> 由 KiOii (*_EM_Cpper*)整理。 (KiOii (*_EM_Cpper*) makes this note.)

Chapter 3. String, Vectors, and Arrays

Contents

- **Section 3.1 Namespace using Declarations**
- **Section 3.2 Library string Type**
- **Section 3.3 Library vector Type**
- **Section 3.4 Introducing Iterators**
- **Section 3.5 Multidimensional Arrays**
- **Chapter Summaty**

3.1 Namespace using Declarations

- *scope operator:* **: :**
  - *code-look:* `std::cin >> num;`
- *using:* not only can define **type alias**,but also **use namespace**
  - *use:* ***using new-type-name = old-type-name;***
    - *code-look:* `using uint = unsigned int; // define type alias`
  - *use:* ***using namespace :: name;***
    - *code-look:* `using std::cout;//using the name "cout" in namesapce which named "std"`

- *use:* ***using namespace namespace-name ;***
  - *code-look:* `using namesapce std; // using all names in namespace which named "std"`

Wraning

- Headers should not include using declaraction

3.2 Library string Type

> The section describes the most common string operations

3.2.1 Defining and Initializing strings

- *the most common ways to initialize string*

```
string s1;             // default initailization
string s2 = s1;        // s2 is a copy of s1
string s3 = "hiya";    // s3 is a copy of the string litral
string s4(10,'c');     // s4 is ccccccccc
```

Deirect and Copy Forms of Initialization

- *copy initialization:* using **=**
- *direct initailization:* not using **=**

```
string s5 = "hiya";         // copy initialization
string s6("hiya");          // direct initialization
string s7(10,'c');          // direct initialization
string s8 = string(10,'c'); // copy initialization,but we should not do that.
```

3.2.2 Operations on string

`os << s`	Writes `s` onto output stream `os`. Returns `os`.
`is >> s`	Reads whitespace-separated string from `is` into `s`. Returns `is`.
`getline(is, s)`	Reads a line of input from `is` into `s`. Returns `is`.
`s.empty()`	Returns `true` if `s` is empty; otherwise returns `false`.
`s.size()`	Returns the number of characters in `s`.
`s[n]`	Returns a reference to the `char` at position `n` in `s`; positions start at 0.
`s1 + s2`	Returns a `string` that is the concatenation of `s1` and `s2`.
`s1 = s2`	Replaces characters in `s1` with a copy of `s2`.
`s1 == s2`	The `strings s1` and `s2` are equal if they contain the same characters.
`s1 != s2`	Equality is case-sensitive.
`<, <=, >, >=`	Comparisons are case-sensitive and use dictionary ordering.

Reading and Writing strings

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

- *cin-string-wraning:* reads and **discards any leading whitespace**(' '、'\t'、'\n'). It then reads characters until the next whitespace.[1]

Reading an Unknown Number of string

```
int main()
{
    string word;
    while(cin >> word)            // read until end-of-file
        cout << word << endl;
    return 0x0;
}
```

Using *getline* to Read an Entire Line

- *not ignore the whitespace:*

```
int main()
{
    string line;
    while(getline(cin,line)) // read input a line at a time until end-of-file
        cout << line << endl;
    return 0x0;
}
```

- *wraning:* read char until **'\n'('\r')**, read but not include newline[2]

The string *empty* and *size* Operations

- *empty operator:* **return bool type**
- *size operator:* **return string::size_type**

The string::size_type Type

> The string class—and most other library types—defines several companion types. These companion types make it possible to use the library types in a machineindependent manner.

- *define:* **unsigned type** and **enough to hold the size of any string**
- *wraning:* we can use auto and decltype

```
auto len = line.size();
```

- *wraning:* avoid use unsigned and int in expression

Comparing strings

```
string str = "Hello";
string phrase = "Hello World";  // phrase > str
string slang = "Hiya";          // slang > phrase
```

Assignment for strings

```
string st1(10,'c'),st2;
st1 = st2;
```

Adding Two strings

```
string s1 = "Hello,",s2 = "World!\n";
string s3 = s1 + s2;
s1 += s2; // s1 = s1+s2;
```

Adding Literals and strings

```
string s1 = "hello";
string s2 = s1 + ',';  // ok,left-hand is string
string s3 = ',' + s1;  // ok,right-hand is string
string s4 = ',' + "hello"; // error,每个+左右两侧必须至少有一个是string
string s5 = (s1 + ',') + "World"; // ok
```

