**目 录**

**[第一题 1](#_Toc7118)**

[各个指针的解释 1](#_Toc32117)

[赋值语句辨析 1](#_Toc14566)

**[第二题](#_Toc15379)** [4](#_Toc15379)

[(1) 请解释下面语句含义 4](#_Toc19969)

[(2) 请解释下面语句含义 4](#_Toc20998)

[(3) 请解释下面语句含义 4](#_Toc13305)

**[第三题](#_Toc12753)** [5](#_Toc12753)

**[第四题](#_Toc17445)** [7](#_Toc17445)

**[第五题](#_Toc17602)** [8](#_Toc17602)

**[第六题](#_Toc19088)** [9](#_Toc19088)

**[第七题](#_Toc22244)** [10](#_Toc22244)

# 第一题

1. char c = '1', \*pc = 0;
2. const char cc = 'a';
3. const char\* pcc = 0;
4. char\* const cpc = &c;
5. const char\* const cpcc = &cc;
6. char\* const \*pcpc;
7. char\* const\* const pccp = &cpc;

## 各个指针的解释

1.char c = '1', \* pc = 0;

// c 是一个char类型的变量，初始化为 '1'。

// pc 是一个指向字符类型的指针，初始化为0。

2.const char cc = 'a';

// cc 是一个 const char类型的变量，初始化为 'a'。

3.const char\* pcc = 0;

// pcc 是一个指向 const char类型的指针，初始化为0。pcc 所指向的**变量不能修改**，但指针可以改变。

4.char\* const cpc = &c;

// cpc 是一个 const字符指针，指向字符变量 c。**指针不可修改**，但它指向的内容可以改变。

5.const char\* const cpcc = &cc;

// cpcc 是一个指向 const 字符的 const 指针。它指向字符 cc，既不能修改指针本身，也不能修改它所指向的内容。

6.char\* const\* pcpc;

// pcpc 是一个**指向** **char\* const** **指针的指针**。这个指针可以指向其他指针，但该指针指向的指着指针不能修改，该指针指向的指针指向的内容可以修改

7.char\* const\* const pccp = &cpc;

// pccp 是一个 const（不可修改的）指向 const 字符指针的指针，指向指针 cpc。既不能修改 pccp 本身，也不能修改它指向的指针 cpc。

## 赋值语句辨析

**c = cc;** //1

**正确**，cc是const char常量字符，可以赋值给普通字符类型c

cc = c; //2编译报错

编译报错cc是const char类型，不可修改

**pcc = &c;** //3

**正确**,pcc是const char\*指针，可以指向c，但不能通过\*pcc进行修改

**pcc = &cc;** //4

**正确**，指针和变量类型对应

**pc = &c;** //5

**正确**，指针和变量类型对应

pc = &cc; //6编译报错

编译报错,**只读单元的指针(地址)不能赋给指向可写单元的指针变量**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 严重性 | 代码 | 说明 | 项目 | 文件 | 行 | 禁止显示状态 | 详细信息 |
| 错误(活动) | E0513 | 不能将 "const char \*" 类型的值分配到 "char \*" 类型的实体 | Project1 | C:\Users\muke\source\repos\Project1\源.cpp | 23 |  |  |

pc = pcc; //7编译报错

编译报错,只读单元的指针(地址)不能赋给指向可写单元的指针变量

pc = cpcc; //8编译报错

编译报错,只读单元的指针(地址)不能赋给指向可写单元的指针变量

cpcc = pc; //9编译报错

编译报错,cpcc是const指针，不可修改指向内容

**\*pc = "ABCD"[2];** //10

正确，pc是普通指针，通过\*pc将字符串"ABCD"的第三个字符赋值给c

cc = a; //11编译报错

编译报错,cc是const char类型，不可修改

\*cpcc = \*pc; //12编译报错

编译报错,cpcc 是指向 const char 的指针，不能修改。

**pc = \*pcpc;** //13运行报错

**运行报错**,pcpc未初始化

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 严重性 | 代码 | 说明 | 项目 | 文件 | 行 | 禁止显示状态 | 详细信息 |
| 错误 | C4700 | 使用了未初始化的局部变量“pcpc” | Project1 | C:\Users\muke\source\repos\Project1\源.cpp | 30 |  |  |

