C++程序设计基础 第四章

前言

本文档由 @ItsJiale 创作，作者博客：https://jiale.domcer.com/，作者依据数学与大数据学院 2024 级大数据教学班的授课重点倾向编写而成。所有内容均为手动逐字录入，其中加上了不少自己的理解与思考，耗费近一周时间精心完成。

此文档旨在助力复习 C++ 程序设计基础，为后续学习数据结构筑牢根基。信计专业的同学，也可参考本文档规划复习内容。需注意，若个人学习过程中存在不同侧重点或对重难点的理解有差异，应以教材内容为准。倘若文档内容存在任何不妥之处，恳请各位读者批评指正。

By：ItsJiale

2025.4.8

第4章 符合类型、string和vector

4.1 引用

引用是指给已经创建的对象重新起一个名字（给变量取别名）。创建引用的时候，编译器只是将这个别名绑定到所引用的对象上，不会把对象的内容复制给引用。

`int &r = a;` （r就是a的别名，且在r的前面必须加上“&”）

注：与下面的情况区分开来

`int *p=&a;`

这时的“&”不是取别名的意思了，而是取地址符

所以当* & 同时出现的时候“&”就是取地址符

（*p 中的“*”是指针的解引用，这个整体代表指针p所指的变量值）

实验6中体现了引用的基本功能

1. `addStudentInfo` 函数参数

```C++
void addStudentInfo(Student students[], int& count) {
    // ...
}
```

- **引用位置**：`int& count` 这里的 `count` 是一个引用参数。
- **作用**：在 `addStudentInfo` 函数中，我们需要更新 `count` 的值，`count` 表示当前存储在 `students` 数组里的学生数量。当成功添加一个新学生信息后，`count` 的值要加 1。要是不使用引用，传递给函数的就只是 `count` 值的副本，函数内对这个副本的修改不会影响到 `main` 函数里的 `count` 变量。而使用引用后，函数内部对 `count` 的修改会直接反映到 `main` 函数中的 `count` 变量上，这样就能准确记录数组中存储的学生数量。

2. `deleteStudentInfo` 函数参数

```C++
void deleteStudentInfo(Student student[], int& count) {
    // ...
}
```

- **引用位置**：同样是 `int& count`，这里的 `count` 也是引用参数。
- **作用**：在 `deleteStudentInfo` 函数中，当成功删除一个学生信息后，需要更新 `count` 的值，也就是让 `count` 减 1。借助引用传递 `count`，函数内部对 `count` 的修改能直接影响到 `main` 函数里的 `count` 变量，确保 `count` 始终能准确表示数组中实际存储的学生数量。

4.1.1 引用const对象

引用一个const对象的语法形式格式如下：

```C++
const int c=0;
const int &r =c; //此处r引用了c
r=1;             //错误 相当于修改const对象的值
```

由上述可知，对于一个const对象，不能修改其修饰的对象，即只能读不能写

4.1.2 auto引用

auto能够根据初始值的类型进行自动类型推导，但有时候，auto并不能正确地推导出相应的类型，例如

`int i = 0, &ri = i;`

`①auto r  =ri;    //r是int类型而不是int类型引用，auto被推导为int`

如果希望定义一个整型引用，则需要显示指出引用类型，如下：

`②auto &r =i;`

①与②相比多了个“&”只有使用&的时候表示的是引用类型，如果没有，则表示的是类型

相同的，利用auto推导一个const引用，也需要明确指出引用类型：

```JavaScript
const int c = 0;
auto &r =c;   //即多了个“&”
```

4.1.3 decltype和引用

decltype能够根据表达式的类型来定义对象，如果表达式是一个对象，delctype会推导出对象的类型，如果表达式是一个引用，decltype也会推导出引用类型：

`int i = 0, &r1=i;  //r1 是 i 的一个别名，对 r1 进行操作就等同于对 i 进行操作。`

`decltype (r1) r2 = i;`

`//decltype(r1): 由于 r1 是一个 int 类型的引用，所以 decltype(r1) 推导出的类型就是 int&（int 类型的引用）。相当于decltype（r1）  == int&`

