“烘焙咖啡”等都是属于“成本面”的功能。UML 用例图就是特别突显系统收益面的功能，期待在系统创建初期就能确保系统切中用户的期待和需要。

换句话说，UML 用例图乃用于表达用户观点下的系统功能需求，其着重于系统的行为需求，例如描述系统如何（How）服务用户，包括系统与用户交互及对话的流程，确保用户能很愉快地使用系统的服务。所以，UML 用例图在确保系统的可用性（Usability）上，是个强有力的工具，它擅长表现用户为什么（Why）要使用（Use）系统，以及如何（How）使用系统。软件开发者从用例图能深入了解许多“为什么”（why）。例如，为什么需要这个软件？为何用户要去接触这个软件？用户想达成什么事？当我们用心探索围绕“使用”的一连串“为什么”之后，就能导出高度可用性的软件了。

20.2 用例的内涵是什么

用户使用系统时，其目的是期望系统提供服务或产品，一个用例就表达一项系统服务。当系统在提供完整的服务或产品的过程中，会执行一连串的小活动；在这些活动当中，也常会跟用户沟通，取得用户的指示而调整其活动或顺序。例如，人们去麦当劳餐厅买汉堡时，柜台人员会向用户询问是“外带”还是“内用”，而决定其包装程序。因此，每个人去买汉堡时，其使用“麦当劳服务系统”的途径会有些相同，也可能有些不同。其中，每个人使用系统的途径就是个实例（Instance），特别称为场景（scenario）或实景。虽然每个人使用系统的场景会有些差异，但是若用户的目标（Goal）是相同的，则其场景也会极为类似。那么这些类似的场景的集合就是个类（Class），这种类特别称为 Use Case，其实例

（Instance）就是场景（Scenario）。至于一般的类，其实例是对象（object），两者有所区别。归纳如下：

- Use Case 类——实例是 Scenario。
- 一般类——实例是 Object。

由上所述，人们是依用户的目的（Goal）来将场景分门别类（Classify），由此得到不同的用例。由于同类（Use Case）内场景大同小异，所以就可把其共同而重要的特点（如活动和顺序）以文字描述出来，根据 Wirfs-Brock 的建议，可只描述用户与系统之间的对话，如图 20-1 所示。

Use Case 描述

UC:买汉堡
{

user action	*system response*
指定购买汉堡并付款	
	收钱、做汉堡 给收据和汉堡
拿走汉堡	

}

图 20-1

这就称为“Use Case 描述”（Use Case description），也称为 Use Case Definition。因此，用例是一群类似 Scenario 的抽象的表达或描述。由于 Scenario 很多，且各有些微差异，不易逐一去描述或定义它，所以就藉由用例来描述一群类似 Scenario 的共同特性（Features），让人们较易于了解与掌握如何使用系统。如此，自然创造出好用的计算机系统了。因此，用例的主要内涵及其表达为：

- 描述该服务的名称及返回的信息类型，它描述 What。
- 描述服务过程中，用户与系统之间有哪些交互（即两者的对话流程），其描述 How 和 When。但是请留意，这个 How 不是员工做汉堡的 How，而是用户如何与系统最好交互的 How。请你务必区别开它们。
- 描述用户（又称为 Use Case 的 Actor）的角色，它描述 Who。

20.2.1 用例图表达 What 及 Who

例如，一个餐厅的用例图如图 20-2 所示。

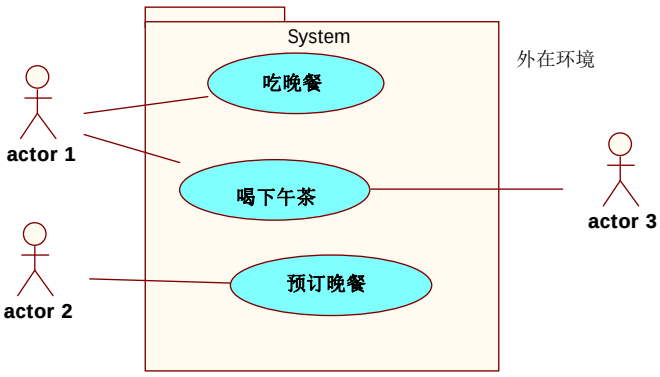


图 20-2

图 20-2 表达了 What 和 Who。在 What 方面，一项用例表达了用户对系统的一项需求，也就是系统的一项责任（Responsibility）或功能（Function）。

在 Who 方面，一个系统常含有许多的用例，而环境里的人或物（即系统的用户）在不同时刻，可能扮演不同角色来使用系统中的不同用例，以取得系统的不同服务。“角色”（role）代表一群对系统怀有相同兴趣或目标的人或物，在 UML 里称为 Actor。

例如在图 20-2 里，当您扮演 actor 2 时（怀有一个目的），就可使用系统，此时系统就执行 UC：预订晚餐来服务你。同样地，当您扮演 actor 1 时，就可使用“UC：吃晚餐”或“UC：喝下午茶”来服务你。

20.2.2 用例描述表达 How 及 When

图 20-2 比较偏重于用例的功能面，主要在行为方面，说明系统如何配合 Actor “使用”系统的过程，才能顺利达成 Actor 的目的。也就是说用例的行为决定于 Actor 的使用过程。因此，在用例描述里应详细描述 Actor 使用系统的途径，以及系统应该配合的行为。例如上图里的“UC：吃晚餐”，可写个 Use Case 描述来描述它们，如图 20-3 所示。

根据 Jacobson 对用例的定义：一个用例就是一连串通过系统的事件（a specific flow of events through system）。其包括系统对这些事件的反应，以及用户对事件的反应，即用例也描述了系统的外观行为。所以，可藉由用例来表达系统的行为需求。然而如何描述这些事件及用户与系统之间的交互情形呢？基本上，我们可用文档来描述，这就是刚才提过的“Use Case 描述”了。

由于 Use Case 描述能更清楚地表达出系统的行为需求，每个用例皆表达了系统行为需求的一部分。由于所有用例的总和等于是系统，因此，用例的总和，就表达出整个系统的行为需求。在 Use Case 图中，只表示出用例的外观行为，而把用例内的行为封装起来。到了系

Use Case 描述

UC: 吃晚餐
{

user action	system response
点餐	
	上开胃菜
	上主餐
请上点心	
	上点心
买单	
	结账
	开收据
拿走收据	

}

图 20-3

统设计阶段，就可使用下一章将介绍的 UML 序列图来表达用例内部的行为（Internal Behavior）了，这是 UML 序列图主要用途之一。当用例内涵得到用户认可时，软件开发者可以针对用例而思考如何安排对象的互相沟通与合作。当系统服务确实能符合用例所表达的内涵时，用户会乐意付钱购买该项服务，这表示系统开发是成功的。

20.3 用例与对象的密切关系

你必须仔细分辨用例（Use Case）、对象（Object）与系统（System）的不同，以及它们的相关性。用例是用来描述系统的服务细项。用例是系统的一个观点（View），是系统用户（User）内心对系统的期待。所以 UML 用例图表达了用户心中期待系统给予的服务，也就是用户对系统的需求（Requirements），属于系统外观（External View），如图 20-4 所示。

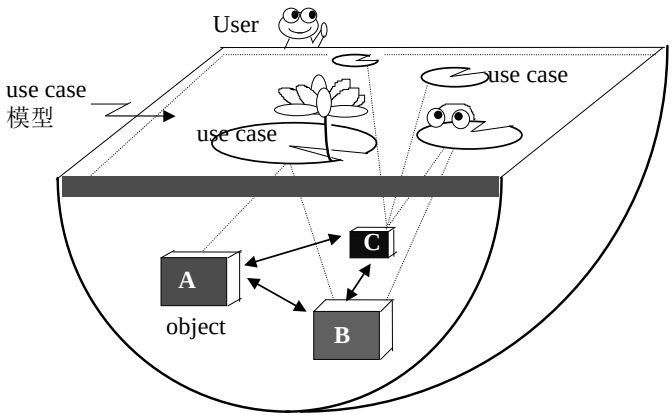


图 20-4

在用户眼中的系统就像青蛙眼中的池塘，用例就像青蛙眼中的荷叶，青蛙眼中的各片荷叶之间并没有相连，也不关心各片荷叶在水面下是如何连接在一起的。至于架构师（Architect）或设计师（Designer）则关心水面下的根茎应如何支撑水面上的荷叶，水面下有许多子系统（Subsystem）或对象，其支持着水面上的各片荷叶。不同的用例可要求同一个对象来提供服务，即一个对象可同时参与许多个用例，如图 20-5 所示。

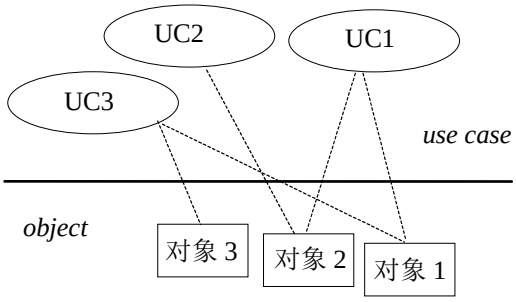


图 20-5

这种共享对象的关系是俯拾皆是的，也是好的用法。只是在 UML 用例图里并不需表达出这种共享对象的关系，所以在用例图层次看不到这种关系，这种关系表现于下一章将介绍的 UML 序列图里。

20.4 用例的经济意义

面向对象软件开发的目标就是通过对象的迅速组合来提供多样化的系统服务。多样化意味着“多功能”，用户想买的（what customer want to buy）功能就是系统的用例。例如咖啡机提供许多按钮像<煮咖啡>、<煮开水>、<加糖>、<加奶精>等给用户使用。所以用例与用户的期待和满意度密切相关，这关系到产品的销售和收益成绩。用户关心的是产品能否满足他的目的（Goal）并且方便使用（Easy way to use）。一个系统好像一个足球，从外观来看，一个足球表面有许多用例，如图 20-6 所示。

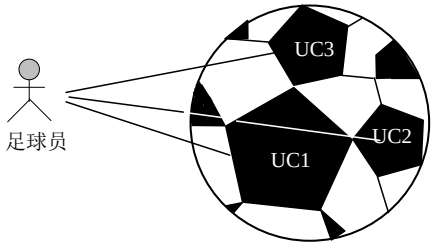


图 20-6

以上从用户角度来看，仅看到系统的外观而已。如果改从工程师角度看，一个用例就代表一群对象联合执行（Join operation）一项服务。每一个用例的幕后都有一群对象互相沟通与合作。好像看到足球的内部结构，如图 20-7 所示。

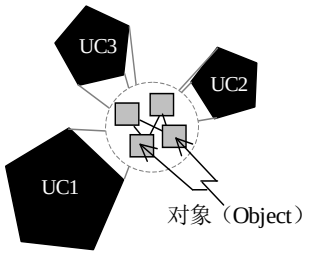


图 20-7

一个对象可参与许多个不同的用例，让对象有更多“复用”（Reuse）的机会，则对象的可靠度提升了，复用机会也随之提升，整个系统的质量也就提升了。开发对象是成本、付出；销售功能和用例是收益、获得；UML 用例图让我们在建模阶段就将收益与成本紧密连结起来，这对系统在市场上的成功与失败至关重要。

再如，用户付钱听交响乐团演奏就是用户（User）对乐团（即 System）的消费，乐团所演奏的每一首乐曲都是用户的一项买单，也就是一个用例。交响乐团由一群演奏者密切合作，演奏出如行云流水般的旋律，让听众心旷神怡。如图 20-8 所示。

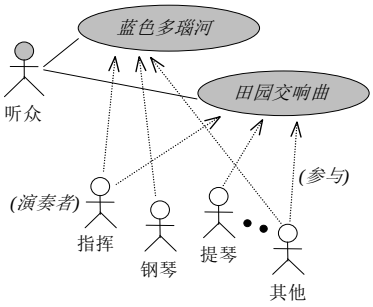


图 20-8

所以，用例连结了乐团的收益（听众买单）与一群演奏者的付出（成本）。从上述的例子中，您能深刻领悟用例的含义，其中的重点是，基于用户的观点来看系统，用例是系统提供给用户的一项服务（收益），而服务的背后则是一系列的活动（成本）。

归纳上面所述，用例是收益与成本的衔接点。在收益方面，用例确保了系统的可用性。直到目前为止，绝大多数的软件开发工具和技术是用来建构软件的，很少是用来分析用户为

什么（Why）要使用（Use）软件系统，或如何（How）使用软件系统的。因为软件的设计者大多带着计算机技术的眼光来看整个相关的事情。十足了解如何运用技术于产品的设计师，却因了解太多技术而忽略用户的观点和感觉。因而，目前的困境是软件人员并不够了解用户的实际工作，以致软件无法符合用户的需要。

我们必须深入去了解许多“为什么”（Why）。例如：为什么需要这个软件？为何用户要去接触这个软件？用户想达成什么事？当我们用心探索围绕“使用”的一连串“为什么”之后，就已经基于“用户的观点”去理清了系统提供各种功能的目的。也基于“用户跟系统交互的角度”去理清了用户跟系统的交互情形，并描述出系统如何协助用户完成其工作。用例让软件人员暂时不考虑软件系统内部的行为和结构，而专注于理清用户“为什么”去“使用”这个系统，充分正确地掌握用户的需求，然后才能设计出好用的软件来。用例是让软件人员去了解用户需求的利器。

以上是属于收益层面的。在成本层面，一个对象可参与许多个不同用例，这鼓励软件开发人员尽量使用现有的对象，让对象有更多“复用”（Reuse）的机会，则对象的可靠度提升了，复用机会也随之提升，整个系统的质量也就提升了。就如同交响乐团，每一首演奏曲目都是演奏者或乐器的复用。也像电池一样，它可用于 MP3，而 MP3 上的“播放”、“录音”等都是 MP3 的用例，都代表电池的复用机会。此外电池还能用于手电筒、电玩等各种场合，也都是电池的复用机会。因此用例引导我们基于“对象交互的角度”去厘清了对象整合、抽换和复用以创造多样化产品的机会和实现方案。

20.5 用例间的关系

就内观而言，各用例在其幕后都是共享同一群对象。也就是用例之间自然就具有“共享模块”的关系。由于 UML 用例图只呈现外观，所以在用例图里看不到这种关系。在外观方面，UML 定义了两种关系，分别是“包含”（Include）和“扩充”（Extend）关系（如图 20-9 所示）。

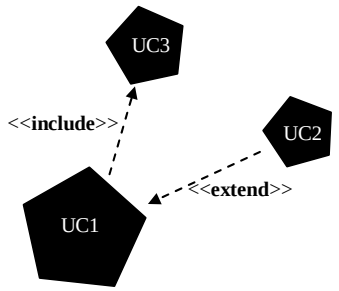


图 20-9

现详细说明如下：

- 包含关系

两个用例之间可以有“包含”（Include）关系，用以表示某一个用例的对话流程中，包含着另一个用例的对话流程。当出现几个用例有相同部分的对话流程时，将相同的流程记录在另一个用例中，前者称为“基础用例”（Base Use Case），后者称为“包含用例”（Inclusion Use Case）。这样一来，这些基础用例就可以共享包含用例，而且未来其他的用例只要创建包含关系，就可以立即享用已经在其他用例定义好的相同对话流程了。如图 20-10 所示。

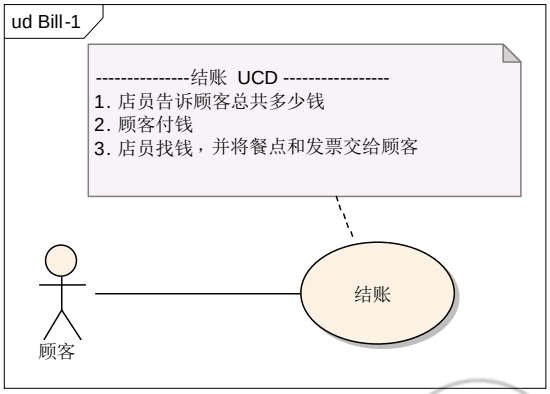


图 20-10

这是“UC:结账”的用例图和 UCD。经过一些时日之后，系统新增功能了，多了一个新的用例如图 20-11 所示。

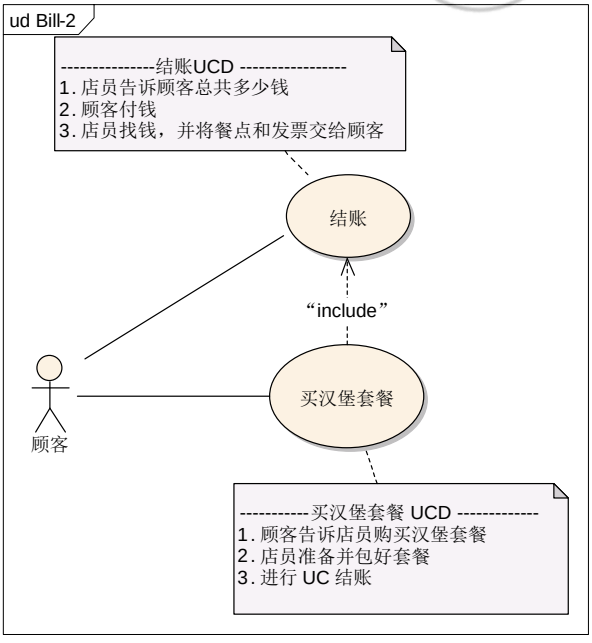


图 20-11

此时，“UC: 买汉堡套餐”的 UCD 就不必重复编写一次“结账”部分了。可单独编写 UCD 如下。

20.5.1 UC 描述：结账

店员告诉用户总共多少钱；

用户付钱；

店员找钱；

店员将餐点和发票交给用户。

20.5.2 UC 描述：买汉堡套餐

用户告诉店员购买汉堡套餐；

店员准备并包好餐点；

进行“UC：结账”。

- 扩充关系

两个用例之间可以有“扩充”（Extend）关系，用以表示某一个用例的对话流程，可能会依条件临时插入另一个用例的对话流程中。前者称为“扩充用例”（Extension Use Case），后者称为“基础用例”（Base Use Case）。有了扩充关系后，便可以将特定条件下才会引发的流程记录于扩充用例中。在执行基础用例期间，可以只是单纯地执行基础用例所记载的流程；但是在特定条件发生时，则会额外插入并执行扩充用例所记载的流程。如图 20-12 所示。

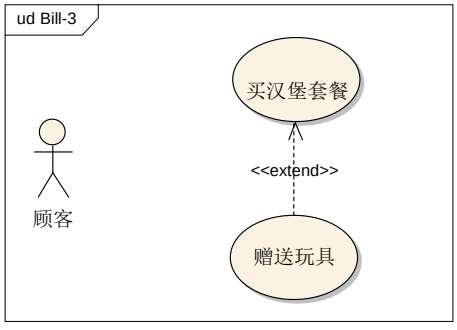


图 20-12

可写 UCD 如下。

20.5.3 UC 描述：结账

店员告诉用户总共多少钱；

用户付钱；

店员找钱；

店员将餐点和发票交给用户。

20.5.4 UC 描述：买汉堡套餐

用户告诉店员购买汉堡套餐；

店员准备并包好餐点；

进行“UC：结账”；

如果是儿童来买，进行“UC：赠送玩具”。

20.5.5 UC 描述：赠送玩具

店员问儿童想要什么玩具；

儿童选取玩具。

将一个用例附加到一个原有的用例，成为原用例的可选择性段落。执行的情境如图 20-13 所示。

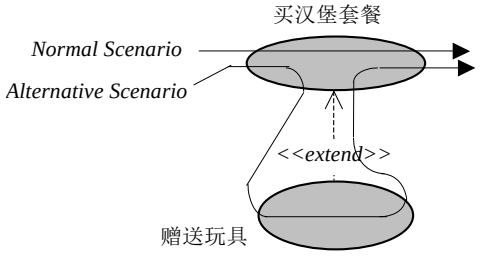


图 20-13

通常，原有的用例表达出正常的情境（Normal Scenario），而新附加的用例表达出特殊或例外的程序（Alternative Scenario）。所以“买汉堡套餐”的 UCD 描述了常态的对话流程。而“买汉堡套餐”的 UCD 加上“赠送玩具”的 UCD，共同描述针对儿童用户的特殊对话流程。

20.6 企业用例与系统用例

20.6.1 上、下层级的用例

前面各节里，并没有对企业（Business）系统与信息系统（Information System）加以区分，因为两者皆是系统，两者皆有其各自的用例和用例图，分别称为企业用例（Business Use Case）和系统用例（System Use Case）。然而，信息系统通常是企业系统的一部分，或者是它的一个子系统。相对而言，企业系统的层级比信息系统高。所以企业用例是指高层级的用例，而系统用例则是低层级的用例，但这只是相对关系而已。

如果你想创建企业的信息系统，通常考虑企业与信息系统两个层级就够了。反之，当你想创建的是嵌入式系统或数码家电等产品时，通常必须考虑多个层级。例如你想规划一个 NoteBook 计算机的用例图，你必须区分出来不同层级的用例图，依据模块大小来分层级。

（1）高层级：NoteBook 层级的用例图，描述 Actor（人）如何使用 NoteBook。

（2）中层级：主板层级的用例图，描述 Actor（设备，如 Hard Disk、Screen 等）如何通过接口（如 USB、RS232）来使用主板。

（3）低层级：CPU 层级的用例图，描述 Actor（设备，如主板里的 IC、内存、OS 软件等）如何通过接口来使用 CPU。

这种上下层级的用例之间具有密切的关联。当你看到“企业用例与系统用例”的名词时，请不要被“企业”这个字眼所局限了，企业与系统只是相对的名词而已，企业一词代表“大饼包小饼”中的大饼，而系统则代表小饼。同样地，谈到鸡蛋时，企业用例代表鸡蛋用例，而系统用例则代表蛋黄用例。依此类推，就以 NoteBook 和主板来说，相对而言，NoteBook 用例是企业用例，而主板用例是系统用例。再就主板与 CPU 来说，相对而言，主板用例是企业用例，而 CPU 用例是系统用例。因此，“企业用例与系统用例”其实就意味着“上层用例与下层用例”的代名词而已。

20.6.2 上、下层级用例的美妙关联

当我们将企业与其信息系统视为一个整体时，企业与系统便成为一体的两个层面（Facet）或层级（Layer）罢了。这样，信息系统便能高效地支持企业来创造出更佳的企业流程（Business process），由更好的流程来提供给用户更满意的服务。就如软件专家 Jacobson 所说：

"The models developed in a business engineering program are an excellent starting point to define architectures, find reusable components, and develop application systems that add value for the customers. "

（进行企业工程时所产出的模型，可作为定义软件架构、找出可重复使用的对象，以及

开发应用程序来服务用户等工作的绝佳基础。）

他又说：

"Using object-oriented business engineering as input, it is a straightforward process to identify the information system models."

（以面向对象的企业工程所产出的模型为基础，来导出信息系统的模型，是个极直截了当的途径。）

他以图示说明导出的步骤，如图 20-14 所示。其详细步骤如下：

- 会使用到系统的 Worker，就成为系统的 Actor。
- 若有些企业 Actor 会用到系统，它们也成为系统的 Actor。
- 针对每一个企业用例，查看该用例的 Actor 及各 Worker，若它们会成为系统 Actor，就为它们个别创建一个系统用例。这意味着每一个企业用例可能由数个系统用例来支持，也就是说，每一个企业用例可能对应到多个系统用例。

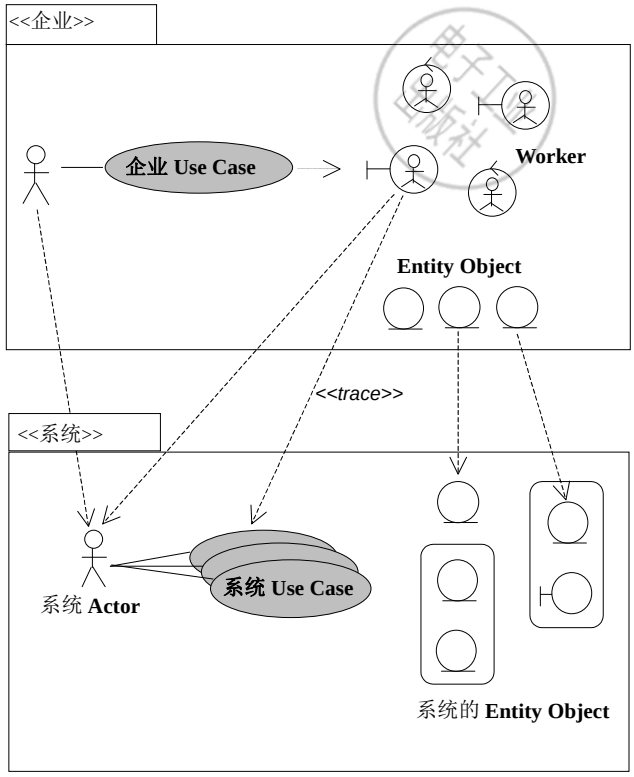


图 20-14

- 企业里的 Entity 对象，代表着 Worker 必须用到的重要事物，这些重要事物必须长期记录与保管，所以它们也成为系统的 Entity 对象。

从图 20-14 中，你可以看到企业用例的幕后有一群 Worker 和 Entity 对象互相合作来提供企业用例所追求的目标，以满足用户的期望。图 20-14 也相当于图 20-15 所示。

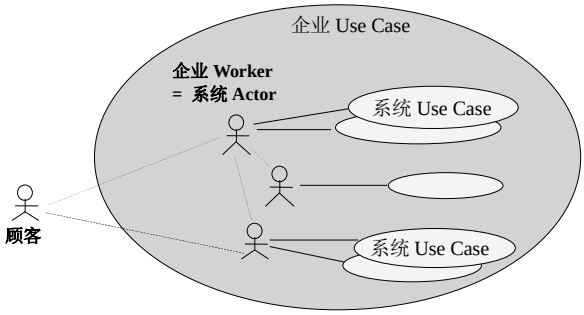


图 20-15

20.6.3 从“上层用例”导出“下层用例”的步骤

上节说明了上、下两层用例之间的美妙关联性，善用此关联性，就能轻易地藉由上层用例的引导而找出优质的下层用例了。为了让你更熟悉导出的程序，下面以 RRR 汉堡快餐店为例详细说明其推导过程，其步骤如下。

Step-1 首先以企业为核心，看看外界有哪些人或单位会跟 RRR 企业来往，如图 20-16 所示。

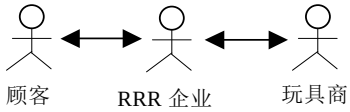


图 20-16

Step-2 确定我们要创建“企业用例图”的对象（即 RRR 汉堡快餐店），想象如图 20-17 所示。

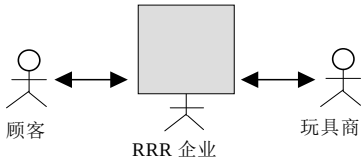


图 20-17

Step-3 把焦点针对用户的目的（Goal）。就 RRR 企业而言，图 20-17 的用户和玩具商皆是它的 Actor。针对各 Actor，确定它与 RRR 企业“打交道”的愿望（Expectation）或称为目的。例如用户跟 RRR 企业打交道的目的有 3 个，如图 20-18 所示。

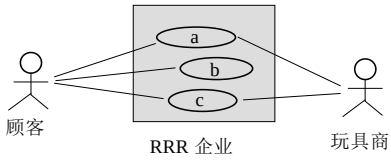


图 20-18

Step-4 把焦点针对企业内部的 Worker。例如，企业里有柜员与信息系统（IS）两个主要 Worker。针对各个用例，厘清 Worker 之间的沟通与合作关系。例如针对“UC: a”而厘清 worker 之间的合作关系，如图 20-19 所示。

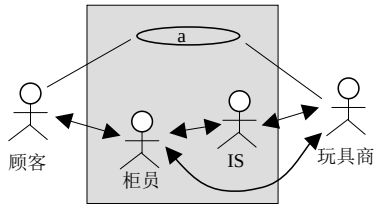


图 20-19

Step-5 确定我们要设计“系统用例图”的对象，如果是 IS 系统，就表示如图 20-20 所示。

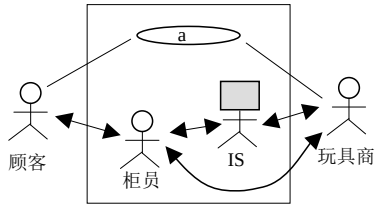


图 20-20

Step-6 针对会与 IS“打交道”的 Worker，逐一厘清这些 Worker 的愿望或目的。例如柜员跟 IS 打交道的目的有两个，如图 20-21 所示。

Step-7 于是，已经从 a 导出 x 及 y 了。接下来，针对另一个“UC：b”，厘清 Worker 之间的沟通与合作关系，如图 20-22 所示。

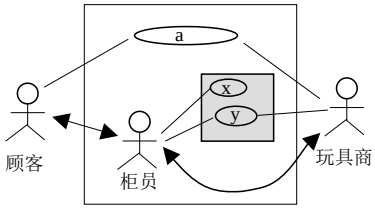


图 20-21

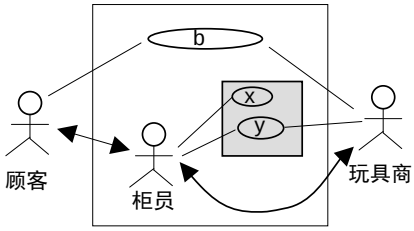


图 20-22

Step-8 在这个新的“UC: b”中，用户通过 Web 直接使用 IS 系统，此时用户也成为 IS 模块的 Actor 了。针对 IS 模块的各 Actor，确定它与 IS 模块“打交道”的愿望或目的。例如用户跟 IS 打交道的目的只有一个：z，如图 20-23 所示。

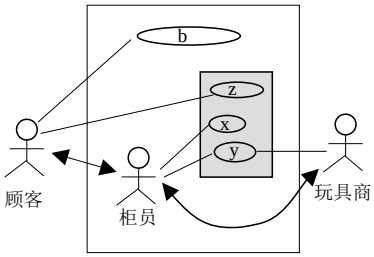


图 20-23

Step-9 就找出优质的系统用例了，如 UML 标准图 20-24 所示。

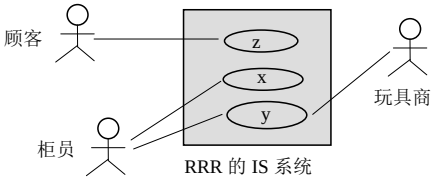


图 20-24

20.7　如何绘制 UML 用例图

——使用免费的 StarUML 工具软件

UML 是世界标准的建模语言，目前有许多相关的工具可用，其中需要付费的工具，以 Enterprise Architect（简称为 EA）最流行，也很便宜。在开放源码方面则以 StarUML 和 JUDE 名气最大。本节简介他们所绘出的用例图，让你有些印象，至于如何熟练其使用方法，并非本书的范围，请你上网阅读他们的使用手册。StarUML 用例图的画面如图 20-25 所示。

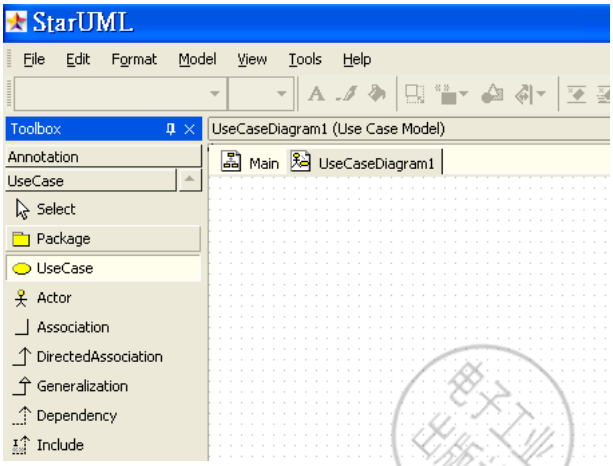


图 20-25

图 20-25 中，左边是 UML 用例图的基本图示，也即俗称的工具箱（ToolBox），绘图时可以直接从工具箱把图示拉出来，摆入屏幕中央的绘图区里。例如，上述 RRR 企业的用例图，可用 StarUML 绘制如图 20-26 所示。

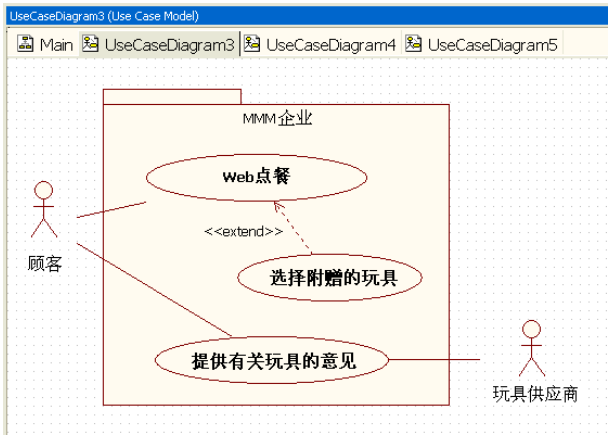


图 20-26

这表达了，用户不仅能上网购餐，并能选择自己喜爱的玩具种类，还可以提供有关玩具的意见。假设 MMM 企业内有两个系统“营运 IS”和“玩具采购 IS”，则依据上述程序，找出 MMM 企业内的 Worker，如图 20-27 所示。

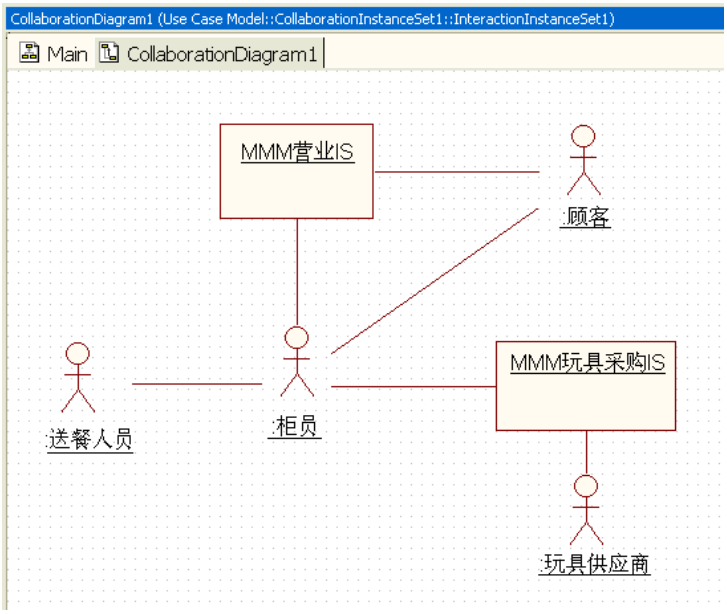


图 20-27

前面已介绍了系统用例的导出步骤：

（1）会使用到信息系统的 Worker，就成为信息系统的 Actor。

（2）若有些企业 Actor 会用到信息系统，它们也成为信息系统的 Actor。

（3）对每一条企业用例，查看该用例的 Actor 及各 Worker，若它们会成为系统 Actor，就为它们个别创建一个系统用例。

例如，针对“MMM 营业 IS”系统，找出它的 Actor，就是用户和柜员。然后厘清当这些 Actor 来使用计算机系统时，其心中怀有哪些目标，就可得到优质的系统（下层）用例了。由于用户直接上网使用 IS 系统，所以用户直接就成为此 IS 的 Actor 了。为了支持上图的“UC:Web 购餐”，此系统不仅需要提供给用户上网点餐的服务，还要提供给柜员来输入送餐记录，于是就可以找到两个优质的系统用例了，如图 20-28 所示。

依样画葫芦，针对“MMM 玩具采购 IS”系统，找出它的 Actor，就是柜员和玩具供货商。为了支持图 20-28 的“UC:选择附赠的玩具”，此系统必须提供给柜员订购玩具的服务，同时还要提供给柜员来查询玩具存量的服务，于是就可以找到两个优质的系统用例了，如图 20-29 所示。

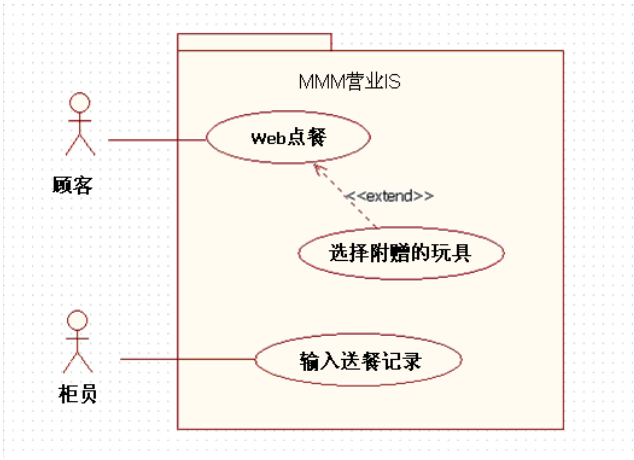


图 20-28

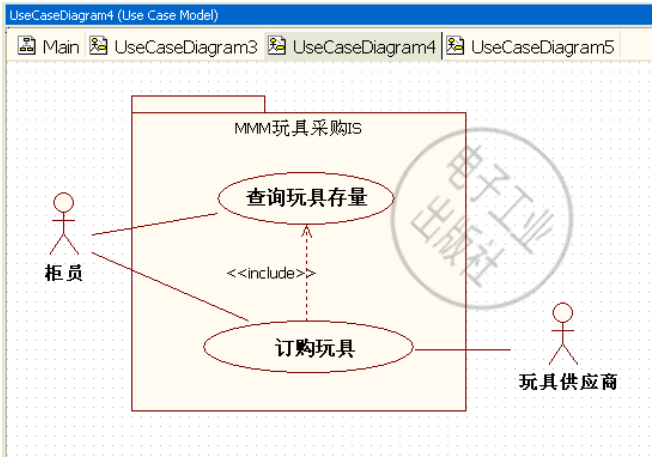


图 20-29

依上所述，是从企业的用例图循序渐进而进行的严谨推导，能确保 IS 系统用例图的正确性。

第 21 章　UML 序列图

21.1 UML 序列图的意义

在上一章介绍过，用例（Use Case）是收益与成本的衔接点。在收益方面，用例引导我们用心探索围绕“使用”的一连串“为什么”的问题，从而厘清了：

- 系统提供各种功能的目的；
- 用户跟系统的交互情形。

用例让软件人员暂时不考虑软件系统内部的行为和结构，而是专注于厘清用户“为什么”去“使用”这个系统。只有充分正确地掌握用户的需求，才能设计出好用的软件来。这些是属于收益层面的。在成本层面，一个对象可参与许多个不同的用例，这须鼓励软件开发人员尽量使用现有的对象（及其类），让对象有更多“复用”（Reuse）的机会，如果对象（及其类）的可靠性提升了，复用机会也随之提升，整个系统的质量也提升了。就像电池一样，它可用于 MP3，而 MP3 上的“播放”、“录音”等都是 MP3 的用例，都代表电池的复用机会。此外，电池还能使用于手电筒、电玩等各种场合，也都是电池的复用机会。因此用例也引导我们基于“对象合作的角度”去进行系统的构建。例如，“开前车灯”的 UCD：当司机需要开前车灯时，就按下前灯开关，摩托车就从电池送电流给前车灯，前车灯就亮了。接着司机可能会调整远近灯，就按下前灯开关，摩托车就变换前车灯（远近灯），替代的灯就亮了。

从系统内部对象的合作角度看，一个用例就是系统内部的一群对象为了共同完成一项用例，而展开的对象交互过程。UML 序列图是用来表达这一交互的情境。例如，图 21-1 就是“开前车灯”用例的序列图。

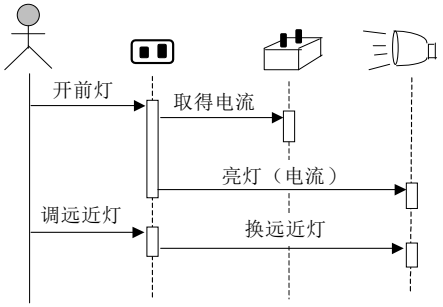


图 21-1

这张序列图准确地表达出用例幕后的对象合作情形。一个用例通常会对应一个序列图，其表达出一群对象经由信息传递、互助合作来达成用户的某项期望。您可以会同用户及领域专家（Domain Expert）一起讨论此序列图的正确性。如果不正确，就得检查用例描述是否正确；如果正确，就得检查对象的角色安排是否正确；如此进行反复讨论及修正，一直到正确为止。

因为这些对象可能还会参与别的用例（如参与演奏交响乐曲）。一个理想称职的乐团成员必须能把每一首曲子都演奏得好。一个好的乐团必须每个团员都能称职和谐地将每一首乐曲演奏得很棒。一个好的系统必须每个对象都能和谐交互达成每一个用例代表的服务。

21.2 UML 序列图的语法

一个用例通常会对应一个序列图，其表达出一群对象经由信息传递、互助合作来达成用户的某项期望。如图 21-2 所示，它表达出银行托收业务的“收件”用例的对象间交互情形。您可以同用户及领域专家（Domain Expert）一起讨论此序列图的正确性。如果不正确，就得检查用例描述是否正确；如果正确，就得检查对象的角色安排是否正确；如此进行反复讨论及修正，一直到正确为止。

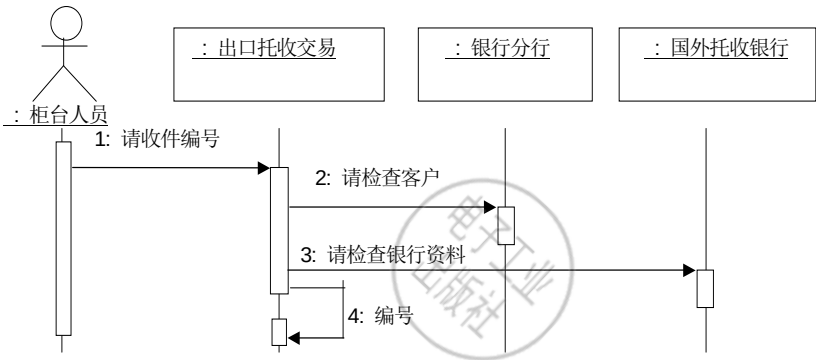


图 21-2

即使图 21-2 是正确的，也不能表示这 3 个对象的角色已经是最佳的了。因为这些对象可能还会参与别的用例（如参与演奏别首交响乐曲）。一个理想称职的乐团成员必须能把每一首曲子都演奏得好。一个好的乐团必须每个成员都能称职和谐地将每一首乐曲演奏得很棒。一个好的系统必须每个对象都能和谐交互地达成每一个用例代表的服务。

现在来简单说明 UML 序列图的语法结构，序列图的上方放置对象，对象下方的虚线代表对象的生命期，此虚线称为对象的生命线。图 21-3 就是“出口托收交易”对象及其生命线。

图 21-3

序列图中以带箭头的实线代表信息与传送方向，箭头端代表接收到信息的对象，非箭头端代表发出信息的对象。对象接收到信息后，会执行某项工作。对象执行工作的期间，称为活动期，UML 以矩形表示活动期。图 21-4 表示出“出口托收交易”对象的有关信息与活动期。

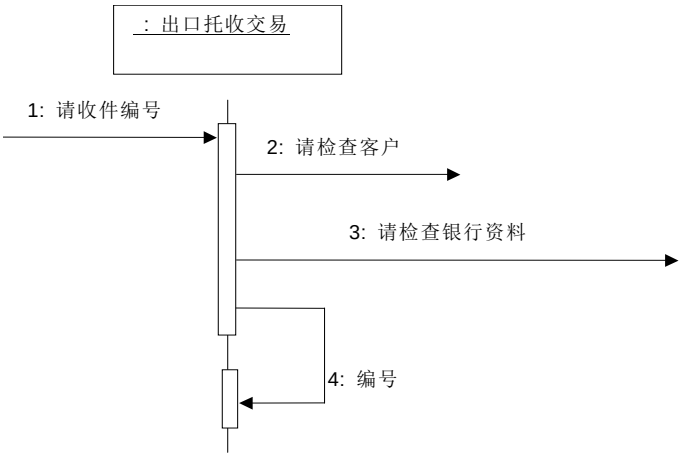


图 21-4

此对象接收了两个信息，就必须执行两项任务，图 21-4 中用两个矩形表示。在执行此两项任务期间，对象发出三个信息，其中两个信息传给别的对象，而有一个是传给自己的。

21.3 如何绘制 UML 序列图

刚刚已经为您简介了序列图的语法和图标（Notation），接着就来介绍序列图的绘制程序吧！如果习惯于使用可视化开发环境像 EA、StarUML 或 JUDE 等工具，UML 序列图的设计工作常会事半功倍。用例是演奏一首乐曲；序列图是描述演奏旋律的五线谱。过去是拿纸和笔来设计乐谱，现在使用 EA 等软件工具就是利用计算机协助创作乐谱（序列图）。在这里将简介在 EA 环境里制作序列图的情形。其设计步骤为：

Step-1 从 EA 的用例 View 里选取特定的用例，如信用卡系统的“刷卡消费”。然后利用小窗口指示要创建新序列图，如图 21-5 所示，EA 就自动开启一个新序列图。

Step-2 决定哪些对象参与用例。

首先从画面用例 View 选取 Actor（卡友），按住鼠标左键将它拖曳到序列图里。Actor 即会出现在序列图内。接着，从画面 Use Case View 选取会参与此用例的对象类，按住鼠标左键将其一一拖曳到序列图里，如图 21-6 所示。

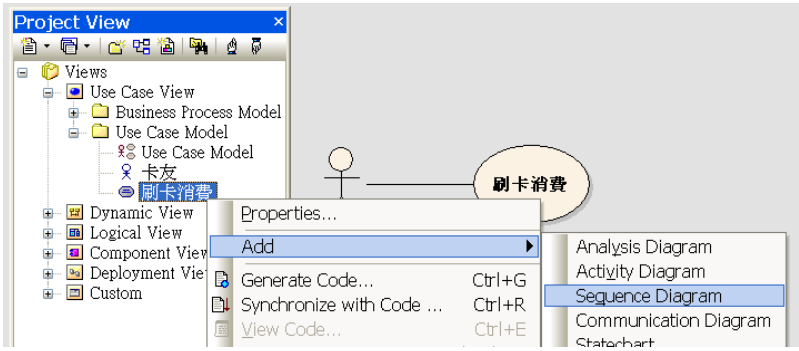


图 21-5

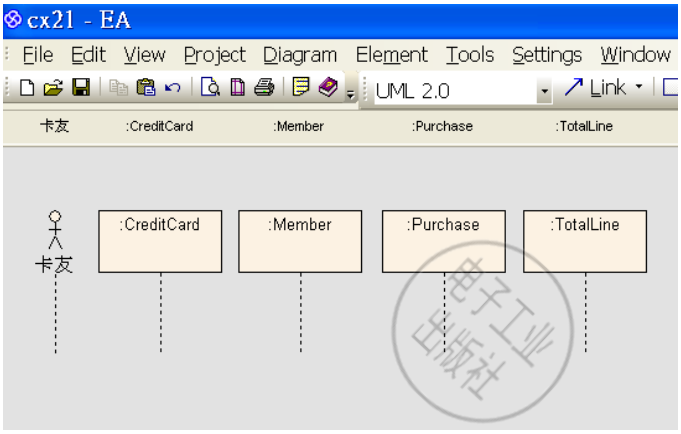


图 21-6

Step-3 安排对象间的合作与交互方式。

[step3.1]

对象是借助互传信息的方式来进行合作和交互的。在画面 Toolbar 里有个代表信息的图示——带箭头的实线图像（icon），使用鼠标左键选取它，而后将鼠标移至序列图中，此刻鼠标光标会变成带箭头的实线。

现在，欲表示出 Actor 传送信息给 CreditCard 对象，便使用鼠标点选 Actor 对象，按住鼠标并拖曳到 CreditCard 对象才放开。就会出现图 21-7 的情形，已表示出 Actor 送信息给 CreditCard 对象，此信息暂定编号为#1。

[step3.2]

现在要对#1 的信息取个名称，叫 MakePurchase。使用鼠标选取（左键双击）#1 的信息图示。画面出现此信息的规格说明（Specification）小窗口，就从其 General 窗体里 Name 子项的下拉式菜单中选取 MakePurchase，然后关闭此规格说明窗口，就为#1 信息取好名字了。

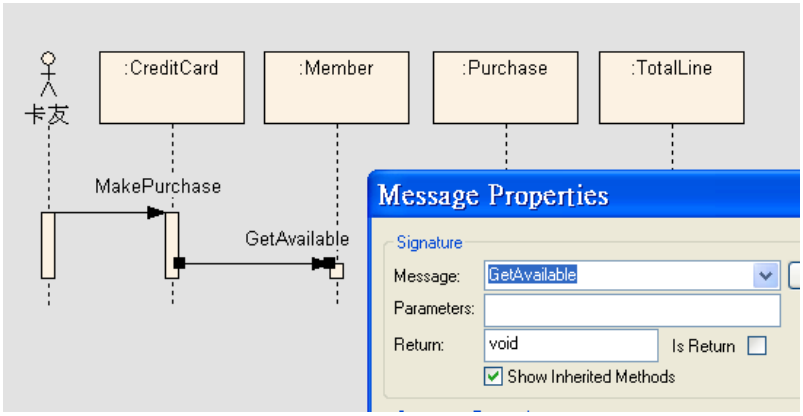


图 21-7

重复上述 step3.1 ~ step3.2 的步骤，就可创建出对象之间互传信息的合作情形了。图 21-8 就是一个序列图。

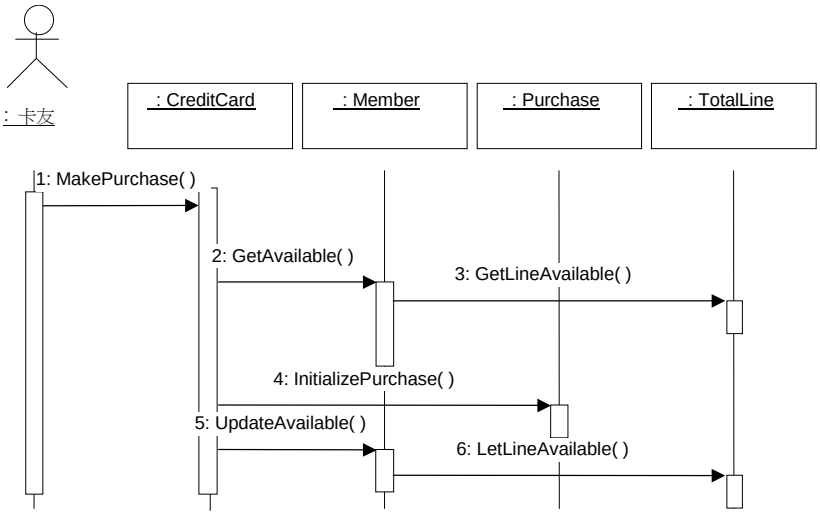


图 21-8

21.4 从序列图落实到 OOPC 类

序列图表达了对象的沟通合作情形，这些对象都是由类所定义的，所以从序列图的对象可对应到类图里的类。而类图里的类会对应到软件里的 OOPC 类，所以 UML 序列图是 OOPC 程序员编写代码时非常有用的指引。很多人就称序列图为系统设计师与程序员头脑相遇的地方，是让双方思维融合为一的关键点。本节将说明如何从序列图对应到 OOPC 代码类。

21.4.1 共享类的考虑

对象不是某个用例专用的，例如电池不仅用来提供电流给前车灯（不只参与“开前车灯”用例），还要提供电流给刹车灯（即参与“刹车”用例）。虽然电池对象在“开前车灯”的序列图里扮演成功的角色（即检验合格了），但是在“刹车”的序列图里可能发现（即检验）它并没有“接头”可直接输出电流到刹车灯！

虽然在已有的用例里，电池对象是很棒的，但面对新需求下的新用例，经由新用例的序列图检验出此电池对象并无法支持新的用例时，这就告诉您：在新的环境需求下，该电池对象已经不合时宜了，有必要修改或是整个换新。事实上，新的用例会促使某些对象变得老旧，因此，必须新陈代谢。大家都知道用户的需求是善变的，用户的期望会随之改变，使得用例本身常会汰旧换新。即使用户的期望并没有改变，用户使用系统的过程也可能有所更动，用例的内涵就会变动，也常促使对象必须修正或替换掉。

所以，无论是新增加用例，还是已有用例的汰旧换新，或是用例内涵有所更动，皆必须再检验有关对象的性能，迅速修正对象或将对象汰旧换新。用例就像一棵树的树叶会不断新陈代谢，对象就像树枝树干也会新陈代谢，于是系统就像树木般自然地新陈代谢，清新而绿意盎然。使用 EA 等开发工具的好处之一就是能快速修改用例图、对象类图及序列图等，无论在系统开发阶段还是维护阶段都有极大的帮助。例如，您可能为了落实到 Web Service 架构上，或者为了增强 Actor 与企业对象（如 CreditCard）之间的独立性，因而加上了一个 Facade 对象。在 EA 环境里能迅速更改序列图，如图 21-9 所示。

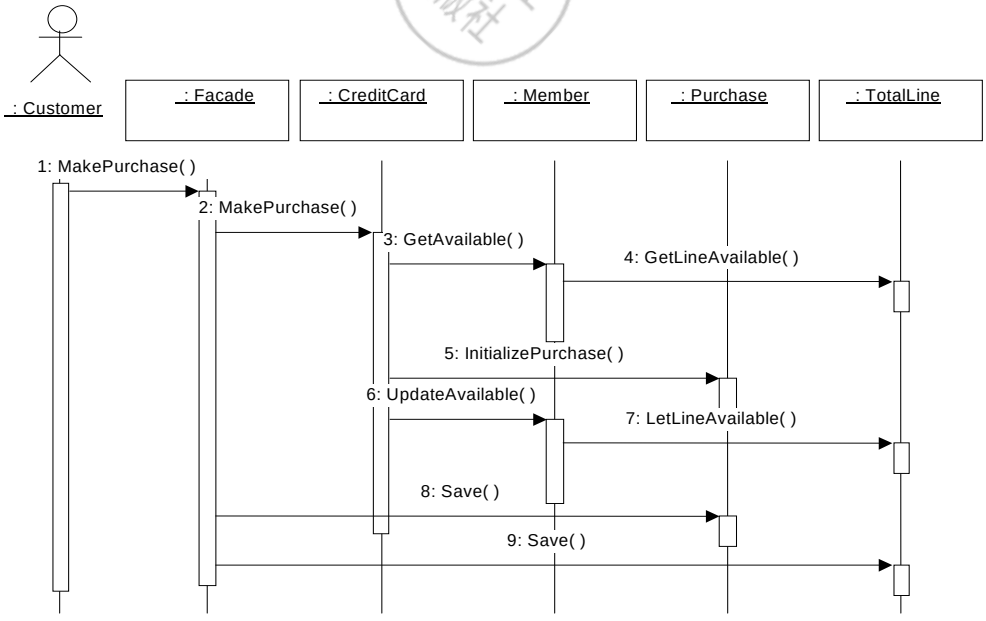


图 21-9

现在已经了解了什么是序列图，并用序列图表示用例与对象之间有密切的动态合作关

系。然而，这些动态的对象合作必须落实到对象的类（class）定义里，对象的类定义必须支持序列图所表达的对象交互需要，才算是正确的对象类定义。所以，从序列图的需要可以“检验”出对象的类定义是否能正确支持此用例的需要。

请您牢记，对象是许多用例共享的，一个对象可能参与数个用例，所以在设计某一序列图时，其参与对象的类定义可能早已存在了，而且参与了几个用例。所以您不是从某一个用例（或序列图）而生成新对象类定义，而是从新的用例（或序列图）来更新对象的类定义，使得对象能具有更好更新的功能，旧友（已参与用例）和新知（新参与的用例）皆大欢喜。

所以，您必须知道如何从序列图对应到对象的类定义，才能知道已有的类定义是否足够支持新的用例的需要。如果发现已足够了，就直接使用（Reuse）原来的类定义即可；如果发现不够，就须对类定义进行修改，新陈代谢一番。

21.4.2 对应到类

现在就来介绍序列图与对象类定义的对应关系，依循这些对应关系，能让您厘清对象类定义是否需要修正。如发现现有的类定义都不适用，就必须定义新的类。也可能发现某个类的角色或内涵有错，就必须修正它。不断运用这些对应关系，持续修正、提炼对象，对象就会被愈来愈多的用例共享，其质量就愈来愈稳定、愈来愈棒了。就像固特异轮胎一样，越多人使用，其质量自然愈好！

21.4.3 对应到函数

首先，请您注意图 21-10 上面的对象，每个对象都必须对应到其类定义，包括逻辑上对应到 UML 类图（Class Diagram）里的类，以及程序实现上对应到编程语言（如 OOPC）的类定义。图 21-10 表明序列图里的对象对应到 OOPC 代码中的类声明。

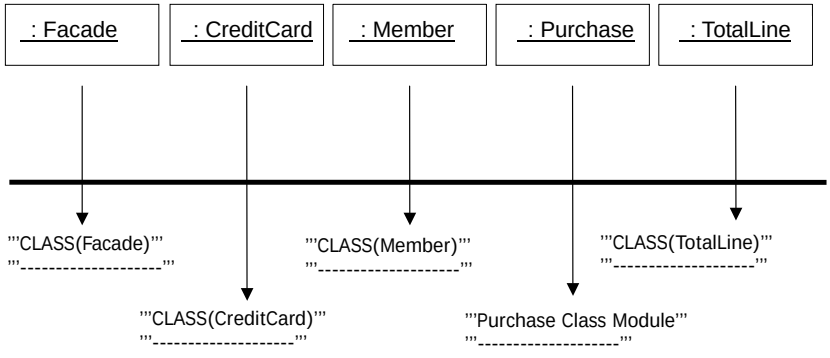


图 21-10

另外，请您注意序列图里的信息，信息的箭头指向某个对象的活动期符号（矩形），意

味着对象接到一个信息时，该对象就会去进行一项运算服务，也就是会去执行类里的一个函数（Function）。所以，当对象接收一个信息时，会执行一个运算，因而对应到代码中的函数声明，如图 21-11 所示。

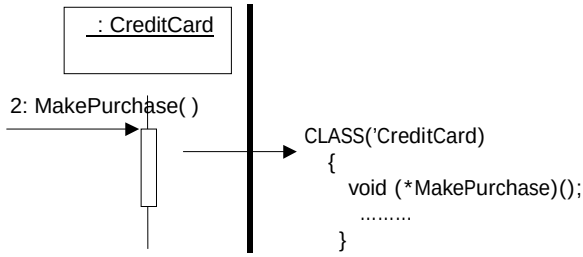


图 21-11

此信息 MakePurchase 是传递给 CreditCard 对象的，信息箭头指向一个矩形，代表此对象会执行类里的 MakePurchase()函数，如果 CreditCard 类里尚无此函数，就应该在类定义里添加此函数，否则 CreditCard 对象将无法处理该信息，此用例就无法顺利完成了。如果 CreditCard 类里已经有此函数，就直接使用而不必添加函数了。

