

第 4 章 复合类型、string 和 vector

目录

① 引用

- 左值引用
- 引用和类型推导
- 右值引用

② 指针

- 指针定义
- const 和指针
- 指针和类型推导
- 指针和引用

③ 数组

- 定义和初始化

④ 指针和数组

- 指针指向数组
- 通过指针访问数组

⑤ string 类型

- string 类型定义和常用操作
- C 风格字符串

⑥ vector 类型

- vector 类型定义和常用操作
- 使用迭代器

⑦ 枚举类型

学习目标

- 理解指针和引用的工作机理；
- 掌握指针、引用和数组的使用方法；
- 理解指针与数组的关系，能够运用指针访问数组元素；
- 学会运用数组、string 和 vector 解决实际问题。

4.1 引用

复合类型

复合类型是指基于其它类型定义的类型，包括指针、引用、数组、函数、类、联合体和枚举类型等

4.1 引用

复合类型

复合类型是指基于其它类型定义的类型，包括指针、引用、数组、函数、类、联合体和枚举类型等

引用

- 为已创建的对象取一个**别名**
- 只将别名绑定到所引用的对象，对象的内容不会复制给引用
- 函数间共享局部对象的重要途径，对于提高程序的效率有重要作用

4.1 引用

复合类型

复合类型是指基于其它类型定义的类型，包括指针、引用、数组、函数、类、联合体和枚举类型等

引用

- 为已创建的对象取一个**别名**
- 只将别名绑定到所引用的对象，对象的内容不会复制给引用
- 函数间共享局部对象的重要途径，对于提高程序的效率有重要作用

说明：

C++11 引入了右值引用，如无提示，引用默认为左值引用

4.1 引用

引用语法格式：

```
int counter = 0;
int &refCnt = counter; //refCnt引用counter对象的内容
int &refCnt2;          //错误：定义引用时必须和一个对象绑定
refCnt = 2;            //修改了counter所在的内存空间的内容
int i = refCnt;        //通过引用读取counter对象的内容，并初始化对象i
```

4.1 引用

引用语法格式：

```
int counter = 0;
int &refCnt = counter; //refCnt引用counter对象的内容
int &refCnt2;          //错误：定义引用时必须和一个对象绑定
refCnt = 2;            //修改了counter所在的内存空间的内容
int i = refCnt;        //通过引用读取counter对象的内容，并初始化对象i
```

定义引用时必须要初始化

```
int &refCnt2;          //错误：定义引用时必须和一个对象绑定
```

4.1 引用

引用语法格式：

```
int counter = 0;
int &refCnt = counter; //refCnt引用counter对象的内容
int &refCnt2;          //错误：定义引用时必须和一个对象绑定
refCnt = 2;            //修改了counter所在的内存空间的内容
int i = refCnt;        //通过引用读取counter对象的内容，并初始化对象i
```

定义引用时必须要初始化

```
int &refCnt2;          //错误：定义引用时必须和一个对象绑定
```

建议：

书写上，把**引用符号与对象名放在一起**，而不是把类型名和引用符号放在一起，这样有助于**提高程序的可读性**。如：

```
int counter = 0;
int& refCnt = counter;
```

4.1 引用

定义引用时，除了需要初始化外，还需要注意以下几点：

1. 定义多个引用时，每个引用必须用 & 标明：

```
int i = 0;  
int &r1 = i, j = 0, &r2 = r1; //r1 和 r2 都是 i 的引用，而 j 是 int 类型
```

4.1 引用

定义引用时，除了需要初始化外，还需要注意以下几点：

1. 定义多个引用时，每个引用必须用 & 标明：

```
int i = 0;  
int &r1 = i, j = 0, &r2 = r1; //r1 和 r2 都是 i 的引用，而 j 是 int 类型
```

2. 只能引用同类型的对象：

```
double d = 0;  
int &r3 = d; //错误： r3 只能引用 int 类型对象
```

4.1 引用

定义引用时，除了需要初始化外，还需要注意以下几点：

1. 定义多个引用时，每个引用必须用 & 标明：

```
int i = 0;  
int &r1 = i, j = 0, &r2 = r1; //r1 和 r2 都是 i 的引用，而 j 是 int 类型
```

2. 只能引用同类型的对象：

```
double d = 0;  
int &r3 = d; //错误：r3 只能引用 int 类型对象
```

3. 引用的对象必须是非 const 左值：

```
int i = 0; const int ci = 0;  
int &r4 = 100, &r5 = i+1, &r6 = ci; //错误：只能引用非 const 左值
```

4.1 引用

练习：

分析以下程序的执行结果：

```
int main(){
    int a(0);
    int &b = a;
    b = 10;
    cout << "a=" << a << endl;
}
```

4.1 引用

练习：

分析以下程序的执行结果：

```
int main(){
    int a(0);
    int &b = a;
    b = 10;
    cout << "a=" << a << endl;
}
```

答案： a=10

4.1 引用—引用 const 对象

引用 const 对象：

```
const int ci = 0;
const int &r1 = ci; //r1 引用const 对象ci
r1 = 1;           //错误：相当于修改const 对象ci 的值
```

4.1 引用—引用 const 对象

引用 const 对象：

```
const int ci = 0;
const int &r1 = ci; //r1 引用const 对象ci
r1 = 1;           //错误：相当于修改const 对象ci 的值
```

const 对象的引用

- 无法通过 const 引用修改其引用对象的内容
- 只能通过 const 引用对绑定的对象进行读操作
- 可用任何类型兼容的对象来初始化 const 引用，例如：
int i = 0;

4.1 引用—引用 const 对象

引用 const 对象：

```
const int ci = 0;
const int &r1 = ci; //r1 引用const 对象ci
r1 = 1;           //错误：相当于修改const 对象ci 的值
```

const 对象的引用

- 无法通过 const 引用修改其引用对象的内容
- 只能通过 const 引用对绑定的对象进行读操作
- 可用任何类型兼容的对象来初始化 const 引用，例如：

```
int i = 0;
const int &r1 = i;      //正确：使用左值对象初始化
```

4.1 引用—引用 const 对象

引用 const 对象：

```
const int ci = 0;
const int &r1 = ci; //r1 引用const 对象ci
r1 = 1;           //错误：相当于修改const 对象ci 的值
```

const 对象的引用

- 无法通过 const 引用修改其引用对象的内容
- 只能通过 const 引用对绑定的对象进行读操作
- 可用任何类型兼容的对象来初始化 const 引用，例如：

```
int i = 0;
const int &r1 = i;      //正确：使用左值对象初始化
const int &r2 = 1;      //正确：使用字面值常量初始化
```

4.1 引用—引用 const 对象

引用 const 对象：

```
const int ci = 0;
const int &r1 = ci; //r1 引用const 对象ci
r1 = 1;           //错误：相当于修改const 对象ci 的值
```

const 对象的引用

- 无法通过 const 引用修改其引用对象的内容
- 只能通过 const 引用对绑定的对象进行读操作
- 可用任何类型兼容的对象来初始化 const 引用，例如：

```
int i = 0;
const int &r1 = i;      //正确：使用左值对象初始化
const int &r2 = 1;      //正确：使用字面值常量初始化
const int &r3 = i+1;    //正确：使用表达式 i+1 的结果初始化
```

4.1 引用—引用 const 对象

引用 const 对象：

```
const int ci = 0;
const int &r1 = ci; //r1 引用const 对象ci
r1 = 1;           //错误：相当于修改const 对象ci 的值
```

const 对象的引用

- 无法通过 const 引用修改其引用对象的内容
- 只能通过 const 引用对绑定的对象进行读操作
- 可用任何类型兼容的对象来初始化 const 引用，例如：

```
int i = 0;
const int &r1 = i;      //正确：使用左值对象初始化
const int &r2 = 1;      //正确：使用字面值常量初始化
const int &r3 = i+1;    //正确：使用表达式 i+1 的结果初始化
const int &r4 = 3.14;   //正确：使用 double 类型数据初始化
```

4.1 引用—auto 和引用

auto 和引用

- auto 不能推导出引用

```
int i = 0, &ri = i;  
auto r = ri; //r 是 int 类型而不是 int 类型引用, auto 被推导为 int
```

4.1 引用—auto 和引用

auto 和引用

- auto 不能推导出引用

```
int i = 0, &ri = i;  
auto r = ri; //r 是 int 类型而不是 int 类型引用, auto 被推导为 int
```

- 定义一个整型引用，需要显式标明引用类型

```
int i = 0;  
auto &r = i; //r是int类型引用
```

4.1 引用—auto 和引用

auto 和引用

- auto 不能推导出引用

```
int i = 0, &ri = i;  
auto r = ri; //r 是 int 类型而不是 int 类型引用, auto 被推导为 int
```

- 定义一个整型引用，需要显式标明引用类型

```
int i = 0;  
auto &r = i; //r是int类型引用
```

- 利用 auto 推导 const 引用也需明确指出引用类型，const 属性被保留：

```
const int ci=0;  
auto &cr = ci; //cr 是 const int 类型引用, auto 被推导为 const int
```

4.1 引用—auto 和引用

以下程序的输出结果是？

1.

```
int a = 0, &b = a;  
auto c = b;  
c = 100;  
cout << "a=" << a << ", " << "b=" << b << endl;
```

2.

```
int a = 0, &b = a;  
auto &c = b;  
c = 100;  
cout << "a=" << a << ", " << "b=" << b << endl;
```

4.1 引用—auto 和引用

以下程序的输出结果是？

1.

```
int a = 0, &b = a;
auto c = b;
c = 100;
cout << "a=" << a << ", " << "b=" << b << endl;
```

2.

```
int a = 0, &b = a;
auto &c = b;
c = 100;
cout << "a=" << a << ", " << "b=" << b << endl;
```

答案：1.a=0,b=0 2.a=100,b=100

4.1 引用—auto 和引用

练习：

3. 以下程序有错误吗？若有，则错在哪里？

```
const int a = 0;
auto &b = a;
b = 100;
cout << "a= " << a << endl;
```

4.1 引用—auto 和引用

练习：

3. 以下程序有错误吗？若有，则错在哪里？

```
const int a = 0;
auto &b = a;
b = 100;
cout << "a= " << a << endl;
```