`//r2: 它是 int 类型的引用，并且被初始化为引用变量 i。这意味着 r2 也是 i 的一个别名，对 r2 进行操作同样等同于对 i 进行操作。`

`decltype (r1+0) r3;`

`//r1 + 0: 这是一个表达式，r1 引用 i，i 是 int 类型，0 也是 int 类型，所以 r1 + 0 的结果是一个 int 类型的值，并非引用。 decltype(r1 + 0): 推导出的类型为 int。`

所以 decltype（非表达式） --> 类型的引用“&”

decltype（表达式）--> 只有类型

4.2 指针

指针是一种数据类型

指针提供了另一种间接访问数据的方式：可以把数据的内存地址存放到专门存放地址的对象里，然后通过这个对象对数据进行访问，这种专门用来存放地址的对象成为指针对象。

4.2.1 指针的定义

通过取地址符（&）获取一个对象的地址，把其存放到一个指针对象中，如下：

C++

```
1   int i =100;
2   int *p=&i;
```

定义一个指向int类型对象的指针p 当把i的地址存放到指针对象p里时，也可以说p指向了i

如果想要访问i的内容，则通过指针的解引用（*）来实现

```C++
cout<<*p <<endl;    //读操作，读取对象i的内容，输出100
*p =10;             //写操作，修改对象i的内容，i的值变成10
```

定义指针对象，需要注意以下几点：

1. 指针的类型必须和所指向的对象的类型一致，void指针和基类指针除外

```C++
int i =10;
double *p =&i;      //错误 p和i的类型不匹配
```

2. 和引用类似，定义多个相同类型的指针对象，每个对象名前面都要加 *

```C++
int i, *p1, *p2;      //i为int类型，p1和p2为指针对象
```

3. 定义指针对象时，如果没有具体的指向对象，则需要用nullpr来初始。如果为初始化的指针在一个语句块内部定义，则它里面存放的是一个随机值

```C++
{
int *p1 = nullptr;    //p1为空指针，没有指向任何对线
int *p2;     //野指针，有潜在风险
}
```

4.2.2 改变指向

利用赋值语句改变一个指针对象的指向，如下：

```C++
int i = 10 , j =100;
int *p1 =&i , *p2 =&j;      //p1指向i,  p2 指向j
p1 = p2;                   //改变p1的指向使其指向j 与 p1= &j等价
p1= nullptr               //改变p1的指向，让p1变成空指针

```

4.2.3 const和指针

和引用一样，可以用const修饰符修饰一个指针对象，使其成为一个指向const对象的指针，即不能通过指针改变指针所指的变量的值，可以改变指针本身的值，如下：

```C++
int j = 0, i =0;
int *const cptr = &i; //*const 是指针常量，定义时初始化，cptr只能指向对象i
cptr = &j;   //错误 不能改变指针cptr的指向
*cptr = 10;  //正确,  可以通过 *cptr 修改其指向的对象i的值
```

```C++
int a=10, b=20;
const int *pa=&a;
//常量指针:不能通过指针改变指针所指变量的值，可以改变指针本身的值
cout<<"pa的值为:"<<pa<<endl;
cout<<"a的地址值为:"<<&a<<endl;
cout<<"b的地址值为:"<<&b<<endl;
```

```C++
int x=10, y=20;
int *const px=&x;
//指针常量：能通过指针改变指针所指变量的值，不可以改变指针本身的值
```

4.2.4 类型推导与指针

如果表达式的值是地址值（右值有&），则auto可以自动推导出指针类型

```cpp
int i =0;
const int ci=10;
auto p =&i;    //p被推导为int * 类型
auto pc =&ci;  //pc被推导const int * 类型，ci的const属性被保留
```

4.2.5 void指针

void指针是一类特殊的指针，它能够指向任何类型的对象。对于这种类型的指针，它只是简单地将对象的地址存储起来。

```cpp
double x = 0;
int i =0 ;
void *p =&x;      //正确   可以存放double类型对象的地址
p=&i;              //正确  也可以存放int 类型对象的地址
```

但不能把void指针随意赋值给一个普通指针 必须确保它们指向相同类型的对象，而且需要进行类型转换，如：

```cpp
double x = 0;
double *pt = &x;
void *p =&x;
pt = static_cast<double *>(pt);    //需要强制类型转换
```