如果对象的生命线上有两个矩形，而且信息名称不相同（即接受两个不同的信息），就必须使两个矩形各对应到一个函数，如图 21-12 所示。

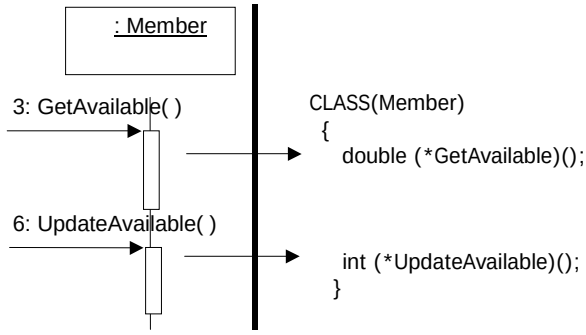


图 21-12

依此类推，对象的生命线上也可能有多个矩形且信息名称不相同（即接受多个不同的信息），各对应到一个函数，如图 21-13 所示。

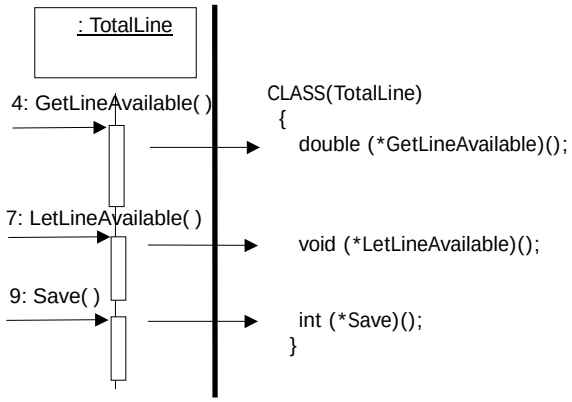


图 21-13

从序列图对应到程序里的类定义，保持序列图里的信息名称与类里的函数名称的一致性，是非常重要的。一般 UML 开发工具（如 EA）所提供的（Round-trip Engineering）机制，都能协助检验以确保这种一致性。如果在您的心中认为流程是不稳定的、善变的，而您必须让它能尽情地变化，就能意会到这种一致性的重要。流程的变化是天生的，您必须去接纳它、顺从它，像古代的大禹治水一般，黄河的善变是必须疏导它而不是去堵住它。所以流程一有变化，序列图必须随之变动，然后从序列图的异动点对应到程序里的类的异动部分。这样就能迅速找出程序的异动点，迅速修正。

以上是针对进入（Incoming）的信息，当对象接到一个信息时，会执行矩形所对应的函数，在执行此函数的过程中，会发出信息给其他对象（即调用其他类的函数），就产生了外传（Outgoing）信息了。

21.4.4 对方对象→信息名称（参数）

在序列图里，矩形的一个外传信息会对应到函数里的一个信息传递指令。如图 21-14 里的 MakePurchase 信息箭头所指向的矩形有三个外传信息。

如果 Facade 类是以 mCreditCard 指针来指向对方对象，则指令会是：

```
cthis->mCreditCard->MakePurchase(…..)
```

刚才已经提过，这里的对象参考名称是 CreditCard 类程序员所取的，序列图并未规范它。所以，当看到 obj 参考名称时，必须回头看看 obj 是参考到哪一个类的对象。例如上述的程序例子里，有个指令 Set obj = New Member，就知道 obj 是参考到 Member 对象的，可检验出程序类与序列图并不一致。

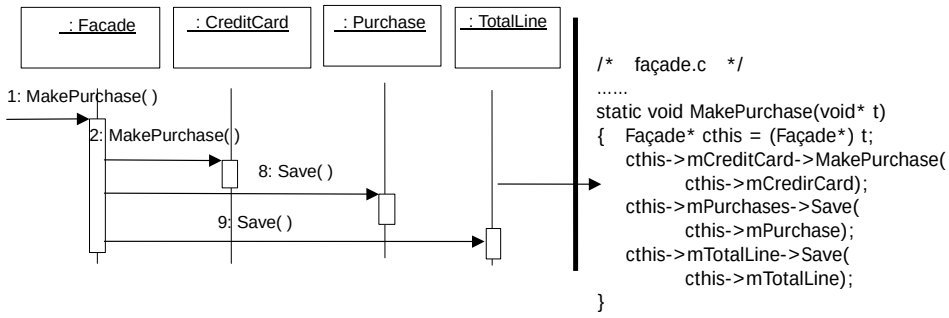


图 21-14

但是请留意，这两个信息分别来自不同的矩形，所以这两个指令必须各自摆在不同的函数里。这两个矩形皆摆在 Member 对象的生命线上，所以这两个函数应该属于 Member 类，而且从矩形左上角的进入（Incoming）信息名称可得知函数的名称分别为 GetAvailable()和 UpdateAvailable()，如图 21-15 右边的代码。

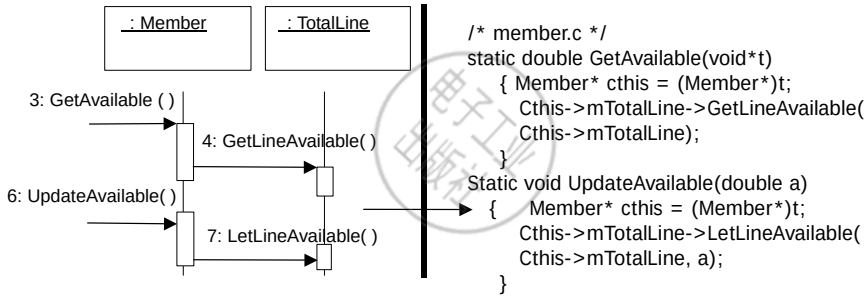


图 21-15

21.5 UML 序列图示例说明

——以 Toggle Light 电灯为例

21.5.1 绘制用例图

在本书第 12 章里，编写过 Toggle Light Controller 软件程序，它设计了两项功能来直接服务用户，如图 21-16 所示。

从 User 的角度来看，整个 Toggle Light 系统需要提供两个用例：

- 使用 Wall Switch 开灯或关灯；
- 使用 Door Switch 开灯或关灯。

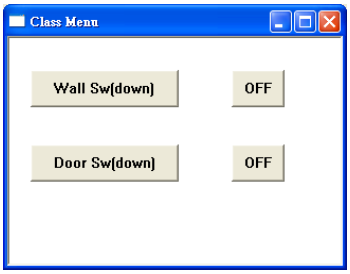


图 21-16

现以 EA 的 Use Case 图表达上述两项功能，如图 21-17 所示。

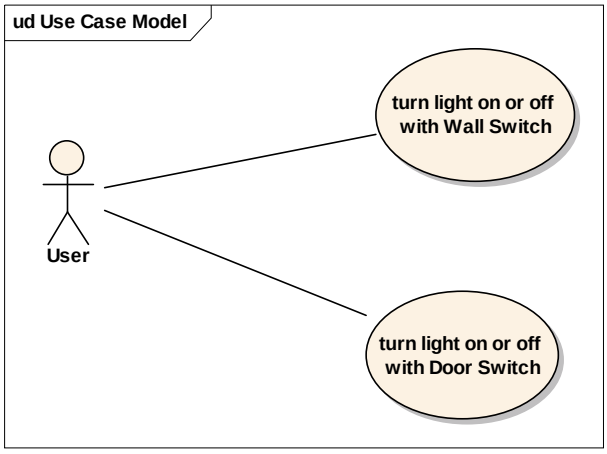


图 21-17

此例子是要开发一个 Toggle Light Controller 软件，现设计三个软件对象（Object）来组成一个 Light Controller。其中，Wall Switch 对象控制 Wall Switch Set 硬件设备；Door Switch 对象控制 Door Switch Set 硬件设备；而 Light 对象控制多个 Light Bulb Set 电灯。

21.5.2 绘制类图

Wall Switch 和 Door Switch 对象的特性和行为是一样的，所以可归为同一个类，取名为 Switch 类。而 Light 对象可归到另一个类，取名为 Light 类。如图 21-18 所示。

这个 Toggle Switch 类用来产生两个 Switch 对象 Wall Switch 和 Door Switch。而 Toggle Light 类用来产生 Light 对象。为什么是一个 Toggle Switch 类产生两个对象呢？而不是设计两个类 Wall Switch 和 Door Switch 各生成一个 Switch 对象呢？这完全是类设计者的专业判断了。一般而言，如果 Wall Switch 和 Door Switch 两个对象的数据属性（Attribute）和行为（Operations）都一样，只需要一个模子就够了，何必要两个类呢？

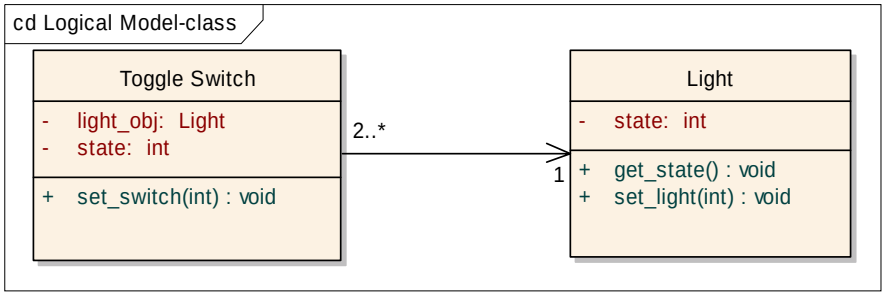


图 21-18

21.5.3 绘制序列图

接着，运用 UML 序列图表达用例背后的对象合作情形。

- 针对 UC-1：turn light on or off with Wall Switch

  其序列图如图 21-19 所示。

- 针对 UC-2：turn light on or off with Door Switch

  其序列图如图 21-20 所示。

用例是系统提供给用户的一项服务，而服务常含有一系列的活动，接受系统一项服务就是接受一项服务的过程，就是用户对系统的一项使用过程，也就是系统内部一群对象的合作交互过程，序列图就表现了这样的过程。

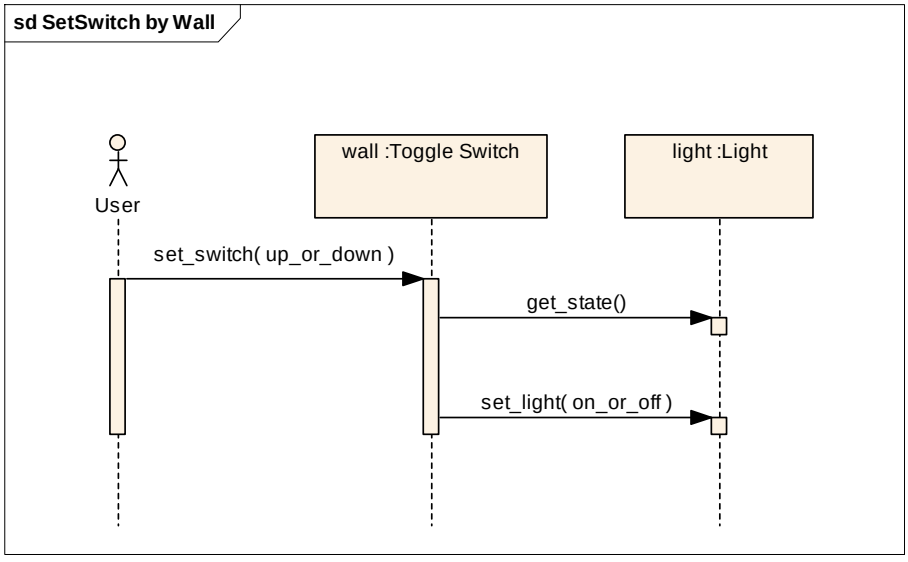


图 21-19

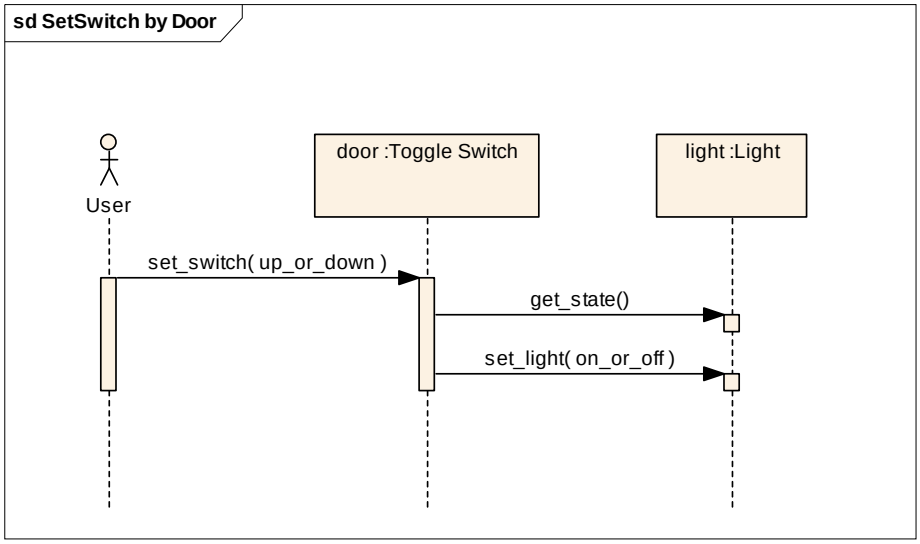


图 21-20

21.5.4 对应到 OOPC 类

序列图里的对象必须对应到类图里的类，如图 21-21 所示。

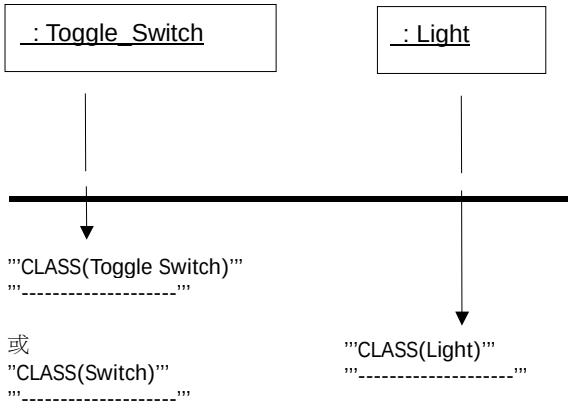


图 21-21

也就是说，序列图的对象也会对应到 OOPC 代码类，例如在第 12 章里就编写了如下的 OOPC 代码：

```
/* cx12-lig.h   */
#ifndef LIGHT_H
#define LIGHT_H
#include "lw_oopc.h"
```

```
CLASS(Light)
{
  int state;
  void (*init)(void*);
  int (*get_state)(void*);
  void (*set_light)(void*, int flag);
};
#endif
```

```
/*   cx12-sw.h   */
#ifndef SWITCH_H
#define SWITCH_H
#include "lw_oopc.h"
#include "cx12-lig.h"

CLASS(Switch)
{
  int state;
  Light* light_obj;
  void (*init)(void*, Light*);
  int (*get_state)(void*);
  void  (*set_switch)(void*);
};
#endif
```

由于 UML 与 OOPC 都基于相同的 OOP 思维，使得 UML 序列图与 OOPC 代码能有完美的对应关系，让 UML 模型设计师与 OOPC 程序员之间有一致的想法和共同的感觉。这对于复杂的软件系统开发，有非常大的帮助。接着，OOPC 程序员又依循图 21-22 所示的对应关系。

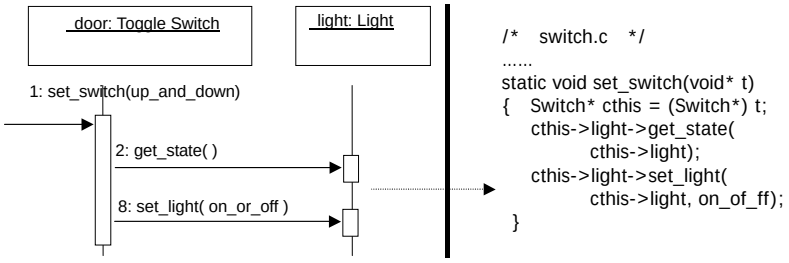


图 21-22

的确，在第 12 章里，OOPC 程序员就编写了如下的代码：

```
/*  cx12-sw.c  */
#include "StdAfx.h"
#include <stdio.h>
#include "cx12-sw.h"

static void init(void *t, Light* light)
{ Switch* cthis = (Switch*) t;
  cthis->light_obj = light;
```

```
  cthis->state = 0;
}

static int get_state(void* t)
{ Switch* cthis = (Switch*) t;
  return cthis->state;
}

static void  set_switch(void* t)
{
  int st;
  Light* light;
  Switch* cthis = (Switch*) t;
  cthis->state = !(cthis->state);
  light = cthis->light_obj;
  st = light->get_state(light);
  if (st == 1)      light->set_light(light, 0);
  else  light->set_light(light, 1);
}

CTOR(Switch)
  FUNCTION_SETTING(init, init);
  FUNCTION_SETTING(get_state, get_state);
  FUNCTION_SETTING(set_switch, set_switch);
END_CTOR
```

这种完美的对应，其实来之不易，需要系统分析员、架构设计师和程序员之间都有一致的面向对象思维才行，希望你的团队能拥有这样的默契。

第 22 章　UML 对象状态图

22.1 Why 状态图

大家都很熟悉在实时（Real-time）和嵌入式系统开发中的状态机（State Machine）。因为状态机能把系统分为多个状态（State），藉由状态将事件（Event）分组，让人们对复杂的事件分而治之，有效避免意外事件的干扰，使系统变得更加稳定，大幅提升了人们对系统的掌控程度。如果你想构建的系统属于控制系统，UML 状态图就可以帮你许多忙。

由于嵌入式或实时系统在跟用户的交互过程中，其行为复杂易变，此时 UML 的状态图是描述这种复杂行为的最佳工具。状态图的主要用途是明确地说明在什么状态下，哪些事件是有效的，哪些是无效的，这统称为事件分组（Partition）。状态图还可让我们能够规划及掌控交互系统的明确行为，以达到准确地控制整个系统的目标。

22.2 简介 UML 状态图

22.2.1 状态、事件与转移

状态图（statechart）可以让我们聚焦在单一种类的对象本身，描述对象一生中可能出现的状态（state）变化。以 Skype 这类的网络通话系统为例，我们设计了用户对象来对应真实世界的用户。针对通话服务，如果我们想要看到用户对象从生成到消失之间全时期的变化状况，这时就可使用状态图来描述用户对象全时期的生命周期，如图 22-1 所示。

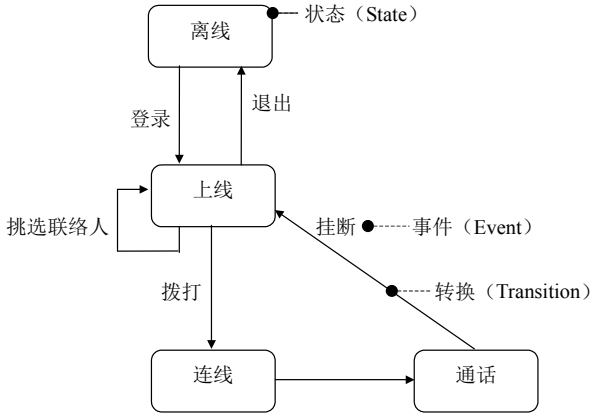


图 22-1

这是用户对象的状态图，简单地将用户对象的一生区分成四个状态，分别为离线、上线、联机和通话。经过不同事件的刺激，会导致用户对象的转移状态。用户对象处在某一个状态中，会对特定的事件有反应，而转移到另一个状态。通过状态图，可以明确且精准地表达出用户对象全时期的行为。

状态图可简单也可复杂，它包括了丰富多样的图示。在后续的各节里，我们会为您逐步介绍。

对象（或系统）从生成到消失的生命期间，会经历一连串的状态转移（Transition）。而且当对象处于每一个特定的状态时，也有其特定的行为模型。一般而言，对象通常不会自动转移状态，必须通过特定事件的发生才会刺激对象转移到下一个状态。因此，我们可以通过状态图来规划某一种对象可以出现的状态，以及促使转移状态的重要事件。同时，针对每一个状态，我们还会设计出对象处于该状态时，可以执行的各项活动（Activity）。这样一来，某种对象一生中的所有状态转变及可以执行的活动，可以清楚明确地跃然于状态图上。如图 22-2 所示。

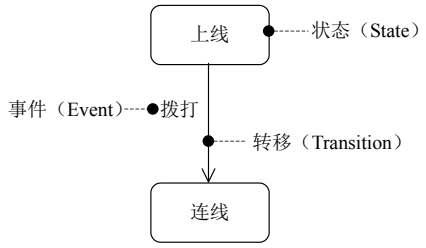


图 22-2

在 UML 里，采用圆角矩形作为状态的图示，以带箭头的实线连接两个状态，并于实线旁标示出刺激转移的事件名称。以图 22-2 为例，一旦发生了拨打事件，处于上线状态的对象会循着箭头方向转移到联机状态。

除了事件之外，转移线旁还可以标示出参数（Parameter）、警戒条件（Guard）及活动（activity），如图 22-3 所示。

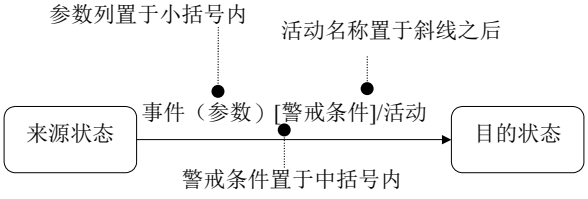


图 22-3

当事件发生时，除了会带进参数外，还必须同时符合转移线旁的警戒条件，才能够转移状态并执行转移线旁的活动。以 Skype 为例，如果我封锁了某位联络人，将无法与他通话。换言之，并不是拨打事件发生就一定会转移状态，如图 22-4 所示。

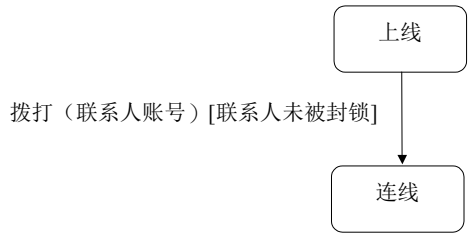


图 22-4

在图 22-4 中，加上了“联络人未被封锁”的警戒条件，表示只有在符合警戒条件的情况下，才会发生状态的转移。如果转移线旁未标示任何事件名称，则意味着该对象执行完状态内所规定的活动之后，会自动转移到下一个状态，如图 22-5 所示。

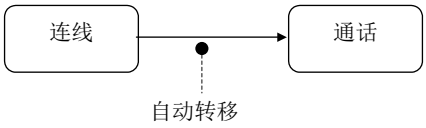


图 22-5

这表示，用户对象在执行完联机状态内的活动之后，可以自动转移到通话状态。再者，对象也可能会有自我转移（Self-transition）的情况发生，意味着对象接收到事件之后，离开现有状态，又再度进入原先的状态。在表示上只要将转移线打个弯连回原状态的圆角矩形图示就行了，如图 22-6 所示。

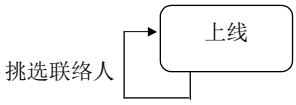


图 22-6

这表示，无论发生了几次挑选联络人的事件，都会转移回到原先的状态。

22.2.2 活动

除了转移瞬间可以执行活动外，对象处于某个状态期间时，也会执行活动。状态内部的活动有两类，一类是预设好、一定会执行的内部活动；另一类是因事件而引发的活动。在状态图示内部可分为三个间隔，分别放置状态名称、一定会执行的活动，以及由事件引发的内部转移，如图 22-7 所示。

图 22-7 中含有三个预设的内部活动，分别为 entry、exit 和 do。请注意，我们在为内部转移标示事件名称时，避免使用这三个关键字，以免造成混淆。顾名思义，entry 意味着对象进入此状态时，必须立即执行且不可被中断的行动。同理，当对象接到事件并打算转移到下

一个状态之际，必须在离开前执行 exit 行动。do 活动则介于 entry 与 exit 之间，对象在执行完 entry 行动之后，就会开始执行 do 活动。对象可能持续执行 do 活动直到发生外部事件而中断，接着执行 exit 行动，而后转移到下一个状态。

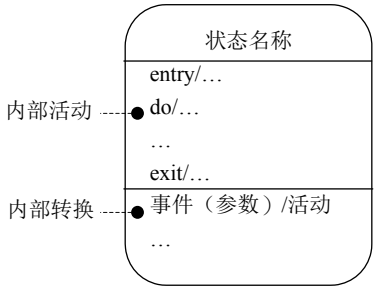


图 22-7

以网络通话为例，用户对象一进入通话状态，就立即执行记录通话起始时间的行动，接着持续执行传送数据包的活动，直到挂断事件发生了，才会结束传送数据包的活动，并且立即执行记录通话结束时间的行动，如图 22-8 所示。

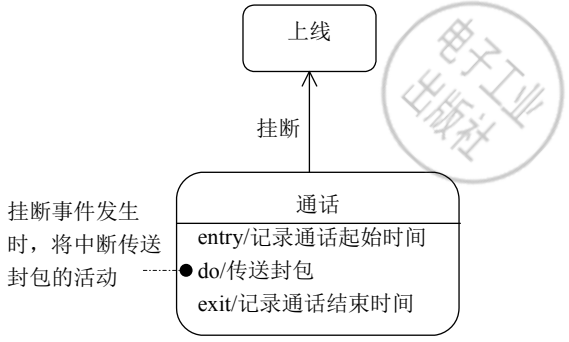


图 22-8

22.2.3　复合状态

当一状态内部含有其他的子状态（Substate）时，我们就称这样的状态为复合状态（Composite State），如图 22-9 里的执行中状态。至于不含子状态的状态，则称为简单状态（Simple State）。

在图 22-9 中，“执行中”复合状态，将联机及通话状态包在里面，成为它的子状态。在复合状态内部，有一个实心圆代表初始状态（Initial State），意味着当进入复合状态内部时，会以初始状态为起点，然后自动转移到联机状态。最后，用户对象会停留在通话子状态中，直到发生挂断或注销事件，才会离开复合状态。

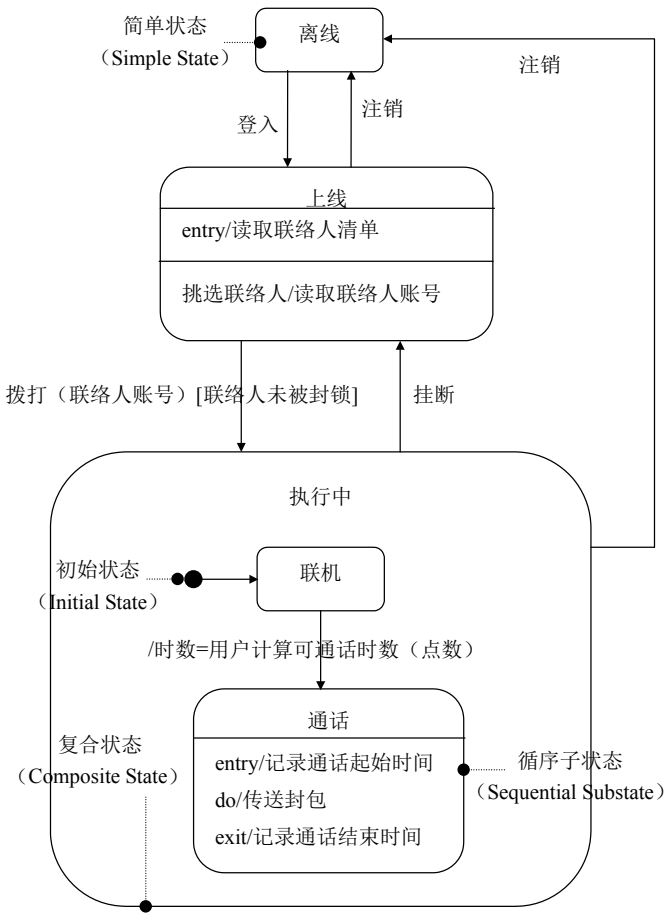


图 22-9

22.2.4 子机状态

在状态图内，还有另一种很好用的子机状态（Submachine State）。当我们想要复用某个状态机设计时，可在状态图内放置子机状态，表示复用某个状态机设计。例如有一个已经设计好的复合状态图，如图 22-10 所示。

在图 22-11 里，就使用子机状态的图示来引用上述既有的状态图，如图 22-11 所示。

换个角度来看，子机状态也是一种复合状态。只不过这个复合状态的设计，呈现在别的状态图中，或许复用我们之前就设计好的状态机，或许引用由别人负责设计的状态机。简单来说，在引用它的这张状态图里，无法得知它的状态设计的细节。

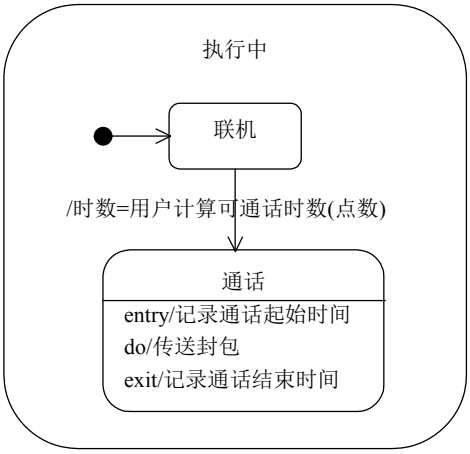


图 22-10

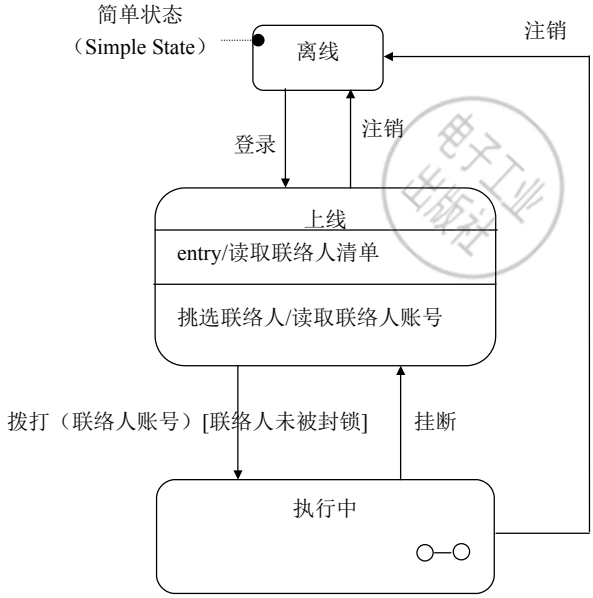


图 22-11

22.2.5 历史状态

历史状态（History State）可以记住并重新进入对象在离开复合状态时的最后一个待过的子状态。有时候，对象会因特殊事件发生而中断正常状态，随后又复原到中断时的状态并继续执行。历史状态在这样的设计中特别好用，如图 22-12 所示。

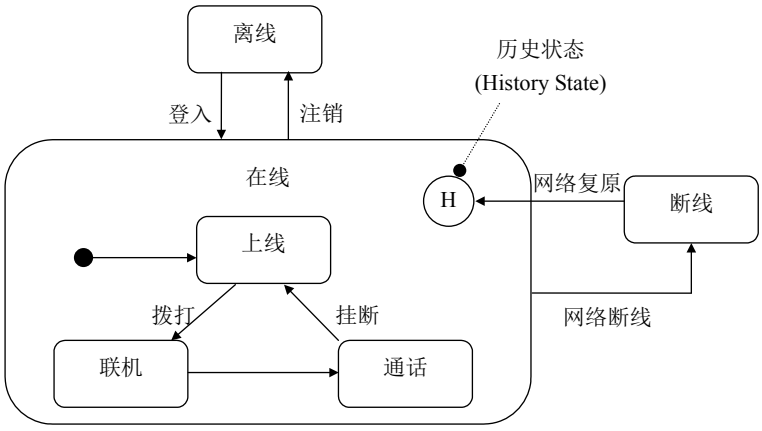


图 22-12

历史状态放置于复合状态内部，它是一个内部标示 H 字眼的圆形。这个设计意味着，对象在在线复合状态的任一子状态期间，可能会突然遭遇网络不正常而断线，因而离开复合状态，并进入断线状态。随后，一旦网络复原，对象会从断线状态再度进入在线复合状态。由于断线状态连接到历史状态，所以对象会获得历史状态的记忆，直接进入先前离开时的子状态中。

对象进入历史状态，就相当于转移进入最近一次的子状态，所有该执行的 entry 行动，一个都不能少。例如，对象是在通话子状态期间突然遭遇网络不正常断线，因此对象会先执行通话子状态里的 exit 行动，记录通话结束时间后，才离开通话子状态。紧接着，对象会开始执行在线复合状态里的 exit 行动，记录在线状态离开时间，最后离开在线复合状态，转移到断线状态。对象从通话子状态转移到断线状态期间，必须依序执行图 22-13 中 1 号及 2 号标示的 exit 行动。

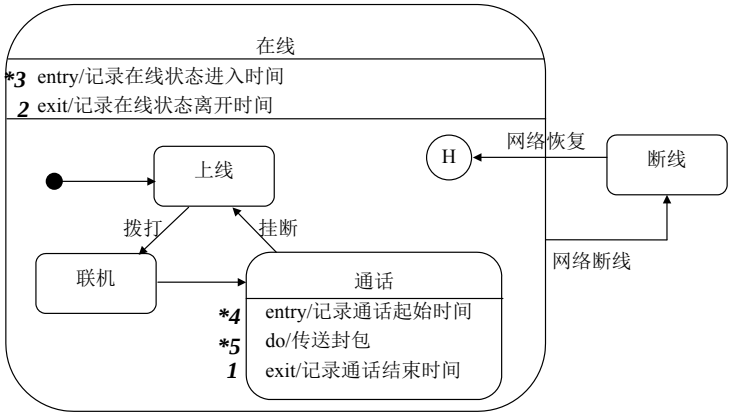


图 22-13

网络恢复之后，对象将从断线状态转入当前离开的通话子状态。首先，对象会先进入在线复合状态，并且执行它的 entry 行动，记录在线状态进入时间。接着，对象才会进入通话

子状态，并依序执行它的 entry 行动，记录通话起始时间，以及它的 do 活动，传送数据包。对象从断线状态转移到通话子状态期间，必须依序执行图 22-13 中*3 号、*4 和*5 号标示的 entry 行动。

22.2.6　决策

决策（Decision）比较简单，我们常常会依据不同条件而决定状态转移的途径，如图 22-14 所示。

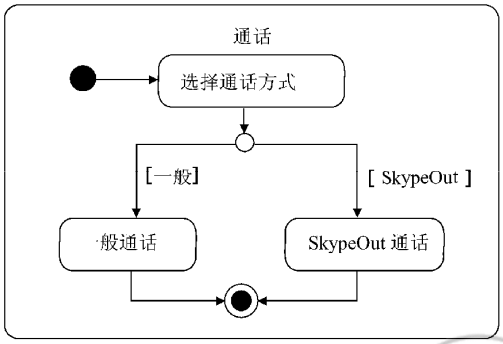


图 22-14

在进入通话状态之后，会根据选择的通话方式而转移到不同状态。

22.2.7　汇合

刚才的决策是一种分岔路，视条件而转移到不同的状态之后，可能汇合（junction）进入同一个状态，如图 22-15 所示。

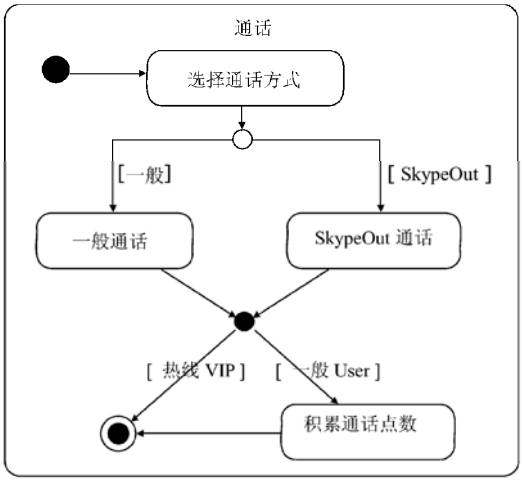


图 22-15

这表示分岔之后，再汇合起来，然后进行另一次的分岔，最后汇合于结束状态。

22.2.8 并行

嵌入式或实时系统的主要特点是，经常同时执行许多个活动（simultaneous activity）。所以系统里会规划许多可并行的线程（thread）来执行这些活动。在 UML 状态图里，可以运用它的并行（concurrent）状态来表示。如将图 22-15 的含义改用下述的并行状态图表示，如图 22-16 所示。

一开始进入通话状态时，就立即进入两个并行的子状态机，各子状态机都会处于单一的状态。一旦要离开通话状态，会执行累计通话点数活动，此时会根据各状态机的状态而决定如何执行累计通话点数活动。

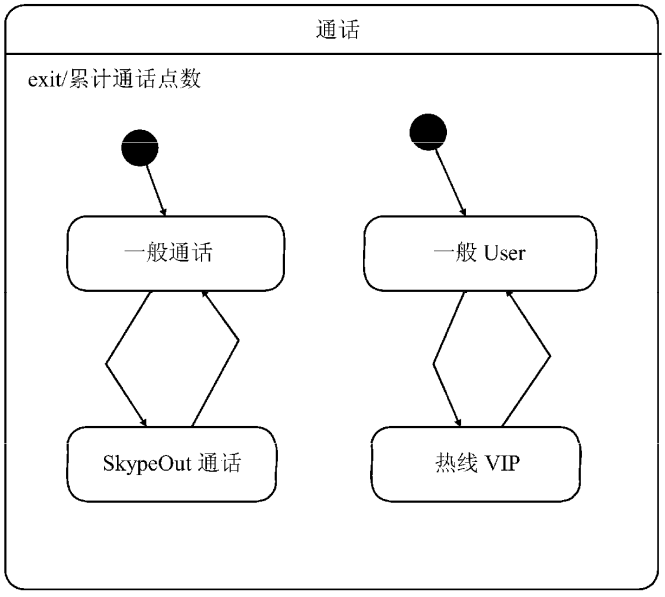


图 22-16

22.2.9 同步

在两个或多个状态并行状态之间，它们有时候会互相协调，以便同时起步或等待大家都完成。这种同步（synchronization）情形在嵌入式和实时系统的设计领域是很常见的，否则硬件模块之间信号传递可能会出错。

现在再来看看 Skype 通话的例子吧，如图 22-17 所示。

通话与录音两个并行状态必须同步才能正确地录下通话的内容，而且两者都必须完成了才结束 Skype 的通话联机。

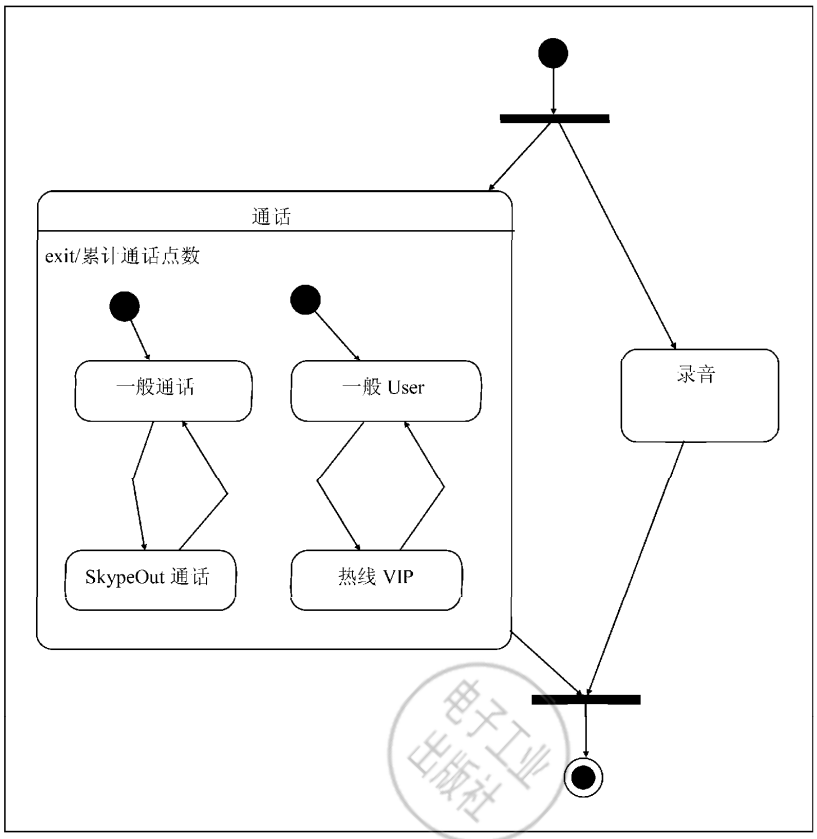


图 22-17

22.3　使用 UML 绘图工具

——以免费的 StarUML 为例

22.3.1　绘图区

UML 是世界标准的建模语言，目前有许多相关的工具可用，其中需要付费的工具方面，以 Enterprise Architect（简称为 EA）最流行，也很便宜。在开放源码方面以 StarUML 和 JUDE 名气最大。本节就以 StarUML 为例介绍其用法。

进入 StarUML 之后，以鼠标右键点击模型浏览器里的 Design Model 之后，在出现的菜单中选取【Add Diagram►Statechart Diagram】，就会出现 StarUML 的状态图绘图区，如图 22-18 所示。

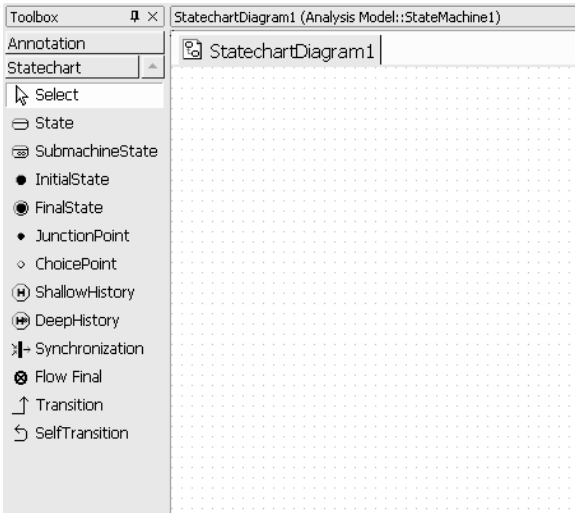


图 22-18

22.3.2 工具箱

在图 22-18 中的左手边就是工具箱，列出了可以放置在状态图面上的相关图示，例如状态、子状态机、状态转移等，详细说明如图 22-19 所示。

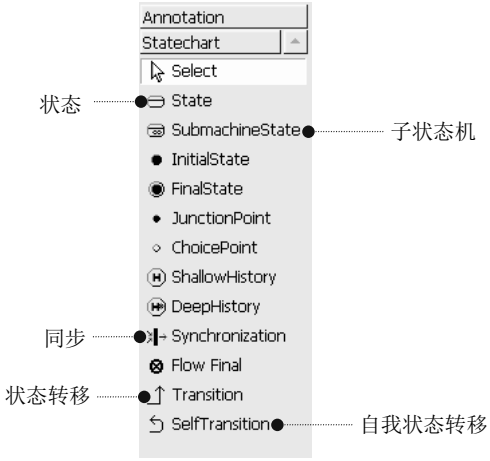


图 22-19

22.3.3 画一个状态

用鼠标左键按下工具箱里的状态按键之后，在状态图的任一空白处，再一次按下鼠标左键，便新增了一个状态，如图 22-20 所示。

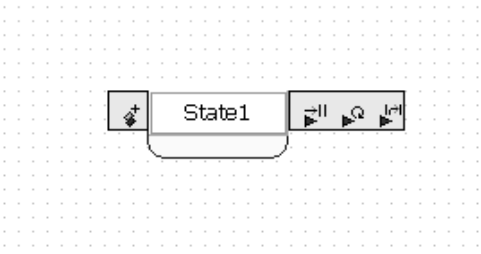


图 22-20

接着，输入状态的名称（如“录音中”）即可。

按照同样的动作，可再画出其他状态（如“已录音”），或者更多的状态。

22.3.4 状态转移

用鼠标左键按下工具箱里的一个状态转移按键，如图 22-21 所示。

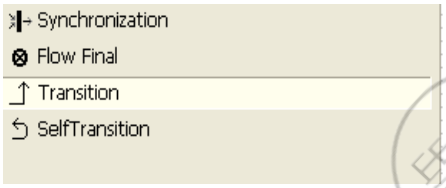


图 22-21

选取状态转移按键之后，在状态图内状态图示处，再一次按下鼠标左键不放，然后拖曳至另一个状态图示处放开，便新增了一个状态转移图示，如图 22-22 所示。

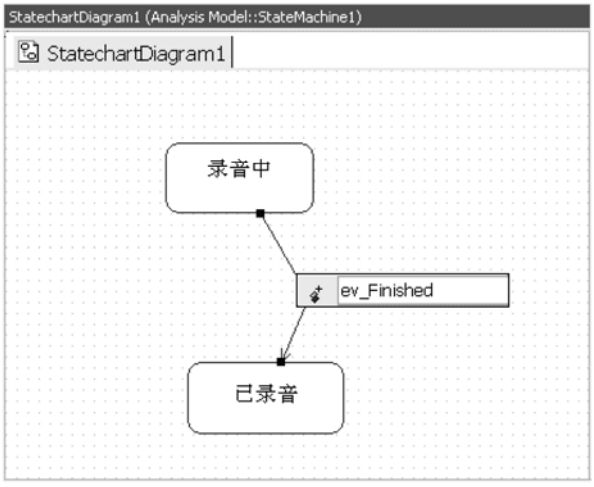


图 22-22

接着，输入状态的名称（如“ev_Finished”）即可。

在状态转移名称的后面，可附加上转移的种类名称，从状态性质表里的 Triggers 选项中，选取该选项即可看到 UML 所定义的四种事件，如下：

- Signal Event — an event due to some external asynchronous process.

  （来自某些外在的异步程序）

- Call Event — an event due to the execution of an operation within the object.

  （来自对象内函数的执行）

- Change Event — an event due to the change in value of an attribute.

  （来自对象属性值的改变）

- Time Event — an event due to the lapse of an interval of time.

  （时间区段已经用完了）

例如，录音时间到了，应按图 22-23 所示执行。

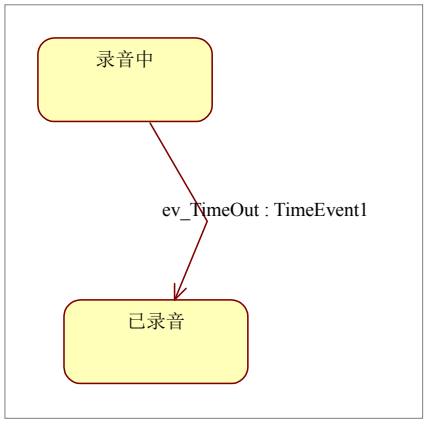


图 22-23

这个 ev_TimeOut 事件可能是由定时器（Timer）发出来的，它促使图中的状态转移。

22.3.5 活动的表示

前面已经介绍过，除了转移瞬间可以执行活动以外，对象处于某个状态期间也会执行活动，包括 entry、exit 和 do 三种活动。顾名思义，entry 意味着对象进入此状态时，必须立即执行且不可被中断的行动。同理，当对象接到事件并打算转移到下一个状态之际，必须在离开前执行 exit 行动。do 活动则介于 entry 与 exit 之间，对象在执行完 entry 行动之后，就开始执行 do 活动。对象可能持续执行 do 活动直到发生外部事件而中断，接着执行 exit 行动，而

后转移到下一个状态。

新增活动的做法是用鼠标光标移至状态图标的名称处，双击鼠标左键会出现新增活动的三个按键符号，如图 22-24 所示。

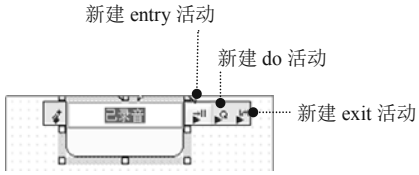


图 22-24

从图 22-24 中可看出，第一个按键符号“->||”用来输入 entry 活动；第二个符号用来输入 do 活动；第三个符号用来输入 exit 活动。点击符号之后，就可以输入活动名称，如图 22-25 所示。

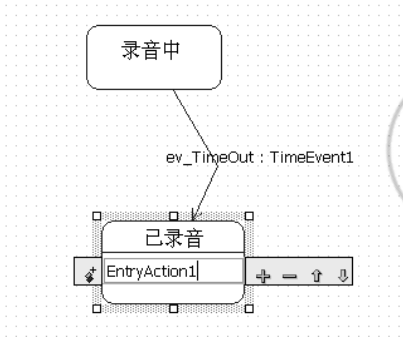


图 22-25

可利用图 22-25 中右边的“加（+）”符号来输入多个活动名称，也可以用“减（–）”符号来删除活动名称。还可以用“上（↑）、下（↓）”符号来上下移动活动名称。至此，您已经学会 StarUML 状态图的基本绘制方法了。基于这些基础方法，您就能更上一层楼，继续学习更多的概念和技术了。

22.4　如何以 OOPC 实现 UML 状态图

22.4.1　举例：以小灯状态为例

以电冰箱为例，冰箱里有个小灯，当冰箱门打开时，它就亮起来；当关上门时，就熄灭了。将冰箱与小灯拟人化（Anthropomorphize），当冰箱知道门被打开时，冰箱的状态就改变，发出 turnOn 信息给小灯。就小灯而言，turnOn 是个事件（Event），这个事件令小灯改变状态

（亮）。当冰箱知道门被关时，冰箱的状态就改变，发出 turnOff 信息给小灯。就小灯而言，turnOff 是个事件（Event），这个事件令小灯改变状态（熄灭）。

turnOn 事件发生时，小灯由 Off 状态转移（Transit）至 On 状态，就亮起来了。turnOff 事件发生时，小灯由 On 状态转移至 Off 状态，就熄灭了。UML 状态图（State Diagram）可表现小灯状态的转移，如图 22-26 所示。

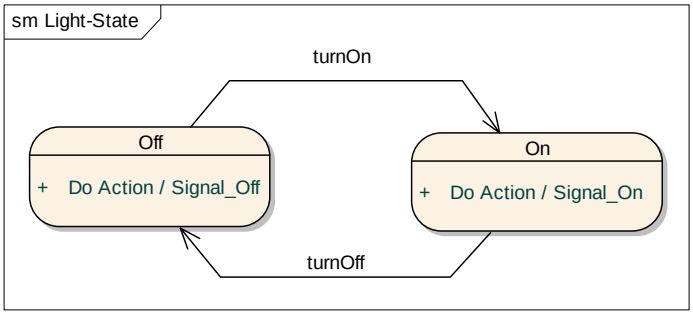


图 22-26

小灯是一个对象，它有图 22-26 所示的两个状态。于是编写 Light 类及其接口的代码如下。

定义 IL 接口及 Light 类

```
/* cx22-lig.h   */
#ifndef LIGHT_H
#define LIGHT_H
#include "lw_oopc.h"

INTERFACE(IL)
{
  void (*init)();
  void (*turnOn)(void*);
  void (*turnOff)(void*);
};

CLASS(Light)
{ IMPLEMENTS(IL);
  char m_state[5];
  void (*go_state_On)(void*);
  void (*go_state_Off)(void*);
};
#endif
```

编写 Light 类代码

```
/* cx22-lig.c  */
#include <stdio.h>
#include "cx22-lig.h"

static void  Signal_On()
{ printf("Light On\n");  }
```

```
static void  Signal_Off()
{ printf("Light Off\n");  }
static void init(void* t)
{ Light* cthis = (Light*)t;
  cthis->go_state_Off(cthis);
}
static void turnOn(void* t)
{ Light* cthis = (Light*) t;
  if(!strcmp(cthis->m_state, "On"))
  {   printf("reject turnOn msg!\n");
      return;
  }
  else
      cthis->go_state_On(cthis);
}
static void turnOff(void* t)
{ Light* cthis = (Light*) t;
  if(!strcmp(cthis->m_state, "Off"))
  {   printf("reject turnOff msg!\n");
      return;   }
  else
      cthis->go_state_Off(cthis);
}
static void  go_state_On(void* t)
{ Light* cthis = (Light*) t;
  strcpy(cthis->m_state, "On");    Signal_On();   }
static void  go_state_Off(void* t)
{ Light* cthis = (Light*) t;
  strcpy(cthis->m_state, "Off");
  Signal_Off();
}
CTOR(Light)
  FUNCTION_SETTING(IL.init, init);
  FUNCTION_SETTING(IL.turnOn, turnOn);
  FUNCTION_SETTING(IL.turnOff, turnOff);
  FUNCTION_SETTING(go_state_On, go_state_On);
  FUNCTION_SETTING(go_state_Off, go_state_Off);
END_CTOR
```

编写主程序

```
/* cx22-ap1.c  */
#include <stdio.h>
#include "cx22-lig.h"

int main()
{ IL* lg = (IL*)LightNew();     lg->init(lg);
  lg->turnOn(lg);     lg->turnOn(lg);    lg->turnOff(lg);
  getchar();
  return 0;
}
```

编写 Turbo-C 的 Project 文件

```
      cx22-lig.c
      cx22-ap1.c
```

执行结果如图 22-27 所示。

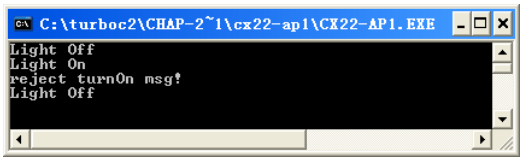


图 22-27

22.4.2 举例：以冰箱的状态为例

请看另一个对象冰箱的状态转移图吧！冰箱有两个状态“开着门”（DoorOpened）与“关着门”（DoorClosed）。打开冰箱门时，事件发生了，冰箱由 DoorClosed 状态转移至 DoorOpened 状态；关起冰箱门时，另一个事件发生了，冰箱由 DoorOpened 状态转移至 DoorClosed 状态。于是冰箱的状态转移图如图 22-28 所示。

打开门（openTheDoor）时，冰箱转移至 DoorOpened 状态。此种状态变化促使冰箱对象产生一项行动——送出 turnOn()信息给小灯，于是小灯转移至 On 状态，灯就亮了。关起门（closeTheDoor）时，冰箱转移到 DoorClosed 状态，此时冰箱产生一项行动——送出 turnOff()信息给小灯，于是小灯转移至 Off 状态，灯熄灭了。从冰箱的状态转移图，可了解到：

- 冰箱接受两种信息——openTheDoor()及 closeTheDoor()。

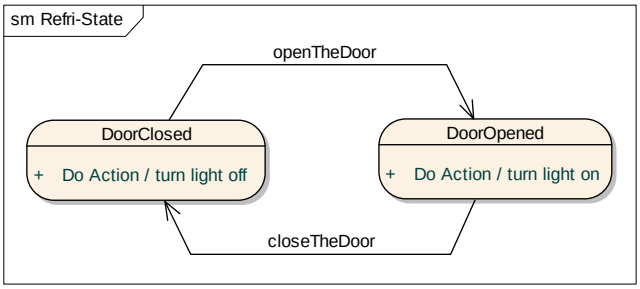


图 22-28

- 当冰箱处于 DoorClosed 状态下时，会接受 openTheDoor()信息，但不接受 closeTheDoor()。
- 当冰箱处于 DoorOpened 状态下时，会接受 closeTheDoor()信息，但不接受 openTheDoor()信息。
- 一旦进入 DoorOpened 状态，必然送出 turnOn()信息给小灯；一旦进入 DoorClosed 状态，就送出 turnOff() 信息给小灯。

接着，从小灯的状态图，可了解到：

- 小灯可接受两种信息—— turnOn()及 turnOff() 。
- 小灯处于 Off 状态下，接受 turnOn()信息，但不接受 turnOff() 。

- 小灯处于 On 状态下，接受 turnOff() 信息，但不接受 turnOn()。

冰箱与小灯都是对象，也各有图 22-28 所示的两个状态。上一节已经编写了 Light 类及其 IL 接口的代码，现在新增一个冰箱（Refri）类的步骤如下：

定义 IR 接口及 Refri 类

```
/* cx22-ref.h   */
#ifndef REFRI_H
#define REFRI_H
#include "lw_oopc.h"

INTERFACE(IR)
{ void (*init)();
  void (*openTheDoor)(void*);
  void (*closeTheDoor)(void*);
};

CLASS(Refri)
{ IMPLEMENTS(IR);
  char m_state[15];
  IL* light;
  void (*go_state_DoorOpened)(void*);
  void (*go_state_DoorClosed)(void*);
};
#endif
```

编写 Refri 类代码

```
/* cx22-ref.c  */
#include <stdio.h>
#include "cx22-lig.h"
#include "cx22-ref.h"

static void init(void* t)
{ Refri* cthis = (Refri*)t;   cthis->light = (IL*)LightNew();
  cthis->go_state_DoorClosed(cthis);
}
static void openTheDoor(void* t)
{
  Refri* cthis = (Refri*) t;
  if(!strcmp(cthis->m_state, "DoorOpened"))
   {   printf("reject openTheDoor msg!\n");    return;  }
  else  cthis->go_state_DoorOpened(cthis);
}
static void closeTheDoor(void* t)
{ Refri* cthis = (Refri*) t;
  if(!strcmp(cthis->m_state, "DoorClosed"))
  {   printf("reject closeTheDoor msg!\n");
     return;   }
  else   cthis->go_state_DoorClosed(cthis);
}
static void  go_state_DoorOpened(void* t)
{ Refri* cthis = (Refri*) t;
  strcpy(cthis->m_state, "DoorOpened");
  cthis->light->turnOn(cthis->light);
}
static void  go_state_DoorClosed(void* t)
```

```
{ Refri* cthis = (Refri*) t;
  strcpy(cthis->m_state, "DoorClosed");
  cthis->light->turnOff(cthis->light);
}
CTOR(Refri)
  FUNCTION_SETTING(IR.init, init);
  FUNCTION_SETTING(IR.openTheDoor, openTheDoor);
  FUNCTION_SETTING(IR.closeTheDoor, closeTheDoor);
  FUNCTION_SETTING(go_state_DoorOpened, go_state_DoorOpened);
  FUNCTION_SETTING(go_state_DoorClosed, go_state_DoorClosed);
END_CTOR
```

编写主程序

```
/* cx22-ap2.c  */
#include <stdio.h>
#include "cx22-lig.h"
#include "cx22-ref.h"

int main()
{
  IR* refri = (IR*)RefriNew();    refri->init(refri);
  refri->openTheDoor(refri);      refri->closeTheDoor(refri);
  refri->openTheDoor(refri);      refri->openTheDoor(refri);
  getchar();
  return 0;
}
```

编写 Turbo C 的 Project 文件

```
cx22-lig.c
cx22-ref.c
cx22-ap2.c
```

执行结果如图 22-29 所示。

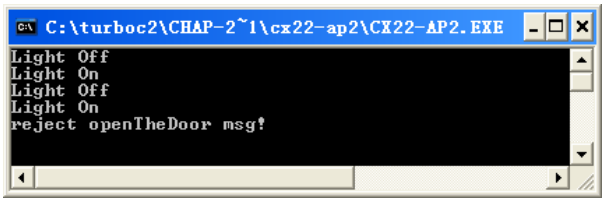


图 22-29

22.4.3 举例：以银行账户（Account）的状态为例

状态与行为

外来的刺激，内心的变化，改变了个人的行为。例如，爱人离去，痴情人心碎且行为改变了（如吃香蕉皮）！外来的信息，状态的变化，改变了个体的行为。例如，四季交替，花儿开了又谢，枫叶由绿而红，再随风飘落，皆是个体的行为。行为的交互作用，构成多姿多

彩的大自然。

当个体状态改变时，所引发的行为可分为两大类：

- 状态变化时，立即产生特殊行为。

例如，银行账户的状态随金额的变化而改变，一旦处于透支状态，就会立即显示出信息来。

- 状态转移时，并不立即产生特殊行为，但却影响后续的行为。例如，人的心理状态。

立即行为

个体处于不同状态，常立即产生不同的行为。灯泡烧坏了，立即“不亮”了。候选人当选了，立即鸣谢赐票。银行账户个体处于透支状态时，立即输出信息提醒用户——您已向银行贷款了。如图 22-30 所示。

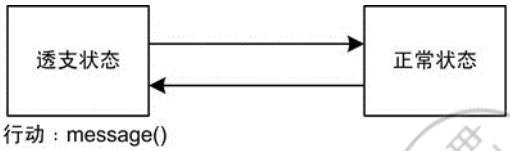


图 22-30

用户提款时，如果 acc 个体处于透支状态，就立即调用 message()函数显示出信息，告知用户共欠银行多少钱。如果处于正常状态，就不会输出信息。必要时，可明确记载 Account 的个体状态。请你看看如何以 OOPC 来实现这个状态图：

```
/* cx22-ap3.c  */
#include <stdio.h>
#include "lw_oopc.h"

INTERFACE(IA)
{
  void (*init)(void*);
  void (*deposit)(void*, double);
  void (*withdraw)(void*, double);
};

CLASS(Account)
{ IMPLEMENTS(IA);
  double amount;
  char state;
  void (*message)(void*);
};

static void init(void* t)
{ Account* cthis = (Account*)t;
  cthis->amount = 0;
  cthis->state='N';
}
```

```
static void deposit(void*t, double a)
{ Account* cthis = (Account*)t;
  cthis->amount += a;
  if(cthis->amount > 0 )   cthis->state='N';
}
static void withdraw(void* t, double a)
{ Account* cthis = (Account*)t;
  cthis->amount = cthis->amount - a;
  if(cthis->amount < 0 )
  {  cthis->state='O';    cthis->message(cthis);    }
}
static void message(void*t)
{ Account* cthis = (Account*)t; printf("You owe the bank $%6.2f\n", -1 *
cthis->amount);   }
CTOR(Account)
  FUNCTION_SETTING(IA.init, init);
  FUNCTION_SETTING(IA.deposit, deposit);
  FUNCTION_SETTING(IA.withdraw, withdraw);
  FUNCTION_SETTING(message, message);
END_CTOR

void main()
{
  Account* acc = (Account*)AccountNew();
  acc->init(acc);
  acc->deposit(acc, 100);     acc->withdraw(acc, 150);
  getchar();    return 0;
}
```