答案：语句 `b = 100;` 错误。`b` 为 `a` 的 `const` 引用，无法修改 `b` 的值

4.1 引用—decltype 和引用

decltype 和引用

4.1 引用—decltype 和引用

decltype 和引用

- 如果表达式是一个对象， decltype 会推导出对象的类型

```
int i = 0, &r1 = i;  
decltype (r1) r2 = i; //r2 为 int 引用
```

4.1 引用—decltype 和引用

decltype 和引用

- 如果表达式是一个对象， decltype 会**推导出对象的类型**

```
int i = 0, &r1 = i;  
decltype (r1) r2 = i; //r2 为 int 引用
```

- 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;  
decltype (r1 + 0) r3; //r3 为 int 类型
```

4.1 引用—decltype 和引用

decltype 和引用

- 如果表达式是一个对象， decltype 会**推导出对象的类型**

```
int i = 0, &r1 = i;  
decltype (r1) r2 = i; //r2 为 int 引用
```

- 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;  
decltype (r1 + 0) r3; //r3 为 int 类型
```

注意：对象名加上圆括号推导出引用

```
int i = 0;  
decltype ((i)) r2; //错误： r2 为 int 引用，必须初始化
```

右值引用:

绑定到右值的引用，通过 `&&` 来定义。

```
int i = 0;
int &&rr1 = i+1; //正确: rr1 为右值引用, 绑定到一个临时对象
int &&rr2 = i;   //错误: rr2 为右值引用, 不能绑定到左值对象
```

4.1 引用—右值引用

右值引用:

绑定到右值的引用，通过 `&&` 来定义。

```
int i = 0;
int &&rr1 = i+1; //正确: rr1 为右值引用, 绑定到一个临时对象
int &&rr2 = i;   //错误: rr2 为右值引用, 不能绑定到左值对象
```

右值引用功能

- 程序员可以操纵右值对象，尤其是临时对象
- 可以通过右值引用获取即将消亡的右值对象的资源

4.1 引用—右值引用

思考：

以下代码会出现什么情况？为什么？

```
int i = 0;  
int &&rr1 = i+1;  
int &&rr3 = rr1;
```

4.1 引用—右值引用

思考：

以下代码会出现什么情况？为什么？

```
int i = 0;  
int &&rr1 = i+1;  
int &&rr3 = rr1;
```

编译器报错：rr1 为左值，rr3 不能绑定到左值对象

4.1 引用—右值引用

将左值显式转换成右值

```
int &&rr3 = std::move(rr1); //将rr1 转换成右值
```

有时候有些左值对象具有“临时性”，可以像右值一样使用。如只会使用一次的左值对象

4.1 引用—右值引用

将左值显式转换成右值

```
int &&rr3 = std::move(rr1); //将rr1 转换成右值
```

有时候有些左值对象具有“临时性”，可以像右值一样使用。如只会使用一次的左值对象

通用引用

右值引用声明 `&&` 与类型推导结合，变成一种**通用引用类型**，可与右值或左值绑定：

```
int i = 0;
auto &&rr1 = 10; //rr1 为右值引用
auto &&rr2 = i; //rr2 为左值引用
```

4.2 指针

访问数据的方式

- 直接访问：
 - 对象名：本质上是数据所在的内存空间的地址映射

4.2 指针

访问数据的方式

- 直接访问：
 - **对象名**：本质上是数据所在的内存空间的地址映射
- 间接访问：
 - **引用**：通过引用访问已经存在的对象的内容，效果上与使用原对象名对数据的读写相同
 - **指针**：把数据的内存地址存放到专门存放地址的对象中，通过地址对象对数据进行访问

4.2 指针—指针的定义

指针语法格式：

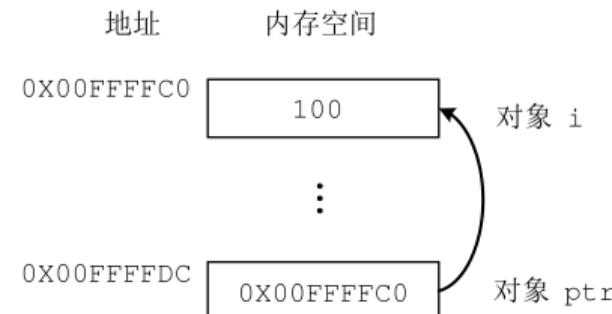
```
int i = 100;  
int *ptr = &i; //用i的地址初始化
```

通过取址符（**&**）获取一个对象的地址，把其存放到一个指针对象

上述代码实现的功能：

定义一个指向 **int** 类型对象的指针对象 **ptr**, **ptr** 中存放的是 **i** 的地址,指向 **i**。

示意图如右图所示



4.2 指针—指针的定义

解引用操作符 (*)：

如果要访问指针指向对象的内容，通过解引用操作符 (*) 来实现：

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

4.2 指针—指针的定义

解引用操作符 (*)：

如果要访问指针指向对象的内容，通过解引用操作符 (*) 来实现：

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

提示：巧读符号

- * 或 & 紧跟类型说明，为指针或引用
- * 或 & 出现在表达式中，为解引用或取址符
例如：

```
int i = 0;  
int *ptr = &i; /* 紧随int，故ptr 为指针；& 在表达式中，故为取址符  
int &ref = *ptr; // & 紧随int，故ref 为引用；* 在表达式中，故为解引用
```

4.2 指针—指针的定义

定义指针对象需注意：

- 指针的类型必须和所指向的对象的类型一致

```
int i = 10;  
double *ptr = &i;      //错误：ptr 和 i 的类型不匹配
```

4.2 指针—指针的定义

定义指针对象需注意：

- 指针的类型必须和所指向的对象的类型一致

```
int i = 10;  
double *ptr = &i; //错误：ptr 和 i 的类型不匹配
```

- 定义多个相同类型的指针对象，每个对象名前面都要加*

```
int i, *ptr1, *ptr2; //i 为 int 类型，ptr1 和 ptr2 为指针对象
```

4.2 指针—指针的定义

定义指针对象需注意：

- 指针的类型必须和所指向的对象的类型一致

```
int i = 10;  
double *ptr = &i; //错误：ptr 和 i 的类型不匹配
```

- 定义多个相同类型的指针对象，每个对象名前面都要加*

```
int i, *ptr1, *ptr2; //i 为 int 类型，ptr1 和 ptr2 为指针对象
```

- 无具体的指向对象时，需用`nullptr`来初始化

```
{  
    int *ptr1 = nullptr; //ptr1 为空指针，没有指向任何对象  
    int *ptr2;           //ptr2 为野指针，有潜在危险  
}
```

4.2 指针—改变指向

改变指针指向

```
int i = 10, j = 100;  
int *ptr1 = &i, *ptr2 = &j; //ptr1 指向i, ptr2 指向j  
ptr1 = ptr2;           //改变ptr1 的指向, 使其指向j, 与ptr1 = &j 等价  
ptr1 = nullptr;        //改变ptr1 的指向, ptr1 变成空指针
```

4.2 指针—const 和指针

const 和指针

- 可以用 `const` 修饰符，使其不能修改所指向对象的值，即指向 `const` 对象的指针。例如：

```
const int ci = 10, cj = 1;  
const int *ptrc = &ci; //ptrc 指向常量ci
```

4.2 指针—const 和指针

const 和指针

- 可以用 `const` 修饰符，使其不能修改所指向对象的值，即指向 `const` 对象的指针。例如：

```
const int ci = 10, cj = 1;  
const int *ptrc = &ci; //ptrc 指向常量ci
```

练习：

下面语句有错误吗？若有，则错在哪里？

```
const int a = 30;  
const int *c = &a;  
*c = 100;
```

4.2 指针—const 和指针

const 和指针

- 可以用 `const` 修饰符，使其不能修改所指向对象的值，即指向 `const` 对象的指针。例如：

```
const int ci = 10, cj = 1;  
const int *ptrc = &ci; //ptrc 指向常量ci
```

练习：

下面语句有错误吗？若有，则错在哪里？

```
const int a = 30;  
const int *c = &a;  
*c = 100;
```

答案：语句 `*c = 100;` 错误，不能修改所指向对象的值

4.2 指针—const 和指针

const 和指针

- `const` 修饰符修饰的指针对象，**可以改变指向**，甚至指向非 `const` 对象。

```
const int ci = 10, cj = 1;
const int *ptrc = &ci; //ptrc 指向常量ci
ptrc = &cj;           //指向另外一个常量
int i = 0;
ptrc = &i;            //还可以指向一个非const 对象
```

4.2 指针—const 和指针

练习：

下面语句有错误吗？若有，则错在哪里？

```
const int a = 30;
const int *c = &a;
int b = 0;
c = &b;
*c = 100;
```

4.2 指针—const 和指针

练习：

下面语句有错误吗？若有，则错在哪里？

```
const int a = 30;
const int *c = &a;
int b = 0;
c = &b;
*c = 100;
```

答案：语句 `*c = 100;` 错误，可以指向非 `const` 对象，但依然不能修改所指向对象的值

4.2 指针—const 和指针

const 和指针

- 一个普通指针，不能指向 const 对象

```
const int ci = 10, cj = 1;  
int *ptr = &ci; //错误：ptr 不能指向常量
```

4.2 指针—const 和指针

const 指针

不允许改变指向的指针，语法格式：

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

4.2 指针—const 和指针

指向 const 对象的 const 指针

```
const int *const cptrc = &ci; //cptrc 是一个指向常量ci 的常量指针
```

第一个 `const` 修饰符表明 `cptrc` 为一个指向 `const` 对象的指针，第二个 `const` 修饰符表明 `cptrc` 不能改变指向

4.2 指针—类型推导和指针

auto 可自动推导出指针类型

如果表达式的值是地址值， auto 可以自动推导出指针类型：

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

4.2 指针—类型推导和指针

提示：

- 符号`&`和`*`从属于对象名，并不是类型名的一部分，`auto`只是一个“占位符”
- 同一条语句中定义多个对象时，对象类型必须一致，例如：

```
int i(0);
auto &ref = i, *ptr = &i; //auto 被推导为int
auto &ref2 = i, ptr2 = &i; //错误：auto 的推导类型不一致
// ref2: auto被推导为int;
// ptr2: auto被推导为 int *
```

4.2 指针—类型推导和指针

练习：

以下程序的输出结果为？

```
int m = 1, n = 2, *p = &m, *r;
auto q = &n;
r = p;
p = q;
q = r;
cout << m << "," << n << "," << *p << "," << *q << endl;
```

