# 7.3 函数和数组

到目前为止的函数示例都很简单，参数和返回值的类型都是基本类型。但是，函数是处理更复杂的类型（如数组和结构）的关键。下面来如何将数组和函数结合在一起。

假设使用一个数组来记录家庭野餐中每人吃了多少个甜饼（每个数组索引都对应一个人，元素值对应于这个人所吃的甜饼数量）。现在想知道总数。这很容易，只需使用循环将所有数组元素累积起来即可。将数组元素累加是一项非常常见的任务，因此设计一个完成这项工作的函数很有意义。这样就不必在每次计算数组总和时都编写新的循环了。

考虑函数接口所涉及的内容。由于函数计算总数，因此应返回答案。如果不分吃甜饼，则可以让函数的返回类型为int。另外，函数需要知道要对哪个数组进行累计，因此需要将数组名作为参数传递给它，为使函数通用，而不限于特定长度的数组，还需要传递数组长度。这里唯一的新内容是，需要将一个形参声明为数组名。下面来看一看函数头及其其他部分：

int sum\_arr(int arr[],int n) // arr = array name,n=size

这看起来似乎合理。方括号指出arr是一个数组，而方括号为空则表明，可以将任何长度的数组传递给该函数。但实际情况并非如此：arr实际上并不是数组，而是一个指针！好消息是，在编写函数的其余部分时，可以将arr看作是数组。首先，通过一个示例验证这种方法可行，然后看看它为什么可行。

程序清单7.5演示如同使用数组名那样使用指针的情况。程序将数组初始化为某些值，并使用sum\_arr()函数计算总数。注意sum\_arr()函数使用arr时，就像是使用数组名一样。

**程序清单7.5 arrfun1.cpp**

// arrfun1.cpp -- 以数组为参数的函数

#include <iostream>

const int ArSize = 8;

int sum\_arr(int arr[], int n); // 函数原型

int main()

{

using namespace std;

int cookies[ArSize] = {1,2,4,8,16,32,64,128};

// some systems require preceding int with static to

// enable array initialization

int sum = sum\_arr(cookies, ArSize);

cout << "Total cookies eaten: " << sum << "\n";

cin.get();

return 0;

}

// 返回一个整数数组的元素之和

int sum\_arr(int arr[], int n)

{

int total = 0;

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

total = total + arr[i];

return total;

}

下面是该程序的输出：

Total cookies eaten:255

从中可知，该程序管用。下面讨论为何该程序管用。

## 7.3.1 函数如何使用指针来处理数组

在大多数情况下，C++和C语言一样，也将数组名视为指针。C++将数组名解释为其第一个元素的地址：

cookies == &cookies[0] // 数组名称就是第一个元素的地址

该规则有一些例外。首先，数组声明使用数组名来标记存储位置；其次，对数组名使用sizeof将得到整个数组的长度（以字节为单位）；第三，将地址运算符&用于数组名时，将返回整个数组的地址，例如&cookies将返回一个32字节内存块的地址（如果int长4字节）。

程序清单7.5执行下面的函数调用：

int sum = sum\_arr(cookies,ArSize);

其中，cookies是数组名，而根据C++规则，cookies是其第一个元素的地址。因此函数传递的是地址。由于数组的元素的类型为int，因此cookies的类型必须是int指针，即int \*。这表明，正确的函数头应该是这样的：

int sum\_arr(int \* arr,int n） // arr = array name,n = size

其中用int \* arr替换了int arr[]。这证明这两个函数头部是正确的，因为在C++中，当（且仅当）用于函数头或函数原型，int \* arr和int arr[]的含义才是相同的。它们都意味着arr是一个int指针。然而，数组表示法（int arr[]）提醒用户，arr不仅指向int，还指向int数组的第一个int。当指针指向数组的第一个元素时，本书使用数组表示法；而当指针指向一个独立的值时，使用指针表示法。别忘了，在其他的上下文中，int \* arr和int arr[]的含义并不相同。例如，不能在函数体中使用int tip[]来声明指针。