此程序输出 You owe the bank $50。

state 变量记载着个体处于哪个状态。透支时 state 值为'O'，表示处于透支状态，正常状态时 state 值为'N'。每回提款时，若 acc 个体处于透支状态，就会有特殊动作——调用 message()函数。所以 message()是透支状态的行动，每次取款时，若处于透支状态，就调用 message()函数。

另一种常见情形是：只有当 acc 个体从正常状态“转移”到透支状态时，才进行 message()动作，如图 22-31 所示。

图 22-31

这意味着原来已处于透支状态，取款后仍处于透支状态，但并不调用 message() 函数。从“透支状态”转移到“正常状态”时，也不调用 message() 。只有当由“正常状态”变换到“透支状态”时才调用 message()。此情形用 C++ 程序表示如下：

```
/* cx22-ap4.c  */
#include <stdio.h>
#include "lw_oopc.h"

INTERFACE(IA)
{
  void (*init)(void*);
  void (*deposit)(void*, double);
  void (*withdraw)(void*, double);
};

CLASS(Account)
{
  IMPLEMENTS(IA);
  double amount;
  char state;
  void (*change_state)(void*);
  void (*message)(void*);
};

static void init(void* t)
{ Account* cthis = (Account*)t;    cthis->amount = 0;
  cthis->state='N';  }
static void deposit(void*t, double a)
{  Account* cthis = (Account*)t;    cthis->amount += a;
   cthis->change_state(cthis);   }
static void withdraw(void* t, double a)
{  Account* cthis = (Account*)t;    cthis->amount -= a;
   cthis->change_state(cthis);   }
static void change_state(void*t)
{ Account* cthis = (Account*)t;
  if(cthis->state == 'N' )
     if( cthis->amount < 0 )
        { cthis->state = 'O';  cthis->message(cthis);   }
     else
        if( cthis->amount >= 0 )  cthis->state = 'N';
}
static void message(void*t)
{  Account* cthis = (Account*)t;   printf("Over(%6.2f)\n",
   cthis->amount);  }
CTOR(Account)
  FUNCTION_SETTING(IA.init, init);
  FUNCTION_SETTING(IA.deposit, deposit);
  FUNCTION_SETTING(IA.withdraw, withdraw);
  FUNCTION_SETTING(change_state, change_state);
  FUNCTION_SETTING(message, message);
END_CTOR

void main()
{
  Account* acc = (Account*)AccountNew();   acc->init(acc);
  acc->deposit(acc, 100);    acc->withdraw(acc, 150);
  acc->withdraw(acc, 30);
  getchar();    return 0;
}
```

此程序输出 Over（-50）。

当 state 值由'N' 变为'O'时，才调用 message() 函数。如果金额一直小于 0，则 acc 个体在透支状态的停留时间可能很长。然而，一般银行账户大多不允许透支，此时 acc 个体短暂处于透支状态，送出信息后，由于未改变金额，所以立即返回正常状态。此情形，用 OOPC 实现如下：

```
/* cx22-ap5.c  */
#include <stdio.h>
#include "lw_oopc.h"

INTERFACE(IA)
{
  void (*init)(void*);
  void (*deposit)(void*, double);
  void (*withdraw)(void*, double);
};

CLASS(Account)
{
  IMPLEMENTS(IA);
  double amount;
  char state;
  void (*state_action)(void*);
  void (*message_O)(void*);
};

static void init(void* t)
       { Account* cthis = (Account*)t;  cthis->amount = 0;
cthis->state='N';    }
static void deposit(void*t, double a)
{ Account* cthis = (Account*)t; cthis->amount += a;    }
static void withdraw(void* t, double a)
       { Account* cthis = (Account*)t;
  if( (cthis->amount - a) < 0 )
      {  cthis->state='O';  cthis->state_action(cthis);
         cthis->state = 'N'; }
  else
      { cthis->amount -= a;   cthis->state_action(cthis);   }
      }
static void message_O(void*t)
       { Account* cthis = (Account*)t;   printf("Over(%6.2f)\n",
cthis->amount);   }
static void message_N()
       { printf("Welcome\n");   }
static void state_action(void*t)
       { Account* cthis = (Account*)t;
  switch( cthis->state )
      {
         case 'O':   cthis->message_O(cthis);   break;
         case 'N':   message_N();    break;
      };
}
CTOR(Account)
  FUNCTION_SETTING(IA.init, init);
  FUNCTION_SETTING(IA.deposit, deposit);
  FUNCTION_SETTING(IA.withdraw, withdraw);
  FUNCTION_SETTING(state_action, state_action);
  FUNCTION_SETTING(message_O, message_O);
```

```
END_CTOR

void main()
{ Account* acc = (Account*)AccountNew();    acc->init(acc);
  acc->deposit(acc, 100.5);    acc->withdraw(acc, 50.5);
  acc->withdraw(acc, 50.6);
  getchar();   return 0;
}
```

此程序输出：

```
Welcome
Over( 50.5 )
```

提款时，当欲取金额大于余款时，acc 个体送出信息，但未改变金额。

后续行为

在上节中银行账户的状态随金额大小而改变，一旦转移到透支状态，立即产生特殊行为——执行 message()。另一种常见情形是：状态转移时并不立即产生特殊行为，但却影响后续的行为。例如，到银行取款时，常须输入密码，如果密码正确，就打开账户，可以取款。如果密码错了，则无法取款。因此，账户可分为两个状态，如图 22-32 所示。

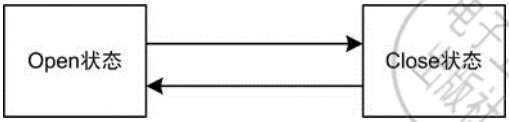


图 22-32

当账户处于 Open 状态，且接到您的信息——“取 200 元”时，账户就给您 200 元钞票。反之，若处于 Close 状态，账户个体就不理您了。用 OOPC 程序表达如下：

```
/* cx22-ap6.c  */
#include <stdio.h>
#include "lw_oopc.h"

INTERFACE(IA)
{
  void (*init)(void*);
  void (*open)(void*);
  void (*close)(void*);
  void (*withdraw)(void*, double);
};

CLASS(Account)
{
  IMPLEMENTS(IA);
  double amount;
  int state;
  int psw;
};

static void init(void* t)
```

```
       { Account* cthis = (Account*)t;  cthis->amount = 1000.525;
         cthis->state = 0;    }
static void open(void*t)
       { Account* cthis = (Account*)t;    cthis->state = 1;   }
static void close(void*t)
       { Account* cthis = (Account*)t;    cthis->state = 0;    }
static void withdraw(void* t, double a)
       { Account* cthis = (Account*)t;
  if( cthis->state == 1 )
     {  printf("welcome.\n");
        cthis->amount -= a;
     }
  else
     printf("account is not open!\n");
}
CTOR(Account)
  FUNCTION_SETTING(IA.init, init);
  FUNCTION_SETTING(IA.open, open);
  FUNCTION_SETTING(IA.close, close);
  FUNCTION_SETTING(IA.withdraw, withdraw);
END_CTOR

void main()
{
  IA* acc = (IA*)AccountNew();  acc->init(acc);
  acc->withdraw(acc, 1.0);      acc->open(acc);
  acc->withdraw(acc, 200.0);    acc->close(acc);
  acc->withdraw(acc, 500.0);
  getchar();
  return 0;
}
```

此程序输出：

```
Account is not open!
Welcome.
Account is not open!
```

信息 open()，其行为会影响 acc 个体的 withdraw()行为。state 值为 1，表示已打开；若 state 值为 0，则表示账户关闭中。

第 23 章　UML+OOPC 实用示例之一

——以涂鸦（Scribble）程序开发为例

——使用 Win32/VC++编译环境

23.1 形形色色的涂鸦程序

涂鸦是一种有趣的玩意儿，也有许多用途，包括幼儿园学生的作业本、军队传达命令、生动活泼的教学，等等。因此，涂鸦程序有许多花样，有的可以配上图片，例如 BugMe! NotePad 上的涂鸦，如图 23-1 所示。

图 23-1

可以在底图上做涂鸦。再如儿童艺术 KinderArt 公司的涂鸦（*http://www.kinderart.com/drawing/scribble.shtml*），如图 23-2 所示。它既可以画线条也可以着色。

图 23-2

还有一些涂鸦可以配上录音，例如高焕堂的《用例入门与实例》一书所附的“用例涂鸦秀 CD”，就是一套既配底图又配同步录音的涂鸦秀程序，而且将它嵌入到微软的 PowerPoint

里，只要您播放该书所附的涂鸦秀 CD，就可以欣赏到美妙的涂鸦情境了，如图 23-3 所示。

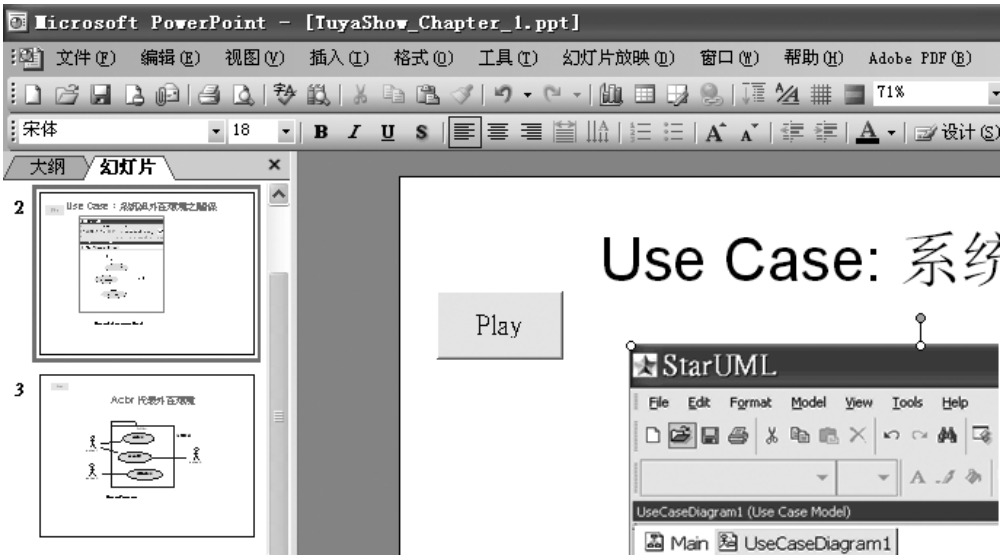


图 23-3

23.2 涂鸦程序示例说明

在本章里，将要设计一个涂鸦程序。其中因为篇幅有限，为了让代码精简易懂，对此示例做了一些需求限制：

- 只画线条，提供黑、红、蓝、绿四种颜色给涂鸦者挑选；
- 不配底图，也不配声音；
- 一边涂鸦，一边录制涂鸦的过程；
- 可以重复播放所录制的涂鸦情境；
- 可将录制的内容存入（save）涂鸦文件里；
- 可从涂鸦文件内容加载（load）到计算机，并重复播放。

例如，你可以如图 23-4 所示进行涂鸦。

只要按下正方形图标按钮，在画线条的过程中，线条的端点和时间会自动存放到内存里，一直到按下休止符按钮为止，你能随时重新播放（按三角图标按钮）。因为轨迹及时间都被存录起来，回放的时候会精准地依据原来绘图的速度和顺序而重新画一次。此外，还可以将所绘的图案存入文件中（按<Save>选项），之后可随时从文件中读取（按<Load>选项）绘图内容，并回放（按三角图标按钮）出来。

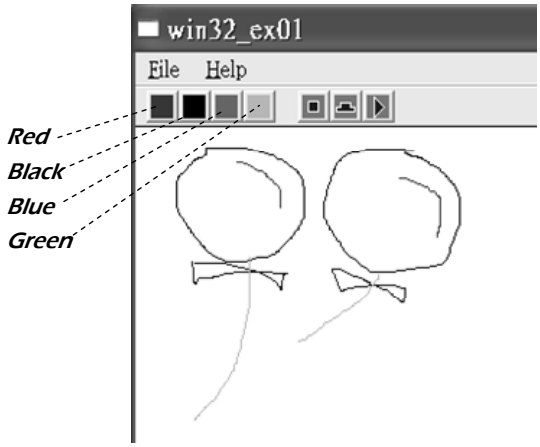


图 23-4

即使是幼儿，也能轻易彩绘各种图样（如生日卡片），画完并存盘之后，可以将文件寄给远方的亲戚朋友。当朋友接到文件时，可以利用图 23-5 所示的<Load>按钮来读取文件里的图，然后按三角图标按钮，就能看到远方传来的祝福了。

图 23-5

23.3 涂鸦系统分析与设计

23.3.1 绘制系统用例（Use Case）图

因为用例图是开发者与用户智能相遇的地方，开发者经常利用<<include>>或<<extend>>关系来表达系统特色，创造对用户的吸引力。如图 23-6 所示，将“涂鸦程序”的特色都凸显出来了。

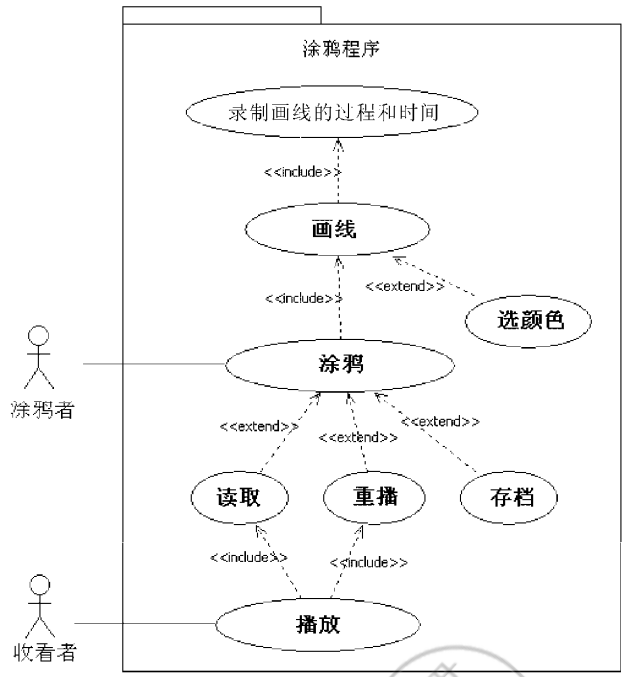


图 23-6

编写主要的用例叙述（UCD）如下：

“UC:涂鸦”的 UCD

用例名称	涂鸦
目的	在计算机上绘制涂鸦式的图片
系统名称	涂鸦程序
Client 种类	人
主要程序	
user action	**system response**
启动涂鸦程序	预设颜色为黑色
开始涂鸦 按下<Begin>按钮 3.2 开始以鼠标画线	4. 绘图与录制 4.1 绘出线条于屏幕窗口里 4.2 录制线条端点、时间和过程
结束 4.1 选取<Exit>选项	5. 关闭窗口
替代或例外程序	
3.2a –画线过程中可以挑选新颜色 4.1a –选取<Exit>之前，可以按下<Draw>回放绘图，或选取 <Save>按钮将涂鸦存盘。	

“UC:播放”的 UCD

用例名称	播放
目的	在计算机上播放涂鸦秀
系统名称	涂鸦程序
Client 种类	人

主要程序：

user action	**system response**
启动涂鸦程序	
开启涂鸦文件 2.1 按下<Load>按钮	读取涂鸦文件
开始回放 4.1 选取<Draw>选项	回放涂鸦
结束 6.1 选取<Exit>选项	7. 关闭窗口

替代或例外程序
无

此例说明了系统的特色，包括：

- 一边涂鸦，一边录制涂鸦过程与时间；
- 可以重复播放涂鸦；
- 可将涂鸦内容存入文件；
- 可从文件载入（load）涂鸦并播放之。

23.3.2 绘制类图

从上述的用例叙述里，可发现数个主要的概念或术语，例如：涂鸦内容、线条端点等，以及 Scribble 和 dwPoint 两个主要类，其中 Scribble 对象代表一张涂鸦，dwPoint 对象代表轨迹（由一连串的小线段所构成）上小线段的端点。再加上负责图形显示的 Painter 类，就形成了如图 23-7 所示的类图。

其中，dwPoint 类 m_x 和 m_y 记载轨迹上小线段的端点坐标，m_type 代表提笔中（即不画出移动轨迹）或下笔中（即会画出移动轨迹）。m_color 代表目前所挑选的颜色。m_timeSpan 代表从第一点到此点的时间差距。Scribble 类会定义一个 poList 串行来储存一系列的端点。

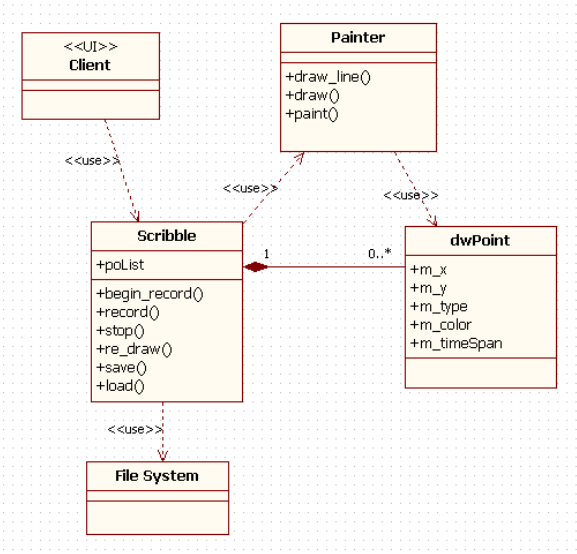


图 23-7

23.3.3　绘制 Scribble 状态图

Scribble 类也扮演着控制的角色，如果觉得 Scribble 的行为有些复杂，可考虑绘制它的 SysML 的状态图，如图 23-8 所示。

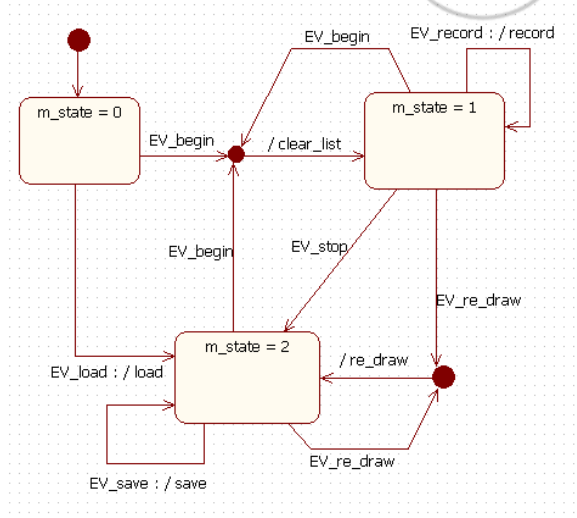


图 23-8

这个状态图说明了：

- Scribble 一开始进入状态 0；

- 当进入状态 1，接到 EV_begin 事件时，就会清除掉（clear）原有的涂鸦动作，并进入状态 1；若接到 EV_load 事件，就执行载入（load）操作，然后进入状态 2；
- 在状态 1，若接到 EV_stop 事件，就进入状态 2；若接到 EV_re_draw 事件，就执行回放（re_draw）的活动，进入状态 2；
- 在状态 2，若接到 EV_re_draw 事件，就再执行回放（re_draw）的活动，然后维持状态 2；若接到 EV_begin 事件，就会清除掉（clear）原有的涂鸦动作，并返回状态 1；若接到 EV_save 事件，就会将目前涂鸦内容存入文件，维持状态 2。

依据这些，Scribble 的行为就非常清晰了。

23.3.4 绘制序列图

序列图是用例图与类图相遇的地方，其表达如何将 Scribble、dwPoint 等对象组合出各用例图里的各项服务。从上述用例图里，可看到两个主要用例“涂鸦”和“播放”。现在就来看看如何用序列图表达这两个用例。

23.3.5 用例：“涂鸦”

由于页面宽度的限制，可将之切分为两个序列图，即图 23-9 和图 23-10。从图 23-9 中可

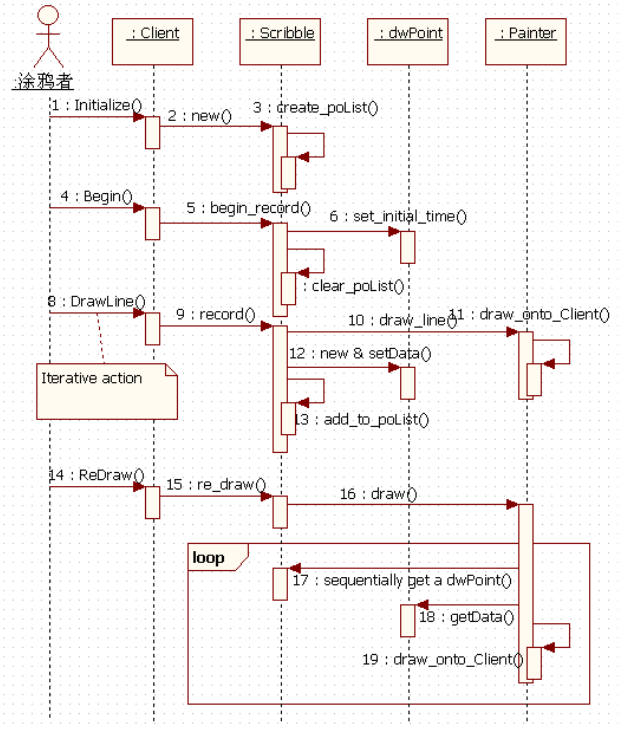


图 23-9

以看出，Scribble 自己不做画图的动作，而委托 Painter 来做。Scribble 将 poList 串行参数传给 Painter 并且请 Painter 将串行里的每一个 dwPoint 绘至屏幕画面上。但是从图 23-10 中可以看出，Scribble 亲自与 File System 沟通以保存（save）及载入（load）。

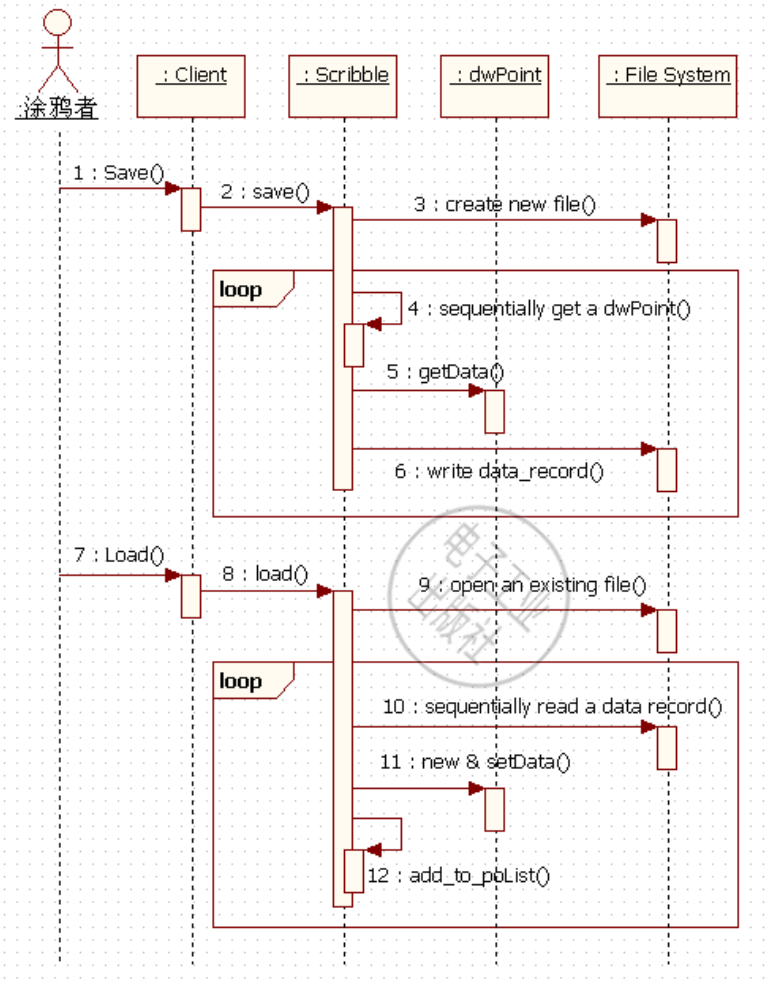


图 23-10

23.3.6　用例：“播放”

设计序列图，如图 23-11 所示。

其实，只是复用前面序列图（即图 23-7 和图 23-8）已经描述过的 load()、re_draw()等功能而已。

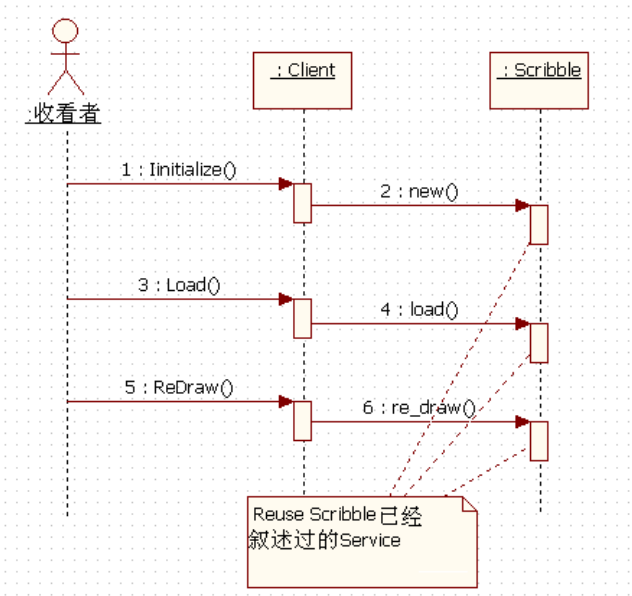


图 23-11

23.4 涂鸦程序的实现：使用 OOPC 语言

基于上述的类图和序列图，可以编写 Win32 平台上的 C 代码，其步骤如下。

Step-1 创建一个 Win32 项目 Win32_ex002。

其内容如图 23-12 所示。

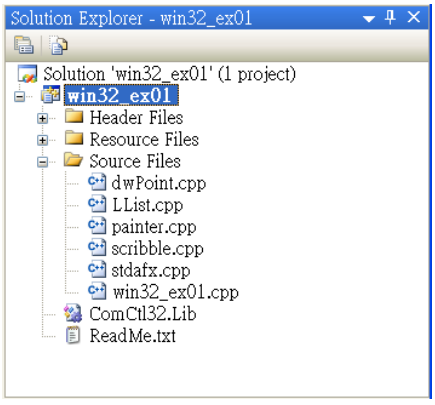


图 23-12

这里展现的是在上一节里，类图（即图 23-7）包含的三个主要类：Scribble、dwPoint 和 Painter，其分别对应到 scribble.cpp、dwPoint.cpp 和 painter.cpp 代码文件。

Step-2 以 LW_OOPC 实现主要类。

这里列出完整的代码。请看 dwPoint 类的内容如下。

定义 dwPoint.h 头文件

dwPoint 类担任记录图点的工作，每一个 dwPoint 对象代表一条线的端点。

```
/* dwPoint.h   */
#ifndef DWPOINT_H
#define DWPOINT_H
#include "lw_oopc.h"

CLASS(dwPoint)
{ int m_x, m_y,  int m_type;
  long m_color, long m_timeSpan;
  void (*init)(void*, int, int, int, long);
  void (*set_initial_time)(void*, long);
};
#endif
```

实现 dwPoint.c 代码

```
/* dwPoint.c   */
#include "stdafx.h"
#include "dwPoint.h"

static long initial_time;
/*  记录点位置和时间   */
static void init(void* t, int x, int y, int ty, long color)
{ dwPoint* cthis = (dwPoint*)t;
  cthis->m_x = x;   cthis->m_y = y;
  cthis->m_type = ty;     cthis->m_color = color;
  cthis->m_timeSpan = GetTickCount() - initial_time;
}
static void set_initial_time(void *t, long iTime)
{ initial_time = iTime;}
CTOR(dwPoint)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(set_initial_time, set_initial_time)
END_CTOR
```

定义 painter.h 头文件

painter 类负责将线条绘至屏幕上。

```
/* painter.h   */
#ifndef PAINTER_H
#define PAINTER_H
#include "StdAfx.h"
#include "lw_oopc.h"
#include "llist.h"
//#pragma once

CLASS(Painter)
```

```
{ INTERFACE(IColl)* m_pList;
  void (*init)(void*);
  void (*draw_line)(void*, int, int, int, int, long);
  void (*draw)(void*, INTERFACE(IColl)* );
 };
#endif
```

实现 painter.c 类

```
/*   painter.cpp   */
//   Utility Class
#include "StdAfx.h"
#include "lw_oopc.h"
#include "llist.h"
#include "dwPoint.h"
#include "painter.h"

static HDC hdc;
static HPEN hpen, holdPen;
static HBRUSH hbrush, holdBrush;
extern HWND hWindow;

static void init(void *t) {}
/*  画线   */
static void draw_line(void *t, int prevX, int prevY, int x, int y, long color)
{ hdc = GetDC(hWindow);
hpen = CreatePen(PS_SOLID, 1, (COLORREF)color); //BLUE
  holdPen = (HPEN)SelectObject(hdc, hpen);
  MoveToEx(hdc, prevX, prevY, (LPPOINT) NULL);
  LineTo(hdc, x, y);   ReleaseDC(hWindow, hdc);
}
/*   画出所有的点  */
static void draw(void *t, INTERFACE(IColl)* list)
{ DWORD curr_time, base_time, draw_time;
  int lastX, lastY, x, y;  int n=0;
  hdc = GetDC(hWindow);
  list->top(list);
  dwPoint* point;
  point = (dwPoint*)list->next(list);
  while(point != NULL)
  {   x = point->m_x;    y = point->m_y;
      hpen = CreatePen(PS_SOLID, 1,(COLORREF)point->m_color);
      holdPen = (HPEN)SelectObject(hdc, hpen);
      draw_time = point->m_timeSpan;
      if(n == 0)    base_time = GetTickCount();
      n++;
      //---------   waiting ----------------------------------
      do {   curr_time =  GetTickCount() - base_time;     }
      while (curr_time < draw_time);
      //-----------------------------------------------------
      if (point->m_type == 0)    { lastX = x; lastY = y;  }
      else {   MoveToEx(hdc, lastX, lastY, (LPPOINT) NULL);
               LineTo(hdc, x, y);   lastX = x;  lastY = y;
           }
      point = (dwPoint*)list->next(list);
   }
  ReleaseDC(hWindow, hdc);
}
CTOR(Painter)
```

```
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(draw_line, draw_line)
  FUNCTION_SETTING(draw, draw)
END_CTOR
```

定义 llist.h 头文件

llist 类实现 LinkedList 数据结构，用来记录所有的线条端点。

```
/*  llist.h   */
#ifndef LLIST_H
#define LLIST_H
#include "lw_oopc.h"

INTERFACE(IColl)
{ void (*init)(void*);
  void (*clear)(void*);
  void (*add)(void*, void*);
  void (*top)(void*);
  void* (*next)(void*);
  void* (*get)(void*, int);
  int (*is_empty)(void*);
};
#endif
```

实现 llist.c 代码


```
/* llist.cpp   */
#include "stdafx.h"
#include "stdio.h"
#include "llist.h"

CLASS(ListNode)
{ void* pItem;
  ListNode* next;
};
CTOR(ListNode)
END_CTOR

/* ---------------------------------  */
CLASS(LList)
{
  IMPLEMENTS(IColl);
  ListNode *head, *tail, *current;
};
static void init(void* t)
{ LList* cthis = (LList*)t;   cthis->head = NULL;
  cthis->tail = NULL;     cthis->current = NULL;
}
static void clear(void* t)
{ LList* cthis = (LList*)t;    ListNode *px, *py;    px = cthis->head;
  while(px != NULL)
  {   py = px;         px = px->next;
      free(py->pItem);     free(py);
   }
  cthis->head = NULL;   cthis->tail = NULL;    cthis->current = NULL;
}
static void add(void* t, void* pi)
```

```
{ LList* cthis = (LList*) t;
  ListNode* pn = (ListNode*)ListNodeNew();
  pn->next = NULL;   pn->pItem = pi;
  if(cthis->head == NULL)
    {  cthis->tail = pn;   cthis->head = pn;   cthis->current = pn;   }
  else
    {  cthis->tail->next = pn;   cthis->tail = pn;   cthis->current = pn;  }
}
static void top(void* t)
{ LList* cthis = (LList*) t;   cthis->current = NULL;  }


static void* next(void* t)
{ LList* cthis = (LList*) t;
  if(cthis->current == NULL)
  {   if(cthis->head == NULL) return NULL;
      else   { cthis->current = cthis->head;  return cthis->current->pItem; }
  }
   else
  {   cthis->current = cthis->current->next;
      if(cthis->current == NULL) return NULL;
      else return cthis->current->pItem;
  }
}
static void* get(void* t, int k)
{ LList* cthis = (LList*) t;   int i;   ListNode* pn;
  if(cthis->head == NULL) return NULL;
  pn = cthis->head;
  for(i=0; i<k; i++)
  {   pn = pn->next;
      if(pn == NULL) return NULL;   }
  return pn->pItem;
}
static int is_empty(void* t)
{ LList* cthis = (LList*) t;
  if(cthis->head == NULL) return 1;
  else return 0;
 }
CTOR(LList);
  FUNCTION_SETTING(IColl.init, init)
  FUNCTION_SETTING(IColl.clear, clear)
  FUNCTION_SETTING(IColl.add, add)
  FUNCTION_SETTING(IColl.top, top)
  FUNCTION_SETTING(IColl.next, next)
  FUNCTION_SETTING(IColl.get, get)
  FUNCTION_SETTING(IColl.is_empty, is_empty);
END_CTOR
```

定义 scribble.h 头文件

scribble 类是涂鸦系统的 controller，负责协调 painter、LinSave 等对象。

```
/*  scribble.h   */
#ifndef CTRL_H
#define CTRL_H
#include "lw_oopc.h"
#include "llist.h"
#include "painter.h"
```

```
CLASS(Scribble)
{
  IColl* poList;
  Painter* pa;
  int m_state;
  void (*init)(void*);
  void (*begin_record)(void* );
  void (*record)(void *t, int, int, int, long);
  void (*stop)(void*);
  void (*re_draw)(void*);
  void (*save_drawing)(void*);
  void (*load_drawing)(void*);
};
#endif
```

实现 scribble.c 代码

```
/*  scribble.c    */
#include "StdAfx.h"
#include "stdio.h"
#include "llist.h"
#include "dwPoint.h"
#include "painter.h"
#include "scribble.h"

extern void* LListNew();
extern void* dwPointNew();
extern void* PainterNew();

/*    ---------------------------------    */
static void init(void *t)
{ Scribble* cthis = (Scribble*)t;
  cthis->poList = (INTERFACE(IColl)*) LListNew();
  (cthis->poList)->init(cthis->poList);
  cthis->pa = (Painter*)PainterNew();
  (cthis->pa)->init(cthis->pa);
}
/*  创建Linked List，准备记录绘图  */
static void begin_record(void *t)
{ Scribble* cthis = (Scribble*)t;
  IColl* list = cthis->poList;   dwPoint* po;   cthis->m_state = 1;
  po = (dwPoint*)dwPointNew();
  po->set_initial_time(po,(long)GetTickCount());  free(po);
  if( ! list->is_empty(list))   list->clear(list);
}
/*  开始绘图  */
static void record(void *t, int x, int y, int ty, long color)
{ Scribble* cthis = (Scribble*)t;
  INTERFACE(IColl)* list = cthis->poList;
  dwPoint* po;
  if(cthis->m_state != 1)   return;
  po = (dwPoint*)dwPointNew();   po->init(po, x, y, ty, color);
  list->add(list, po);
}
/*  结束绘图  */
static void stop(void* t)
{ Scribble* cthis = (Scribble*)t;
  if(cthis->m_state != 1) return;
```

```
  else cthis->m_state = 2;
}
/*  画出先前的录图  */
static void re_draw(void* t)
{ Scribble* cthis = (Scribble*)t;  Painter* pa = cthis->pa;
  if(cthis->m_state != 1 && cthis->m_state != 2) return;
  cthis->m_state = 2;
  pa->draw(pa, cthis->poList);
}
/*   将 Linked List 的录图数据存盘   */
static void save_drawing(void* t)
{ Scribble* cthis = (Scribble*)t;
  IColl* list = cthis->poList;
  struct dwpo
  {   long time_span;
      int x, y;  char t, cc;
  };
  struct dwpo po;     dwPoint* point;     FILE *fd;
  fd = fopen("c:\\scribble.dat","wb");
  list->top(list);
  point = (dwPoint*)list->next(list);
  while(point != NULL)
  {   po.x = point->m_x;
      po.y = point->m_y;
      po.t = (char)point->m_type;
      if(point->m_color == (long)RGB(255,0,0)) po.cc = 0;
      else if(point->m_color == (long)RGB(0,255,0)) po.cc = 1;
      else if(point->m_color == (long)RGB(0,0,255)) po.cc = 2;
      else if(point->m_color == (long)RGB(0,0,0)) po.cc = 3;
      po.time_span = point->m_timeSpan;
      fwrite(&po, sizeof(struct dwpo), 1, fd);
      point = (dwPoint*)list->next(list);
  }
  fclose(fd);
}
  /*  从文件中把绘图数据加载到 Linked List 里   */
static void load_drawing(void* t)
{ Scribble* cthis = (Scribble*)t;
  IColl* list = cthis->poList;
  struct dwpo
  {   long time_span;
      int x, y; char t, cc;
  };
  struct dwpo po;       long col;   FILE *fd;
  dwPoint* new_point;
  fd = fopen("c:\\scribble.dat","rb");
  list->clear(list);
  while(1)
  {   fread(&po, sizeof(struct dwpo), 1, fd);
      if( feof(fd) )     break;
      else
         {   if(po.cc == 0) col = RGB(255,0,0);
             else if(po.cc == 1) col = RGB(0,255,0);
             else if(po.cc == 2) col = RGB(0,0, 255);
             else if(po.cc == 3) col = RGB(0,0,0);
             new_point = (dwPoint*)dwPointNew();
             new_point->m_x = po.x;
             new_point->m_y = po.y;
             new_point->m_type = (int)po.t;
```

```
            new_point->m_color = col;
            new_point->m_timeSpan = po.time_span;   list->add(list,
            new_point);   }
        }
      fclose(fd);
      cthis->m_state = 2; /* Stop  */
} /* Save */
CTOR(Scribble)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(begin_record, begin_record)
  FUNCTION_SETTING(record, record)
  FUNCTION_SETTING(stop, stop)
  FUNCTION_SETTING(re_draw, re_draw)
  FUNCTION_SETTING(save_drawing, save_drawing)
  FUNCTION_SETTING(load_drawing, load_drawing)
END_CTOR
```

Step-3 用 VC++实现 UI 画面。

实现 win32_ex01.cpp 代码

```
// win32_ex01.cpp : Defines the entry point for the application.
#include "stdafx.h"
#include "windows.h"
#include "tchar.h"
#include <commctrl.h>
#include "painter.h"
#include "scribble.h"
#include "win32_ex01.h"

#define MAX_LOADSTRING 100
extern void* ScribbleNew();
extern void* PainterNew();
Scribble* scr;   Painter* pa;
/*  Global Variables:  */
HINSTANCE hInst;                        // current instance
TCHAR szTitle[MAX_LOADSTRING];          // The title bar text
TCHAR szWindowClass[MAX_LOADSTRING]; // the main window class name
HWND hWindow;
/*   ------------------------------------   */
int mState;  long m_Color;   int lastX, lastY;
HWND hWndToolbar;   BOOL bMouseDown;   POINTS ptsEnd;
static char g_szClassName[] = "MyWindowClass";
/*  Forward declarations of functions included in this code module:  */
ATOM    MyRegisterClass(HINSTANCE hInstance);
BOOL    InitInstance(HINSTANCE, int);
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK    About(HWND, UINT, WPARAM, LPARAM);
int APIENTRY _tWinMain(HINSTANCE hInstance,  HINSTANCE hPrevInstance,
                       LPTSTR   lpCmdLine, int  nCmdShow)
{
  UNREFERENCED_PARAMETER(hPrevInstance);
  UNREFERENCED_PARAMETER(lpCmdLine);
  //-----------------------------------------------------------------
  MSG msg;
  HACCEL hAccelTable;

  // Initialize global strings
  LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
```

```
  LoadString(hInstance, IDC_WIN32_EX01, szWindowClass, MAX_LOADSTRING);
  MyRegisterClass(hInstance);

  // Perform application initialization:
  if (!InitInstance (hInstance, nCmdShow))  return FALSE;
  hAccelTable = LoadAccelerators(hInstance,
                                 MAKEINTRESOURCE(IDC_WIN32_EX01));
  // Main message loop:
  while (GetMessage(&msg, NULL, 0, 0))
  {   if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
      {   TranslateMessage(&msg);   DispatchMessage(&msg);    }
  }
  return (int) msg.wParam;
}
ATOM MyRegisterClass(HINSTANCE hInstance)
{
  WNDCLASSEX wcex;
  wcex.cbSize = sizeof(WNDCLASSEX);
  wcex.style          = CS_HREDRAW | CS_VREDRAW;
  wcex.lpfnWndProc    = WndProc;
  wcex.cbClsExtra       = 0;      wcex.cbWndExtra       = 0;
  wcex.hInstance       = hInstance;
  wcex.hIcon          = LoadIcon(hInstance,
  MAKEINTRESOURCE(IDI_WIN32_EX01));
  wcex.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wcex.hbrBackground    = (HBRUSH)(COLOR_WINDOW+1);
  wcex.lpszMenuName    = MAKEINTRESOURCE(IDC_WIN32_EX01);
  wcex.lpszClassName    = szWindowClass;
  wcex.hIconSm       = LoadIcon(wcex.hInstance,
  MAKEINTRESOURCE(IDI_SMALL));
  return RegisterClassEx(&wcex);
}
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{ hInst = hInstance; // Store instance handle in our global variable
  hWindow = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL,
                         hInstance, NULL);
  if (!hWindow)  return FALSE;
  INITCOMMONCONTROLSEX InitCtrlEx;
  InitCtrlEx.dwSize = sizeof(INITCOMMONCONTROLSEX);
  InitCtrlEx.dwICC  = ICC_BAR_CLASSES;
  InitCommonControlsEx(&InitCtrlEx);
  TBBUTTON tbrButtons[10];
  tbrButtons[0].iBitmap   = 0;     tbrButtons[0].idCommand = 0;
  tbrButtons[0].fsState   = TBSTATE_ENABLED;
  tbrButtons[0].fsStyle   = TBSTYLE_SEP;
  tbrButtons[0].dwData    = 0L;  tbrButtons[0].iString   = 0;

  tbrButtons[1].iBitmap   = 0;
  tbrButtons[1].idCommand = CM_RED; //IDM_FILE_NEW;
  tbrButtons[1].fsState   = TBSTATE_ENABLED;
  tbrButtons[1].fsStyle   = TBSTYLE_BUTTON;
  tbrButtons[1].dwData    = 0L;   tbrButtons[1].iBitmap   = 0;
  tbrButtons[1].iString   = 0;

  tbrButtons[2].iBitmap   = 1;
  tbrButtons[2].idCommand = CM_BLACK; //IDM_FILE_OPEN;
  tbrButtons[2].fsState   = TBSTATE_ENABLED;
  tbrButtons[2].fsStyle   = TBSTYLE_BUTTON;
```

```
  tbrButtons[2].dwData   = 0L;   tbrButtons[2].iString   = 0;

  tbrButtons[3].iBitmap   = 2;
  tbrButtons[3].idCommand = CM_BLUE; //IDM_ARROW;
  tbrButtons[3].fsState   = TBSTATE_ENABLED;
  tbrButtons[3].fsStyle   = TBSTYLE_BUTTON;
  tbrButtons[3].dwData    = 0L;  tbrButtons[3].iString   = 0;

  tbrButtons[4].iBitmap   = 3;
  tbrButtons[4].idCommand = CM_GREEN; //IDM_DRAW_LINE;
  tbrButtons[4].fsState   = TBSTATE_ENABLED;
  tbrButtons[4].fsStyle   = TBSTYLE_BUTTON;
  tbrButtons[4].dwData    = 0L;  tbrButtons[4].iString   = 0;

  tbrButtons[5].iBitmap   = 0;    tbrButtons[5].idCommand = 0;
  tbrButtons[5].fsState   = TBSTATE_ENABLED;
  tbrButtons[5].fsStyle   = TBSTYLE_SEP;
  tbrButtons[5].dwData    = 0L;  tbrButtons[5].iString   = 0;

  tbrButtons[6].iBitmap   = 0;    tbrButtons[6].idCommand = 0;
  tbrButtons[6].fsState   = TBSTATE_ENABLED;
  tbrButtons[6].fsStyle   = TBSTYLE_SEP;
  tbrButtons[6].dwData    = 0L;   tbrButtons[6].iString   = 0;

  tbrButtons[7].iBitmap   = 4;
  tbrButtons[7].idCommand = IDM_CLEAR_AND_BEGIN;//CM_YELLOW_SQUARE;
//IDM_FILE_OPEN;
  tbrButtons[7].fsState   = TBSTATE_ENABLED;
  tbrButtons[7].fsStyle   = TBSTYLE_BUTTON;
  tbrButtons[7].dwData    = 0L;  tbrButtons[7].iString   = 0;
  tbrButtons[8].iBitmap   = 5;
  tbrButtons[8].idCommand = IDM_STOP; //CM_BLUE_CIRCLE; //IDM_ARROW;
  tbrButtons[8].fsState   = TBSTATE_ENABLED;
  tbrButtons[8].fsStyle   = TBSTYLE_BUTTON;
  tbrButtons[8].dwData    = 0L;    tbrButtons[8].iString   = 0;

  tbrButtons[9].iBitmap   = 6;
  tbrButtons[9].idCommand = IDM_STOP_AND_DRAW; //CM_GREEN_SQUARE;
//IDM_DRAW_LINE;
  tbrButtons[9].fsState   = TBSTATE_ENABLED;
  tbrButtons[9].fsStyle   = TBSTYLE_BUTTON;
  tbrButtons[9].dwData    = 0L;    tbrButtons[9].iString   = 0;
  hWndToolbar = CreateToolbarEx(hWindow, WS_VISIBLE | WS_CHILD | WS_BORDER,
                                IDB_BITMAP2, 10, //NUMBUTTONS,
                hInst, IDB_BITMAP2, tbrButtons, 10, //NUMBUTTONS,
                           16, 16, 16, 16, sizeof(TBBUTTON));
                ShowWindow(hWindow, nCmdShow);
  UpdateWindow(hWindow);
  return TRUE;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{ int wmId, wmEvent;    PAINTSTRUCT ps;    HDC hdc;    int ret;
  switch (message)
  {   case WM_CREATE:
         scr = (Scribble*)ScribbleNew();    scr->init(scr);
         pa = (Painter*)PainterNew();       pa->init(pa);
         mState = 0;    break;
     case WM_COMMAND:
```

```
        wmId    = LOWORD(wParam);wmEvent = HIWORD(wParam);
        switch (wmId)
        {   case CM_RED:  m_Color = (long)RGB(255,0, 0);   break;
            case CM_BLUE:   m_Color = (long)RGB(0,0, 255);   break;
            case CM_GREEN:  m_Color = (long)RGB(0,255, 0);   break;
            case CM_BLACK:  m_Color = (long)RGB(0,0, 0);    break;
            case IDM_CLEAR_AND_BEGIN:
            if(mState == 0 )
            {   mState = 1;   scr->begin_record(scr);  }
            else  //mState != 0
            {   ret = MessageBox(NULL,_T("Clear current drawing?"),_T(""),
                                 MB_YESNO | MB_ICONINFORMATION);
                if(ret == 6) // Yes
                {   InvalidateRect(hWindow, NULL, TRUE);
                    mState = 1;   scr->begin_record(scr);
                }
                else   { // do nothing    }
            }
            break;
            case IDM_STOP:
                if(mState == 0)  {
                    MessageBox(NULL,_T("No drawing to stop!"),_T(""), MB_OK |
                               MB_ICONINFORMATION);
                    return 0;    }
                if(mState == 2)  {
                    MessageBox(NULL,_T("Already Stoped!"),_T(""), MB_OK |
                               MB_ICONINFORMATION);
                    return 0;
                }
                mState = 2;
                break;
            case IDM_STOP_AND_DRAW:
                if(mState != 1 && mState != 2)
                {   MessageBox(NULL,_T("Nothing to draw!"),_T(""),MB_OK|
                               MB_ICONINFORMATION);
                    return 0;
                }
                scr->re_draw(scr);
                break;
            case ID_FILE_SAVE:  scr->save_drawing(scr);break;
            case ID_FILE_LOAD:  scr->load_drawing(scr); mState = 2;  break;
            case ID_FILE_CLEAR: InvalidateRect(hWindow, NULL, TRUE); break;
            case IDM_ABOUT:
                DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
                break;
            case IDM_EXIT:   DestroyWindow(hWindow);   break;
            default:  return DefWindowProc(hWnd, message, wParam, lParam);
        }
        break;
    case WM_LBUTTONUP:   bMouseDown = false ;    break;
    case WM_RBUTTONDOWN:    break;
    case WM_LBUTTONDOWN:   bMouseDown = true ;
        ptsEnd = MAKEPOINTS(lParam);  //按下 mouse 左键
        scr->record(scr, ptsEnd.x, ptsEnd.y, 0, m_Color);  /*  记录下笔点   */
        lastX = ptsEnd.x;
        lastY = ptsEnd.y;
        break;
    case WM_MOUSEMOVE:   /*  记录移动中途点   */
        if (wParam & MK_LBUTTON)
```

```
        {   if (bMouseDown )
            {   ptsEnd = MAKEPOINTS(lParam);
                pa->draw_line(pa, lastX, lastY, ptsEnd.x, ptsEnd.y,m_Color);
                scr->record(scr,ptsEnd.x, ptsEnd.y, 1, m_Color);
               lastX = ptsEnd.x;  lastY = ptsEnd.y;  }
         }
         break;
     case WM_PAINT:    hdc = BeginPaint(hWnd, &ps);    EndPaint(hWnd, &ps);
        break;
     case WM_CLOSE:  DestroyWindow(hWnd);   break;
     case WM_DESTROY:  PostQuitMessage(0);   break;
     default:  return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return 0;
}
// Message handler for about box.
INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
  UNREFERENCED_PARAMETER(lParam);
  switch (message)
  {   case WM_INITDIALOG:  return (INT_PTR)TRUE;
     case WM_COMMAND:
        if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
        {   EndDialog(hDlg, LOWORD(wParam));  return (INT_PTR)TRUE;  }
           break;
        }
  return (INT_PTR)FALSE;
}
```

Step-4　执行上述程序。

此程序启动后，即可依循前面的用例图及用例叙述来操作这个程序。例如涂鸦如图 23-13 所示。

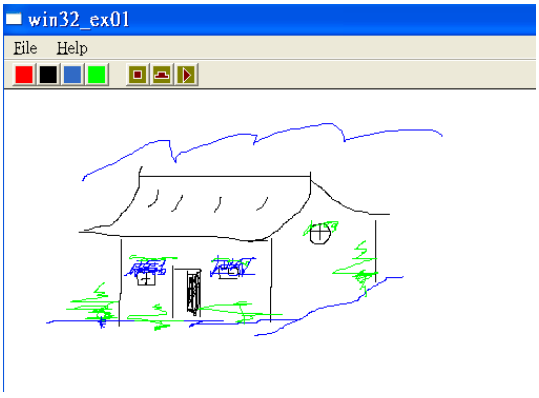


图 23-13

画完并存盘之后，可以寄给亲戚朋友；他们接到文件后，使用<Load>和三角形图标按钮，就能看到你的杰作了。

第 24 章　UML+OOPC 实用示例之二

——以录音/播放（Recorder）程序开发为例

——使用 Win32/VC++编译环境

24.1 认识“录音”概念和技术

24.1.1 认识 PCM 规格

在你的计算机上，可以使用许多软件来进行录音。本章是假设你想自己写个录音程序，通过麦克风来录制自己的声音，并且存入.wav 类型的声音文件。由于.wav 是标准的音频文件格式，你可使用许多种软件来播放它，包括 Microsoft 的 Windows Media Player 等流行的播放软件。

在编写录音程序方面，Windows 提供了完整的 API 函数可供你去调用，这样就能用来录制.wav 音频文件了。通常以 8、11、16、22 kHz 取样后再依据 PCM（Pulse Code Modulation）音轨来进行编码、数字化之后就成为.wav 格式的音频文件。由于音波本质上可以分解成为数个正弦波形及其频率的整数倍数，所以 PCM 能依固定的周期而进行取样。例如，若采取“11.025 kHz, stereo, 8-bit”模型，并且使用双声道来录音的话，则理清下述的数据关系，对于设计及编写录音程序会有很大帮助。

- 每次取样 8bit = 1Byte；
- 每秒钟取样 11025（即 11.025×1000）次；
- 录音声道数= 2；
- 可算出每秒钟将录制（$11025 \times 1 \times 2$）Bytes 的音频数据；
- 若录制 N 秒，共需要（$11025 \times 2 \times N$）Bytes 的内存空间来存放。

其中，.wav 格式的音频文件除了 8 bits 的取样模型之外，还可以选择 16 bits 取样模型。8 bits 取样意味着每次取样 8 bits，所以声音文件的每笔数据长度是 1 Byte；若为 16 bits 音频文件，则每笔数据长度是 2 Bytes。当取样频率一样时，每次取样较大者可得到较佳的音质，但是需要较大的储存空间。

.wav 音频文件又分为单声道（channel）和双声道。单声道录音时，.wav 声音文件是由单一数据所构成；使用双声道录音时，会同时接收两笔数据，一笔由左声道输出的，另一笔是由右声道输出的，依序写入.wav 音频文件里。

24.1.2 设定录音格式

在你的 VB.NET 录音程序里，只需要将上述的参数赋值（assign）给如下的 format 结构里，然后调用 Win32 API 就可以了，代码如下。

```
……
Private format As WAVEFORMATEX
   ……..
   '   "11.025 kHz, stereo, 8-bit" 且双声道
   format.wFormatTag = Wave.WAVE_FORMAT_PCM
   format.nChannels = 2
   format.nSamplesPerSec = 11025
   format.wBitsPerSample = 8
   format.nBlockAlign = format.nChannels * (format.wBitsPerSample / 8)
   format.nAvgBytesPerSec = format.nSamplesPerSec * format.nBlockAlign
   format.cbSize = 0
   ……
```

Windows 依据 format 里的信息进行音频的采样，其过程中会不断地把采集到的音频数据存入缓冲区（buffer）里，而你的程序必须及时给予足够的缓冲区，或者即刻从缓冲区取出数据。

24.1.3 设定缓冲区格式

你的程序必须安排缓冲区，并决定其大小，其定义的格式如下。

缓冲区格式的内容：

```
Structure WAVEHDR
lpData As IntPtr  ---- 指向你所安排的录音缓冲区(例如 Byte 数组)
dwBufferLength As Integer ---- 缓冲区的长度值
dwBytesRecorded As Integer ---- 供给录音系统使用，说明已经录了几个 Byte
dwUser As IntPtr ---- 指向原来调用 waveInxxx 来录音的对象
dwFlags As Integer ---- 相关的 Flag
dwLoops As Integer ---- 循环 Counter
lpNext As IntPtr ---- 指向下一个缓冲区定义(WAVEHDR)
reserved As Integer ---- 保留作为其他用途
End Structure
```

设定好缓冲区之后，你的程序就可以从缓冲区不断取出音频数据。取得数据后，可以将它存入文件里。因此，你必须知道.wav 文件的格式，才能准确地保存好文件。

24.1.4 将音频数据写入.wav 声音文件

在.wav 的音频文件里，开头 44 Bytes 必须记载我们采取的是 8 bits 还是 16 bits，也必须记载我们选择几声道方式录音。这 44 Bytes 长的区域通称为.wav 音频文件的文件头（Header），继文件头之后（即从 45 Bytes 起）到文件尾才储存所录的音频数据。如表 24-1 所示。

表 24-1

（文件头：固定占 44 个 Byte）	
Char(4)	储存 "RIFF" 字符串
Int32	储存 FileSize – 8 整数值（表示从此到文件尾的距离长度）
Char(4)	储存 "WAVE" 字符串
Char(4)	储存 "fmt" 字符串
Int32	储存 16 整数值（表示后续 16Bytes 记载声音文件的格式）
[音频文件格式区] Int16	wFormatTag 值（即声音文件规格, 0x0001 表示 PCM 规格）
Int16	nChannels 值（即声道数）
Int32	nSamplesPerSec 值（即每秒取样几次——频率）
Int32	nAvgBytesPerSec 值（即每秒平均数据量）
Int16	nBlockAlign 值（即取样 Bit 数/8 的整数值）
Int16	wBitsPerSample 值（即每次取样的 Bit 数——样本大小）
Char（4）	储存 "data" 字符串
Int32	储存 FileSize – 44 整数值（即实际音频数据的长度）
（文件内容：占 FileSize – 44 个 Byte）	
Char（FileSize – 44）	储存实际所录制的音频数据

24.1.5 使用 Win32 所提供的录音 API

在录音方面，Win32 提供如下的常用 API 函数：

```
waveOutGetNumDevs() ---- 侦测可用的音频设备的个数
waveOutGetDevCaps() ---- 侦测可用的音频设备的能力
waveInOpen() ---- 找到可用的录音设备，然后开启它
waveInPrepareHeader() ---- 给予缓冲区（Buffer）的格式
waveInAddBuffer() ---- 实际增添一个缓冲区
waveInStart() ---- 开始录音
waveInUnprepareHeader() ---- 除去缓冲区（Buffer）的格式
waveInReset() ---- 立即停止录音
waveInStop() ---- 待目前缓冲区满时，立即停止录音
waveInClose() ---- 关闭录音设备
```