4.2 指针—类型推导和指针

练习：

以下程序的输出结果为？

```
int m = 1, n = 2, *p = &m, *r;
auto q = &n;
r = p;
p = q;
q = r;
cout << m << "," << n << "," << *p << "," << *q << endl;
```

答案： 1, 2, 2, 1

利用 decltype 进行指针类型推导

```
int i = 0, *ptr = &i;  
decltype (ptr) ptr2;      //ptr2 为 int *  
decltype (*ptr) refi = i;//正确: refi 为 int &, 必须初始化  
decltype (*ptr+0) j;      //正确: j 为 int 类型
```

4.2 指针—void 指针

void 指针

- 能够指向任何类型的对象

```
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
```

4.2 指针—void 指针

void 指针

- 能够指向任何类型的对象

```
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
```

提示：

将 void 指针赋值给普通指针，必须确保它们指向的对象类型相同，需要进行类型转换
例如：

```
double x = 0, *ptrd = &x;  
void *ptr = &x;  
ptrd = ptr; //报错：不能直接将void指针赋值double *类型的指针。正确语句：  
ptrd = static_cast<double *>(ptr);
```

4.2 指针 多级指针

二级指针

将一个指针对象的地址存放到另一个指针对象中即构成**多级指针**(二级指针)。格式如下：

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

可用三种方式访问对象 i：

```
cout << i << '\t' << *ptr << '\t' << **pptr << endl;
```

4.2 指针 多级指针

二级指针

将一个指针对象的地址存放到另一个指针对象中即构成**多级指针**(二级指针)。格式如下：

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

可用三种方式访问对象 i：

```
cout << i << '\t' << *ptr << '\t' << **pptr << endl;
```

三级指针

```
int ***ppptr = &pptr;  
cout << ***ppptr << endl; //输出1
```

4.2 指针—引用和指针

引用和指针的区别：

- 定义引用时必须初始化，定义指针时不需要初始化：

```
int i = 0, j = 1;  
int &r; // 错误：引用在定义时需要给定初始值  
int *p; // 正确
```

4.2 指针—引用和指针

引用和指针的区别：

- 定义引用时必须初始化，定义指针时不需要初始化：

```
int i = 0, j = 1;  
int &r; // 错误：引用在定义时需要给定初始值  
int *p; // 正确
```

- 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；

4.2 指针—引用和指针

引用和指针的区别：

- 定义引用时必须初始化，定义指针时不需要初始化：

```
int i = 0, j = 1;  
int &r; // 错误：引用在定义时需要给定初始值  
int *p; // 正确
```

- 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；
- 赋值行为不同。对引用赋值修改与其相绑定的对象的值，对指针赋值改变其指向的对象，例如：

```
int i = 0, j = 1, &r = i, *p = &i;  
r = 4; // 修改与 r 相绑定的对象 i 的值  
p = &j; // 修改指针 p 的值，使其指向 j
```

引用和指针

引用的行为实际上类似于**const** 指针的行为，可以把引用看作是支持自动解引用操作的 **const** 指针：

```
int i = 0;
int *const p = &i; //不允许指针p指向其他对象
int &ri = i;      //引用 ri 只能与 i 绑定
```

4.2 指针—引用和指针

引用和指针

引用的行为实际上类似于const指针的行为，可以把引用看作是支持自动解引用操作的const指针：

```
int i = 0;
int *const p = &i; //不允许指针p指向其他对象
int &ri = i;      //引用 ri 只能与 i 绑定
```

建议：

能用引用的地方，不要用指针。

处理批量数据

已知一个班级 30 名学生的英语成绩，求平均分和标准差。

处理批量数据

已知一个班级 30 名学生的英语成绩，求平均分和标准差。

```
int s1,s2,s3,...,s30;  
int sum=0;  
cin>>s1>>s2>>...s30;  
sum=s1+s2+...+s30;  
...
```

4.3 数组—数组的定义和初始化

数组—处理批量数据

数组是由有限个同类型元素组成的有序集合，所有元素顺序存放在一段连续的内存空间中。
如下定义一个存储 5 个整型元素的数组：

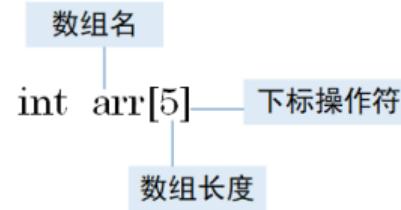
```
int arr[5];
```

4.3 数组—数组的定义和初始化

数组—处理批量数据

数组是由有限个同类型元素组成的有序集合，所有元素顺序存放在一段连续的内存空间中。如下定义一个存储 5 个整型元素的数组：

```
int arr[5];
```



4.3 数组—数组的定义和初始化

数组—处理批量数据

数组是由有限个同类型元素组成的有序集合，所有元素顺序存放在一段连续的内存空间中。如下定义一个存储 5 个整型元素的数组：

```
int arr[5];
```

数组名
int arr[5] 下标操作符
数组长度

地址	内存空间
0X0000FFB0	arr[0]
0X0000FFB4	arr[1]
0X0000FFB8	arr[2]
0X0000FFBC	arr[3]
0X0000FFC0	arr[4]

4.3 数组—数组的定义和初始化

数组长度

数组长度必须为大于 0 的整型常量表达式：

```
unsigned cnt = 10;  
int arr[cnt];           // 错误, cnt 不是常量表达式  
constexpr int sz = 10; // 常量表达式  
int arri[sz];          // 正确: 存放10 个整型数据的数组  
float arrf[10.];        // 错误: 数组长度必须是整型
```

其中，第三行代码中的 `constexpr` 可用 `const` 代替

4.3 数组—数组的定义和初始化

数组的初始化

- 未显式初始化，则用默认的方式初始化。通常采用**列表初始化**显式初始化数组元素：

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

4.3 数组—数组的定义和初始化

数组的初始化

- 未显式初始化，则用默认的方式初始化。通常采用**列表初始化**显式初始化数组元素：

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

- 显式初始化部分数组元素：

```
int arr[5] = {1, 2, 3}; //等价于arr[5] = {1, 2, 3, 0, 0}
```

4.3 数组—数组的定义和初始化

数组的初始化

- 未显式初始化，则用默认的方式初始化。通常采用**列表初始化**显式初始化数组元素：

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

- 显式初始化部分数组元素：

```
int arr[5] = {1, 2, 3}; //等价于arr[5] = {1, 2, 3, 0, 0}
```

- 编译器可以根据列表中提供的元素的个数，推断数组的长度：

```
int arr[] = {1, 2, 3, 4, 5}; //数组arr 的长度为5
```

字符串数组

- 采用字符串字面值来初始化，例如：

```
char name[] = "Lisha"; //自动添加字符串结束符'\0'
```

字符串数组

- 采用字符串字面值来初始化，例如：

```
char name[] = "Lisha"; //自动添加字符串结束符'\0'
```

- 这种方式等价于：

```
char name[] = {'L', 'i', 's', 'h', 'a', '\0'};
```

4.3 数组—数组的定义和初始化

字符串数组

- 采用字符串字面值来初始化，例如：

```
char name[] = "Lisha"; //自动添加字符串结束符'\0'
```

- 这种方式等价于：

```
char name[] = {'L', 'i', 's', 'h', 'a', '\0'};
```

提示：

上面的语句是**初始化操作**，不是赋值操作。

4.3 数组—数组的定义和初始化

注意：数组中的数据不能整体操作

不能用一个数组初始化另外一个数组，也不能用一个数组赋值给另外一个数组

```
char n1[] = "Lisha";
char n2[] = n1; //错误：不能用数组初始化数组
n2 = n1;        //错误：数组不能执行赋值操作
```

4.3 数组—数组的定义和初始化

注意：数组中的数据不能整体操作

不能用一个数组初始化另外一个数组，也不能用一个数组赋值给另外一个数组

```
char n1[] = "Lisha";
char n2[] = n1; //错误：不能用数组初始化数组
n2 = n1;        //错误：数组不能执行赋值操作
```

为什么数组不能执行赋值操作

例如，不能将字符串常量赋值给一个数组

```
name = "Lisha"; //错误：数组不允许赋值操作
```

复杂数组的定义

- 数组元素的类型是指针，即指针数组：

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

```
int *arrp[5]; // 含有5个int*类型元素的数组，每个元素都是指针
```

复杂数组的定义

- 数组元素的类型是指针, 即指针数组:

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

- 数组指向其他数组, 即数组指针:

```
int (*parr)[5] = &arr; // 指向含有5个int类型元素的数组的指针
```

复杂数组的定义

- 数组元素的类型是指针, 即指针数组:

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

- 数组指向其他数组, 即数组指针:

```
int (*parr)[5] = &arr; // 指向含有5个int类型元素的数组的指针
```

- 数组引用其他数组, 即数组的引用:

```
int (&rarr)[5] = arr; // 定义arr的一个引用定义
```

4.3 数组—数组的定义和初始化

定义一个指向 arrp 的指针或引用：

```
int *arrp[5];  
int *(parrp)[5] = &arrp;  
int *(&rarrp)[5] = arrp;
```

parrp 和 rarrp 分别为指向指针数组 arrp 的指针和引用。

4.3 数组—访问数组元素

通过下标操作符 [] 访问数组元素：

```
int arr[5] ={1, 2, 3, 4, 5} ;
arr[0] = 10;           //写操作：修改第一个元素的值
cout << arr[0] << " " << arr[4] << endl; //读操作，输出结果为： 10 5
```

提示：

C++ 不检查下标索引值是否有效，如：

```
cout << arr[5] << endl; //编译器不提示错误，程序可以运行，输出为： -858993460
```

4.3 数组—访问数组元素

范围 for (range for) 语句

语法格式如下：

```
for(decl : expr){  
    statement;  
}
```

- expr 必须是**对象序列**，比如数组、容器（vector）或字符串（string）
- decl 是与序列中数据元素类型相同的对象，通常用 auto 来推导数据元素的类型

4.3 数组—访问数组元素

示例：

```
int arr[5]={1,2,3,4,5}; //定义并初始化一个含有5 个整型数的数组
for(auto i: arr){      //i 为arr 中当前元素的副本
    cout << i << endl; //打印输出当前获取的整数
}
```

4.3 数组—访问数组元素

示例：

```
int arr[5]={1,2,3,4,5}; //定义并初始化一个含有5个整型数的数组
for(auto i: arr){      //i 为arr 中当前元素的副本
    cout << i << endl; //打印输出当前获取的整数
}
```



