C++ Basic Course

Chapter 3:
String, Vector
and Array

群杜工作室

Last Time

- decltype
- HW1 and Lab2

decltype

- auto tells the compiler to deduce the type from the initializer.
- decltype tells the compiler to deduce type from an expression. The compiler analyzes the expression to determine its type but does not evaluate the expression.

See Note

Namespace

- Mechanism for putting names defined by a library into a single logical place.
- Namespaces help avoid name clashes (牴觸). The names defined by the C++ library are in the namespace std.

```
std::cout
```

- A using declaration allows us to access a name from a namespace without the cumbersome prefix namespace_name:: (e.g., std::)

See Note

Headers **Should Not** Include using Declaration

- Inside header files, we should *always* use the fully qualified library names, that is, **DO NOT** use using declaration. (why?)

```
#ifndef COORDH
#define COORDH

struct Coord {
    double x;
    double y;
    double z;
    void print_x() {std::cout << x;}
};
#endif
```

```
#ifndef COORDH
#define COORDH
using namespace std;

struct Coord {
    double x;
    double y;
    double z;
    void print_x() {cout << x;}
};
#endif
```

string type

- The string type supports variable-length character strings.
- The library takes care of managing the memory and provides various useful operations.

string s1;	**Default constructor; s1 is the empty string**
string s2(s1);	**Initialize s2 as a copy of s1**
string s3("value");	**Initialize s3 as a copy of the string literal**
string s4(n, 'c');	**Initialize s4 with n copies of the character 'c'**

string I/O

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

- Reads and discards any leading whitespace (e.g., spaces, newlines, tabs)
- It then reads characters until the next whitespace character is encountered.

```
string line;
getline(cin, line);
```

- Reads the next line of input stream and store what it reads, not including the newline.

See Note

Dealing with characters in a string

- See Table 3.3 cctype function (see note).

```
for (string::size_type index = 0; index != s.size();
      ++index)
      if (ispunct(s[index])) ++punct_cnt;

for (auto index = 0; index != s.size(); ++index)
      if (ispunct(s[index])) ++punct_cnt;


for (auto e: s)
      if (ispunct(e)) ++punct_cnt;
```

Exercise, see note

vector type

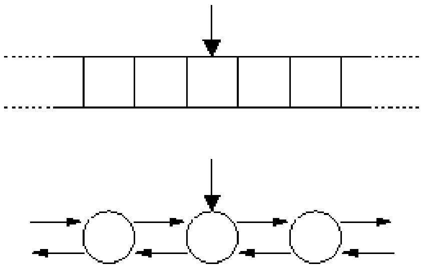
- A vector is a collection of objects of a single type, each of which has an associated integer index.
- A vector is a **class template**. To declare objects of a type generated from vector, we must supply what type of objects the vector will contain. We specify the type by putting it between a pair of angle brackets following the template's name:

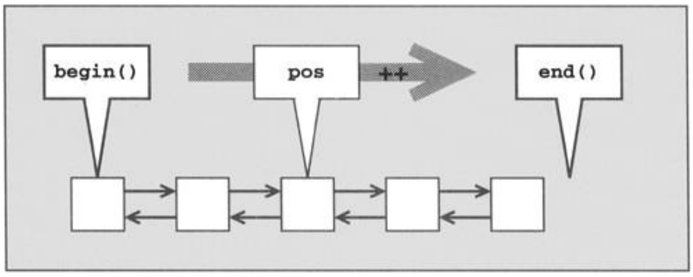
```
vector<int> ivec;
vector<Sales_item> salesVec;
vector<vector<int> > matInt;
```

See Note

Iterator

- An iterator is a generalized pointer with a mechanism that lets us:
  - identify the position and access the elements in a container
  - navigate from one element to another
- Except for vector, modern C++ programs tend to use iterators rather than subscripts to access container elements.

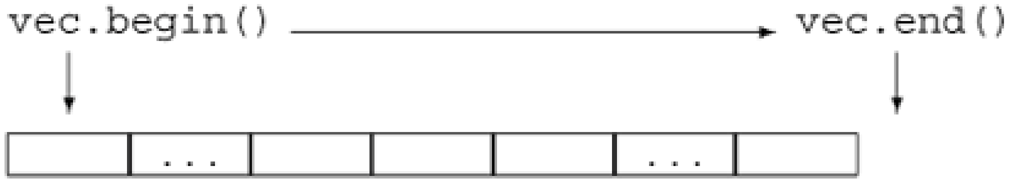




- Each container defines its own iterator type.

```
vector<int>::iterator iter;
vector<Sales_item>::iterator it;
set<double>::iterator it2;
```

- Each container defines a pair of functions begin and end that return iterators and cbegin and cend that return const_iterators.



vec.end(): an iterator positioned “one past the end”

- In general, we do not care the precise type of iterator

```
vector<int> vec; …
auto b = vec.beign()
```

- Iterator is a pointer and it uses the dereference operator (the * operator) to access the element to which the iterator refers

```
*iter = 0;
```

- Iterators use the increment operator (++) to advance an iterator to the next element in the container.

```
++iter;
```

- Looping through a container using iterator and const_iterator for reading only (see note).

Array

- An array consists of a type specifier (int), an identifier (myArray, yourArray), and a dimension.

- The type specifier indicates what **type** the elements stored in the array. The dimension specifies how many elements the array will contain.

```
int  intArray[10]; // an array of 10 ints

Sales_item  item[10]; // an array of 10 Sales_items
```

- Unlike vector, array has fixed size for better run-time performance (but at the cost of lost flexibility)

- The dimension must be a constant expression (see note).

Initializing Array Elements

```
int  intArray[3] = {0, 1, 2}; // element initialization

int  intArray[] = {0, 1, 2}; // element initialization
```

```
char  ca1[] = {‘C’, ‘+’, ‘+’}; // dim = 3
char  ca2[] = {‘C’, ‘+’, ‘+’, ‘\0’}; // dim = 4
char  ca3[] = “C++”; // dim = 4
```

char array is special

- If we do not supply explicit initialization, elements in an array are default initialized.

```
int  intArray[3];
string sArray[3];
```

See Note

Pointers and Arrays

- When we use the name of an array in an expression, that name is automatically converted into a pointer to the first element of the array:

```
int ia[] = {0,2,4,6,8};
int *ip = ia; // ip points to ia[0]
```

- We can use pointer arithmetic to compute a pointer to an element by adding (or subtracting) an integral value to (or from) a pointer to another element in the array:

```
int *ip2 = ip+4; // ip2 points to ia[4]
```

See Note

Until Next Time

- Lab3
- HW2
- [Reading] Chapter 6 (function) and Chapter 7 (Class).