24.2 单纯“录音”的示例分析

请先分析一个单纯录音 10 秒钟的例子，从这个例子里可容易地理解计算机录音的基本技巧。

24.2.1 绘制系统用例图

在本示例里，因为操作需求简单，只是系统执行过程较复杂，所以系统用例图并不复杂，如图 24-1 所示。

图 24-1 显示出用户需求是简单的：只是想要录音 10 秒钟，然后存入.wav 格式的文件。

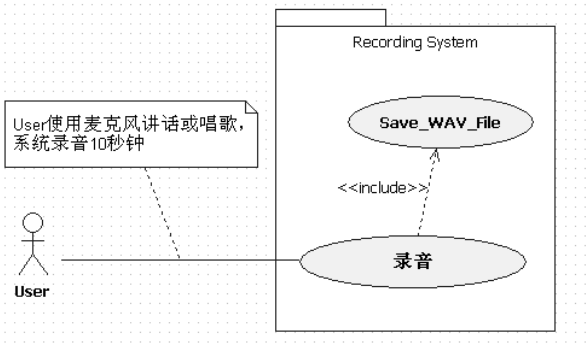


图 24-1

24.2.2 绘制类图

此为典型的软硬件集成设计的系统，其中包括三种不同类型的模块。

- 硬件设备

  例如麦克风、硬盘等。

- 驱动软件（driver）或服务提供者（service provider）

  例如录音的 MMSYSTEM（即 winmm.lib）程序。

- 主控软件（或称为应用软件）

  例如“声音录制”的 OOPC 程序。

为了明确表达上述三者之间的密切关系，先绘制一个软硬集成设计的类图，如图 24-2 所示。

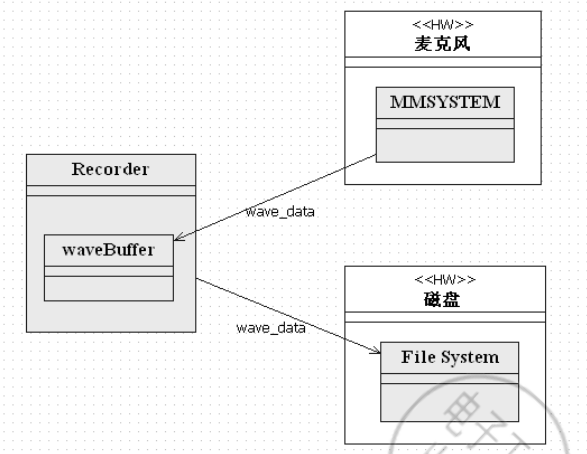


图 24-2

接着，把焦点放在软件（包括驱动程序和应用程序）部分，并且将驱动程序视为应用程序与硬件设备之间的接口，就能绘制以软件为主的类图，如图 24-3 所示。

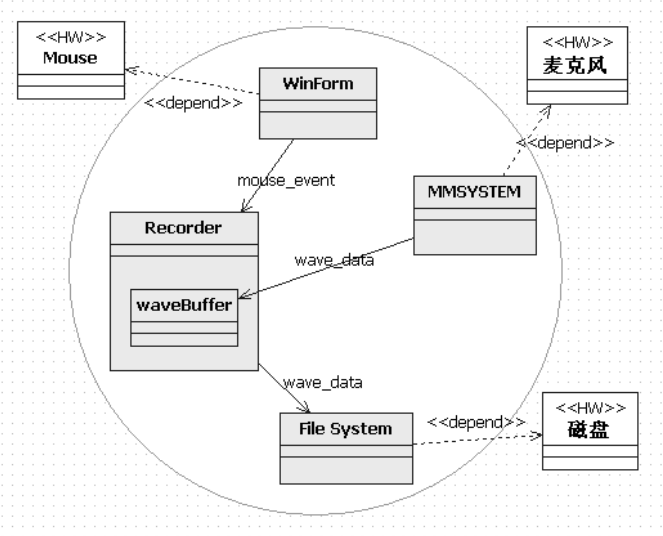


图 24-3

传统上，我们常常把硬件视为坚强主干，它支撑着许许多多的软件系统，软件就像树叶一般挂在树干或树枝上。而图 24-3 则采取比较新潮的观点，就是把软件视为树干，而硬件就像树叶一般挂在软件上。由于两个观点都没有错，但是换个新观点总是有益无害的。

24.2.3 绘制序列图

在此，藉由序列图来表达上述 Use Case 幕后的系统流程，如图 24-4 所示。

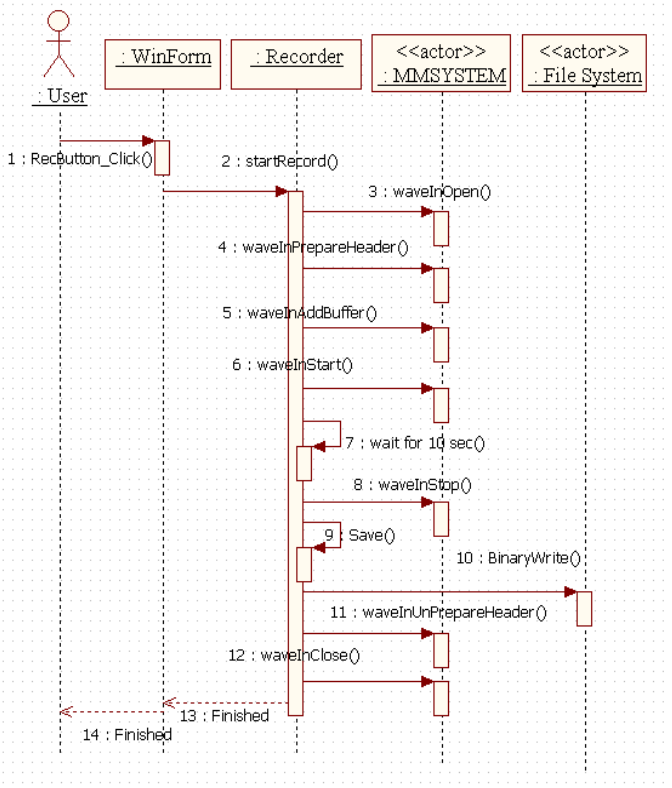


图 24-4

把驱动程序视为接口，则硬件模块就像汽车的轮胎，驱动程序就像轮盘（就是车辆与轮胎的接口），应用程序就是车辆本身。这种新观点能带来极大的利益，即硬件模块很容易像轮胎一样迅速汰旧换新。

24.3 “录音/播放”示例的分析

24.3.1 绘制系统用例图

本示例的用例图稍微复杂一些，如图 24-5 和图 24-6 所示。

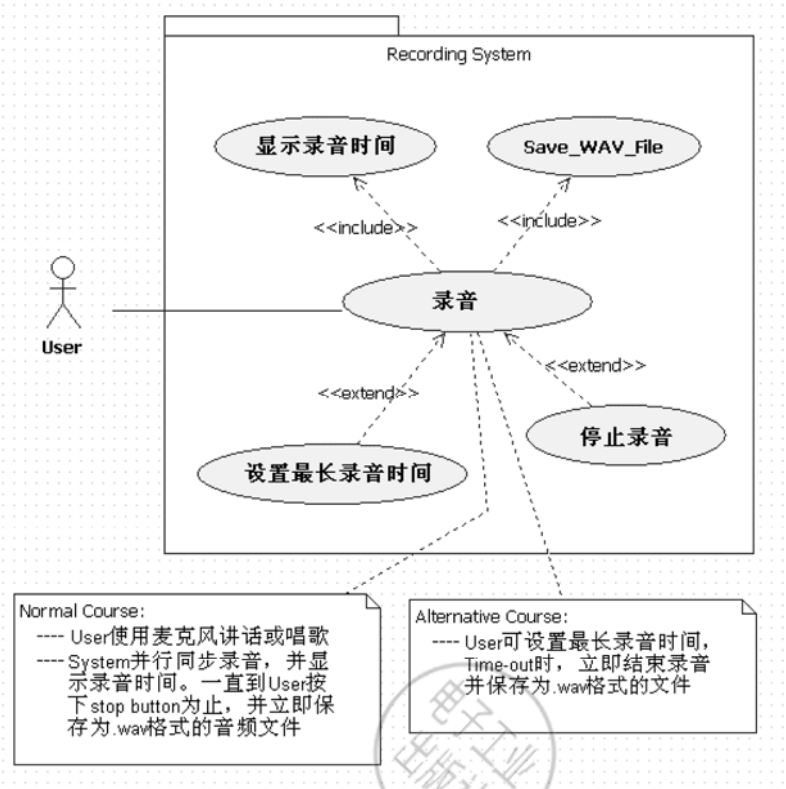

Recording System
显示录音时间
Save_WAV_File
<<include>>
<<include>>
User
录音
<<extend>>
<<extend>>
停止录音
设置最长录音时间
Normal Course:
---- User使用麦克风讲话或唱歌
---- System并行同步录音，并显示录音时间。一直到User按下stop button为止，并立即保存为.wav格式的音频文件
Alternative Course:
---- User可设置最长录音时间，Time-out时，立即结束录音并保存为.wav格式的文件

图 24-5

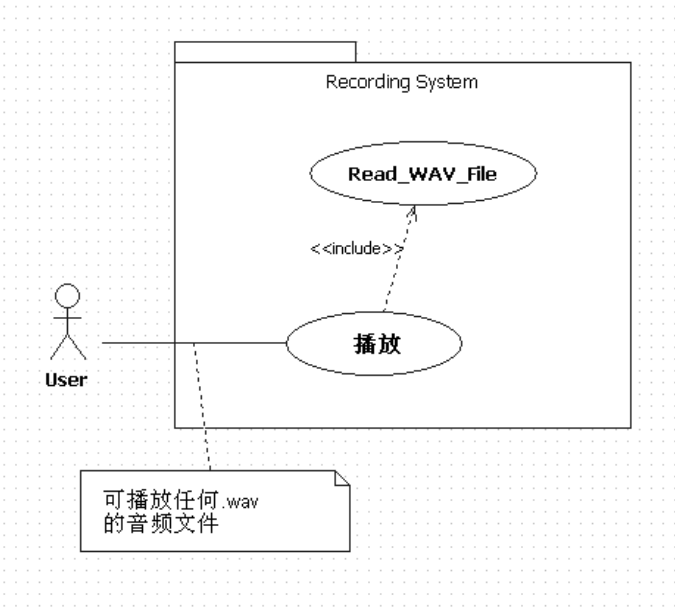

Recording System
Read_WAV_File
<<include>>
User
播放
可播放任何.wav的音频文件

图 24-6

24.3.2 绘制类图

此类图与上一节相同，就不再绘制了，请参照图 24-2 和图 24-3。

24.3.3 绘制序列图

序列图录音部分较复杂，如图 24-7 所示。

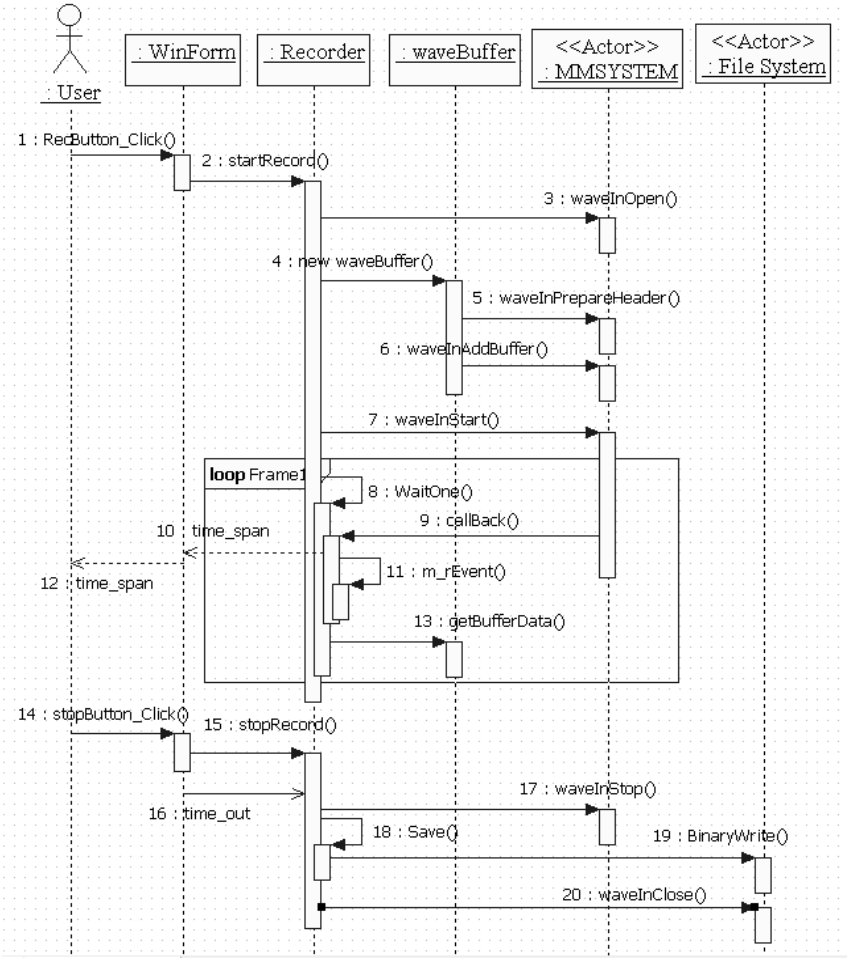


图 24-7

24.4 “录音/播放”示例的实现

——使用 OOPC

Step-1 先创建 VC++项目，内容如图 24-8 所示。

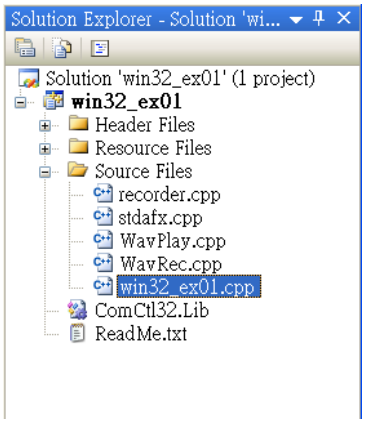


图 24-8

Step-2 定义 WavRec 类，担任录音的工作，录好之后，会存入"c:\\SymSound.wav"音频文件里。

编写 WavRec.h 文件

```
/*  WavRec.h   */
#ifndef WAVREC_H
#define WAVREC_H
#include "lw_oopc.h"

CLASS(WavRec)
{ void (*init)(void*);
  void (*prepare)(void*);
  void (*start)(void*);
  void (*close)(void*);
};
#endif
```

编写 WavRec.c 代码

```
/*  WavRec.c    */
#include "StdAfx.h"
#include "stdio.h"
#include <io.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys\stat.h>
#include <MMSystem.h>
#include "wavrec.h"

#define OFFSET_FORMATTAG          20
#define OFFSET_CHANNELS           22
#define OFFSET_SAMPLESPERSEC      24
#define OFFSET_AVGBYTESPERSEC     28
#define OFFSET_BLOCKALIGN         32
#define OFFSET_BITSPERSAMPLE      34
#define OFFSET_WAVEDATA           44
#define HEADER_SIZE               OFFSET_WAVEDATA
```

```
/*  定义.wav 文件的标头部分  */
typedef struct
{ char R1;   char I2;    char F3;   char F4;   unsigned long WAVElen;
  struct
  {  char W1;   char A2;   char V3;   char E4;   char f5;   char m6;   char t7;
     char space;   unsigned long fmtlen;
     struct
     {  unsigned short FormatTag;       unsigned short Channels;
        unsigned long SamplesPerSec;    unsigned long AvgBytesPerSec;
        unsigned short BlockAlign;
        unsigned short BitsPerSample; /* format specific for PCM */
      } fmt;
     char d8;   char a9;    char t10;     char a11;
     unsigned long datalen;
  } WAVE;
} RIFF;

long total_len  = 0;    HWAVEIN m_hwIn;
WAVEFORMATEX m_format;    WAVEHDR m_wHdr;
RIFF header;   int  hfile, xfile;

/*   每次录音 buffer 满时，将音频数据写入 Buffer 里  */
static void ProcessHeader(WAVEHDR * pvr)
{ write(hfile, pvr->lpData, pvr->dwBytesRecorded);
  total_len +=  pvr->dwBytesRecorded;
if(waveInAddBuffer(m_hwIn, pvr, sizeof(WAVEHDR)) != 0)
  MessageBox( NULL, _T("waveInAddBuffer error!"), _T(""), MB_OK |
              MB_ICONINFORMATION);
}

*   这是录音 thread 的 call back 函数  */
static void CALLBACK waveInProc(HWAVEIN hwi,UINT uMsg,DWORD dwInstance,DWORD
dwParam1,DWORD dwParam2)
{ WAVEHDR *pHdr=NULL;
  switch(uMsg)
  {  case WIM_CLOSE:     break;
     case WIM_DATA:
        ProcessHeader((WAVEHDR *)dwParam1);    break;
     case WIM_OPEN:   break;
     default:  break;  }
}
/*  先写入文件头，但音频数据写完之后，必须回来 Update 它  */
static void writeFileHeader()  /*  This is called by prepare()  */
{ header.R1 = 'R';   header.I2 = 'I';   header.F3 = 'F';   header.F4 = 'F';
  header.WAVElen = 36;  /* sizeof(header.WAVE);  */
  header.WAVE.W1 = 'W';   header.WAVE.A2 = 'A';
  header.WAVE.V3 = 'V';    header.WAVE.E4 = 'E';
  header.WAVE.f5 = 'f';      header.WAVE.m6 = 'm';
  header.WAVE.t7 = 't';      header.WAVE.space = ' ';
  header.WAVE.fmtlen = 16;
  header.WAVE.fmt.FormatTag = WAVE_FORMAT_PCM;
  header.WAVE.d8 = 'd';   header.WAVE.a9 = 'a';
  header.WAVE.t10 = 't';   header.WAVE.a11 = 'a';
  header.WAVE.datalen = 0; /* we don't know yet *///暂时而已，写完数据必须回来
  update
  header.WAVE.fmt.BitsPerSample = m_format.wBitsPerSample;
  header.WAVE.fmt.Channels = m_format.nChannels ;
  header.WAVE.fmt.SamplesPerSec = m_format.nSamplesPerSec;
  header.WAVE.fmt.AvgBytesPerSec =m_format.nAvgBytesPerSec;
```

```
  header.WAVE.fmt.BlockAlign =m_format.nBlockAlign ;

/*  Open Wave file named: c:\\SymSound.wav   */
/* 本示例只写入赋值的文件名  */
hfile = open("c:\\SymSound.wav",O_CREAT | O_TRUNC | O_RDWR | O_BINARY,
S_IWRITE);
if(!hfile || hfile == -1)
  MessageBox(NULL,_T("can not open .wav!"),_T(""), MB_OK |
MB_ICONINFORMATION);
/* 已经打开文件了 */
/* Write WaveFile Header     */
if(write(hfile, &header, sizeof(header)) != sizeof(header))
  MessageBox(NULL,_T("write header Err!"),_T(""), MB_OK |
             MB_ICONINFORMATION);
/* 已经写入文件头了  */
}
static void prepare(void* t)
{ int kk;  MMRESULT ret=0;
  m_format.wFormatTag     = WAVE_FORMAT_PCM;
  m_format.nChannels      = 2; //channels;  1=mono, 2=stereo
  m_format.nSamplesPerSec = 44100; // or 11025;
  m_format.wBitsPerSample = 16; //bitsPerSample;
                              //16 for high quality, 8 for telephone-grade
  m_format.nBlockAlign = m_format.nChannels * (m_format.wBitsPerSample/8);
  m_format.cbSize   = 0; //sizeof(WAVEFORMATEX);
  m_format.nAvgBytesPerSec = m_format.nSamplesPerSec * m_format.nChannels *
  (m_format.wBitsPerSample / 8);
  writeFileHeader();
/*  开启录音设备   */
  ret=waveInOpen(&m_hwIn,WAVE_MAPPER,&m_format,(DWORD_PTR)waveInProc,
                 (DWORD_PTR)NULL,CALLBACK_FUNCTION);
  if(ret!=MMSYSERR_NOERROR)
  {  MessageBox(NULL,_T("waveInOpen err ..."),_T(""), MB_OK |
                MB_ICONINFORMATION);
     return;
  }
/*------------------------>> PrepareBuffers <<---------------------*/
  m_wHdr.lpData=(LPSTR)HeapAlloc(GetProcessHeap(),8,
                                 m_format.nAvgBytesPerSec*8);
  m_wHdr.dwBufferLength=m_format.nAvgBytesPerSec*8;
  m_wHdr.dwUser=0;
  if(waveInPrepareHeader(m_hwIn,&m_wHdr, sizeof(WAVEHDR)) != 0)
  {  MessageBox(NULL,_T("waveInPrepareHeader error!"),_T(""), MB_OK |
                MB_ICONINFORMATION);
  return;
  }
  if( waveInAddBuffer(m_hwIn,&m_wHdr,sizeof(WAVEHDR)) != 0)
  {  MessageBox(NULL,_T("waveInAddBuffer error!"),_T(""), MB_OK |
                MB_ICONINFORMATION);
     return;
  }
  return;
}
/*   正式开始录音   */
static void start(void*t)
{ if (waveInStart(m_hwIn) != 0)
  {  MessageBox(NULL,_T("waveInStart error!"),_T(""), MB_OK |
                MB_ICONINFORMATION);
     return;   }
```

```
  return;
}
//-------------------------------------------------
/* 结束录音 */
static void close(void*t)
{ /*------>>   UnPrepareBuffers <<-----------------*/
  waveInStop(m_hwIn);
  /*------->> Close Device <<---------------------*/
  DWORD curr_time, base_time, draw_time;
  // Waiting for thread finished
  base_time = GetTickCount();
  do {   curr_time =  GetTickCount() - base_time;     }
  while (curr_time < 300);
  if(m_wHdr.lpData)
  {
     waveInUnprepareHeader(m_hwIn,&m_wHdr,sizeof(WAVEHDR));
     HeapFree(GetProcessHeap(),0,m_wHdr.lpData);
     ZeroMemory(&m_wHdr, sizeof(WAVEHDR));
  }
  waveInClose(m_hwIn);

/*  更新文件头的音频数据长度 */
  header.WAVE.datalen += total_len;
  header.WAVElen += total_len;
  lseek(hfile,0,SEEK_SET);
  write(hfile, &header, sizeof(header));  // Update Wave File header
                              // for recording the real data length
  lseek(hfile,0,SEEK_END);
  close(hfile);
  MessageBox(NULL,_T("SaveToFile OK"),_T(""), MB_OK | MB_ICONINFORMATION);
}
static void init(void *t)
{ WavRec* cthis = (WavRec*)t;  }
CTOR(WavRec)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(prepare, prepare)
  FUNCTION_SETTING(start, start)
  FUNCTION_SETTING(close, close)
END_CTOR
```

录好之后，会存入"c:\\SymSound.wav"音频文件里。

Step-3 定义 WavPlay 类，负责从 Linter DB 读取歌曲内容，然后播放出来。

编写 WavPlay.h 文件

```
/*  WavPlay.h   */
#ifndef WAVPLAY_H
#define WAVPLAY_H
#include "lw_oopc.h"

CLASS(WavPlay)
{ char* m_buffer;
  int m_file_size;
  void (*init)(void*);
  void (*play_File)(void*);
};
#endif
```

编写 WavPlay.c 代码

```
/*  WavPlay.c   */
#include "StdAfx.h"
#include "stdio.h"
#include <MMSystem.h>
#include "wavplay.h"

#define OFFSET_FORMATTAG            20
#define OFFSET_CHANNELS             22
#define OFFSET_SAMPLESPERSEC        24
#define OFFSET_AVGBYTESPERSEC       28
#define OFFSET_BLOCKALIGN           32
#define OFFSET_BITSPERSAMPLE        34
#define OFFSET_WAVEDATA             44
#define HEADER_SIZE                 OFFSET_WAVEDATA

/*   播放 thread 的 Call Back 函数   */
static void CALLBACK VoicePlayProc(HWAVEOUT hwo, UINT uMsg, DWORD dwInstance,
DWORD dwParam1, DWORD dwParam2)
{ switch(uMsg)
  {  case WOM_OPEN:   break;
       case WOM_DONE:
       {  WAVEHDR *whdr = (WAVEHDR*)dwParam1;
          if(whdr->dwUser)
            waveOutWrite(hwo, whdr, sizeof(WAVEHDR));
          else
          {  waveOutUnprepareHeader(hwo, whdr, sizeof(WAVEHDR));
            delete whdr;  }
       }
       break;
    case WOM_CLOSE:   break;
  }
}
static void init(void *t)
{ WavPlay* cthis = (WavPlay*)t;  }

/*  play wave data from c:\\SymSound.wav   */
/*  开启 c:\\SymSound.wav 并且播放  */
static void play_File(void*t)
{ char* buffer;   int file_size;
  WAVEFORMATEX m_Format;
  HWAVEOUT m_hPlay;

  // load wave data from c:\\SymSound.wav to buffer[]
  FILE *pf = fopen("c:\\SymSound.wav", "rb");
  if(!pf)
     {  MessageBox(NULL, "File not found", NULL, NULL);
        return;   }
  fseek(pf, 0, SEEK_END);      file_size = ftell(pf);
  fseek(pf, 0, SEEK_SET);      buffer = new char [file_size];
  fread(buffer, 1, file_size, pf);   fclose(pf);

  // play the wave data in buffer[]
  m_Format.wFormatTag = *((WORD* )(buffer + OFFSET_FORMATTAG    ));
  m_Format.nChannels = *((WORD* )(buffer + OFFSET_CHANNELS     ));
  m_Format.nSamplesPerSec = *((DWORD*)(buffer + OFFSET_SAMPLESPERSEC ));
  m_Format.nAvgBytesPerSec = *((DWORD*)(buffer + OFFSET_AVGBYTESPERSEC));
  m_Format.nBlockAlign = *((WORD* )(buffer + OFFSET_BLOCKALIGN   ));
```

```
  m_Format.wBitsPerSample = *((WORD* )(buffer + OFFSET_BITSPERSAMPLE ));


  WAVEHDR *wHdr = new WAVEHDR;
  ZeroMemory((void*)wHdr, sizeof(WAVEHDR));

  wHdr->lpData = buffer + HEADER_SIZE;
  wHdr->dwBufferLength = file_size - HEADER_SIZE;
  wHdr->dwUser = (DWORD)false;

  /*   Find a waveOut device and open it   */
  for(UINT devid = 0; devid < waveOutGetNumDevs(); devid++)
  {  if(devid == waveOutGetNumDevs())   // Error, no free devices found
     return ;
     if(waveOutOpen(&m_hPlay, WAVE_MAPPER, &m_Format,
                    (DWORD)VoicePlayProc, 0,
                     CALLBACK_FUNCTION) == MMSYSERR_NOERROR)
        break;   /*  Usable device found, stop searching  */
  }

  if(waveOutPrepareHeader(m_hPlay, wHdr, sizeof(WAVEHDR))!=MSYSERR_NOERROR)
     return;
  if(waveOutWrite(m_hPlay, wHdr, sizeof(WAVEHDR)) != MMSYSERR_NOERROR)
     return;
  return;
}
//-----------------------------------------
CTOR(WavPlay)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(play_File, play_File)
END_CTOR
```

编写 recorder.h 文件

```
/*   recorder.h   */
#ifndef RECORDER_H
#define RECORDER_H
#include "lw_oopc.h"
#include "WavPlay.h"
#include "WavRec.h"

CLASS(Recorder)
{ WavPlay* wp;
  WavRec* wr;
  int m_state;
  void (*init)(void*);
  void (*play_WaveFile)(void*);
  void (*prepare_rec)(void*);
  void (*start_rec)(void*);
  void (*close_rec)(void*);
};
#endif
```

编写 recorder.c 代码

```
/*  recorder.c    */
/*   这里控制 WavRec 和 WavPlay 两个对象   */
#include "StdAfx.h"
```

```
#include "stdio.h"
#include "recorder.h"
#include "WavPlay.h"
#include "WavRec.h"

extern void* WavPlayNew();
extern void* WavSaveNew();
extern void* WavRecNew();
/*   ----------------------------------   */
/*  生成WavPlay 对象负责播放    */
/*  生成WavRec 对象负责录音    */
static void init(void *t)
{ Recorder* cthis = (Recorder*)t;
  (cthis->wp) = (WavPlay*)WavPlayNew();   (cthis->wp)->init(cthis->wp);
  (cthis->wr) = (WavRec*)WavRecNew();    (cthis->wr)->init(cthis->wr);
}
static void play_WaveFile(void* t)  /* play wave data in file  */
{ Recorder* cthis = (Recorder*)t;
  (cthis->wp)->play_File(cthis->wp);
}
// writting wave file header
/*   准备录音   */
static void prepare_rec(void* t)
{ Recorder* cthis = (Recorder*)t;
  (cthis->wr)->prepare(cthis->wr);
  cthis->m_state = 1;
 }
/*   开始录音   */
static void start_rec(void* t)
{ Recorder* cthis = (Recorder*)t;
  (cthis->wr)->start(cthis->wr);
  cthis->m_state = 1;
}
static void close_rec(void* t)
{ Recorder* cthis = (Recorder*)t;
  (cthis->wr)->close(cthis->wr);
  cthis->m_state = 2;
}
CTOR(Recorder)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(play_WaveFile, play_WaveFile)
  FUNCTION_SETTING(prepare_rec, prepare_rec)
  FUNCTION_SETTING(start_rec, start_rec)
  FUNCTION_SETTING(close_rec, close_rec)
END_CTOR
```

编写 WIN32 应用程序代码

```
// win32_ex01.cpp : Defines the entry point for the application.
#include "stdafx.h"
#include "windows.h"
#include "tchar.h"
#include <commctrl.h>
#include "recorder.h"
#include "win32_ex01.h"

#define MAX_LOADSTRING 100
extern void* RecorderNew();
Recorder* scr;
```

```
HINSTANCE hInst;
TCHAR szTitle[MAX_LOADSTRING];
TCHAR szWindowClass[MAX_LOADSTRING];
HWND hWindow;
int mState;   HWND hWndToolbar;   BOOL bMouseDown;   POINTS ptsEnd;
static char g_szClassName[] = "MyWindowClass";

ATOM             MyRegisterClass(HINSTANCE hInstance);
BOOL             InitInstance(HINSTANCE, int);
LRESULT CALLBACK   WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK   About(HWND, UINT, WPARAM, LPARAM);
int APIENTRY _tWinMain(HINSTANCE hInstance,  HINSTANCE hPrevInstance,
                       LPTSTR  lpCmdLine,  int nCmdShow)
{ UNREFERENCED_PARAMETER(hPrevInstance);
  UNREFERENCED_PARAMETER(lpCmdLine);

  MSG msg;   HACCEL hAccelTable;
  LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
  LoadString(hInstance, IDC_WIN32_EX01, szWindowClass, MAX_LOADSTRING);
  MyRegisterClass(hInstance);
  if (!InitInstance (hInstance, nCmdShow))   return FALSE;
  hAccelTable = LoadAccelerators(hInstance,
  MAKEINTRESOURCE(IDC_WIN32_EX01));
  while (GetMessage(&msg, NULL, 0, 0))
  {  if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
     {  TranslateMessage(&msg);
        DispatchMessage(&msg);  }
  }
  return (int) msg.wParam;
}
ATOM MyRegisterClass(HINSTANCE hInstance)
{ WNDCLASSEX wcex;
  wcex.cbSize = sizeof(WNDCLASSEX);
  wcex.style          = CS_HREDRAW | CS_VREDRAW;
  wcex.lpfnWndProc    = WndProc;
  wcex.cbClsExtra       = 0;
  wcex.cbWndExtra       = 0;
  wcex.hInstance       = hInstance;
  wcex.hIcon          = LoadIcon(hInstance,
  MAKEINTRESOURCE(IDI_WIN32_EX01));
  wcex.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wcex.hbrBackground    = (HBRUSH)(COLOR_WINDOW+1);
  wcex.lpszMenuName    = MAKEINTRESOURCE(IDC_WIN32_EX01);
  wcex.lpszClassName    = szWindowClass;
  wcex.hIconSm       = LoadIcon(wcex.hInstance,
  MAKEINTRESOURCE(IDI_SMALL));
  return RegisterClassEx(&wcex);
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{ hInst = hInstance;
  hWindow = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
                         NULL, NULL, hInstance, NULL);
  if (!hWindow)  return FALSE;
  INITCOMMONCONTROLSEX InitCtrlEx;
  InitCtrlEx.dwSize = sizeof(INITCOMMONCONTROLSEX);
  InitCtrlEx.dwICC  = ICC_BAR_CLASSES;
```

```
InitCommonControlsEx(&InitCtrlEx);
TBBUTTON tbrButtons[10];
tbrButtons[0].iBitmap   = 0;               tbrButtons[0].idCommand = 0;
tbrButtons[0].fsState   = TBSTATE_ENABLED;
tbrButtons[0].fsStyle   = TBSTYLE_SEP;
tbrButtons[0].dwData    = 0L;             tbrButtons[0].iString   = 0;
tbrButtons[1].iBitmap   = 0;
tbrButtons[1].idCommand = CM_RED; //IDM_FILE_NEW;
tbrButtons[1].fsState   = TBSTATE_ENABLED;
tbrButtons[1].fsStyle   = TBSTYLE_BUTTON;
tbrButtons[1].dwData= 0L;   tbrButtons[1].iBitmap= 0;
tbrButtons[1].iString= 0;
tbrButtons[2].iBitmap   = 1;
tbrButtons[2].idCommand = CM_BLACK;
tbrButtons[2].fsState   = TBSTATE_ENABLED;
tbrButtons[2].fsStyle   = TBSTYLE_BUTTON;
tbrButtons[2].dwData    = 0L;      tbrButtons[2].iString   = 0;

tbrButtons[3].iBitmap   = 2;
tbrButtons[3].idCommand = CM_BLUE;
tbrButtons[3].fsState   = TBSTATE_ENABLED;
tbrButtons[3].fsStyle   = TBSTYLE_BUTTON;
tbrButtons[3].dwData    = 0L;     tbrButtons[3].iString   = 0;

tbrButtons[4].iBitmap   = 3;
tbrButtons[4].idCommand = CM_GREEN;
tbrButtons[4].fsState   = TBSTATE_ENABLED;
tbrButtons[4].fsStyle   = TBSTYLE_BUTTON;
tbrButtons[4].dwData    = 0L;    tbrButtons[4].iString   = 0;

tbrButtons[5].iBitmap   = 0;   tbrButtons[5].idCommand = 0;
tbrButtons[5].fsState   = TBSTATE_ENABLED;
tbrButtons[5].fsStyle   = TBSTYLE_SEP;
tbrButtons[5].dwData    = 0L;     tbrButtons[5].iString   = 0;

tbrButtons[6].iBitmap   = 0;    tbrButtons[6].idCommand = 0;
tbrButtons[6].fsState   = TBSTATE_ENABLED;
tbrButtons[6].fsStyle   = TBSTYLE_SEP;
tbrButtons[6].dwData    = 0L;  tbrButtons[6].iString   = 0;

tbrButtons[7].iBitmap   = 4;
tbrButtons[7].idCommand = IDM_CLEAR_AND_BEGIN;
tbrButtons[7].fsState   = TBSTATE_ENABLED;
tbrButtons[7].fsStyle   = TBSTYLE_BUTTON;
tbrButtons[7].dwData    = 0L;
tbrButtons[7].iString   = 0;

tbrButtons[8].iBitmap   = 5;
tbrButtons[8].idCommand = IDM_STOP;
tbrButtons[8].fsState   = TBSTATE_ENABLED;
tbrButtons[8].fsStyle   = TBSTYLE_BUTTON;
tbrButtons[8].dwData    = 0L;
tbrButtons[8].iString   = 0;

tbrButtons[9].iBitmap   = 6;
tbrButtons[9].idCommand = IDM_STOP_AND_DRAW;
tbrButtons[9].fsState   = TBSTATE_ENABLED;
tbrButtons[9].fsStyle   = TBSTYLE_BUTTON;
tbrButtons[9].dwData    = 0L;    tbrButtons[9].iString   = 0;
```

```
  hWndToolbar = CreateToolbarEx(hWindow, WS_VISIBLE | WS_CHILD | WS_BORDER,
                                IDB_BITMAP2, 10,
                                hInst, IDB_BITMAP2,  tbrButtons,  10,
                                16, 16, 16, 16,  sizeof(TBBUTTON));
  ShowWindow(hWindow, nCmdShow);
  UpdateWindow(hWindow);
  return TRUE;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{ int wmId, wmEvent;  TCHAR sss[100];
  PAINTSTRUCT ps;   HDC hdc;   int ret;
  switch (message)
  {  case WM_CREATE:
       scr = (Recorder*)RecorderNew();   scr->init(scr);    mState = 0;
       break;
     case WM_COMMAND:
       wmId   = LOWORD(wParam);   wmEvent = HIWORD(wParam);
       /* Parse the menu selections:
        _stprintf (sss, _T("X  is: %d "), wmId);
       MessageBox(NULL,sss,_T(""), MB_OK | MB_ICONINFORMATION);
       */
       switch (wmId)
       {  case IDM_CLEAR_AND_BEGIN:  /*  begin recording  */
            scr->prepare_rec(scr);  scr->start_rec(scr); mState = 1;
            break;
          case IDM_STOP:  /* stop recording and save wave data to file  */
            if(mState == 0)
            {  MessageBox(NULL,_T("No wave data to save!"),_T(""), MB_OK |
                          MB_ICONINFORMATION);
               return 0;
            }
            if(mState == 2)
            {  MessageBox(NULL,_T("Already Saved in file!"),_T(""),
                          MB_OK | MB_ICONINFORMATION);
               return 0;
            }
            scr->close_rec(scr);  /*  Save to c:\\SymSound.wav  */
            mState = 2;
            break;
          case IDM_STOP_AND_DRAW:
            if(mState == 1) return 0;  /*   is recording   */
               scr->play_WaveFile(scr);
               break;
          case IDM_ABOUT:
            DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
            break;
          case IDM_EXIT:  DestroyWindow(hWindow);   break;
          default:  return DefWindowProc(hWnd, message, wParam, lParam);
       }
       break;
     case WM_PAINT:   hdc = BeginPaint(hWnd, &ps);   EndPaint(hWnd, &ps);
       break;
     case WM_CLOSE:  DestroyWindow(hWnd);   break;
     case WM_DESTROY:    PostQuitMessage(0);    break;
     default:   return DefWindowProc(hWnd, message, wParam, lParam);
     }
  return 0;
}
```

```
/*  Message handler for about box.  */
INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{ UNREFERENCED_PARAMETER(lParam);
  switch (message)
  {  case WM_INITDIALOG:    return (INT_PTR)TRUE;
     case WM_COMMAND:
        if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
        {  EndDialog(hDlg, LOWORD(wParam));   return (INT_PTR)TRUE;  }
        break;
  }
  return (INT_PTR)FALSE;
}
```

程序启动后，出现如图 24-9 所示的画面。

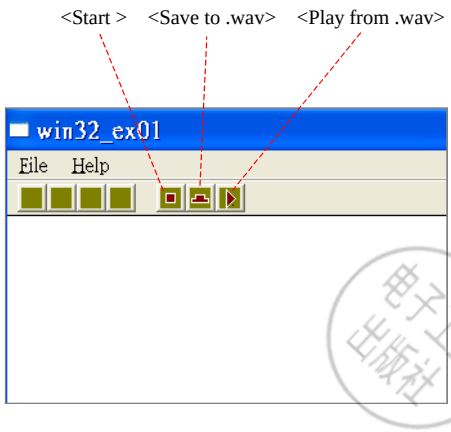


图 24-9

按下“方块”图标时开始录音，你就可以对着麦克风唱歌或讲话，一直到你按下“休止符”图标，就停止录音并存盘。之后，按下“三角形”图标，就开始播放你刚才所录制的声音了。

第 25 章　UML+OOPC 实现示例之三

——以层次分析（AHP）程序开发为例

——使用 Turbo　C 编译环境

25.1 层次分析（AHP）法简介

在前面第 13~17 章里，曾经介绍过如何编写 Vector、Matrix 及 LList 等数学计算和数据结构。在本章里，将以 AHP 为例，继续将 OOPC 应用于更深入的数学计算。

层次分析法（Analytical Hierarchy Process, 简称 AHP）是个很有趣又很有用的东西，它提供一个高效的方法去进行复杂的决策，无论在一般生活、商业或学术研究上，都有很精彩的应用。例如：

- 软件开发管理的应用

在微软的 MSDN 文档里，其利用 AHP 方法来评析与比较三个信息系统的质量，以决定哪一个系统的质量最好。请参阅：

http://msdn.microsoft.com/msdnmag/issues/05/06/TestRun/default.aspx

Test Run: The Analytic Hierarchy Process—MSDN Magazine, June 2005

- 一般生活上的应用

例如本章所举的例子，想找一个理想的工作，其所谓理想的评选标准有三个：钱多、事少、离家近。那么可以利用 AHP 方法来从多个工作机会中评选出一个比较合乎理想的工作。

- 商业上的应用

例如全球性运输公司利用 AHP 方法评选最佳转运港口。请参阅：

http://econpapers.repec.org/article/palmarecl/

v_3A6_3Ay_3A2004_3Ai_3A1_3Ap_3A70-91.htm

简而言之，AHP 是将复杂的决策情境切分为几个小部分，再将这些部分组织成为一个树状的层次结构。然后，对每一个部分的相对重要性给予权数值，再分析出各个部分的优先权。对决策者而言，用层次结构去组织有关替代方案（alternative）的评选条件或标准（criteria）、权数（weight）和分析（analysis），非常有助于对事物的了解。此外，AHP 可协助捕捉主观和客观的评估测度，检验评估的一致性，以及团队所建议的替代方案，减少团队决策的失误，如失焦、无计划、无参与等。AHP 将整个问题细分为多个较不重要的评估，但还须维持整体的决策。

AHP 方法是由 Thomas L. Saaty 教授研究开发出来的，适合多评选标准（Multi-Criteria）的复杂决策。目前市面上有许多软件工具可用，包括最著名的 Expert Choice 软件系统，以及免费的网上 AHP 软件或服务，例如：

http://www.di.unipi.it/~morge/software/JAHP.html

可下载到好用的 AHP 系统。在本章里，先介绍 AHP 的基本知识，接着说明是如何进行软件

的分析与设计，并且以 Java 语言来实现的。有一天，当你遇到复杂的决策问题时，AHP 程序就能随时随地替你解惑，协助你成为有智能的决策者。

25.2 AHP 的分析步骤

AHP 分析包含四个步骤。

Step-1 分解（Decomposing）

将整个问题分解为多个小问题。例如，整个问题是想找一个理想的工作。各项工作都有三个属性（attribute），因而将理想分为三个评选条件：钱多、事少、离家近。

Step-2 加权（Weighing）

赋予三个评选条件的权数，例如：钱多（0.643）、事少（0.283）、离家近（0.074）。其表示主观上认定“钱多”比其他两项重要。如图 25-1 所示，从图中可看出，相对上，Job-2 对“离家近”的贡献度高于 Job-1；但是在决策者心目中“离家近”的相对权数只有 0.074 而已，意味着决策者并不太在意“离家近”这项条件。

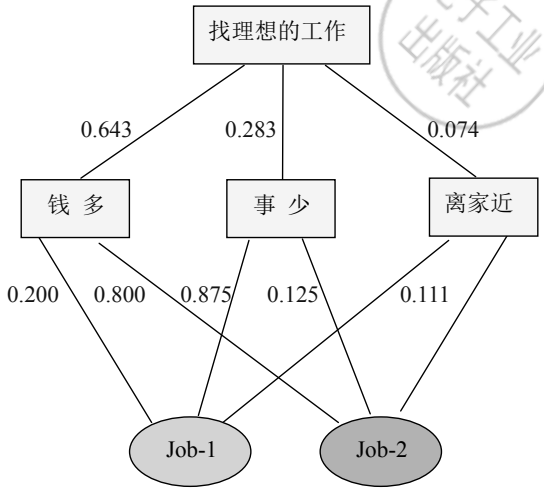


图 25-1

Step-3 评估（Evaluating）

针对 Job-1

Job-1 对“钱多”的贡献度为 0.200，而“钱多”对总目标（即“理想”）的贡献度为 0.643，所以 Job-1 通过“钱多”对总目标的贡献度为 $0.200 \times 0.643 = 0.129$。Job-1 对“事少”的贡

献度为 0.875，而“事少”对总目标（即“理想”）的贡献度为 0.283，所以 Job-1 通过“事少”对总目标的贡献度为 0.875 × 0.283 = 0.248。Job-1 对“离家近”的贡献度为 0.111，而“离家近”对总目标（即“理想”）的贡献度为 0.074，所以 Job-1 通过“离家近”对总目标的贡献度为 0.111 × 0.074 = 0.008。于是可算出 Job-1 所表现的理想度为 0.129 + 0.248 + 0.008 = 0.385。

针对 Job-2

依据同样的程序，可算出 Job-2 的情形：

- Job-2 通过“钱多”对总目标的贡献度为 0.800 × 0.643 = 0.514。
- Job-2 通过“事少”对总目标的贡献度为 0.125 × 0.283 = 0.035。
- Job-2 通过“离家近”对总目标的贡献度为 0.889 × 0.074 = 0.066。

于是可算出 Job-2 所表现的理想度为 0.514 + 0.035 + 0.066 = 0.615。

Step-4 选择（Selecting)

从上述 Step-3 分析出：

- Job-1 的理想度为 0.385。
- Job-2 的理想度为 0.615。

所以建议：Job-2 是较好的选择。

25.3 如何得到权数值

——采用“成对相比”法

25.3.1 成对相比

从图 25-1 里可看出钱多、事少、离家近三者的权数比为 0.643 : 0.283 : 0.074。有时候，并不容易得到这个权数值，此时两两成对相比，会比较简单。例如，图 25-2 里只有两个 Job 相比，每个人都很容易说出两个 Job 的比较值。图 25-2 所示的三角形偏向 Job-2，从其偏移的比例推算出其权数为 0.2: 0.8。

依此类推，从图 25-3 的三个三角形的两两比较的偏移比例，可以联合推算出其权数比为 Wx : Wy : Wz。

所以，在 AHP 方法里，通常都输入 x:y、x:z 和 y:z 的比值，如图 25-4 所示。

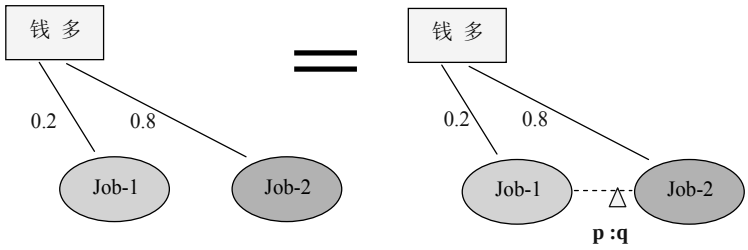


图 25-2

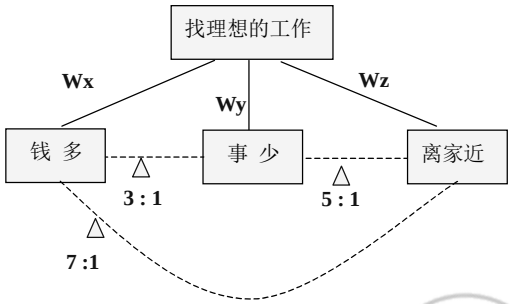


图 25-3

	钱 多	事 少	离家近	
钱 多	**1**	**x:y**	**x:z**	
事 少	**y : x**	**1**	**y:z**	
离家近	**z :x**	**z:y**	**1**	

图 25-4

然后，可通过下一节（25.3.2）叙述的计算步骤而演算出 Wx、Wy 和 Wz 的权数值，如图 25-5 所示。

	钱 多	事 少	离家近	
钱 多	**1**	**x:y**	**x:z**	**Wx**
事 少	**y:x**	**1**	**y:z**	**Wy**
离家近	**z :x**	**z:y**	**1**	**Wz**

图 25-5

总而言之，人们经常不容易说出 Wx : Wy : Wz 三者之间的比值，但是比较容易说出两两相比的 x:y、y:z 和 x:z 的比值。在 AHP 方法里，通常使用图 25-6 里的刻度表来叙述人们心中的相对权重。

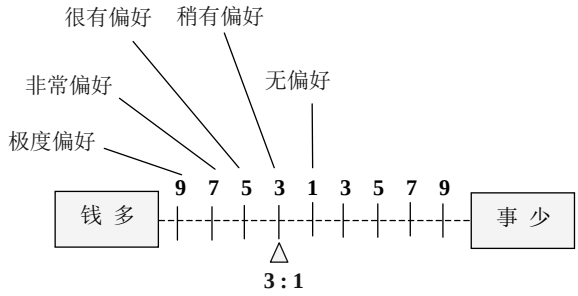


图 25-6

例如，此刻度代表偏好程度，3:1 表示对“钱多”稍有偏好，也就是说，选择工作时，钱多一点比较重要，事少并非最主要的考虑因素。于是填入表格中，如图 25-7 所示。

	钱 多	事 少	离家近	
钱 多	1	3:1		
事 少	1:3	1		
离家近			1	

图 25-7

由于比值为 3:1，表示钱多与事少两者相比，钱多稍为重要一些，但差距并不很大，再如图 25-8 所示。

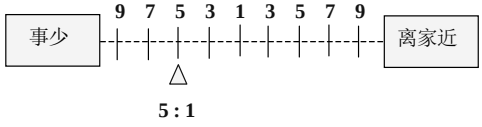


图 25-8

此图的比值为 5:1，表示对“事少”的偏好程度是“很有偏好”。填入表格中，如图 25-9 所示。

	钱 多	事 少	离家近	
钱 多	**1**	**3/1**		
事 少	**1/3**	**1**	**5/1**	
离家近		**1/5**	**1**	

图 25-9

再如图 25-10 所示。

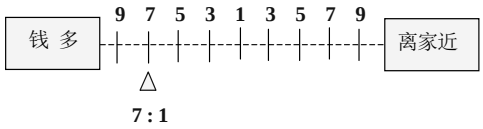


图 25-10

图 25-10 的比值是 7:1，这表示对“钱多”很有偏好。填入表格中，如图 25-11 所示。

	钱 多	事 少	离家近	
钱 多	**1**	**3**	**7**	
事 少	**1/3**	**1**	**5**	
离家近	**1/7**	**1/5**	**1**	

图 25-11

这就是两两成对相比的矩阵了。待会儿，在下一节里，将说明如何从此矩阵而演算出 Wx、Wy 和 Wz 的权数值。

25.3.2 从“成对比值”算出“权数值”

基于上一节的矩阵而演算出 Wx、Wy 和 Wz 权数值的计算步骤如下。

Step-1 计算各列的总和，如图 25-12 所示。

	钱 多	事 少	离家近	
钱 多	**1**	**3**	**7**	
事 少	**1/3**	**1**	**5**	
离家近	**1/7**	**1/5**	**1**	
总 和	**31/21**	**21/5**	**13**	

图 25-12

Step-2 各个值除以该列的总和，如图 25-13 所示。

	钱 多	事 少	离家近	
钱 多	**21/31**	**5/7**	**7/13**	
事 少	**7/31**	**5/21**	**5/13**	
离家近	**3/31**	**1/21**	**1/13**	

图 25-13

Step-3 计算各列的平均值。

钱 多：（21/31 + 5/7 + 7/13）/ 3 = 0.643

事 少：（7/31 + 5/21 + 5/13）/ 3 = 0.283

离家近：（3/31 + 1/21 + 1/13）/ 3 = 0.074

这些平均值，通称为优先向量（Priority Vector），简称 PV 值，如图 25-14 所示。

	钱 多	事 少	离家近	**权 数（优先向量)**
钱 多	**0.677**	**0.714**	**0.538**	**0.643**
事 少	**0.226**	**0.238**	**0.385**	**0.283**
离家近	**0.097**	**0.048**	**0.077**	**0.074**
总 和	**31/21**	**21/5**	**13**	

图 25-14

Step-4 于是计算出 Level-1 的权数值，如图 25-15 所示。

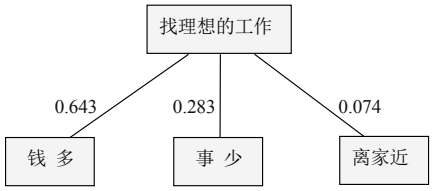


图 25-15

Step-5 开始演算 Level-2 的“钱多”权数值，如图 25-16 所示。

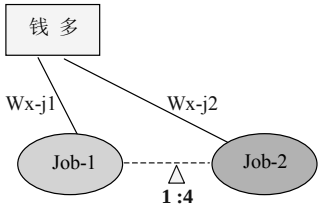


图 25-16

图 25-16 的比值为 1:4，表示 Job-2 对钱多的贡献“稍强”于 Job-1。填入表格中，如图 25-17 所示。

	Job-1	Job-2	
Job-1	**1**	**1/4**	
Job-2	**4**	**1**	

图 25-17

依据刚才的 Step-1 ~ Step-3 进行演算：

（1）计算各列的总和；（2）各个值除以该列的总和；（3）计算各列的平均值。

于是，计算出权数（即 PV 值）如图 25-18 所示。

	Job-1	Job-2	PV
Job-1	**0.20**	**0.20**	**0.20**
Job-2	**0.80**	**0.80**	**0.80**

图 25-18

Step-6 开始演算 Level-2 的“事少”权数值，如图 25-19 所示。

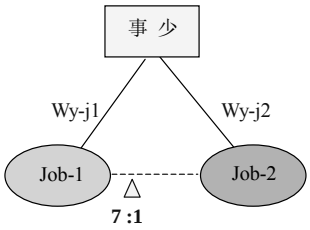


图 25-19

图 25-19 的比值为 7:1，表示 Job-1 对事少的贡献“非常强”于 Job-2。填入表格中，如图 25-20 所示。

	Job-1	Job-2	
Job-1	**1**	**7**	
Job-2	**1/7**	**1**	

图 25-20

依据刚才的 Step-1 ~ Step-3 进行演算，计算各列的总和，并且各个值除以该列的总和，然后计算各列的平均值。于是，计算出 PV 值如图 25-21 所示。

	Job-1	Job-2	PV
Job-1	**0.875**	**0.875**	**0.875**
Job-2	**0.125**	**0.125**	**0.125**

图 25-21

Step-7 开始演算 Level-2 的“离家近”权数值，如图 25-22 所示。

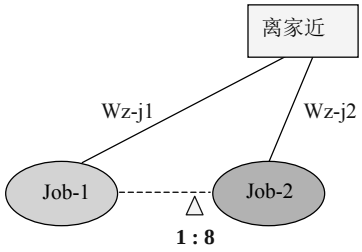


图 25-22

图 25-22 的比值为 1:8，表示 Job-2 对离家近的相对贡献强度介于“非常强”与“极强”之间。填入表格中，如图 25-23 所示。

	Job-1	Job-2	
Job-1	**1**	**1/8**	
Job-2	**8**	**1**	

图 25-23

依据刚才的 Step-1 ~ Step-3 进行演算，计算各列的总和，并且各个值除以该列的总和，然后计算各列的平均值。于是，计算出的 PV 值如图 25-24 所示。

	Job-1	Job-2	PV
Job-1	**0.111**	**0.111**	**0.111**
Job-2	**0.889**	**0.889**	**0.889**

图 25-24

于是计算出 Level-2 的权数值，如图 25-25 所示。

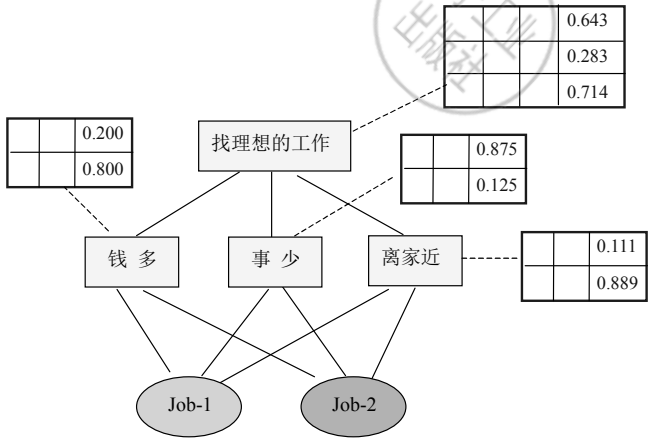


图 25-25

图 25-25 与前面的图 25-1 是一致的。

Step-8 开始进行评估。

Job-1 对“钱多”的贡献度为 0.200，而“钱多”对总目标（即“理想”）的贡献度为 0.643，所以 Job-1 通过“钱多”对总目标的贡献度为：$0.200 \times 0.643 = 0.129$。Job-1 对“事少”的贡献度为 0.875，而“事少”对总目标（即“理想”）的贡献度为 0.283，所以 Job-1 通过“事少”

对总目标的贡献度为：0.875 × 0.283 = 0.248。Job-1 对“离家近”的贡献度为 0.111，而“离家近”对总目标（即“理想”）的贡献度为 0.074，所以 Job-1 通过“离家近”对总目标的贡献度为 0.111 × 0.074 = 0.008。于是可算出 Job-1 所表现的理想度为：0.129 + 0.248 + 0.008 = 0.385。依据同样的程序，可算出 Job-2 的情形：

- Job-2 通过“钱多”对总目标的贡献度为：0.800 × 0.643 = 0.514。
- Job-2 通过“事少”对总目标的贡献度为：0.125 × 0.283 = 0.035。
- Job-2 通过“离家近”对总目标的贡献度为：0.889 × 0.074 = 0.066。

于是可算出 Job-2 所表现的理想度为：0.514 + 0.035 + 0.066 = 0.615。 两者相比，Job-2 是较理想的选择。

25.3.3 “成对比值”的一致性检验

由于“成对相比”可能会出现自我矛盾的现象而不自知，所以 AHP 方法也能检验出是否有矛盾的现象。例如图 25-26 里的比值，其中 3:1 可表示为“钱多 > 事少”。而另外 5:1，可表示为“事少 > 离家近”。依循逻辑，可推理而得：“钱多 > 离家近”。再看看 7:1，可表示为“钱多 > 离家近”，这与上述的推理是一致的，这意味着经过上述程序所计算出来的 Wx、Wy 和 Wz 权数值是一致的，并没有矛盾。

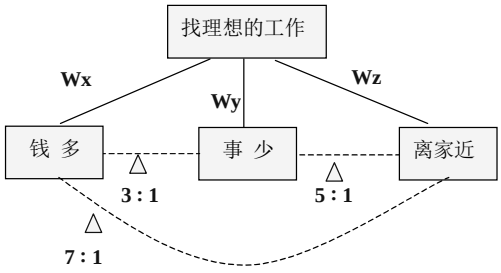


图 25-26

但是有些情况会出现不一致的矛盾现象（待会儿将举例说明）。因此，在计算每一组权数时，也需要检验其一致性。其计算步骤如下。

Step-1 基于上一节的 Step-3 所计算的总和及 PV 值，就可逐步计算并检验其一致性了。例如上一节的 Step-3 所计算的总和及 PV 值，如图 25-27 所示。

Step-2 计算最大 Eigen 值，其公式为：各行总和与各列 PV 相乘的和。于是可算出：

$$\lambda_{max} = （1.476 \times 0.643）+（4.2 \times 0.283）+（13 \times 0.074）= 3.097$$

	钱 多	事 少	离家近	PV
钱 多				**0.643**
事 少				**0.283**
离家近				**0.074**
总 和	**31/21** 等于 **（1.476）**	**21/5** 等于 **（4.2）**	**13**	

图 25-27

Step-3 计算一致性指标（Consistency Index），简称 CI，其公式为：

$$CI = (\lambda max - n) / (n - 1)$$

其中 n 值就是选择准则的个数，例如图 25-27 的 n 值为 3。所以可算出：

$$CI = (3.097 - 3) / (3 - 1) = 0.048$$

Step-4 计算一致性比率（Consistency Ratio），简称 CR，其公式为：

$$CR = CI / RI$$

其中 RI 代表随机一致性指标（Random Consistency Index）值，如下表所示：

n	1	2	3	4	5	6	7	8	9	10
RI	0	0	0.58	0.9	1.12	1.24	1.32	1.41	1.45	1.49

例如，图 25-27 的 n 值为 3，经查表可得 CI 值为 0.58。所以可算出：

$$CR = 0.048 / 0.58 = 0.083$$

Step-5 判断一致性：如果 CR 值小于 0.1，表示具有相当的一致性，所以上述例子是具有一致性的。反之，如果 CR 值大于 0.1，表示呈现显著的不一致性。例如，将上述例子更改为如图 25-28 所示。

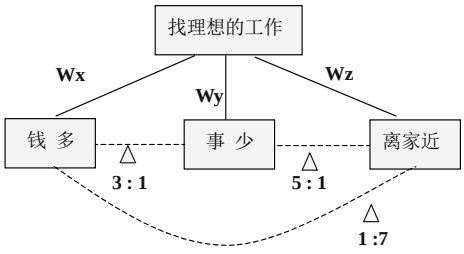


图 25-28

则计算出来的 CR 值是 2.639，远大于 0.1，呈现出明显的不一致性。因为“钱多 > 事少 > 离家近”很明显与“钱多 < 离家近”是互相矛盾的。

25.4 “AHP”示例分析与设计

25.4.1 绘制系统用例图

AHP 的用户——分析师，其主要目的是希望 AHP 程序帮他进行分析，以便做出较好的决策。所以主要的用例只有一个：AHP 决策分析。

此外，因为用例图是开发者与用户智能相遇的地方，开发者经常利用<<include>>或<<extend>>关系来表达系统特色，创造对用户的吸引力。就像客人想买中秋月饼，卖中秋月饼的商店在包装纸上画出月饼的内涵来吸引客人。于是，可设计 AHP 系统用例图如图 25-29 所示。