3.2.3 Dealing with the Characters in a string

- *how change characteristics of a character:*

`isalnum(c)`	`true` if `c` is a letter or a digit.
`isalpha(c)`	`true` if `c` is a letter.
`iscntrl(c)`	`true` if `c` is a control character.
`isdigit(c)`	`true` if `c` is a digit.
`isgraph(c)`	`true` if `c` is not a space but is printable.
`islower(c)`	`true` if `c` is a lowercase letter.
`isprint(c)`	`true` if `c` is a printable character (i.e., a space or a character that has a visible representation).
`ispunct(c)`	`true` if `c` is a punctuation character (i.e., a character that is not a control character, a digit, a letter, or a printable whitespace).
`isspace(c)`	`true` if `c` is whitespace (i.e., a space, tab, vertical tab, return, newline, or formfeed).
`isupper(c)`	`true` if `c` is an uppercase letter.
`isxdigit(c)`	`true` if `c` is a hexadecimal digit.
`tolower(c)`	If `c` is an uppercase letter, returns its lowercase equivalent; otherwise returns `c` unchanged.
`toupper(c)`	If `c` is a lowercase letter, returns its uppercase equivalent; otherwise returns `c` unchanged.

- *wraning:* use <**ctype**> but <**type.h**>

Use Range-Based for

- *use:* **for(declaration : expression) { statement }**
  - *declaration:* define a variable
  - *expression:* an **object** of a type that represents a **sequence**
  - *code-look:*

```
string str("some thing");
for(auto c : str)                // for every char in str
    cout << c << endl;
```

Using a Range for to Change the Characters in a string

- *declaration:* a **reference type**
- *code-look:*

```
string s("Hello World!");
for(auto &c : s)
    c = toupper(c);
cout << s << endl;
```

Processing Obly Some Characters?

- *two ways:* **subscript** or **iterator**[3]
- *subscript operator:* **[ ]**,takes a **string::size_type**

- *last char:* **str[str.size() - 1]**

3.3 Library vector Type

- *define:* **a collection of objects**,all of which have the same type
- *every object int vector:* has an associated **index**,**access** to that object.
- *include:* <**vector**>
- *wraning:* **A vector is a class template**
- *instantiation:* create **classes** or **function** from **template**

```
vector<int>             ivec;             // ivec holds objects of type int
vector<Sales_item>      Sales_vec;
vector<vector<string>>  file;
```

3.3.1 Defining and Initailizing vectors

Table 3.4. Ways to Initialize a vector

`vector<T> v1`	`vector` that holds objects of type `T`. Default initialization; `v1` is empty.
`vector<T> v2(v1)`	`v2` has a copy of each element in `v1`.
`vector<T> v2 = v1`	Equivalent to `v2(v1)`, `v2` is a copy of the elements in `v1`.
`vector<T> v3(n, val)`	`v3` has `n` elements with value `val`.
`vector<T> v4(n)`	`v4` has `n` copies of a value-initialized object.
`vector<T> v5{a,b,c...}`	`v5` has as many elements as there are initializers; elements are initialized by corresponding initializers.
`vector<T> v5 = {a,b,c...}`	Equivalent to `v5{a,b,c...}`.

List Initializing a vector

```
vector<string> atr{"a","an","the"};    // direct,initialized_list
vector<string> atr = {"a","an","the"}; // copy,initialized_list
// C++11  {} initialized
```

Craeting a Specified Number of Elements

```
vector<int> ivec(10,-1);  //ten int elements,each initialized to -1
```

Value Initialization

```
vector<int> ivec(10);      // ten elements,each initialized to 0
vector<int> vi = 10;       // error
```

List Initializer or Element Count?

- *construct:* **()**
- *list initialize:* **{ }**
- *copy construct or assign:* **=**

3.3.2 Adding Elements to a vector

- **push_back**
- *wraning:* we can not use **range for**,because it change the size of vector object

3.3.3 Other vector Operations

Table 3.5. vector Operations

`v.empty()`	Returns `true` if `v` is empty; otherwise returns `false`.
`v.size()`	Returns the number of elements in `v`.
`v.push_back(t)`	Adds an element with value `t` to end of `v`.
`v[n]`	Returns a reference to the element at position `n` in `v`.
`v1 = v2`	Replaces the elements in `v1` with a copy of the elements in `v2`.
`v1 = {a,b,c...}`	Replaces the elements in `v1` with a copy of the elements in the comma-separated list.
`v1 == v2`	`v1` and `v2` are equal if they have the same number of elements and each element in `v1` is equal to the corresponding element in `v2`.
`v1 != v2`	
`<, <=, >, >=`	Have their normal meanings using dictionary ordering.

- *wraning:* We can compare two vectors only if we can compare the elements in those vectors.