思考：

上述 range for 语句可以对对象的内容进行修改吗？

```
for(auto i: arr){
    i = 0;
}
```

4.3 数组—访问数组元素

利用 range for 对数组元素进行写操作

需将 decl 声明为引用

```
for(auto &i: arr){ //i为arr中当前元素的引用  
    i = 0;           //写操作：每一个元素设置为0  
}
```

4.3 数组—访问数组元素

例 4.1：

计算一个班级 30 名学生的数学科目的平均成绩和标准差。学生成绩随机生成。

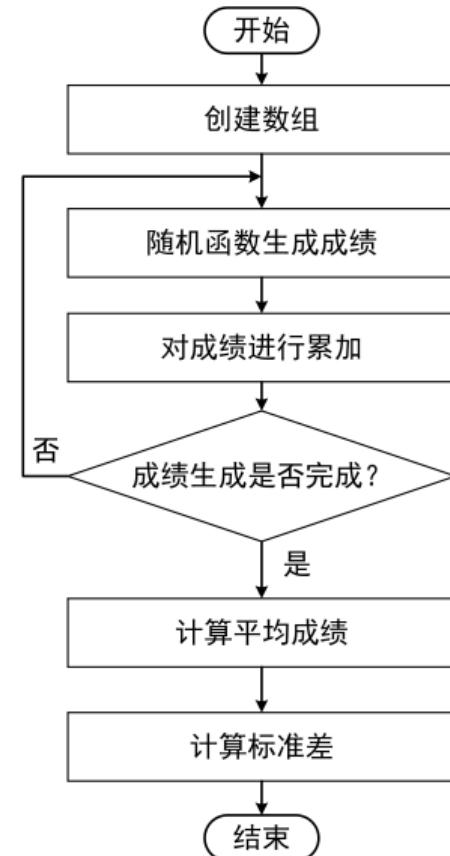
提示：标准差公式： $\sigma = \sqrt{\frac{\sum_{i=1}^N (X_i - \mu)^2}{N}}$

4.3 数组—访问数组元素

例 4.1：

计算一个班级 30 名学生的数学科目的平均成绩和标准差。学生成绩随机生成。

提示：标准差公式： $\sigma = \sqrt{\frac{\sum_{i=1}^N (X_i - \mu)^2}{N}}$



4.3 数组—访问数组元素

代码清单 4.1，例 4.1：

```
1 #include <cstdlib>
2 #include <cmath>
3 #include <iostream>
4 using namespace std;
5 int main() {
6     srand(0); // 使用固定种子，每次运行得到一样的结果，有助于调试
7     constexpr int sz = 30;
8     int score[sz]; // 定义一个数组，存放30个学生的成绩
9     int mean = 0; // 存放平均分数，初始值必须为0
10    for (auto &i:score) // 使用范围for语句访问，注意引用&不能丢
11        i = 50 + rand() % 51; // 成绩随机分布在50到100之间
12        mean += i; // 累加每一个学生成绩到mean里面
13    }
14    mean /= sz; // 计算平均成绩
```

4.3 数组—访问数组元素

代码清单 4.1，例 4.1：

```
14     double dev = 0;
15     for (int i = 0; i < sz; ++i)
16         dev += pow(score[i]- mean,2); //函数pow(x,a)计算x^a
17     }
18     dev = sqrt(dev / sz);
19     cout << "平均成绩: " << mean << " 标准差: " << dev << endl;
20     return 0;
21 }
```

4.3 数组—访问数组元素

例 4.2：

在 8×8 的国际象棋棋盘上摆放八个皇后，使其不能相互攻击，即任意两个皇后不得处在同一行、同一列或者同一对角斜线上。下图所示是一种符合条件的摆放方案。本题计算出一种方案即可。

提示：用回溯法求解。回溯法基本思想：当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，即走不通就退回重新走。

4.3 数组—访问数组元素

例 4.2：

在 8×8 的国际象棋棋盘上摆放八个皇后，使其不能相互攻击，即任意两个皇后不得处在同一行、同一列或者同一对角斜线上。下图所示是一种符合条件的摆放方案。本题计算出一种方案即可。

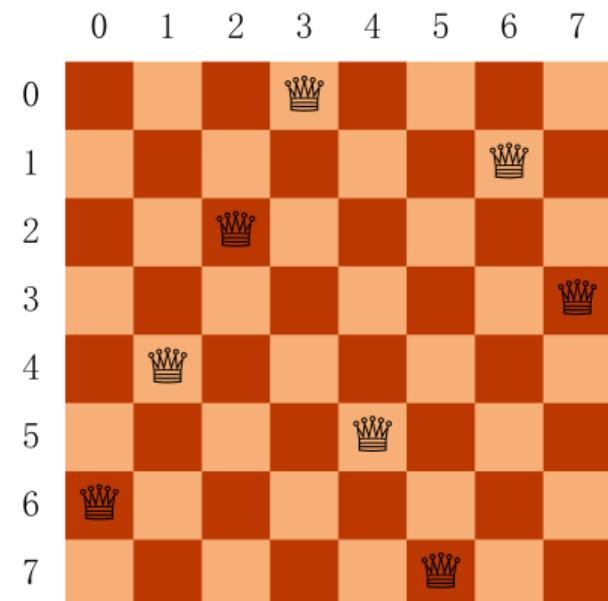
提示：用回溯法求解。回溯法基本思想：当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，即走不通就退回重新走。

4.3 数组—访问数组元素

例 4.2：解的表示

用一个数组 `que[8]` 来存放每一个皇后的位置，
如 `que[0]=3` 代表第 0 行的皇后在第 3 列。

`que[8] = {3, 6, 2, 7, 1, 4, 0, 5};`



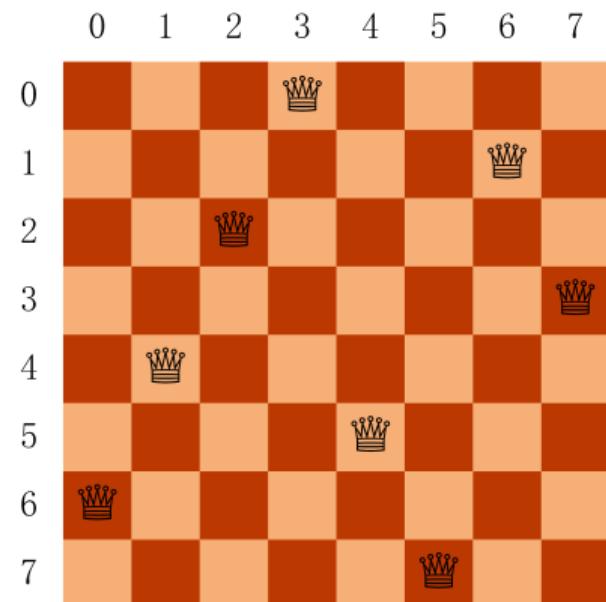
4.3 数组—访问数组元素

例 4.2：解的表示

用一个数组 `que[8]` 来存放每一个皇后的位置，
如 `que[0]=3` 代表第 0 行的皇后在第 3 列。

`que[8] = {3, 6, 2, 7, 1, 4, 0, 5};`

冲突情况：



4.3 数组—访问数组元素

例 4.2：解的表示

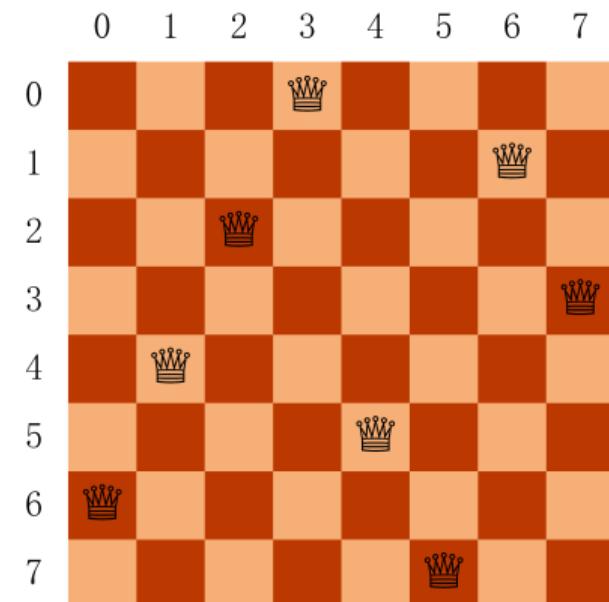
用一个数组 `que[8]` 来存放每一个皇后的位置，
如 `que[0]=3` 代表第 0 行的皇后在第 3 列。

