C/C++中的指针、数组和动态变量

上海交通大学　危国锐

最后修改：2019-11-16

1. 变量名

变量名是某块内存空间的**标识符**，可以通过变量名访问此内存空间，谓之**直接访问**。变量的数据类型和其他限定符（例如 const）规定了通过此变量名能进行的操作。

2. 引用类型

定义

```
T var; T &VAR = var;
```

就是将 VAR 作为 T 类型的变量 var 所标识的内存空间的另一个标识符。若定义

```
const T &VAR = var;
```

则尽管 VAR 和 var 都是同一块内存空间的标识符，但如果要赋值，只能通过标识符 var。

另一种理解：引用是一种隐式指针常量，且自带解引用运算符*。定义

```
T &VAR = var;
```

相当于定义

```
T *const VAR = &var;
```

调用 VAR 事实上是*VAR，但强制省略*。

3. 引用运算符*和取地址运算符&

定义

```
T var;T *ptr = &var;
```

就是在内存中申请一块存放 T 类型的变量所需的那么大的空间，并为这块空间定义一个标识符 var。表达式&var 返回的是由 T 类型的变量名 var 标识的内存空间的首地址，其类型定义为 T*。

注：虽然*在语法上属于变量名，但不允许再定义一个标识符为 var 的变量。所以在逻辑上，认为*属于类型名，不会引起错误。

定义了一个 T*类型的变量 ptr，此变量在内存中通常占一个主机字长(64 位机是 8 字节)。由变量名 ptr 标识的内存空间中存放的是 var 的地址，例如 0xFFFFFFFF。称 ptr 是一个**指针**，“指向”var，也称 ptr 是 T*类型的变量。

当一个变量 m 存放的是另一个变量 n 的地址，就说 m“指向”n，表达式 m（被变量名 m 标识的内存空间上存储的数据）和&n（被变量名 n 标识的内存空间的首地址）具有相同的值，m 的类型是 type(n)*。在逻辑表示中，可用一个 m 方框加指向 n 方框的箭头表示 m。有时省略 m 方框，只用一个指向 n 的名为 m 的箭头表示 m，也不会引起错误。

定义后，表达式*ptr 返回的是指针变量 ptr 所指内存空间的临时标识符，可通过此标识符访问那块内存空间，也可通过语句

```
T &VAR = *ptr;
```

为那块内存空间定义一个永久标识符 VAR。

4. 返回引用的函数

本源代码中的函数 reftest，若原型无&，则为普通函数，返回的是 return 后的表达式的值（存放于临时变量，取代主函数中的函数调用）。若有&，则 return 后的表达式应是某块在函数返回后仍可访问的内存空间（如存放于全局变量区的全局变量或存放于堆中的动态变量）的标识符：例如某个变量名（包括下标变量），或指针引用表达式如本文第 3 节所述之*ptr 所返回的临时标识符。

称此函数返回了**一个变量的引用**。可在主调函数中代函数调用以一个临时标识符（临时变量名），进而可通过这个函数调用为一块内存空间定义永久标识符（即定义引用类型的变量），也可通过这个函数调用访问一块内存空间，正如通过变量名访问那样。

5. 多级指针

如果变量 ptr 是 T*类型的，即 ptr 是指向 T 类型变量的一级指针，那么表达式&ptr 返回的是 T*类型的变量 ptr 的地址，其类型定义为 T**。这样，语句

```
T **ptr2 = &ptr;
```

定义了一个 T**类型的变量 ptr2，保存的是 T*类型的变量 ptr 的内存首地址，称 ptr2 是指向 ptr 的指针，ptr2 是 T 类型的二级指针，ptr2 是 T**类型的变量。进而，表达式&ptr2 返回的是一个 T***类型的值，可通过它定义一个指向 ptr2 的三级指针：

```
T ***ptr3 = &ptr2。
```

6. 指针变量、数组名和指向数组的指针

语句

```
int a[2][3][4];
```

定义了一个数组名为 a 的三维数组，也称 a 是 int[2][3][4]型的变量（数组）。数组名 a 是指向数组首维首元素的指针常量，即 a 指向 a[0]。a[0]视作一个具有规模[3][4]的 int 型二维数组的数组名，所以 a 是指向 int[3][4]型变量 a[0]的指针常量，定义 a 的类型是 int(*)[3][4]，表示指向具有规模[3][4]的 int 型数组的指针。

