【C++】Day66

Class	C++
Date	@March 1, 2022
Material	
Series Number	
Summary	

【Ch14】Overloaded Operations and Conversions

14.7 Member Access Operators

The dereference(`*`) and arrow(`->`) operators are often used in classes that represent iterators and in smart pointer classes.

We can logically add these operators to our `StrBlobPtr` class:

```
class StrBlobPtr {
public:
  std::string &operator*() const {
    check(curr, "dereference past end");
    return (*p)[curr];
  }

  std::string *operator->() const {
    return &this->operator*();
  }
}
```

The arrow operator avoids doing any work of its own by calling the dereference operator and returning the address of the element returned by that operator.

Note: Operator arrow must be a member. The dereference operator is not required to be a member but usually should be a member as well.

We can use these operators the same way that we've used the corresponding operations on pointers or vector iterators:

```
StrBlob a1 = { "hi", "bye", "now" };
StrBlobPtr p(a1);
*p = "okay";
cout << p->size() << endl;
cout << (*p).size() << endl;
```

Note: The overloaded arrow operator must return either a pointer to a class type or an object of a class type that defines its own operator arrow.

14.8 Function-Call Operator

Classes that overload the call operator allow objects of its type to be used as if they were a function. Because such classes can also store state, they can be more flexible than ordinary functions.

The following struct, named `absInt`, has a call operator that returns the absolute value of its argument:

```
struct absInt {
  int operator()(int val) const {
    return val < 0 ? -val : val;
  }
};
```

We use the call operator by applying an argument list to an `absInt` object in a way that looks like a function call.

```
int i = -42;
absInt absObj;
int result = absObj(i); // passes i to absObj.operator()
```

Note: The function-call operator must be a member function. A class may define multiple versions of the call operator, each of which must differ as to the number or types of their parameters.

Objects of classes that define the call operator are referred to as function objects. Such objects “act like functions” because we can call them.

Function-Object Classes with State

Like any other class, a function-object class can have additional members aside from `operator()`. Function-object classes often contain data members that are used to customize the operations in the call operator.

As an example, we’ll define a class that prints a string argument. By default, our class will write to `cout` and will print a space following each string. We’ll also let users of our class provide a different stream on which to write and provide a different separator.

```
class PrintString {
public:
  PrintString(ostream &o = cout, char c = ' ') : os(o), sep(c) {}
  void operator()(const string &s) const { os << s << sep; }

private:
  ostream &os; // stream on which to write
  char sep; // character to print after each output
};
```

When we define `PrintString` objects, we can use the defaults or supply our own values for the separator or output stream:

```
PrintString printer; // uses the defaults;  print to cout
printer(s); // prints s followed by a space on cout
PrintString errors(cerr, '\n');
errors(s); // prints s followed by a newline on cerr
```

Function objects are most often used as arguments to the generic algorithms. For example, we can use the library `for_each` algorithm and our `PrintString` class to print the contents of a container.

```
for_each(vs.begin(), vs.end(), PrintString(cerr, '\n'));
```

The third argument to `for_each` is a temporary object of type `PrintString` that we initialize from `cerr` and a newline character.

Exercise

Exercise 14.34: Define a function-object class to perform an if-then-else operation: The call operator for this class should take three parameters. It should test its first parameter and if that test succeeds, it should return its second parameter; otherwise, it should return its third parameter.

See 14_34.cpp for code

Exercise 14.35: Write a class like `PrintString` that reads a line of input from an `istream` and returns a `string` representing what was read. If the read fails, return the empty `string`.

See 14_35.cpp