# 7.6 函数和结构

为结构编写函数比为数组编写函数要简单得多。虽然结构变量和数组一样，都可以存储多个数据项，但在涉及到函数时，结构变量的行为更接近于基本的单值变量。也就是说，与数组不同，结构将其数据组合成单个实体或数据对象，该实体被视为一个整体。前面讲过，可以将一个结构赋给另外一个结构。同样，也可以按值传递结构，就像普通变量那样。在这种情况下，函数将使用原始结构的副本。另外，函数也可以返回结构。与数组名就是数组第一个元素的地址不同的是，结构名只是结构的名称，要获得结构的地址，必须使用地址运算符&。在C语言和C++中，都使用符号&来表示地址运算符；另外，C++还使用该运算符来表示引用变量，这将在第8章讨论。

使用结构编程时，最直接的方式是像处理基本类型那样来处理结构；也就是说，将结构作为参数传递，并在需要时将结构用作返回值使用。然而，按值传递结构有一个缺点。如果结构非常大，则复制结构将增加内存要求，降低系统运行的速度。出于这些原因（同时由于最初C语言不允许按值传递结构），许多C程序员倾向于传递结构的地址，然后使用指针来访问结构的内容。C++提供了第三种选择——按引用传递（将在第8章介绍）。下面介绍其他两种传递方式，首先介绍传递和返回整个结构。

## 7.6.1 传递和返回结构

当结构比较小时，按值传递结构最合理，下面来看两个使用这种技术的示例。第一个例子处理行程时间。有些地图指出，从Thunder Falls到Bingo城需要3小时50分钟，而从Bingo城到Gotesquo需要1小时25分钟，对于这种时间，可以使用结构来表示——一个成员表示小时值，另一个成员表示分钟值。将两个时间加起来需要一些技巧，因为可能需要将分钟值转换为小时。例如，前面列出的两个时间的总和为4小时75分钟，应将它转换为5小时15分钟。下面开发用于表示时间值的结构，然后再开发一个函数，它接受两个这样的结构为参数，并返回表示参数的和的结构。

定义结构的工作很简单：

struct travel\_time

{

int hours;

int mins;

}

接下来，看一下返回两个这种结构的总和的sum()函数的原型。返回值的类型应为travel\_time，两个参数也应为这种类型。因此，原型应如下所示：

travel\_time sum(travel\_time t1,travel\_time t2);

要将两个时间相加，应首先将分钟成员相加。然后通过整数除法（除数为60）得到小时值，通过求模运算符（%）得到剩余的分钟数。程序清单7.11在sum()函数中使用了这种计算方式，并使用show\_time()函数显示travel\_time结构的内容。

**程序清单7.11 travel.cpp**

// travel.cpp -- 使用结构的函数

#include <iostream>

struct travel\_time

{

int hours;

int mins;

};

const int Mins\_per\_hr = 60;

travel\_time sum(travel\_time t1, travel\_time t2);

void show\_time(travel\_time t);

int main()

{

using namespace std;

travel\_time day1 = {5, 45}; // 5 hrs, 45 min

travel\_time day2 = {4, 55}; // 4 hrs, 55 min

travel\_time trip = sum(day1, day2);

cout << "Two-day total: ";

show\_time(trip);

travel\_time day3= {4, 32};

cout << "Three-day total: ";

show\_time(sum(trip, day3));

cin.get();

return 0;

}

travel\_time sum(travel\_time t1, travel\_time t2)

{

travel\_time total;

total.mins = (t1.mins + t2.mins) % Mins\_per\_hr;

total.hours = t1.hours + t2.hours +

(t1.mins + t2.mins) / Mins\_per\_hr;

return total;

}

void show\_time(travel\_time t)

{

using namespace std;

cout << t.hours << " hours, "

<< t.mins << " minutes\n";

}

其中，travel\_time就像是一个标准的类型名，可以用来声明变量、函数的返回类型和函数的参数类型。由于total和t1变量是travel\_time结构，因此可以对它们使用句点成员运算符。由于sum()函数返回travel\_time结构，因此可以将其用作show\_time()函数的参数。由于在默认情况下，C++函数按值传递参数，因此函数调用show\_time(sum(trip,day3))将执行函数调用sum(trip,day3)，以获得其返回值。然后，show\_time()调用将sum()的返回值（而不是函数自身）传递给show\_time()。下面是该程序的输出：

Two-day total: 10 hours, 40 minutes

Three-day total: 15 hours, 12 minutes

## 7.6.2 另一个处理结构的函数示例

前面介绍的有关函数和C++结构的大部分知识都可用于C++类中，因此有必要介绍另一个示例。这次要处理的是空间，而不是时间。具体地说，这个例了将定义两个结构，用于表示两种不同的描述位置的方法，然后开发一个函数，将一种格式转换为另一种格式，并显示结果。这个例子用到的数学知识比前一个要多，但并不需要像学习数学那样学习C++。