注：语句

```
int *p[3][4];
```

表示定义一个具有规模[3][4]的 int*型二维数组 p。

语句

```
int(*ptr3)[3][4] = a;
```

定义了一个指向 int[3][4]型数组的指针 ptr3。由于 a 是指向 a[0]的指针，由 3，语句

```
int(*ptr3)[3][4] = &a[0];
```

与前述者等价。此后，可通过变量名 ptr3 如同变量名 a 那样以三级引用的形式访问任何下标变量。本代码测试了这种访问形式的代数性质。

a[0]又可视作一个二维数组的变量名，从而 a[0]是指向其首维首元素 a[0][0]的指针。a[0][0]视作一个具有规模[4]的 int 型一维数组的变量名，所以 a[0]指向了一个 int[4]类型的变量 a[0][0]，a[0]就是 int(*)[4]类型的。语句

```
int(*ptr2)[4] = a[0];
```

或

```
int(*ptr2)[4] = &a[0][0];
```

定义了指向一维数组的指针 ptr2，可以如同 a[0]那样以 ptr2 的二级引用的形式访问任何下标变量。

同理，a[0][0]又可视作一个一维数组的变量名，从而 a[0][0]指向 int 型变量 a[0][0][0],从而是 int*类型的。语句

```
int* ptr1 = a[0][0];
```

与

```
int *ptr1 = &a[0][0][0];
```

等效，此后可通过 ptr1 的一级引用形式访问任何下标变量。

注：**多级引用**可通过引用运算符*和/或方括号[]实现，见代码。

事实上，常量 a，a[0]，a[0][0]，&a[0][0][0]的值都是同一个地址，即下标变量 a[0][0][0]的地址。但前三者的数据类型不同，分别是指向二维数组、一维数组、零维数组即普通变量的指针，从而分别可以进行三级、二级、一级引用。多级引用时，都适用基类型代数运算，均可访问整个三维数组的任何下标变量。

小结：n 维数组的数组名是一个指向 n-1 维数组的指针，普通变量可视作 0 维数组。n 维数组名的 1 级引用可视作一个 n-1 维数组的数组名，即一个指向 n-2 维数组的指针。

7. 动态变量的创建和消亡

（1）操作符 new 和 delete

new 操作符用来向内存的堆区申请一块连续的空间，并返回该空间的首地址即指向空间首元素的指针。例如语句

```
int **p = new int *[d];
```

意为申请一块规模为 3 的连续空间，存放类型为 int*型的变量，返回指向首元素（int*型）的指针，即 int**型。

delete[]操作符用来释放一块连续的空间，其操作数是一个指针，且要求该指针指向某个连续空间的首元素。若无[]，则只释放操作数指向的空间。

注：视一个普通变量是 0 维数组，其规模为 1。若不指定 new 的规模，则视作数组规模为 1 或维数为 0，这样就能统一理解动态变量和动态数组。

（2）在函数调用中创建动态变量

在被调函数中 new 的动态变量（包括数组），一定有一个局部指针变量指向它（对于数组，则是指向数组首元素）。可以通过指针参数传递或引用传递、或返回引用的函数，把指向动态变量的指针（通过参数传递）、或标识动态变量所属内存空间的临时标识符（通过返回引用），赋给主调函数中的指针变量、或进行引用类型定义即在主调函数中定义永久标识符（见 4.）。这样，在主调函数中就可以继续访问 new 的变量。

对于动态数组，delete[]操作符的操作数必须是数组首地址，即指向数组首元素的指针或&name，name 是数组首地址的一个标识符。

8. 动态高维数组的创建和消亡

下面以动态三维数组为例，说明如何创建和消亡动态高维数组。首先定义数组的各维规模 int d1 = 2, d2 = 3, d3 = 4;。

（1）动态高维数组的创建

若声明 int ***a;或 int a[d1][d2][d3];，则表达式 a[i][j][k]与表达式

((*(a + i) + j) + k)