Computing a vector Index

- *wraning:* These indices are between **0** and scores.**size() - 1**

Subscripting Does Not Add Elements

3.4 Introucing Iterators

> *access the elements:* **iterators** and **subscript**

- *what:* indirect access to an object，like pointer

3.4.1 Using Iterators

- *begin member:* **return an iterator** that denotes the **first element**
- *end member:* **one past the end** of the associated container (***off-the-end iterator***)[4]
- *wraning:* If the container is empty, **begin** returns **the same iterator** as the one returned by **end.**

```
auto b = v.begin(),e = v.end();
```

Iterator Operations

Table 3.6. Standard Container Iterator Operations

`*iter`	Returns a reference to the element denoted by the iterator `iter`.
`iter->mem`	Dereferences `iter` and fetches the member named `mem` from the underlying element. Equivalent to `(*iter).mem`.
`++iter`	Increments `iter` to refer to the next element in the container.
`--iter`	Decrements `iter` to refer to the previous element in the container.
`iter1 == iter2` `iter1 != iter2`	Compares two iterators for equality (inequality). Two iterators are equal if they denote the same element or if they are the off-the-end iterator for the same container.

Moveing Iterators from One Element to Another

- *increment operator:* **++**

Iterator Types

```
vector<int>::iterator it;   // can read and write vector<int> elements
string::iterator it2;       // can read and write characters int a string
vector<int>::const_iterator it3; // can read but not write vector<int> element
```

The *begin* and *end* Operations

- *return type:*
  - *const_iterator:* **when the object is const**
  - *iterator:* **when the object is not const**
- *cbegin and cend:* the same to **begin and end**,but **return const_iterator**
  - *defined standard:* **C++11**

Combining Dereference and Member Access

- *dereference:* **get the object that the iterator denotes.**
  - *use:* *****
    - *code-look:* `(*it).empty() ; // dereference it and calls the member empty`
  - *use:* **->**
    - *code-look:* `it->empty(); // the same to (*it).empty()`

Some vector Operations Invalidate Iterators

- we cannot add elements to a vector inside a range for loop
- we cannot use old iterator if we change the size of a vector

3.4.2 Iterator Arithmetic

Table 3.7. Operations Supported by vector and string Iterators

`iter + n` `iter - n`	Adding (subtracting) an integral value `n` to (from) an iterator yields an iterator that many elements forward (backward) within the container. The resulting iterator must denote elements in, or one past the end of, the same container.
`iter1 += n` `iter1 -= n`	Compound-assignment for iterator addition and subtraction. Assigns to `iter1` the value of adding `n` to, or subtracting `n` from, `iter1`.
`iter1 - iter2`	Subtracting two iterators yields the number that when added to the right-hand iterator yields the left-hand iterator. The iterators must denote elements in, or one past the end of, the same container.
`>, >=, <, <=`	Relational operators on iterators. One iterator is less than another if it refers to an element that appears in the container before the one referred to by the other iterator. The iterators must denote elements in, or one past the end of, the same container.

Using Iterator Arithmetic

```
auto beg = text.begin() ,end = text.end();
auto mid = text.begin() + (end - beg)/2;
while(mid != end && *mid != sought){
    if(sought < *mid)
        end = mid;
    else
        beg = mid + 1;
    mid = beg + (end - beg)/2;
}
```

3.5 Arrays

- we access by position
- arrays have fixed size

3.5.1 Defining and Initializing Built-in Arrays

- *declarator:* ***name[dimension]***
    - *dimension:* the number of elements
        - must be greater than zero
        - must be a constant expression

- *wraning:* we **cannot use auto to deduce the type from a list of initializers.**
- *wraning:* **no arrays of references**[5]

Explicitly Initializing Array Elements

```
const unsigned sz = 3; // constant expression
int ial[sz] = {0,1,2}; // ok,dimension 3,{ 0, 1, 2}
int a2[] = {0,1,2};    // ok,dimension 3,{ 0, 1, 2}
int a3[5] = {0,1,2};   // 0k,dimension 5,{ 0, 1, 2, 0, 0}  value initialized
```

Character Arrays Are Special

```
char a1[] = {'c','+','+'};      // list initialization,no null
char a2[] = {'c','+','+','\0'}; // list initialization,explict null
char a3[] = "c++";              // add null automatically
char a4[3] = "c++";             // error,need dimension 4
```

No Copy or Assignment

```
int a[] = {0,1,2};
int a2[3] = a;      // error
a2 = a;             // error
```

Understanding Complicated Array Declarations

> Arrays is object
>
> We can define both pointers and references to arrays

```
int *ptr[10];               // array,element type is pointer,points to int
int &refs[10];              // error,no element type is reference
int (*pArray)[10] = &arr;   // pointer,points to an array of ten elements
int (&refArr)[10] = arr;    // reference,refers to an array of ten elements
```

- inside to outside
- right to left

```
int *(&arry)[10] = ptrs;    // reference,refers to an array,element is a pointer,points to int
```

3.5.2 Accessing the Elements of an Array

- *subscript-type:* **size_t**

    - we have learn the **size_type**
    - *size_t:* a **machine-specific unsigned type** and **large enough** to hold the size of any object in memory.

