【C++】Day16

Class	C++
Date	@December 6, 2021
Material	
Series Number	
Summary	Argument Passing

【Ch6】Functions

6.2 Argument Passing

As we've seen, each time we call a function, its parameters are created and initialized by the arguments passed in the call.

Note: Parameter initialization works the same way as variable initialization.

The type of a parameter determines the interaction between the parameter and its arguments. If the parameter is a reference, then the parameter is bound to its argument. Otherwise, the argument's value is copied.

When a parameter is a reference, we say that its corresponding argument is "passed by reference" or that the function is "called by reference."

When the argument value is copied, the parameter and argument are independent objects. We say such arguments are "passed by value" or alternatively that the function is "called by value."

6.2.1 Passing Arguments by Value

When we initialize a nonreference type varaible, the value of the initializer is copied. Changes made to the variable have no effect on the initializer:

```
int n = 0; //ordinary variable of type int
int i = n; //i is a copy of hte value in n
i = 42; //value in i is changed, n is unchanged
```

Passing an argument by value works eaxctly the same way; nothing the function does to the parameter can affect the argument.

Pointer Parameters

Pointers behave like any other nonreference type. When we copy a pointer, the value of the pointer is copied. After the copy, the two pointers are distinct.

However, a pointer also gives us indrirect access to the object to which that pointer opints. We can change the value of that object by assigning through the pointer

Best Practices: Programmers accustomed to programming in C often use pointer parameters to access objects outside a function. In C++, programmers generally use reference parameters instead.

6.2.2 Passing Arguments by Reference

Recall that operations on a reference are actually operations on the object to which the reference refers.

```
void reset(int &i) {
  i = 100; //changes the value of the object to which i refers
}
```

Using References to Avoid Copies

It can be inefficient to copy object of large class types or large containers. Moreoever, some class types(including the IO types) cannot be copied. Functions must use reference parameters to operate on object of a type that cannot be copied.

As an example, we''ll write a function to compare the length of tw ostrings. Because strings can be long, we'd like to avoid copying them, so we'll make our parameters references. Because comparing two strings does not involve changing the strings, we'll make the parameters references to const:

```
bool isShorter(const string &s1, const string &s2) {
  return s1.size() < s2.size();
}
```

Functions should use references to const for reference parameters they do not need to change.

Best Practices: Reference parameters that are not changed inside a function should be refernences to const.

Using Reference Parameters to Return Additional Information

A function can return only a single value. However, sometimes a function has more than one value to return. Reference parameters let us effecitvely return multiple results.

Consider the following example:

```
// returns the index of the first occurence of c in s
// the reference parameter occurs counts how often c occurs
string::size_type find_char(const string &s, char c, string::size_type &occurs) {
  auto ret = s.size();
  occurs = 0;
  for(decltype(ret) i = 0; i != s.size(); ++i) {
    if(s[i] == c) {
      if(ret == s.size())
        ret = i; //remember the first occurence of c
      ++occurs; //increment the occurrence count
    }
  }
  return ret; //count is returned implicityly in occurs
}
```