等价。所谓数组和指针的关系，正是基于运算符*和运算符[]的这种等价性提出的。

这启发我们用一个三级指针 int ***a;去模拟三维数组（名）的行为。如同本文第 6 节指出的那样，三维数组名 a（在此为 int***型）指向其首维首元素 a[0]。a[0]又可视作一个二维数组名。为了用指针模拟数组，a[0]应是二级指针即 int**型的。所以可以 new 一个存放 a[0]~a[d1 – 1]的连续空间，即一个 int**型的数组，并用 a 指向数组首元素 a[0]：

int ***a = new int **[d1];。

a[i]是一个二级指针，用来模拟二维数组（名）的特性，所以 a[i]应是一个指向连续空间 a[i][0] ~ a[i][d2 – 1]的首元素 a[i][0]（int*型）的指针。每个 a[i]都是如此，故应为每个 a[i]申请一个规模为 d2 的 int*型数组：

a[i] = new int *[d2];。

a[i][j]是一个一级指针，用来模拟一维数组（名）的特性，所以 a[i][j]应是一个指向连续空间a[i][j][0] ~ a[i][j][d3 – 1]的首元素a[i][j][0]（int 型）的指针。每个 a[i][j]都是如此：

a[i][j] = new int[d3];。

经过上述过程，可在内存的堆区形成如下**存储逻辑**：

a → a[0] ~ a[d1 – 1]，

a[i] → a[i][0] ~ a[i][d2 – 1]，i = 0, 1,…, d1 – 1，

a[i][j] → a[i][j][0] ~ a[i][j][d3 – 1]，j = 0, 1,…, d2 – 1。

小结：在用多级指针模拟多维数组的行为时，应建立这样的理解：n 维数组名用 n 级指针 a 模拟，a 是指向其首维首元素 a[0]的指针；a 的一级引用 a[i]视作一个 n-1 维数组名，用 n-1 级指针模拟。

用多级指针模拟高维数组的本质是**多级引用**，其依据是上文指出的**存储逻辑**。此逻辑可作为本节所述方法的一种形式理解。

（2）动态高维数组的消亡

本文第 7 节指出，操作符 delete[]的操作数是一块连续空间的首地址。根据存储逻辑，消亡用多级指针模拟的动态高维数组时，应从最低维向最高维逐级消亡。首先消亡最低维：

delete[] a[i][j];，

然后消亡次低维：

delete[] a[i];，

直至消亡最高维：

delete[] a;。

（3）顺序存储的动态高维数组

用多级指针虽可模拟高维数组的访问语法，却因各元素存储空间非连续而不适用**基类型代数**，从而无法仅用最低维下标访问数组中的所有元素。事实上，在 C++中可以创建动态规模、顺序存储的高维数组。

本文第 6 节指出，n 维数组的数组名是一个指向基类型 n-1 维数组的指针，故可用语句

```
int(*a)[d2][d3] = (int(*)[d2][d3]) new int[d1 * d2 * d3];
```

申请一个规模为 d1 * d2 * d3 的连续空间，并用类型为 int(*)[d2][d3]的指针变量 a 指向其首元素。因存储空间连续，故成立基类型代数

a[i][j][k] == a[0][0][k + j * d3 + i * d3 * d2]，

所以可仅用最低维下标访问整个数组。

由于存储空间连续，故要消亡顺序存储的动态高维数组 a，只需执行

delete[] a;。

参考资料

[1]https://www.cnblogs.com/chenyangyao/p/5222696.html
[2]https://stackoverflow.com/questions/20312619/cant-declare-dynamic-2d-array-in-c