假设要描述屏幕上某点的位置，或地图上某点相对于原点的位置，则一种方法是指出该点相对于原点的水平偏移量和垂直偏移量。传统上，数学家使用x表示水平偏移量，使用y表示垂直偏移量。x和y一起构成了直角坐标（rectangular coordinates）。可以定义由两个坐标组成的结构来表示位置：

struct rect

{

double x; // 水平偏移量

double y; // 垂直偏移量

};

另一种描述点的位置的方法是，指出它偏离原点的距离和方向。传统上，数学家从正水平轴开始按逆时针方向度量角度。距离和角度一起构成了极坐标（polar coordinates）。可以定义另一个结构来表示这种位置：

struct polar

{

double distance; // 离开原点的距离

double angle; // 偏离原点的方向角度

};

下面来创建一个显示polar结构的内容的函数。C++库（从C语言借鉴而来）中的数学函数假设角度的单位为弧度，因此应以弧度为单位来测量角度。但为了便于显示，我们将弧度值转换为角度值。这意味着需要将弧度值乘以180/π一一约为57.29577951。该函数的代码如下：

// 显示极坐标，并将弧度转换为角度

void show\_polar (polar dapos)

{

using namespace std;

const double Rad\_to\_deg = 57.29577951;

cout << "distance = " << dapos.distance;

cout << ", angle = " << dapos.angle \* Rad\_to\_deg;

cout << " degrees\n";

}

请注意，形参的类型为polar。将一个polar结构传递绘该函数时，该结构的内容将被复制到dapos结构中，函数随后将使用该拷贝完成工作。由于dapos是一个结构，因此该函数便用成员运算符句点来标识结构成员。

接下来，让我们试着再前进一步，编写一个将直角坐标转换为极坐标的函数。该函数接受一个rect参数，并返回一个polar结构，这需要使用数学库中的函数，因此程序必须包含头文件cmath（在较旧的系统中为math.h）。另外，在有些系统中，还必须命令编译器载入数学库。可以根据勾股定理，使用水平和垂直坐标来计算距离：

distance = sqrt(x\*x + y\*y)

数学库中的atan2()函数可根据x和y的值计算角度：

angle = atan2 (y,x)

还有一个atan()函数，但它不能区分180度之内和之外的角度。

有了这些公式后，便可以这样编写该函数：

// 将直角坐标转换为极坐标

polar rect\_to\_polar(rect xypos)

{

using namespace std;

polar answer;

answer.distance =

sqrt( xypos.x \* xypos.x + xypos.y \* xypos.y);

answer.angle = atan2(xypos.y, xypos.x);

return answer; // 返回一个极坐标结构

}

编写好函数后，程序的其他部分编写起来就非常简单了。程序清单7.17列出了程序的代码。

**程序清单7.12 strctfun.cpp**

// strctfun.cpp -- 使用结构为参数的函数

#include <iostream>

#include <cmath>

// 结构声明

struct polar

{

double distance; // 离开原点的距离

double angle; // 偏离原点的方向角度

};

struct rect

{

double x; // 水平偏移量

double y; // 垂直偏移量

};

// 函数原型

polar rect\_to\_polar(rect xypos);

void show\_polar(polar dapos);

int main()

{

using namespace std;

rect rplace;

polar pplace;

cout << "Enter the x and y values: ";

while (cin >> rplace.x >> rplace.y) // slick use of cin

{

pplace = rect\_to\_polar(rplace);

show\_polar(pplace);

cout << "Next two numbers (q to quit): ";

}

cout << "Done.\n";

return 0;

}

// 将直角坐标转换为极坐标

polar rect\_to\_polar(rect xypos)

{

using namespace std;

polar answer;

answer.distance =

sqrt( xypos.x \* xypos.x + xypos.y \* xypos.y);

answer.angle = atan2(xypos.y, xypos.x);

return answer; // 返回一个极坐标结构

}

// 显示极坐标，并将弧度转换为角度

void show\_polar (polar dapos)

{

using namespace std;

const double Rad\_to\_deg = 57.29577951;

cout << "distance = " << dapos.distance;

cout << ", angle = " << dapos.angle \* Rad\_to\_deg;

cout << " degrees\n";

}

下面是该程序的运行情况：

Enter the x and y values: 30 40

distance = 50, angle=53.1301 degrees

Next two numbers(q to quit): -100 100

distance = 141.421 angle=135 degrees

Next two numbers(q to quit):q

### 程序说明

程序清单7.12中的两个函数已经在前面讨论了，因此下面复习一下该程序如何使用cin来控制while循环：

while (cin >> rplace.x >> rplace.y)

