

基础知识3

#字符串数组例题

```
#include <iostream>
#include <cstring>

const int NameLen = 20;

void order(char name[][NameLen], int n) {
    char temp[NameLen];
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (strcmp(name[i], name[j]) > 0) {
                // 交换字符串
                strcpy(temp, name[i]);
                strcpy(name[i], name[j]);
                strcpy(name[j], temp);
            }
        }
    }
}

int find(char name[][NameLen], int n, char searchname[NameLen]) {
    for (int i = 0; i < n; i++) {
        if (strcmp(name[i], searchname) == 0) {
            return i + 1; // 找到, 返回位置
        } else if (strcmp(name[i], searchname) > 0) {
            return 0; // 未找完, 但找不到
        }
    }
    return 0; // 找完, 找不到, 返回
}

int main() {
    char NameTab[][NameLen] = {"GongJing", "LiuNa", "HuangPin", "ZhouZijun", "LianXiaolei",
    "ChenHailing", "CuiPeng", "LiuPing"};
    char searchname[NameLen];
    int n = sizeof(NameTab) / NameLen;

    order(NameTab, n);

    for (int i = 0; i < n; i++) {
        // 输出排序后的
        std::cout << i + 1 << '\t' << NameTab[i] << std::endl;
    }

    std::cout << "Input the searching name: ";
    std::cin >> searchname;
```

```
int result = find(NameTab, n, searchname);

if (result > 0) {
    std::cout << "Position: " << result << std::endl;
} else {
    std::cout << "Not found!" << std::endl;
}

return 0;
}
```

你的感觉是正确的，字符串数组中的第一个索引通常用于表示行数，第二个索引用于表示每行中的字符位置。这是因为在C++中，字符串被存储为字符数组，而每个字符串可以看作是一个字符数组。

例如，如果有一个字符串数组 `char NameTab[3][20];`，你可以将其想象为包含3个字符串的表格，每个字符串最多包含19个字符（20个位置减去一个用于存储字符串结束符`'\0'`）。

cpp

Copy code

```
char NameTab[3][20] = {  
    "Alice", // 第一个字符串  
    "Bob",   // 第二个字符串  
    "Charlie" // 第三个字符串  
};
```

在这个例子中，`NameTab[0]` 表示第一个字符串 "Alice"，`NameTab[1]` 表示第二个字符串 "Bob"，以此类推。而每个字符串内部的长度由第二个索引（例如，`'20'`）来控制。

通常，第一个索引表示行数，第二个索引表示列数。在二维字符数组中，这种惯例有助于更清晰地理解和处理字符串数组。

在 `strcmp(name[i], searchname)` 中，`strcmp` 函数会返回一个整数值，具体含义如下：

- 如果第一个字符串 `name[i]` 按字典顺序在第二个字符串 `searchname` 之前，则 `strcmp` 返回负数。
- 如果两个字符串相等，则 `strcmp` 返回零。
- 如果第一个字符串 `name[i]` 按字典顺序在第二个字符串 `searchname` 之后，则 `strcmp` 返回正数。

所以，`strcmp(name[i], searchname) > 0` 表示 `name[i]` 在字典顺序上排在 `searchname` 之后。

如果你想检查 `name[i]` 是否在字典顺序上排在 `searchname` 之前，可以使用 `strcmp(name[i], searchname) < 0`，这表示 `name[i]` 在字典顺序上排在 `searchname` 之前。

选择 `> 0` 还是 `< 0` 取决于你希望实现的比较逻辑。

算子	短路
&&	如果左侧表达式为真，则组合结果为真（从不计算右侧表达式）。falsefalse
	如果左侧表达式为真，则组合结果为真（从不计算右侧表达式）。true>true

当右边的表达式有副作用（例如更改值）时，这一点最为重要：

```
1 if ( (i<10) && (++i<n) ) { /*...*/ } // note that the condition increments i
```