`que[8] = {3, 6, 2, 7, 1, 4, 0, 5};`

冲突情况：

- ① 第 i 行和第 j 行的皇后在同一列：

`que[i] == que[j]`



4.3 数组—访问数组元素

例 4.2：解的表示

用一个数组 `que[8]` 来存放每一个皇后的位置，
如 `que[0]=3` 代表第 0 行的皇后在第 3 列。

`que[8] = {3, 6, 2, 7, 1, 4, 0, 5};`

冲突情况：

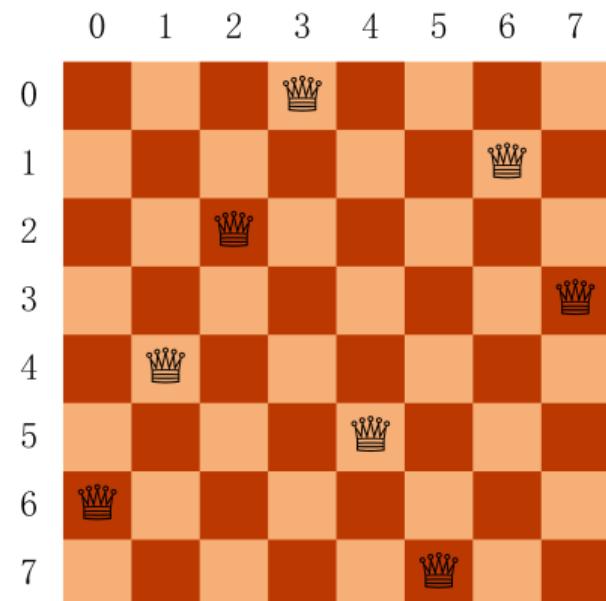
- ① 第 i 行和第 j 行的皇后在同一列：

`que[i] == que[j]`

- ② 第 i 行和第 j 行的皇后在同一对角线上：

$|que[i] - que[j]| == |i - j|$

比如 `que[0]=1, que[1]=2` 表明第 0 行和第 1 行的
两个皇后在同一个对角线上。



```
1 int main() {
2     constexpr int sz = 8;
3     int que[sz] = { 0 };           //每一行皇后都从第0列开始摆放
4     int i = 0;                   //从第0行开始摆放
5     while (i >= 0){
6         int k = 0;
7         while (k<i){           //检查前面所有皇后是否和第i行皇后冲突
8             if(que[k]!=que[i]&&(abs(que[i]-que[k])!=abs(i-k))) ++k;
9             else break;          //第k行和第i行皇后产生冲突，退出，转到第11行
10        }
11        if (k < i) {            //检测到冲突
12            ++que[i];           //处理冲突：移动第i行皇后到当前位置的下一列
13            while (que[i] == sz){ //当前行所有尝试都失败，需要回溯
14                que[i] = 0;        //重置当前行皇后位置
15                --i;              //回溯到上一行
16                if (i < 0) break; //如果回溯到第0行之前，结束运行，转到第19行
17                ++que[i];          //前一行皇后后移一列
18            }
19            continue;           //重新检测是否与前面已安排皇后冲突，转到第5行
20        }else {                //没有检测到冲突，安排下一行皇后
21            ++i;                //移动到下一行
22            if (i < sz) continue; //安排下一行皇后，已安排在第0列，转到第5行
23            for (k = 0; k<sz; ++k) cout << que[k]; //找到一个方案并输出
24            break;               //结束运行
25        }
26    }
27 }
```

多维数组

多维数组指的是数组中的元素类型为数组类型。

- 二维数组

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

- 三维数组

```
int a3d[2][3][5];
```

无论有多少维数，数组元素都存放在一段连续的内存空间。一维数组可以对应数学中的向量，二维数组可对应矩阵。

多维数组初始化

- 用**列表方式初始化**多维数组

如：

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

多维数组初始化

- 用**列表方式初始化**多维数组

如：

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

- 内嵌的花括号可以省略：

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

多维数组初始化

- 显式初始化部分数组元素：

```
int a2d[3][5] = {0, 1, 2};
```

多维数组初始化

- 显式初始化部分数组元素：

```
int a2d[3][5] = {0, 1, 2};
```

- 显式初始化每个一维数组中的第一个元素：

```
int a2d[3][5] = {{0}, {1}, {2}};
```

多维数组初始化

- 显式初始化部分数组元素：

```
int a2d[3][5] = {0, 1, 2};
```

- 显式初始化每个一维数组中的第一个元素：

```
int a2d[3][5] = {{0}, {1}, {2}};
```

- 通过列表元素让编译器自动推断第一维长度：

```
int a2d[] [5] = {0, 1, 2, 1, 4, 7, 5, 4, 5, 7, 0, 8};
```

或者：

```
int a2d[] [5] = {{0}, {1}, {2}};
```

4.3 数组—多维数组

练习：

以下不能对二维数组 `a` 进行正确初始化的是？

- A. `int a[2][3] = { 0 };`
- B. `int a[] [3] = { { 1,2 }, { 0 } };`
- C. `int a[2][3] = { { 1,2 }, { 3,4 }, { 5,6 } };`
- D. `int a[] [3] = { 1,2,3,4,5,6 };`

4.3 数组—多维数组

练习：

以下不能对二维数组 `a` 进行正确初始化的是？

- A. `int a[2][3] = { 0 };`
- B. `int a[] [3] = { { 1,2 }, { 0 } };`
- C. `int a[2][3] = { { 1,2 }, { 3,4 }, { 5,6 } };`
- D. `int a[] [3] = { 1,2,3,4,5,6 };`

答案：C

访问多维数组元素

```
int a2d[2][2] = { {1,2}, {3,4}};
```

- 利用下标运算符访问：

```
int val = a2d[1][1];
cout << "val=" << val << endl;//输出val= 4
```

4.3 数组—多维数组

访问多维数组元素

```
int a2d[2][2] = { {1,2}, {3,4}};
```

- 利用下标运算符访问：

```
int val = a2d[1][1];
cout << "val=" << val << endl; //输出val= 4
```

- 用嵌套的 for 语句访问：

```
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
        cout << "a2d[" << i << "][" << j
            << "]：" << a2d[i][j] << endl;
    }
}
```

4.3 数组—多维数组

访问多维数组元素

```
int a2d[2][2] = { {1,2}, {3,4}};
```

- 利用下标运算符访问：

```
int val = a2d[1][1];
cout << "val=" << val << endl; //输出val= 4
```

- 用嵌套的 for 语句访问：

```
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
        cout << "a2d[" << i << "][" << j
            << "]：" << a2d[i][j] << endl;
    }
}
```

- 用范围 for 语句访问：

```
for (auto &row : a2d) {
    for (auto &col : row)
        cout << col << " ";
    cout << endl;
}
```

4.3 数组—多维数组

练习：

1. 以下程序的输出结果是？

```
int a[] [3] = { { 1,2 },{ 0 } };
for (auto &row : a) {
    for (auto &col : row)
        cout << col << " ";
    cout << endl;
}
```

4.3 数组—多维数组

练习：

1. 以下程序的输出结果是？

```
int a[] [3] = { { 1,2 }, { 0 } };
for (auto &row : a) {
    for (auto &col : row)
        cout << col << " ";
    cout << endl;
}
```

答案：

1 2 0

0 0 0

4.3 数组—多维数组

注意

- 除了最内层的循环外，其它各层循环中必须使用引用，例如：

```
int a2d[2][2] = { { 1,2 },{ 3,4 } };
for (auto &row : a2d) {//row被推导为int(&)[2] 类型
    for (auto col : row) //正确，row为一维数组的引用
        cout << col << " ";
    cout << endl;
}
```

4.3 数组—多维数组

注意

- 除了最内层的循环外，其它各层循环中必须使用引用，例如：

```
int a2d[2][2] = { { 1,2 },{ 3,4 } };
for (auto &row : a2d) { //row被推导为int(&)[2] 类型
    for (auto col : row) //正确，row为一维数组的引用
        cout << col << " ";
    cout << endl;
}
```

- 省去 row 前面的 &，无法通过编译：

```
int a2d[2][2] = { { 1,2 },{ 3,4 } };
for (auto row : a2d) { //row 被推导为int *类型
    for (auto col : row) //错误：row不是列表类型，不能使用范围 for
        cout << col << " ";
    cout << endl;
}
```

4.3 数组—多维数组

例 4.3：打印扫雷游戏的地图

在一个 $n \times n$ 网格化的地图上，随机分布一些地雷，要求在每个没有设置地雷的网格内标记出其相邻区域内地雷的数目，每个网格相邻区域只包括同一行和同一列紧邻的 4 个网格。

	0	1	2	3	4	5	6	7
0	💣	2	0	0	2	💣	2	0
1	💣	💣	1	1	💣	3	💣	2
2	2	💣	1	1	2	2	💣	💣
3	1	1	1	💣	💣	💣	💣	2
4	💣	2	2	💣	3	💣	💣	2
5	💣	💣	💣	💣	1	3	💣	💣
6	💣	3	2	💣	2	💣	💣	2
7	💣	💣	1	1	0	2	💣	1

4.3 数组—多维数组

代码清单 4.3，例 4.3：

```
1 using namespace std;
2 int main() {
3     srand(time(0));
4     constexpr int sz = 8;
5     char map[sz][sz];
6     for (auto &row : map) //每个元素的引用
7         for (auto &col : row) //内嵌数组中每个元素的引用
8             int num = rand() % 100;
9             if (num <= 40) //以0.4的概率设置每个方格的地雷
10                 col = '*';
11             else
12                 col = '0'; //没有地雷的方格初始化为字符0
13 }
14 }
```

4.3 数组—多维数组

代码清单 4.3，例 4.3：

```
15     for (int i = 0; i < sz; ++i) {
16         for (int j = 0; j < sz; ++j) {
17             if (map[i][j] != '*') continue;           //跳过地雷的方格
18             if (i + 1 < sz && map[i + 1][j] != '*') map[i + 1][j] += 1;
19             if (i - 1 >= 0 && map[i - 1][j] != '*') map[i - 1][j] += 1;
20             if (j + 1 < sz && map[i][j + 1] != '*') map[i][j + 1] += 1;
21             if (j - 1 >= 0 && map[i][j - 1] != '*') map[i][j - 1] += 1;
22         }
23     }
24     for (int i = 0; i < sz; ++i) {
25         for (int j = 0; j < sz; ++j) {
26             cout << map[i][j] << " ";
27         }
28         cout << endl;
29     }
30 }
```

4.4 指针和数组—指针指向数组

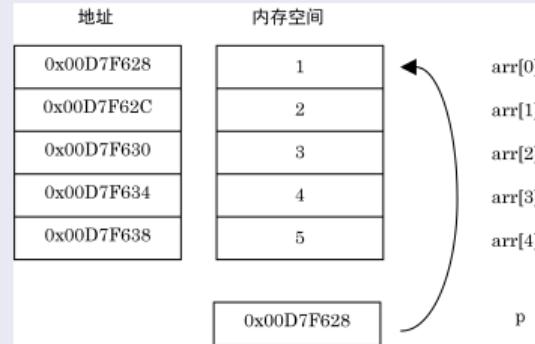
指针和数组的关系

- 数组名被转换成第一个元素的地址

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

第二句等价于：

```
int *p = &arr[0];
```



4.4 指针和数组—指针指向数组

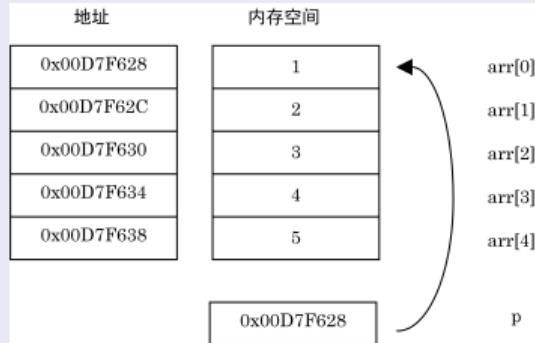
指针和数组的关系

- 数组名被转换成第一个元素的地址

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

第二句等价于：

```
int *p = &arr[0];
```



对于以下程序段：

```
int a[3] = { 5,7,3 }, *p = a;  
cout << a << " " << &a[0] << " " << p << endl;  
cout << a[0] << " " << *p << endl;
```