- *using range for*

```
unsigned scores[11] = {};
for(auto i : scroes)
{
    //...
}
```

3.5.3 Pointers and Arrays

```
string nums[]  = {"one","two","three"};
string *p = nums;   // compiler will change the same to :  p = &num[0];
```

- *auto in array*

```
int ia[] = {0,1,2,3};
auto ia2(ia);            // auto is int*
```

- *decltype in array*

```
int ia[] = {0,1,2,3};
decltype(ia) ia2 = {0,1,2,3};  // decltype(ia) :int[4]
```

Pointers are Iterators

> Pointers to array elements support the same operations as iterators on vector or string.

The library begin and end Function

- *define standard:* **C++11**, <**iterator**>

```
int ia[] = {0,1,2,3};
int *beg = begin(ia);
int *end = end(ia);
```

difference_type and ***ptrdiff_t***

- *defference_type:* the return-type of **Iterator - Iterator**
- *ptridiff_t:* the return-type of **Pointer - Pointer**

Subscripts and Pointers

- *subscript*、*dimension:*

- *subscript in string and vector:* **an unsigned type**
- *dimension:* **an unsigned type**
- *subscript in array:* **an signed type**

```
int *p = &arr[2];
int k = p[-2];    // the same to : k = arr[0];
```

3.5.4 C-Style Character Strings

> C-style strings are a surprisingly rich source of bugs and are the root cause of many security problems.

C Library String Functions

Table 3.8. C-Style Character String Functions

`strlen(p)`	Returns the length of `p`, *not counting the null*.
`strcmp(p1, p2)`	Compares `p1` and `p2` for equality. Returns `0` if `p1 == p2`, a positive value if `p1 > p2`, a negative value if `p1 < p2`.
`strcat(p1, p2)`	Appends `p2` to `p1`. Returns `p1`.
`strcpy(p1, p2)`	Copies `p2` into `p1`. Returns `p1`.

3.5.5 Interfacing to Older Code

- *C-style strings to string:*
  - `string s{"Hello World"};`
- *string to C-style:*

```
char buffer[10] = {};
string s = "Hello World";
strcpy(buffer,s.c_str());
```

Using an Array to Initialize a vector

```
int int_arr = {0,1,2,3};
vector<int> ivec(begin(int_arr), end(int_arr));
```

3.6 Multidimensional Arrays

> there are no multidimensional arrays in C++.

> multidimensional arrays are actually arrays of arrays.[6]

```
int ia[3][4];        // array of size 3;each element is an array of ints of size 4
```

Initializing the Elements of a Multidimensional Array

```
int ia[3][4] = {
    {0,1,2,3},
    {4,5,6,7},
    {8,9,10,11}
};
// the same to
int ia[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Using a Range for with Multidimensional Arrays

```
int ia[3][4]= {0};
size_t cnt = 0;
for(auto &row : ia){
    for(auto &col : row){
        col = cnt;
        ++cnt;
    }
}
```

- *wraning:* **in this example**
  - **auto &row**: the type of row is: **int (&)[4]**
  - **auto row**:the type of row is:**int***,it may be error in multidimensional arrays

Pointers and Multidimensional Arrays

```
for(auto row = ia;p != ia + 3;++row) // we have say:arrays of arrays
    for(auto col = *row;col != *row + 4;++col)
        cout << *col << ' ';
```

Type Aliases Simplify Pointers to Multidimensional Arrays

```
using int_array = int[4];  // the same to :typedef int int_array[4];
for(int array *row = ia;row != ia + 3; ++row)
    for(int *col = *row;col != *row + 4;++col)
        cout << *col << ' ';
```

1. 在进行读取操作时，string对象会自动忽略开头的部分，并从一个真正的字符开始读取，直到遇到下一个空白为止↩

2. 读取了'\n',但是并没有存储到string object 中,被丢弃↩

3. 下标 或者 迭代器↩

4. 尾元素的下一个位置的迭代器↩

5. 不存在元素是索引的数组↩

6. 不存在多维数组，通常地说，多维数组就是数组的数组↩