在这里，组合的条件表达式将增加 1，但前提是左边的条件是，否则，右边的条件 () 永远不会计算。i&&true++i<n

● 显式类型转换运算符

类型转换运算符允许将给定类型的值转换为另一种类型。在 C++ 中有几种方法可以做到这一点。最简单的一种是从 C 语言继承而来的，是在要转换的表达式之前，用括号 () 括起来的新类型：

```
1 int i;
2 float f = 3.14;
3 i = (int) f;
```

前面的代码将浮点数转换为整数值 () ;其余部分丢失。在这里，类型转换运算符是 (int) 。在 C++ 中执行相同操作的另一种方法是在要按类型转换的表达式之前使用函数表示法，并将表达式括在括号之间：3.143(int)

```
1 i = int (f);
```

这两种强制转换类型在 C++ 中都有效。

#stringstream

在C++中，`>>` 是输入流操作符，通常用于从流（如标准输入流`std::cin`或字符串流`std::stringstream`）中读取数据。在你的代码中，`>>` 用于从`std::stringstream`中提取数据并将其存储到变量中。

具体来说，`stringstream` 类支持输入和输出流操作，它可以像`cin`一样用于从字符串中提取数据，或者将数据写入字符串。`>>` 是输入流操作符，用于从流中提取数据。

在这里是你的代码示例：

cpp

Copy code


```
string mystr("1204");
int myint;
stringstream(mystr) >> myint;
```

这段代码的意思是，将字符串"1204"放入`std::stringstream`对象中，然后使用`>>`操作符从流中提取整数，并将其存储在`myint`变量中。最终，`myint`的值将是字符串"1204"转换为整数后的结果。

```

1 // stringstream
2 #include <iostream>
3 #include <string>
4 #include <sstream>
5 using namespace std;
6
7 int main ()
8 {
9     string mystr;
10    float price=0;
11    int quantity=0;
12
13    cout << "Enter price: ";
14    getline (cin,mystr);
15    stringstream(mystr) >> price;
16    cout<<price<<endl;
17    cout << "Enter quantity: ";
18    getline (cin,mystr);
19    stringstream(mystr) >> quantity;
20    cout<<quantity<<endl;
21    cout << "Total price: " << price*quantity << endl;
22    return 0;
23 }

```

Link to this code:  [\[copy\]](#)

options compilation execution

```

Enter price: 11asd
0
Enter quantity: 65
65
Total price: 0

```

由此可知输出的字符串得是数字 所以是数字字符串转化为数字

#基于循环容器for例题字符串

这种for循环遍历了 中的所有元素，其中声明了一些变量，能够获取该范围内元素的值。范围是元素的序列，包括数组、容器以及支持函数和的任何其他类型;本教程中尚未介绍其中的大多数类型，但我们已经熟悉了至少一种范围：字符串，即字符序列。

使用字符串的基于范围的 for 循环示例： `for (declaration : range) statement;`
`range``declaration``begin``end`

```
// range-based for loop
#include <iostream>
#include <string>
using namespace std;

int main ()
{
    string str {"Hello!"};
    for (char c : str)
    {
        cout << "[" << c << "];"
    }
    cout << '\n';
}
```

```
[H][e][l][l][o][!]
```

continue 语句

该语句使程序跳过当前迭代中循环的其余部分，就好像已到达语句块的末尾一样，从而导致它跳转到下一个迭代的开始。例如，让我们跳过倒计时中的数字 5: `continue`

```
// continue loop example
#include <iostream>
using namespace std;

int main ()
{
    for (int n=10; n>0; n--) {
        if (n==5) continue;
        cout << n << ", ";
    }
    cout << "liftoff!\n";
}
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1, liftoff!
```

goto 语句

`goto`允许绝对跳转到程序中的另一个点。这种无条件跳转会忽略嵌套级别，并且不会导致任何自动堆栈展开。因此，这是一个需要谨慎使用的功能，最好是在同一语句块中，尤其是在存在局部变量的情况下。