第四行中 利用强制类型转换将p从void*类型转换为double类型，成功转换的前提是p和pt所指向的对象的数据类型必须一致，否则会出现错误

4.2.6 多级指针

多级指针：存放指针的指针

如果把一个指针的地址存放到另一个指针对象里，则形成了指针的指针，即一个二级指针

```C++
int i =1 ;
int *ptr = &i;
int **pptr = &ptr;        //用指针对象ptr的地址初始化pptr
```

因此，可以用三种方法访问对象i：

`cout<<i<<endl;`

`cout<<*ptr<<endl;`

`cout<<**pptr<<endl;`

4.2.7 引用和指针

区别：

1. 定义引用时必须初始化，定义指针不需要初始化；
2. 不存在空引用。引用必须与有效的内存单元关联，指针可以为nullptr
3. 赋值行为不同。对引用赋值修改与相绑定的对象的值，对指针赋值改变其指向的对象

4.2.8 对象指针

对象指针的声明形式

`类名（数据类型） *指针名 = &类对象;`

用于指向这个类的对象（或这种数据类型）

`Base b1;   //Base 是一个类`

`Base *pb = &b1;`

`pb->show( );   //等价于b1.show(  );    调用了Base类里的show () 方法`

4.3 数组

4.3.1 数组的定义和初始化

`int arr[5];    其中arr代表在内存中的首地址    可以使用 int *p =arr;`

arr为数组的名字，[ ]为下标操作符，里面的值代表数组的长度（元素的个数），其值必须大于0的整型常量表达式：

```
int line1[3]= {1,0,1};
```

```
void f(int *p);
```

```
f(line1);
```

注：

1. 若没有显式初始化数组，则采用默认的方式初始化，即：

```
int arr[5]={1,2,3};
```

```
//等价于 arr[5]={1,2,3,0,0};
```

2. 另外编译器可以根据列表中提供的元素个数推断数组的长度

```
int arr[  ] = {1,2,3,4,5};
```

1. 字符数组的初始化

由于字符数组的特殊性，可以采用字符串字面值来初始化，例如：

```
char name[  ] = "Lisha";
```

这种方式等价于：

```
char name[  ]={'L', 'i', 's', 'h', 'a' ,  '\0'}    //自动添加字符串结束符'\0'
```

注意，上面的语句是初始化操作，不是赋值操作。编译器根据字符串常量占用的存储空间为字符数组分配相应大小的内存空间，并依次初始化每个元素（包括结束符'\0'）

警告：不能用一个数组初始化另一个数组，也不能用一个数组赋值给另一个数组

2. 复杂数组的定义

有时候，需要定义一些复杂的数组，比如数组元素的类型是指针或者是引用（指向）其他数组：

```
int arr[5];     //定义一个含有5个int类型元素的数组
```

```
int *arrp[5];  //定义一个含有5个int*类型元素的数组，每个元素都是指针，即：指针数组
```

```
int (*parr)[5] =&arr;   //定义一个指向含有5个int类型元素的数组的指针，即指针指向这个arr数组
```

```
int (&rarr)[5]=arr;  //定义arr的一个引用，即：取别名
```

`int *arrp[5];`

`int * (*parrp)[5]=&arrp;    //指向指针数组的指针`

`int * (&rarrp)[5]=arrp;    //指针数组别名的引用`

4.3.2 访问数组元素

可以通过下标操作符[ ]访问数组元素，例如：

`int arr[5] = {1,2,3,4,5};`

`arr[0] =10;`

`cout<<arr[0]<<endl;`

除了传统的for语句外，C++11还引入了范围for语句，用来遍历数组或其他序列中的所有元素，如下

`for(decl : expr){`

`statement;`

`}    //类似于Java中的增强for循环`

如果想对其进行写操作，则需要将其变量声明为引用

实验4-1

C++

```
for (char &c : trans) {
        if (c >='a' && c < 'z') {
            c = c + 1;
        }
        else if (c == 'z') {
            c = 'a';
        }

    }
```

此处&c就是当前元素的引用

4.3.3 多维数组

1. 多维数组初始化

对于多维数组，依然可以用列表方式来初始化，不过为了增强可读性，通常用花括号把每个元素括起来，例如：

```
int a2d[3][5] = {    //三行五列
{0,1,2,1,4},
{7,5,4,5,7},
{0,8,5,2,9}
};
```