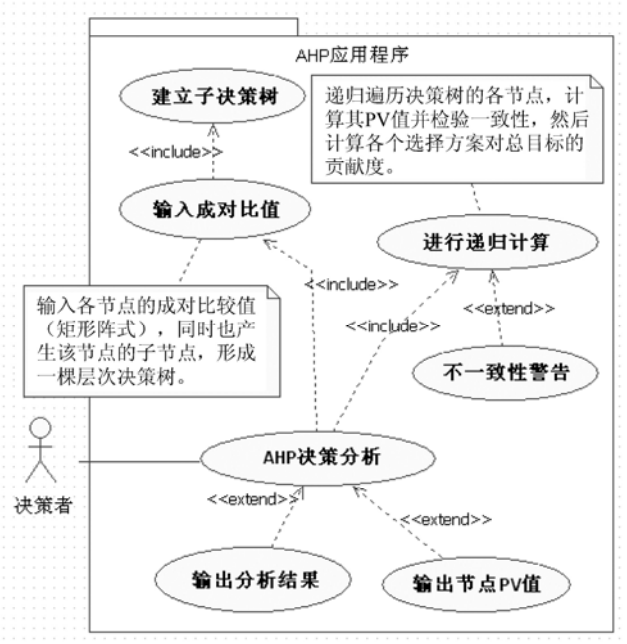


图 25-29

此图说明了系统的特色，包括：

- 输入成对比较矩阵值，同时据之创建层次决策树；树上的每一个节点（node）都含有一个矩阵。
- 递归决策树上的节点，逐一演算 PV 值及贡献度，并检验一致性。

- 可输出各节点 PV 值，以及整体贡献度。

25.4.2 绘制类图

从上述的用例叙述里，可发现几个主要的概念或术语，例如：成对比较矩阵、决策树和节点等。

于是找到 PCMatrix 和 AHPNode 两个主要类，再加上 AHP_Client 和 LList 两个辅助性类，就形成类图，如图 25-30 所示。

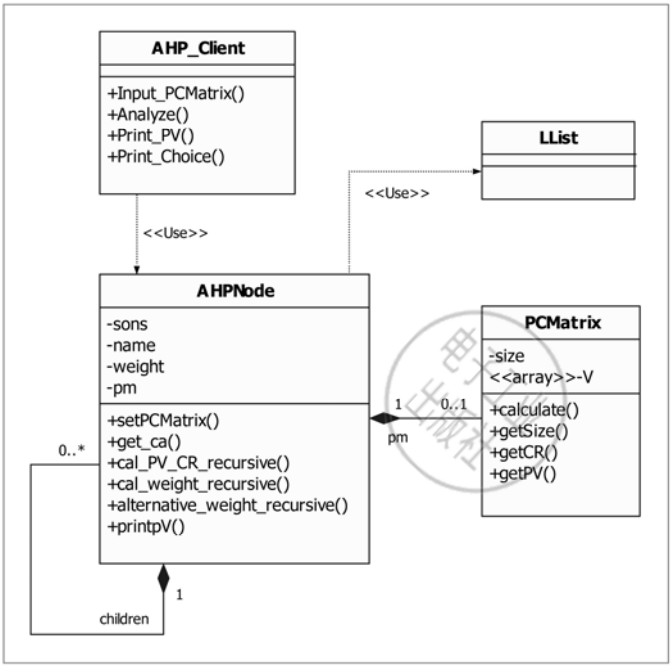


图 25-30

其中，LList 是链表（Linked List）类，用来将 AHPNode 节点串成一个树状结构。AHP_Client 是用户输入矩阵数据的地方，也是用户接口模块。PCMatrix 的 calculate()担任矩阵的运算，包括：

- 计算各列的总和，并且各个值除以该列的总和。
- 计算各列的平均值。

AHPNode 的 setPCMatrix()的任务是：

- 生成 AHPNode 对象，代表目前节点，并将矩阵存入节点里。
- 依据矩阵的大小 N，而生成 N 个子节点（也就是 AHPNode 对象），然后以 LList 对象创

建层级（父子）关系。

AHPNode 的 cal_PV_CR_recursive()负责递归树上的每一个节点，逐一演算 PV 值及贡献度，并检验一致性。cal_weight_recursive()计算各节点对总目标的贡献度。alternative_weight_recursive()负责计算各选择方案的评选结果。

25.4.3 绘制序列图

序列图是用例图与类图相遇的地方，其表达如何将 AHPNode 等对象组合出各用例图里的各项服务。

用例：“输入成对比值”和“创建子决策树”

从前面的用例图里，可看到两个用例“输入成对比值”和“创建子决策树”。现在就来看看如何以序列图表达这两个用例。由于页面宽度的限制，可将之切分为两个序列图，即图 25-31 和图 25-32。

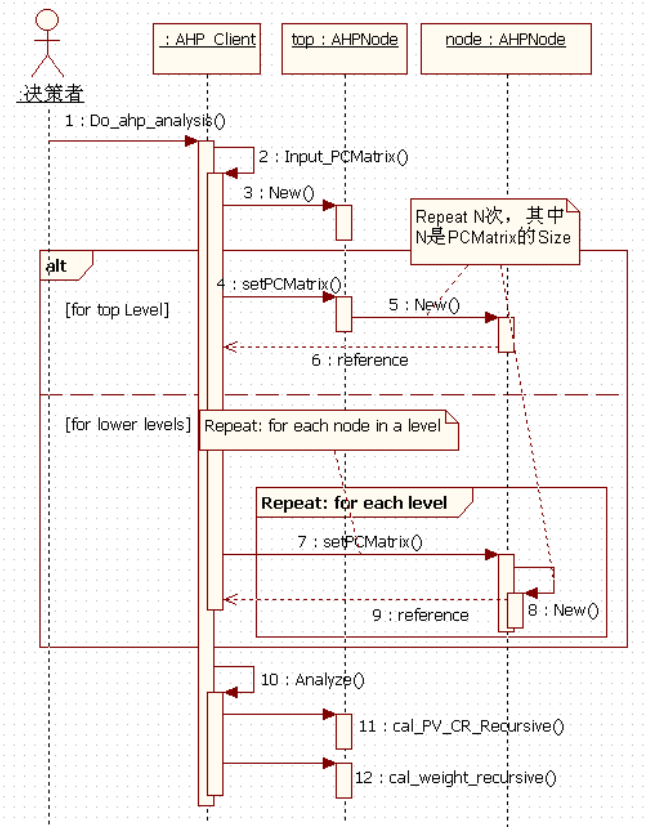


图 25-31

top 就是决策树的根（root）节点，而 node 代表其他各层级的节点。每一个节点会生成它的子节点。此外还会生成 PCMatrix 对象来储存矩阵值。

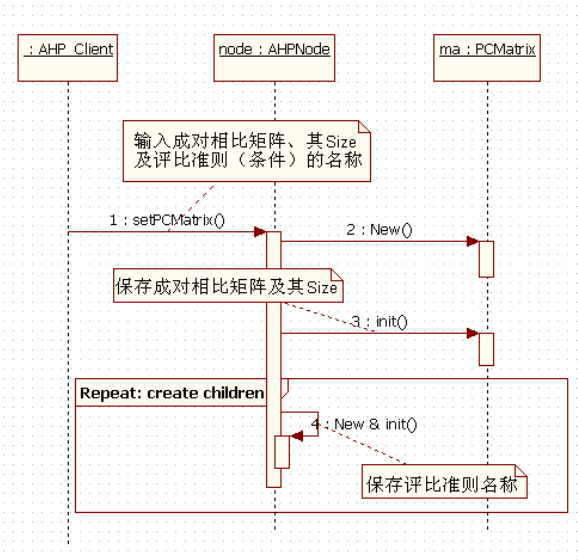


图 25-32

用例：“进行递归计算”

从上述用例图里，可看到两个用例“进行递归计算”。其包含两个递归程序，就以两个序列图来表示，分别为图 25-33 和图 25-34。

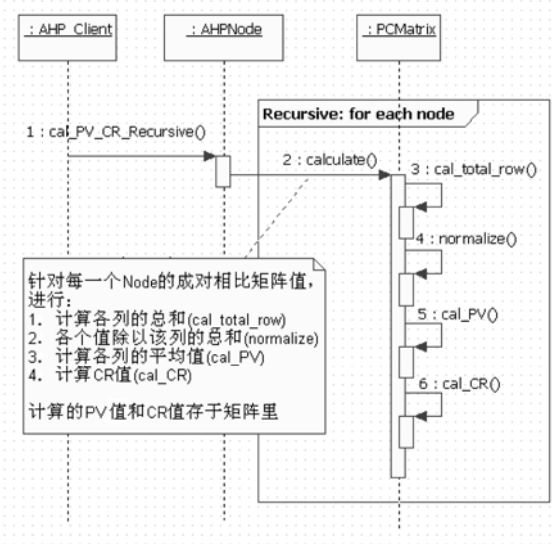


图 25-33

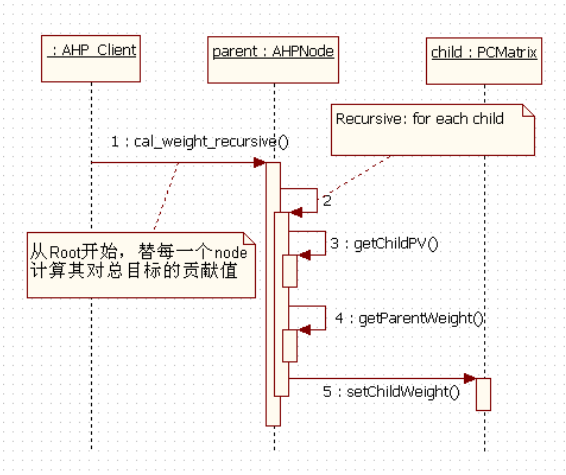


图 25-34

就 parent 节点而言，其所含的矩阵里就存有各 child 节点的 PV 值。此 parent 节点的总目标贡献度（即 weight 值）乘以某 child 节点的 PV 值，就是该 child 的总目标贡献度。

25.5 “AHP”示例的实现：使用 OOPC

25.5.1 准备决策数据

——以微软 MSDN 里的 AHP 应用为例

在此，将以微软 MSDN 的 AHP 应用数据为例。这些示例数据摘自于：

http://msdn.microsoft.com/msdnmag/issues/05/06/TestRun/default.aspx

Test Run: The Analytic Hierarchy Process -- **MSDN** Magazine, June 2005

其层次结构如图 25-35 所示。

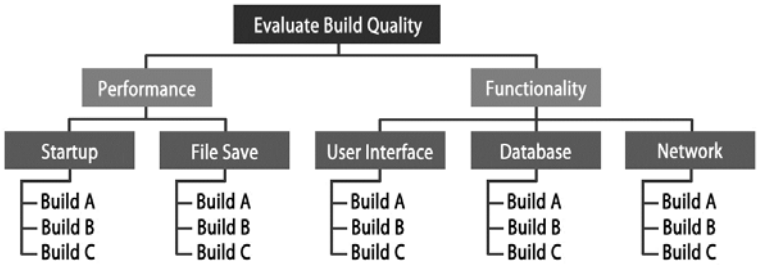


图 25-35

Top 层级的矩阵数据如图 25-36 所示。

	Performance	Functionality
Performance	1	0.250
Functionality	4	1

图 25-36

Level-1 层级的矩阵数据如图 25-37 所示。

	Startup	File Save
Startup	1	5
File Save	0.200	1

	User Interface	Database	Network
User Interface	1	3	7
Database	0.333	1	2
Network	0.143	0.500	1

图 25-37

Level-2 层级的矩阵数据如图 25-38 所示。

Startup	Build A	Build B	Build C
Build A	1	3	5
Build B	0.333	1	2
Build C	0.200	0.500	1

File Save	Build A	Build B	Build C
Build A	1	2	4
Build B	0.500	1	2
Build C	0.250	0.500	1
User Interface	**Build A**	**Build B**	**Build C**
Build A	1	1	3
Build B	1	1	2
Build C	0.333	0.500	1
Database	**Build A**	**Build B**	**Build C**
Build A	1	3	6
Build B	0.333	1	4
Build C	0.167	0.250	1
Network	**Build A**	**Build B**	**Build C**
Build A	1	4	5
Build B	0.250	1	5
Build C	0.200	0.200	1

图 25-38

分析结果如图 25-39 所示。

Performance			Functionality			Overall
	Startup	File Save	User Interface	Database	Network	
Build A	0.648	0.571	0.443	0.639	0.638	0.531
Build B	0.230	0.286	0.387	0.274	0.273	0.329
Build C	0.122	0.143	0.170	0.087	0.089	0.140

图 25-39

25.5.2 以 OOPC 编写 AHP 程序

基于上述的序列图，可以编写 Eclipse/Java 代码，其步骤如下。

Step-1 开启 TurboC 环境。

类图（即图 25-30）里有两个主要类 PCMatrix 和 AHPNode，其分别实现为 PCMatrix.java 和 AHPNode.java 代码。

Step-2 以 OOPC 编写 PCMatrix 类代码。

定义 PCMatrix 类

```
/*  pcmatrix.h   */
#include "lw_oopc.h"
#ifndef IPCM_H
#define IPCM_H

INTERFACE(IPCM)
{ void (*init)(void*, double**, int);
  void (*calculate)(void*);
  double (*getCR)(void*);
  double (*getPV)(void*, int);
  int (*getSize)(void*);
};
#endif
```

编写 PCMatrix 类代码

```
/*  pcmatrix.c  */
/*  每一个 AHPNode 对象里都含有一个 PCMatrix 对象  */
#include "stdio.h"
#include "lw_oopc.h"
#include "pcmatrix.h"

CLASS(PCMatrix)
{
  IMPLEMENTS(IPCM);   /*  支持 IPCM 接口   */
  int size;                          double** V;
  void (*cal_total_row)(void*);    void (*normalize)(void*);
  void (*cal_PV)(void*);           void (*cal_CR)(void*);
  double (*calLambda)(void*);      double (*calCI)(void*);
  double (*calCR)(void *);
};

/*  给予初始数组值，以及数组大小   */
```

```
static void init(void* t, double** A, int sz)
{ int i, j;
  PCMatrix* cthis = (PCMatrix*)t;
  cthis->size = sz;
  cthis->V = (double**)malloc((sz+1) * sizeof(double *));
  for(i=0; i < sz+1; i++)
  cthis->V[i] = (double*)malloc((sz+2) * sizeof(double));
  for(j=0; j<sz; j++)
     for(i=0; i<sz; i++)
       cthis->V[j][i] = A[j][i];
}
/*  进行 AHP 分析与计算  */
static void calculate(void* t)
{ PCMatrix* cthis = (PCMatrix*) t;
  cthis->cal_total_row(cthis);             cthis->normalize(cthis);
  cthis->cal_PV(cthis);
  cthis->cal_CR(cthis);
}
static double getCR(void* t)
{ PCMatrix* cthis = (PCMatrix*) t;
  return cthis->V[2][cthis->size+1];
}
static double getPV(void* t, int k)
{ PCMatrix* cthis = (PCMatrix*) t;
  return cthis->V[k][cthis->size];
}
static int getSize(void* t)
{ PCMatrix* cthis = (PCMatrix*) t;
  return cthis->size;
}
/*-------------------------------------------------------*/
/*  为 calculate()所调用  */
static void cal_total_row(void* t)
{ double ss;
  int sz,i,j;
  PCMatrix* cthis = (PCMatrix*) t;
  sz = cthis->size;
  for (j = 0; j < sz; j++)
  {  ss = 0;
     for (i = 0; i < sz; i++)  ss += cthis->V[i][j];
       cthis->V[sz][j] = ss;   }
}
static void normalize(void* t)
{ int sz,i,j;
  PCMatrix* cthis = (PCMatrix*) t;
  sz = cthis->size;
  for (j = 0; j < sz; j++)
     for (i = 0; i < sz; i++)
       cthis->V[i][j] = cthis->V[i][j] / cthis->V[sz][j];
}
static void cal_PV(void* t)
{ int sz, i, j;    double ss;
  PCMatrix* cthis = (PCMatrix*)t;
  sz = cthis->size;
  for (i = 0; i < sz; i++)
    {  ss = 0;
       for (j = 0; j < sz; j++)    ss += cthis->V[i][j];
         cthis->V[i][sz] = ss/sz;    }
}
```

```
static void cal_CR(void* t)
{ int sz;
  PCMatrix* cthis = (PCMatrix*)t;
  sz = cthis->size;
  cthis->V[0][sz+1]= cthis->calLambda(cthis);
  cthis->V[1][sz+1] = cthis->calCI(cthis);
  cthis->V[2][sz+1] = cthis->calCR(cthis);
}
static double calLambda(void* t)
{ int sz, k;
  double lambda = 0.0;
  PCMatrix* cthis = (PCMatrix*)t;
  sz = cthis->size;
  for (k = 0; k < sz; k++)
     lambda += cthis->V[sz][k] * cthis->V[k][sz];
  return lambda;
}
static double calCI(void* t)
{ int sz;   double ci;
  PCMatrix* cthis = (PCMatrix*)t;
  sz = cthis->size;   ci = (cthis->V[0][sz+1] - sz)/ (sz-1.0);
  return ci;
}
static double calCR(void *t)
{ PCMatrix* cthis = (PCMatrix*)t;
  int sz = cthis->size;
  if(sz <= 2) return 0.0;
  else return cthis->calCI(cthis)/getRI(sz);
}
/*-------------------------------------------------------------*/
static double getRI( int size)
{ switch (size)
  {  case 0: return 0.00;  case 1: return 0.00;  case 2: return 0.00;
     case 3: return 0.58;  case 4: return 0.90;  case 5: return 1.12;
     case 6: return 1.24;  case 7: return 1.32;  case 8: return 1.41;
     case 9: return 1.45;  case 10: return 1.49;  default: return 1.5;   }
}
CTOR(PCMatrix)
  FUNCTION_SETTING(IPCM.init, init)
  FUNCTION_SETTING(IPCM.calculate, calculate)
  FUNCTION_SETTING(IPCM.getCR, getCR)
  FUNCTION_SETTING(IPCM.getPV, getPV)
  FUNCTION_SETTING(IPCM.getSize, getSize)
  FUNCTION_SETTING(cal_total_row, cal_total_row)
  FUNCTION_SETTING(normalize, normalize)
  FUNCTION_SETTING(cal_PV, cal_PV)
  FUNCTION_SETTING(cal_CR, cal_CR)
  FUNCTION_SETTING(calLambda, calLambda)
  FUNCTION_SETTING(calCI, calCI)
  FUNCTION_SETTING(calCR, calCR)
END_CTOR
```

Step-3 以 OOPC 编写 AHPNode 类代码。

定义 AHPNode 类

```
/*  iahpnode.h   */
#include "lw_oopc.h"
```

```
#ifndef IAHP_H
#define IAHP_H

INTERFACE(IAHP)
{
  void (*init)(void*, char*);
  void (*setPCMatrix)(void*, double**, int, char**);
  int (*size)(void*);
  void* (*get_ca)(void*, int);
  void (*cal_PV_CR_recursive)(void*);
  void (*cal_weight_recursive)(void*);
  void (*goal_initial)(void*);
  void (*alternative_wieght_recursive)(void* t, char* na);
  void (*printPV)(void*);
  void (*go)(void*);
};
#endif
```

编写 AHPNode 类代码

```
/* ahpnode.c   */
/*  一个 AHPNode 对象就是 AHP 树状结构里的节点  */
#include "stdio.h"
#include "string.h"
#include "llist.h"
#include "pcmatrix.h"
#include "iahpnode.h"
double temp = 0.0;

CLASS(AHPNode)
{
  IMPLEMENTS(IAHP);
  IColl* sons;
  IPCM* pm;
  char name[120];
  double weight;
 };

extern void* LListNew();
extern void* PCMatrixNew();
static void init(void* t, char* na)
{ AHPNode* cthis = (AHPNode*)t;
  strcpy(cthis->name, na);           cthis->pm = NULL;
  cthis->sons = (IColl*) LListNew();    (cthis->sons)->init(cthis->sons);
}
/*  每一个 AHPNode 对象里都含有一个 PCMatrix 对象  */
static void setPCMatrix(void* t, double** A, int n, char** caNames)
{ int k;    AHPNode* pa;
  AHPNode* cthis = (AHPNode*)t;         IColl* psons = cthis->sons;
  cthis->pm = (IPCM*) PCMatrixNew();    (cthis->pm)->init(cthis->pm, A, n);
  for(k = 0; k < n; k++ )
  {  pa = (AHPNode*)AHPNodeNew();   pa->IAHP.init(pa, caNames[k]);
     psons->add(psons, pa);      }
}
static int size(void* t)
{ AHPNode* cthis = (AHPNode*)t;
```

```
  return (cthis->pm)->getSize(cthis->pm);
}
static void* get_ca(void*t, int k)
{ AHPNode* cthis = (AHPNode*)t;
  IColl* pc = cthis->sons;
  return pc->get(pc, k);
}
static void printPV(void* t)
{ AHPNode* cthis = (AHPNode*)t;
  IPCM* ma = cthis->pm;
  printf("%s\n", cthis->name);
  printf("   %7.3f\n", ma->getPV(ma, 0));
  printf("   %7.3f\n", ma->getPV(ma, 1));
  printf("   %7.3f\n", ma->getPV(ma, 2));
}
/*  处理 AHP tree 上的每一个 AHPNode 对象，其都含有一个 PCMatrix 对象，
    对 PCMatrix 进行 AHP 分析计算   */
static void cal_PV_CR_recursive(void* t)
{ AHPNode* cthis = (AHPNode*)t;    AHPNode* nd;    int sz, k;
  IPCM* ma = cthis->pm;
  if(ma == NULL) return;
  ma->calculate(ma);
  printf("%s's CR=>%7.3f\n", cthis->name, ma->getCR(ma));
  if(ma->getCR(ma) > 0.1) /* is Not Consistency */
     printf("%s's CR is NOT Consistent! %7.4f!\n", cthis->name,
            ma->getCR(ma));
  sz = ma->getSize(ma);
  for(k=0; k < sz; k++ )
  {  nd = (AHPNode*)cthis->IAHP.get_ca(cthis, k);
     nd->IAHP.cal_PV_CR_recursive(nd);   }
}
/*  处理 AHP tree 上的每一个 AHPNode 对象，其都含有一个 PCMatrix 对象，
    对 PCMatrix 进行 weights 计算   */
static void cal_weight_recursive(void* t)
{ AHPNode* cthis = (AHPNode*)t;      AHPNode* nd;      int sz, k;
  IPCM* ma = cthis->pm;
  if(ma == NULL) return;
  sz = ma->getSize(ma);
  for(k=0; k < sz; k++ )
  {  nd = (AHPNode*)cthis->IAHP.get_ca(cthis, k);
     nd->weight = cthis->weight * ma->getPV(ma, k);
     nd->IAHP.cal_weight_recursive(nd);  }
  }
static void goal_initial(void* t)
{ AHPNode* cthis = (AHPNode*)t;
  cthis->weight = 1.0;
}
static void go(void* t)
{ int sz, k;     double a;    AHPNode* cthis = (AHPNode*)t;
  IPCM* ma = cthis->pm;
  ma->calculate(ma);     a = ma->getCR(ma);
  printf("CR = %7.3f\n", a);
  sz = ma->getSize(ma);
  for(k=0; k<sz; k++)
  {  a = ma->getPV(ma, k);    printf("PV[%d] = %7.3f\n", k, a);    }
}
```

```
/*  处理AHP tree上的Leaf  AHPNode对象，进行weight加总   */
static void alternative_wieght_recursive(void* t, char* na)
{ AHPNode* cthis = (AHPNode*)t;    int sz, k;     AHPNode* nd;
  IPCM* ma = cthis->pm;
  if(ma == NULL)
  {  if(!strcmp(cthis->name, na))
       temp += cthis->weight;
     sz =0;  }
  else
     sz = ma->getSize(ma);
  for(k=0; k < sz; k++ )
     {  nd = (AHPNode*)cthis->IAHP.get_ca(cthis, k);
        d->IAHP.alternative_wieght_recursive(nd, na);      }
}
CTOR(AHPNode)
  FUNCTION_SETTING(IAHP.init, init)
  FUNCTION_SETTING(IAHP.setPCMatrix, setPCMatrix)
  FUNCTION_SETTING(IAHP.size, size)
  FUNCTION_SETTING(IAHP.get_ca, get_ca)
  FUNCTION_SETTING(IAHP.cal_PV_CR_recursive, cal_PV_CR_recursive)
  FUNCTION_SETTING(IAHP.cal_weight_recursive, cal_weight_recursive)
  FUNCTION_SETTING(IAHP.goal_initial, goal_initial)
  FUNCTION_SETTING(IAHP.alternative_wieght_recursive,
  alternative_wieght_recursive)
  FUNCTION_SETTING(IAHP.printPV, printPV)
  FUNCTION_SETTING(IAHP.go, go)
END_CTOR
```

Step-4 编写主程序。

```
/*  main_ahp.c  */
#include "stdio.h"
#include "string.h"
#include "lw_oopc.h"
#include "pcmatrix.h"
#include "iahpnode.h"
extern void* AHPNodeNew();
extern double temp;
IAHP *goal, *ca0, *ca1, *ca00, *ca01, *ca10, *ca11, *ca12;

void Input_PCMatrix()
{ double a;    int i;    char ** caNames;    double ** A;
  /*  创建AHP tree 的 Root  */
  goal = (IAHP*)AHPNodeNew();
  goal->init(goal, "Evaluate Build Quality");    goal->goal_initial(goal);
  caNames = (char**)malloc(2 * sizeof(char *));
  for(i=0; i < 2; i++)
    caNames[i] = (char*)malloc(40 * sizeof(char));
  strcpy(caNames[0], "Performance");
  strcpy(caNames[1], "Functionality");
  /*   准备各PCMatrix的初期值  */
  A = (double**)malloc(2 * sizeof(double *));
  for(i=0; i < 2; i++)
  {  A[i] = (double*)malloc(2 * sizeof(double));    A[i][i] = 1.0;  }
  A[0][1] = 1.0/4.0;    A[1][0] = 4.0;
  goal->setPCMatrix(goal,A,2, caNames);
```

```
  free(caNames);  free(A);
/* ---------------------------------------- */
  caNames = (char**)malloc(2 * sizeof(char *));
  for(i=0; i < 2; i++)
    caNames[i] = (char*)malloc(40 * sizeof(char));
  strcpy(caNames[0], "StartUp");  strcpy(caNames[1], "File Save");
  A = (double**)malloc(2 * sizeof(double *));
  for(i=0; i < 2; i++)
    { A[i] = (double*)malloc(2 * sizeof(double));     A[i][i] = 1.0;   }
  A[0][1] = 5.0;  A[1][0] = 1.0/5.0;
  ca0 = (IAHP*)goal->get_ca(goal, 0);
  ca0->setPCMatrix(ca0,A,2, caNames);
  free(caNames);   free(A);
/*  ------------------------------------  */
  caNames = (char**)malloc(3 * sizeof(char *));
  for(i=0; i < 3; i++)
    caNames[i] = (char*)malloc(40 * sizeof(char));
  strcpy(caNames[0], "User Interface");  strcpy(caNames[1], "Database");
  strcpy(caNames[2], "Network");
  A = (double**)malloc(3 * sizeof(double *));
  for(i=0; i < 3; i++)
    { A[i] = (double*)malloc(3 * sizeof(double));    A[i][i] = 1.0;   }
  A[0][1] = 3.0;      A[1][0] = 1.0/3.0;      A[0][2] = 7.0;
  A[2][0] = 1.0/7.0;   A[1][2] = 2.0;        A[2][1] = 1.0/2.0;
  ca1 = (IAHP*)goal->get_ca(goal, 1);
  ca1->setPCMatrix(ca1,A,3, caNames);
/* -------------  Level 2  ---------------- */
  strcpy(caNames[0],  "Build  A");        strcpy(caNames[1],  "Build  B");
  strcpy(caNames[2], "Build C");
  A[0][1] = 3.0;     A[1][0] = 1.0/3.0;     A[0][2] = 5.0;
  A[2][0] = 1.0/5.0;  A[1][2] = 2.0;       A[2][1] = 1.0/2.0;
  ca00 = (IAHP*)ca0->get_ca(ca0, 0);
  ca00->setPCMatrix(ca00,A,3, caNames);
/* ---------------------------------------- */
  A[0][1] = 2.0;     A[1][0] = 1.0/2.0;     A[0][2] = 4.0;
  A[2][0] = 1.0/4.0;  A[1][2] = 2.0;       A[2][1] = 1.0/2.0;
  ca01 = (IAHP*)ca0->get_ca(ca0, 1);
  ca01->setPCMatrix(ca01,A,3, caNames);
/* ---------------------------------------- */
  A[0][1] = 1.0;     A[1][0] = 1.0/1.0;     A[0][2] = 3.0;
  A[2][0] = 1.0/3.0;  A[1][2] = 2.0;       A[2][1] = 1.0/2.0;
  ca10 = (IAHP*)ca1->get_ca(ca1, 0);
  ca10->setPCMatrix(ca10,A,3, caNames);
/* ---------------------------------------- */
  A[0][1] = 3.0;     A[1][0] = 1.0/3.0;     A[0][2] = 6.0;
  A[2][0] = 1.0/6.0;  A[1][2] = 4.0;       A[2][1] = 1.0/4.0;
  ca11 = (IAHP*)ca1->get_ca(ca1, 1);
  ca11->setPCMatrix(ca11,A,3, caNames);
/* ---------------------------------------- */
  A[0][1] = 4.0;     A[1][0] = 1.0/4.0;     A[0][2] = 5.0;
  A[2][0] = 1.0/5.0;  A[1][2] = 5.0;       A[2][1] = 1.0/5.0;
  ca12 = (IAHP*)ca1->get_ca(ca1, 2);
 ca12->setPCMatrix(ca12,A,3, caNames);
}
/* ---------------------------------------- */
/*   开始分析   */
```

```
void Analyze()
{ goal->cal_PV_CR_recursive(goal);
  goal->cal_weight_recursive(goal);
}
/*   打印   */
void Print_PV()
{ ca00->printPV(ca00);     ca01->printPV(ca01);     ca10->printPV(ca10);
  ca11->printPV(ca11);     ca12->printPV(ca12);
}
void Print_Choice()
{ printf("Overall\n");
  temp = 0.0;    goal->alternative_wieght_recursive(goal, "Build A");
  printf("Build A=%7.3f\n", temp);
  temp = 0.0;    goal->alternative_wieght_recursive(goal, "Build B");
  printf("Build B=%7.3f\n", temp);
  temp = 0.0;    goal->alternative_wieght_recursive(goal, "Build C");
  printf("Build C=%7.3f\n", temp);
}
void ahp()
{ Input_PCMatrix();    Analyze();    Print_PV();  Print_Choice();  }
void main()
{ ahp();    getchar();   }
```

Step-5 以 TurboC 执行上述代码，其输出结果如图 25-40 所示。

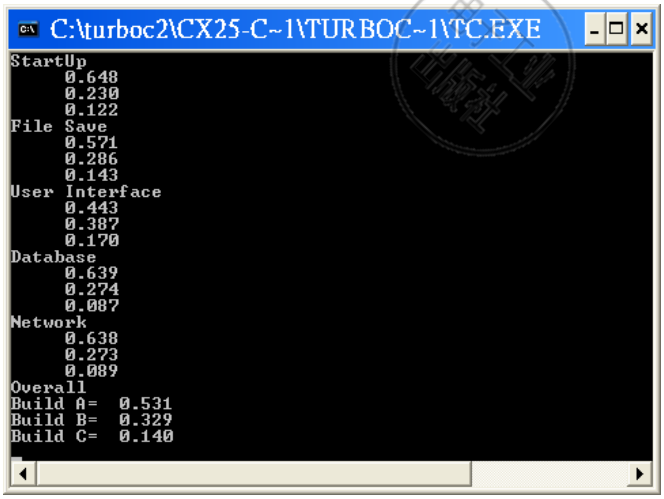


图 25-40

其结果与 MSDN 上的结果（如 25.5.1 节所示）是一致的。

第 26 章　UML+OOPC 实用示例之四

——以硬件半加器（Half_adder）仿真程序为例

——使用 Win32/VC++编译环境

26.1 什么是半加器

嵌入式系统本来就是 C 语言的天下，OOPC 只是在 ANSI-C 上添加了约 20 行宏而已，不仅毫不妨碍 C 语言的威力，还让 C 语言加上完美的结构，更适合嵌入式系统的天职：让软件与硬件紧密结合。本章就以半加器（Half Adder）为例说明如何流畅地模拟硬件模块的行为，深刻体会 OOPC 在嵌入式系统开发上的优雅风味。

在硬件电路上，半加器可说是最简单的单元了。现以 UML 模型图加以说明，如图 26-1 所示。

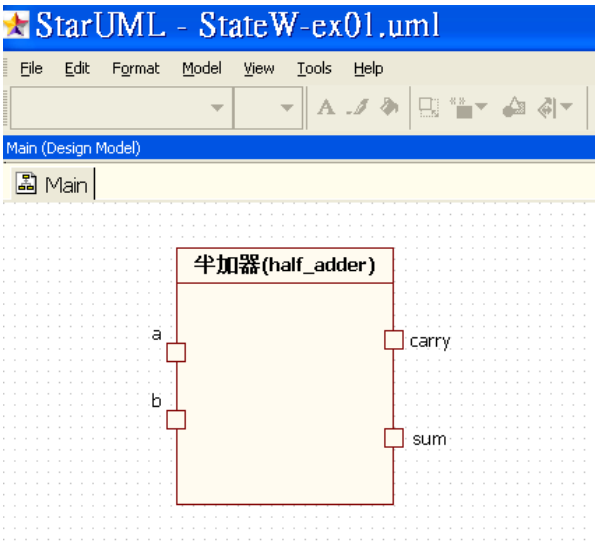


图 26-1

半加器的功能很少，只是进行两个位（bit）的相加而已。当你把两个位值分别送入 a 和 b 两个端口（Port）时，半加器就将这两个位值相加，其总和就从 sum 端口送出，并且将进位值由 carry 端口送出。两个半加器可以组成为全加器（Full Adder），如图 26-2 所示。

全加器可以做两个多位二进制值的相加，例如 011 和 110 两个二进制数的加法运算，如下：

```
  0 1 1
+ 1 1 0
-----------
1 0 0 1
```

相加的结果是二进制的 1001。

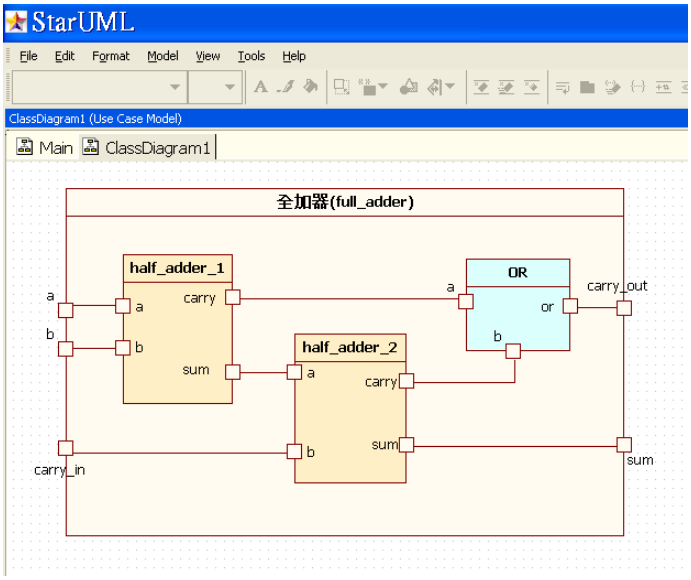


图 26-2

26.2　设计一个“位计算器”

26.2.1　以软件模拟硬件的意义

在嵌入式系统开发过程中，以软件来模拟硬件是一个非常重要的技巧，也愈来愈流行。其原因是，在传统上，软件人员等待硬件先完成，才开始创建软件；但是硬件又必须不断与软件沟通才能检验硬件的正确性，因此常常处于蛋生鸡、鸡生蛋的困境里。也就是说，在兼具软件和硬件的复杂系统里，通常等待硬件先完成之后，才会进行软件的细节设计。因为软件人员依赖实际的硬件环境来测试其软件。但是这种测试用的真实硬件环境，其创建是很复杂的。这种依赖关系，常导致项目的一筹莫展。

如果我们能够运用软件来建构硬件的模型，借由硬件的模型测试软件，就不必依赖真实硬件环境了，也不必等待硬件模块先创建完成，就能及早开发软件。

当我们以硬件的模型来测试软件，软件就能提前与硬件同步开发了，这使我们能够在创建硬件之前，先检验硬件与软件的交互接口。如此就化解了上述所面临的传统困境。

传统上，常藉由 Verilog 或 VHDL 模拟环境来检验硬件功能；然而当系统较为复杂时，此种仿真的速度就变得很慢。当仿真速度太慢时，常以硬件仿真（Hardware Emulation）来替代之，但是硬件仿真模拟环境又很昂贵。于是促使了 System C 的问世，以 System C 来创建硬件的模型，并进行软硬件整合仿真，化解了上述的困境。类似地，OOPC 也很适合像 System C 一样来模拟硬件模块，模拟过程中可以不断修正硬件模块的接口及其内部设计，直到稳定下来了，才实际进入硬件的制造程序，这样可以降低许多成本。

26.2.2 设计单位计算器的操作画面

软件半加器加上操作画面，就成为一个单位的计算器了。当其执行时，进入 Ready 状态，出现如图 26-3 所示的界面，等待您输入 0 或 1：

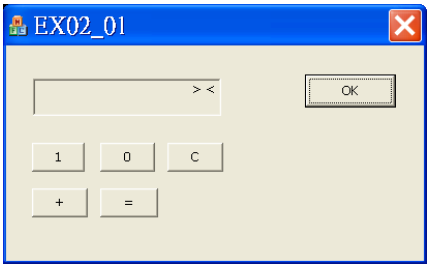


图 26-3

当您按下<1>键时，进入 FirstDigitInputed 状态，如图 26-4 所示，等待您输入<+>键。

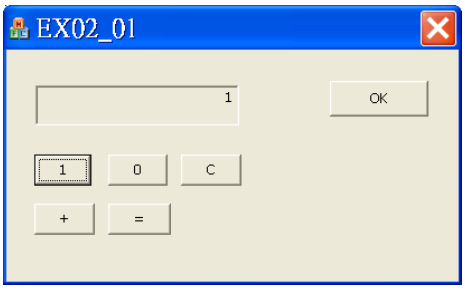


图 26-4

当您按下<+>键时，就会转移到 PlusInputed 状态，如图 26-5 所示，等待您输入第 2 位数字。

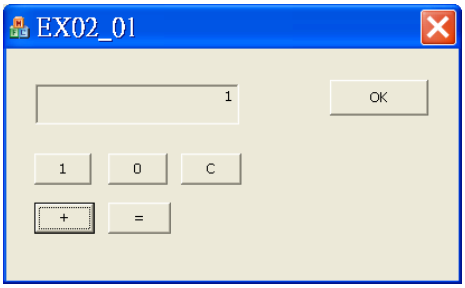


图 26-5

当您按下<0>键时，就会进入 SecondDigitInputed 状态，如图 26-6 所示，等待您输入<=>键。

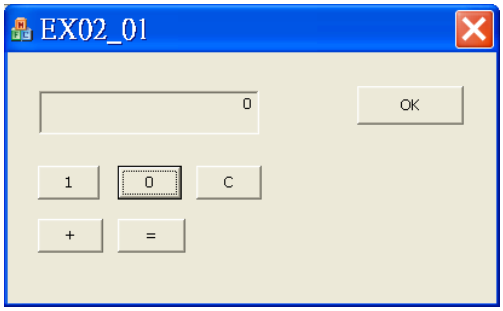


图 26-6

当您按下<=>键时，就会转移到 Calculating 状态。一旦进入 Calculating 状态，就立即执行两个位值的相加，并显示出结果。这就是模拟半加器所计算的结果了。结果输出于画面上，如图 26-7 所示。

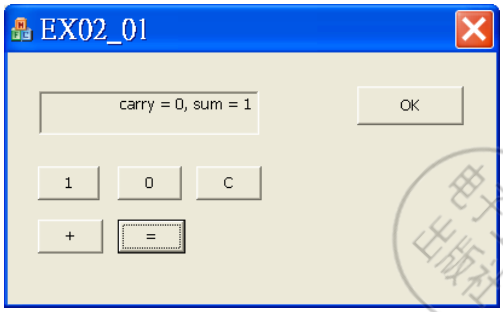


图 26-7

26.2.3　设计单位计算器的 UML 状态图

我们为设计单位计算器创建一个 UML 状态图，如图 26-8 所示。

在这个单位计算器的操作上，可分为两个基本状态 Ready 和 Working。当它处于 Ready 状态，并且接到您的信息 EvDigitPress 时，就转移到 Working 状态。此时若接到您的信息 EvCPress，就转回到 Ready 状态。

Working 状态里又分为 4 个小状态，这称为复合状态（Composit State）。当进入到 Working 状态时，必定是处于该 4 个状态之一。

图 26-8 的“　”代表“Initial”或“Default”。它意味着：当第 1 次进入 Working 状态时，必先进入 FirstDigitInputed 状态，这个状态就称为默认状态（Default State）。

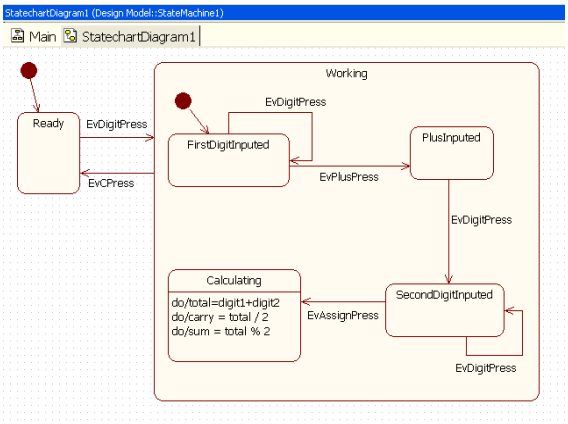


图 26-8

26.3 实现位计算器：使用 OOPC

Step-1 使用 VC++环境，创建一个 Win32 项目，取名为 Win32_ex01，如图 26-9 所示。

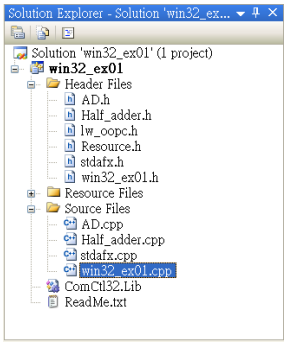


图 26-9

Step-2 这个例子里最主要的领域概念是半加器和位计算器。接着将这两个概念对应到 OOPC 的类：Half_adder 和 AD。

定义 Half_adder 类

```
/*  Half_adder.h   */
#ifndef HA_H
#define HA_H
#include "lw_oopc.h"

CLASS(Half_adder)
{   int a, b;
int carry, sum;
    void (*init)(void*);
    void (*start_thread)(void*);
};
#endif
```

编写 Half_adder 类代码

```
/*  Half_adder.c    */
#include "StdAfx.h"
#include "stdio.h"
#include "half_adder.h"

static void init(void* t)
{  Half_adder* cthis = (Half_adder*)t;
   cthis->a = -1;   cthis->b = -1;
}

/*  采用Multi-thread技巧, 由独立的thread执行Half_adder的运算  */
static void start_thread(void* t)
{  Half_adder* cthis = (Half_adder*)t;
   int temp;
   temp = cthis->b;
   DWORD wait_time, base_time, curr_time;
   while(1)
   {     wait_time = 100;
         base_time = GetTickCount();
         /*---------   waiting ---------------------------- */
         do {  curr_time =  GetTickCount() - base_time;  }
         while (curr_time < wait_time);
         /*-------------------------------------------------- */
         if(temp != cthis->b)
             { cthis->carry = cthis->a & cthis->b;
                 cthis->sum = cthis->a ^ cthis->b;   }
         temp = cthis->b;
}}
CTOR(Half_adder)
   FUNCTION_SETTING(init, init)
   FUNCTION_SETTING(start_thread, start_thread)
END_CTOR
```

定义 AD（位计算器）类

```
/*  AD.h   */
#ifndef RECORDER_H
#define RECORDER_H
#include "lw_oopc.h"
#include "Half_adder.h"

CLASS(AD)
{   Half_adder* adder;
    int digit_1, digit_2, carry, sum, state, d;
    void (*init)(void*);
void (*EvDigitPress)(void*, int);
    void (*EvPlusPress)(void*);
    void (*EvAssignPress)(void*);
    void (*EvCPress)(void*);
    void (*go_state_Ready)(void*);
    void (*go_state_First)(void*);
void (*go_state_Plus)(void*);
    void (*go_state_Second)(void*);
    void (*go_state_Cal)(void*);
  };
#endif
```

编写 AD 类代码

```
/*  AD.c    */
#include "StdAfx.h"
#include "stdio.h"
#include "AD.h"
#include "Half_adder.h"

extern void* Half_adderNew();
HANDLE hThread1;
HANDLE hThread2;
Half_adder* m_adder;
extern void signal_to_UI(LPCWSTR);

/*  生成新的 thread 去执行 Half_adder 对象  */
static DWORD WINAPI ThreadProc1(LPVOID lpParameter)
   {  m_adder->start_thread(m_adder);
      return 0;
   }
static void createThread1()
   {  DWORD threadId;
      hThread1 = CreateThread(NULL,0,ThreadProc1,NULL,0,&threadId);
   }
static DWORD WINAPI ThreadProc2(LPVOID lpParameter)
   { /* Reserved */
     return 0;
   }
static void createThread2()
   {  DWORD threadId;
     hThread2 = CreateThread(NULL,0,ThreadProc2,NULL,0,&threadId);
   }
static void init(void *t)
{   AD* cthis = (AD*)t;
   cthis->adder = (Half_adder*)Half_adderNew();
   cthis->adder->init(cthis->adder);
m_adder = cthis->adder;
   createThread1();
   cthis->go_state_Ready(cthis);
}
/* 处理 EvDigitPress 事件  */
static void EvDigitPress(void* t, int d)
{   AD* cthis = (AD*)t;
   cthis->d = d;
   if(cthis->state == 0)         cthis->go_state_First(cthis);
else if(cthis->state == 1)   cthis->go_state_First(cthis);
else if(cthis->state == 2)   cthis->go_state_Second(cthis);
   else if(cthis->state == 3)   cthis->go_state_Second(cthis);
}
/* 处理 EvPlus 事件  */
static void EvPlusPress(void* t)
{   AD* cthis = (AD*)t;
   if(cthis->state == 1)    cthis->go_state_Plus(cthis);
}
/* 处理 EvAssignPress 事件  */
static void EvAssignPress(void* t)
{   AD* cthis = (AD*)t;
   if(cthis->state == 3)     cthis->go_state_Cal(cthis);
}
/* 处理 EvCPress 事件  */
```

```
static void EvCPress(void* t)
{   AD* cthis = (AD*)t;
    cthis->go_state_Ready(cthis);
}
static void go_state_Ready(void*t)
{  AD* cthis = (AD*)t;
  cthis->state = 0;
  cthis->digit_1 = 0;   cthis->digit_2 = 0;
  cthis->adder->init(cthis->adder);
  signal_to_UI(_T("><"));
}
static void go_state_First(void*t)
{
 AD* cthis = (AD*)t;
   cthis->state = 1;
   if(cthis->d == 1)      signal_to_UI(_T("1"));
   else              signal_to_UI(_T("0"));
   cthis->digit_1 = cthis->d;
}
static void go_state_Plus(void*t)
{   AD* cthis = (AD*)t;
    cthis->state = 2;
}
static void go_state_Second(void*t)
{  AD* cthis = (AD*)t;
   cthis->state = 3;
 if(cthis->d == 1)      signal_to_UI(_T("1"));
   else               signal_to_UI(_T("0"));
 cthis->digit_2 = cthis->d;
}
static void go_state_Cal(void*t)
{   AD* cthis = (AD*)t;
    cthis->state = 4;
    cthis->adder->a = cthis->digit_1;
    cthis->adder->b = cthis->digit_2;
 /*  -------------- wait ---------------- */
    /*  等待 Half_adder 完成运算  */
 DWORD base_time, curr_time;
    base_time = GetTickCount();
    do {  curr_time =  GetTickCount() - base_time;     }
    while (curr_time < 200);
/*----------------------------------------- */
    cthis->carry = cthis->adder->carry;
    cthis->sum = cthis->adder->sum;
    if(cthis->carry == 1 & cthis->sum == 1)        signal_to_UI(_T("[11]"));
else if(cthis->carry == 1 & cthis->sum == 0)    signal_to_UI(_T("[10]"));
    else if(cthis->carry == 0 & cthis->sum == 1)    signal_to_UI(_T("[01]"));
    else                                  signal_to_UI(_T("[00]"));
}
    /*  也可以采用下述指令输出字符串  */
    /*  TCHAR sss[100];
_stprintf (sss, _T("X  is: %d "), wmId);
MessageBox(NULL,sss,_T(""), MB_OK | MB_ICONINFORMATION);
    */
CTOR(AD)
   FUNCTION_SETTING(init, init)
   FUNCTION_SETTING(EvDigitPress, EvDigitPress)
   FUNCTION_SETTING(EvPlusPress, EvPlusPress)
   FUNCTION_SETTING(EvAssignPress, EvAssignPress)
```

```
  FUNCTION_SETTING(EvCPress, EvCPress)
  FUNCTION_SETTING(go_state_Ready, go_state_Ready)
  FUNCTION_SETTING(go_state_First, go_state_First)
  FUNCTION_SETTING(go_state_Plus, go_state_Plus)
  FUNCTION_SETTING(go_state_Second, go_state_Second)
  FUNCTION_SETTING(go_state_Cal, go_state_Cal)
END_CTOR
```

Step-3 编写主程序的代码。

```
/* win32_ex01.cpp  */
#include "stdafx.h"
#include "windows.h"
#include "tchar.h"
#include <commctrl.h>
#include "AD.h"
#include "win32_ex01.h"
#define MAX_LOADSTRING 100

extern void* ADNew();
AD* ad;
HINSTANCE hInst;
TCHAR szTitle[MAX_LOADSTRING];
TCHAR szWindowClass[MAX_LOADSTRING];
HWND hWindow;

int mState;   HWND hWndToolbar;   BOOL bMouseDown;   POINTS ptsEnd;
static char g_szClassName[] = "MyWindowClass";
ATOM             MyRegisterClass(HINSTANCE hInstance);
BOOL             InitInstance(HINSTANCE, int);
LRESULT CALLBACK   WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK   About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                 LPTSTR  lpCmdLine, int nCmdShow)
{   UNREFERENCED_PARAMETER(hPrevInstance);
   UNREFERENCED_PARAMETER(lpCmdLine);
     MSG msg;   HACCEL hAccelTable;

LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
   LoadString(hInstance, IDC_WIN32_EX01, szWindowClass, MAX_LOADSTRING);
   MyRegisterClass(hInstance);
   if (!InitInstance (hInstance, nCmdShow))   {  return FALSE;   }
   hAccelTable                =                LoadAccelerators(hInstance,
MAKEINTRESOURCE(IDC_WIN32_EX01));
   while (GetMessage(&msg, NULL, 0, 0))
   {  if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
      {   TranslateMessage(&msg);  DispatchMessage(&msg);  }
   }
   return (int) msg.wParam;
}
ATOM MyRegisterClass(HINSTANCE hInstance)
{
   WNDCLASSEX wcex;
   wcex.cbSize = sizeof(WNDCLASSEX);
   wcex.style          = CS_HREDRAW | CS_VREDRAW;
   wcex.lpfnWndProc    = WndProc;
   wcex.cbClsExtra       = 0;     wcex.cbWndExtra       = 0;
   wcex.hInstance       = hInstance;
```

```
   wcex.hIcon                                          =   LoadIcon(hInstance,
MAKEINTRESOURCE(IDI_WIN32_EX01));
   wcex.hCursor       = LoadCursor(NULL, IDC_ARROW);
   wcex.hbrBackground   = (HBRUSH)(COLOR_WINDOW+1);
   wcex.lpszMenuName   = MAKEINTRESOURCE(IDC_WIN32_EX01);
   wcex.lpszClassName   = szWindowClass;
   wcex.hIconSm                                    =    LoadIcon(wcex.hInstance,
MAKEINTRESOURCE(IDI_SMALL));
   return RegisterClassEx(&wcex);
}
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{  hInst = hInstance;
  hWindow = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
     CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
  if (!hWindow)  {   return FALSE;   }
  INITCOMMONCONTROLSEX InitCtrlEx;
InitCtrlEx.dwSize = sizeof(INITCOMMONCONTROLSEX);
initCtrlEx.dwICC  = ICC_BAR_CLASSES;
InitCommonControlsEx(&InitCtrlEx);
ShowWindow(hWindow, nCmdShow);
  UpdateWindow(hWindow);
  return TRUE;
}
HWND hWndButton3, hWndButton4, hWndButton5, hWndButton6;
HWND hWndButton7, hWndButton8, hWndButton9;
void signal_to_UI(LPCWSTR ss)  {   SetWindowText(hWndButton8, ss); }

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{    int wmId, wmEvent, ret;
   PAINTSTRUCT ps; HDC hdc;
   switch (message)
   {  case WM_CREATE:{
HINSTANCE hInstance = (HINSTANCE)GetWindowLong(hWnd, GWL_HINSTANCE);
/*  生成 Button  */
         hWndButton3= CreateWindowEx( 0, _T("BUTTON"), _T("1"),
          WS_VISIBLE | WS_CHILD, 20, 130, 80, 35,
          hWnd, (HMENU) IDB_BUTTON3, hInstance, NULL);

          hWndButton4= CreateWindowEx(0, _T("BUTTON"), _T("0"),
          WS_VISIBLE | WS_CHILD,  120, 130, 80, 35,
          hWnd, (HMENU) IDB_BUTTON4, hInstance, NULL);

          hWndButton5= CreateWindowEx(0, _T("BUTTON"), _T("C"),
          WS_VISIBLE | WS_CHILD, 220, 130, 80, 35,
          hWnd, (HMENU) IDB_BUTTON5, hInstance, NULL);

          hWndButton6= CreateWindowEx( 0, _T("BUTTON"), _T("+"),
          WS_VISIBLE | WS_CHILD,  20, 200, 80, 35,
          hWnd, (HMENU) IDB_BUTTON6, hInstance, NULL);

          hWndButton7= CreateWindowEx(0,  _T("BUTTON"), _T("="),
          WS_VISIBLE | WS_CHILD,  120, 200, 80, 35,
          hWnd, (HMENU) IDB_BUTTON7, hInstance, NULL);

          hWndButton8= CreateWindowEx(WS_EX_CLIENTEDGE | WS_EX_RIGHT,
          _T("EDIT"), _T("0"),  WS_VISIBLE | WS_CHILD,
          20, 60, 260, 35, hWnd, (HMENU) IDB_BUTTON8, hInstance, NULL);

hWndButton9= CreateWindowEx(WS_EX_CLIENTEDGE,
```

```
          _T("BUTTON"), _T("Exit"), WS_VISIBLE | WS_CHILD,
          350, 60, 80, 35, hWnd, (HMENU) IDB_BUTTON9, hInstance, NULL);
          mState = 0;
          /* ------------------------------------------------ */
          ad = (AD*)ADNew();   /*   生成 AD 对象  */
ad->init(ad);
        }
          break;
    case WM_COMMAND:
      wmId    = LOWORD(wParam);    wmEvent = HIWORD(wParam);
      switch (wmId)
      {   case IDB_BUTTON3:  /* 按下<1> */
                ad->EvDigitPress(ad, 1);    break;
          case IDB_BUTTON4:  /* 按下<0>  */
              ad->EvDigitPress(ad, 0);   break;
            case IDB_BUTTON5:  /* 按下<C>  */
              ad->EvCPress(ad);  break;
          case IDB_BUTTON6:  /* 按下<+>  */
              ad->EvPlusPress(ad);  break;
          case IDB_BUTTON7: /* 按下<=>*/
              ad->EvAssignPress(ad);  break;
          case IDB_BUTTON9:  /* 按下< Exit>  */
             DestroyWindow(hWindow);
           break;
      case IDM_ABOUT:
           DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
          break;
      case IDM_EXIT:
          DestroyWindow(hWindow);   break;
      default:    return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:     hdc = BeginPaint(hWnd, &ps);  EndPaint(hWnd, &ps);
break;
   case WM_CLOSE:     DestroyWindow(hWnd);  break;
   case WM_DESTROY:  PostQuitMessage(0);   break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{  UNREFERENCED_PARAMETER(lParam);
switch (message)
   {   case WM_INITDIALOG:    return (INT_PTR)TRUE;
       case WM_COMMAND:
         if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
           {              EndDialog(hDlg,    LOWORD(wParam));       return
(INT_PTR)TRUE;    }
          break;
   }
   return (INT_PTR)FALSE;
}
```

Step-4 开始执行。

此系统执行时，AD 进入 Ready 状态，等待您输入 0 或 1 数字。当您按下<1>键时，主程序就执行指令：

```
            case IDB_BUTTON3:  /* 1 */
                ad->EvDigitPress(ad, 1);
                break;
```

就发出 EvDigitPress 事件给 AD 对象，就执行 AD 的 EvDigitPress()函数：

```
    static void EvDigitPress(void* t, int d)
       {
         AD* cthis = (AD*)t;
         cthis->d = d;
         if(cthis->state == 0)
          cthis->go_state_First(cthis);
            ……
       }
```

此刻 AD 正处于 Ready 状态（0 值），于是 AD 得到 d 值，然后进入 FirstDigitInputed 状态，如图 26-10 所示，等待您输入<+>键。

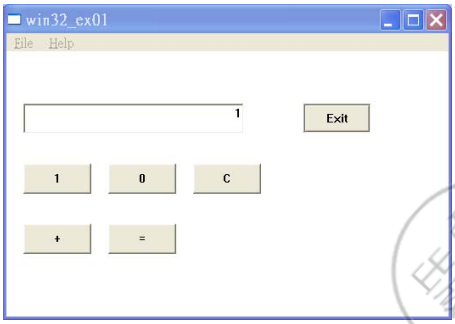


图 26-10

当您按下<+>键时，主程序就发出 EvPlusPress 事件给 AD。此刻 AD 正处于 FirstDigitInputed 状态，于是就转移到 PlusInputed 状态，如图 26-11 所示，等待您输入第 2 位数字。当您按下<0>键时，就发出 EvDigitPress 事件给 AD 对象，就执行 AD 的 EvDigitPress()函数，此时 AD 得到 d 值，然后进入 SecondDigitInputed 状态，如图 26-11 所示，等待您输入<=>键。

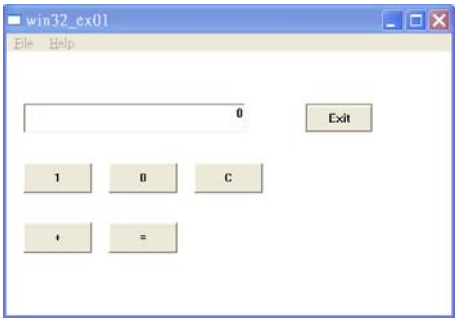


图 26-11

当您按下<=>键时，就会发出 EvAssignPress 事件给 AD。此刻 AD 正处于 SecondDigit-

Inputed 状态，于是就转移到 Calculating 状态。一旦进入 Calculating 状态，就立即执行：

```
 static void go_state_Cal(void*t)
{
 AD* cthis = (AD*)t;
 cthis->state = 4;
 cthis->adder->a = cthis->digit_1;
 cthis->adder->b = cthis->digit_2;

 /* -------------- wait ---------------- */
 DWORD base_time, curr_time;
 base_time = GetTickCount();
 do {  curr_time =  GetTickCount() - base_time;    }
 while (curr_time < 200);
 /*---------------------------------------- */
 cthis->carry = cthis->adder->carry;
 cthis->sum = cthis->adder->sum;

}
```