目标点由标签标识，然后用作语句的参数。标签由后跟冒号 () 的有效标识符组成。

通常被认为是一种低级功能，在通常与 C++ 一起使用的现代高级编程范式中沒有特定的用例。但是，举个例子，这里是使用 `goto` 的倒计时循环的一个版本: `goto:goto`

```
// goto loop example
#include <iostream>
using namespace std;

int main ()
{
    int n=10;
mylabel:
    cout << n << ", ";
    n--;
    if (n>0) goto mylabel;
    cout << "liftoff!\n";
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```

#函数重载

在 C++ 中，如果两个不同的函数参数不同，则可以具有相同的名称，要么是因为它们具有不同数量的参数，要么是因为它们的任何参数属于不同的类型。例如：


```
// overloading functions
#include <iostream>
using namespace std;

int operate (int a, int b)
{
    return (a*b);
}

double operate (double a, double b)
{
    return (a/b);
}

int main ()
{
    int x=5,y=2;
    double n=5.0,m=2.0;
    cout << operate (x,y) << '\n';
    cout << operate (n,m) << '\n';
    return 0;
}
```

```
10
2.5
```

 在 [cpp.sh](#) 上编辑和运行

在此示例中，有两个名为 `operate` 的函数，但其中一个具有两个类型的参数，而另一个具有 `int` 类型的参数。编译器通过检查调用函数时作为参数传递的类型来了解在每种情况下要调用哪一个。如果使用两个参数调用它，则调用具有两个参数的函数，如果使用两个 `s` 调用，则调用具有两个 `s` 的函数。在此示例中，两个函数具有完全不同的行为，版本将其参数相乘，而版本将它们相除。

这通常不是一个好主意。通常，两个同名的函数至少具有相似的行为，但此示例表明，它们完全有可能不这样做。两个重载函数（即两个同名的函数）具有完全不同的定义；出于各种目的，它们是不同的功能，只是碰巧具有相同的名称。

请注意，函数不能仅通过其返回类型进行重载。其至少一个参数必须具有不同的类型。`operateintdoubleintintdoubledoubleintdouble`

#函数模板

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

template <class T>
T sum (T a, T b)
{
    T result;
    result = a + b;
    return result;
}

int main () {
    int i=5, j=6, k;
    double f=2.0, g=0.5, h;
    k=sum<int>(i,j);
    h=sum<double>(f,g);
    cout << k << '\n';
    cout << h << '\n';
    return 0;
}
```

模板是一项功能强大且用途广泛的功能。它们可以有多个模板参数，并且函数仍然可以使用常规的非模板化类型。例如：sumT

```
// function templates
#include <iostream>
using namespace std;

template <class T, class U>
bool are_equal (T a, U b)
{
    return (a==b);
}

int main ()
{
    if (are_equal(10,10.0))
        cout << "x and y are equal\n";
    else
        cout << "x and y are not equal\n";
    return 0;
}
```

x and y are equal

#超长整数相加用string

```
#include <iostream>
#include <string>

using namespace std;

string addLargeIntegers(const string& num1, const string& num2) {
    int carry = 0;
    string result;

    int maxLength = max(num1.size(), num2.size());
    string paddedNum1(maxLength - num1.size(), '0');
    string paddedNum2(maxLength - num2.size(), '0');
    paddedNum1 += num1;
    paddedNum2 += num2;

    for (int i = maxLength - 1; i >= 0; --i) {
        int digit1 = paddedNum1[i] - '0';
        int digit2 = paddedNum2[i] - '0';

        int sum = digit1 + digit2 + carry;
        carry = sum / 10;

        result = static_cast<char>('0' + sum % 10) + result;
    }

    if (carry > 0) {
        result = static_cast<char>('0' + carry) + result;
    }

    return result;
}

int main() {
    string num1 = "123456789012345678901234567890";
```



```
string num2 = "987654321098765432109876543210";

string sum = addLargeIntegers(num1, num2);

cout << "Sum: " << sum << endl;

return 0;
}
```

在这行代码中，`paddedNum1` 是一个 `std::string` 类型的变量，用于存储经过填充零之后的 `num1`。

让我们分解这行代码：

1. `maxLength - num1.size()` 计算了需要填充零的数量。假设 `num1` 的长度为 `len`，那么这部分计算得到的值就是 `maxLength - len`，即需要添加的零的个数。
2. `'0'` 是字符面值，代表数字零。这个字符会被复制多次，以创建一个由零字符组成的字符串。
3. 最终，`paddedNum1` 就是一个由 `'0'` 字符填充的字符串，使其长度达到 `maxLength`。

