深蓝学院
shenlanxueyuan.com

C++ 基础

第 3 章：数组、 vector 与字符串

主讲人 李 伟

微软高级工程师

《 C++ 模板元编程实战》作者

目录

数组—引入

c++ compiler options: --pedantic-errors // b[x] 就可以通过，尽量不要使用这个扩展，会导致程序报错

- 将一到多个相同类型的对象串连到一起，所组成的类型
  - int a → int b[10]
    b的类型其实是int[10]
  - 数组的初始化方式：
    - 缺省初始化
    - 聚合初始化（aggregate initialization）

偶尔在struct会需要定义一个int b[0];来进行缩放。具体可以查询
arrays of length zero
更多的使用在c中

int b[3] = {1, 3, 2};
int b[] = {1, 2, 3}; 编译器能够自动推导出size为3.

- 注意事项
  - 不能使用 auto 来声明数组类型
  - 数组不能复制
    c++是一个注重效率的语言。
    复制很耗时间。所以数组不能复制
  - 元素个数必须是一个常量表达式（编译期可计算的值）
  - 字符串数组的特殊性

```
char str[] = "Hello"; // char[6]
```

```
// 查看b的type类型：
auto b = {1, 3, 4};
std::cout << typeid(b).name() << std::endl;
```

```
test1@test1-UX31A:~/demo/demo/Debug$ ./demo
St16initializer_listIiE
test1@test1-UX31A:~/demo/demo/Debug$ ./demo | c++filt -t
std::initializer_list<int>
```

```
int b[] = {1, 3, 4};
auto a = b;
std::cout << std::is_same_v<decltype(a), int*> << std::endl;
```

如果使用auto来对数组进行复制，那么得到的a是个指针，而不是数组
数组作为右值的时候会退化为指针

```
int b[] = {1, 3, 4};
auto& a = b;
std::cout << std::is_same_v<decltype(a), int(&)[3]> << std::endl;
```

数组—引入

- 数组的复杂声明
  - 指针数组与数组的指针
  - 声明数组的引用

```
int b[3];
int (&a)[3] = b;
```

- 数组中的元素访问
  - 数组对象是一个左值 在c++中，左值的概念被改为了locator value
  - 使用时通常会转换成相应的指针类型
  - x[y] → *((x) + (y))

*(a+1)
*(1+a)
1[a]
都是有效且相等的

```
int b[3];
int (*a)[3] = &b;
```

数组指针。a的解引用 *a的类型是int[3]

```
int (*a)[3];
std::cout << std::is_same_v<decltype(a), int(*)[3]> << std::endl;
```

c++中，如果我们加了个括号，我们就要把这个括号看成一个整体。
a本身是个指针，如果我们把对象解引用的话，它将指向包含三个元素，每个元素是int类型的数组。
int *a[3];
表示a是一个数组，这个数组里面存的是指针。

```
int b[3];
int (&a)[3] = b;
std::cout << std::is_same_v<decltype(a), int(&)[3]> << std::endl;
```

如下的写法是不被允许的

```
int x1;
int x2;
int x3;
int& a[3] = {x1, x2, x3};
```

```
int a[3] = {1, 2, 3};
std::cout << std::is_same_v<decltype((a)), int(&)[3]> << std::endl;
```

数组——从数组到指针

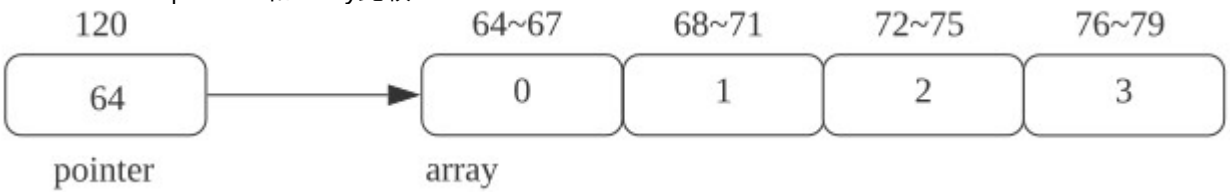
```cpp
int a[3] = {1, 2, 3};
std::cout << a[0] << std::endl;

auto b = a; // decay
```

decay：数组退化为了一个相对简单的类型。退化为了指针。

- 数组到指针的隐式转换
  - 使用数组对象时，通常情况下会产生数组到指针的隐式转换
  - 隐式转换会丢失一部分类型信息
  - 可以通过声明引用来避免隐式转换

pointer和array比较



  - 注意：不要使用 extern 指针来声明数组
    - Unknown Bounded Array 声明

```cpp
extern int* array; // error
extern int array[]; // correct unknown bounded array,
int arrayp[]; // imcomplete type
```

