【C++】Day eight(2)

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

【Ch3】String Type

Using getline to Read an Entire Line

Sometime we do not want to ignore the whitespace in our input. In such cases, we can use the `getline` function instead of the `>>` operator. The `getline` function takes an input stream and a string.

This function reads the given stream up to and including the first newline and stores what it read-not including the newline-in its string argument. After `getline` sees a newline, even if it is the first character in the input, it stops reading and returns. If the first character in the input is a newline, then the resulting string is the empty string.

Like the input operator, getline returns its istream argument.

Rewrite the previous program:

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

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

The string empty and size Operations

The `empty` function does what one would expect: It returns a bool indicating whether the string is empty. To call this function, we use the dot operator to specify the object on which we want to run the empty function.

We can revise the previous program to only print lines that are not empty:

```
string line;
//read input a line at a time and discard blank lines
while(getline(cin, line)) {
  if(!line.empty())
    cout << line << endl;
}
```

The `size` member returns the length of a string. We can use size to print only lines longer than 5 characters:

```
string line;
while(getline(cin, line)) {
  if(line.size() > 5)
    cout << line << endl;
}
```

The string::size_type Type

It might be logical to expect that size function returns an int or an unsigned. Instead, size returns a `string::size_type` value.

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 machine-independent manner. The type `size_type` is one of these companion types. To use the `size_type` defined by string, we use the scope operator to say that the name `size_type` is defined in the string class.

Although we don't know the precise type of `string::size_type`, we do know that it is an unsigned type big enough to hold the size of any string. Any variable used to store the result from the string size operation should be of type `string::size_type`.

We can ask the compiler to provide the appropriate type by using `auto` or `decltype` to avoid explicitly defining a `string::size_type` variable:

```
auto strLen = str.size(); //strLen has type string::size_type
```

Comparing strings

The string class defines several operators that compare strings. These operators work by comparing the characters of the strings. The comparisons are case-sensitive—upper and lowercase versions of a letter are different characters.

The equality operators(`==` and `!=`) test whether two strings are equal or unequal, respectively.

Two strings are equal if they are the same length and contain the same characters.

The relational operators `<`, `≤`, `>`, `≥` test whether one string is less than, less than or equal to, greater than, or greater than or equal to another.

1. If two strings have different lengths and if every character in the shorter string is equal to the corresponding character of the longer string, then the shorter string is less than the longer one.
2. If any characters at corresponding positions in the two strings differ, then the result of the string comparison is the result of comparing the first character at which the strings differ.

See the following code for an example:

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

1. By the first rule, str is less than phrase
2. By the second rule, phrase is less than slang.

Adding Two strings

Adding two strings yields a new string that is the concatenation of the left-hand followed by the right-hand operand. That is, when we use the plus operator `+` on strings, the result is a new string whose characters are a copy of those in the left-hand operand followed by those from the right-hand operand.

The compound assignment operator `+=` appends the right-hand operand to the left-hand string:

```
string s1 = "hello", s2 = "world\n";
string s3 = s1 + s2; //s3 is hello, world\n
s1 += s2; //equivalent to s1 = s1 + s2
```

When we mix strings and string or character literals, at least one operand to each + operator must be of string type:

```
string s4 = s1 + ", "; //ok: adding a string and a literal
string s5 = "hello" + ", "; //error:no string operand
string s7 = "hello" + ", " + s2; //error: can't add string literals
```

The initialization of s5 and s7 does not involve a string operand. Thus, it would return an error.