若数组 a 的第二个元素的地址为 004FFB74，则该程序的输出为？

4.4 指针和数组—指针指向数组

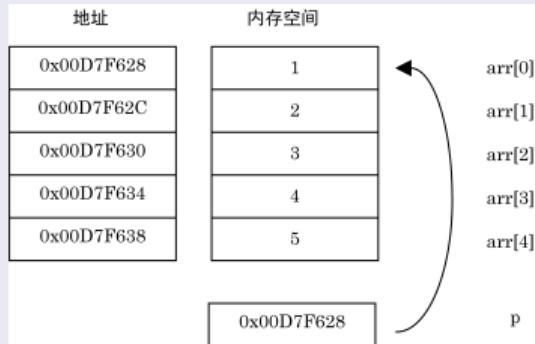
指针和数组的关系

- 数组名被转换成第一个元素的地址

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

第二句等价于：

```
int *p = &arr[0];
```



对于以下程序段：

```
int a[3] = { 5,7,3 }, *p = a;  
cout << a << " " << &a[0] << " " << p << endl;  
cout << a[0] << " " << *p << endl;
```

若数组 a 的第二个元素的地址为 004FFB74，则该程序的输出为？

004FFB70 004FFB70 004FFB70

5 5

4.4 指针和数组—指针指向数组

指针和数组的关系

- 利用auto进行类型推导，得到的是一个指针

```
int arr[] = {1, 2, 3, 4, 5};  
auto pa = arr;           //pa 为 int * 类型，显然是一个指针  
cout << *pa;           //输出arr[0] 的值1
```

4.4 指针和数组—指针指向数组

指针和数组的关系

- 利用 `auto` 进行类型推导，得到的是一个指针

```
int arr[] = {1, 2, 3, 4, 5};  
auto pa = arr;           //pa 为 int * 类型，显然是一个指针  
cout << *pa;            //输出arr[0] 的值1
```

- 利用 `decltype` 定义新数组时，数组名 `arr` 不会转换为指针

```
decltype (arr) ar2; //ar2 为存放5 个整型数的一维数组
```

4.4 指针和数组—指针指向数组

指针和数组的关系

- 可用指针指向多维数组

```
int a2d[3][5];  
int (*p2d)[5] = a2d; //指向a2d 的第一个元素
```

提示：上面的指针定义中，圆括号不能省略，如：

```
int *p2d[5];//p2d是一个含有5个指向整型对象的指针数组
```

4.4 指针和数组—指针指向数组

数组名和指针对象的关系

一个数组名可理解为一个 `const` 指针，但两者并不完全等价
如：

```
int arr[] = { 1, 2, 3, 4, 5 };
int * const p = &arr[0]; //arr 可以理解为const 指针p
cout << sizeof(arr) << " " << sizeof(p);
```

使用运算符 `sizeof` 测试的输出结果为：20 8，分别为一个含有 5 个整型元素的数组和一个指向整型类型的指针对象的大小

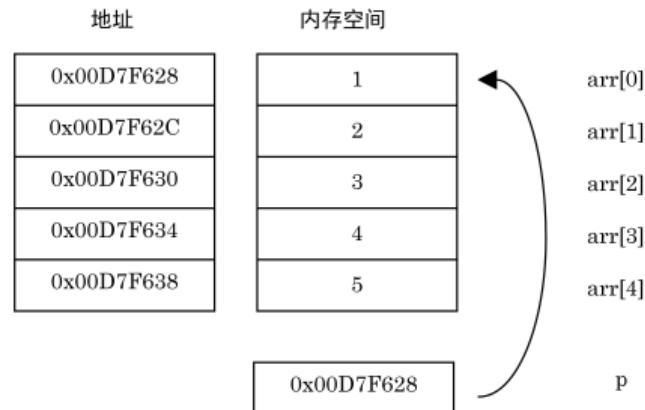
4.4 指针和数组—利用指针访问数组

利用指针访问数组

对于如下数组和指针：

```
int arr[] = {1, 2, 3, 4, 5};  
int *p = arr; //p 指向数组arr
```

示意图如下图所示：



4.4 指针和数组—利用指针访问数组

利用指针访问数组

当指针和数组数组关联时，C++ 支持如下指针运算：

- 指针的移动

```
int arr[] = {1, 2, 3, 4, 5};  
int *p = arr;      //p指向数组arr  
int *p2 = p + 3; //返回p后面第3 个元素的地址，即&arr[3]  
int *p3 = p++;   //p后移一个位置，p3指向p原来的位置&arr[0]  
int *p4 = ++p;   //p继续后移一个位置，p4和p指向同一个位置&arr[2]
```

4.4 指针和数组—利用指针访问数组

利用指针访问数组

当指针和数组数组关联时，C++ 支持如下指针运算：

- 指针的移动

```
int arr[] = {1, 2, 3, 4, 5};  
int *p = arr;      //p指向数组arr  
int *p2 = p + 3; //返回p后面第3 个元素的地址，即&arr[3]  
int *p3 = p++;   //p后移一个位置，p3指向p原来的位置&arr[0]  
int *p4 = ++p;   //p继续后移一个位置，p4和p指向同一个位置&arr[2]
```

- 关系运算

$p == p4$, $p4 > p3$, $p3 \leq p2$ 等都为真。

4.4 指针和数组—利用指针访问数组

利用指针访问数组

当指针和数组数组关联时，C++ 支持如下指针运算：

- 指针的移动

```
int arr[] = {1, 2, 3, 4, 5};  
int *p = arr;      //p指向数组arr  
int *p2 = p + 3; //返回p后面第3 个元素的地址，即&arr[3]  
int *p3 = p++;   //p后移一个位置，p3指向p原来的位置&arr[0]  
int *p4 = ++p;   //p继续后移一个位置，p4和p指向同一个位置&arr[2]
```

- 关系运算

$p == p4$, $p4 > p3$, $p3 \leq p2$ 等都为真。

- 指针相减

两个指针相减的结果为所指向数组元素的位置距离，比如表达式 $p4 - p3$ 的结果为 2。

4.4 指针和数组—利用指针访问数组

关于指针运算需注意：

- **指针在运算的过程中不能越界。**第一个元素到最后一个元素的下一个位置为有效位置，比如：

`p2 = &arr[0]; //正确：指向第一个元素，等价于p2 = arr;`

`p2 = &arr[5]; //正确：指向尾元素后面的一个位置，等价于p2=arr + 5;`

虽然不存在元素 `arr[5]`，但可以计算该位置的地址。

4.4 指针和数组—利用指针访问数组

关于指针运算需注意：

- 指针在运算的过程中不能越界。第一个元素到最后一个元素的下一个位置为有效位置，比如：

`p2 = &arr[0]; //正确：指向第一个元素，等价于p2 = arr;`

`p2 = &arr[5]; //正确：指向尾元素后面的一个位置，等价于p2=arr + 5;`

虽然不存在元素 `arr[5]`，但可以计算该位置的地址。

- 仅当两个指针指向同一个数组时，它们之间的运算才有意义。

4.4 指针和数组—利用指针访问数组

利用指针访问数组：

```
int arr[] = {1, 2, 3, 4, 5};  
int *p = arr;           //p 指向数组 arr  
int val = *(p + 2) + 1; //等价于 int val = arr[2] + 1;  
int val2 = p[2];        //等价于 int val2 = arr[2];
```

4.4 指针和数组—利用指针访问数组

利用指针访问二维数组

```
int a2d[3][5] = { { 0 }, { 1 }, { 2 } };
int(*p2d)[5] = a2d;
cout << p2d[1][1] << endl; //输出为0
```

```
cout <<"a2d: "<<a2d<<"&a2d[0]: "<<&a2d[0]<<"\t p2d: "<<p2d<< endl;
cout <<"a2d[0]: "<<a2d[0]<<"\t &a2d[0][0]: "<<&a2d[0][0]<<"\t *p2d: "<<*p2d<< endl;
cout <<"a2d[0][0]: "<<a2d[0][0]<<"\t\t **p2d: "<<**p2d<< endl;
//输出:
```

```
a2d: 008FFD98          &a2d[0]: 008FFD98      p2d: 008FFD98
a2d[0]: 008FFD98        &a2d[0][0]: 008FFD98  *p2d: 008FFD98
a2d[0][0]: 0             **p2d: 0
```

4.4 指针和数组—利用指针访问数组

利用指针访问二维数组

```
int a2d[3][5] = { { 0 }, { 1 }, { 2 } };
int(*p2d)[5] = a2d;
cout << p2d[1][1] << endl; //输出为0
```

```
cout <<"a2d: "<<a2d<<"&a2d[0]: "<<&a2d[0]<<"\t p2d: "<<p2d<< endl;
cout <<"a2d[0]: "<<a2d[0]<<"\t &a2d[0][0]: "<<&a2d[0][0]<<"\t *p2d: "<<*p2d<< endl;
cout <<"a2d[0][0]: "<<a2d[0][0]<<"\t\t **p2d: "<<**p2d<< endl;
```

//输出：

```
a2d: 008FFD98          &a2d[0]: 008FFD98      p2d: 008FFD98
a2d[0]: 008FFD98        &a2d[0][0]: 008FFD98  *p2d: 008FFD98
a2d[0][0]: 0            **p2d: 0

a2d ⇔ &a2d[0] ⇔ p2d;
```

4.4 指针和数组—利用指针访问数组

利用指针访问二维数组

```
int a2d[3][5] = { { 0 }, { 1 }, { 2 } };
int(*p2d)[5] = a2d;
cout << p2d[1][1] << endl; //输出为0
```

```
cout <<"a2d: "<<a2d<<"&a2d[0]: "<<&a2d[0]<<"\t p2d: "<<p2d<< endl;
cout <<"a2d[0]: "<<a2d[0]<<"\t &a2d[0][0]: "<<&a2d[0][0]<<"\t *p2d: "<<*p2d<< endl;
cout <<"a2d[0][0]: "<<a2d[0][0]<<"\t\t **p2d: "<<**p2d<< endl;
```