**\*\*pcpc = \*pc;** //14运行报错

**运行报错**,pcpc未初始化

**\*pc = \*\*pcpc;** //15运行报错

**运行报错**,pcpc未初始化

\*pcc = 'b'; //16编译报错

编译报错,pcc 是指向 const char 的指针，不能修改。

\*pcpc = 'c'; //17编译报错

编译报错,pcpc是指向 char\*const 的指针，char\*const指针自身不能修改。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 严重性 | 代码 | 说明 | 项目 | 文件 | 行 | 禁止显示状态 | 详细信息 |
| 错误(活动) | E0137 | 表达式必须是可修改的左值 | Project1 | C:\Users\muke\source\repos\Project1\源.cpp | 34 |  |  |

\*cpcc = 'd'; //18编译报错

编译报错,cpcc 指向 const char内容，不能修改其指向的内容。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 严重性 | 代码 | 说明 | 项目 | 文件 | 行 | 禁止显示状态 | 详细信息 |
| 错误(活动) | E0137 | 表达式必须是可修改的左值 | Project1 | C:\Users\muke\source\repos\Project1\源.cpp | 36 |  |  |

\*pcpc = pc; //19编译报错

编译报错,\*pcpc是char\*const指针，指针本身不能修改

**pcpc = &cpc;** //20

**正确**,cpc是char\* const指针，其地址可以赋值给pcpc

pccp = &cpc; //21编译报错

编译报错,pccp是const指针，不是可修改的左值

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 严重性 | 代码 | 说明 | 项目 | 文件 | 行 | 禁止显示状态 | 详细信息 |
| 错误(活动) | E0137 | 表达式必须是可修改的左值 | Project1 | C:\Users\muke\source\repos\Project1\源.cpp | 38 |  |  |

故运行正确语句为1、3、4、5、10、20

编译报错为2、6、7、8、9、11、12、16、17、18、19、21

运行报错为13、14、15

# 第二题

请解释语句含义，下面是一个示例：

char (\*pf)(int);

解释：该语句声明一个函数指针pf，指向这样一个函数：参数为int，返回char

可以用typeid操作符打印一个标识符或表达式求值结果的类型，使用该操作符时，必须#include <typeinfo>

std::cout << typeid(pf).name() << std:endl;

## 请解释下面语句含义

1. int\* (\*p)[4];

该语句声明一个指针p，指向一个大小为4的数组，数组的每个元素是一个指向int的指针

## 请解释下面语句含义

1. int\* (\*t)(int) = 0;
2. typedef int\* (\*F)(int);
3. F f = t;

第一条语句定义一个指针t，指向这样一个函数:参数为int，返回int\*

第二条语句定义了一个函数指针类型F，函数类型同上

第三条语句声明一个F类型（函数指针）的变量f，并初始化为t（类型对应）

## 请解释下面语句含义

1. int (\*(\*g)[10])(int, int) = 0;
2. typedef int (\*G[10])(int, int);
3. G\* pg = g;

第一条语句定义一个指针g，指向一个大小为10的数组，数组的每个元素是一个函数指针，指向这样一个函数：参数是两个int，返回值为int

第二条语句定义了一个数组G，G的元素是一个函数指针，函数类型同上

第三天语句定义一个G\*类型的变量pg（即指向G类型的指针），并初始化为g（类型对应）

# 第三题

如果给出下面的定义

1. int i = 1024;
2. const int ci = 1024;
3. double d = 3.14;
4. int f(int x) { return x; }
5. int& g(int& x) { return x; }