鉴于变量arr实际上就是一个指针，函数的其余部分是合理的。第4章在介绍动态数组时指出过，同数组名或指针一样，也可以用方括号数组表示法来访问数组元素。无论arr是指针还是数组名，表达式arr[3]都指的是数组的第4个元素。就目前而言，记住下面两个恒等式，将不会有任何坏处：

arr[i] == \*(ar+i) // 两种标记法都能表示值

&arr[i] == ar + i; // 两种标记法都能表示地址

记住，将指针（包括数组名）加1，实际上是加了一个与指针指向的类型的长度（以字节为单位）相等的值。对于遍历数组而言，使用指针加法和数组下标时等效的。

## 7.3.2 将数组作为参数意味着什么

我们来看一看程序清单7.5暗示了什么。函数调用sum\_arr(cookies，ArSize)将cookies数组第一个元素的地址和数组中的元素数目传递给sum\_arr()函数。sum\_arr()函数将cookies的地址赋给指针变量arr，将ArSize赋给int变量n。这意味着，程序清单7.5实际上并没有将数组内容传递给函数，而是将数组的位置（地址）、包含的元素种类（类型）以及元素数目（n变量）提交给函数。有了这些信息后，函数便可以使用原来的数组。传递常规变量时，函数将使用该变量的拷贝；但传递数组时，函数将使用原来的数组。实际上，这种区别并不违反C++按值传递的方法，sum\_arr()函数仍传递了一个值，这个值被赋给一个新变量，但这个值是一个地址，而不是数组的内容。

数组名与指针对应确实是一件好事。将数组地址作为参数可以节省复制整个数组所需的时间和内存。如果数组很大，则使用拷贝的系统开销将非常大；程序不仅需要更多的计算机内存，还需要花费时间来复制大块的数据。另一方面，使用原始数据增加了破坏数据的风险。我们修改程序清单7.5，以演示数组函数是如何运作的。程序清单7.6表明，cookies和arr的值相同。它还演示了指针概念如何使sum\_arr函数比以前更通用。该程序使用限定符std::而不是编译指令using来提供对cout和endl的访问权。

**程序清单7.6 arrfun2.cpp**

// arrfun2.cpp -- 以数组为参数的函数

#include <iostream>

const int ArSize = 8;

int sum\_arr(int arr[], int n);

// use std:: instead of using directive

int main()

{

int cookies[ArSize] = {1,2,4,8,16,32,64,128};

// some systems require preceding int with static to

// enable array initialization

std::cout << cookies << " = array address, ";

// some systems require a type cast: unsigned (cookies)

std::cout << sizeof cookies << " = sizeof cookies\n";

int sum = sum\_arr(cookies, ArSize);

std::cout << "Total cookies eaten: " << sum << std::endl;

sum = sum\_arr(cookies, 3); // a lie

std::cout << "First three eaters ate " << sum << " cookies.\n";

sum = sum\_arr(cookies + 4, 4); // another lie

std::cout << "Last four eaters ate " << sum << " cookies.\n";

std::cin.get();

return 0;

}

// 返回一个整数数组的元素之和

int sum\_arr(int arr[], int n)

{

int total = 0;

std::cout << arr << " = arr, ";

// some systems require a type cast: unsigned (arr)

std::cout << sizeof arr << " = sizeof arr\n";

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

total = total + arr[i];

return total;

}

下面是该程序的输出（地址值和数组的长度将随系统而异）：

003EF9FC = array address , 32 =size of cookies

003EF9FC = arr , 4 = sizeof arr

Total cookies eaten:255

003EF9FC =arr , 4 = sizeof arr

First three eaters ate 7 cookies

003EF9FC =arr ,4 = sizeof arr

Last four eaters ate 240 cookies

注意，地址值和数组的长度随系统而异。另外，有些C++实现以十进制而不是十六进制格式显示地址，还有些编译器以十六进制显示地址时，会加上前缀0x。

### 程序说明

程序清单7.6说明了数组函数的一些有趣的地方。首先，cookies和arr指向同一个地址。但sizeof cookies的值为32，而sizeof arr为4，这是由于sizeof cookies是整个数组的长度，而sizeof arr只是指针变量的长度（上述程序运行结果是从一个使用4字节地址的系统中获得的）。顺便说一句，这也是必须显式传递数组长度，而不能在sum\_arr()中使用sizeof arr的原因；指针本身并没有指出数组的长度。