讲了大端和小端，值得再看一下。20分钟之后的

我们不能混用数组和指针这两个概念。虽然有的时候a可以表示第一个元素的地址。但是这是c++的一种decay.我们不能混用他们。

数组——从数组到指针

- 获得指向数组开头与结尾的指针：std::(c)begin, std::(c)end
- 指针算数：
  - 增加、减少
  - 比较
  - 求距离
  - 解引用
  - 指针索引

```
int a[3] = {1, 2, 3};
auto ptr = a;
ptr = ptr + 2;

auto ptr2 = a + 3;
(ptr == ptr2);
(ptr != ptr2);
(ptr > ptr2); // 通常不建议指针做大于小于的操作。
              但是如果参与比较的两个指针是位于
              一个数组内的。那么可以进行比较

ptr2 - ptr; // 求距离
*ptr; // 解引用
*a;
ptr[2];
```

```
int a[3] = {1, 2, 3};
&(a[0]);// 第一个元素的指针
a; ;
指向结尾的指针不是指向3的地址，而是3后面的一个地址。
// 指向结尾的指针
a + 3
&(a[3])
```

```
int a[3] = {1, 2, 3};
std::cout << a << ' ' << &(a[0]) << ' ' << std::begin(a) << std::endl;
std::cout << a + 3 << ' ' << &(a[3]) << ' ' << std::end(a) << std::endl;
```

```
std::begin(a); // int*
std::cbegin(a); // const int*
```

```
auto b = a;
std::cout << std::begin(b) << std::endl;
std::cout << std::end(b) << std::endl;
```

报错，丢失了一些信息。

```
auto& b = a;
std::cout << std::begin(b) << std::endl;
std::cout << std::end(b) << std::endl;
```

通过。表示b是a的一个别名。不会丢失信息

数组—其它操作

- 求元素的个数
  - sizeof 方法
  - std::size 方法
  - (c)end - (c)begin 方法

c语言常见方法，对象或类型所占的尺寸。

```
int a[3];
sizeof(a); // 12
sizeof(int); // 4
sizeof(a) / sizeof(int); // 3
```

```
int a[3];
std::size(a); // 3
```

```
int a[3];
std::end(a)-std::begin(a); // 3，这个方法在运行期才会得到结果。会使运行期速度相对变慢，因此不推荐这个方法
```

- 元素遍历
  - 基于元素个数
  - 基于 (c)begin/(c)end
  - 基于 range-based for 循环

推荐c++ insights，可以看见c++的等价代码

```cpp
int a[4] = {2, 3, 5, 7};

size_t index = 0;
while (index < std::size(a))
{
    std::cout << a[index] << std::endl;
    index = index + 1;
}
```



```cpp
int a[4] = {2, 3, 5, 7};

auto ptr = std::cbegin(a);
while (ptr != std::cend(a))
{
    std::cout << *ptr << std::endl;
    ptr = ptr + 1;
}
```

```cpp
int a[4] = {2, 3, 5, 7};

for (int x : a)
{
    std::cout << x << std::endl;
}
```

数组— C 字符串

- C 字符串本质上也是数组

```
char str[] = "hello"; // 最后一个字符是null character/ null-terminated string
```

- C 语言提供了额外的函数来支持 C 字符串相关的操作 : strlen, strcmp…

```
#include <cstring>
char str[] = "hello";
auto ptr = str;
strlen(str); // 5
strlen(ptr); // 5
```

数组——多维数组

```
int x1[3];

int x2[3][4];
std::cout << sizeof(int) << std::endl;
std::cout << sizeof(x2[0]) << std::endl;
std::cout << std::is_same_v<decltype(x2[0]), int(&)[4]> << std::endl;
```


- 本质：数组的数组
  - int a[3][4];
- 多维数组的聚合初始化：一层大括号 V.S. 多层大括号
- 多维数组的索引与遍历
  - 使用多个中括号来索引
  - 使用多重循环来遍历
- 指针与多维数组
  - 多维数组可以隐式转换为指针，但只有最高维会进行转换，其它维度的信息会被保留
  - 使用类型别名来简化多维数组指针的声明
  - 使用指针来遍历多维数组

```
int x2[3][4] = {1, 2, 3, 4, 5};
size_t index0 = 0;
while (index0 < std::size(x2))
{
    size_t index1 = 0;
    while (index1 < std::size(x2[index0]))
    {
        std::cout << x2[index0][index1] << std::endl;
        index1 = index1 + 1;
    }
    index0 = index0 + 1;
}
```

```
#include <iostream>
#include <type_traits>

int main()
{
    int x2[3][4] = {1, 2, 3, 4, 5};
    for (auto& p : x2)
    {
        for (auto q : p)
        {
            std::cout << q << '\n';
        }
    }
}
```