则下面的语句哪些编译成功，哪些会编译报错，请说明原因。

1. int& ri\_1 = i; *//1*

正确，ri\_1是int类型的引用，i是int类型的变量

1. int& ri\_2 = ci; *//2*

**错误**，普通引用不能绑定const对象

1. int& ri\_3 = d; *//3*

**错误**，除了二种例外情况，引用类型都要与绑定的对象严格匹配

1. int& ri\_4 = i \* 2; *//4*

**错误**，普通引用只能绑定到左值

1. int& ri\_5; *//5*

**错误**，引用必须在定义时初始化

1. int& ri\_6 = i++; *//6*

**错误**，普通引用只能绑定到左值

1. int& ri\_7 = --i; *//7*

正确，--i是左值，额可以被ri\_7绑定

1. int& ri\_8 = f(i); *//8*

**错误**，普通引用只能绑定到左值

1. int& ri\_9 = g(i); *//9*

正确，g返回的是i的引用，ri\_9可以绑定到左值

1. const int& cri\_1 = i; *//10*

正确，const引用可以绑定非const左值

1. const int& cri\_2 = 1; *//11*

正确，const引用可以绑定右值

1. const int& cri\_3 = d; *//12*

正确，const引用可以绑定不同类型变量

1. const int& cri\_4 = 3.14; *//13*

正确，const引用可以绑定不同类型右值

1. const int& cri\_5 = f(); *//14*

**错误**，函数缺少参数

1. int&& rri\_1 = 1; *//15*

正确，右值引用绑定到右值

1. int&\* pri; *//16*

**错误**，不允许使用指向引用的指针

1. int&& rri\_2 = 1; *//17*
2. int&& rri\_3 = i; *//18*

**错误**，右值引用不能绑定左值

1. int&& rri\_4 = f(i); *//19*

正确，右值引用绑定到右值，函数f返回值是右值

1. int&& rii\_5 = rri\_4; *//20*

**错误**，rri\_4 是一个右值引用，但它本身是一个左值,右值引用不能绑定左值

1. int& ri\_10 = rri\_4; *//21*

正确，普通引用绑定左值rri\_4

故报错语句为2、3、4、5、6、8、14、16、18、20

# 第四题

如果给出下面的定义

1. int a[3] = { 1,2,3 };
2. void f1(int \*p){}
3. void f2(int\*& pr);
4. void f3(int \* const &cpr){}
5. void f4(int(&ra)[3]) {}

请问下面test函数里四个调用语句是否成立，请说明原因

1. void test() {
2. f1(a); *//1*
3. f2(a); *//2*
4. f3(a); *//3*
5. f4(a); *//4*
6. }
7. 成立，数组名a等价为首元素地址即int\*类型，可以被f1接受
8. **不成立**，数组名a被视作常量指针，普通引用不能绑定const类型
9. 成立，与f2相比，改为const引用便能绑定a了
10. 成立，函数接受的参数为int[3]类型的引用，与a的类型匹配

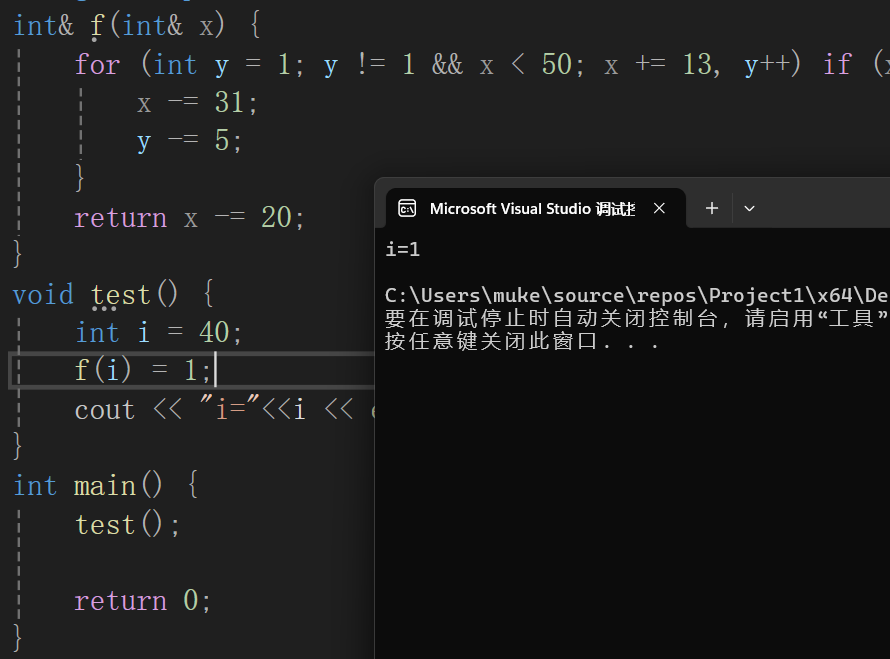
# 第五题

如果给出下面的定义

1. int& f(int &x){
2. for (int y = 1; y != 1 && x < 50; x += 13, y++) if (x > 49) { x -= 31;
3. y -= 5; }
4. return x -= 20;
5. }
6. void test() {
7. int i = 40;
8. f(i) = 1;
9. }