//输出：

```
a2d: 008FFD98          &a2d[0]: 008FFD98      p2d: 008FFD98
a2d[0]: 008FFD98        &a2d[0][0]: 008FFD98  *p2d: 008FFD98
a2d[0][0]: 0            **p2d: 0

a2d ⇔ &a2d[0] ⇔ p2d;    a2d[0] ⇔ &a2d[0][0] ⇔ *p2d;
```

4.4 指针和数组—利用指针访问数组

利用指针访问二维数组

```
int a2d[3][5] = { { 0 }, { 1 }, { 2 } };
int(*p2d)[5] = a2d;
cout << p2d[1][1] << endl; //输出为0

cout <<"a2d: "<<a2d<<"&a2d[0]: "<<&a2d[0]<<"\t p2d: "<<p2d<< endl;
cout <<"a2d[0]: "<<a2d[0]<<"\t &a2d[0][0]: "<<&a2d[0][0]<<"\t *p2d: "<<*p2d<< endl;
cout <<"a2d[0][0]: "<<a2d[0][0]<<"\t\t **p2d: "<<**p2d<< endl;
//输出:
a2d: 008FFD98          &a2d[0]: 008FFD98      p2d: 008FFD98
a2d[0]: 008FFD98        &a2d[0][0]: 008FFD98  *p2d: 008FFD98
a2d[0][0]: 0            **p2d: 0

a2d ⇔ &a2d[0] ⇔ p2d;    a2d[0] ⇔ &a2d[0][0] ⇔ *p2d;    a2d[0][0] ⇔ **p2d
```

4.4 指针和数组—利用指针访问数组

利用指针访问二维数组

```
int a2d[3][5] = { { 0 }, { 1 }, { 2 } };
int(*p2d)[5] = a2d;
cout << p2d[1][1] << endl; //输出为0

cout <<"a2d: "<<a2d<<"&a2d[0]: "<<&a2d[0]<<"\t p2d: "<<p2d<< endl;
cout <<"a2d[0]: "<<a2d[0]<<"\t &a2d[0][0]: "<<&a2d[0][0]<<"\t *p2d: "<<*p2d<< endl;
cout <<"a2d[0][0]: "<<a2d[0][0]<<"\t\t **p2d: "<<**p2d<< endl;
//输出:
a2d: 008FFD98          &a2d[0]: 008FFD98      p2d: 008FFD98
a2d[0]: 008FFD98        &a2d[0][0]: 008FFD98  *p2d: 008FFD98
a2d[0][0]: 0            **p2d: 0

a2d ⇔ &a2d[0] ⇔ p2d;    a2d[0] ⇔ &a2d[0][0] ⇔ *p2d;    a2d[0][0] ⇔ **p2d

p2d[1][1] = 1 与下面代码等价:
*(*(p2d + 1) + 1) = 1;      *(*(a2d + 1) + 1) = 1;
*(p2d[1] + 1) = 1;           *(a2d[1] + 1) = 1;
```

4.4 指针和数组—利用指针访问数组

利用 auto 简化代码

```
int a2d[3][5] = { { 1 },{ 1 },{ 1 } };
for (auto p = a2d; p < a2d + 3; ++p) { //p的类型为int (*)[5]
    for (auto q = *p; q < *p + 5; ++q) { //q的类型为int *
        cout << *q << " ";
    }
    cout << endl;
}
```

4.4 指针和数组—利用指针访问数组

利用 auto 简化代码

```
int a2d[3][5] = { { 1 },{ 1 },{ 1 } };
for (auto p = a2d; p < a2d + 3; ++p) { //p的类型为int (*)[5]
    for (auto q = *p; q < *p + 5; ++q) { //q的类型为int *
        cout << *q << " ";
    }
    cout << endl;
}
```

输出为：

```
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
```

4.4 指针和数组—利用指针访问数组

利用数组元素连续的特性遍历数组：

```
int a2d[3][5] = { { 1 }, { 1 }, { 1 } };
for (auto p = &a2d[0][0]; p < a2d[0] + 15; ++p) {
    if ((p - a2d[0]) % 5 == 0) //a2d[0] 等价于 &a2d[0][0]
        cout << endl;           //每打印5个元素后换行
    cout << *p << " ";
}
```

4.4 指针和数组—利用指针访问数组

利用数组元素连续的特性遍历数组：

```
int a2d[3][5] = { { 1 }, { 1 }, { 1 } };
for (auto p = &a2d[0][0]; p < a2d[0] + 15; ++p) {
    if ((p - a2d[0]) % 5 == 0) //a2d[0] 等价于 &a2d[0][0]
        cout << endl;           //每打印5个元素后换行
    cout << *p << " ";
}
```

输出为：

```
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
```

4.5 string 类型

string 类型是 C++ 标准库类型

支持变长的字符串和常用的字符串操作。

using 声明

如下声明来引入单个名字:

```
using std::cin;
```

一劳永逸地引入 std 命名空间内所有的名字:

```
using namespace::std;
```

string 类型—定义和初始化 string 对象

定义 string 类型

```
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的一个拷贝  
string str3 = "Rosita"; //复制初始化  
string str4("Rosita"); //直接初始化  
string str5(5, 'R');   //直接初始化， str5 的内容为 RRRRR
```

4.5 string 类型—string 类型常用操作

string 对象的输入和输出

```
string s;  
cin >> s; //遇到空白字符停止  
cout << s; //输出s的内容
```

利用 `getline` 函数读取空白字符：

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

当输入 "hello C++ " 时（注意里面的空格），`s` 的内容为 "hello C++ "。

4.5 string 类型—string 类型常用操作

string 对象的大小

```
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())           //如果 s 非空，则输出其内容
    cout << s;
```

调用类的成员函数需在对象名后加**.**操作符。指针对象需用**->**操作符：

```
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
```

4.5 string 类型—string 类型常用操作

string 对象的关系运算

比较规则：如果两个 string 对象长度不一样，且较短的 string 对象和较长的对象前面的每个字符都一样，则较短的 string 对象小于较长的 string 对象；否则返回相同位置上第一对不同字符的比较结果（字典排序靠前的字符小），例如：

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

依据上述规则，s1 大于 s2，s2 小于 s3。

string 对象的加法运算

```
string s1 = "Hello ", s2 = "C++";
string s3 = s1 + s2;
s1 += s2; //s3和s1的内容都是"Hello C++"
string s4 = "Hello " + s2; //string对象可与字面值常量相加
```

4.5 string 类型—string 类型常用操作

访问单个字符

- 利用下标运算和 at 函数访问单个字符

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

下标运算和 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。利用 at 成员函数访问是安全的，它会自动检查位置的合法性

- 利用 front 和 back 操作访问第一个和最后一个字符：

```
cout << s.front() << " " << s.back() << endl; //打印输出h o
```

4.5 string 类型—string 类型常用操作

例 4.4：

猜单词游戏，其中一个玩家给定一个单词，让另外一个玩家猜。规则如下：每次猜测单词里面可能的一个字母，并给定猜错的最大次数，比如 3 次。每次猜测要给出相关提示，包括猜测的字母是否正确、字母是否已经猜测过、剩余机会次数以及当前猜测的进度，未猜中的字母用符号 * 代替。如果在给定的最大猜错次数内正确猜出单词的所有字母，则挑战成功，否则失败。失败时给出单词的全部字母。

4.5 string 类型—string 类型常用操作

代码清单 4.4，例 4.4：

```
1 using namespace std;
2 int main() {
3     string target;
4     cout << "请给出一个单词: ";
5     cin >> target;
6     cout << string(100, '\n');           //输出100个换行，用来隐藏输入的单词
7     int length = target.length();
8     string attempt(length, '*'), badchars; //分别记录当前正确和错误的猜测
9     int guesses = 5;                      //最大尝试次数
10    cout << "单词已准备好，它有" << length << "个字母: " << attempt << endl;
```

4.5 string 类型—string 类型常用操作

代码清单 4.4，例 4.4：

```
11 do{  
12     char letter;  
13     cout << "请猜测一个字母：";  
14     cin >> letter;           //badchars或attempt中已有letters  
15     if (badchars.find(letter) != string::npos || attempt.find(letter) != string::npos)  
16         cout << "已经猜过该字母，请重猜" << endl;  
17     continue;                //string::npos匹配失败标志位  
18 }  
19 auto loc = target.find(letter); //使用auto自动推导loc类型  
20 if (loc == string::npos) {  
21     cout << "没有此字母！" << endl;  
22     --guesses;              //允许错误次数-1  
23     badchars += letter;    //猜错的字母放到badchars里  
24 }
```

4.5 string 类型—string 类型常用操作

代码清单 4.4，例 4.4：

```
26     else {
27         cout << "有这个字母，继续加油！" << endl;
28         do {                                //把attempt里面相应的*用猜对的字母替换
29             attempt[loc]=letter;           //如果找到，下一次搜索从loc+1开始
30             loc = target.find(letter, loc + 1);
31         } while (loc != string::npos);
32     }
33     cout << "你猜测的单词：" << attempt << endl;
34     if (attempt != target)
35         cout << "剩余" << guesses << " 次猜错机会" << endl;
36 } while (guesses > 0 && attempt != target);
37 if (guesses > 0)
38     cout << " 成功了，恭喜你！" << endl;
39 else
40     cout<<"对不起，失败了，下次再挑战吧，单词是"<<target<<endl;
41 }
```

4.5 string 类型—c 风格字符串

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 ('\\0') 的字符数组，例如：

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

strlen(s)

返回 s 的长度，不包含结束符 '\\0'

strcmp(s1,s2)

字符串比较函数。和 C++ string 类型对象相比较的规则一样：如果 s1==s2，返回 0；如果 s1>s2，返回正值；如果 s1<s2，返回负值

strcpy(s1,s2)

字符串复制函数。将字符串 s2 复制给 s1，返回 s1

strcat(s1,s2)

字符串链接函数。将字符串 s2 附加到 s1 之后，返回 s1

4.5 string 类型—c 风格字符串

利用指针处理 C 风格字符串

```
char cstr[] = "Hello";
char *ps = cstr;           //指向字符数组cstr
const char *ps2 = "C++";
cout << ps << "," << ps2 << endl; //输出Hello,C++
```

4.5 string 类型—c 风格字符串

使用处理 C 风格字符串的函数时应注意：

- 每个操作对象必须以空字符'\\0' 结尾，否则会产生未定义的行为：

```
char cs[] = {'C', '+', '+'};  
cout << strlen(cs) << endl; //错误：cs没有以空字符结束
```