源代码

```
#include <iostream>
using namespace std;

double &reftest(double input, double **output);
int main(int argc, char const *argv[])
{
    // 测试动态变量/数组的创建和释放，以及返回引用的函数。
    int vi = 5, *iptr = &vi;
    int &ni = *iptr; // legal
    cout << vi << "\t" << *iptr << "\t" << ni << endl;

    double *dptr_main; //不能再定义 double**dptr_main;
    int *p[3][4];
    int(*pp)[3][4];
    double &ref_pa = reftest(5.0, &dptr_main);
    delete[] & ref_pa; // 可以在主调函数中 delete。但若被调函数返回的是动态数组非首元素的引用，则不能 delete。

    short a[2][3][4] = {'\0'};
    for (auto i : {0, 1})
        for (auto j : {0, 1, 2})
```

```
21                 for (auto k : {0, 1, 2, 3})
22                     a[i][j][k] = k + 4 * j + 12 * i;
23 
24         short(*ptr4)[2][3][4] = &a;     // 允许 4 级引用，类似四维数组。
25         short(*ptr3)[3][4] = a;         // 允许 3 级引用，类似三维数组。
26         short(*ptr2)[4] = a[0];         // 允许 2 级引用，类似二维数组。
27         short(*ptr2_te)[4] = &a[0][0]; // 允许 2 级引用，类似二维数组。
28         short *ptr1 = a[0][0];          // 允许 1 级引用，类似一维数组。
29 
30         // char ***ptr = a; // 非法。
31         // 说明以下各量都是同一内存地址的标识符，但这些标识符具有不同的数据类型。
32         cout << "\n&a = " << &a << endl;
33         cout << "&a[0] = " << &a[0] << endl;
34         cout << "&a[0][0] = " << &a[0][0] << endl;
35         cout << "&a[0][0][0] = " << &a[0][0][0] << endl;
36 
37         // 以下标识符指向同一个地址。由于数据类型不同，它们被允许的访问操作也不同。
38         cout << "\na = " << a << endl;
39         cout << "a[0] = " << a[0] << endl;
40         cout << "a[0][0] = " << a[0][0] << endl;
41         cout << "a[0][0][0] = " << a[0][0][0] << endl;
42 
43         // 说明高维数组在内存中连续存储。
44         cout << "\nptr2[5][0] = " << ptr2[5][0] << endl;
45         cout << "ptr2[0][20] = " << ptr2[0][20] << endl;
46         cout << "ptr3[-1][4][16] = " << ptr3[-1][4][16] << endl;
47         cout << "(*(ptr3[-1] + 4))[16] = " << (*(ptr3[-1] + 4))[16] << endl;
48 
49         // 说明数组名的指针特性，不同数据类型允许不同的引用操作。指针的代数运算。
50         cout << "\na[1][2][3] = " << a[1][2][3] << endl;
51         cout << "*(*(*(a + 1) + 2) + 3) = " << *(*(*(a + 1) + 2) + 3) << endl;
52         cout << "*(*(*(*(ptr4 + 2) - 2) - 10) + 39) = " << *(*(*(*(ptr4 + 2) - 2
    ) - 10) + 39) << endl;
53         cout << "*(*(*(ptr3 + 2) - 1) + 3) = " << *(*(*(ptr3 + 2) - 1) + 3) << e
    ndl;
54         cout << "*(*(ptr2 + 4) + 7) = " << *(*(ptr2 + 4) + 7) << endl;
55         cout << "*(ptr1 + 23) = " << *(ptr1 + 23) << endl;
56 
57         // 申请动态高维数组。
58         int d1 = 2, d2 = 3, d3 = 4;
59         // 申请
60         /* 直接 */
61         int(*arr)[d2][d3] = (int(*)[d2][d3]) new int[d1 * d2 * d3];
62         /* 多级 */
```

```
63          int ***arrp = new int **[d1];
64          for (int i = 0; i < d1; i++)
65              arrp[i] = new int *[d2];
66          for (int i = 0; i < d1; i++)
67              for (int j = 0; j < d2; j++)
68                  arrp[i][j] = new int[d3];
69
70          // 初始化
71          /* 直接 */
72          for (int k = 0; k < d1 * d2 * d3; k++)
73              arr[-1][-2][k + d3 * 2 + d2 * d3] = k;
74          /* 多级 */
75          // 不能用基类型代数，而要逐级引用，因为空间非连续。
76          for (int i = 0; i < d1; i++)
77              for (int j = 0; j < d2; j++)
78                  for (int k = 0; k < d3; k++)
79                      arrp[i][j][k] = k + d3 * j + d3 * d2 * i;
80
81          // 输出
82          /* 直接 */
83          for (int i = 0; i < d1 * d2; i++)
84          {
85              cout << endl;
86              for (int j = 0; j < d3; j++)
87                  cout << (*(arr[0] + i))[j] << '\t';
88          }
89          /* 多级 */
90          cout << endl;
91          for (int i = 0; i < d1; i++)
92              for (int j = 0; j < d2; j++)
93                  for (int k = 0; k < d3; k++)
94                  {
95                      if (!(k % d3))
96                          cout << endl;
97                      cout << arrp[i][j][k] << '\t';
98                  }
99
100         // 释放
101         /* 直接 */
102         delete[] arr;
103         /* 多级 */
104         for (int i = 0; i < d1; i++)
105             for (int j = 0; j < d2; j++)
106                 delete[] arrp[i][j];
```

```
107     for (int i = 0; i < d1; i++)
108         delete[] arrp[i];
109     delete[] arrp;
110 
111     while (true)
112     {
113     }
114 
115     return 0;
116 }
117 
118 double &reftest(double input, double **output)
119 {
120     double *dptr = new double(input);
121     double *daptr = new double[3]{input, 2.0};
122     *output = dptr;
123     //delete (daptr + 1); // 不能 delete 非首元素。要么 delete 首元素导致泄露，要么 delete 整个数组。
124     return *(daptr);
125 }
126 
```