则当执行完test()函数后，i的值是多少？请说明原因。

由于函数返回的是引用，f(i)=1等价于i=1,所以最后i为1



# 第六题

如果给出下面的定义

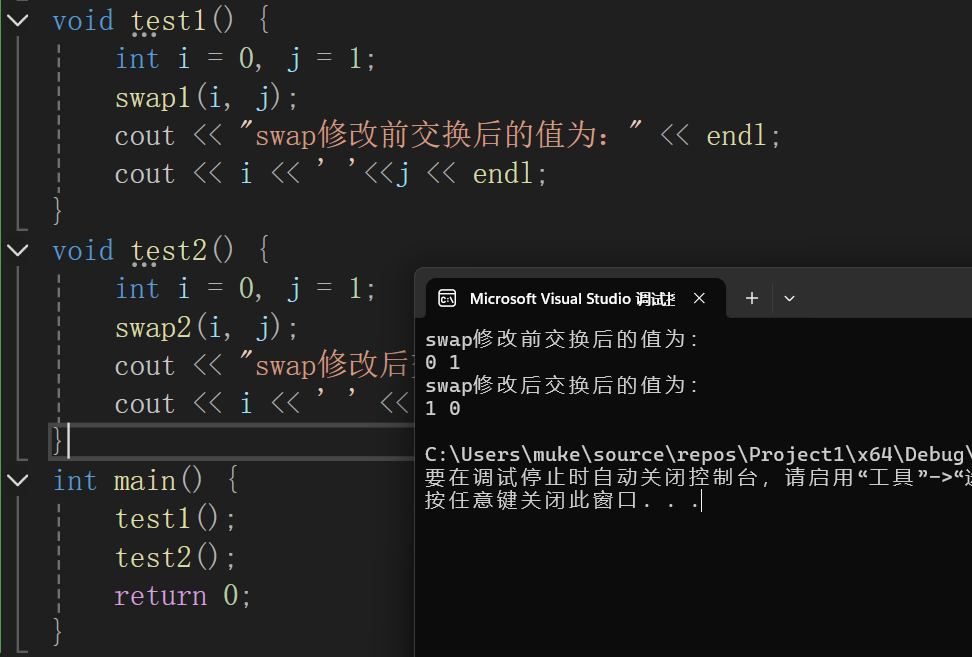
1. void swap(int x, int y) {
2. *//交互xy的值*
3. int temp = x;
4. x = y;
5. y = temp;
6. }