当然，列表中内嵌的花括号可以省略，会自动判断每5个为一组

二维数组的几种初始化方式：

> （1）部分元素显式初始化：比如int a2d[3][5] = {0,1,2}; 这只会显式初始化二维数组，a2d中第一个一维数组里的前三个元素，其他元素自动初始化为0
>
> （2）显式初始化每一个一维数组的第一个元素：像 int a2c[3][5]={{0},{1},{2} }; 这样就分别把三个一维数组的第一个元素初始化为0、1、2
>
> （3）让编译器推断第一维长度：例如 int a2d[ ][5]={0,1,2,1,4,7,5,4,5,7,0,8}; 编译器会根据第二维长度（这里是5）以及给出数据的个数，来推断出第一维的长度。还有 int a2d[ ][5] = { {0},{1},{2} }; 通过花括号的层数（这里是3层）显式指定了第一维长度为3 ***并且强调在多维数组里，只有第一维的长度是可以省略的***

2. 访问多维数组元素

和一维数组一样，依然可以利用下标运算符访问多维数组的元素，而且经常会用嵌套for语句或范围for语句访问数组元素

4.4 指针和数组

4.4.1 指针指向数组

指针和数组的关联非常紧密。一般情况下，编译器对数组的操作都会转换成对指针的操作。例如，数组名通常会被转换成数组第一个元素地址，而且是个右值，例如：

```
int arr[  ] = {1,2,3,4,5};
```

```
int *p =arr ;    //arr被转换成arr[0]的地址
```

上面第二条语句等价于：

```
int *p = &arr[0];   //仅仅是arr[0]
```

等效形式*p即arr[0]，*(p+1)即为a[1]，*（p+i）即为a[i]

所以当利用auto进行类型推导时，得到的是一个指针***而不是数组***

```
auto pa =arr;   //pa为int * 类型，显然是一个指针
```

```
cout<< *pa<<endl;  //输出arr[0]的值为1
```

4.4.2 利用指针访问数组

1.访问一维数组

当一个指针和一个数组关联起来时，就可以通过指针来访问数组元素了

```
int arr[  ] = {1,2,3,4,5};
```

```
int *p =arr;    //p指向数组arr
```

2.访问二维数组

二维数组的数组名是一个指针（二级指针）

与利用指针访问一维数组类似，可以利用指针访问二维数组，例如：

```
int a2d[3][5];
```

```
int (*p2d)=a2d;
```

上面定义的指针p2d指向二维数组a2d的第一个元素，可以将下标运算符作用于指针来访问数组元素，例如

```
p2d[1][1] =1;
```

4.5 string类型

string类型是非常重要的C++标准库类型，它支持变长的字符串和常用的字符串操作。

string是类类型，可以采用如下方法定义一个string类型对象：

```
string str1;              //默认初始化，定义一个空字符串
string str2(str1);        //等价于string str2 =str1; str2是str1的一个副本
string str3="Rat";        //复制初始化
string str4("Rat");       // 直接初始化
string str5(5,'R');       //直接初始化，str5的内容为RRRRR
```

4.5.1 string类型常用操作

和基本内置类型一样，string类也支持输入、输出、比较、相加等操作

1. string对象的输入和输出

先定义一个空的string对象，然后利用cin和cout进行读写操作

```
string s;
cin>>s;
cout<<s;
```

在读取输入流的内容时，如果遇到空白字符（空格符、制表符和回车符），则忽略空白字符并停止字符的读入。如果想要读取空白字符，则可以用getline函数。getline函数读取一整行的输入，直到遇到换行符为止（读入的内容包括换行符），并把所有的内容放到string对象中（不包括换行符）里，例如：

```
getline(cin ,s );
```

2. string对象的大小

可以用string类的成员函数size或者length获取一个string对象里面字符（char类型）的个数，还可以用empty函数测试一个string对象是否为空，例如：

```
string s ;
cin>>s;
```

```
cout<<s.size( )<<endl;                    //输出s里面字符的个数，与s.length( ) 等价
if (!s.empty( ) )
    cout<<s;
```

调用类的一个成员函数，需要在对象的名字后面加上“.”操作符。如果对象是指针类型，则使用->操作符，例如：

```
string *ps = &s;                          //定义一个指针对象指向string对象s
cout<<ps ->size( )<<endl;                 //通过指针调用size成员函数
```