由于sum\_arr只能通过第二个参数获知数组中的元素数量，因此可以对函数“说谎”。例如，程序第二次使用该函数时，这样调用它：

sum\_arr(cookies,3);

通过告诉该函数cookies有3个元素，可以让它计算前3个元素的总和。

还可以提供假的数组起始位置：

sum\_arr(cookies+4,4);

由于cookies是第一个元素的地址，因此cookies+4是第5个元素的地址。这条语句将计算数组第？、6、7、8个元素的总和。请注意输出中第三次函数调用选择将不同于前两个调用的地址赋给arr的。可以将&cookies[4]，而不是cookies+4作为参数。它们的含义是相同的。

注意：为将数组类型和元素数量告诉数组处理函数，请通过两个不同的参数来传递它们：

void fillArray(int arr[],int size); // 函数原型

而不要试图使用方括号表示法来传递数组长度：

void fillArray(int arr[size]); // 不行，错误的函数原型

## 7.3.3 更多数组函数示例

选择使用数组来表示数据时，实际上是在进行一次设计方面的决策。但设计决策不仅仅是确定数据的存储方式，还涉及到如何使用数据。程序员常会发现，编写特定的函数来处理特定的数据操作是有好处的（这里讲的好处指的是程序的可靠性更高，修改和调试更为方便）。另外，构思程序时将存储属性与操作结合起来，便是朝OOP思想迈进的重要的一步；以后将证明这是很有好处的。

来看一个简单的案例。假设要使用一个数组来记录房地产的价值（假设拥有房地产）。在这种情况下，程序员必须确定要使用哪种类犁。当然，double的取值范围比int和long大，并且提供了足够多的有效位数来精确地表示这些值。接下来必须决定数组元素的数目。（对于使用new创建的动态数组来说，可以稍后再决定，但我们希望使事情简单一点）。如果房地产数目不超过5个，则可以使用一个包含5个元素的double数组。

现在，考虑要对房地产数组执行的操作。两个基本的操作分别是，将值读入到数组中和显示数组内容。我们再添加另一个操作：重新评估每种房地产的值。为简单起见，假设所有房地产都以相同的的比率增加或者减少。接下来，为每项操作编写个函数，然后编写相应的代码。下面首先介绍这些步骤，然后将其用于一个完整的示例中。

### 1．填充数组

由于接受数组名参数的函数访问的是原始数组，而不是其副本，因此可以通过调用该函数将值赋给数组元素。该函数的一个参数是要填充的数组的名称。通常，程序可以管埋多个人的投资，因此需要多个数组，因此不能往函数中设置数组长度，而要将数组长度作为第二个参数传递，就像前一个示例那样。另外，用户也可能希望在数组被填满之前停止读取数据，因此需要在函数中建立这种特性。由于用户输入的元素书目可能少于数组的长度，因此函数应返回实际输入的元素数目。因此，该函数的原型如下：

int fill\_array(double ar[]，int limit）;

该函数接受两个参数，一个是数组名，另一个指定了要读取的最大元素数；该函数返回实际读取的元素数。例如，如果使用该函数来处理一个包含5个元素的数组，则将5作为第二个参数。如果只输入3个值，则该函数将返回3。

可以使用循环连续地将值读入到数组中，但如何提早结束循环呢？一种方法是，使用一个特殊值来指出输入结束。由于所有的属性都不为负，因此可以使用负数来指出输入结束。另外，该函数应对错误输入作出反应，如停止输入等。这样，该函数的代码如下所示：

int fill\_array(double ar[], int limit)

{

using namespace std;

double temp;

int i;