首先，把 digit_1 和 digit_2 两个位值传给半加器，半加器会自动进行运算。继续执行到指令 `cthis->carry = cthis->adder->carry;`就从半加器的 carry 端口取得计算结果。其结果输出于画面上，如图 26-12 所示。

当您按下<=>键时，就会转移到 Calculating 状态。一旦进入 Calculating 状态，就立即执行两个位值的相加，并显示出结果。这就是模拟半加器所计算的结果了。

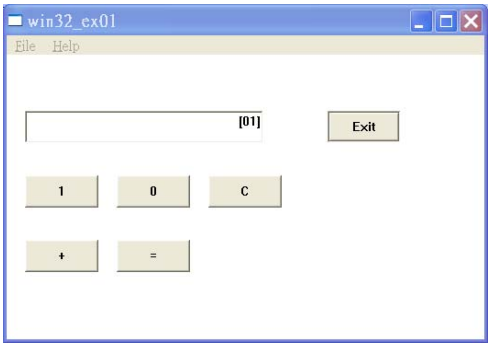


图 26-12

第 27 章　替 Keil C51 黄袍加身

——基于μVision3 IDE 环境里的评估

27.1 以 200 Bytes 代价换得优雅架构

——以 200 Bytes 代价换得优雅架构

OOPC 是“Object-Oriented Programming in C”的简写，而 LW_OOPC 是轻量（Light-weight）级 OOPC 的意思。这是针对嵌入式环境而设计的轻薄短小型的 OOPC 语言。只要你的嵌入式 C 程序允许增加 200 Bytes 的大小，就能运用面向对象机制，撰写美好的软件架构。如果你的程序很小，小到连 200 Bytes 都要计较的话，也可以采用更精简的做法，它只需要付出几十 Byte 的代价就能拥有基本的面向对象机制了。所以从今天开始，只要加入 lw_oopc_kc.h 标头文件就能编写面向对象 Keil C 程序代码了。

其中 lw_oopc_kc.h 头文件内容是由 MISOO 团队开发，其著作权和知识产权为本书作者高焕堂 所拥有，仅供本书读者学习使用，或者应用于非营利的学术研究上。任何营利性、商业性使用，必须先取得本书作者的同意。相关信息请看 LW_OOPC 语言官方网站 *www.misoo.com.tw* 或 *misoo.vm999.com*。

近年来，C 语言书籍的销售扶摇直上，起因于嵌入式（Embedded）软件应用愈来愈广，如数码家电、手机、数字化汽车等。而嵌入式软件开发所使用的语言中，C 语言仍约占 80%之多。

由于嵌入式软件应用愈来愈广，软件质量大大影响了数码产品的稳定性和可靠性，因此如何提升 C 程序的简洁性、易读性及重复使用性，乃是当今软件业的热门话题。例如世界知名的麦肯锡（McKinsey）顾问公司，在 2006 年的报告（“Getting Better Software to Manufactured Products”）呼吁嵌入式软件业必须积极提升其系统分析及架构设计的技术能力，才能解决软件含量愈来愈多的数码产品的信赖性问题。

如何解决上述问题呢？其方向已经很清楚了，就是：让 C 语言与面向对象程序设计（Object- Oriented Programming，简称 OOP）技术结合。就像当今的其他主流计算机语言（如 VB.Net、C#、Java 等）一般。由于当今的世界标准系统分析与架构设计的建模语言 UML，也是基于面向对象技术而发展出来的。一旦 C 语言与面向对象技术结合了，也就与 UML 结合了，便能逐渐提升系统分析与设计的质量了。

也许你会问到：在 1986 年时，贝尔（Bell）实验室已经将 C 语言与面向对象技术结合成为 C++语言了，为何还需要 OOPC 呢？其答案是：C++语言有些贪心，将整套的面向对象技术包括进去，导致 C++的效率远比单纯的 C 语言慢了许多。由于嵌入式软件所处的环境里，硬件资源大都极为有限，对程序执行效率斤斤计较，所以在今天嵌入式软件开发上，使用最广的仍是 C 语言。

而本书所介绍的面向对象 C 语言并不是一个新的语言，它只运用单纯 C 语言的宏（Macro）技巧，实现了面向对象的基本技术，让系统分析与设计阶段的 UML 模型能与 C 程序紧密对应，以提升 C 程序的质量。此外，这些宏在编译阶段就被翻译为单纯的 C 程序代码了，仍然保持其单纯 C 的高效率，符合嵌入式软件环境的需要。

俗语说，“天下没有白吃的午餐”。要在 Keil C 里加上美好的面向对象机制，需要付出多大代价呢？答案是：不到 1 KB。以下就以μVision3 IDE 的评估来说明这是一项很划算的策略。

27.2 3 种弹性又高雅的写法

嵌入式系统的花样很多，由于硬件资源有限，所以必须斤斤计较、精确计算。如果能搭配上弹性的选择，就能顺利地找出最佳的解决方案。本书所介绍的 LW_OOPC 语言提供 3 种形式给你选择，各形式所需付出的代价并不相同，其提供的方便程度也有所不同。这 3 种形式就是：

27.2.1 动态型（Dynamic，昵称为豪华型）

这种类型会使用到 malloc()函数，在执行期间调用 malloc()函数来取分配对象（Object）的内存空间，所以需要连接到 malloc()库函数（Library Function），使得程序大小（Size）也增加了。从图 27-1 可以看出，在动态形式里，你在 C 程序里写入第 1 个类时，你的程序大小会增加约 600 Bytes。如果你继续加入其他类的话，每增加一个类，都会增加约 60 Bytes。一般而言，如果系统环境里搭配有 RTOS（Real-Time Operating Sytstem）的话，通常不会斤斤计较 600 Bytes 的空间，所以大多会采取这种动态形式。

27.2.2 静态型（Static，昵称为标准型）

这种类型避免使用 malloc()函数。其对象并不是在程序执行期间才去分配空间的。而是与一般的 int、char 型态的变量一样，在编译期间就留下内存空间了，不需要在执行期通过 malloc()函数来取得内存空间，所以这种“静态”用法的执行速度会快些，程序也比较小。从图 27-1 可以看出，在静态形式里，你在 C 程序里写入第 1 个类时，你的程序大小会增加约 160 Bytes。如果你继续加入其他类的话，每增加一个类，都会增加约 50 Bytes。一般而言，如果系统环境里并没搭配 RTOS，而且对内存斤斤计较的话，适宜采取这种静态形式。

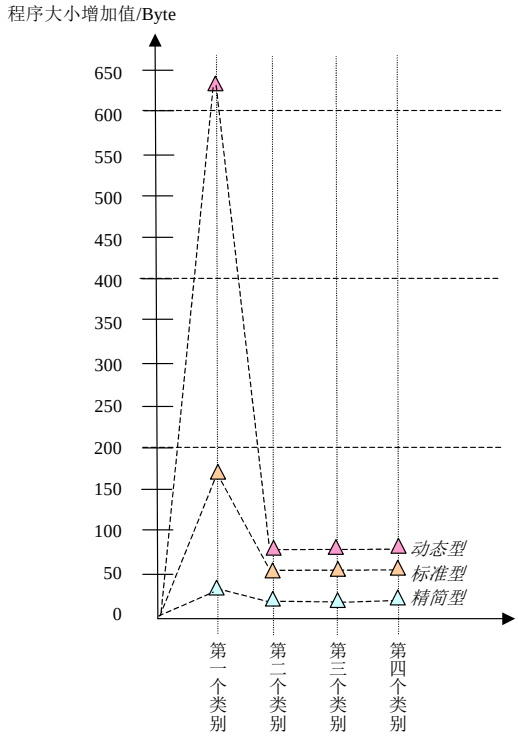


图 27-1

27.2.3 纯粹静态型（Pure Static，昵称为精简型）

这种类型避免使用指针（Pointer）来调用函数。这种“纯粹静态”用法的执行速度会更快，程序也更小。从图 27-1 可以看出，在纯粹静态形式里，你在 C 程序里写入第 1 个类时，你的程序大小会增加约 25 Bytes。如果你继续加入其他类的话，每增加一个类，都会增加约 15 Bytes。如果系统对内存非常计较的话，才采取这种形式。

这 3 形式可提供给你弹性的选择。此外，你并不一定要把整个程序全部对象化，而可以精确计算出你的可用内存容量，然后依据图 27-1 的评估而算出可以编写多少个类（Class）。由于 LW_OOPC 是由 C 语言的宏（Macro）所建立的，在编译之前就已经全部被转为一般非对象化的 C 程序了。所以类能与类之外的一般函数共存于你的程序里。因此你能对哪些函数该纳入类里，而哪些函数不需要，做出最好的安排。

27.3 静态（Static）型写法及其评估

现以μVision3 IDE 上所附的 BLINKY（\keil\C51\EXAMPLES\BLINKY\）范例作为评估

的基础；μVision3 所附的 BLINKY.C 源代码如下：

```
/* EX27-01.C  */
/* BLINKY.C - LED Flasher for the Keil MCBx51 Evaluation Board with 80C51 device*/
#include <REG51F.H>
void wait (void) {                /* wait function */
  ;                             /* only to delay for LED flashes */
}

void main (void) {
  unsigned int i;                  /* Delay var */
  unsigned char j;                 /* LED var */

  while (1) {                     /* Loop forever */
    for (j=0x01; j< 0x80; j<<=1) {    /* Blink LED 0, 1, 2, 3, 4, 5, 6 */
      P1 = j;                    /* Output to LED Port */
      for (i = 0; i < 10000; i++) {     /* Delay for 10000 Counts */
       wait ();                   /* call wait function */
      }
    }

    for (j=0x80; j> 0x01; j>>=1) {     /* Blink LED 6, 5, 4, 3, 2, 1 */
      P1 = j;                    /* Output to LED Port */
      for (i = 0; i < 10000; i++) {      /* Delay for 10000 Counts */
       wait ();                   /* call wait function */
      }
    }
  }
}
```

在以μVision3 IDE 上编译及连接结果，程序的大小为 82 Bytes，其信息如图 27-2 所示。

```
Build target 'Simulator'
compiling BLINKY.C...
linking...
Program Size: data=9.0 xdata=0 code=82
"BLINKY" - 0 Error(s), 0 Warning(s).
```

图 27-2

现在，利用 OOPC 的机制，定义一个 LED 类，于是将上述的 BLANKY.C 源代码改写为有类（Class）的美好结构如下：

```
/* EX27-02.C  */
#include <REG51F.H>
#define LW_OOPC_STATIC
#include "lw_oopc_kc.h"

void wait (void) {
      ;
    }

CLASS(LED)
```

```
{
  void (*run)();
};

static void run(){
   unsigned int i;  unsigned char j;
   for (j=0x01; j< 0x80; j<<=1)  {
       P1 = j;
       for (i = 0; i < 10000; i++)  {
          wait ();
       }
   }
   for (j=0x80; j> 0x01; j>>=1)  {
     P1 = j;
      for (i = 0; i < 10000; i++)  {
        wait ();
     }
   }
}

CTOR(LED)
  FUNCTION_SETTING(run, run)
END_CTOR
/* ---------------------------------------------- */
void main (void)  {
   LED le;
   LEDSetting(&le);
   while (1) {
      le.run();
   }
}
```

在μVision3 IDE 上编译及连接之后，其可执行文件的大小增加为：

```
Program Size: data=15.0 xdata=0 code=247
```

其增加了 165 Bytes。编写第 1 个类的代价比较高，但后续类需要的代价则会较低。当你看到这个 OOPC 程序时，可能对 CLASS、CTOR 等宏很好奇。不过，在本章里，只把焦点放在 OOPC 机制对程序大小的影响上，等到后续各章才会对这些宏逐一说明。

为了测量第 2 个类对程序大小的影响，我们把上述范例加以扩充，加了一个一般函数，先测量此程序的大小；之后再将此函数纳入第 2 个类里，再测量程序的大小，就行了。现在就来加入一个 show()函数如下：

```
/* EX27-03.C  */
#include <REG51F.H>
#define LW_OOPC_STATIC
#include "lw_oopc_kc.h"

void wait (void)  {
      ;
    }

CLASS(LED)
{
```

```
   void (*run)();
};

static void run(){
    unsigned int i;  unsigned char j;
   for (j=0x01; j< 0x80; j<<=1)  {
      P1 = j;
      for (i = 0; i < 10000; i++)  {
          wait ();
       }
   }
   for (j=0x80; j> 0x01; j>>=1)  {
      P1 = j;
      for (i = 0; i < 10000; i++)  {
         wait ();
     }
}
}

CTOR(LED)
   FUNCTION_SETTING(run, run)
END_CTOR
/* ------------------------------------ */
char NUMB[10] = {0xC0, 0xf9, 0xa4, 0xb0, 0x99, 0x92, 0x83, 0xf8, 0x80, 0x98};
static void show(){
    unsigned int x, i;
 for (x=0; x < 10; x++)  {
       P0 = NUMB[x];
       for (i = 0; i < 10000; i++)  {
          wait ();
       }
   }
 }
/* ------------------------------------ */
void main (void)  {
   LED le;
   LEDSetting(&le);

   while (1) {
      le.run();
      show();
   }
}
```

这个 show()函数将 NUM[]数组的值逐一送往 P0。如果 P0 连接到共阳级（Common Anode）7 段 LED 显示器的话，显示器会循环出现 0~9 的阿拉伯数字。现在暂时不管 show()函数的细节，先把焦点放在程序大小的评估上。这个程序在μVision3 IDE 上在编译及连接之后，可执行文件的大小为：

```
Program Size: data=25.0 xdata=0 code=430
```

这比上一个程序增加了 183 Bytes，这不是由增加类引起的，而是由增加的 show()函数所影响的。我们先以此程序大小为准，然后将 show()函数纳入类里，再测量其大小，就可以看出新增类会付出多少代价了。现在就将 show()函数纳入如下新类里：

```
/* EX27-04.C  */
#include <REG51F.H>
#define LW_OOPC_STATIC
#include "lw_oopc_kc.h"

void wait (void)  {
      ;
    }

CLASS(LED)
{
  void (*run)();
};

static void run(){
    unsigned int i;   unsigned char j;
   for (j=0x01; j< 0x80; j<<=1)  {
      P1 = j;
      for (i = 0; i < 10000; i++)  {
         wait ();
      }
   }
   for (j=0x80; j> 0x01; j>>=1)  {
     P1 = j;
     for (i = 0; i < 10000; i++)  {
        wait ();
     }
   }
}

CTOR(LED)
  FUNCTION_SETTING(run, run)
END_CTOR
/* ------------------------------------- */
char NUMB[10] = {0xC0, 0xf9, 0xa4, 0xb0, 0x99, 0x92, 0x83, 0xf8, 0x80, 0x98};

CLASS(SEG)
{
  void (*show)();
};

static void show(){
unsigned int x, i;
    for (x=0; x < 10; x++)  {
      P0 = NUMB[x];
      for (i = 0; i < 10000; i++)  {
         wait ();
      }
   }
 }

CTOR(SEG)
  FUNCTION_SETTING(show, show)
END_CTOR
/* ------------------------------------- */
void main (void)  {
   LED le;          SEG sg;
   LEDSetting(&le);    SEGSetting(&sg);
   while (1) {
```

```
      le.run();
      sg.show();
   }
}
```

这个程序在μVision3 IDE 上编译及连接之后，得到可执行文件的大小为：

```
Program Size: data=28.0 xdata=0 code=477
```

其增加了 47 Bytes。这是加入第 2 个类 SEG 的代价。接着，我们通过新增一个 Controller 类来继续评估加入第 3 个类时对程序大小的影响。

这里先简单说明 Controller 类的特性。在上一个程序里，整个 while()大循环摆在 main()函数里，这意味着 main()是整个程序的控制中心。我们可以将这个控制中心移到一个新类里，让 main()变成一个驱动者（Drive），这样可促进 Controller 类的重用（Reuse）机会。尤其当 Controller 类较为复杂时，其重用价值就更加显著了。现在再把焦点放在程序大小的评估上，于是新增一个 Controller 类如下：

```
/* EX27-05.C  */
#include <REG51F.H>
#define LW_OOPC_STATIC
#include "lw_oopc_kc.h"

void wait (void)  {
      ;
    }

CLASS(LED)
{
  void (*run)();
};
/* ---------------------------------------------*/
static void run(){
    unsigned int i;
    unsigned char j;

    for (j=0x01; j< 0x80; j<<=1)  {
       P1 = j;
       for (i = 0; i < 10000; i++)  {
          wait ();
       }
   }

   for (j=0x80; j> 0x01; j>>=1)  {
      P1 = j;
      for (i = 0; i < 10000; i++)  {
         wait ();
     }
   }
}

CTOR(LED)
  FUNCTION_SETTING(run, run)
END_CTOR
/* ------------------------------------ */
char NUMB[10] = {0xC0, 0xf9, 0xa4, 0xb0, 0x99, 0x92, 0x83, 0xf8, 0x80, 0x98};
```

```
CLASS(SEG)
{
  void (*show)();
};

static void show(){
    unsigned int x, i;

    for (x=0; x < 10; x++)  {
       P0 = NUMB[x];
       for (i = 0; i < 10000; i++)  {
          wait ();
       }
   }
 }

CTOR(SEG)
  FUNCTION_SETTING(show, show)
END_CTOR
/* ---------------------------------------- */
CLASS(Controller)
{
   void (*doing)();
};

static void doing()
{
      LED le;          SEG sg;
   LEDSetting(&le);   SEGSetting(&sg);

   while (1) {
      le.run();
      sg.show();
   }
}

CTOR(Controller)
  FUNCTION_SETTING(doing, doing);
END_CTOR
/* ---------------------------------------- */
void main (void)  {
  Controller ctrl;
  ControllerSetting(&ctrl);
  ctrl.doing();
}
```

这个程序在μVision3 IDE 上在编译及连接之后，得到的可执行文件的大小为：

```
Program Size: data=34.0 xdata=0 code=527
```

其增加了 50 Bytes，这是加入第 3 个类（Controller）的代价。接着，我们继续评估加入第 4 个类时对程序大小的影响。在上一个程序里，新增的是主控型的 Controller 类，现在加入一个简单类看看，如下代码所示：

```
/* EX27-06.C  */
#include <REG51F.H>
```

```
#define LW_OOPC_STATIC
#include "lw_oopc_kc.h"

void wait (void)  {
      ;
    }

CLASS(LED)
{
   void (*run)();
};

static void run(){
    unsigned int i;
    unsigned char j;

    for (j=0x01; j< 0x80; j<<=1)  {
       P1 = j;
       for (i = 0; i < 10000; i++)  {
          wait ();
       }
   }

   for (j=0x80; j> 0x01; j>>=1)  {
      P1 = j;
      for (i = 0; i < 10000; i++)  {
        wait ();
     }
   }
}

CTOR(LED)
   FUNCTION_SETTING(run, run)
END_CTOR
/* ------------------------------------ */

char NUMB[10] = {0xC0, 0xf9, 0xa4, 0xb0, 0x99, 0x92, 0x83, 0xf8, 0x80, 0x98};

CLASS(SEG)
{
   void (*show)();
};

static void show(){
    unsigned int x, i;

    for (x=0; x < 10; x++)  {
       P0 = NUMB[x];
       for (i = 0; i < 10000; i++)  {
          wait ();
       }
   }
 }

CTOR(SEG)
   FUNCTION_SETTING(show, show)
END_CTOR
/* ------------------------------------ */
```

```
CLASS(LED2)
{
   void (*run2)();
};

static void run2(){
  P3 = 0xaa;
}

CTOR(LED2)
   FUNCTION_SETTING(run2, run2)
END_CTOR

/* ---------------------------------------- */
CLASS(Controller)
{
    void (*doing)();
};

static void doing()
{
LED le;          SEG sg;            LED2 le2;
LEDSetting(&le);   SEGSetting(&sg);        LED2Setting(&le2);

    while (1) {
       le2.run2();
        le.run();
       sg.show();
    }
}

CTOR(Controller)
   FUNCTION_SETTING(doing, doing);
END_CTOR
/* ---------------------------------------- */
void main (void)  {
   Controller ctrl;
   ControllerSetting(&ctrl);
   ctrl.doing();
}
```

Controller 类担任总指挥的角色，它指挥了 LED、SEG 和 LED2 等 3 个类。再把焦点放在程序大小的评估上。这个程序在μVision3 IDE 上在编译及连接之后，得到的可执行文件的大小为：

```
Program Size: data=37.0 xdata=0 code=581
```

其增加了 54 Bytes，这是增加一个阳春类的代价。以上总共加了 4 个类，除了第 1 个类付出比较高的代价（165 Bytes）之外，后续各类都约增加 50 Bytes 而已。如图 27-3 所示。

从刚才的 EX27-03 里可以看到，LED 类能与一般的 show()函数共存于一个程序里，这意味着，你可以视你的内存资源的宽裕情形而决定写入多少个类，并非全部类化（即对象化）不可。所以程序大小与执行速度并不是决定要不要采用 LW_OOPC 类机制的关键因素。反而，如何通过 LW_OOPC 改善嵌入式程序架构，以提升系统的稳定性和可靠性才是采用 LW_OOPC 语言的最主要原因。

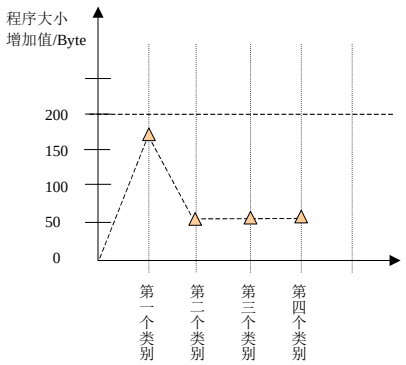


图 27-3

27.4 纯粹静态（Pure Static）型写法及其评估

精简型与标准型之间只有一点点差别而已，就是标准型有个像 LEDSetting()这样的函数，它采取指针方式设定类的函数，而精简型则不提供这样的函数，程序员必须自行设定类的函数。这样可以节省约 150 Bytes 左右。我们再拿μVision3 IDE 上所附的 BLINKY（\keil\C51\EXAMPLES\BLINKY\）范例作为评估的基础；在上一节的 EX27-01 范例就是μVision3 所附的 BLINKY.C 的源代码。上节已经测得 EX27-01 范例程序的大小为 82 Bytes。现在以精简形式的写法，撰写第一个类如下：

```
/* EX27-07.C */
/* BLINKY.C - LED Flasher for the Keil MCBx51 Evaluation Board with 80C51 device*/
#include <REG51F.H>
#define LW_OOPC_PURE_STATIC
#include "lw_oopc_kc.h"

static void wait (void)  {                /* wait function */
  ;                               /* only to delay for LED flashes */
}

CLASS(LED)
{
  void (*run)();
};

static void running()
{
  unsigned int i;            /* Delay var */
  unsigned char j;
 for (j=0x01; j< 0x80; j<<=1)  {   /* Blink LED 0, 1, 2, 3, 4, 5, 6 */
     P1 = j;                       /* Output to LED Port */
     for (i = 0; i < 10000; i++)  {  /* Delay for 10000 Counts */
      wait ();                     /* call wait function */
     }
    }
  for (j=0x80; j> 0x01; j>>=1)  {   /* Blink LED 6, 5, 4, 3, 2, 1 */
     P1 = j;                       /* Output to LED Port */
```

```
     for (i = 0; i < 10000; i++)  {  /* Delay for 10000 Counts */
      wait ();                    /* call wait function */
     }
   }
}
/* --------------------------------*/
void main (void)  {
  LED le;
  le.run = running;
  while(1) {
    le.run();
  }
}
```

在 main()函数里有个指令 le.run = running，是让 le.run 指针指向 running()函数。在标准型写法里，这个动作是由 LEDSetting(&le)函数所执行的。只是调用 LEDSetting()时，必须传递 le 对象的指针给它，再通过指针去进行上述操作，相对以上较为动态一些，所以会多付出一些代价。这个程序在μVision3 IDE 上在编译及连接之后，可执行文件的大小为：

```
 Program Size: data=12.0 xdata=0 code=107
```

跟 EX27-01 范例的大小：82 Bytes 相比，只增加 25 Bytes 而已。

回顾一下 EX27-02 的标准型写法里，其撰写第一个类时，付出代价是 165 Bytes；而目前这 EX27-07 的精简写法里，加入第一个类只增加了 25 Bytes 而已。因此精简写法可以节省内存空间。接着，以精简写法加入第二个类看看它对程序大小的影响，如下：

```
/*  EX27-08.C  */
#include <REG51F.H>
#define LW_OOPC_PURE_STATIC
#include "lw_oopc_kc.h"

static void wait (void)  {               /* wait function */
  ;                             /* only to delay for LED flashes */
}

CLASS(LED)
{
  void (*run)();
};

void run()
{
  unsigned int i;            /* Delay var */
  unsigned char j;
 for (j=0x01; j< 0x80; j<<=1)  {   /* Blink LED 0, 1, 2, 3, 4, 5, 6 */
     P1 = j;                      /* Output to LED Port */
     for (i = 0; i < 10000; i++)  {  /* Delay for 10000 Counts */
      wait ();                    /* call wait function */
     }
   }

   for (j=0x80; j> 0x01; j>>=1)  {   /* Blink LED 6, 5, 4, 3, 2, 1 */
     P1 = j;                      /* Output to LED Port */
     for (i = 0; i < 10000; i++)  {  /* Delay for 10000 Counts */
      wait ();                    /* call wait function */
```

```
    }
  }
}

/* ------------------------------------ */
CLASS(Controller)
{
   void (*doing)();
};

static void doing()
{
   LED le;
   le.run = run;
   while (1) {
       le.run();
   }
}

/* ---------------------------------------- */
void main (void)  {
  Controller ctrl;
  ctrl.doing = doing;
  ctrl.doing();
}
```

这个程序在μVision3 IDE 上在编译及连接之后，可执行文件的大小为：

```
Program Size: data=12.0 xdata=0 code=107
```

只增加 15 Bytes 而已。回顾一下 EX27-04 的标准型写法里，其编写第 2 个类时，付出代价是 47 Bytes；而目前在 EX27-08 的精简写法里，加入第 2 个类只增加了 15 Bytes 而已。因此精简写法可以节省内存空间。

当你以精简型写法继续加入更多类时，你会发现每一个类大约都增加 15 Bytes 左右。统计图如图 27-4 所示。

此图表说明，如果你的 Keil C 程序使用 LW_OOPC 提供的机制，而且写了 n 个类时，其程序大小的增加量大约为：

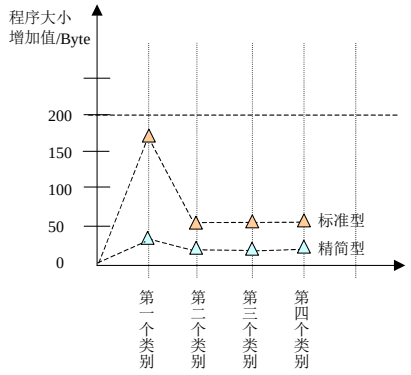


图 27-4

精简型写法： 25 + n*15

标准型写法： 165 + n*50

如果写了 10 的类（即 n 值为 10）的话，精简型付出 175 Bytes 的代价；而标准型付出 665 Bytes 的代价。一般而言 Keil C 程序写到 10 个类已经相当复杂了，只付出不到 200 Bytes 代价，却能让复杂的程序得到美好的结构和高度稳定性，真是非常划算的策略。

27.5 动态（Dynamic）型写法及其评估

动态型写法的特点在于，它依赖 malloc()库函数（Library Function）来动态分配内存空间给新产生的对象，所以程序必须连接到库里的 malloc()函数，因而使得程序变大了。而标准型和精简型都不依赖 malloc()函数，所以程序会小一些。基于这样的理由，你可以理解到：如果你采取标准写法，但你在函数内使用到 malloc()指令时，则程序在连接时也会连接到 malloc()库函数，则无论你采取标准或动态写法，对程序大小而言就没有差别了。

一般而言，比较复杂的数据结构，如链表（Linked List）、树形（Tree）等必须等到程序执行期间才能得知数据量的多少，这种情形就必须使用到 malloc()、calloc()等来执行动态内存分配的任务了。在 LW_OOPC 语言上，有些程序也只能等到程序执行期间才能得知对象数量的多少，此时就必须使用动态型的写法了。在动态型写法下，除了可以继续使用像 LEDSetting（&le）的函数之外，还增加了像 LEDNew()这样的函数。例如，前面 EX27-02 范例采取标准型写法，其 main()函数的内容为：

```
/*  采标准型的 EX27-02 范例 */
void main (void)  {
   LED le;
   LEDSetting(&le);
   while (1) {
      le.run();
}
```

现在改为动态型写法，可以使用 LEDNew()函数如下：

```
/*  采动态型的 EX27-02 范例 */
void main (void)  {
   LED *pe;
   pe = (LED*)LEDNew();
   while (1) {
      pe->run();
}
```

这种动态型写法，在执行期间才由 LEDNew()通过 malloc()函数来取得内存空间，所以执行时间会稍微增加；这种动态（Dynamic）用法需要连接到 malloc()函数，程序大小也稍微增加。反之，标准型写法不仅在程序大小上达到节约的目的，其程序执行速度也比较快。例如，其对象 le 是在指令 LED le; 就声明了，它和一般的 int、char 型态的变量一样，是在编译期间

就留下了内存空间，不需要在执行期间通过 malloc()函数来取得内存空间，所以这种“静态”（Static）用法的执行速度会快些。

接下来，我们再来实际观察动态型写法对程序大小的影响吧！再拿μVision3 IDE 上所附的 BLINKY（\keil\C51\EXAMPLES\BLINKY\）范例作为评估的基础。前面的 EX27-01 范例就是μVision3 所附的 BLINKY.C 的源代码。先前已经测得 EX27-01 范例程序的大小为 82 Bytes。现在以动态型写法，撰写第一个类如下：

```
/*  EX27-09.C  */
#include <REG51F.H>
#include "lw_oopc_kc.h"
/*  定义 LED 类  */
CLASS(LED)
{
  void (*run)();     /*  定义 LED 类里的函数  */
};

void wait (void)  {
      ;
    }

/*  实现 LED 类里的 run()函数  */
static void run(){
    unsigned int i;
    unsigned char j;

    for (j=0x01; j< 0x80; j<<=1)  {
     P1 = j;
     for (i = 0; i < 10000; i++)  {
      wait ();
     }
   }
   for (j=0x80; j> 0x01; j>>=1)  {
     P1 = j;
     for (i = 0; i < 10000; i++)  {
      wait ();
     }
   }
}

/*   LED 类的构造器  */
CTOR(LED)
  FUNCTION_SETTING(run, run)
END_CTOR

/*   主函数          */
char xdata MemPool[1024];
void main (void)  {
   LED *pe;        /*  声明 LED 类型的指针  */
init_mempool(MemPool,sizeof(MemPool));
   pe = LEDNew();  /*  调用构造器 LEDNew()来产生 LED 类的对象,
                   并且令 pe 指向该对象  */
   while (1) {
      pe->run();    /*  调用该对象的 run()函数  */
```

```
    }
}
```

在编译及连接之后，其可执行文件的大小增加为：

```
Program Size: data=16.0 xdata=1032 code=707
```

跟 EX27-01 范例的大小 82 Bytes 相比，增加了 625 Bytes。此程序没有使用 #define LW_OOPC_STATIC 或#define LW_OOPC_PURE_STATIC 语句，表示采取动态用法，而且涵盖静态用法，所以能同时使用静态的 LEDSetting()函数和动态的 LEDNew()函数。从上述分析里，你已经知道了，动态用法就是在程序执行期间由 LEDNew()等建构式调用 malloc()来安排对象的内存空间。在 Keil C 的一般环境里，因为没有强大的 OS（Operation System）负责管理内存空间，所以程序里必须由程序员特别声明内存空间，并且用 init_mempool()函数来清理该空间，之后调用 malloc()时就从空间取出依块分配给新产生的对象。例如上述的主函数程序代码：

```
/*  EX27-09 的主函数  */
char xdata MemPool[1024];
void main (void)  {
    LED *pe;
    init_mempool(MemPool,sizeof(MemPool));
    pe = (LED*)LEDNew();
    pe->run();
}
```

其中，指令——char xdata MemPool[1024]；保留了 1024 Bytes 的空间。

而另一指令——init_mempool（MemPool，sizeof(MemPool)）；清理保留的空间。

之后才能调用 LEDNew()去产生新对象。这是因为没有强大 OS 协助才需要在 C 程序里处理内存的事宜；如果有 OS 的协助的话，就不必这么麻烦了。例如，在 Windows 或 Linux 操作系统环境里使用 malloc()函数，就不需要上述两个指令了。

接着，以动态写法加入第二个类看看它对程序大小的影响，如下：

```
/* EX27-10.C  */
#include <REG51F.H>
#include "lw_oopc_kc.h"
/*  定义 LED 类  */
CLASS(LED)
{
  void (*run)();     /*  定义 LED 类里的函数  */
};

void wait (void)  {
      ;
    }

/*  实现 LED 类里的 run()函数  */
static void run(){
```

```
    unsigned int i;   unsigned char j;
    for (j=0x01; j< 0x80; j<<=1)  {
      P1 = j;
      for (i = 0; i < 10000; i++)  {
          wait ();
      }
   }
   for (j=0x80; j> 0x01; j>>=1)  {
     P1 = j;
     for (i = 0; i < 10000; i++)  {
      wait ();
     }
   }
}

/*   LED 类的构造器  */
CTOR(LED)
  FUNCTION_SETTING(run, run)
END_CTOR
/* ------------------------------------ */
CLASS(Controller)
{
   void (*doing)();
};

static void doing()
{
     LED *pl = LEDNew();
    while (1) {
      pl->run();
   }
}

CTOR(Controller)
  FUNCTION_SETTING(doing, doing);
END_CTOR
/* ------------------------------------ */
/*       主函数          */
char xdata MemPool[1024];
void main (void)  {
   Controller *pc;           /*  声明LED型态的指针  */
   init_mempool(MemPool,sizeof(MemPool));
   pc = ControllerNew();
   pc->doing();          /*  调用该对象的run()函数  */
}
```

这个程序在μVision3 IDE 上在编译及连接之后，可执行文件的大小为：

```
Program Size: data=19.0 xdata=1032 code=767
```

增加了 60 Bytes 而已。回顾一下 EX27-04 的标准型写法里，其撰写第二个类时，付出的代价是 47 Bytes；而目前这 EX27-10 的动态写法里，加入第二个类增加了 60 Bytes，其比标准写法付出更多代价。当你以动态型写法继续加入更多类时，你会发现每一个类大约都增加 60 Bytes 左右。如图 27-5 所示。

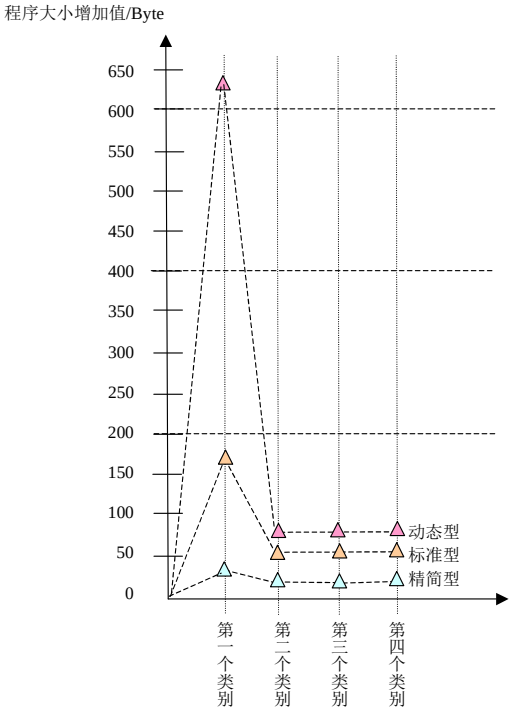


图 27-5

从上述可知，动态型与标准型两种写法的区别在于：是否在程序执行期间（Run-time）使用 malloc()函数来安排内存空间给对象。动态用法比较有弹性，而且有些复杂的数据结构，不得不使用动态用法才能建立起来。静态用法就像一般变量声明一样地声明类的变量（即对象），对于比较简单的程序或数据结构，这种用法常是足够的。对于 lw_oopc_kc.h 头文件来说，其默认的是动态用法。例如程序代码：

```
/* EX27-09.C  */
/* BLINKY.C */
#include <REG51F.H>
#include "lw_oopc_kc.h"
……
```

这就是采用的动态用法，而且涵盖了静态用法。

以上区分为动态型、标准型和精简型，只是为了配合不同硬件资源而让程序员能调节程序的大小而已。在撰写程序时，其实都是可以混合使用的，这让程序员能非常自由地决定程序的形式，一方面满足硬件资源的限制，另一方面又能追求顶级的美好结构。因此，对一个 C 程序员及其编写的嵌入式软件而言，LW_OOPC 只是给它（或他）黄袍加身，并不会伤害到龙体本身。

第 28 章　Keil C51 的特殊数据类型

28.1 8051 的 CODE 存储区

程序的 ZCODE 存储区是用来储存程序代码的区域，通常是某种类型的只读存储器（ROM），而 CPU 就从这区域取得指令去执行。标准 8051 内建有 4K 字节（Bytes）的程序内存，并可扩充至 64K Bytes。虽然标准 8051 是 8 位的芯片，其寄存器（Register）和数据总线（Data Bus）都是 8 位。但是它却拥有 16 位的地址空间，可直接寻址到 64K Bytes 的空间。这意味着，可同时具有存取 64K Bytes 的 code 内存和 data 内存的空间。

28.2 8051 的 DATA 存储区

8051 的 code 存储区与数据（data）存储区是分开的，两者的存取不会互相冲突。8051 使用 256 Bytes 的内部存储器（RAM）空间来储存程序执行期间所产生的数据。其中，

地址 00H ~ 1FH 之间：共 32 Bytes 为寄存器集（Register Bank）。共分为 8 个寄存器，称为 R0，R1，…，R7。

地址 20H ~ 2FH 之间：共 16 Bytes = 128 bits 空间，称为可位寻址区域，适合利用位运算指令（如&、|、^等）来对此区域的位直接运算。

地址 30H ~ 7FH 之间：共 80 Bytes 空间，作为用户数据及堆栈区。

地址 80H ~ FFH 之间：共 128 Bytes 空间，为特殊功能寄存器区。例如 P0、P1、IP 等寄存器就定义在此区。

既然划分为这么多区块，那么写 Keil C 程序时又该如何指定呢？答案是：可使用 Keil C 的存储类型关键字，例如：

```
/* EX28-01.c */
#include <REG51F.H>
#include "lw_oopc_kc.h"

CLASS(Point)
{  void (*init)(char, char, Point*);
   unsigned char x, y;
};
static void init(char a, char b, Point* t) {  t->x = a;   t->y = b;  }
CTOR(Point)
   FUNCTION_SETTING(init, init);
END_CTOR
/* ---------------------------------- */
char xdata MemPool[8];
void main (void){
   Point *po;
   init_mempool(MemPool,sizeof(MemPool));
   po = PointNew();
   po->init(10, 3, po);
```

```
   P0 = po->x;       P1 = po->y;
   while(1) {
    ;
    }
}
```

其中的指令：char xdata MemPool[8]；

使用 xdata 关键字，就将 MemPool[]定位于外部数据存储器区域。如果将上述主程序部分修改为：

```
/* ------------------------------- */
char xdata MemPool[8];
char bdata temp;
void main (void) {
   Point *po;
   init_mempool(MemPool,sizeof(MemPool));
   po = PointNew();
   po->init(15, 16, po);
   temp = po->x;
   temp |= po->y;
   P0 = temp;
   while(1) {
    ;
    }
}
```

使用 bdata 关键字，就将 temp[]定位于可位寻址的内部数据存储器，也就是地址 20H ~ 2FH 之间共 16 Bytes 的空间。现将 Keil C 的存储类型关键字列表如下（见表 28-1）：

表 28-1

存储类型	说　明
data	直接寻址的内部数据存储器，存取速度最快，范围是 00H ~ 7FH。
bdata	可位寻址的内部数据存储器，允许位与字节混合存取，范围是 0x20 ~ 0x2f。
idata	间接寻址的内部数据存储器，允许存取全部内部地址，范围是 0x80 ~ 0xff。
pdata	以 R0、R1 分页寻址的外部数据存储器（256 Bytes 之内），以 MOVX @Ri 指令存取。
xdata	以 DPTR 寻址的外部数据存储器(64K Bytes 之内)，以 MOVX @DPTR 指令存取。

28.3 Keil C 的存储模式

Keil C 提供 3 种内存模式（Memory Model），包括 SMALL、COMPACT 和 LARGE。

- SMALL 模式：将所有变量默认摆放在内部存储器，数据存取非常快。但是 SMALL 模式的地址空间有限。对于小型程序而言，其变量和数据摆放在 data 内部数据存储器中，速度快，是好的选择。然而对于较大程序，data 区最好只摆放常用的数据（如循环计数、数据索引等），而将较大的数据摆放到其他区域。
- COMPACT 模式：将所有的变量默认摆放在外部内存的一页（Page）的空间里，此内存最多为 256 Bytes。使用此模式，就如同明确宣告为 pdata 存储类型一般，以 R0、R1 分

页寻址的方式来存取。

- LARGE 模式：将所有的变量默认摆放在外部内存，此范围可达 64K Bytes。使用此模式，就如同明确宣告为 xdata 存储类型一般，以 DPTR 寻址方式来存取。

28.4 Keil C 的专用数据类型

为了充分发挥 8051 硬件的作用，Keil C 特别提供了 4 种特殊的数据类型，见表 28-2。

表 28-2

类型	位数	范围	说明
bit	1	0 to 1	代表 1 位数据，会被指定到 20H~2FH 的区域。
sbit	1	0 to 1	代表 1 位数据,可存取 20H~2FH 的区或 SFR 特殊功能寄存器。
sfr	8	0 to 255	用于 80H~FFH 区域的特殊功能寄存器，一次存取 8 位。
sf16	16	0 to 65 535	一次存取 16 位的特殊功能寄存器，如 Timer 等。

sfr 和 sfr16 可直接对 8051 的特殊功能寄存器进行定义，定义格式如下：

```
sfr 寄存器名称 = 寄存器地址常数;
```

sbit 能对可位寻址的位进行定义。定义格式如下：

- sbit 位名称 =位地址常数;
- sbit 位名称 = 特殊功能寄存器名称 ^ 位位置;
- sbit 位名称 = 字节地址 ^ 位位置;

例如 Keil C 程序代码如下：

```
/* EX08-02.c  */

sfr LED_PORT = 0x80; // 定义 LED_PORT，其地址 80H
sbit Switch_pin = LED_PORT ^ 0;

void main (void)
{
  LED_PORT = 0xf0;
  Switch_pin = 1;

  while(1) {
      ;
    }
}
```

LED_PORT 代表 80H 地址上的字节（Byte），而 Switch_pin 代表其中的第 0 位。所以此程序输出：11110001。请在看一个 Keil C 程序范例：

```
/* EX28-03.c  */
```

```
sfr DATA_PORT = 0x80;
unsigned char bdata ch;
sbit bit77 = ch ^ 7;
sbit bit66 = ch ^ 6;

void main (void)
{
  ch = 0xaa;
  DATA_PORT = 0x00 | bit77;

  while(1) {
      ;
    }
}
```

DATA_PORT 代表 80H 地址上的字节（Byte），而 bit66 和 bit77 分别代表其中的第 6 和第 7 位。所以此程序输出 00000001。

第 29 章　以 Keil C51 定义类

29.1 定义类

前面已经介绍过，类是一群具有共同重要特性的对象。类的定义就是说明这群对象具有什么样的重要特性，特性包括对象的特征及行为。软件中的对象以数据来表达特征，以函数来表达行为。因此，类的定义就是说明软件中的对象，应含哪些数据及哪些函数。例如，欲描述手中的一朵花，而这朵花是一朵（is a）玫瑰花，则可得知手上的花是对象，而玫瑰花是类。为了描述手上的玫瑰花，就得定义叫 Rose 的类。如果您想描述其颜色，也想描述其最适合做哪个月份的生日花，则可知 Rose 类应包含两项重要数据 color 和 month。于是，就可运用 lw_oop_kc.h 宏来定义 Rose 类，并得到对象如下：

```
/* EX29-01.C */
#include <REG52.H>
#include <stdio.h>
#include "lw_oopc_kc.h"

CLASS(Rose)
   {
      int  price;
      int  month;
     void (*say)();
   };

void say() {  printf("color is RED...\n");   }

void main()
   {
    Rose rose;
    rose.price = 20;    rose.month = 6;
    rose.say = say;
     /* ---------------- Serial Windows --------------- */
   SCON = 0x50;   TMOD |= 0x20;
    TH1  = 221;    TR1   = 1;     TI = 1;
   /* ------------------------------------------------- */
   rose.say();
    while(1) {
      ;
   }
}
```

此时，Rose 类的对象皆具有一项共同行为——说出其颜色。在软件中，靠 say()来表达这项行为。对于指令 rose.say(); 可解释为：将信息 say() 传送给 Rose 的对象。其意义是“请问你是什么颜色呢？”当此对象接到信息 say()时，便启动其内含的 say()函数，并执行 say()函数内之指令。上述 Rose 之对象已具有一个函数 say()，支持一项重要行为——Rose 的对象能输出自己的内容。如果对其他行为有兴趣，可继续增加 Rose 类的函数，使其对象具有多样化的行为。

29.2 构造器（Constructor）

在 LW_OOPC 程序中，设计类是一项重要的工作，其目的是：由它产生对象。其中有个

幕后工作者，它依照类的定义产生对象，可称之为对象之母。这个幕后工作者就是“构造器”（Constructor）函数。其主要功能为依照类的定义分配内存空间给所声明的对象。在 lw_oopc_kc.h 头文件里已经定义了构造器宏，可以直接使用。例如：

```
/* EX29-02.C */
#include <REG52.H>
#include <stdio.h>
#include "lw_oopc_kc.h"

CLASS(Rose)
   {
      int  price, month;
     void (*say)(Rose*);
   };

void say(Rose *t) {
    printf("%d,  %d\n", t->price, t->month);
 }
CTOR(Rose)
  FUNCTION_SETTING(say, say);
END_CTOR

char xdata MemPool[1024];
void main() {
    Rose *pr;
   init_mempool(MemPool,sizeof(MemPool));
   pr = RoseNew();
    pr->price = 20;
    pr->month = 6;
      /* ------------------- Serial Window  ----------------- */
   SCON  = 0x50;     TMOD |= 0x20;
    TH1   = 221;    TR1   = 1;     TI = 1;
   /* ------------------------------------------------------- */
   pr->say(pr);
   while(1) {
      ;
   }
}
```

CTOR 就是 Constructor 的简写，这个宏里定义了 RoseNew()构造器，它会产生 Rose 的对象，传回该对象的指针值，并存入 pr 指针变量里。

29.3 Keil C51 类设计之实例说明

——以红绿灯（Traffic Light）类为例

29.3.1 分析与设计

大家都知道街道上的红绿灯有 3 种状态：亮红灯、亮绿灯和亮黄灯。例如在图 29-1 里，就处于黄灯状态。

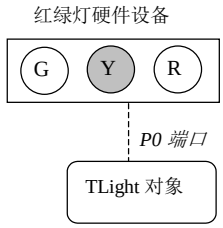


图 29-1

图 29-1 说明，可以设计一个 TLight 软件对象来控制（通过 P0 端口）红绿灯的硬件组件。为了产生 TLight 对象，就得先定义一个 TLight 类，但是这个类该包含哪些数据和函数呢？由于红绿灯对象的状态很明确，我们就可以从状态分析开始，例如使用 UML 的状态图表示，如图 29-2 所示。

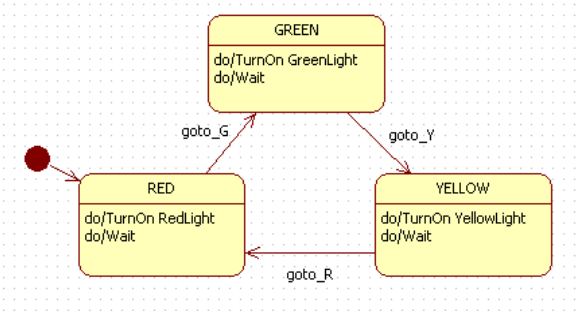


图 29-2

图 29-2 说明，TLight 对象会一直循环于 3 个状态中不断地变换。当处于 RED 状态时，会做两件事情：

1. TurnOn RedLight：改变 P0 端口的 P0^0、P0^1 和 P0^2 针脚的值，使红灯亮（也关闭绿灯和黄灯）。
2. Wait：让红灯亮着持续一段时间。

依此类推，当处于另外两个状态时，也会做两件事情，如图 29-2 所示。于是，得知 TLight 对象的行为了，将它们对应到类的函数，就可绘出 UML 类图（见图 29-3）。

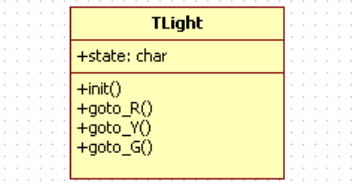


图 29-3

29.3.2 以 Keil C 实现红绿灯控制程序

依据前面所介绍的 lw_oopc_kc.h 宏，就可以使用 Keil C 来编写 TLight 类了，如下述的 EX29-lig.h 内容：

定义 TLight 类

```
/* EX29-lig.h  */
#include "lw_oopc_kc.h"
CLASS(TLight)
   {
        char state;
        void (*init)(TLight*);
        void (*run)(TLight*);
      void (*goto_R)();
      void (*goto_Y)();
      void (*goto_G)();
   };
```

类里的函数定义格式为：

```
返回值类型 (*函数名称)();
```

或

```
返回值类型 (*函数名称)(参数类型1, 参数类型2, ……);
```

类定义好了，就开始编写函数的实现内容：

编写 TLight 类

```
/* EX29-lig.c   */
#include <REG51F.H>
#include <stdio.h>

#define LW_OOPC_STATIC
#include "ex29-light.h"

sbit RED_pin = P0^0;
sbit YELLOW_pin = P0^1;
sbit GREEN_pin = P0^2;
void g_delay(unsigned int ms) {
  int i, j;
  for(i=0; i<ms; i++)
    for(j=0; j<120; j++);
}
static void init(TLight* t) { P0 = 0x00;  t->state = 'R';   }
static void run(TLight* t) {
     switch(t->state) {
     case 'R':   t->state = 'G';  t->goto_G();   break;
     case 'G':   t->state = 'Y';  t->goto_Y();   break;
     case 'Y':   t->state = 'R';  t->goto_R();   break;
   }
}
static void goto_R() {
     RED_pin = 1;     YELLOW_pin = 0;     GREEN_pin = 0;
```

```
     g_delay(10000);
    }
static void goto_Y() {
     RED_pin = 0;     YELLOW_pin = 1;     GREEN_pin = 0;
     g_delay(3000);
    }
static void goto_G() {
     RED_pin = 0;     YELLOW_pin = 0;     GREEN_pin = 1;
     g_delay(10000);
    }
CTOR(TLight)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(run, run)
  FUNCTION_SETTING(goto_R, goto_R)
  FUNCTION_SETTING(goto_Y, goto_Y)
  FUNCTION_SETTING(goto_G, goto_G)
END_CTOR
```

这个 FUNCTION_SETTING(run, run)宏的用意是：让类定义（.h 文件）的函数名称能够与实现的函数名称不同。例如在 EX11-lig.c 里可写为：

```
static void running()
{ …. . }

CTOR(TLight)
{
   ……
FUNCTION_SETTING(run, running);
……
}
```

这是创造.c 文件自由替换的空间，是实现接口的重要基础。最后看看如何编写主程序：

编写 main()主函数

```
/* EX29-ap-1.c  */
#include <REG51F.H>
#include <stdio.h>
#define LW_OOPC_STATIC
#include "ex29-light.h"

extern void TLightSetting(TLight*);
void main()  {
     TLight light;
     TLightSetting(&light);
     light.init(&light);
     while(1) {
        light.run(&light);
        }
   }
```

LightSetting()是由 CTOR 宏所产生的函数。由于它是定义在其他.C 文件（即 EX29-lig.c）之中的，所以必须加上 extern void TLightSetting();指令。

29.3.3 红绿灯类的另一种写法

如果对象可以明显地区分出它的不同状态，就能有效降低其复杂性。降低的方法就像上一小节中 TLight 类的写法。由于其复杂性降低了，系统可靠性也获得提升。一旦分析出对象的状态变化，就能实现为 Keil C 程序了，以下将展示如何以另一种方式来写出 TLight 类。在你理解了这两种常见写法之后，就能视情况与需要而挑选最适当的写法。现将上一小节的 TLight 类改写如下：

定义 TLight 类

```
/* EX29-light.h  */
#include "lw_oopc_kc.h"
CLASS(TLight)
   {   char state;
       unsigned int wait_R, wait_Y, wait_G;
       void (*init)(TLight*);
       void (*run)(TLight*);
       void (*perform)(TLight*);
   };
```

一般而言，如果像 goto_R()、goto_Y()和 goto_G()等内容并不复杂的函数，可以将它们合并为一个函数（如 perform()函数）。合并之后，其程序代码如下：

编写 TLight 类

```
/* EX29-light.c */
#include <REG51F.H>
#include <stdio.h>

#define LW_OOPC_STATIC
#include "ex29-light.h"

sbit RED_pin = P0^0;
sbit YELLOW_pin = P0^1;
sbit GREEN_pin = P0^2;

void g_timer_delay() {
   TMOD &= 0xF0;   TMOD |= 0x01;
   ET0 = 0;   TH0 = 0x3C;   TL0 = 0xB0;
   TF0 = 0;   TR0 = 1;
   while(TF0 == 0);
   TR0 = 0;
}
void g_delay(unsigned int sec) {
   int i, j;
   for(i=0; i<sec; i++)
      for(j=0; j<20; j++)
         g_timer_delay();
}
static void init(TLight* t) {
    P0 = 0x00;
    t->state = 'R';    t->wait_R = 10;  t->wait_Y = 2;  t->wait_G = 10;
   }
static void run(TLight* t) {
```

```
     switch(t->state) {
     case 'R':   t->state = 'G';  t->perform(t);   break;
     case 'G':   t->state = 'Y';  t->perform(t);   break;
     case 'Y':   t->state = 'R';  t->perform(t);   break;
    }
}
static void perform(TLight *t) {
    switch(t->state) {
     case 'R':
         RED_pin = 1;   YELLOW_pin = 0;   GREEN_pin = 0;
         g_delay(t->wait_R);
         break;
     case 'Y':
         RED_pin = 0;    YELLOW_pin = 1;  GREEN_pin = 0;
         g_delay(t->wait_Y);
         break;
     case 'G':
         RED_pin = 0;    YELLOW_pin = 0;  GREEN_pin = 1;
         g_delay(t->wait_G);
         break;
     }
   }

CTOR(TLight)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(run, run)
  FUNCTION_SETTING(perform, perform)
END_CTOR
```

其实，针对图 29-2 的状态图，可以有许多种 Keil C 的实现方法，完全视你的喜好而定。接着，编写主程序如下。

编写 main()主函数

```
/* EX29-ap-2.c  */
#include <REG51F.H>
#include <stdio.h>
#define LW_OOPC_STATIC
#include "ex29-light.h"

extern TLightSetting(TLight*);
void main()  {
     TLight light;
     TLightSetting(&light);
     light.init(&light);
     while(1) {
        light.run(&light);
        }
   }
```

以上，你已经看到两种方法可以用来实现图 29-2 的状态图，设计出理想的类。不过，一般而言前一种方法比较流行。

第 30 章　应用范例一

——Keil C51 表现对象的沟通

30.1　以 Toggle Light 电灯为例

30.2　以红绿灯控制系统为例

30.1 以 Toggle Light 电灯为例

30.1.1 分析与设计

此例是要开发一个 Toggle Light Controller 软件，现设计 3 个对象（Object）来组成一个 Light Controller。其中，Wall Switch 对象控制 Wall Switch Set 硬件设备；Door Switch 对象控制 Door Switch Set 硬件设备；而 Light 对象控制多个 Light Bulb Set 电灯。这 3 个对象间的沟通情形如图 30-1 所示。

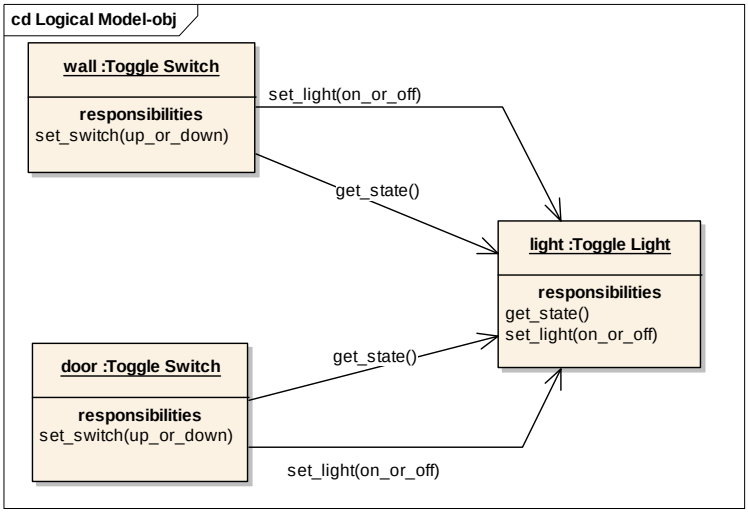


图 30-1

这是信息传递的表示方式。3 个软件对象一起合作（Collaboration），相互传递信息。3 个对象互助合作完成 Light Controller 对象所应实现的功能。

30.1.2 设计 OOPC 类

Wall Switch 和 Door Switch 对象的特性和行为是一样的，所以可归为同一个类，取名为 Switch 类。而 Light 对象可归到另一个类，取名为 Light 类。如图 30-2 所示。

这个 Toggle Switch 类将用来产生两个 Switch 对象：Wall Switch 和 Door Switch。而 Toggle Light 类将用来产生 Light 对象。为什么用一个 Toggle Switch 类产生两个对象而不设计两个类，由 Wall Switch 和 Door Switch 各产生一个 Switch 对象呢？这完全属于类设计者的专业判断。一般而言，如果 Wall Switch 和 Door Switch 两对象的数据属性（Attribute）和行为（Operations）都一样，只需要一个模子就够了，何必用两个类别呢？

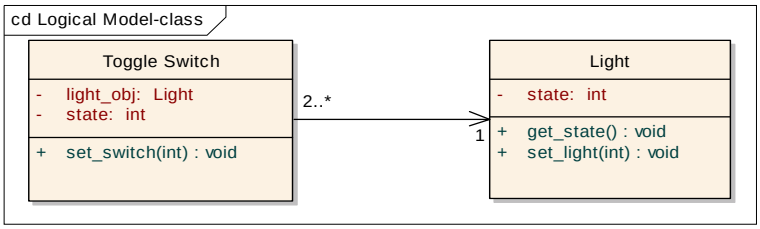


图 30-2

30.1.3 以 OOPC 实现：使用 Keil C

现在编写 Keil C 程序来实现上述的系统分析与设计，并在μVision3 IDE 环境里执行；其步骤如下。

Step-1 编写 Light 类。

小灯 Light 的定义代码

```
/* EX30-lig.h   */
#ifndef LIGHT_H
#define LIGHT_H
#include "lw_oopc_kc.h"

CLASS(Light)
{
    int state;
    void (*init)(void*);
    int (*get_state)(void*);
    void (*set_light)(void*, int flag);
};
#endif
```