3. string对象的关系运算

string类重载了用于比较字符串的关系运算符，比如 == 、！ =、>=、> 、 < 、 <=等，两个string对象按照字典顺序比较（字典排序靠前的字符小）

```
string s1 ="Hello C++";
string s2 = "Hello";
string s3 ="Hi";
```

一句上述规则，s1大于s2 ，s2小于s3

4. string对象的加法运算

把两个string 对象的内容连接起来，形成一个新的string对象，可以利用加法运算符（+），例如：

```
string s1 = "Hello" , s2 = "C++";
string s3 = s1 + s2;
s1+=s2;
```

s3 和 s1 的内容都是 "Hello C++" string对象还可以和字面值常量相加。例如

```
string s4 = "Hello" +s2;
```

5. 访问单个字符

string类提供了string独享里面单个字符的访问操作，下面语句访问呢一个string对象里的第二个字符：

```
string s ="hello";
s[1]="H";    //对第二个元素进行写操作
cout<<s.at(1)<<endl;
```

其中下标运算符和at函数都要求一个有效的位置值，最小值为0，最大值为对象的长度-1

C++11还支持front和back操作访问第一个和最后一个字符，即

```
cout<<s.front( )<<"  " <<s.back(   )<endl;
```

4.5.2 C风格字符串

C风格字符串不是一种类型，而是以空字符结尾的（'\0'）的字符数组，例如：

```
char cstr[  ] = "Hello";
```

4.6 vector类型

数组存放的元素数目是固定的，但有时候元素的数目是未知的，而且元素的数目还可以能随着程序的运行而发生变化。为了解决这个问题，C++标准库提供了vector类型。vector和数组一样，也是有序元素的集合，但它支持变长操作，对容量的大小可根据需要进行动态调整。

类似于Java中的ArrayList数组

vector是一种容器类型，能够存放类型相同的元素，并且提供了一组常用操作，使用它，需要在程序中包含头文件

```
#include<vector>
```

4.6.1 定义和初始化vector对象

```
vector <int> v1;                                //存放整数的空vector
vector <int> v2 = {0,1,2};                      //v2有三个元素，值分别为0、1和2
vector <int> v3{10};                            //v3可存放10个整数，值为默认值0
vector <int>v4 (10,1);                          //v4存放10个整数1
vector <string> v5 = {"Hi","Lisha","Mandy","Rosita"};
```

```
vector <vector<int>> v6 (10,v2);    //二维
```

4.6.2 vector类型常用操作

1. 添加、删除元素

```
vector <int> vi;
for(int i = 0; i<100; i++){
vi.push_back( i );                  //依次添加100个数  0~99
}
```

> push_back成员函数可以从容器的尾部添加新的元素

类似于的，可以从尾部移除一个元素

> vi.pop_back( );

或者移除所有元素

> vi.clear( );

2. 访问元素

访问容器里面元素，可以用下标运算符或者at成员函数，如输出vi的第二个元素

```
cout<<vi.at(1);   //或者cout<<vi[1];
```

4.6.3 使用迭代器

string和vector支持对元素的随机访问，也就说支持下标运算符，但是C++标准库里还有许多其他的容器类型不支持随机访问。为了统一，C++标准库提出了迭代器。

迭代器的行为类似于指针，支持对数据的间接访问，也支持在元素间移动。

获取容器中一个元素的迭代器（指向该元素的指针），通常借助于容器的成员函数begin和end：

```
vector<int> vi = {0,1,2,3};
auto itb = vi.begin( );        //itb指向vi的第一个元素
auto ite = vi.end( );          //lit指向vi的尾后元素
```

4.7 枚举类型

枚举类型提供了一种简单的方式来使用和维护一组整数值

关键字enum可用来定义不限定作用域的枚举类型，例如：

```
enum color {red,green,blue};
```

其中，color为枚举类型的类型名，它有三个枚举成员，三个枚举成员的作用域与枚举类型本身的作用域相同，因此，如果定义如下新的枚举类型：

```
enum emotion{happy,calm,blue};                  //错误，枚举成员blue已经定义过
```

可以使用限定作用域的方式来解决上面的问题：

```
enum class stopligt {red,green,yellow};
```

其中关键字class还可以用关键词struct来代替，stoplight类型中的三个枚举成员的作用域只在其类型内部，在外部是无法访问的