这个过程是为了确保两个操作数具有相同的长度，以便进行逐位相加。如果两个操作数的长度不同，需要在较短的操作数的前面填充零，使它们的长度相等。这样，就可以从最低位开始逐位相加，处理进位等情况。

在这部分代码中，`for` 循环是用于从最低位到最高位逐位相加的过程。让我们逐行解释这部分代码：

1. `for (int i = maxLength - 1; i >= 0; --i)`：这个 `for` 循环从 `maxLength - 1`（最高位）开始，逐渐减小到 0（最低位），以便逐位相加。
2. `int digit1 = paddedNum1[i] - '0';` 和 `int digit2 = paddedNum2[i] - '0';`：这两行代码将字符数字转换为整数。由于字符数字是以 ASCII 编码表示的，减去字符 `'0'` 可以得到相应的整数值。
3. `int sum = digit1 + digit2 + carry;`：计算当前位的和，包括上一位的进位。
4. `carry = sum / 10;`：更新进位，即下一位需要加上的值。
5. `result = static_cast<char>('0' + sum % 10) + result;`：将当前位的数字转换为字符，然后拼接到结果字符串的前面。这里使用 `static_cast` 进行类型转换。

整个循环会逐位相加，同时考虑进位的影响。最终得到的 `result` 字符串即为两个超长整数相加的结果。

需要注意的是，这里的处理方式是基于十进制的逐位相加，进位等情况。如果要处理更大的整数，可能需要采用更复杂的算法，比如使用数组来存储每一位的数字。

字符数字转换为整数这个规定么

ChatGPT

是的，字符数字到整数的转换是根据 ASCII 码表规定的。在 ASCII 码表中，数字 '0' 到 '9' 的字符编码是依次递增的。因此，可以通过减去字符 '0' 来获得相应的整数值。

具体地说，ASCII 码表中的 '0' 的值是 48，'1' 的值是 49，以此类推。因此，当你减去字符 '0' 时，实际上就是在获取对应数字字符所表示的整数值。

例如，如果 `char c = '5';`，那么 `int digit = c - '0';` 就会得到整数值 5。这种方式在处理字符数字时非常常见。



You

会被加到结果的最左边这个时系统自动的么



ChatGPT

是的, `carry` 的值会在每一次迭代中被加到结果的最左边, 这是因为 `result` 是一个字符串, 而每一次迭代都是将当前计算的个位数加到结果的最左边。

在这段代码中, `result = static_cast<char>('0' + sum % 10) + result;` 将当前计算的个位数转换为字符, 并加到结果的最左边。如果 `carry` 不为 0, 它将在下一次迭代时被加到 `sum` 中。这样, 进位的影响会一直传递到结果的最左边, 确保最终的结果是正确的。



You

result = result+static_cast<char>('0' + sum % 10) 要是这么写的话是不是就位置调换了加在右边是不



#指针字符串数组超长整数求和

```
#include <iostream>
#include <cstring>

char* ladd(const char* s1, const char* s2) {
    int n1 = strlen(s1);
    int n2 = strlen(s2);
    int n = n1 > n2 ? n1 : n2;
    char* res = new char[n + 2];
    char c = 0;

    for (int i = n + 1; i >= 0; --i) {
        res[i] = i > n - n1 ? s1[i - n + n1 - 1] : '0';
    }

    for (int i = n; i >= 0; --i) {
        int tchar = i > n - n2 ? (res[i] - '0' + s2[i - n + n2 - 1] - '0' + c) : (res[i] - '0'
+ c);

        c = (tchar > 9) ? 1 : 0;
        res[i] = (c > 0) ? (tchar - 10 + '0') : (tchar + '0');
    }

    return res;
}

int main() {
    char num1[100], num2[100];
    std::cin >> num1 >> num2;
    char* num = ladd(num1, num2);
    std::cout << "Sum: " << num << std::endl;
```

```

    delete[] num; // 释放内存
    return 0;
}

