Templates of templates

Posted on August 21, 2014 by Glennan Carnie

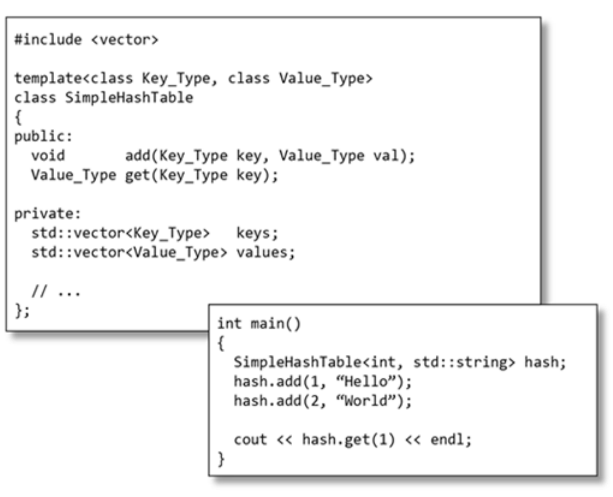
Introduction

In this brief article we'll have a look at the topic of building template classes whose parameters are themselves templates.

I'm assuming you're reasonably familiar with template classes. If not, here's a quick introduction.

The problem

Let's start with a case study. Suppose we want to build a simple hash table class. Our class should be generic for different types of keys and values*. Obviously, it should be a template class.

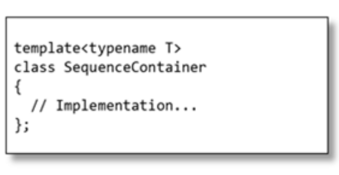


```
#include <vector>

template<class Key_Type, class Value_Type>
class SimpleHashTable
{
public:
  void       add(Key_Type key, Value_Type val);
  Value_Type get(Key_Type key);

private:
  std::vector<Key_Type>   keys;
  std::vector<Value_Type> values;

  // ...
};
```

```
int main()
{
  SimpleHashTable<int, std::string> hash;
  hash.add(1, "Hello");
  hash.add(2, "World");

  cout << hash.get(1) << endl;
}
```

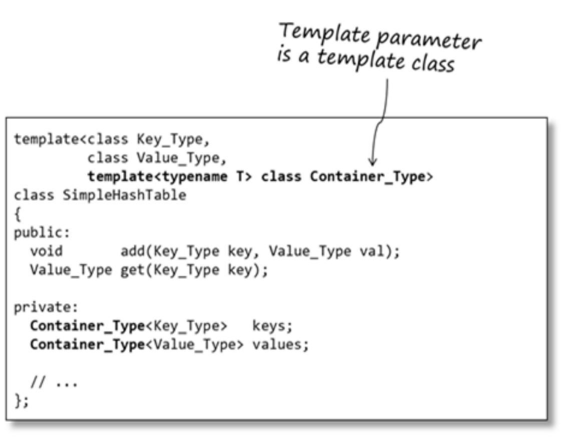
We might want to add more flexibility to our design by allowing the client to choose the type of container class used by the `SimpleHashTable` to store its keys and values (for example, for performance reasons we may want to use a fixed-size container)

We need to add the container class as a template parameter. However, a container class is typically a template class itself.



```
template<typename T>
class SequenceContainer
{
  // Implementation...
};
```

Let's modify the `SimpleHashTable` to allow the container type to be specified:

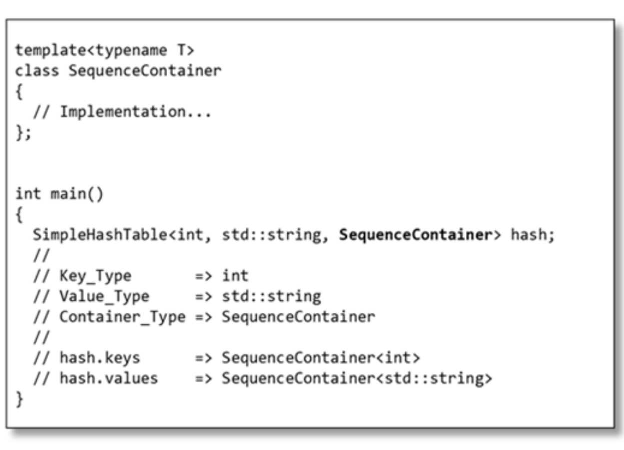


```
template<class Key_Type,
         class Value_Type,
         template<typename T> class Container_Type>
class SimpleHashTable
{
public:
  void       add(Key_Type key, Value_Type val);
  Value_Type get(Key_Type key);

private:
  Container_Type<Key_Type>   keys;
  Container_Type<Value_Type> values;

  // ...
};
```

We must tell the compiler that the template parameter `Container_Type` is itself a template class. Notice that the template parameter for the template class must not match any of the other identifiers in the template list.

