【C++】Day eight

Class	C++
Date	@November 26, 2021
Material	
Series Number	
Summary	

【Ch3】Namespace using Declarations

3.1 Namespace using Declarations

Up to now, our programs have explicitly indicated that each library name we use is in the `std` namespace. For example, to read from the standard input, we write `std::cin`. These names use the scope operator `::`, which says that the compiler should look in the scope of the left-hand operand for the name of the right-hand operand. Thus, `std::cin` says that we want to use the name `cin` from the namespace `std`.

A `using` declaration lets us use a name from a namespace without qualifying the name with a `namespace_name::prefix`. A using declaration has the form:

```
using namespace::name;
```

Once the using declaration has been made, we can access name directly:

```
#include <iostream>
//using declaration: When we use the name cin, we get the one from the namespace std
using std::cin;
int main() {
  int i;
  cin >> i; //ok: cin is a synonym for std::cin
  coud << i; //error: no using declaration; we must use the full name
  std::cou << i; //ok: explicityly use cout from namespace std
}
```

Note: There must be a using declaration for each name we use, and each declaration must end in a semicolon.

Headers Should Not Include using Declarations

Code inside headers ordinarily should not use `using` declarations. The reason is that the contents of a header are copied into the including program's text. If a header has a using declaration, then ever program that includes that hdeader gets that same using declaration. As a result, a program that didn't intend to use the specified library name might encounter unexpected name conflicts.

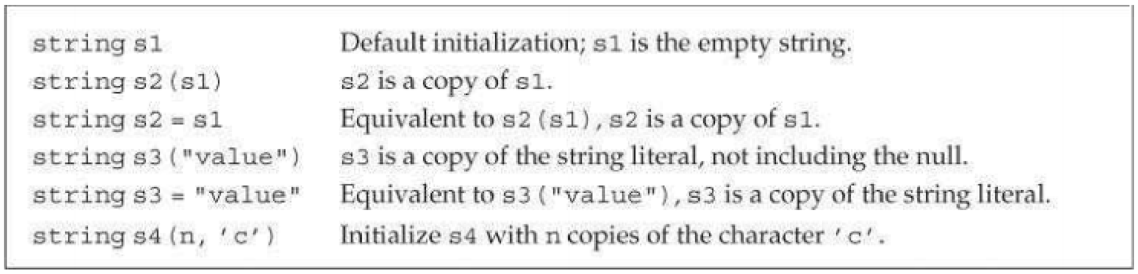
3.2 Library string Type

A string is a variable-length sequence of characters. To use the string type, we must include the string header. string is defined in the std namespace.

3.2.1 Defining and Initializing strings

The following table shows different ways of defining a string.

Table 3.1. Ways to Initialize a string



`string s1`	Default initialization; `s1` is the empty string.
`string s2(s1)`	`s2` is a copy of `s1`.
`string s2 = s1`	Equivalent to `s2(s1)`, `s2` is a copy of `s1`.
`string s3("value")`	`s3` is a copy of the string literal, not including the null.
`string s3 = "value"`	Equivalent to `s3("value")`, `s3` is a copy of the string literal.
`string s4(n, 'c')`	Initialize `s4` with `n` copies of the character `'c'`.

```
string s1; //default initialization; s1 is the empty string
string s2 = s1; //s2 is a copy of s1
string s3 = "Hiya"; //Equivalent to s3("Hiya")
string s4(10, 'c'); //Initialize s4 with n copies of the character 'c'.
```

Direct and Copy Forms of Initialization

When we initializae a variable using `=`, we are asking the compiler to copy initialize the object by copying the initializer on the right-hand side into the object being created. Otherwise, when we omit the =, we use direct initialization.

When we have a single initializer, we can use either the direct or copy form of initialization. When we initialize a variable from more than one value, such as in the initialization of s4 above, we must use the direct form of initialization.

When we want to use several values, we can indirectly use the copy form of initialization by explicitly creating a (temporary) object to copy:

```
string s8 = string(10, 'c'); //copy initialization
```

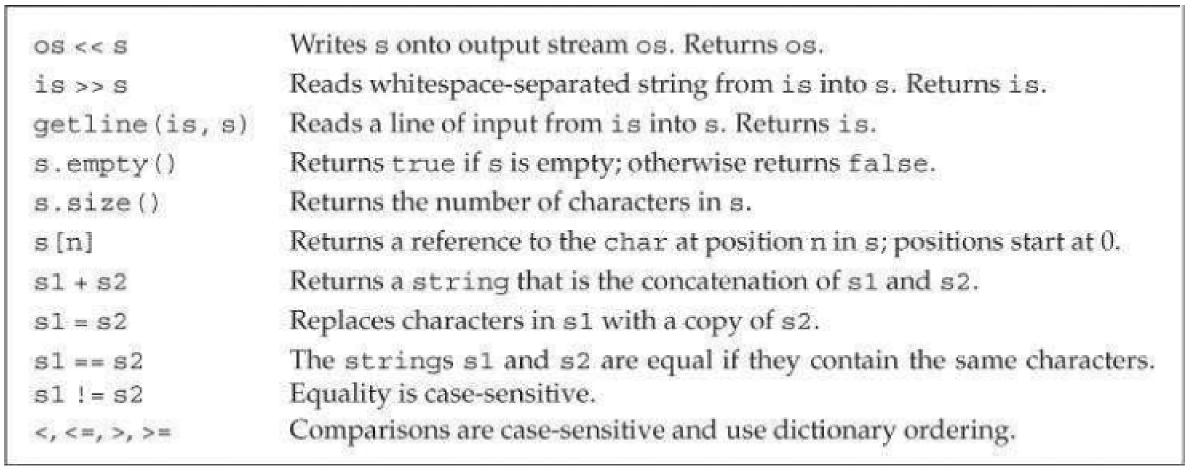
The initializer of `s8` creates a string of the given size and character value and then copies that value into s8. It is as if we had written:

```
string temp(10, 'c');
string s8 = temp;
```

Although the code sued to initialize s8 is legal, it is less readable and offers no compensating advantage over the way we initialized s7.

3.2.2 Operations on strings

The following table lists the most common string operations:



`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`
`s1 != s2`
	The `strings` `s1` and `s2` are equal if they contain the same characters. Equality is case-sensitive.

`<, <=, >, >=`	Comparisons are case-sensitive and use dictionary ordering.

reading an Unknown Number of strings

```
#include <iostream>
using std::cin;
using std::cout;
using std::string;
using std::endl;

int main () {
  string str;
  while(cin >> str)
    cout << str << endl;
  return 0;
}
```

In this program, we read into a string. The condition tests the stream after the read completes. If the stream is valid, meaning it hasn't hit end-of-file or encountered an invalid input, then the body of the while is executed.

Note: cin will ignore the whitespace in between and will only copy the characters in front of the whitespace into the string.