```

- 1.在这个代码中首先先讲一个数进行补位然后在在补完位置的前提下每一位进行相加
- 2.在但是每一个位置都得从后面开始每一位进行相加，所以是减减，但是有的前提不能忘，所以运用了三目运算，但是需要注意的是，对于每一个循环我们需要注意i的范围所以我们在写是需要认真注意范围
- 3.为了进位所以需要这么写，在第二个for循环里面相当于第一步是将字符转化位数字，方便每一位进行，相加，第二部时将数字转化为字符，好存储在字符串中。

`tchar - 10 + '0')`这个地方为什么要-10啊



ChatGPT

这是因为我们在计算每一位相加时，要考虑进位的情况。在ASCII码中，数字 '0' 对应的是整数值 48，而 '9' 对应的是整数值 57。当我们进行数字相加时，如果结果超过了9，就需要进位。

假设 `tchar` 的值为11，也就是两个数字相加结果为11，我们不能直接将其转换为字符，因为 '1' 的ASCII码值是 49，而我们需要表示的是整数1，所以需要减去一个 '0' 的ASCII码值。这样，`tchar - 10 + '0'` 就能得到正确的字符表示。

这种处理方式确保了在每一位相加时都能正确地将进位考虑进去。



#指针存储函数地址

```

#include <iostream>

int add(int a, int b) {
    return a + b;
}

int sub(int a, int b) {
    return a - b;
}

int mul(int a, int b) {
    return a * b;
}

int divi(int a, int b) {
    if (b == 0) return 0x7fffffff; // 避免除零错误
    else return a / b;
}

int (*menu[])(int, int) = {add, sub, mul, divi};

```

```

int main() {
    int num1, num2, choice;

    std::cout << "Select operator:" << std::endl;
    std::cout << "1: add" << std::endl;
    std::cout << "2: sub" << std::endl;
    std::cout << "3: multiply" << std::endl;
    std::cout << "4: divide" << std::endl;

    std::cin >> choice;

    std::cout << "Input numbers (a, b): ";
    std::cin >> num1 >> num2;


    std::cout << "Result: " << menu[choice - 1](num1, num2) << std::endl;

    return 0;
}

```

没关系，让我解释一下这个代码段。这实际上是一个函数指针数组的声明和初始化，不是枚举类型。

cpp

 Copy code

```
int (*menu[])(int, int) = {add, sub, mul, divi};
```

这一行声明了一个名为 `menu` 的数组，数组的元素是指向函数的指针。每个数组元素对应一个函数，这些函数都接受两个整数参数并返回一个整数。

- `int(*)` 表示这是一个指向函数的指针。
- `int (*)(int, int)` 表示指向接受两个整数参数并返回整数的函数的指针。
- `menu[]` 表示这是一个数组。

所以，`menu` 是一个数组，它存储了指向四个不同函数的指针，分别是 `add`、`sub`、`mul` 和 `divi`。这样，通过选择数组的索引，你可以调用相应位置的函数，实现不同的运算。

这种用法通常用于实现简单的计算器或者菜单选择功能，通过用户输入的选择来调用不同的函数。

● 存储类

具有全局或命名空间范围的变量的存储是在程序的整个持续时间内分配的。这被称为静态存储，它与局部变量（在块中声明的变量）的存储形成鲜明对比。这些使用所谓的自动存储。局部变量的存储仅在声明它们的块期间可用；之后，相同的存储可以用于某些其他函数的局部变量，或者以其他方式使用。

但是，具有静态存储的变量和具有自动存储的变量之间还有另一个实质性的区别：

- 具有静态存储的变量（例如全局变量）如果未显式初始化，则会自动初始化为零。
- 具有自动存储功能的变量（例如局部变量）如果未显式初始化，则未初始化，因此具有未确定的值。

例如：

```
// static vs automatic storage
#include <iostream>
using namespace std;

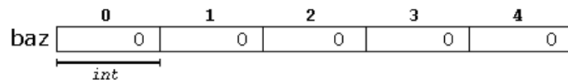
int x;

int main ()
{
    int y;
    cout << x << '\n';
    cout << y << '\n';
    return 0;
}
```