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

{

cout << "Enter value #" << (i + 1) << ": ";

cin >> temp;

if (!cin) // bad input

{

cin.clear();

while (cin.get() != '\n')

continue;

cout << "Bad input; input process terminated.\n";

break;

}

else if (temp < 0) // signal to terminate

break;

ar[i] = temp;

}

return i;

}

注意，代码中包含了对用户的提示。如果用户输入的是非负值，则这个值将被赋给数组，否则循环结束。如果用户输入的都是有效值，则循环将在读取最大数目的值后结束。循环完成的最后一项工作是将i加1，因此循环结束后，i将比最后一个数组索引大1，即等于填充的元素数目。然后，函数返回这个值。

### 2．显示数组及用const保护数组

创建显示数组内容的函数很简单。只需将数组名和填充的元素数目传递给函数，然后该函数使用循环来显示每个元素。然而，还有另一个问题——确保显示函数不修改原始数组。除非函数的目的就是修改传递给它的数据，否则应避免发生这种情况。使用普通参数时，这种保护将自动实现，这是由于C++按值传递数据，而且函数使用数据的副本。然而，接受数组名的函数将使用原始数据，这正是fill\_array()函数能够完成其工作的原因。为防止函数无意中修改数组的内容，可在声明形参时使用关键const：

void show\_array(const double ar[],int n);

该声明表明，指针ar指向的是常量数据。这意味着不能使用ar修改该数据，也就是说，可以使用像ar[0]这样的值，但不能修改；注意，这并不是意味着原始数组必须是常量，而只是意味着能在show\_array()函数中使用ar来修改这些数据。因此，show\_array()将数组视为只读数据。假设无意间在show\_array()函数中执行了下面的操作，从而违反了这种限制：

ar[0]+=10;

编译器将禁止这样做。

C++将声明const double ar[]解释为const double \* ar。因此，该声明实际上是说，a指向的是一个常量值。结束这个例子后，我们将详细讨论这个问题。下而是show\_array()函数的代码：

// 下面的函数只能使用，但不能修改地址为ar的数组

void show\_array(const double ar[], int n)

{

using namespace std;

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

{

cout << "Property #" << (i + 1) << ": $";

cout << ar[i] << endl;

}

}

### 3．修改数组

在这个例子中，对数组进行的第三项操作是将每个元素与同一个重新评估因子相乘。需要给函数传递3个参数：因子、数组和元素数目。该函数不需要返回值，因此其代码如下：

// 将ar[]数组中的每个元素乘以r

void revalue(double r, double ar[], int n)

{

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

ar[i] \*= r;

}

由于这个函数将修改数组的值，因此在声明ar时，不能使用const。

### 4．将上述代码组合起来

至此，您根据数据的存储方式（数组）和使用方式（3个函数）定义了数据的类型，因此可以将它们组合成一个程序。由于已经建立了所有的数组处理工具，因此main()的编程工作非常简单。该程序检查用户输入的是否是数字，如果不是，则要求用户这样做。余下的大部分编程工作只是让main()调用前面开发的函数。程序清单7.7列出了最终的代码，它将编译指令using放在那些需要iostream工具的函数中。

**程序清单7.7 arrfun3.cpp**

// arrfun3.cpp -- 数组处理函数和const

#include <iostream>

const int Max = 5;

// 函数原型

int fill\_array(double ar[], int limit);

void show\_array(const double ar[], int n); // 不能改变数据

void revalue(double r, double ar[], int n);

int main()

{

using namespace std;

double properties[Max];

int size = fill\_array(properties, Max);

show\_array(properties, size);

if (size > 0)

{

cout << "Enter revaluation factor: ";

double factor;

while (!(cin >> factor)) // bad input

{

cin.clear();

while (cin.get() != '\n')

continue;

cout << "Bad input; Please enter a number: ";

}

revalue(factor, properties, size);

show\_array(properties, size);

}

cout << "Done.\n";

cin.get();

cin.get();

return 0;

}

int fill\_array(double ar[], int limit)

{

using namespace std;

double temp;

int i;