这里必须有引用。否则会编译错误。因为auto p; x2中的p会变成一个指针，造成auto q: p 无法编译

```
int x2[3][4];
auto ptr = std::begin(x2);
while (ptr != std::end(x2))
{
    auto ptr2 = std::begin(*ptr);
    while (ptr2 != std::end(*ptr))
    {
        std::cout << *ptr2 << std::endl;
        ptr2 = ptr2 + 1;
    }
    ptr = ptr + 1;
```

```
using A = int[4];

int main()
{
    A x2[3];
    A* ptr = x2;
}
```

```
using A2 = int[4][5];

int main()
{
    int x2[3][4][5];
    A2* ptr = x2;
    auto ptr1 = ptr[0];
}
```

vector

- 是 C++ 标准库中定义的一个类模板

内建数组速度更快

- 与内建数组相比，更侧重于易用性
  - 可复制、可在运行期动态改变元素个数
- 构造与初始化
  - 聚合初始化
  - 其它的初始化方式
- 其它方法
  - 获取元素个数、判断是否为空
  - 插入、删除元素 `x1.pop_back(); //从x1中删除元素`
  - vector 的比较

```
std::vector<int> x1 = {1, 2, 3};
std::vector<int> x2 = {1, 3, 2};
std::cout << (x1 == x2) << std::endl;
std::cout << (x1 > x2) << std::endl;
```

输出0，0

vector

- vector 中元素的索引与遍历：
  - [] V.S. at
  - (c)begin / (c)end 函数　V.S. (c)begin / (c)end　方法
- 迭代器
  - 模拟指针的行为
  - 包含多种类别，每种类别支持的操作不同
  - vector 对应随机访问迭代器
    - 解引用与下标访问
    - 移动
    - 两个迭代器相减求距离
    - 两个迭代器比较

两个迭代器相减求距离

```
std::vector<int> x1 = {1, 2, 3};
auto b = std::begin(x1);
auto e = x1.end();
std::cout << e - b << std::endl;
```

下标访问

```
std::vector<int> x1 = {1, 2, 3};
auto b = std::begin(x1);
b[1] -> *(b + 1)
```

移动：

```
auto b = std::begin(x1);
b = b + 1
```

vector

- vector 相关的其它内容
  - 添加元素可能使迭代器失效
  - 多维 vector
  - 从 . 到 -> 操作符
  - vector 内部定义的类型
    - size_type
    - iterator / const_iterator

Iterator invalidation

Operations	Invalidated
All read only operations	Never
swap, std::swap	end()
clear, operator=, assign	Always
reserve, shrink_to_fit	If the vector changed capacity, all of them. If not, none.
erase	Erased elements and all elements after them (including end())
push_back, emplace_back	If the vector changed capacity, all of them. If not, only end().
insert, emplace	If the vector changed capacity, all of them. If not, only those at or after the insertion point
resize	If the vector changed capacity, all of them. If not, only end() and any elements erased.
pop_back	The element erased and end().

```
std::vector<std::vector<int>> x;
x.push_back(std::vector<int>());
x[0].push_back(1);
std::cout << x[0][0] << std::endl;
```

```
std::vector<int> x;
std::cout << x.size() << std::endl;

std::vector<int>* ptr = &x;
std::cout << (*ptr).size() << std::endl;
std::cout << ptr->size() << std::endl;
```

string

```
std::string x = "Hello world";
std::string y = x;
y = y + " !";
```

- 是 C++ 标准库中定义的一个类模板特化别名，用于内建字符串的代替品

```
std::string y("Hello");
std::cout << (y < x) << '\n';
auto ptr = y.c_str();
std::cout << ptr << std::endl;
```

- 与内建字符串相比，更侧重于易用性
  - 可复制、可在运行期动态改变字符个数
- 构造与初始化
- 其它方法
  - 尺寸相关方法（ size / empty ）
  - 比较
  - 赋值
  - 拼接
  - 索引
  - 转换为 C 字符串

深蓝学院
shenlanxueyuan.com

感谢聆听！

Thanks for Listening