小灯 Light 的实现代码

```
/* EX30-lig.c */
#include <REG52.H>
#include "EX30-lig.h"

static void init( void *t ) {
   Light* cthis = (Light*) t;   cthis->state = 0;
}
static int get_state(void* t) {
   Light* cthis = (Light*) t;   return cthis->state;
}
static void  set_light(void* t, int flag) {
 Light* cthis = (Light*) t;
 cthis->state = flag;
 if(cthis->state == 0)     P0 = 0x00;
 else                P0 = 0x3c;
}
CTOR(Light)
  FUNCTION_SETTING(init, init);
  FUNCTION_SETTING(get_state, get_state);
  FUNCTION_SETTING(set_light, set_light);
```

```
END_CTOR
```

Step-2 编写 Switch 类。

开关 Switch 的定义代码

```
/*   EX30-sw.h   */
#ifndef SWITCH_H
#define SWITCH_H
#include "lw_oopc_kc.h"
#include "EX30-lig.h"

CLASS(Switch)
{
  int state;
  Light* light_obj;
  void (*init)(void*);
  int (*get_state)(void*);
  void  (*set_switch)(void*);
};
#endif
```

开关 Switch 的实现代码

```
/*  EX30-sw.c  */
#include <stdio.h>
#include "EX30-sw.h"

static void init(void *t) {
   Switch* cthis = (Switch*) t;   cthis->state = 0;
}
static int get_state(void* t){
   Switch* cthis = (Switch*) t;   return cthis->state;
}
static void set_switch(void* t){
  int st;  Light* light;
  Switch* cthis = (Switch*) t;
  cthis->state = !(cthis->state);
  light = cthis->light_obj;
  st = light->get_state(light);
  if (st == 1)   light->set_light(light, 0);
  else        light->set_light(light, 1);
}
CTOR(Switch)
   FUNCTION_SETTING(init, init);
   FUNCTION_SETTING(get_state, get_state);
   FUNCTION_SETTING(set_switch, set_switch);
END_CTOR
```

Step-3 编写 main()函数。

main()函数

```
/*  EX30-app-1.c   */
#include <REG52.H>
#include <stdio.h>
#include "EX30-lig.h"
#include "EX30-sw.h"
```

```
extern void* LightNew();
extern void* SwitchNew();

void g_delay(unsigned int ms) {
   int i, j;
   for(i=0; i<ms; i++)
      for(j=0; j<120; j++);
}

char xdata MemPool[1024];
void main(){
    Light* light;  Switch* wall;   Switch* door;
    init_mempool(MemPool,sizeof(MemPool));
    light = (Light*)LightNew();
    wall = (Switch*)SwitchNew();
    door = (Switch*)SwitchNew();
    light->init(light);

    wall->light_obj = light;    door->light_obj = light;
    wall->set_switch(wall);
    while(1) {
       /* 上楼: press PSW1  */
       wall->set_switch(wall);     g_delay(8000);
       /*  press PSW2  */
       door->set_switch(door);     g_delay(8000);
       /* 下楼:  press PSW2  */
       door->set_switch(door);    g_delay(8000);
       /*  press PSW1  */
       wall->set_switch(wall);    g_delay(8000);
    }
}
```

现作以下说明，在 main()里有如下指令：

```
/* 上楼: press PSW1  */
wall->set_switch(wall);
g_delay(8000);
```

表示欲上楼梯时，按下墙壁上的开关，送出 set_switch 信息给 wall 对象，P0 就送出开灯的信号，如图 30-3 所示。

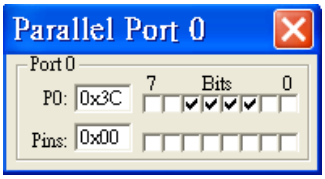


图 30-3

指令：

```
/*  press PSW2  */
door->set_switch(door);
g_delay(8000);
```

表示已经上楼了，就按下门上的开关，送出 set_switch 信息给 door 对象，P0 就送出关灯的信号，如图 30-4 所示。

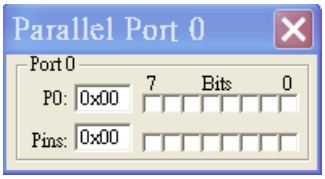


图 30-4

指令：

```
/* 下楼： press PSW2 */
door->set_switch(door);
g_delay(8000);
```

表示欲下楼梯时，就按下门上的开关，送出 set_switch 信息给 door 对象，P0 就送出开灯的信号，如图 30-5 所示。

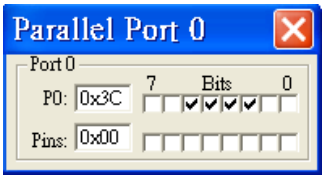


图 30-5

指令：

```
        /* press PSW1 */
       wall->set_switch(wall);
       g_delay(8000);
```

表示已经下楼了，按下墙壁上的开关，送出 set_switch 信息给 wall 对象， P0 就送出关灯的信号，如图 30-6 所示。

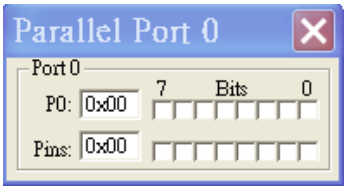


图 30-6

30.2 以红绿灯控制系统为例

30.2.1 分析与设计

在第 11.7 节里，曾经设计过一个 TLight 对象来控制一个红绿灯组件的状态变化；然而，在一般的十字路口上，通常需要同步控制两个红绿灯，以便同时协调两边的车辆。所以在本节里，把第 11.7 节的范例加以扩大，编写一个 Keil C 程序来同步控制一组（含有两个或 4 个红绿灯对象）组件，如图 30-7 所示。

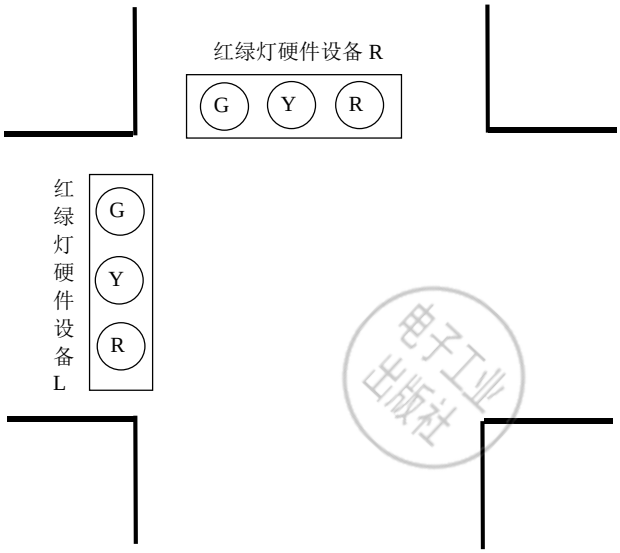


图 30-7

这两个红绿灯组件的状态变化必须互相协调，例如一个亮绿灯时，另一个必须亮红灯 。所以这个范例会比第 11.7 节的范例来得复杂一些，此时可以设计两个 TLight 对象来分别控制一个红绿灯硬件组件，并且设计一个 CTRL 对象来指挥这两个 TLight 对象，以构成层级式控制的架构，如图 30-8 所示。

步骤如下。

Step-1 初步的类规划。

现在从图 30-8 可知道需要 3 个对象。那么，总共需要几个类呢？这要视两个 TLight 对象行为上的差异而定，如果差别很大，宜设计出两个类，各产生一个对象。反之，如果两个 TLight 对象的行为很相似，可只设计一个 TLight 类即可。在本节的范例里，将采取前者，设计出两个类，如图 30-9 所示。

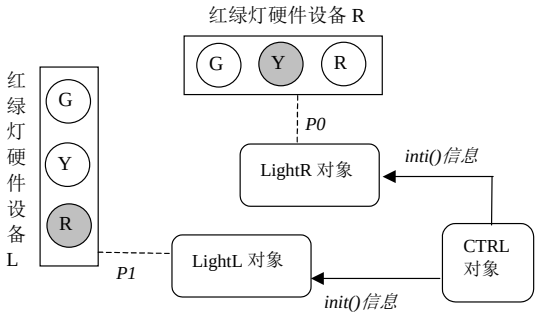


图 30-8

图 30-8 是一张草稿图，并非标准的 UML 图，但它可协助我们思考如何设计如图 30-9 的类图。

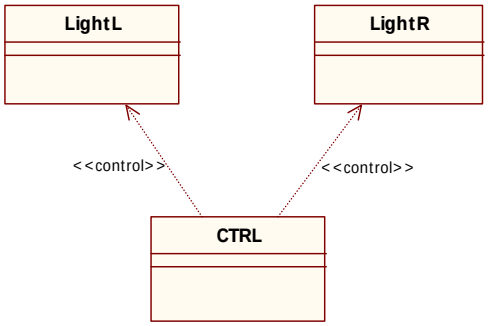


图 30-9

现在知道共需要几个类了，但内部的数据和函数都还欠缺，我们将借助对象状态分析来寻找类的函数。

Step-2 分析对象的状态变化。

针对 LightR 和 LightL 对象

上述只是初步的类图，还没有列出其内部的函数。那么，要如何决定类应含有哪些函数呢？在控制系统中，最常见的办法是观察其状态的变化，LightR 和 LightL 类的对象行为是一致的，其状态如图 30-10 所示。

其中，ev_Y 和 ev_RG 是两个事件，也可以将它们视为信息。这是 CTRL 对象发出的指挥信息，来驱动 LightR 和 LightL 对象的状态变化。例如，当 LightR 对象处于 GREEN 状态时，接到 CTRL 对象送来 ev_Y 事件（或信息），此时对象就转移到 YELLOW 状态。

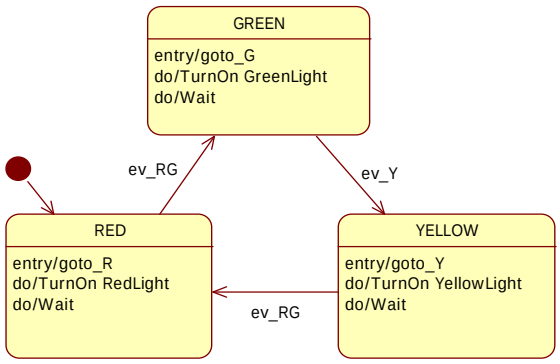


图 30-10

同理，当对象处于 YELLOW 状态时，接到 CTRL 对象送来 ev_RG 事件（或信息），此时对象就转移到 RED 状态，依此类推。于是，图 30-10 可协助我们思考 LightR 和 LightL 类该有哪些函数。

针对 CTRL 对象

接着，看看 CTRL 对象的状态变化吧！如果十字路口的两方车道每隔 145 秒钟交换一次通断状态，而且黄灯停留时间为 15 秒的话，则 CTRL 就只有两种状态了，称之为“S”状态与“L”状态。可绘制如图 30-11 所示的状态图。

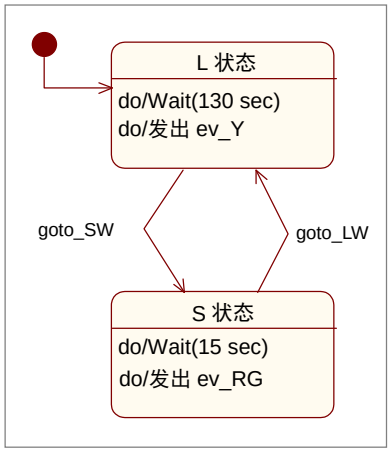


图 30-11

一开始就进入“L”状态，此刻有一组红绿灯处于 GREEN 状态，而另一组处于 RED 状态。停留 130 秒之后发出 ev_Y 信息，表示 GREEN 状态维持 130 秒。从图 30-10 可看出，另一组处于 RED 状态的红绿灯并不会接受 ev_Y 信息，所以 RED 状态会维持 145 秒。

Step-3 绘制完整的类图。

从图 30-7 的状态行为可协助我们思考 CTRL 类该有哪些函数。完整的类图如图 30-12 所示。

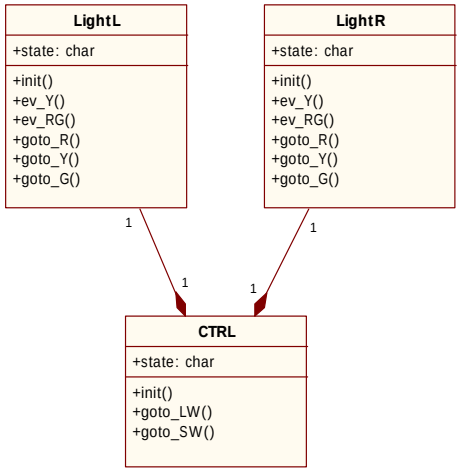


图 30-12

图中的菱形表示 CTRL 对象包含一个 LightR 对象和一个 LightL 对象。而“黑色”菱形表示该两个对象是由 CTRL 对象所产生的。

Step-4 设计对象间的互动细节。

刚才已经设计出类了，这些类能用来产生对象，它们会互相沟通互动，现以 UML 的对象合作图来表示，如图 30-13 所示。

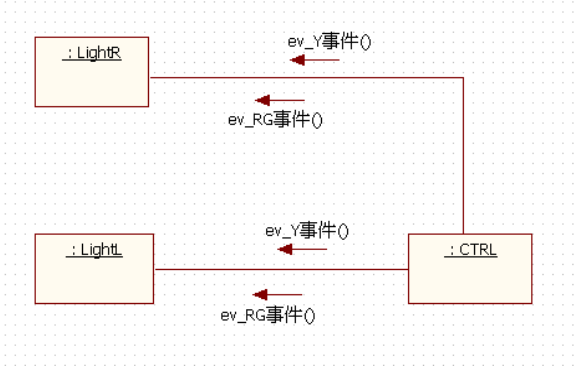


图 30-13

这使用了信息传递的沟通方式。

30.2.2 以 LW_OOPC 实现：使用 Keil C

现在编写 Keil C 程序来实现上述的系统分析与设计，并在μVision3 IDE 环境里执行；其步骤如下。

Step-1 编写 LightR 类别。

LightR 的定义代码

```
/* EX30-light-r.h  */
#include "lw_oopc_kc.h"

CLASS(LightR) {
   char state;
   void (*init)(char, LightR*);
   void (*ev_Y)(LightR*);
   void (*ev_RG)(LightR*);
   void (*goto_R)();
   void (*goto_Y)();
   void (*goto_G)();
   };
```

LightR 的实现代码

```
/* EX30-light-r.c  */
#include <REG51F.H>
#include <stdio.h>

#define LW_OOPC_STATIC
#include "EX30-light-r.h"
sbit RED_pin = P0^0;
sbit YELLOW_pin = P0^1;
sbit GREEN_pin = P0^2;

static void init(char st, LightR* t) {
   P0 = 0x00;
   t->state = st;
   if(st == 'G') t->goto_G();
   else if(st == 'R') t->goto_R();
      else t->goto_Y();
  }
static void ev_Y(LightR* t) {
     if( t->state == 'G') {
         t->state = 'Y';  t->goto_Y();
      }
}
static void ev_RG(LightR* t) {
     if( t->state == 'Y') {
         t->state = 'R';  t->goto_R();
      }
     else if(t->state == 'R') {
         t->state = 'G';  t->goto_G();
         }
}
static void goto_R() {
     RED_pin = 1;     YELLOW_pin = 0;     GREEN_pin = 0;
   }
```

```
static void goto_Y() {
      RED_pin = 0;     YELLOW_pin = 1;     GREEN_pin = 0;
   }
static void goto_G() {
      RED_pin = 0;     YELLOW_pin = 0;     GREEN_pin = 1;
   }

CTOR(LightR)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(ev_Y, ev_Y)
  FUNCTION_SETTING(ev_RG, ev_RG)
  FUNCTION_SETTING(goto_R, goto_R)
  FUNCTION_SETTING(goto_Y, goto_Y)
  FUNCTION_SETTING(goto_G, goto_G)
END_CTOR
```

就 LightR 而言，ev_Y 和 ev_RG 是两个外来的事件或信息。LightR 对象会调用 goto_R() 等函数来响应这些外来的事件。

Step-2 编写 LightL 类。

LightL 的定义代码

```
/* EX30-light-l.h  */
#include "lw_oopc_kc.h"

CLASS(LightL)
   {
   char state;
   void (*init)(char, LightL*);
   void (*ev_Y)(LightL*);
   void (*ev_RG)(LightL*);
   void (*goto_R)();
   void (*goto_Y)();
   void (*goto_G)();
   };
```

LightL 的实现代码

```
/* EX30-light-l.c  */
#include <REG51F.H>
#include <stdio.h>
#define LW_OOPC_STATIC
#include "EX30-light-l.h"

sbit RED_pin = P1^0;
sbit YELLOW_pin = P1^1;
sbit GREEN_pin = P1^2;

static void init(char st, LightL* t) {
   P1 = 0x00;
   t->state = st;
   if(st == 'G') t->goto_G();
   else if(st == 'R') t->goto_R();
      else t->goto_Y();
  }
static void ev_Y(LightL* t) {
     if( t->state == 'G') {
```

```
            t->state = 'Y';   t->goto_Y();
        }
}
static void ev_RG(LightL* t) {
     if( t->state == 'Y') {
         t->state = 'R';  t->goto_R();
      }
     else if(t->state == 'R') {
         t->state = 'G';   t->goto_G();
      }
}
static void goto_R() {
     RED_pin = 1;     YELLOW_pin = 0;     GREEN_pin = 0;
   }
static void goto_Y() {
     RED_pin = 0;     YELLOW_pin = 1;     GREEN_pin = 0;
   }
static void goto_G() {
     RED_pin = 0;     YELLOW_pin = 0;     GREEN_pin = 1;
   }

CTOR(LightL)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(ev_Y, ev_Y)
  FUNCTION_SETTING(ev_RG, ev_RG)
  FUNCTION_SETTING(goto_R, goto_R)
  FUNCTION_SETTING(goto_Y, goto_Y)
  FUNCTION_SETTING(goto_G, goto_G)
END_CTOR
```

Step-3 编写 CTRL 类。

开关 CTRL 的定义代码

```
/* EX30-ctrl.h  */
#include "ex30-light-r.h"
#include "ex30-light-l.h"

CLASS(CTRL)
   {
     char state;
     LightR lg_right;
    LightL lg_left;
    void (*init)(CTRL*);
     void (*run)(CTRL*);
    void (*goto_LW)(CTRL*);
    void (*goto_SW)(CTRL*);
   };
```

CTRL 的实现代码

```
/* EX30-ctrl.c  */
#include <REG51F.H>
#include <stdio.h>
#define LW_OOPC_STATIC
#include "ex30-ctrl.h"

extern LightRSetting(LightR*);
extern LightLSetting(LightL*);
```

```
void g_timer_delay() {
   TMOD &= 0xF0;   TMOD |= 0x01;
   ET0 = 0;
   TH0 = 0x3C;   TL0 = 0xB0;
   TF0 = 0;      TR0 = 1;
   while(TF0 == 0);
   TR0 = 0;
}

void g_delay(unsigned int sec) {
   int i, j;
   for(i=0; i<sec; i++)
      for(j=0; j<20; j++)
         g_timer_delay();
}

static void init(CTRL* t) {
    LightR *pr; LightL *pl;
   pr = &(t->lg_right);    pl = &(t->lg_left);
      LightRSetting(pr);      LightLSetting(pl);
    t->state = 'L';              // 起始状态为 'L'
   pr->init('Y', pr);            // 一开始设定 LightR 亮黄灯
   pl->init('R', pl);            // 也设定 LightL 亮红灯
   }
static void run(CTRL* t) {       // 执行期间循环于两个状态之中
    while(1) {
     switch(t->state) {
     case 'L':  t->goto_SW(t);   break;
     case 'S':  t->goto_LW(t);   break;
     }
   }
}
static void goto_LW(CTRL *t) {
      LightR *pr; LightL *pl;
      t->state = 'L';
      pr = &(t->lg_right);
       pl = &(t->lg_left);
       g_delay(3);                // 在'L' 状态停留 3 秒之后送出
      pr->ev_Y(pr);               // ev_Y 信息给两个红绿灯
      pl->ev_Y(pl);
   }
static void goto_SW(CTRL *t) {
   LightR *pr; LightL *pl;
   t->state = 'S';
   pr = &(t->lg_right);
    pl = &(t->lg_left);
   g_delay(1);                    // 在'S' 状态停留 1 秒之后送出
   pr->ev_RG(pr);                 // ev_RG 信息给两个红绿灯
   pl->ev_RG(pl);
   }

CTOR(CTRL)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(run, run)
  FUNCTION_SETTING(goto_LW, goto_LW)
  FUNCTION_SETTING(goto_SW, goto_SW)
END_CTOR
```

CTRL 对象与 LightR、LightL 对象的状态变化是不一样的，但是会有同步（Synchronize）的现象，CTRL 的状态变化会带动 LightR、LightL 对象的状态变化。

Step-4 编写 main()函数。

main()函数的代码

```
/* EX30-app-2.c  */
#include <REG51F.H>
#include <stdio.h>
#define LW_OOPC_STATIC
#include "EX30-ctrl.h"

extern CTRLSetting(CTRL*);
void main()  {
     CTRL ctrl;
     CTRLSetting(&ctrl);
     ctrl.init(&ctrl);
    ctrl.run(&ctrl);
   }
```

此程序执行时，输出如图 30-14 所示。

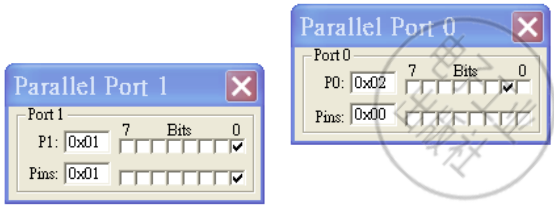


图 30-14

表示 LightR 处于 YELLOW 状态（P0^1 为 1），而 LightL 处于 RED 状态（P1^0 为 1）。停留一段时间之后，发出 ev_RG 信息给 LightR 和 LightL 对象，如图 30-15 所示。

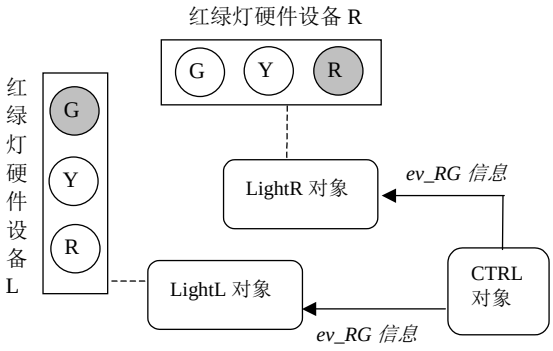


图 30-15

此时输出如图 30-16 所示。

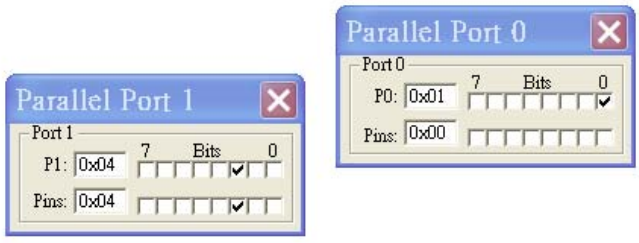


图 30-16

这表示 LightR 由 YELLOW 状态转为 RED 状态（P0^0 为 1）。而 LightL 由 RED 状态转为 GREEN 状态（P1^2 为 1）。停留一段时间之后，发出 ev_Y 信息给 LightR 和 LightL 对象。此时输出如图 30-17 所示。

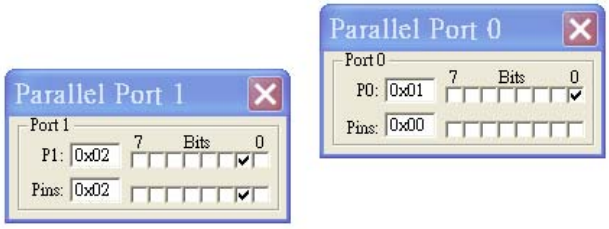


图 30-17

这表示 LightL 由 GREEN 状态转为 YELLOW 状态（P1^1 为 1）。LightL 则维持 RED 状态。如此一直周而复始，循环下去。

第 31 章　应用范例二

——以 Keil C51 表现七节 LED 灯的界面

31.1 界面用途：从硬件的 PnP 谈起

由于硬件结构上处处是接口，如果能将软件对象的接口与硬件组件接口同时考虑，对于嵌入式系统则会有极大的帮助。因为如此的话，软、硬件不仅可以同步设计，而且可以互相替换，软、硬件都能达到便捷 PnP 的美好境界。

31.1.1 硬件端口（Port）就是接口

8051 硬件的端口（Port）其实就是接口，通过接口可以连接到 LED、蜂鸣器及键盘（Keyboard）等。如图 31-1 所示。

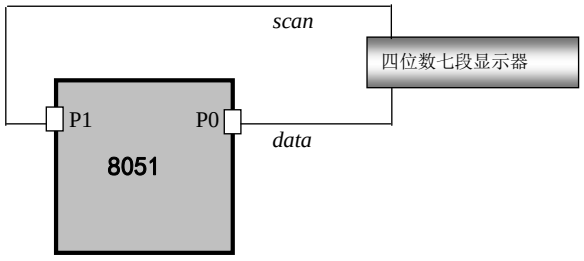


图 31-1

有了像 P0、P1 等接口，一个 8051 单片机就可以与 LED 显示器分合自如，也即俗称的 PnP（Plug and Play）。

31.1.2 软件接口

在 8051 硬件组件里含有嵌入式软件，如图 31-2 所示。

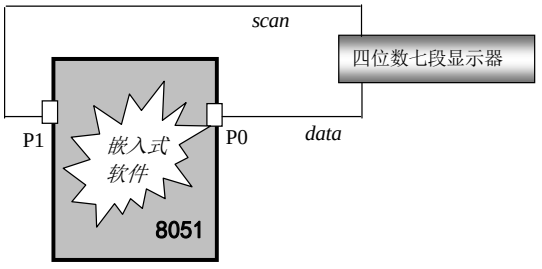


图 31-2

如果软件也有端口可连接硬件端口时，关系如图 31-3 所示。

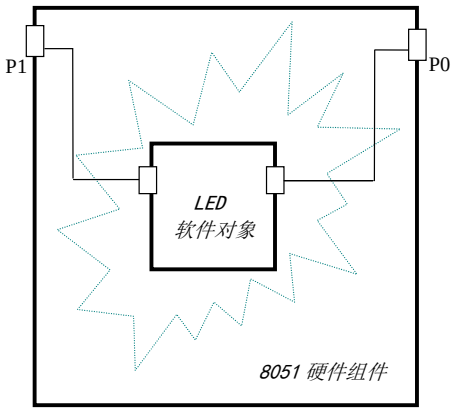


图 31-3

LED 对象的 data_port 连接到 8051 硬件组件 P0 时，其关系就相当于硬件组件 P0 连接到 LED 显示器。基于这个对应关系可得到下面的效果：

- 你已经理解：8051 硬件组件很容易将 LED 显示器 PnP 掉。
- 同理可知道：LED 软件对象很容易将 8051 硬件组件 PnP 掉。

当我们从 8051 硬件组件 PnP 掉 LED 显示器时，意味着：8051 硬件组件（如同车体）的重用（Reuse），而不是 LED 显示器（如同轮胎）的重用。许多人误解为接口设计是在追求小对象的重用，其实是不对的。想一想，当您的台灯灯管坏了，把坏灯管拔掉，换上新灯管，结果是：重用了整个台灯，而不是重用灯管。再想一想，一部 Benz 轿车轮胎坏掉了，把坏轮胎替换掉，整部汽车恢复完好，所以重用“整部汽车”了。当然，轮胎也有其重用的价值，只是价值不高，而重用整部汽车的价值非常高。

31.2 LED 显示器控制设计（1）

31.2.1 分析与设计

现在我们使用 Keil C 来逐步实现图 31-3 的软件接口。从图 31-4 可知 LED 类需要定义两个端口。

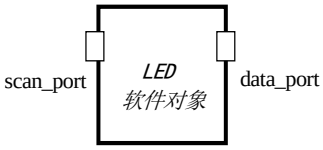


图 31-4

以 LW_OOPC 定义 LED 类如下：

```
CLASS(LED){
 void (*data_port)(char);
   void (*scan_port)(char);
   ……
};
```

接下来，设计两个 channel（可呈现为对象或函数）。例如，

```
static void channel_0(char y) {
  P0 = y;
}
static void channel_1(char x) {
  P1 = x;
}
```

其中，channel_0 已经连接到 P0，channel_1 已经连接到 P1。现在就拿 channel_0 连接到 LED 对象的 data_port，同时也拿 channel_1 连接到 LED 对象的 scan_port 上。其指令写法为：

```
    LED *t;
    ……
      t->data_port = channel_0;
      t->scan_port = channel_1;
```

于是就将 LED 对象与 8051 硬件组件连接起来了，如图 31-5 所示。

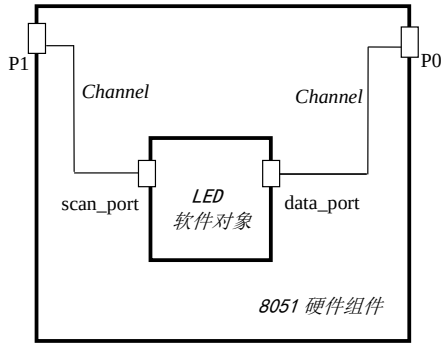


图 31-5

31.2.2 分析与设计

图 31-4 和图 31-5 都不是 UML 标准图示，现以 StarUML 工具绘制 UML 的组合结构图（Composite-Structure Diagram）如图 31-6 所示。

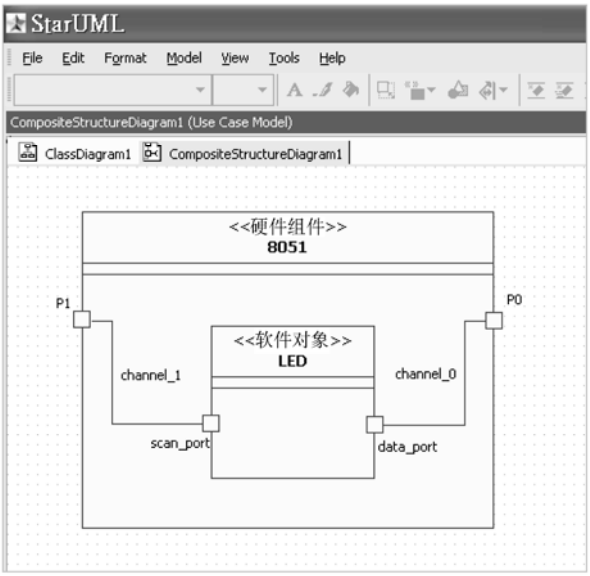


图 31-6

31.2.3 以 Keil C 实现范例

首先编写 LED 类的定义：

LED 类的定义代码

```
/* EX31-led.h */
#ifndef LED_H
#define LED_H

CLASS(LED) {
    void (*data_port)(char);
    void (*scan_port)(char);
    void (*run)(LED*);
};
#endif
```

定义了 data_port 和 scan_port 两个函数来实现 LED 对象对外的端口。接着定义一般软件内部函数 run()来处理对象内部的事务。如下述程序代码：

LED 类的实现代码

```
/*  EX31-led.c   */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex31-led.h"

void g_delay(unsigned long ms) {
   long i, j;
   for(i=0; i<ms; i++)
      for(j=0; j<120; j++);
```

```
}
static unsigned char SEGTAB[] =
     {0xC0, 0xf9, 0xa4, 0xb0, 0x99, 0x92, 0x83, 0xf8, 0x80, 0x98};
static unsigned char SCANLINE[] = {0xf7, 0xfb, 0xfd, 0xfe};
static void channel_1(char x) {
   P1 = x;
}
static void channel_0(char y) {
   P0 = y;
}
static void run(LED *t) {
    unsigned char ch, sc;
    t->data_port = channel_0;
    t->scan_port = channel_1;

    /*
---------------------------------------------------------------------- */
    P0 = 0xf0;
     ch = SEGTAB[0];    sc = SCANLINE[0];
    t->scan_port(0xff);     t->data_port(ch);     t->scan_port(sc);
    g_delay(20000);
    /*
---------------------------------------------------------------------- */
    ch = SEGTAB[9];  sc = SCANLINE[1];
    t->scan_port(0xff);    t->data_port(ch);    t->scan_port(sc);
    g_delay(20000);
}

CTOR(LED)
    FUNCTION_SETTING(run, run)
END_CTOR
```

执行 run()函数里的指令：

```
    t->data_port = channel_0;
    t->scan_port = channel_1;
```

就把 channel_0()和 channel_1()连接到 LED 对象的两个端口上。接下来的指令：

```
ch = SEGTAB[0];     //取得'0'这个数字的 data_code
sc = SCANLINE[0];   //取得显示在第 0 个位置的 scan_code
t->scan_port(0xff); //清除
t->data_port(ch);   //送出 data_code 到 P0
t->scan_port(sc);   //送出 scan_code 到 P1
```

于是 LED 显示器的第 1 位数出现'0'。再来的指令：

```
ch = SEGTAB[9];     //取得'9'这个数字的 data_code
sc = SCANLINE[1];   //取得显示在第 1 个位置的 scan_code
t->scan_port(0xff); //清除
t->data_port(ch);   //送出 data_code 到 P0
t->scan_port(sc);   //送出 scan_code 到 P1
```

于是 LED 显示器的第 2 位数出现'9'。最后编写 main()函数：

main()主函数的实现代码

```
/* EX31-ap-1.c  */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex31-led.h"

extern void* LEDNew();
char xdata MemPool[1024];
void main (void) {
   LED *led;
   init_mempool(MemPool,sizeof(MemPool));
   led = LEDNew();
   led->run(led);
}
```

前面图 31-6 是 UML 的组合结构图，善于呈现空间感，并且凸显软件内在（Software Inside）的角色。一旦软、硬件接口厘清了，就可使用 UML 类图（Class Diagram）来呈现详细的软件对象接口，如图 31-7 所示。

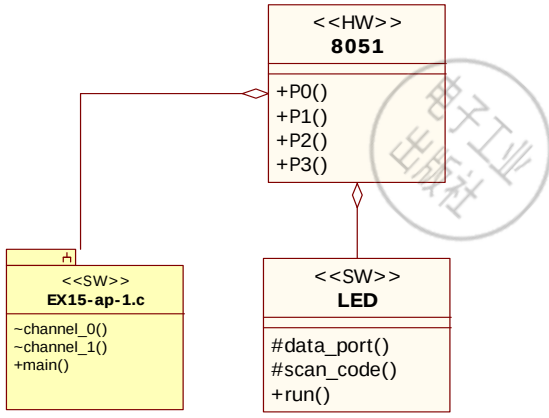


图 31-7

该图包含软件和硬件组件，并且采用 UML 的标签<<HW>>和<<SW>>来区别软件对象与硬件组件。其中，

- +P0 和+main()的'+'号表示这是对外沟通的端口（或函数）。
- #data_port 的'#'号表示在自己（Local）的.C 文件里并未实现（Implement）这个函数。
- ~channel_0 的'~'号表示它实现在自己的.C 文件里，它将连接到别的对象的'#'型（如#scan_port）函数，成为该'#'型函数的实现函数。
- 图中，比较有趣的是如何看待 P0、P1 等硬件的端口。我们会采取两种观点：

1. 外部观点（Outside In View）：我们把 LED 显示器连接到 P0 和 P1 上，就如同把轮胎装

到轮盘上。此时我们通过 P0 及 P1 来使用 8051 组件内部的实现，所以嵌入于 8051 的 LED 软件对象成为 P0 及 P1 的实现（或称为实现）。

2. 内部观点（Inside Out View）：从软件往外看，P0 和 P1 连接到 LED 显示器，而软件 LED 对象通过 P0 和 P1 控制 LED 显示器，此时 LED 显示器就成为 P0 和 P1 的实现了。

以上两个观点，我们无法二选一，只有同时兼顾两个观点，才能达到周详的全面性设计。

31.3 LED 显示器控制设计（2）

31.3.1 分析与设计

在上一节的图 31-7 里，设计了 一个 LED 软件对象，通过 P0 和 P1 来控制 LED 显示器。如果系统继续扩大，例如 P3 连接到蜂鸣器，我们也会设计一个 Beep 软件对象来控制蜂鸣器。像 LED、Beep 等对象增多了，我们通常会额外规划一个控制对象，担任指挥中心。如图 31-8 所示。

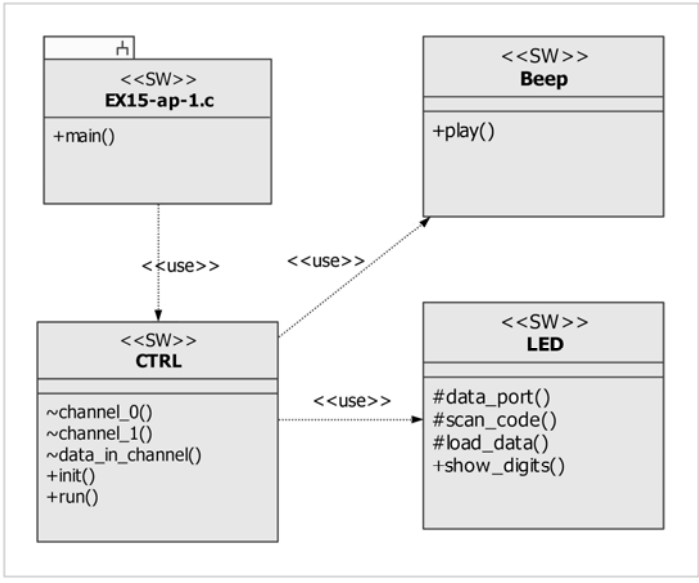


图 31-8

CTRL（即 Controller 的简写）担任了原来 main()函数的工作，因而简化了 main()。此外，CTRL 扮演软件主机板（Software Motherboard）的角色。LED 对象和 Beep 对象就 PnP 到 CTRL 对象上，就如同一般 PC 计算机的 Keyboard、硬盘等连接到硬件主机板一般。于是呈现出标准的嵌入式软、硬件整合架构：

- main()启动 CTRL 对象开始运行。

- CTRL 依循复杂的逻辑规则，有节奏（可以使用硬件的 Timer 组件）且井然有序地指挥 LED 和 Beep 等小对象运行。
- LED 和 Beep 等小对象通过 P0、P1 等接口而控制外部的 LED 显示器等硬件组件。

31.3.2 以 Keil C 实现范例

首先编写 LED 类别的定义：

LED 类的定义代码

```
/* EX31-led.h */
#ifndef LED_H
#define LED_H

CLASS(LED)
{
   void (*scan_port)(char);
   void (*data_port)(char);
   void (*load_data)(LED *t);
   void (*show_digits)(LED *t);
   int digits[4];
};
#endif
```

定义了 data_port 和 scan_port 两个函数来实现 LED 对象对外的端口。并声明一个 int 型态的数组：digits[]。如果 digits[]的内容为{3，5，7，1}，就会在 LED 显示器出现这 4 个阿拉伯数字。

LED 类的实现代码

```
/*  EX31-led.c   */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex31-led.h"

void g_delay(unsigned long ms) {
   long i, j;
   for(i=0; i<ms; i++)
      for(j=0; j<120; j++);
}

static unsigned char SEGTAB[] =
{0xC0, 0xf9, 0xa4, 0xb0, 0x99, 0x92, 0x83, 0xf8, 0x80, 0x98};
static unsigned char SCANLINE[] = {0xf7, 0xfb, 0xfd, 0xfe};

static void show_digits(LED *t) {
   unsigned char ch, sc;   int i;
P0 = 0xf0;
   for(i=0; i<4; i++) {
ch = SEGTAB[t->digits[i]];
sc = SCANLINE[i];
t->scan_port(0xff);  t->data_port(ch);       t->scan_port(sc);
g_delay(20000);
      }
```

```
}
CTOR(LED)
    FUNCTION_SETTING(show_digits, show_digits)
END_CTOR
```

此 show_digits()函数从 SEGTAB[]取得数字的 data_code，并从 SCANLINE[]取得显示在第 i 个位置的 scan_code。然后执行下述指令：

```
t->scan_port(0xff);         //清除
t->data_port(ch);           //送出 data_code 到 P0
t->scan_port(sc);           //送出 scan_code 到 P1
```

于是 LED 显示器的第 i 位数出现该阿拉伯数字。再来编写 CTRL 类：

CTRL 类的定义代码

```
/* EX31-ctrl.h */
#ifndef CTRL_H
#define CTRL_H
#include "ex31-led.h"

CLASS(CTRL)
{     void (*init)(CTRL *t);
    void (*run)(CTRL *t);
    LED *ple;
};
#endif
```

CTRL 类的实现代码

```
/* EX31-ctrl.c  */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex31-led.h"
#include "ex31-ctrl.h"

extern void* LEDNew();
extern void g_delay(unsigned int);
static void channel_1(char x) {
   P1 = x;
}
static void channel_0(char y) {
   P0 = y;
}
static int buffer[10] = {7, 2, 0, 4};
static void data_in_channel(LED *t) {
    t->digits[0] = buffer[0];
    t->digits[1] = buffer[1];
    t->digits[2] = buffer[2];
    t->digits[3] = buffer[3];
}
static void init(CTRL *t){
   t->ple = LEDNew();
   t->ple->scan_port = channel_1;
   t->ple->data_port = channel_0;
   t->ple->load_data = data_in_channel;
}
```

```
static void run(CTRL *t) {
  while(1) {
     t->ple->load_data(t->ple);
     t->ple->show_digits(t->ple);
     P0 = 0x00;      P1 = 0x00;
     g_delay(10000);
    }
}
CTOR(CTRL)
   FUNCTION_SETTING(init, init)
   FUNCTION_SETTING(run, run)
END_CTOR
```

一开始在 init()函数里的指令：

```
t->ple->scan_port = channel_1;
t->ple->data_port = channel_0;
t->ple->load_data = data_in_channel;
```

执行到 run()函数的指令：

```
t->ple->load_data(t->ple);
t->ple->show_digits(t->ple);
```

再通过 load_data()函数来填入 digits[]的内容，然后以 show_digits()函数显示出来。最后编写主程序：

main()主函数的实现代码

```
/* EX31-ap-2.c  */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex31-ctrl.h"

extern void* CTRLNew();
char xdata MemPool[1024];

void main (void) {
  CTRL *controller;
  init_mempool(MemPool,sizeof(MemPool));
  /* ------------------------- */
  controller = CTRLNew();
  controller->init(controller);
  controller->run(controller);
}
```

以上范例，凸显了 CTRL 对象的重要性。如果系统继续扩大，例如 P0 等连接到蜂鸣器、键盘等更多硬件组件时，CTRL 就更显得举足轻重了。因为它与硬件主板所扮演的角色是一致的。

31.4 LED 显示器控制设计（3）

在上一个范例里，LED 对象的 data_port 和 scan_port 两个接口都是以函数形式出现的，

其实它们还能以对象指针形式呈现。意思是，上一个例子，将 channel_0 和 channel_1 定义为函数，其实也能将它们定义为类。在本范例里，就来展示这种做法。如图 31-9 所示的 UML 类图。

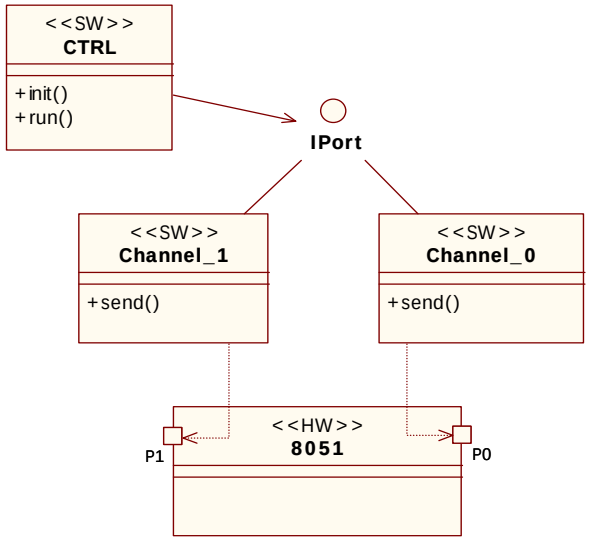


图 31-9

首先替 Channel_0 和 Channel_1 类定义其共同的接口 IPort。

IPort 接口的定义文件

```
/* EX31-ip.h */
#ifndef IP_H
#define IP_H

INTERFACE(IPort)
{
   void (*send)(char);
};
#endif
```

Channel_0 和 Channel_1 类的定义及实现代码

```
/*  EX31-channel.c   */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex31-ip.h"

CLASS(Channel_0)
{
  IMPLEMENTS(IPort);
};

static void send_code_P0(char x) {
```

```
    P0 = x;
}
CTOR(Channel_0)
    FUNCTION_SETTING(IPort.send, send_code_P0);
END_CTOR
/*------------------------------------------- */
CLASS(Channel_1)
{
  IMPLEMENTS(IPort);
};

static void send_code_P1(char y) {
    P1 = y;
}
CTOR(Channel_1)
    FUNCTION_SETTING(IPort.send, send_code_P1);
END_CTOR
```

LED 类的定义文件

```
/* EX31-led.h */
#ifndef LED_H
#define LED_H
#include "ex31-ip.h"

CLASS(LED)
{
    void (*load_data)(LED *t);
    void (*show_data)(LED *t);
    IPort *scan_port, *data_port;
    int digits[4];
};
#endif
```

其中，scan_port 和 data_port 变成为 IPort* 类型的指针，它们可以指向 Channel_0 或 Channel_1 类的对象。

LED 类的实现代码

```
/*  EX31-led.c   */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex31-led.h"

void g_delay(unsigned int ms) {
   int i, j;
   for(i=0; i<ms; i++)
      for(j=0; j<120; j++);
}

static unsigned char SEGTAB[] =
     {0xC0, 0xf9, 0xa4, 0xb0, 0x99, 0x92, 0x83, 0xf8, 0x80, 0x98};
static unsigned char SCANLINE[] = {0xf7, 0xfb, 0xfd, 0xfe};

static void show_data(LED *t) {
    unsigned char ch, sc;    int i;
    IPort *pss = t->scan_port;
    IPort *pdd = t->data_port;
```

```
    P0 = 0xf0;
    for(i=0; i<4; i++)  {
        ch = SEGTAB[t->digits[i]];       sc = SCANLINE[i];
        pss->send(0xff);       pdd->send(ch);       pss->send(sc);
        g_delay(20000);
       }
}
CTOR(LED)
    FUNCTION_SETTING(show_data, show_data)
END_CTOR
```

此 show_data()函数从 SEGTAB[]取得数字的 data_code，并从 SCANLINE[]取得显示在第 i 个位置的 scan_code。然后执行下述指令：

```
t->scan_port(0xff);        //清除
t->data_port(ch);          //送出 data_code 到 P0
t->scan_port(sc);          //送出 scan_code 到 P1
```

于是 LED 显示器的第 i 位数出现该阿拉伯数字。

CTRL 类的定义代码

```
/* EX31-ctrl.h */
#ifndef CTRL_H
#define CTRL_H
#include "ex31-ip.h"
#include "ex31-led.h"

CLASS(CTRL)
{
     void (*init)(CTRL *t);
    void (*run)(CTRL *t);
    LED *ple;
    IPort *ps, *pd;
};
#endif
```

CTRL 类的实现代码

```
/* EX31-ctrl.c  */
#include <REG52.H>
#include "lw_oopc_kc.h"
//#include "ex31-led.h"
#include "ex31-ctrl.h"

extern void* LEDNew();
extern void* Channel_0New();
extern void* Channel_1New();
extern void g_delay(unsigned int);

static int buffer[10] = {7, 2, 0, 4};

static void load_data(LED *t) {
    t->digits[0] = buffer[0];
    t->digits[1] = buffer[1];
    t->digits[2] = buffer[2];
    t->digits[3] = buffer[3];
```

```
}

static void init(CTRL *t)
{
   t->ple = LEDNew();
   t->ps = Channel_0New();       t->pd = Channel_1New();
   t->ple->load_data = load_data;
   t->ple->scan_port = t->ps;
   t->ple->data_port = t->pd;
}

static void run(CTRL *t) {
      t->ple->load_data(t->ple);   t->ple->show_data(t->ple);
}
CTOR(CTRL)
    FUNCTION_SETTING(init, init)
    FUNCTION_SETTING(run, run)
END_CTOR
```

main()主函数的定义及实现代码

```
/* EX31-ap-3.c  */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex31-ctrl.h"

extern void* CTRLNew();
char xdata MemPool[1024];

void main (void) {
   CTRL *drive;
   init_mempool(MemPool,sizeof(MemPool));
   /* ------------------------- */
   drive = CTRLNew();
   drive->init(drive);
   drive->run(drive);
}
```


此范例凸显了 LED 等软件对象接口的多样化，可以呈现为函数，也可呈现为对象指针，至于哪个较适当，就得视你的需要而定了。

综合本章所述，当我们谈到“LED”时，这个词汇就很适合当作类的名称。将跟它有关的函数汇集起来，然后再把这些函数公用的数据项集合起来，就成为一个很棒的类了。简而言之，类设计的过程为：

Step-1 关注术语或词汇，用它们来作类名称。

Step-2 把跟某一词汇有关的函数汇集起来。

Step-3 把函数共享的数据项汇集起来。

当你找到 LED 这个重要词汇，就找到重要类了。这个类将产生对象来控制一个 4 位数 7 节 LED 显示灯。所以 LED 对象将有两个重要的函数：

- 送出 data_code 到硬件的 P0 或其他 Port。
- 送出 scan_code 到硬件的 P1 或其他 Port。

此外，可能需要一个 load_data()函数，由外界给予有关的数据，经过一些处理之后，分别送出 data_code 和 scan_code。因此从行为层面找到了 LED 类的重要函数。此时，LED 对象的 data_port 和 scan_port 两个接口都以函数形式出现，其实它们还能以对象指针形式呈现。

有人常问：设计嵌入式软件要做什么呢？大家都会很有把握地回答：用软件来控制硬件。“控制”是一项任务，也是一个重要的词汇，于是设计一个“CTRL”（即 Controller 名词的简写）来担任这项任务。于是 CTRL（软件）对象指挥 LED（软件）对象，然后 LED 对象再送出信号到硬件的 P0、P1 等硬件接口来控制像 4 位数 7 节 LED 显示灯等对象。

第 32 章　应用范例三

——Keil C51 结合 IoC 设计模式

32.1 模式观念

模式（Pattern）是人们遭遇到特定问题时惯用的应对方式。模式可用来解决问题，而且是有效、可靠的。掌握愈多模式，运用愈成熟，就愈是杰出的设计专家。换句话说，模式是专家们针对特定环境（Context）下经常出现的问题，从经验中总结的惯用解决之道（Solution）。例如，围棋有棋谱、烹饪有食谱、武功有招式、战争有兵法等，皆是专家和高手的经验心得。

模式是从经验中孕育出来、去芜存菁后的间接性方案（Indirect Solution）。 例如，孔明的“空城计”是细心推敲而得的构思，而不是信手拈来的简单直接方案，必须从丰富的经验之中提炼出来。模式引导您去套用它，修正它、加上外在环境因素，而得到具体可行的方案（Solution）。它告诉您理想的方案是什么、有哪些特性；同时也告诉您一些规则，让您依循，进而在您的脑海里产生适合环境的具体方案。只要灵活掌握规则、充分融入大环境因素，即能瞬间得到具体有效的方案。所以建筑大师亚历山大（Christopher Alexander）做了如下的定义：

“模式（Pattern）是某外在环境（Context）下，对特定问题（Problem）的惯用解决之道（Solution）。”

举个例子，在平坦的校园里（Context），为了维护一个安静又可以交流智慧的学习环境，我们发现下述两项需求是互相冲突的：

需求 1: 学生需要交通工具代步。

需求 2: 校园空气必须保持清洁。

于是，就产生问题（Problem）了，此时矛盾该如何化解呢？在一般校园里，惯用解决方法（Solution）是：规定在校园中只能骑自行车，不能骑有污染的摩托车或开汽车。这就是一个模式了，任何校园都可以参考这个模式，实施之后可获得安静又安全的校园环境。然而，此法是否有效，还取决于外在环境（Context）。例如，在不平坦的清华大学校园中，上述方法就需要做些修正了。这是因为外在环境（Context）不同的缘故。

例如看到前面有恶狗挡路，苦思对策，脑海中突然想到三十六计的“调虎离山”，灵光一现举一反三，丢个肉包子引诱恶狗离开，问题于是迎刃而解了。虽然“调虎离山”与“诱狗离路”情境（Context）不尽相同，但是解决方法却很类似。

例如刚才的清华大学校园交通问题，修正为：提供电动车供学生在校园中使用。就是一个有效的解决方案。

32.2 软件设计模式

32.2.1 Why 设计模式

模式是专家惯用的解决之道。只要有专家，就会有各式各样的模式出现。例如，在软件开发上，有系统的分析模式、架构模式，以及细节设计模式（Design Pattern）等。自从 1991

年以来，亚历山大的模式理论逐渐应用于软件的设计上，用以解决软件设计中遇到的问题。

在设计过程中，常会面临环境的各种需求和条件，来自不同方面的需求可能会互相冲突而呈现不和谐的现象。因而不断运用模式来化解冲突使其变得均衡和谐，亦即不断把环境因素注入模式中而产生有效的方案来使冲突的力量不再互相作用。有效的设计专家，会大量运用其惯用的模式，而不会一切从头创造新方案（Reinvent the wheel）。

模式运用得好，能化解冲突，问题迎刃而解，自然令人感到舒畅。好的设计师以流畅的方式组合模式而成的设计，自然令人感到满意。

由于上述缘由，模式在程序设计上扮演着重要角色——极佳的沟通媒介。当程序员使用模式时，使用者能轻易地由模式体会出程序的深层意义，从经验中千锤百炼出来的设计模式，配合现实条件，形成精致的细节设计，系统自然散发出高雅的韵味。

32.2.2 设计模式的起源

1964 年，著名建筑学家 Christopher Alexander 出版了一本书：

Notes on the Synthesis of Form

他提出“型的组合”观念，认为设计师可经由型的组合来化解环境中互相冲突的需求，使冲突变成为和谐。后来他把型的观念改称为“模式”（Pattern），模式可引导设计师逐步创造出形形色色的奇特组合，以便化解互相冲突的需求。到了 20 世纪 70 年代，Alexander 任教于加州柏克莱大学，和其他同事共同研究 Pattern 观念，并出版了 4 本书：

1. The Timeless Way of Building

   ——完整地介绍他的 Pattern 观念，以及 Pattern Language 观念。

2. A Pattern Language

   ——实际列举了 253 个建筑方面的模式。

3. The Oregon Experiment

   ——叙述在俄勒冈大学的实验过程。

4. The Production of Horses

   ——叙述在墨西哥的实验情形，以及该实验并未成功的原因。

自从 20 世纪 80 年代起，随着面向对象技术的日益普及，这时 Alexander 的模式观念再度影响到软件的设计方法。1987 年，Ward Cunningham 和 Kent Beck 两人首先尝试将面向对象技术与模式观念结合起来。他们的研究着重于使用者接口（User Interface）方面，并在 OOPSLA/87 会议上发表其成果。然而，他们的研究并未立即引起重视。

到了 1990 年，在欧洲的 OOPSLA/90 会议上，由 Bruce Andreson 主持的“Architectural

Handbook”研讨会中，Erich Gamma 和 Richard Helm 等人开始谈论有关模式的话题。1991 年 Erich Gamma 完成了他的博士论文——“Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools ”

在 OOPSLA/92 会议上，Andreson 再次主持研讨会，模式观念逐渐成为热门的话题。在研讨会中，伊利诺大学教授 Ralph Johnson 发表其有关模式与应用架构（Application Framework）关系的研究成果。Peter Coad 在 ACM 期刊上发表了 OOA 方面的 7 个模式。

1995 年，Erich Gamma 和 Ralph Johnson 等人出版 *Design Patterns: Elements of Reusable Object-Oriented Software* 一书，成为软件设计模式的经典。也是软件模式发展上极为重要的里程碑。

自从 Gamma 的“Design Pattern”一书上市之后，逐渐在软件业界流行起来，十多年来，新的软件模式不断涌现，已经处处可见到软件模式的应用已初具规模。

32.3 IoC 模式简介

以索尼（SONY）公司的随身听产品开发为例，说明了良好的相依性设计和管理可以得到的满意效果。在软件系统设计上也不例外，例如软件开发时常遭遇如下问题：

软件模块（Module）之间的耦合度太高，不易个别替换模块，而失去了软件的“软性”与“和谐”。

解决方案是：强化模块间的独立性（即降低不必要的相依性），以提高系统弹性（Flexibility），降低软件系统整合与维护的成本。

32.3.1 传统细节设计方法：高度耦合性

首先以 Keil C 程序来叙述“问题”之所在：

Integer 类的定义代码

```
/* EX32-int.h */

CLASS(Integer)
{
  void (*init)(Integer*);
  void (*display)(Integer*);
  int value;
};
```

Integer 类的实现代码

```
/* EX32-int.c */
#include <REG52.H>
#include <stdio.h>
#include "lw_oopc_kc.h"
#include "ex32-int.h"
```

```
static void init(Integer *t)
   {  t->value = 100;  }

static void display(Integer *t)
  {  P0 = (char)t->value;  }

CTOR(Integer)
   FUNCTION_SETTING(init, init);
   FUNCTION_SETTING(display, display);
END_CTOR
```

Document 类及 main()主函数的定义及实现代码

```
/* EX32-ap-1.c  */
#include <REG52.H>
#include <stdio.h>
#include "lw_oopc_kc.h"
#include "ex32-int.h"

extern void* IntegerNew();
CLASS(Document) {
     void (*init)(Document *t);
     void (*display)(Document *t);
     Integer *pi;
  };

static void init(Document *t) {
     t->pi = IntegerNew();
     t->pi->init(t->pi);
 }
static void display(Document *t) {
     t->pi->display(t->pi);
}
CTOR(Document)
   FUNCTION_SETTING(init, init);
   FUNCTION_SETTING(display, display);
END_CTOR
/* ---------------------------------- */
char xdata MemPool[1024];
void main() {
   Document *pdoc;
   init_mempool(MemPool,sizeof(MemPool));

   pdoc = DocumentNew();
   pdoc->init(pdoc);
   pdoc->display(pdoc);
   while(1) {
      ;
   }
}
```

Document 与 Integer 两个类的耦合度很高，原因是 Document 类的指令：

```
   CLASS(Document) {
     ……..
     Integer *pi;
    };
   static void init(Document *t) {
```

```
    t->pi = IntegerNew();
    ……..
}
```

直接使用 Integer 字眼，且使用两次。于是，“若必须将 Integer 类名称改为 Float 时，必须得更换 Document 类中的 Integer 字眼”。亦即，替换 Integer 类时，会牵连到 Document 类，替换过程将会很麻烦。

32.3.2 使用接口方法：中等耦合性

在前面各章里，本书极力主张善用接口，它可以大幅提升类的可替换性。例如上述的 Integer 类如果提供一个界面给 Document 使用的话，确实能有效降低两者的耦合度。例如：

IDisplay 接口的定义代码

```
/* EX32-idisp.h */

INTERFACE(IDisplay)
{
  void (*init)(void*);
  void (*display)(void*);
  int value;
};
```

Integer 类的定义代码

```
/* EX32-int.h */

CLASS(Integer)
{
  IMPLEMENTS(IDisplay);
  int value;
};
```

Integer 类的实现代码

```
/* EX32-int.c */
#include <REG52.H>
#include <stdio.h>
#include "lw_oopc_kc.h"
#include "ex32-idisp.h"
#include "ex32-int.h"

static void init(Integer *t) {
  t->value = 100;
  }
static void display(Integer *t) {
  P0 = (char)t->value;
 }
CTOR(Integer)
   FUNCTION_SETTING(IDisplay.init, init);
   FUNCTION_SETTING(IDisplay.display, display);
END_CTOR
```

