【C++】Day79

Class	C++
Date	@April 4, 2022
Material	
Series Number	
Summary	

【Ch16】Templates and Generic Programming

The containers, iterators, and algorithms are all examples of generic programming.

When we write a generic program, we write the code in a way that is independent of any particular type.

Templates are the foundation of generic programming. We can use and have used templates without understanding how they are defined.

16.1 Defining a Template

If we want to compare two values, we might define two functions as below:

```
int compare(const int &v1, const int &v2) {
  if(v1 < v2)
    return -1;
  else if(v1 > v2)
    return 1;
  return 0;
}

int compare(const double &v1, const double &v2) {
  if(v1 < v2)
    return -1;
  else if(v1 > v2)
    return 1;
  return 0;
}
```

These functions are nearly identical: The only difference is the type of their parameters.

16.1.1 Function Templates

Rather than defining a new function for each type, we can define a function template. A function template is a formula from which we can generate type-specific versions of that function.

The template version of `compare` looks like

```
template <typename T>
int compare(const T &v1, const T &v2) {
  if(v1 < v2)
    return -1;
  else if(v2 < v1)
    return 1;
  return 0;
}
```

A template definition starts with the keyword template followed by a template parameter list, which is a comma-separated list of one or more template parameters bracketed by the less-than(<) and greater-than(>) tokens.

Note: In a template definition, the template parameter list cannot be empty.

Instantiating a Function Template

When we call a function template, the compiler uses the arguments of the call to deduce the template arguments for us.

When we call `compare`, the compiler uses the type of the arguments to determine what type to bind to the template parameter T.

For example, in this call:

```
cout << compare(1, 0) << endl; // T is int
```

the arguments have type `int`. The compiler will deduce `int` as the template argument and will bind that argument to the template parameter T.

The compiler uses the deduced template parameters to instantiate a specific version of the function for us.

When the compiler instantiates a template, it creates a new “instance” of the template using the actual template arguments in place of the corresponding template parameters. For example, given the calls

```
// instantiates int compare(const int&, const int&)
cout << compare(1, 0) << endl;
// instantiates int compare(const vector<int>&, const vector<int>&)
vector<int> vec1{1, 2, 3}, vec2{4, 5, 6};
cout << compare(vec1, vec2) << endl; // T is vector<int>
```

the compiler will instantiate two different versions of `compare`.

Template Type Parameters

Our compare function has one template type parameter. In general, we can use a type parameter as a type specifier in the same way that we use a built-in or class type specifier.

In particular, a type parameter can be used to name the return type or a function parameter type, and for variable declarations or casts inside the function body:

```
// ok: same type used for the return tyep and parameter
template <typename T> T foo(T* p) {
  T tmp = *p;
  // ...
  return temp;
}
```

Each type parameter must be preceded by the keyword `class` or `typename`:

```
template <typename T, U> T clac(const T&, const U&)
```

A template parameter list can use both keywords:

```
// ok: no distinction between typename and class in a template parameter list
template <typename T, class U> calc (const T&, const U&);
```

Nontype Template

We can define templates that take nontype parameters. A nontype parameter represents a value rather than a type. Nontype parameters are specified by using a specific type name instead of the `class` or `typename` keyword.

When the template is instantiated, nontype parameters are replaced with a value supplied by the user or deduced by the compiler.

As an example, we can write a version of compare that will handle string literals. Such literals are arrays of const char.

```
template<unsigned N, unsigned M>
int compare(const char (&p1)[N], const char (&p2)[M]) {
  return strcmp(p1, p2);
}
```

Note: Templates arguments used for nontype template parameters must be constant expressions.