0
4285838

```
1 int baz [5] = { };
```

这将创建一个包含五个值的数组，每个值都使用零值进行初始化：



当为数组提供值的初始化时，C++ 允许将方括号留空的可能性。在这种情况下，编译器将自动假定数组的大小与大括号中包含的值数相匹配：
`int[]{};`

```
1 int foo [] = { 16, 2, 77, 40, 12071 };
```

在此声明之后，数组的长度为 5，因为我们提供了 5 个初始化值。

最后，C++的发展也导致了数组的通用初始化的采用。因此，声明和初始值设定项之间不再需要等号。这两个语句是等效的：`foo int`

```
1 int foo[] = { 10, 20, 30 };
2 int foo[] { 10, 20, 30 };
```

静态数组以及直接在命名空间中声明的数组（在任何函数之外）始终被初始化。如果未指定显式初始值设定项，则所有元素都将默认初始化（对于基本类型，使用零）。

多维数组	伪多维数组
<pre> #define WIDTH 5 #define HEIGHT 3 int jimmy [HEIGHT][WIDTH]; int n,m; int main () { for (n=0; n HEIGHT; n++) for (m=0; m WIDTH; m++) { jimmy[n][m]=(n+1)*(m+1); } } </pre>	<pre> #define WIDTH 5 #define HEIGHT 3 int jimmy [HEIGHT * WIDTH]; int n,m; int main () { for (n=0; n HEIGHT; n++) for (m=0; m WIDTH; m++) { jimmy[n*WIDTH+m]=(n+1)*(m+1); } } </pre>

```
1 char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
2 char myword[] = "Hello";
```

在这两种情况下，字符数组都声明为 6 个类型的元素：组成单词的 5 个字符，加上最后一个空字符（`\0`），它指定序列的结尾，在第二种情况下，当使用双引号（`"`）时，它会自动附加。

请注意，这里我们谈论的是在一个字符数组被声明的那一刻初始化它，而不是关于以后（一旦它们已经被声明）给它们赋值。事实上，由于字符串文字是常规数组，因此它们具有与这些数组相同的限制，并且不能赋值。

表达式（一旦 `myword` 已如上所述声明），例如：`mywordchar"Hello"\0'`

```
1 myword = "Bye";
2 myword[] = "Bye";
```

不会是无效的，就像两者都不是：

```
1 myword = { 'B', 'y', 'e', '\0' };
```

这是因为不能为数组赋值。但请注意，它的每个元素都可以单独分配一个值。例如，这是正确的：

```
1 myword[0] = 'B';
2 myword[1] = 'y';
3 myword[2] = 'e';
```

#数组和指针简单例子

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
    int numbers[5];
    int * p;
    p = numbers; *p = 10;
    p++; *p = 20;
    p = &numbers[2]; *p = 30;
    p = numbers + 3; *p = 40;
    p = numbers; *(p+4) = 50;
    for (int n=0; n<5; n++)
        cout << numbers[n] << " ";
    return 0;
}
```

10, 20, 30, 40, 50,

```
*p++ // same as *(p++): increment pointer, and dereference unincremented address
*++p // same as *(++p): increment pointer, and dereference incremented address
++*p // same as ++(*p): dereference pointer, and increment the value it points to
(*p)++ // dereference pointer, and post-increment the value it points to
```


C++ 提供了两种标准机制来检查分配是否成功：

一种是处理异常。使用此方法时，当分配失败时，将引发类型的异常。异常是一个强大的 C++ 功能，稍后将在这些教程中介绍。但现在，您应该知道，如果抛出此异常并且未由特定处理程序处理，则程序执行将终止。

此异常方法是默认使用的方法，并且是声明中使用的方法，如下所示： `int foofoofoo[0]*foofoo[1]*(foo+1)newnewnewbad_allocnew`

```
1 foo = new int [5]; // if allocation fails, an exception is thrown
```

另一种方法称为 `nothrow`，使用时发生的情况是，当内存分配失败时，返回的指针不是抛出异常或终止程序，而是空指针，程序继续正常执行。

此方法可以通过使用在 header 中声明的特殊对象来指定，作为：`nothrowbad_allocnothrow<new>new`