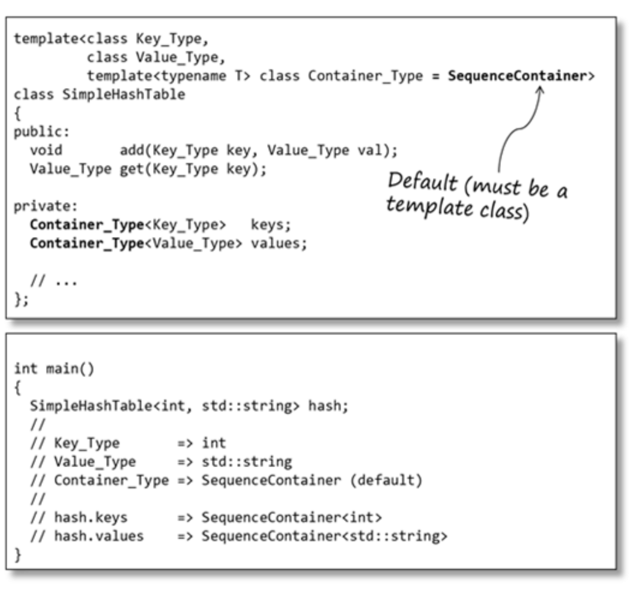
The `Container_Type` parameter is used to instantiate a new container within the `SimpleHashTable`, whose template parameter is one of the template parameters supplied to the `SimpleHashTable`. In other words, there is a 'cascading' of template parameters within the class: a parameter supplied to the owning template is used to instantiate a nested template class within.

Let's have a look at what gets created when we instantiate our `SimpleHashtable` class:



```
template<typename T>
class SequenceContainer
{
  // Implementation...
};


int main()
{
  SimpleHashTable<int, std::string, SequenceContainer> hash;
  //
  // Key_Type       => int
  // Value_Type     => std::string
  // Container_Type => SequenceContainer
  //
  // hash.keys      => SequenceContainer<int>
  // hash.values    => SequenceContainer<std::string>
}
```

As with all template parameters, template-template parameters can be defaulted (the usual rules for defaults apply). Note, however, that the template default must be a template class.



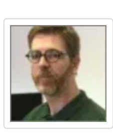
```
template<class Key_Type,
         class Value_Type,
         template<typename T> class Container_Type = SequenceContainer>
class SimpleHashTable
{
public:
  void       add(Key_Type key, Value_Type val);
  Value_Type get(Key_Type key);

private:
  Container_Type<Key_Type>   keys;
  Container_Type<Value_Type> values;

  // ...
};
```

```
int main()
{
  SimpleHashTable<int, std::string> hash;
  //
  // Key_Type       => int
  // Value_Type     => std::string
  // Container_Type => SequenceContainer (default)
  //
  // hash.keys      => SequenceContainer<int>
  // hash.values    => SequenceContainer<std::string>
}
```

Summary

Once we start building template classes it becomes natural to incorporate them into other, more complex, template classes. The aim is to build our generic code with as much flexibility as possible.

**This is a deliberately artificial example. You wouldn't implement a hash table this way, typically; and you don't need to since one is supplied with the Standard Library.*



Glennan Carnie
Technical Consultant at Feabhas Ltd

Glennan is an embedded systems and software engineer with over 20 years experience, mostly in high-integrity systems for the defence and aerospace industry.

He specialises in C++, UML, software modelling, Systems Engineering and process development.

Like (8) Dislike (0)

Glennan Carnie

Website | + posts

Glennan is an embedded systems and software engineer with over 20 years experience, mostly in high-integrity systems for the defence and aerospace industry.

He specialises in C++, UML, software modelling, Systems Engineering and process development.

This entry was posted in C/C++ Programming and tagged C++, composition, containers, Templates, templates of templates. Bookmark the permalink.

10 Responses to *Templates of templates*

EnglishBob *says:*
August 27, 2014 at 3:50 am

Is the use of 'class', rather than 'typename' intentional in these examples? In every other article you exclusively use typename.

Great set of articles btw.

Like (1) Dislike (1)

Olumide *says:*
September 5, 2014 at 12:50 am

class is used because template template parameters can only be classes. This is not just for emphasis but is the rule.

Like (1) Dislike (0)

CppWut *says:*
September 9, 2014 at 10:04 am

Yes, this is the rule, but it should be possible to use both template and class keywords in c++17.

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4051.html

Like (1) Dislike (0)

Leandro *says:*
October 3, 2014 at 12:55 am

In your example "Template parameter is a template class" should actually be written as "Template parameter is a class template". This slight inversion of order makes a total difference (check the C++ Standard for example). Your other example "Default (must be a template class) is correct, since in this case you're really pointing to a instantiation of a class template.

Like (1) Dislike (0)

ArtemT *says:*
October 3, 2014 at 3:18 am

Wanted to point out that using template of template here is not only useless, but is unnecessarily restrictive. In your case simple "class Container_Type" would do. Otherwise container is restricted to a class with a single argument, not more, not less.

There surely are cases where template of template is useful and even necessary, but this example as it is written is not one of them)

Like (0) Dislike (3)

Pingback: *C++ Template of template | Heresy's Space*

Kranti Madineni *says:*
April 23, 2018 at 8:10 am

Another tough concept explained nicely without hiccups..

Like (2) Dislike (0)

Pingback: *Overload operator for both std::vector and std::list | Developer FAQs*

GM *says:*
June 19, 2019 at 12:57 am

Wondering about debugging (which is already difficult on embedded systems)...do we loose any flexibility in later stages, such as debugging?

Like (0) Dislike (0)

Pingback: *How to do compile-time recursion over a given set of template template classes? – Windows Questions*