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

{

cout << "Enter value #" << (i + 1) << ": ";

cin >> temp;

if (!cin) // bad input

{

cin.clear();

while (cin.get() != '\n')

continue;

cout << "Bad input; input process terminated.\n";

break;

}

else if (temp < 0) // signal to terminate

break;

ar[i] = temp;

}

return i;

}

// 下面的函数只能使用，但不能修改地址为ar的数组

void show\_array(const double ar[], int n)

{

using namespace std;

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

{

cout << "Property #" << (i + 1) << ": $";

cout << ar[i] << endl;

}

}

// 将ar[]数组中的每个元素乘以r

void revalue(double r, double ar[], int n)

{

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

ar[i] \*= r;

}

下面是两次运行改程序时的输出：

Enter value #l: 100000

Enter value #2 : 80000

Enter value #3 : 222000

Enter value #4 : 240000

Enter value #5 : 118000

Property #1: $100000

Property #2 : $8000

Property #3 : $222000

Property #4 : $240000

Property #5 : $118000

Enter revaluation factor : 0.8

Property #1: $80000

Property #2; $64000

Property #3 : $177600

Property #4 : $192000

Property #5 : $94400

Done.

Enter value #1: 200000

Enter value #2 : 84000

Enter value #3 : 160000

Enter value #4 : -2

Property #1: $200000

Property #2 : $84000

Property #3; $160000

Enter reevaluation factor : 1.20

Property #1: $240000

Property #2 : $100800

Properry #3 : $192000

Done.

fill\_array()指出，当用户输入5项房地产值或负值后，将结束输入。第一次运行演示了输入5项房地产值的情况，第二次运行演示了输入负值的情况。

### 5．程序说明

前面已经讨论了与该示例相关的重要编程细节，因此这里回顾一下整个过程。我们首先考虑的是通过数据类型和设计适当的函数来处理数据，然后将这些函数组合成一个程序。有时也称为**自下而上的程序设计（bottom-up programming）**，因为设计过程从组件到整体进行。这种方法非常适合于OOP——它首先强调附是数据表示和操纵。而传统的过程性编程倾向于**从上而下的程序设计（top-down programming）**，首先指定模块化设计方案，然后再研究细节。这两种方法都很有用，最终的产品都是模块化程序。

### 6．数组处理函数的常用编写方式

假设要编写一个处理double数组的函数。如果该函数要修改数组，其原型可能类似于下面这样：

void f\_modify(double ar[],int n);

如果函数不修改数组，其原型可能类似于下面这样：

void f\_no\_change(const double ar[],int n);

当然，在函数原型中可以省略变量名，也可将返回类型指定为类型。这里的要点是，ar实际上是一个指针，指向传入的数组的第一个元素：另外，由于通过参数传递了元素数，这两个函数都可使用任何长度的数组，只要数组的类型为double：

double rewards[1000];

double faults[50];

…

f\_modify(rewards,1000);

f\_modify(faults,50);

这种做法是通过传递两个数字（数组地址和元素数）实现的，但函数缺少一些有关原始数组的信息；例如，它不能使用sizeof来获悉原始数组的长度，而必须依赖于程序员传入正确的元素数。

## 7.3.4 使用数组区间的函数

对于处理数组的C++函数，必须将数组中的数据种类、数组的起始位置和数组中元素数量提交给它；传统的C/C++方法是，将指向数组起始处的指针作为一个参数，将数组长度作为第二个参数（指针指出数组的位置和数据类型），这样便给函数提供了找到所有数据所需的信息。

还有另一种给函数提供所需信息的方法，即指定元素**区间（range）**，这可以通过传递两个指针来完成：一个指针标识数组的开头，另一个指针标识数组的尾部。例如，C++标准模板库（STL，将在第16章介绍）将区间方法广义化了。STL方法使用“超尾”概念来指定区间。也就是说，对于数组而言，标识数组结尾的参数将是指向最后一个元素后面的指针。例如，假设存这样的声明：

double elboud[20];