- 如果对操作对象进行修改，必须要有足够大的内存空间：

```
char small[] = "C++", big[] = "Programming";  
cout << strcpy(small, big) << endl; //错误：small 内存空间不足
```

4.5 string 类型—c 风格字符串

string 类对象使用 c 风格字符串处理函数

需要通过 string 类成员函数 `c_str` 来获取 string 对象存储的字符串的首地址，例如：

```
string str = "hello";
char carr[10];
strcpy(carr, str.c_str());
```

string 的成员函数 `c_str` 返回 `const char*` 类型的指针，确保其指向的对象不被修改。

4.6 vector 类型

vector 类型

- vector 和数组都是有序元素的集合
- vector 支持**变长操作**，容量大小可根据需要动态调整
- vector 是一种容器类型，能够存放类型相同的元素
- 使用 vector 需要在程序中包含 vector 头文件:

```
#include <vector>
```

4.6 vector 类型—定义和初始化 vector 对象

vector<T> v1	定义一个存放 T 类型元素的空对象 v1
vector<T> v2(v1)	复制 v1 里面所有元素到 v2
vector<T> v3(n)	指定初始元素为 n 个
vector<T> v4(n,value)	指定 n 个值为 value 的元素
vector<T> v5={a,b,c,...}	采用列表初始化, v5 的元素个数为列表里面值的个数
vector<T> v6{a,b,c,...}	等价于 v5={a,b,c,...}

定义 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);
```

与数组类似，vector 里面的元素也是顺序存放在连续的内存空间内

4.6 vector 类型—vector 类型常用操作

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
```

vi.push_back() 成员函数称为尾插，即从容器尾端添加元素

vi.pop_back() 成员函数可从容器尾端移除一个元素

vi.clear() 成员函数可移除容器所有元素

访问元素

可用下标运算符或at成员函数访问容器里的元素

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

at 函数会自动检查访问位置的合法性而下标运算符不会

4.6 vector 类型—使用迭代器

借助容器的成员函数获取元素迭代器

迭代器行为与指针类似，支持对数据的间接访问以及在元素间移动

```
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(); //ite 指向 vi 的尾后元素
```

利用解引用获取迭代器指向对象的内容

```
cout << *itb << endl; //输出第一个元素值0
```

4.6 vector 类型—使用迭代器

指向 vector 类型的迭代器支持指针运算

```
for(auto it = vi.begin(); it != vi.end(); ++it){  
    *it *= 2; //每个元素乘 2  
    cout << *it << endl;  
}
```

建议：在 for 循环的结束条件中，为了代码的通用性习惯上为迭代器选择!= 运算，而不是 < 运算

4.6 vector 类型—使用迭代器

使用成员选择运算符

当迭代器指向的元素类型为类类型时，可以用`.或->`运算符进行成员选择，例如：

```
vector<string> vs = {"Hi", "Lisha", "Mandy", "Rosita"};
for(auto it = vs.begin(); it != vs.end(); ++it ){
    cout << (*it).size() << endl; //选择 string 类成员函数 size
}
```

注意：迭代器外面的圆括号不可缺少，否则将表达完全不同的意思：

```
*it.size(); //错误：迭代器 it 没有成员函数 size，相当于 *(it.size());
```

4.6 vector 类型—使用迭代器

使用-> 运算符简化表达

```
for(auto it = vs.begin(); it != vs.end(); ++it )  
    cout << it->size() << endl;  
}
```

4.6 vector 类型—使用迭代器

例 4.5

为例 4.3 中的扫雷游戏地图中的每个方格编号，编号从 0 开始，按照从上到下、从左到右的顺序依次编号。例如，第 0 行第 0 列编号为 0，第 0 行第 1 列编号为 1，依次类推。

要求：若玩家点击的方格无地雷，则找出所有与该方格连通的非雷区。如点击编号为 49 的方格，则下图中编号下划线的方格都是与之相邻的非雷区

提示：采用宽度优先搜索（breadth-first search）策略来求解。

	0	1	2	3	4	5	6	7
0	💣	💣	💣	💣	4	5	6	7
1	8	💣	10	💣	💣	13	💣	💣
2	💣	17	18	19	20	21	22	💣
3	24	💣	26	💣	💣	29	💣	31
4	💣	<u>33</u>	💣	💣	💣	38	39	
5	💣	<u>41</u>	💣	43	44	45	💣	💣
6	<u>48</u>	<u>49</u>	💣	💣	52	💣	54	💣
7	💣	<u>57</u>	💣	59	💣	💣	💣	💣

图：宽度优先

4.6 vector 类型—使用迭代器

例 4.5

为例 4.3 中的扫雷游戏地图中的每个方格编号，编号从 0 开始，按照从上到下、从左到右的顺序依次编号。例如，第 0 行第 0 列编号为 0，第 0 行第 1 列编号为 1，依次类推。

要求：若玩家点击的方格无地雷，则找出所有与该方格连通的非雷区。如点击编号为 49 的方格，则下图中编号下划线的方格都是与之相邻的非雷区

提示：采用宽度优先搜索（breadth-first search）策略来求解。

	0	1	2	3	4	5	6	7
0	💣	💣	💣	💣	4	5	6	7
1	8	💣	10	💣	💣	13	💣	💣
2	💣	17	18	19	20	21	22	💣
3	24	💣	26	💣	💣	29	💣	31
4	💣	<u>33</u>	💣	💣	💣	38	39	
5	💣	<u>41</u>	💣	43	44	45	💣	💣
6	<u>48</u>	<u>49</u>	💣	💣	52	💣	54	💣
7	💣	<u>57</u>	💣	59	💣	💣	💣	💣

步骤	添加	移除	nobomb	result
1	41, 48, 57	49	41, 48, 57	49
2	33	41	48, 57, 33	49, 41
3		48	57, 33	49, 41, 48
4		57	33	49, 41, 48, 57
5		33		49, 41, 48, 57, 33

图：宽度优先

4.6 vector 类型—使用迭代器

代码清单 4.5，例 4.5：

```
1 using namespace std;
2 int main() {
3     srand(0);
4     constexpr int sz = 8;
5     char map[sz][sz];
6     for (auto &row : map)
7         for(auto &col:row)          //以0.5的概率设置地雷，使用条件表达式简化代码
8             col = rand() % 100 < 50 ? '*' : '0';
9     for (int i = 0; i < sz; ++i) { //打印地图，函数setw设置打印字符的宽度（见10.2.2节）
10        for (int j = 0; j < sz; ++j) {
11            if (map[i][j] == '*') cout << setw(3)<< "*";
12            else cout << setw(3) << i*sz+j;
13        }
14        cout << endl;
15    }
```

4.6 vector 类型—使用迭代器

代码清单 4.5，例 4.5：

```
17 int cell;
18 cout << "请输入选择的方格编号[0-" << sz*sz - 1 << "]:" ;
19 cin >> cell;
20 if (map[cell/sz][cell%sz] == '*') {
21     cout << "选择的是地雷" << endl;
22 }else //容器nobomb存放未处理的方格编号，初始值为选择的方格编号
23     vector<int> result, nobomb(1,cell);
24 map[cell / sz][cell%sz] = '1'; //标记该方格已经遍历
25 do {
26     cell = nobomb.front(); //取出nobomb中第一个待处理的方格编号，找到与cell相邻的方格
27     int neibor[]={((cell/sz>0)?cell-sz:-1,(cell%sz>0)?cell-1:-1,
28     (cell/sz<sz-1)?cell+sz:-1,(cell%sz<sz-1)?cell+1:-1};
29     //neibor存放与cell相邻的4个方格编号，如果没有对应方格编号标记为-1
```

4.6 vector 类型—使用迭代器

代码清单 4.5，例 4.5：

```
30     for (auto &k : neibor)          //注意k!=-1必须放到逻辑与的左侧
31         if (k != -1 && map[k/sz][k%sz] == '0'){
32             nobomb.push_back(k);    //所有与cell相邻的无雷方格放到nobomb中
33             map[k/sz][k%sz] = '1'; //标记该方格已经遍历过
34         }
35     }
36     result.push_back(cell);        //将处理完的方格编号cell放到result中
37     nobomb.erase(nobomb.begin()); //将cell从nobomb中移除
38 } while (!nobomb.empty());
39 for (auto i : result)
40     cout << i << " ";
41 }
42 return 0;
43 }
```

4.7 枚举类型—定义枚举类型

增加常量

```
const int red = 0; const int green = 1; const int blue = 2; ...
```

4.7 枚举类型—定义枚举类型

增加常量

```
const int red = 0; const int green = 1; const int blue = 2; ...
```

定义枚举类型

- 不限定作用域

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

三个枚举成员的作用域与枚举类型本身的作用域相同

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

4.7 枚举类型—定义枚举类型

增加常量

```
const int red = 0; const int green = 1; const int blue = 2; ...
```

定义枚举类型

- 不限定作用域

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

三个枚举成员的作用域与枚举类型本身的作用域相同

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

- 限定作用域

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

color c = red; //正确，可以访问color 类型的枚举成员

stoplight a = red; //错误：stoplight 类型的枚举成员red在此不可访问

```
stoplight b = stoplight::red; //正确
```

4.7 枚举类型—定义枚举类型

定义枚举类型

每个枚举成员都有一个**常量整数值**，默认值从 0 开始，依次加 1。也可以指定枚举成员的值：

```
enum class week{Sunday = 7, Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday};
```

枚举类型 week 中成员 Sunday 值为 7, Monday 值为 1, Tuesday 值为 2、Wednesday 值为 3, 依次类推。

4.7 枚举类型—使用枚举类型

使用枚举类型

编译器不会把一个整型值自动转换为枚举类型:

```
color c1 = 1; // 错误：类型不匹配
```

要用强制类型转换将一个整型值转换为一个枚举常量:

```
color c2 = static_cast<color>(1);
```

4.7 枚举类型—使用枚举类型

用 switch 分支结构列举枚举成员

```
stoplight l= stoplight::red;
switch (l){
case stoplight::red:
    cout << "stop!" << endl;
    break;
case stoplight::green:
    cout << "pass carefully" << endl;
    break;
case stoplight::yellow:
    cout << "slow down" << endl;
    break;
default:
    cout << "light broken, call 122" << endl;
    break;
}
```

课后作业

作业本

- ① 习题 4.3、4.9 和 4.16

上机练习

- ① 实验指导书：第四章

本章结束