Document 类及 main()主函数的定义及实现代码

```
/* EX32-ap-2.c  */
#include <REG52.H>
#include <stdio.h>
#include "lw_oopc_kc.h"
#include "ex32-idisp.h"
#include "ex32-int.h"

extern void* IntegerNew();

CLASS(Document)
  {
    void (*init)(Document *t);
    void (*display)(Document *t);
    IDisplay *pi;
  };
static void init(Document *t){
    t->pi = IntegerNew();
    t->pi->init(t->pi);
 }
static void display(Document *t){
    t->pi->display(t->pi);
 }
CTOR(Document)
   FUNCTION_SETTING(init, init);
   FUNCTION_SETTING(display, display);
END_CTOR

/* ---------------------------------- */
char xdata MemPool[1024];
void main() {
   Document *pdoc;
   init_mempool(MemPool,sizeof(MemPool));
   pdoc = DocumentNew();
   pdoc->init(pdoc);
   pdoc->display(pdoc);
   while(1) {
     ;
   }
}
```


删掉 1 个 Integer 字眼，已有些进步了，不是吗？但问题尚未全部解决。在 init()函数：

```
static void init(Document *t) {
  t->pi = IntegerNew();
  ……..
}
```

Integer 字眼仍留在 Document 类里，还没达到完美境界。

32.3.3 使用 IoC 模式：低度耦合性

为了更进一步解决上述问题，IoC 模式（Pattern）就派上用场了。它藉由 Factory 类来包装 IntegerNew()指令，于是“Integer”字眼就不再出现于 Document 类里面了。如下面的程序代码：

IDisplay 接口的定义代码

```
/* ex32-idisp.h */
#ifndef _IDISP_H
#define _IDISP_H

INTERFACE(IDisplay)
{
  void (*init)(void*);
  void (*display)(void*);
};
#endif
```

Integer 类的定义代码

```
/* ex32-int.h */

CLASS(Integer)
{
  IMPLEMENTS(IDisplay);
  int value;
};
```

Integer 类的实现代码

```
/* EX32-int.c */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex32-idisp.h"
#include "ex32-int.h"

static void init(Integer *t) {
    t->value = 15;
  }
static void display(Integer *t){
   P0 = (char)t->value;
}
CTOR(Integer)
   FUNCTION_SETTING(IDisplay.init, init);
   FUNCTION_SETTING(IDisplay.display, display);
END_CTOR
```

Factory 类的定义代码

```
/*   EX32-fac.h */
#include "ex32-idisp.h"

CLASS(Factory)
{
   IDisplay* (*get_object)();
};
```

Factory 类的实现代码

```
/* ex32-fac.c */
#include "lw_oopc_kc.h"
#include "ex32-idisp.h"
#include "ex32-int.h"
#include "ex32-fac.h"
```

```
extern void* IntegerNew();

static IDisplay* get_object(){
    IDisplay *pi;
    pi = IntegerNew();    pi->init(pi);
    return pi;
}
CTOR(Factory)
  FUNCTION_SETTING(get_object, get_object)
END_CTOR
```

Document 类及 main()主函数的定义及实现代码

```
/* EX32-ap-3.c  */
#include "lw_oopc_kc.h"
#include "ex32-idisp.h"
#include "ex32-fac.h"

extern void* FactoryNew();
Factory *pfac;

CLASS(Document)
   {
     void (*display)();
   };

static void display()
   {  IDisplay *pi;
        pi = pfac->get_object();
     pi->init(pi);     pi->display(pi);
    }
CTOR(Document)
    FUNCTION_SETTING(display, display);
END_CTOR

/* ---------------------------------- */
char xdata MemPool[2048];
void main() {
Document *pd;
init_mempool(MemPool,sizeof(MemPool));
pfac = FactoryNew();
pd = DocumentNew();
pd->display();
    while(1) {
       ;
    }
}
```

上述 Document 类不再使用“Integer”字眼，修改时就不会发生涟漪现象，从而具备了良好的可替换性。例如在 Sprint Framework 环境里，只要修改 Configuration 文件内容，就能改变 Factory 对象的行为。

当然也可以直接更改 Factory 类而轻易达到替换 Integer 类的目的。即使直接更改 Factory 类，也是轻松愉快的，因为“Integer”字眼集中于 Factory 类内，只需要更改一个类就行了。例如，有一天必须将 Integer 类改为 FloatNumber 类时，只需更改 Factory 如下的内容：

```
static IDisplay* get_object(){
     IDisplay *pi;
     pi = FloatNumberNew();    pi->init(pi);
     return pi;
}
```

只更改了一个指令，其他都保持原状，很棒吧！使用 IoC 模式，一切变得简单顺手多了。

32.4 以 8051 控制 LED 显示器为例

在前面各章里，曾经借 LED 显示器的范例说明了各种优雅的程序写法。本节将进一步介绍如何将 IoC 模式运用到 LED 等对象沟通上，以便更细化嵌入式系统的设计。

32.4.1 分析与设计

IoC 模式最大的特色就是 Factory 对象的角色，顾名思义，它是由对象的“工厂”负责产生各个对象，并建立对象间的关系。例如，如图 32-1 所示的黑色菱形表示 Factory 对象产生 CTRL、LED_P0 及 LED_P1 类的对象，所以它拥有（就是黑色菱形的原意）这些对象。

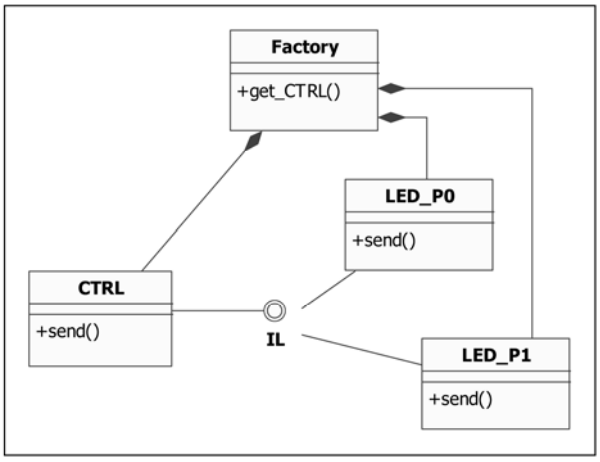


图 32-1

在程序开始执行时，main()函数产生 Factory 对象，此时 Factory 的 init()会产生 CTRL、LED_P0 及 LED_P1 类的对象。接着，main()会调用 Factory 的 get_CTRL()函数，此时 Factory 将 CTRL 对象指针传给 main()函数。于是，main()调用 CTRL 的 send()函数，此 send()再通过 IL 接口而调用到 LED_P0 和 LED_P1 的 send()函数。

在此 IoC 模式下，可以随时更改 LED_P0 或 LED_P1 类名称，且不会影响到 CTRL，因为在 CTRL 类里，都没有用到 LED_P0 或 LED_P1 类名称。这就是 Factory 的功劳。

32.4.2 以 Keil C 实作范例

现在，以对象导向 Keil C 来实现图 32-1 的设计，如下述程序代码：

IL 接口的定义文件

```
/*  EX32-il.h */
#ifndef _IL_H
#define _IL_H
#include "lw_oopc_kc.h"

INTERFACE(IL)
{
   void (*send)(unsigned char);
};
#endif
```

LED_P0 及 LED_P1 类的定义文件

```
/* EX32-led.h   */
#include "lw_oopc_kc.h"
#include "ex32-il.h"

CLASS(LED_P0)
{
  IMPLEMENTS(IL);
};

CLASS(LED_P1)
{
  IMPLEMENTS(IL);
};
```


LED_P0 及 LED_P1 类的实现代码

```
/* EX32-led.c */
#include <REG51F.H>
#include "lw_oopc_kc.h"
#include "ex32-led.h"

static void send(unsigned char hx) {
  P0 = hx;
}
CTOR(LED_P0)
  FUNCTION_SETTING(IL.send, send)
END_CTOR

//---------------------------------------
static void send2(unsigned char hx) {
  P1 = hx;
}
CTOR(LED_P1)
  FUNCTION_SETTING(IL.send, send2)
END_CTOR
```

CTRL 类的定义代码

```
/*  EX32-ctrl.h */
#include "ex32-il.h"

CLASS(CTRL)
{   void (*send)(CTRL*, char);
    IL *pa, *pb;
};
```

CTRL 类的实现代码

```
/* EX32-ctrl.c */
#include <REG51F.H>
#include "lw_oopc_kc.h"
#include "ex32-ctrl.h"

static void send(CTRL* t, char hx){
    t->pa->send(hx);    t->pb->send(hx);
}
CTOR(CTRL)
  FUNCTION_SETTING(send, send)
END_CTOR
```

CTRL 与 LED_P0 或 LED_P1 对象之间是通过 IL 接口沟通的，而且 CTRL 又不必亲自产生 LED_P0 或 LED_P1 对象，所以维持低度耦合性。

Factory 类的定义文件

```
/*  EX32-factory.h */
#include "ex32-il.h"

CLASS(Factory)
{
   void (*init)(Factory*);
   CTRL* (*get_CTRL)(Factory*);
   void (*destroy)(Factory*);
   IL* list[2];
   CTRL *ctrl;
};
```

Factory 类的实现代码

```
/* EX32-factory.c */
#include "lw_oopc_kc.h"
#include "ex32-ctrl.h"
#include "ex32-factory.h"

extern void* CTRLNew();
extern void* LED_P0New();
extern void* LED_P1New();

static void init(Factory *t) {
    t->ctrl = CTRLNew();
    t->list[0] = (IL*)LED_P0New();    t->list[1] = (IL*)LED_P1New();
    t->ctrl->pa = t->list[0];         t->ctrl->pb = t->list[1];
}
```

```
static CTRL* get_CTRL(Factory *t){
     return t->ctrl;
  }
static void destroy(Factory *t) {
   free(t->list[0]);    free(t->list[1]);
}
CTOR(Factory)
  FUNCTION_SETTING(init, init)
  FUNCTION_SETTING(get_CTRL, get_CTRL)
  FUNCTION_SETTING(destroy, destroy)
END_CTOR
```

CTRL 的 init()函数内容

```
static void init(Factory *t) {
    t->ctrl = CTRLNew();
    t->list[0] = (IL*)LED_P0New();
    t->list[1] = (IL*)LED_P1New();
    ……
    }
```

就产生 3 个对象。还有，此 init()函数的内容：

```
static void init(Factory *t) {
 ……
       t->ctrl->pa = t->list[0];
       t->ctrl->pb = t->list[1];
    }
```

就把 CTRL 与 LED_P0 或 LED_P1 对象连接起来。

main()主函数的实现代码

```
/* EX32-ap-4.c */
#include "lw_oopc_kc.h"
#include "ex32-il.h"
#include "ex32-ctrl.h"
#include "ex32-factory.h"

extern void* FactoryNew();
char xdata MemPool[1024];
void main (void){
   Factory *fa;     CTRL *ctrl;
   init_mempool(MemPool,sizeof(MemPool));
   fa = FactoryNew();
   fa->init(fa);
   ctrl = fa->get_CTRL(fa);
 ctrl->send(ctrl, 0xaa);
   while(1) {
    ;
    }
}
```

其中，指令：

```
     fa = FactoryNew();          // 产生 Factory 对象。
     fa->init(fa);               // 产生 CTRL、LED_P0 和 LED_P1 对象。
```

```
ctrl = fa->get_CTRL(fa);   // 取得 CTRL 对象。
ctrl->send(ctrl, 0xaa);    // 调用 CTRL 的 send()函数。
```

32.5 IoC 与 COR 模式的携手合作

COR 模式就是 Chain Of Responsibility 模式，顾名思义，“Chain”就是像火车一样，一节接一节而形成一整串，各节有各自特殊的“Responsibility”（责任）。就像手电筒一般，由一节接一节的电池相联结起来，贡献各自的电力。如图 32-2 所示的手电筒就含有两节串联的电池。

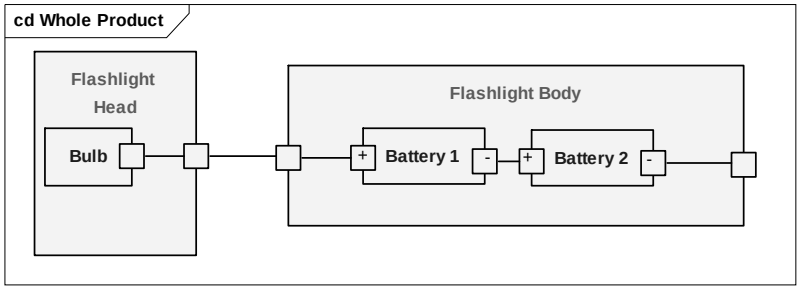


图 32-2

专家们建议采用 COR 模式。从 Gamma 的《Design Patterns》一书里可得知 COR 模式的构造如图 32-3 所示。

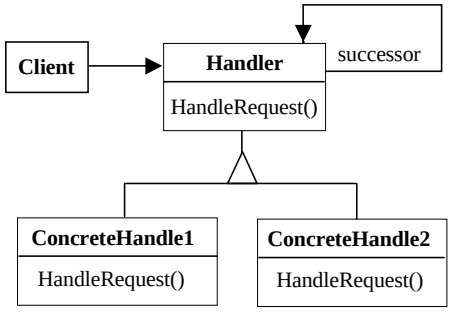


图 32-3 Chain Of Responsibility 模式之结构

将它对应到手电筒的电池结构如图 32-4 所示。

这个程序在计算机里建立出如图 32-5 的对象链（Object Chain），而且各负责不同的任务，所以称为 Chain Of Responsibility。

在执行期间，手电筒对象将各个电池对象组成为一条链（Chain）。然后传递信息（如 getPower）给第 1 个电池对象，第 1 个电池会执行其该尽的责任，执行完毕时，会视信息的

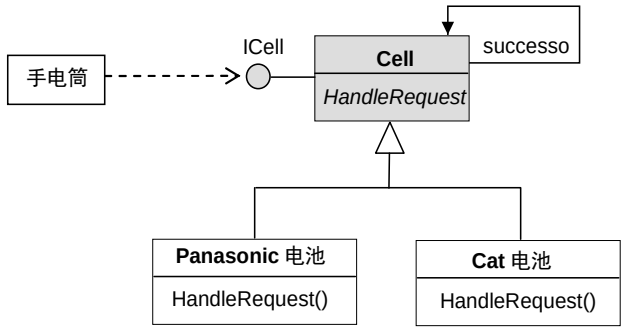


图 32-4

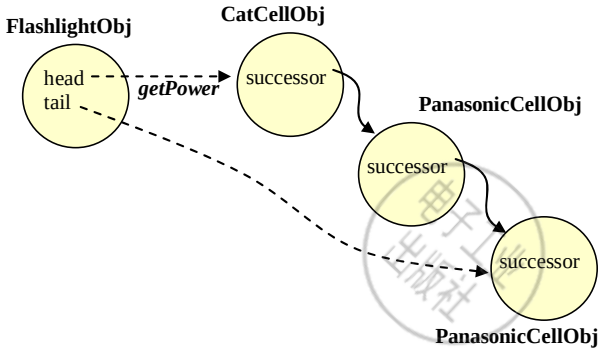


图 32-5

含义而决定是否继续将信息传给后续的电池对象。如此就达到各尽其责的目标了。现在，也将 COR 模式对应到 LED_P0 和 LED_P1 等对象，如图 32-6、图 32-7 所示。

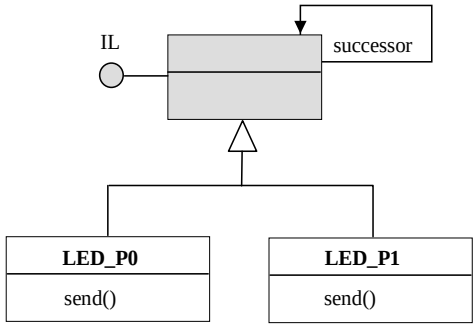


图 32-6

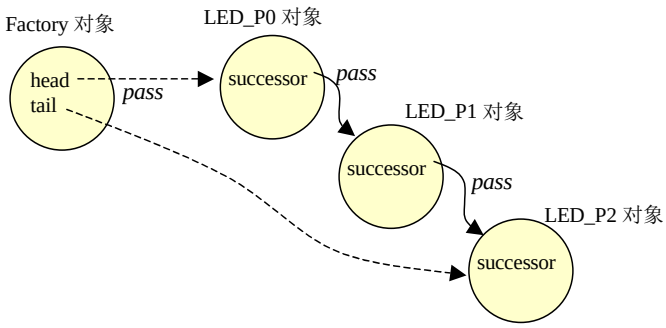


图 32-7

Factory 对象调用 LED_P0 对象的 pass()函数，紧接着 LED_P0 可以调用后面 LED_P1 对象的 pass()函数，而 LED_P1 可以调用后面的 LED_P2 对象的 pass()函数，一直往后调用。请看其程序代码：

IL 接口的定义文件

```
/*  EX32-il.h */
#ifndef _IL_H
#define _IL_H
#include "lw_oopc_kc.h"

INTERFACE(IL){
    int (*pass)(char, char, void*);
};
#endif
```

LED_P0 和 LED_P1 类的定义文件

```
/* EX32-led.h   */
#include "lw_oopc_kc.h"
#include "ex32-il.h"

CLASS(LED_P0){
  IMPLEMENTS(IL);
  void (*init)(LED_P0*);
  IL* next;
};
CLASS(LED_P1){
  IMPLEMENTS(IL);
  void (*init)(LED_P1*);
  IL* next;
};
CLASS(LED_P2){
  IMPLEMENTS(IL);
  void (*init)(LED_P2*);
  IL* next;
};
```

这里定义了 3 个 LED 类：LED_P0、LED_P1 和 LED_P2。

LED_P0 和 LED_P1 类的实现代码

```
/* EX32-led.c */
#include <REG51F.H>
#include "lw_oopc_kc.h"
#include "ex32-led.h"

// -------------------------------------------------
 static void init_0(LED_P0* t) {
      t->next = NULL;
}
static int pass_0(char ty, char hx, void *t) {
LED_P0 *cthis = (LED_P0*)t;
   IL* ps = cthis->next;
   if(ty=='0') {
      P0 = hx;  return 1;
    }
if(ty=='A') P0 = hx;
if(ps == NULL) return 0;
else  return ps->pass(ty, hx, ps);
}
CTOR(LED_P0)
  FUNCTION_SETTING(init, init_0)
  FUNCTION_SETTING(IL.pass, pass_0)
END_CTOR
//-------------------------------------------------
static void init_1(LED_P1* t) {
      t->next = NULL;
}
static int pass_1(char ty, char hx, void *t) {
   LED_P1 *cthis = (LED_P1*)t;
IL* ps = cthis->next;
   if(ty=='1') {
      P1 = hx;   return 1;
   }
if(ty=='A') P1 = hx;
if(ps == NULL) return 0;
else  return ps->pass(ty, hx, ps);
}
CTOR(LED_P1)
  FUNCTION_SETTING(init, init_1)
  FUNCTION_SETTING(IL.pass, pass_1)
END_CTOR
/* ------------------------------------------------- */
static void init_2(LED_P2* t) {
      t->next = NULL;
}

static int pass_2(char ty, char hx, void *t) {
   LED_P2 *cthis = (LED_P2*)t;
IL* ps = cthis->next;
   if(ty=='2') {
      P2 = hx;   return 1;
    }
if(ty=='A') P2 = hx;
if(ps == NULL) return 0;
else  return ps->pass(ty, hx, ps);
}
CTOR(LED_P2)
```

```
  FUNCTION_SETTING(init, init_2)
  FUNCTION_SETTING(IL.pass, pass_2)
END_CTOR
```

其中，pass_0()函数有 2 个重要参数 ty 和 hx。ty 为各对象的识别。例如：

```
        if( ty == '0' ) {
         P0 = hx;
         return 1;
      }
```

如果 ty 值为'0'，表示这是 LED_P0 的责任，于是将另一个参数 hx 值送到 P0。如果 ty 值为'A'，表示这是所有对象都有的责任，所以将 hx 值送到 P0 之后必须继续调用后面对象的 pass()函数，如下面的程序：

```
     if( ty == 'A' ) P0 = hx;
     if(ps == NULL) return 0;
     else  return ps->pass(ty, hx, ps);
```

现在来定义 Factory 类：

Factory 类的定义文件

```
/* EX32-factory.h */
#include "ex32-il.h"

CLASS(Factory) {
   void (*init)(Factory*);
   IL* (*get_LED_LIST)(Factory*, int);
   IL *head;
  };
```

Factory 类的实现代码

```
/* EX32-factory.c */
#include "lw_oopc_kc.h"
#include "ex32-led.h"
#include "ex32-factory.h"

extern void* LED_P0New();
extern void* LED_P1New();
extern void* LED_P2New();

static void init(Factory *t) {
   LED_P0* ps0;   LED_P1* ps1;   LED_P2* ps2;
   ps0 = LED_P0New();   ps0->init(ps0);
   t->head = (IL*)ps0;
   ps1 = LED_P1New();   ps1->init(ps1);
   ps0->next = (IL*)ps1;
   ps2 = LED_P2New();   ps2->init(ps2);
   ps1->next = (IL*)ps2;
  }
static IL* get_LED_LIST(Factory *t) {
   return t->head;
  }
CTOR(Factory)
  FUNCTION_SETTING(init, init)
```

```
  FUNCTION_SETTING(get_LED_LIST, get_LED_LIST)
END_CTOR
```

main()主函数的定义及实现代码

```
/* EX32-ap-5.c */
#include <REG51F.H>
#include "lw_oopc_kc.h"
#include "ex32-il.h"
#include "ex32-factory.h"

void g_timer_delay() {
   TMOD &= 0xF0;    TMOD |= 0x01;    ET0 = 0;
   TH0 = 0x3C;       TL0 = 0xB0;
   TF0 = 0;            TR0 = 1;
   while(TF0 == 0);
   TR0 = 0;
}
void g_delay(unsigned int sec) {
   int i, j;
   for(i=0; i<sec; i++)
      for(j=0; j<20; j++)
         g_timer_delay();
}

extern void* FactoryNew();
char xdata MemPool[1024];
void main (void){
   IL *leds; Factory *fa;
   init_mempool(MemPool,sizeof(MemPool));
   fa = FactoryNew();
   fa->init(fa);
   leds = fa->get_LED_LIST(fa);

   leds->pass('0', 0x77, leds);     g_delay(2);
   leds->pass('1', 0x33, leds);     g_delay(2);
   leds->pass('3', 0xaa, leds);   g_delay(2);
   leds->pass('2', 0xaa, leds);   g_delay(2);
   P0 = 0x00;   P1 = 0x00;   P2 = 0x00;
   g_delay(2);
   leds->pass('A', 0xf0, leds);
   while(1) {
      ;
   }
}
```

程序代码说明如下：

指令：leds = fa->get_LED_LIST(fa); // 取得第一个 LED 对象的指针。

指令：leds->pass('0', 0x77, leds); // 调用第一个 LED 对象的 pass()函数，并把两个参数传递出去。

参数'0' 表示指定由 LED_P0 对象执行（即送出 0x77 值）。

指令：leds->pass('A', 0xf0, leds); // 表示全部的 LED 对象都要执行，都送出 0x77。

第 33 章　应用范例四

——Keil C51 结合 State 设计模式

33.1 前言

嵌入式系统大多属于控制系统，需要极高的可靠性。精致的细节设计是达成高度可靠性的不二法门。在设计控制系统时，最常见又有效的途径就是运用状态机（State Machine）观念。

在前面各章的范例程序里，曾经分析过像 CTRL 等控制对象的状态，然后编写 Keil C 的类，并设计其处于不同状态下的行为，来达到控制整个系统行为的目的。如果你仔细观察，会发现我们的传统做法是：

Step-1 从系统中分析出许多个对象，然后编写这些对象的类。

Step-2 针对各对象（尤其是 CTRL 控制对象）分析它的状态变化，然后编写函数来实现各状态下的行为。

针对中小型系统而言，这种途径已经足够了。然而，对于具有复杂细节的对象而言，通常会增加一个步骤：

Step-3 把复杂对象的“状态”视为对象，然后编写这种状态类来实现各状态下的复杂行为。

在 Gamma 着的《Design Patterns》一书里，将这个步骤定为一种专家心中的模式，称为 State 模式。在本章里，将详述如何活用 State 模式来进行精致的细节设计。

33.2 活用 State 模式

在本章开头说明过，针对一个复杂控制系统的设计，专家常用的方法是：

Step-1 从系统中分析出许多个对象，然后编写这些对象的类。

Step-2 针对各对象（尤其是 CTRL 控制对象）分析它的状态变化，然后编写函数来实现各状态下的行为。

Step-3 把复杂对象的“状态”视为对象，然后编写这种状态类来实现各状态下的复杂行为。

而 Gamma 在 *Design Patterns* 一书里，建议使用一种专家心中的模式，称为 State 模式，能有效做好精致的细节设计。

33.2.1 温故而知新

首先温习一下传统的做法：即上述的 Step-1 和 Step-2，在此基础上，才能进一步体会 State 模式的深层含义。例如，一个电梯的控制对象的状态变化情形如图 33-1 所示。

依据传统分析方法，通常会设计一个类，如图 33-2 所示。

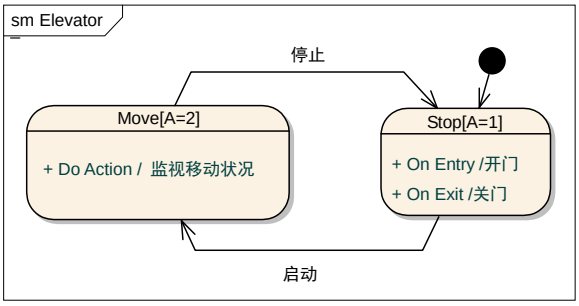


图 33-1

图 33-2

由于该系统并不复杂，把所有事件、状态及行为逻辑都写在一个类里，是比较合理的。如下述的 Keil C 程序代码：

```
/* EX33-ap-1.c */
#include <REG52.H>
#include "lw_oopc_kc.h"

void g_delay(unsigned int ms) {
   int i, j;
   for(i=0; i<ms; i++)
      for(j=0; j<120; j++);
}

CLASS(ElevatorController)
{  void (*init)(ElevatorController*);
   void (*move)(ElevatorController*);
   void (*stop)(ElevatorController*);
   void (*goto_state_Move)(ElevatorController*);
   void (*goto_state_Stop)(ElevatorController*);
   int state_var;
};

static void init(ElevatorController *t) {
    t->goto_state_Stop(t);
 }
static void goto_state_Stop(ElevatorController *t) {
   t->state_var = 1;   P0 = 0xF0;   g_delay(6000);
}
static void goto_state_Move(ElevatorController *t){
```

```
    t->state_var = 2;   P0 = 0x0F;  g_delay(6000);
  }
static void move(ElevatorController *t) {
  if(t->state_var == 1)  t->goto_state_Move(t);
  }
static void stop(ElevatorController *t){
  if( t->state_var == 2)  t->goto_state_Stop(t);
 }
CTOR(ElevatorController)
    FUNCTION_SETTING(init, init)
    FUNCTION_SETTING(move, move)
    FUNCTION_SETTING(stop, stop)
    FUNCTION_SETTING(goto_state_Stop, goto_state_Stop)
    FUNCTION_SETTING(goto_state_Move, goto_state_Move)
END_CTOR
/* ------------------------------------------------------ */
char xdata MemPool[1024];
void main (void) {
    ElevatorController *ele;
    init_mempool(MemPool,sizeof(MemPool));
ele = ElevatorControllerNew();
ele->init(ele);
while(1) {
ele->move(ele);
ele->stop(ele);
     }
}
```

33.2.2 State 模式

上述代码把两个状态的行为部分写在同一个类里。如果把这两个状态的行为部分分开写在不同的类里，进行分而治之，便成为极细腻精致的设计了。这就是 Gamma 的 *Design Patterns* 一书介绍 State 模式的用意。现在请先看看 Gamma 所定义的 State 模式，其结构如图 33-3 所示。

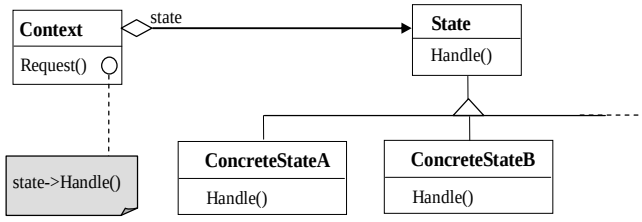


图 33-3

在这里，我们依据状态切分对象，让您体会一种切分对象的新手艺。当您发现有更多状态时，只需要增加新的状态类就行了，除了做更精细的设计之外，系统的弹性也提升了。现在依据 State 模式，重新绘制如图 33-4 所示的电梯系统类图。

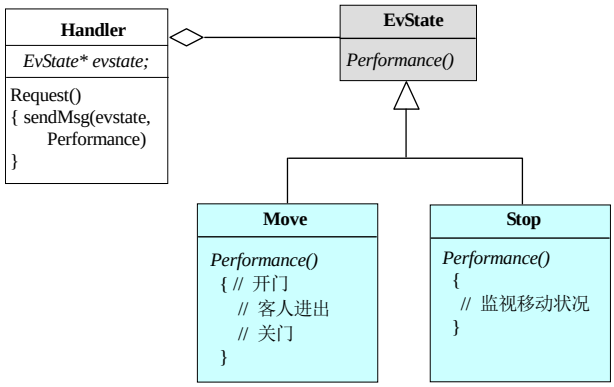


图 33-4

Move 和 Stop 就成为两个状态类，EvState 成为接口，然后再增添一个 Handler 类来整合状态类及其接口，于是原来的 Elevator Controller 就分解成为 Handler、Move 和 Stop 3 个小对象，共同合作来执行原来 Elevator Controller 的复杂功能。这样，人们能更精细地掌握复杂的细节行为，以提升系统的稳定性和可靠性。与图 33-4 所对应的 Keil C 程序代码如下：

IEvState 接口的定义文件

```
/* EX33-evs.h */
#ifndef EVS_H
#define EVS_H
#include "lw_oopc_kc.h"

INTERFACE(IEvState)
{
   void (*perform)();
};
#endif
```

Stop 状态类的定义及实现代码

```
/* EX33-stop.c  */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex33-evs.h"

CLASS(Stop)
{
  IMPLEMENTS(IEvState);
};

static void perform() {
  P0 = 0x0F;
}
CTOR(Stop)
   FUNCTION_SETTING(IEvState.perform, perform)
END_CTOR
```

Move 状态类的定义及实现代码

```
/* EX33-move.c  */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex33-evs.h"

CLASS(Move)
{
   IMPLEMENTS(IEvState);
};

static void perform() {
   P0 = 0xF0;
}
CTOR(Move)
    FUNCTION_SETTING(IEvState.perform, perform)
END_CTOR
```

现在已经定义了 Move 和 Stop 两个状态类，接着定义 Handler 类来整合状态类。

Handler 状态类的定义文件

```
/* EX33-handler.h */
#include "EX33-evs.h"

CLASS(Handler)
{
   void (*init)(Handler*);
   void (*change_state)(Handler*);
   char state;
   IEvState *list[2], *es_obj;
};
```

Handler 状态类的实现代码

```
/* EX33-handler.c  */
#include <REG52.H>
#include <string.h>
#include "lw_oopc_kc.h"
#include "ex33-evs.h"
#include "ex33-handler.h"

extern void* StopNew();
extern void* MoveNew();
extern void* g_delay(unsigned int);

static void init(Handler*t) {
    t->list[0] = StopNew();   t->list[1] = MoveNew();
    t->state = 'I';
 }
static void perform_action(Handler *t) {
    t->es_obj->perform();
}
static void goto_S(Handler *t){
   t->state = 'S';
   t->es_obj = t->list[0];
}
static void goto_M(Handler *t){
```

```
   t->state = 'M';
   t->es_obj = t->list[1];
}
static void change_state(Handler *t) {
    switch(t->state) {
       case 'I':  goto_S(t);  break;
       case 'S':  goto_M(t);  break;
       case 'M':  goto_S(t);  break;
      }
      perform_action(t);
      g_delay(6000);
}

CTOR(Handler)
    FUNCTION_SETTING(init, init)
    FUNCTION_SETTING(change_state, change_state)
END_CTOR
```

就 Handler 而言，它有两个主要状态：'S'和 'M'，再加上初期状态'I'。在 init()函数里，其指令为：

```
static void init(Handler*t) {
    t->list[0] = StopNew();
    t->list[1] = MoveNew();
    t->state = 'I';
 }
```

该指令产生状态类的对象，并存入 list[]数组里。一旦进入'M'状态时，执行 goto_S()函数：

```
static void goto_S(Handler *t){
   ……..
   t->es_obj = t->list[0];
}
```

这令 es_obj 指向 Stop 状态对象。一旦进入'S'状态时，执行 goto_M()函数：

```
static void goto_M(Handler *t){
   ……..
   t->es_obj = t->list[1];
}
```

这又令 es_obj 指向 Move 状态对象。最后编写主函数：

main()函数实现代码

```
/* EX33-ap-2.c  */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex33-evs.h"
#include "ex33-handler.h"

void g_delay(unsigned int ms) {
   int i, j;
   for(i=0; i<ms; i++)
      for(j=0; j<120; j++);
}

extern void* HandlerNew();
```

```
char xdata MemPool[1024];
void main (void)
{
   Handler *hnd;
   init_mempool(MemPool,sizeof(MemPool));
   hnd = HandlerNew();
   hnd->init(hnd);
   while(1) {
      hnd->change_state(hnd);
      g_delay(6000);
    }
}
```

这里，while(1)循环不断调用 change_state()函数，每一次调用，Handler 对象就改变一次状态。于是，就循环于 'S' 与 'M' 两个状态之间。

33.3 以飞机状态控制为例

飞机在不同状态下会有不同的行为，State 模式很适合设计与飞机有关的软件对象、状态及其行为。在本节里，就以飞机对象为例，将飞机对象的 4 个状态视为对象，配上 State 模式，精致地描述飞机的行为细节。

33.3.1 需求分析（Analysis）

飞机具有 4 个典型的飞行状态，一开始处于 Preparing 状态，然后进入 Taking Off 状态，代表飞机正起飞中。不久，进入 Flying 状态，代表飞机正平稳飞行中。欲下降时，进入 Landing 状态，逐渐降落于机场跑道。其状态变化，如图 33-5 所示。

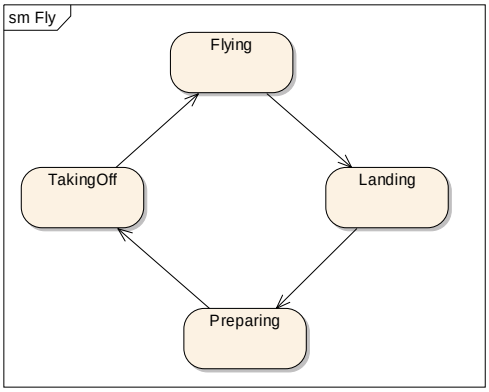


图 33-5

确定了飞机的 4 个状态之后，就能进行：

- 设计 4 个状态类，以及一个接口。
- 并设计一个 CTRL（即 Controller）类。设计图如图 33-6 所示。

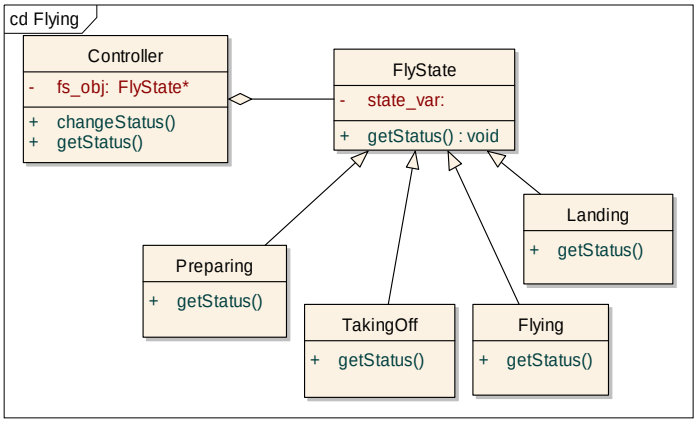


图 33-6

33.3.2 以 Keil C 实现范例（1）

针对图 33-5 和图 33-6，可编写 Keil C 程序代码如下：

IFlyState 接口的定义文件

```
/* EX33-ifs.h */
#ifndef IFS_H
#define IFS_H
#include "lw_oopc_kc.h"

CLASS(STATE)
{
   unsigned char state_code;
   int height;
   int position_x;
   int position_y;
};

INTERFACE(IFlyState)
{
    void (*init)(void*);
    void (*handle)(void*);
};
#endif
```

这个 IFlyState 就是 4 个状态类的接口。

Preparing 状态类的定义及实现代码

```
/* EX33-pre.c  */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex33-ifs.h"

CLASS(Preparing)
{  IMPLEMENTS(IFlyState);
   STATE st;
```

```
};

static void init(void *t){
   Preparing *cthis = t;   cthis->st.state_code = 0x11;
   }
static void handle(IFlyState *t) {
    Preparing *cthis = t;    P0 = cthis->st.state_code;
}
CTOR(Preparing)
   FUNCTION_SETTING(IFlyState.init, init)
   FUNCTION_SETTING(IFlyState.handle, handle)
END_CTOR
```

进入 Preparing 状态，代表飞机正准备起飞。

TakingOff 状态类的定义及实现代码

```
/* EX33-take.c  */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex33-ifs.h"

CLASS(TakingOff)
{
   IMPLEMENTS(IFlyState);
   STATE st;
};

static void init(void *t){
   TakingOff *cthis = t;   cthis->st.state_code = 0x33;
  }
static void handle(IFlyState *t) {
    TakingOff *cthis = t;     P0 = cthis->st.state_code;
}
CTOR(TakingOff)
   FUNCTION_SETTING(IFlyState.init, init)
   FUNCTION_SETTING(IFlyState.handle, handle)
END_CTOR
```

进入 TakingOff 状态，代表飞机正起飞中。

Flying 状态类的定义及实现代码

```
/* EX33-fly.c  */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex33-ifs.h"

CLASS(Flying)
{
   IMPLEMENTS(IFlyState);
   STATE st;
};

static void init(void *t){
   Flying *cthis = t;
   cthis->st.state_code = 0x77;
   cthis->st.position_x = 2;
   cthis->st.position_y = 2;
```

```
  }
static void handle(IFlyState *t) {
    Flying *cthis = t;
    P0 = cthis->st.state_code;
}
CTOR(Flying)
   FUNCTION_SETTING(IFlyState.init, init)
   FUNCTION_SETTING(IFlyState.handle, handle)
END_CTOR
```

经过几分钟后，进入 Flying 状态，代表飞机正平稳飞行中。

Landing 状态类的定义及实现代码

```
/* EX33-land.c  */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex33-ifs.h"

CLASS(Landing)
{
   IMPLEMENTS(IFlyState);
   STATE st;
};

static void init(void *t){
   Landing *cthis = t;
   cthis->st.state_code = 0xaa;
  }
static void handle(IFlyState *t) {
    Landing *cthis = t;
    P0 = cthis->st.state_code;
}
CTOR(Landing)
   FUNCTION_SETTING(IFlyState.init, init)
   FUNCTION_SETTING(IFlyState.handle, handle)
END_CTOR
```

进入 Landing 状态，代表飞机开始下降。经过几分钟后，回到 Preparing 状态，代表飞机成功着陆了。依序循环呈现出飞机在 4 个状态之间的转移。接下来，编写 Handler 类（即图 33-6 的 Controller 类）来控制这些状态的变化。

Handler 类的定义文件

```
/* EX33-handler.h */
#include "EX33-ifs.h"

CLASS(Handler)
{
   void (*init)(Handler*);
   void (*run)(Handler*);
   IFlyState* array[4];
   char state;
};
```

Handler 类的实现代码

```
/* EX33-handler.c  */
```

```
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex33-ifs.h"
#include "ex33-handler.h"

extern void* LandingNew();
extern void* FlyingNew();
extern void* PreparingNew();
extern void* TakingOffNew();

void g_delay(unsigned int ms) {
   int i, j;
   for(i=0; i<ms; i++)
     for(j=0; j<120; j++);
}

static void init(Handler *t){
    IFlyState *fs;
     fs = PreparingNew();  fs->init(fs);    t->array[0] = fs;
    fs = TakingOffNew();   fs->init(fs);     t->array[1] = fs;
    fs = FlyingNew();    fs->init(fs);     t->array[2] = fs;
    fs = LandingNew();    fs->init(fs);     t->array[3] = fs;
    t->state = 'I';
}

static void goto_P(Handler *t){
   t->state = 'P';   t->array[0]->handle(t->array[0]);
}
static void goto_T(Handler *t){
   t->state = 'T';   t->array[1]->handle(t->array[1]);
}
static void goto_F(Handler *t){
   t->state = 'F';   t->array[2]->handle(t->array[2]);
}
static void goto_L(Handler *t){
   t->state = 'L';   t->array[3]->handle(t->array[3]);
}

static void run(Handler *t) {
    switch(t->state) {
       case 'I':         goto_P(t);         break;
       case 'P':         goto_T(t);         break;
       case 'T':     goto_F(t);         break;
       case 'F':     goto_L(t);         break;
       case 'L':     goto_P(t);         break;
      }
      g_delay(20000);
  }

CTOR(Handler)
    FUNCTION_SETTING(init, init)
    FUNCTION_SETTING(run, run)
END_CTOR
```

加上初期状态 'I'，总共有 5 个状态。Init()函数产生 4 个状态类的对象，并将其指针存于 array[]数组里。随着状态的不同，就通过 IFlyState 接口而调用不同对象的 handle()函数，而产生不同的行为。最后，编写主函数：

main()主函数的定义及实现档

```
/* EX33-ap-3.c  */
#include <REG52.H>
#include "lw_oopc_kc.h"
#include "ex33-ifs.h"
#include "ex33-handler.h"

extern void* HandlerNew();
extern void* FlyingNew();
char xdata MemPool[1024];

void main (void)
{
   Handler *hnd;
   init_mempool(MemPool,sizeof(MemPool));
   //--------------------------------------
   hnd = HandlerNew();
   hnd->init(hnd);
   while(1) {
      hnd->run(hnd);
    }
}
```

这里，while(1)循环不断调用 Handler 的 run()函数，每一次调用，Handler 对象就改变一次状态。于是，就循环于 4 个状态之间，周而复始。

33.3.3 以 Keil C 实现范例（2）

在上一节里，周而复始的控制者（Controller）及触发者（Trigger）是 main()函数及其内部的大循环：

```
void main (void) {
   ……..
   while(1) {
      hnd->run(hnd);
    }
}
```

而且 g_delay()函数不太精确。在本节里，将把上一节的 Keil C 程序实现以下功能：

- 采用 Timer 来进行精确计时，并触发系统状态的变化。
- 由于受限于μVision3 IDE 环境测试版的 2K 限制，本范例将删减两个状态。
- 添加一个 CTRL 类，它担任产生状态对象的任务，其角色相当于 IoC 模式里的 Factory 类。
- 采用两个 Timer，以 Timer0 触发 Handler 对象的行为，并且以 Timer1 触发 CTRL 对象的行为。因而 CTRL 和 Handler 两个对象的行为是同步且并行（Concurrent）的。如图 33-7 所示。

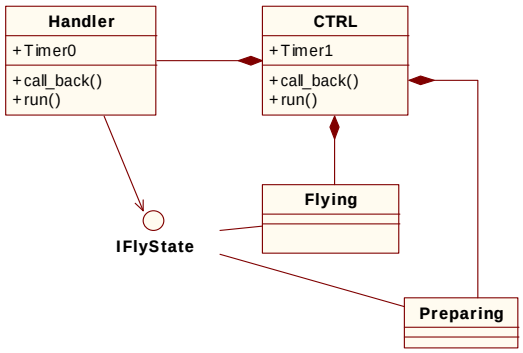


图 33-7

首先定义 IFlyState 接口如下：

IFState 接口的定义文件

```
/* EX33-ifs.h */
#ifndef IFS_H
#define IFS_H
#define LW_OOPC_STATIC
#include "lw_oopc_kc.h"

CLASS(STATE)
{
   unsigned char state_code;
   int height;
   int position_x;
   int position_y;
};

INTERFACE(IFlyState)
{
    void (*init)(void*);
    void (*handle)(void*);
};
#endif
```

Preparing 状态类的定义文件

```
/* EX33-pre.h */
CLASS(Preparing)
{
   IMPLEMENTS(IFlyState);
   void (*init)(Preparing*);
   STATE st;
};
```

Preparing 状态类的实现代码

```
/* EX33-pre.c  */
#include <REG52.H>
```

```
#include "ex33-ifs.h"
#include "ex33-pre.h"

static void init(void *t){
   Preparing *cthis = t;
   cthis->st.state_code = 0x11;
   }
static void handle(IFlyState *t) {
    Preparing *cthis = t;
    P0 = cthis->st.state_code;
}
CTOR(Preparing)
   FUNCTION_SETTING(init, init)
   FUNCTION_SETTING(IFlyState.handle, handle)
END_CTOR
```

Flying 状态类的定义文件

```
/* EX33-fly.h */
CLASS(Flying)
{
   IMPLEMENTS(IFlyState);
   void (*init)(Flying*);
   STATE st;
};
```

Flying 状态类的实现代码


```
/* EX33-fly.c  */
#include <REG52.H>
#include "ex33-ifs.h"
#include "ex33-fly.h"

static void init(void *t){
   Flying *cthis = t;
   cthis->st.state_code = 0x77;
   cthis->st.position_x = 2;
   cthis->st.position_y = 2;
  }
static void handle(IFlyState *t) {
    Flying *cthis = t;
    P0 = cthis->st.state_code;
}
CTOR(Flying)
   FUNCTION_SETTING(init, init)
   FUNCTION_SETTING(IFlyState.handle, handle)
END_CTOR
```

Handler 类的定义文件

```
/* EX33-handler.h */
#include "EX33-ifs.h"

CLASS(Handler)
{
   void (*init)(Handler*);
   void (*start)();
   IFlyState  *fs;
```

```
};
```

Handler 类的实现代码

```
/* EX33-handler.c  */
#include <REG52.H>
#include "ex33-ifs.h"
#include "ex33-handler.h"

static Handler *pL;
static long count;
static int max;

void init_timer0() {
  TMOD &= 0xF0;    TMOD |= 0x01;
  TH0 = 0x00;       TL0 = 0x00;
  ET0 = 1;          TR0 = 1;  EA = 1;
  }

void reload_timer0() {
  TR0 = 0;     TH0 = 0x00;
  TL0 = 0x00;  TR0 = 1;
  }

static void init(Handler *t){
    t->fs = NULL;      pL = t;     max = 400;
}
static void run(Handler *t)  { t->fs->handle(t->fs);  }
static void call_back2() interrupt 1 {
       count++;
       if(count > max)  run(pL);
       reload_timer0();
}

static void start() {
    long i;    P0 = 0x00;   count = 0;
   for(i=0; i<100000; i++) ;  /* 让 Timer0 落后 Timer1  */
   init_timer0();
  }

CTOR(Handler)
   FUNCTION_SETTING(init, init)
   FUNCTION_SETTING(start, start)
END_CTOR
```

Timer0 定时调用 Handler 的 call_back()函数，然后调用到 run()函数，再调用到 Preparing 等状态对象的 handle()函数。所以这里，run()并不是由 CTRL 来调用的。

CTRL 类的定义文件

```
/* EX33-ctrl.h */
#include "ex33-pre.h"
#include "ex33-fly.h"
#include "ex33-handler.h"

CLASS(CTRL)
{
  void (*init)(CTRL*);
```

```
   void (*start)(CTRL*);
   Handler hObj;
   char state;
   Preparing pObj;
   Flying fObj;
};
```

CTRL 类的实现代码

```
/* EX33-ctrl.c  */
#include <REG52.H>
#include "ex33-ifs.h"
#include "ex33-ctrl.h"

extern void* HandlerSetting(void*);
extern void* PreparingSetting(void*);
extern void* FlyingSetting(void*);
static CTRL *pL;   static long count;    static int max;

void init_timer1() {
  TMOD &= 0x0F;    TMOD |= 0x10;
  TH1 = 0x00;       TL1 = 0x00;
  ET1 = 1;          TR1 = 1;       EA = 1;
  }

void reload_timer1() {
  TR1 = 0;     TH1 = 0x00;
  TL1 = 0x00;  TR1 = 1;
  }
static void init(CTRL *t){
    PreparingSetting(&t->pObj);     t->pObj.init(&t->pObj);
    FlyingSetting(&t->fObj);        t->fObj.init(&t->fObj);
    HandlerSetting(&t->hObj);       t->hObj.init(&t->hObj);
    t->state = 'I';     pL = t;    max = 400;
}
static void goto_P(CTRL *t){
   t->state = 'P';
   t->hObj.fs = (IFlyState*)(&t->pObj);
}
static void goto_F(CTRL *t){
   t->state = 'F';
   t->hObj.fs =(IFlyState*)(&t->fObj);
}
static void run(CTRL *t) {
    count = 0;
    switch(t->state) {
       case 'I':   goto_P(t);     break;
       case 'P':   goto_F(t);     break;
       case 'F':   goto_P(t);  break;
      }
  }
static void call_back() interrupt 3 {
    count++;
    if(count > max)  run(pL);
    reload_timer1();
}
static void start(CTRL *t) {
    count = 0;
    init_timer1();
```

```
    t->hObj.start();
}
CTOR(CTRL)
    FUNCTION_SETTING(init, init)
    FUNCTION_SETTING(start, start)
END_CTOR
```

Timer1 定时调用 CTRL 的 call_back()函数，然后调用到 run()函数来变换 CTRL 的状态，并且切换不同的状态对象。

main()主函数的定义及实现代码

```
/* EX33-ap-4.c  */
#include <REG52.H>
#include "ex33-ifs.h"
#include "ex33-ctrl.h"

extern void* CTRLSetting(void*);
char xdata MemPool[1024];

void main (void){
   CTRL controller;
   init_mempool(MemPool,sizeof(MemPool));
   //---------------------------------
   CTRLSetting(&controller);
   controller.init(&controller);
   controller.start(&controller);
   while(1) {
      ;
    }
}
```

这里，main()函数只是启动 CTRL 对象而已，不再扮演周而复始的控制者（Controller）及触发者（Trigger）角色了。本范例展示了并行及同步的复杂系统的 Keil C 程序设计方法。

33.4 结语：类、接口与模式

本书的主要贡献在于制作了 lw_oopc.h 和 lw_oopc_kc.h 两个头文件，替 ANSI-C 和 Keil C51 添加类、接口的定义机制，让 C 程序员能将优雅的面向对象技术与高效的 C 语言相结合。

基于类和接口机制，便能进一步采用 UML 分析及设计技术，搭配专家经验和常用模式的运用，让嵌入式程序员能精确掌握系统需求和设计优美的系统架构，以求大幅提高嵌入式系统的可靠性和质量。在本书里，已经详细介绍上述的类、接口、UML 和基本模式。有了上述的良好基础，未来你的经验增加了，便能更上一层楼，创造出属于你自己的模式，并让别人来分享。例如，笔者就创造了 9 个接口设计模式。

接口设计是软硬件协同架构规划的核心，好的接口设计，可以在复杂的模块关系中找到适当的依赖性。恰到好处的依赖性，是集成系统能否如期完成的关键因素。例如哈佛大学两

位专家 Lee Fleming 和 Olav Sorenson[1]就特别强调模块之间的依赖性（dependency）是产品成功的关键要素。以喷墨打印机为例，自从 1867 年由 Loard Kelvin 提出以来，尽管投入了大量金钱，但其仍然在 100 多年后才开始商业化。其症结在于：模块间的依赖性太高，包括墨水的化学性质，模块安排方式、电阻体的组合等，彼此之间都密切相关。

另一方面，他举 SONY 随身听为例，其工程师采取高度标准化及易于更换的模块，从而迅速设计出市场上极受欢迎的随身听。因为适当的依赖性，才能迅速产出有用的新产品。

于此，本节列出 9 个接口模式，来与您分享。如果您想要更详细理解这些模式的话，可以从 *www.umlchina.com* 网站的嘉宾讲座下载笔者于北京中关村的演讲稿和录音文件。演讲日期是 2007 年 4 月 12 日，题目是“如何提升软件设计力”。现列出这些接口设计模式如下：

1. Know Unknown pattern

Intent: 以接口来承接未知或变动的需求。

Force 1: 系统分析员（system analyst）经常面对未知的需求（unknown requirements）。

Force 2: 程序员（programmer）不能在程序里写上“unknown”字眼。

Solution: 设计师会设计接口（interface）去面对未知的需求。

Consequences: 知之为知之，不知为不知，是知也。SA 可以追求真知，承接真知是设计力的极佳表现。

2. Program to Interface pattern

Intent: 让接口的双方能独立成长又互相辉映。

Force 1: 两个相关的系统经常需要异地分工及并行开发，开发过程各自改变在所难免。

Force 2: 两个系统需要不断配合对方改变，导致沟通成本提高，并行开发效率下降。

Solution: 双方都“根据接口开发程序”（program to an interface）。

Consequences: 双方依据接口而分工，施工细节互不干涉，高效并行开发，进而提早上市（time-to-market）。

3. Interface Creation pattern

Intent: 接口设计在先，分工在后。

Force 1: 分工生产依赖接口。

Force 2: 接口是两个最终产品的衔接点。

Force 3: 在分工时，最终产品还未诞生。

---

[1] Lee Fleming & Olav Sorenson, “The Dangers of Modularity”, Harvard Business Review, Sept. 2001.

Solution: 对最终产物构思其形（form），从两个形磨合出接口。

Consequences: 以精致的接口（well-designed interface）为分工的依据，分工之后也能自然整合。

4. Integrity pattern

Intent: 透过接口整合，达到优质的系统集成。

Force 1: 大型系统经常包含许多接口。

Force 2: 接口之间难免互相冲突。

Solution: 各模块（part）的接口先汇集，相互磨合、修正到合乎整体（whole）和谐的要求。

Consequences: 通过接口磨合，既能达到系统集成，又能维持 PnP。避免因追求系统集成而失去 PnP 的效益。

5. Organic Order pattern

According to Christopher Alexander, an organic order is achieved when there is a perfect balance between the needs of parts and the needs of the whole.（模块和整体都各有需要，当两者之间取得完美的平衡时，有机秩序就于焉而生了。如图 33-8 所示）。

Intent: 一种形（form），让整体（whole）能有机成长。

Force 1: The needs of parts 经常互相冲突。

Force 2: The needs of whole 极需要和谐（harmony）。

Solution: 规划一个模块代表整体以维持和谐，规划一个模块来衔接其它模块以支持 PnP。

Consequences: 以柔软而逐渐成长的架构（architecture），强力支持模块的蜕变，接近生物的有机成长。

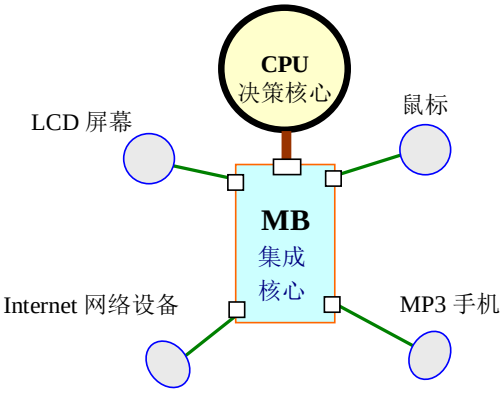


图 33-8

6. Adapter pattern

Intent: 封装外来的专用接口，降低依赖性。

Force 1: 使用外来系统，经常需要使用它的专用接口，因而产生依赖性（dependency）。

Force 2: 依赖外来接口，就无法高效 PnP。

Solution: 善用 adapter，把外来专用接口转变为自己定制的接口。

Consequences: adapter 如同壁虎（即系统）的尾巴，将连同外来系统一起被 PnP 掉，但不影响自己的系统（即壁虎）。

7. Software Motherboard pattern

Intent: 降低 adapter 间的依赖性。

Force 1: 系统经常需要与其它系统沟通。

Force 2: adapter 是壁虎的尾巴，会随 system 而 PnP。

Force 3: 在 PnP 掉 adapter 时，经常牵动多个其它 adapter。

Solution: 设计 adapter 的共同 adapter。

Consequences: 此共同 adapter 就是一个软件主板（software MB），而小 adapter 就成为它的端口（port），如图 33-9 所示。

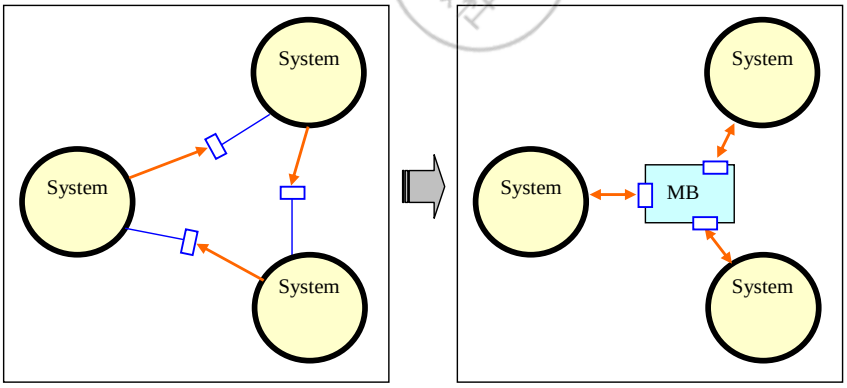


图 33-9

8. CPU pattern

Intent: 统一管理整合性的法则（rule）。

Force 1: 各系统经常需要提供整体性的数据给它的用户（user），其整合性的法则常分散于各系统里。

Force 2: 为了维持整体和谐，需要对整合性法则进行统一管理。

Solution: 设计一个系统，它代表整体（whole）而负责协调其它系统，确保完整性和实时性。

Consequences: 此系统就是一个 software CPU，成为整体的决策中心，它也很容易 PnP 它来面对法则的迅速变化，如图 33-10 所示。

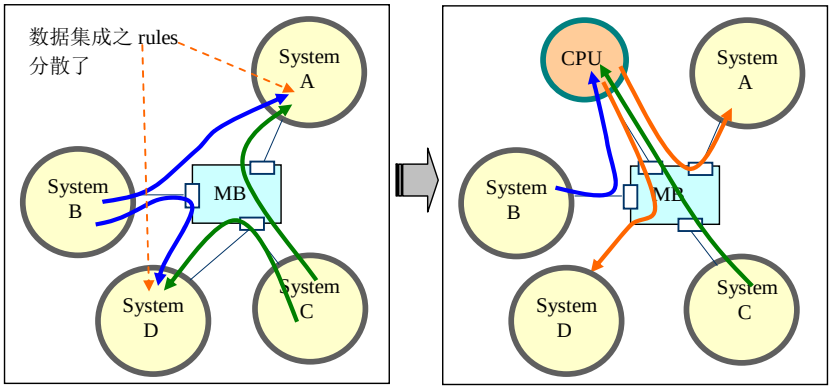


图 33-10

9. SW supporting HW pattern

Intent: 一种形（form），让软硬件协同系统能有机成长。

Force 1: 硬件（HW）不易改变，需要 MB 来支持 HW 的蜕变（PnP）。

Force 2: SW-HW Co-design 的目标是整体有机成长。

Solution: 规划软件 MB，以其逐渐成长来支撑 HW 的蜕变。

Consequences: 软件是圣诞树，而硬件是树上的礼物或糖果。

33.5 祝福您

在本书的各示例里，皆不着痕迹地运用了上述的设计模式，例如在 31.3 节的图 31-8 里就运用了 Software Motherboard 设计模式而设计出 CTRL 类别，只要细心一点，就能找出 Motherboard 在哪里了。希望本书的各示例能让您品尝一些模式的滋味，能逐渐将各种模式应用于您身边的实际案例中。随着经验的累积，您对模式的思维会愈来愈丰富，您的创意和能力也会迅速提升，期望届时您能与笔者共享。谨此祝福您!!

UML+OOPC嵌入式C语言开发精讲

本书由浅入深，从C语言的复习开始，然后讲述C语言如何与OOP相结合，接着从面向对象技术进入UML，最后教读者从实践应用出发，活用UML+OOPC开发流程，做好系统分析和架构设计，实现质量的嵌入式软件系统。

ISBN：978-7-121-07108-9

书价：68.00