```
1 foo = new (nothrow) int [5];
```

在这种情况下，如果此内存块的分配失败，则可以通过检查是否为空指针来检测故障：`foo`

```
1 int * foo;
2 foo = new (nothrow) int [5];
3 if (foo == nullptr) {
4     // error assigning memory. Take measures.
5 }
```

此方法生成的代码可能比异常效率低，因为它意味着显式检查每次分配后返回的指针值。因此，通常首选例外机制，至少对于关键分配而言。尽管如此，由于其简单性，大多数即将到来的示例都将使用该机制。`nothrownothrow`

你的声明 `const int *p = {1, 3, 5};` 是有问题的。在 C++ 中，使用大括号初始化列表时，不能直接将其赋值给指针，而应该使用数组。

如果你想要创建一个包含整数的数组，并将指针指向该数组的开头，可以这样做：

cpp

Copy code

```
const int arr[] = {1, 3, 5};
const int *p = arr; // 将指针指向数组的开头
```

或者直接使用初始化列表：

cpp

Copy code

```
const int arr[] = {1, 3, 5};
const int *p = &arr[0]; // 将指针指向数组的开头
```

在这两种情况下，`p` 将成为指向整数数组的常量指针，指向数组的第一个元素。请注意，这里使用 `const` 修饰符表示指针是常量指针，不允许通过指针修改数组的内容。

```

#include <iostream>
using namespace std;
int main()
{
    char* p[] = { "asdcfssad" };
    char s = "asdad";
    char s[] = "asdad";
    char s = a;
    char s2 = 'a';
    char s3[] = { 'a', 's', 'c' };
    char m[10];
    m[10] = "asasddd ";
    for (int i = 0; i < 10; i++)
    {
        m[i] = 'a';
    }
    const char* p11 = "asdsas";
    const char* p34[] = "asd";
    const char* p4 = asas;
    const char* p90[18] = "asdasd";
    int* p[] = { 1, 2, 3 };

    const *int p = {1, 3};
    const int *p={1, 3, 5}
}

```

但它们也可以直接访问。字符串文本是正确数组类型的数组，包含其所有字符以及终止 null 字符，每个元素都属于类型（作为文本，它们永远不能修改）。例如：cout

```
1 const char * foo = "hello";
```

```

char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char myword[] = "Hello";

```

表达式（一旦 myword 已如上所述声明），例如：myword`char`"Hello"``\0``

 12 	 myword = "Bye"; myword[] = "Bye";
------------	--

不会是无效的，就像两者都不是：

 1 	 myword = { 'B', 'y', 'e', '\0' };
-----------	---

这是因为不能为数组赋值。但请注意，它的每个元素都可以单独分配一个值。例如，这是正确的：

 1234 	 myword[0] = 'B'; myword[1] = 'y'; myword[2] = 'e'; myword[3] = '\0';
--------------	---

#字符串中字母重复

这个是错的

```
#include <iostream>
#include<string>
using namespace std;
int main()
{
    string s;
    cin >> s;
    int n = s.size();
    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (s[i] == s[j])
                s[j] = '\t';
        }
    }
    int m = 0;
    cout << s;
    int n2 = s.size();
    for (int i = 0; i < n; i++)
    {
        if (s[i] != '1')
            m++;
    }
    cout << m;
}
```

你的代码看起来是想将字符串中重复出现的字符替换为制表符 ('\t')。然而，你在内层循环中直接替换字符可能会导致问题。在内层循环中将 `s[j]` 替换为 `'\t'`，会破坏原始字符串，可能导致外层循环的索引 `i` 和 `j` 不再指向相邻的字符。

如果你的目标是将字符串中重复的字符替换为制表符，可以使用如下方式：

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

int main() {
```

```
string s;
cin >> s;

int n = s.size();

for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        if (s[i] == s[j] && s[j] != '\t') {
            s[j] = '\t';
        }
    }
}

cout << s << endl;

return 0;
}
```