则指针elboud和elboud+20定义了区间。首先，数组名elboud指向第一个元素。表达式elboud+19指向最后一个元素（即elboud[19]），因此，elboud+20指向数组结尾后面的一个位置。将区间传递给函数将告诉函数应处理哪些元素。程序清单7.8对程序清单7.6做了修改，使用两个指针来指定区间。

**程序清单7.8 arrfun4.cpp**

// arrfun4.cpp -- 使用数组区间的函数

#include <iostream>

const int ArSize = 8;

int sum\_arr(const int \* begin, const int \* end);

int main()

{

using namespace std;

int cookies[ArSize] = {1,2,4,8,16,32,64,128};

// some systems require preceding int with static to

// enable array initialization

int sum = sum\_arr(cookies, cookies + ArSize);

cout << "Total cookies eaten: " << sum << endl;

sum = sum\_arr(cookies, cookies + 3); // 前3个元素

cout << "First three eaters ate " << sum << " cookies.\n";

sum = sum\_arr(cookies + 4, cookies + 8); // 最后4个元素

cout << "Last four eaters ate " << sum << " cookies.\n";

cin.get();

return 0;

}

// 返回一个整数数组的元素之和

int sum\_arr(const int \* begin, const int \* end)

{

const int \* pt;

int total = 0;

for (pt = begin; pt != end; pt++)

total = total + \*pt;

return total;

}

下面是该程序的输出：

Total cookies eaten:255

First three eaters ate 7 cookies.

Last four eaters ate 240 cookies.

### 程序说明

请注意程序清单7.8中sum\_array()函数中的for循环：

for (pt = begin; pt != end; pt++)

total = total + \*pt;

它将pt设置为指向要处理的第一个元素（begin指向的元素）的指针，并将\*pt（元素的值）加入到total中。然后，循环通过递增操作来更新pt，使之指向下一个元素。只要pt不等于end．这一过程就将继续下去。当pt等于end时，它将指向区间中最后一个元素后面的一个位置，此时循环将结束。

其次，请注意不同的函数调用是如何指定数组中不同的区间的：

int sum = sum\_arr[cookies,cookies+ ArSize);

…

sum = sum\_arr(cookies , cookies+3);

…

sum = sum\_arr(cookies+4 , cookies + 8);

指针cookies+ArSize指向最后一个元素后面的一个位置（数组有ArSize个元素，因此cookies[ArSize-1]是最后一个元素，其地址为cookies+ ArSize -1）。因此，区间[cookies , cookies+ArSize]指定的是整个数组。同样，cookies，cookies+3指定了前3个元素，依此类推。

请注意，根据指针减法规则，在sum\_arr()中，表达式end-begin是一个整数值，等于数组的元素数目。

另外，必须按正确的顺序传递指针，因为这里的代码假定begin在前面，end在后面。

## 7.3.5 指针和const

将const用于指针有一些很微妙的地方，我们来详细探讨一下。可以用两种不同的方式将const关键字用于指针。第一种方法是让指针指向一个常量对象，这样可以防止使用该指针来修改所指向的值，第二种方法是将指针本身声明为常量，这样可以防止改变指针指向的位置。下面来看细节。

首先，声明一个指向常量的指针pt：

int age=39;

const int\* pt = &age;

该声明指出，pt指向一个const int（这里为39），因此不能使用pt来修改这个值。换句话说，\*p的值为const，不能被修改：

\*pt += 1 // 无效，因为pt指向一个const int

cin >> \*pt // 无效，理由一样

现在来看一个微妙的问题。pt的声明并不意味着它指向的值实际上就是一个常量，而只是意味着对pt而言，这个值是常量。例如，pt指向age，而age不是const。可以直接通过age变量来修改age的值，但不能使用pt指针来修改它：