前面讲过，cin是istream类的一个对象。抽取运算符（>>）被设计成使得cin>>rplacc.x也是一个istream对象。正如第11章将介绍的，类运算符是使用函数实现的。使用cin>>rplace.x时，程序将调用一个函数，该函数返回一个istream值。将抽取运算符用于cin>>rplace.x对象（就像cin>>rplace.x>>rplace.y这样），也将获得一个istream对象。因此，整个while循环的测试表达式的最终结果为cin，而cin被用于测试表达式中时，将根据输入是否成功，被转换为bool值true或false。例如，在程序清单7.12中的循环中，cin期望用户输入两个数字，如果用户输入了q（前面的输出示例就是这样做的），cin>>将知道q不是数，从而将q留在输入队列中，并返回一个将被转换为fasle的值，导致循环结束。

请将这种读取数字的方法与下面更为简单的方法进行比较：

for(int i = 0;i < limit; i++)

{

cout<<"Enter value#”<< (i+1) << “:”;

cin>>temp;

if（temp < 0）

break;

ar[i] = temp;

}

要提早结束该循环，可以输入一个负值。这将输入限制为非负值。这种限制符合某些程序的需要，但通常需要一种不会将某些数值排除在外的、终止循环的方式。将cin>>用作测试条件消除了这种限制，因为它接受任何有效的数字输入。在需要使用循环来输入数字时，别忘了考虑使用这种方式。另外请记住，非数字输入将设置一个错误条件，禁止进一步读取输入。如果程序在输入循环后还需要进行输入，则必须使用cin.clear()重置输入，然后还可能需要通过读取不合法的输入来丢弃它们。程序清单7.7演示了这些技术。

## 7.6.3 传递结构的地址

假设要传递结构的地址而不是整个结构以节省时间和空间，则需要重新编写前面的函数，使用指向结构的指针。首先来看一看如何重新编写show\_polar()函数。需要修改三个地方：

* 调用函数时，将结构的地址（&pplace）而不是结构本身（pplace）传递给它；
* 将形参声明为指向polar的指针，即polar \*类型。由于函数不应该修改结构，因此使用了const修饰符；
* 由于形参是指针而不是结构，因此应间接成员运算符（->），而不是成员运算符（句点）。

完成上述修改后，该函数如下所示：

void show\_polar (const polar \* pda)

{

using namespace std;

const double Rad\_to\_deg = 57.29577951;

cout << "distance = " << pda->distance;

cout << ", angle = " << pda->angle \* Rad\_to\_deg;

cout << " degrees\n";

}

接下来对rect\_to\_polar进行修改。由于原来的rect\_to\_polar函数返回一个结构，因此修改工作更复杂些。为了充分利用指针的效率，应使用指针，而不是返回值。为此，需要将两个指针传递给该函数，第一个指针指向要转换的结构，第二个指针指向存储转换结果的结构。函数不返回一个新的结构，而是修改调用函数中已有的结构。因此，虽然第一个参数是const指针，但第二个参数却不是。也可以像修改函数show\_polar()修改这个函数。程序清单7.13列出了修改后的程序。

**程序清单7.1.3 strctptr.cpp**

// strctptr.cpp -- 使用结构指针为参数的函数

#include <iostream>

#include <cmath>

// 结构

struct polar

{

double distance; // 离开原点的距离

double angle; // 偏离原点的方向角度

};

struct rect

{

double x; // 水平偏移量

double y; // 垂直偏移量

};

// 函数原型

void rect\_to\_polar(const rect \* pxy, polar \* pda);

void show\_polar (const polar \* pda);

int main()

{

using namespace std;

rect rplace;

polar pplace;

cout << "Enter the x and y values: ";

while (cin >> rplace.x >> rplace.y)

{

rect\_to\_polar(&rplace, &pplace); // 传递地址

show\_polar(&pplace); // 传递地址

cout << "Next two numbers (q to quit): ";

}

cout << "Done.\n";

return 0;

}

// 显示极坐标，并将弧度转换为角度

void show\_polar (const polar \* pda)

{

using namespace std;

const double Rad\_to\_deg = 57.29577951;

cout << "distance = " << pda->distance;

cout << ", angle = " << pda->angle \* Rad\_to\_deg;

cout << " degrees\n";

}

// 将直角坐标转换为极坐标

void rect\_to\_polar(const rect \* pxy, polar \* pda)

{

using namespace std;

pda->distance =

sqrt(pxy->x \* pxy->x + pxy->y \* pxy->y);

pda->angle = atan2(pxy->y, pxy->x);

}

从用户的角度来说，程序清单7.13的行为与程序清单7.12相同。它们之间的差别在于，程序清单7.12使用的是结构副本，而程序清单7.13使用的是指针，让函数能够对原始结构进行操作。