那么下面的函数test()执行完后，实参i和j的值是否互换？请解释原因。

1. void test() {
2. int i = 0, j = 1;
3. swap(i, j);
4. }

不会，函数调用是值传递的，形参的变化不会影响实参的值。正确做法如下

1. void swap(int &x, int &y) {
2. int temp = x;
3. x = y;
4. y = temp;
5. }

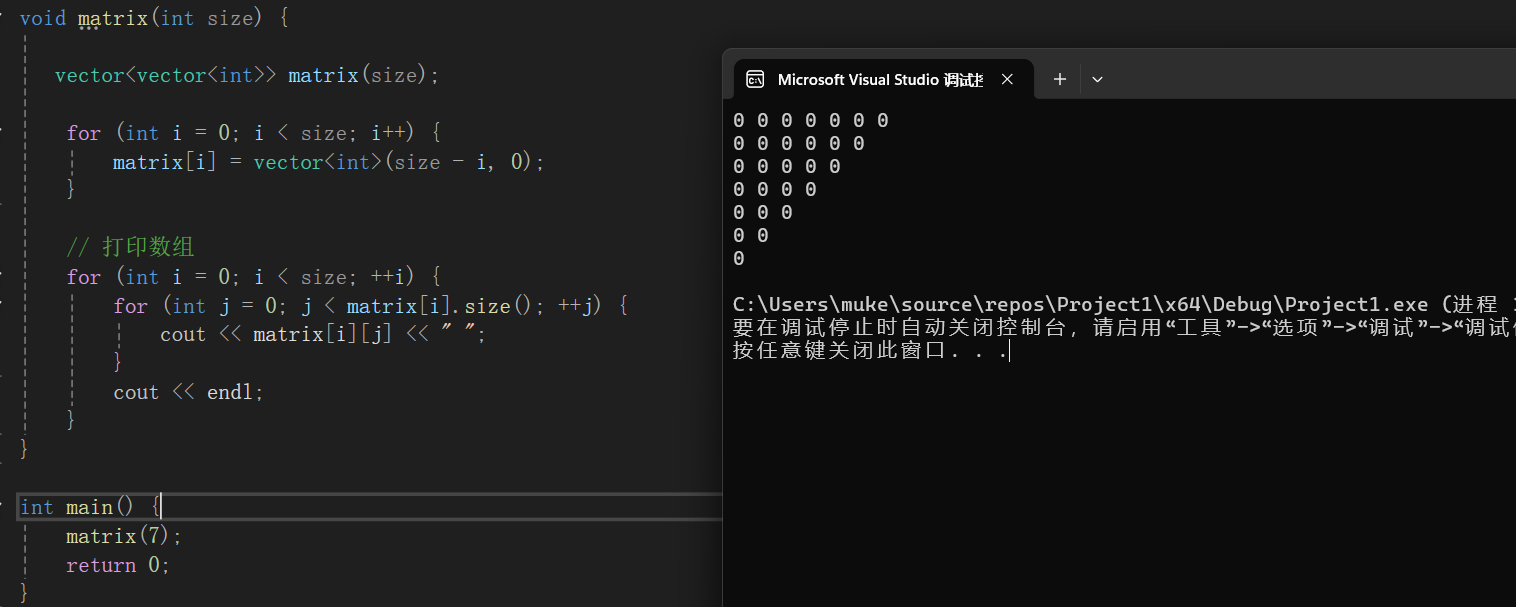


# 第七题

请实现如下函数创建如下的不规则数组，并打印在屏幕上



采用自定义的简单vector类实现动态数组



源码

1. #include"vector.hpp"
2. void matrix(int size) {
4. vector<vector<int>> matrix(size);
5. for (int i = 0; i < size; i++) {
6. matrix[i] = vector<int>(size - i, 0);
7. }
8. *// 打印数组*
9. for (int i = 0; i < size; ++i) {
10. for (int j = 0; j < matrix[i].size(); ++j) {
11. cout << matrix[i][j] << " ";
12. }
13. cout << endl;
14. }
15. }
16. int main() {
17. matrix(7);
18. return 0;
19. }
20. vector.hpp
21. #include<iostream>
22. template <class T>
23. class vector {
24. public:
25. *// 数据*
26. T\* data;
27. *// 大小*
28. int Size;
29. *// 容量*
30. int capacity;
31. *// 构造函数*
32. vector() : Size(0), capacity(1), data(new T[1]) {}
33. *// 析构函数*
34. ~vector() {
35. delete[] data;
36. }
37. *//含参构造函数(初始化大小,元素)*
38. vector(int n, T t) : Size(n), capacity(n), data(new T[n]) {
39. for (int i = 0; i < n; i++) {
40. data[i] = t;
41. }
42. }
43. *// 拷贝构造函数*
44. vector(const vector& v) : Size(v.Size), capacity(v.capacity), data(new T[v.capacity]) {
45. std::copy(v.data, v.data + v.Size, data);
46. }
47. *// 赋值运算符*
48. vector& operator=(const vector& v) {
49. if (this != &v) {
50. T\* new\_data = new T[v.capacity];
51. std::copy(v.data, v.data + v.Size, new\_data);
52. delete[] data;
53. data = new\_data;
54. Size = v.Size;
55. capacity = v.capacity;
56. }
57. return \*this;
58. }
59. *// 添加元素*
60. void push\_back(T t) {
61. if (Size == capacity) {
62. resize(capacity \* 2);
63. }
64. data[Size++] = t; *// 添加元素*
65. }
66. *// 删除元素*
67. void pop\_back() *// 删除最后一个元素*
68. {
69. if (Size > 0) Size--;
70. }
71. *// 返回大小*
72. int size() const {
73. return Size;
74. }
75. *// 返回元素*
76. T& operator[](int index) {
77. if (index < 0 || index >= Size) {
78. throw std::out\_of\_range("Index out of range");*//抛出异常*
79. }
80. return data[index];
81. }
82. *//判断是否为空*
83. bool empty() const
84. {
85. return Size == 0;
86. }
87. void clear() {
88. Size = 0;
89. }
90. *// 调整容量*
91. void resize(int new\_capacity) {
92. T\* new\_data = new T[new\_capacity];*//新建一个数组*
93. std::copy(data, data + Size, new\_data);*//复制数据*
94. delete[] data;*//删除原来的数组*
95. data = new\_data;
96. capacity = new\_capacity;
97. }
98. };