\*pt = 20; // 错误，因为pt指向一个const int

age = 20; // 无效，因为age并没有被声明为const

以前我们将常规变量的地址赋给常规指针，而这里将常规变量的地址赋给指向const的指针。因此还有两种可能：将const变量的地址赋给指向const的指针、将const的地址赋给常规指针。这两种操作都可行吗？第一种可行，但第二种不可行：

const float g\_earth = 9.80;

const float \*pe = &g\_earth; // 正确

const float g\_moon = 1.63;

float \*pm = &g\_moon; // 无效

对于第一种情况来说，既不能使用g\_earth来修改值9.80，也不能使用pe来修改。C++禁止第二种情况的原因很简单——如果将g\_moon的地址赋给pm，则可以使用pm来修改g\_moon的值，这使得g\_moon的const状态很荒谬，因此C++禁止将const的地址赋给非const指针。如果读者非要这样做，可以使用强制类型转换来突破这种限制，详情请参阅第15章中对运算符const\_cast的讨论。

如果将指针指向指针，则情况将更复杂。前面讲过，假如涉及的是一级间接关系，则将非const指针赋给const指针是可以的：

int age = 39 // age++是有效的操作

int \*pd = &age // \*pd = 41是有效的操作

const int \*pt = pd // \*pt = 42是无效的操作

然而，进入两级间接关系时，与一级间接关系一样将const和非const混合的指针赋值方式将不再安全。如果允许这样做，则可以编写这样的代码：

const int \*\*pp2;

int \*p1;

const int n = 13;

pp2 = &p1; // 不允许。but suppose it were

\*pp2 = &n; // 有效，都是const，p1设置为指向n的指针

\*p1= 10; // 有效，修改了const n

上述代码将非const地址（&p1）赋给了const指针（pp2），因此可以使用p1来修改const数据。因此，仅当只有一层间接关系（如指针指向基本数据类型）时，才可以将非const地址或指针赋给const指针。

**注意**：如果数据类型本身并不是指针，则可以将const数据或非const数据的地址赋给指向const的指针，但只能将非const数据的地址赋给非const指针。

假设有一个由const数据组成的数组：

const int months[12] = {31,28,31,30,31,30,31,31,30,31，30，31};

则禁止将常量数组的地址赋给非常量指针将意味着不能将数组名作为参数传递给使用非常量形参的函数：

int sum(int arr[],int n); // should have been const int arr[]

…

int j = sum(months,12); // 不允许

上述函数调用试图将const指针（months）赋给非const指针（arr），编译器将禁止这种函数调用。

**尽可能使用const**

将指针参数声明为指向常量数据的指针有两条理由：

* 这样可以避免由于无意间修改数据而导致的编程错误；
* 使用const使得函数能够处理const和非const实参，否则将只能接受非const数据。

如果条件允许，则应将指针形参声明为指向const的指针。

为说明另一个微妙之处，请看下面的声明：

int age = 39;

const int \*pt = &age;

第二个声明中的const只能防止修改pt指向的值（这里为39），而不能防止修改pt的值。也就是说，可以将一个新地址赋给pt：

int sage = 80;

pt = &sage; // 将指针指向另一个地址是可以的

但仍然不能使用pt来修改它指向的值（现在为80）。

第二种使用const的方式使得无法修改指针的值：

int sloth = 3;

const int \*ps = &sloth; // 一个指向常量int的指针

int \* consc finger = &sloth; // 一个指向int的常量指针

在最后一个声明中，关键字const的位置与以前不同。这种声明格式使得finger只能指向sloth，但允许使用finger来修改sloth的值。中间的声明不允许使用ps来修改sloth的值，但允许将ps指向另一个位置。简而言之，finger和\*ps都是const，而\*finger和ps不是。

如果愿意，还可以声明指向const对象的const指针：

double trouble = 2.0E30;

const double \*const stick = &trouble;

其中，stick只能指向trouble，而stick不能用来修改trouble的值。简而言之，stick和\*stick都是const。

通常，将指针作为函数参数来传递时，可以使用指向const的指针来保护数据。例如，程序清单7.5中的show\_array()的原型：

void show\_array(const double ar[],int n);

在该声明中使用const意味着show\_array()不能修改传递给它的数组中的值。只要只有一层间接关系，就可以使用这种技术。例如，这里的数组元素是基本类型，但如果它们是指针或指向指针的指针，则不能使用const。