【C++】Day ten(2)

Class	C++
Date	@November 28, 2021
Material	
Series Number	
Summary	

【Ch3】Array

3.5 Arrays

An array is a data structure that is similar to the library vector type but offers a different trade-off between performance and flexibility. Like a vector, an array is a container of unnamed objects of a single type that we access by position.

Unlike a vector, arrays have fixed size; we cannot add elements to an array. Because arrays have fixed size, they sometimes offer better run-time performance for specialized applicaions. However, that run-time advantage comes at the cost of lost flexibility

Tip: If you don't know exactly how many elements you need, use a vector.

3.5.1 Defining and Initializing Built-in Arrays

Arrays are a compound type. An array declarator has the form `a[d]`, where a is the name being defined and d is the dimension of the array. The dimension specifies the number of elements and must be greater than zero.

The dimension must be known at compile time, which means that the dimension must be a constant expression.

```
unsigned cnt = 42; //not a constant expression
constexpr unsigned sz = 42; //constant expression

int arr[10]; //array of ten ints
int *parr[sz]; //array of 42 pointers to int
string bad[cnt]; //error: cnt is not a constant expression
string strs[get_size()]; //ok if get_size is constexpr, error otherwise
```

When we define an array, we must specify a type for the array. As with vector, arrays hold object. Thus, there are no arrays of references.

Explicitly Initializing Array Elements

We can list initialize the elements in an array. When we do so, we can omit the dimension.

- If we omit the dimension, the compiler infers it from the number of initializers.
- If we specify a dimension, the number of initializers must not exceed the specified size.
- If the dimension is greater than the number of initializers, the initializers are used for the first elements and any remaining elements are value initialized.

```
const unsigned sz = 3;
int ial[sz] = {0, 1, 2}; //array of three ints with values 0, 1, 2
int a2[] = {0, 1, 2}; //an array of dimension of 3
int a3[5] = {0, 1, 2}; //equivalent to a3[] = {0, 1, 2, 0, 0};
string a4[3] = {"hi", "bye"}; //same as a4[] = {"hi", "bye", ""}
int a5[2] = {0, 1, 2}; //error: too many initializers
```

Character Arrays Are Special

Character arrays have an additional form of initialization: We can initialize such arrays from a string literal.

```
char a[] = "C++"; //null terminator added automatically
const char a2[6] = "Daniel"; //error: no space for the null!
```

No Copy or Assignment

We cannot initialize an array as a copy of another array, nor is it legal to assign one array to another

```
int a[] = {0, 1, 2}; //array of three ints
int a2[] = a; //error: cannot initialize one array with anotehr
```

Understanding Complicated Array Declarators

```
int *ptrs[10]; //ptrs is an array of ten pointers to int
int &refs[10] = /*..*/; //error: no arrays of references
int (*Parray)[10] = &arr; //Parray points to an array of ten ints
int (&arrRef)[10] = arr; //arrRef refers to an array of ten ints
int *(&arry)[10] = ptrs; //arry is a reference to an arry of ten int pointers
```

Tips: It can be easier to understand array declarations by starting with the array's name and reading them from the inside out.

3.5.2 Accessing the Elements of An Array

When we use a variable to subscript an array, we normally should define that variable to have type `size_t`. `size_t` is a machine-specific unsigned type that is guaranteed to be large enough to hold the size of any object in memory. The size_t type is defined in the cstddef header, which is the C++ version of the stddef.h header from the C library.

Warning: The most common source of secuirty problems are buffer overflow bugs. Such bugs occur when a program fails to check a subscript and mistakenly uses memory outside the range of an array or similar data structure.

3.5.3 Pointers and Arrays

In C++ pointers and arrays are closely intertwined. In particular, when we use an array, the compiler ordinarily converts the array to a pointer.

We can obtain a pointer to an array element by taking the address of that element:

```
string nums[] = {"one", "two", "three"}; //array of strings
string *p = &nums[0]; //p points to the first element in nums
```

Pointers are Iterators

We can use the increment operator to move from one element in an arry to the next:

```
int arr[] = {0, 1, 2, 3};
int *p = arr; //p points to the first element in arr
++p; //p points to arr[1]
```

Just as we can use iterators to traverse the elements in a vector, we can use pointers to traverse the elements in an array. In order to do so, we need to obtain pointers to the first and one past the last element.

We can obtain a pointer to the first element by using the array itself or by taking the address-of the first element.

We can obtain an off-the-end pointer by using another special property of arrays. We can take the address of the nonexistent element one past the last element of an array:

```
int *e = &arr[10]; //pointer just past the last element in arr
```

An off-the-end pointer does not point to an element. As a result, we may not dereference or increment an off-the-end pointer.

Using these pointers we can write a loop to print the elements in arr as follows:

```
for(int *b = arr; b != e; ++b)
  cout << *b << endl; //print the elements in arr
```

The Library being and end Functions

Although we can compute an off-the-end pointer, doing so is error-prone. To make it easier and safer to use pointers, the new library includes two functions, named `begin` and `end`. These functions act like the similarly names container members. However, arrays are not class types, so these functions are not member functions. Instead, they take an argument that is an array:

```
int ia[] = {0, 1, 2, 3, 4};
int *beg = begin(ia); //pointer to the first element in ia
int *last = end(ia); //pointer one past the last element in ia
```

`begin` returns a pointer to the first, and `end` returns a pointer one past the last element in the given array: These functiosn are defined in the iterator header.

Note: A pointer "one past" the end of a built-in array behaves the same way as the iterator returned by the end operation of a vector. In particular, we may not dereference or increment an off-the-end pointer.

Subscripts and Pointers

In most places when we use the name of an array, we are really using a pointer to the first element in that array. One place where the compiler does this transformation is when we subscript an arary.

```
int ia[] = {0, 2, 4, 6, 8};
int i = ia[2]; //ia is converted to a pointer to the first element in ia. ia[2] fetches the element to which (ia + 2) points
int *p = ia; //p points to the first element in ia
i = *(p + 2); //equivalent to i = ia[2]
```

We can use the subscript operator on any pointer, as long as that pointer pointse to an element(or one past the last element) in an array:

```
int *p = &ia[2]; //p points to the element indexed by 2
int j = p[1]; //p[1] is equivalent to *(p + 1)
int k = p[-2]; //p[-2] is the same element as ia[0]
```