# 【C++】Day45

| | |
|---|---|
| Class | C++ |
| Date | @January 25, 2022 |
| Material | |
| Series Number | |
| Summary | |

## 【Ch11】Associative Container

### 11.2.2 Requirements on Key Type

*Using a Comparison Function for the Key Type*

The type of the operation that a container uses to organize its elements is part of the type of that container.

To specify our own operation, we must supply the type of that operation when we define the type of an associative container.

For example, we can't directly define a `multiset` of `Sales_data` because `Sales_data` doesn't have a `<` operator. However, we can use the `compareIsbn` function to define a multiset.

```
bool compareIsbn(const Sales_data &item1, const Sales_data &item2) {
  return item1.isbn() < item2.isbn();
}
```

To use our own operation, we must define the `multiset` with two types: the key type, `Sales_data`, and the comparison type, which is a function pointer type that can point to `compareIsbn`.

When we define objects of this type, we supply a pointer to the operation we intend to use. In this case,  we supply a pointer to `compareIsbn`:

```
multiset<Sales_data, decltype<compareIsbn>*> bookstore(compareIsbn);
```

Remember that when we use `decltype` to form a function pointer, we must add a `*` to indicate that we're using a pointer to the given function type.

*Exercise*

**Exercise 11.10: Could we define a `map` from `vector<int>::iterator` to `int`? What about from `list<int>::iterator` to `int`? In each case, if not, why not?**

We can define `map<vector<int>::iterator, int>` because `vector<int>::iterator` supports `<`.

However, `list<int>::iterator` does not support the `<` operator and thus cannot be used as the key type.

**Exercise 11.11: Redefine `bookstore` without using `decltype`.**

```
multiset<Sales_data, bool (*)(const Sales_data&, const Sales_data&)> bookstore(compareIsbn);
```

### 11.2.3 The pair Type

We know need to learn about the library type named `pair`, which is defined in the `utility` header.

A `pair` holds two data members. Like the containers, `pair` is a template from which we generate specific types. We must supply two type names when we create a pair.

The data members of the pair have the corresponding types. There is no requirement that the two types be the same:

```
pair<string, string> anon; //holds two strings
pair<string, size_t> word_count; //holds a string and an size_t
pair<string, vector<int>> line; //holds string and vector<int>
```

The default pair constructor value initializes the data members.

We can also provide initializers for each member:

```
pair<string, string> author{"James", "Joyce"};
```

Unlike other library types, the data members of pair are public. These members are named `first` and `second`, respectively. We access these members using the normal member access notation.

```
std::cout << w.first << " : " << w.second << std::endl;
```

Here, `w` is a reference to an element in a map.

The library defines only a limited number of operations on pairs, listed below:

**Table 11.2. Operations on pairs**

| | |
|---|---|
| `pair<T1, T2> p;` | `p` is a `pair` with value initialized (§ 3.3.1, p. 98) members of types `T1` and `T2`, respectively. |
| `pair<T1, T2> p(v1, v2);` | `p` is a `pair` with types `T1` and `T2`; the `first` and `second` members are initialized from `v1` and `v2`, respectively. |
| `pair<T1, T2> p = {v1, v2};` | Equivalent to `p(v1, v2)`. |
| `make_pair(v1, v2)` | Returns a `pair` initialized from `v1` and `v2`. The type of the `pair` is inferred from the types of `v1` and `v2`. |
| `p.first` | Returns the (`public`) data member of `p` named `first`. |
| `p.second` | Returns the (`public`) data member of `p` named `second`. |
| `p1` *relop* `p2` | Relational operators (`<`, `>`, `<=`, `>=`). Relational operators are defined as dictionary ordering: For example, `p1 < p2` is `true` if `p1.first < p2.first` or if `!(p2.first < p1.first) && p1.second < p2.second`. Uses the element's `<` operator. |
| `p1 == p2`<br>`p1 != p2` | Two `pair`s are equal if their `first` and `second` members are respectively equal. Uses the element's `==` operator. |

### *A Function to Create pair Objects*

Imagine we have a function that needs to return a `pair`. Under the new standard we can list initialize the return value:

```
pair<string, int> process(vector<int> &v) {
  if(v.empty())
    return pair<string, int>(); //explicitly constructed return value.
  return pair<string, int> {v.back(), v.back().size()};
}
```

Alternatively, we could have used `make_pair` to generate a new pair of the appropriate type from its two arguments:

```
return make_pair(v.back(), v.back().size());
```

### *Exercise*

**Exercise 11.12: Write a program to read a sequence of `string`s and `int`s, storing each into a `pair`. Store the `pair`s in a `vector`.**

See 11_12.cpp for code

**Exercise 11.13: There are at least three ways to create the `pair`s in the program for the previous exercise. Write three versions of that program, creating the `pair`s in each way. Explain which form you think is easiest to write and understand, and why.**

See 11_13.cpp for code