سلسلة تعلم البرمجة بلغة C++ الحديثة

Learn Modern C++ Programming Series

إعداد المهندس أحمد الديب

#42: Instantiation P1

Template Instantiation

- From a class template and a set of template arguments, the compiler needs to generate the definition of a class and the definitions of those of its member functions that were used in the program.
- From a template function and a set of template arguments, a function needs to be generated.
- The generated classes and functions are called specializations. When we need to distinguish between generated specializations and specializations explicitly written by the programmer, we refer to generated specializations and explicit specializations (user-defined specialization), respectively.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

When Is Instantiation Needed?

```
template <typename T>
class Node;


Node<int>* n1;  // no instantiation of Node<int> needed


template <typename T>
class Node {
  Node* next;  // OK: no definition of Node needed
  // ...
};


Node<int> n2;  // now we need to instantiate Node<int>
```

It is necessary to generate a specialization of a class template only if the class's definition is needed. In particular, to declare a pointer to some class, the actual definition of a class is not needed.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

When Is Instantiation Needed?

```
template <typename T>
class List {
 public:
  void sort() { op.sort(); }
  void print() { op.print(); }


 private:
  T op;
};

int main() {
  List<Sorter> ls;
  List<Printer> lp;


  ls.sort();
  lp.print();
}
```

```
class Sorter {
 public:
  void sort() {  //
    std::cout << "Sorting...\n";
  }
};

class Printer {
 public:
  void print() {  //
    std::cout << "Printing...\n";
  }
};
```

An implementation instantiates a template function only if that function has been called or had its address taken. In particular, instantiation of a class template does not imply the instantiation of all of its member functions.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

When Is Instantiation Needed?

```
// Explicit instantiation
template class List<Sorter>;
// error: no member named 'print' in 'Sorter'
```

Note: When a class template is explicitly instantiated,
every member function is also instantiated.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Dependencies

```
bool print;

template <typename T>
T sum(std::vector<T>& vec) {
  T sum{0};
  for (auto& item : vec) {
    sum += item;
  }

  if (print) {
    std::cout << "sum = " << sum << std::endl;
  }

  return sum;
}

int main() {
  std::vector<int> myVec{1, 2, 3};
  print = true;
  sum(myVec);
}
```

Define template functions to minimize dependencies on non-local information. The reason is that a template will be used to generate functions and classes based on unknown types and in unknown contexts.

We try to make template definitions as self-contained as possible and to supply much of what would otherwise have been global context in the form of template arguments.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Name Binding

- The process of finding the declaration for each name explicitly or implicitly used in a template is called name binding.
- **Name Categories:**
  - Dependent names: bound at a point of instantiation.
  - Non-dependent names: bound at the point of definition of the template.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Dependent Names

```
template <typename T>
T foo(T a) {
  return bar(a);  // bar is a dependent name.
}


struct Point {
  int x, y;
};


Point bar(Point x) { return x; }


int main() {
  Point z = foo(Point{2, 3});  // point of instantiation of foo
}
```

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Non-Dependent Names

```
struct Point {
  int x, y;
};


Point bar(Point x) { return x; }


template <typename T>
T foo(T a) {
  return bar(Point{2, 3});  // bar is not a dependent name.
}


int main() {
  Point z = foo(Point{2, 3});  // point of instantiation of foo
}
```

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Dependent Names

```
template <typename T>
auto sum(T& con) -> typename T::value_type {
  typename T::value_type sum{};
  for (auto& item : con) sum += item;
  return sum;
}


int main() {
  std::vector<int> vec{1, 2, 3, 4, 5};
  std::cout << sum(vec) << std::endl;
  return 0;
}
```

By default, a dependent name is assumed to name something that is not a type. So, to use a dependent name as a type, you have to say so, using the keyword typename. We can avoid such awkward use of typename by introducing a type alias.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Dependent Names

```
template <typename T>
using Value_type = typename T::value_type;


template <typename T>
Value_type<T> sum(T& con) {
  Value_type<T> sum{};
  for (auto& item : con) sum += item;
  return sum;
}


int main() {
  std::vector<int> vec{1, 2, 3, 4, 5};
  std::cout << sum(vec) << std::endl;
  return 0;
}
```

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Dependent Names

```
template <typename T>
class Node {
 public:
  Node(Node* next, T data) : next{next}, data{data} {}
  Node* next;
  T data;
};

template <typename Iterator, typename T>
void print_data(Iterator& iter) {
  std::cout << iter.template getData<T>() << std::endl;
  // without template the compiler will give error:
  // use 'template' keyword to treat 'getData' as
  // a dependent template name
}


int main() {
  Node<int> n1{nullptr, 1};
  NodeIter iter{&n1};
  print_data<NodeIter<int>, int>(iter);


  return 0;
}
```

```
template <typename T>
class NodeIter {
 public:
  NodeIter(Node<T>* node) : _node{node} {}
  template <typename U>
  U getData() const {
    return _node->data;
  }

 private:
  Node<T>* _node;
};
```

Naming a member template after a . (dot), -->, or :: requires similar use of the keyword template.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Thank you