Templates in C++

Stefan Micheelsen
smiche11@student.aau.dk
Studentnumber: 20112707

1. April 2015

This report will describe my work on template programming in C++. The main focus will be a comparison with the experience I have with generics in C#. A lot of the work in the report will be experiments with features that are not present in the C# language and are therefore not known to me. There is no main program as such, but there are several methods in the *"Eksamensprojekt.cpp"* that explores the classes and templates that have been created.

1 Templates in C++

Templates is the C++ implementation of what is called generic programming. Generic programming is a technique that aims a making it possible to implement general algorithms. [1, p. 700].

Without generic programming it would be necessary to write a variation of a list for each data type you would want to put in the list. Generic programming provides a tool to provide functionality for each data type, without having to write each of them by hand.

In C# generic types are compiled to a annotated version in the intermediate code, and the specialized type is generated in the runtime when it is needed[2]. On the other hand templates in C++ are generated from the principle of duck typing, meaning that the use of a type parameter determines the choice of implementation and thus the behavior of a method.[1, p. 700].

2 Overview

In order to experiment with templates in C++ I choose to implement two very simple data structures, the stack and the binary search tree. Because the emphasis is on templates and not the workings of these data structures, not all features are implemented, only what is necessary in order to demonstrate templates and the features templates provide. The implementations are taken from Cormen et al. [3].

3 The first template

When designing a template Bjarne Stroustrup recommends starting out by implementing the class without type parameters[1, p. 670]. I therefore started out by investigating the stack data structure and implemented it to be a stack of integers. The initial implementation can be found in *"StackOfInt.h"* and *"StackOfInt.cpp"* . The implementation follows the implementation in Cormen et al. [3, 232].

Looking at this relatively simple class it should be a simple job to use a type parameter instead of the hard coded integer type.

The implementation of the stack with type parameters can be seen in *"StackTemplate.h"* . The method `Push(T i)` is shown below.

```
1 template<typename T>
2 void StackTemplate<T>::Push(T i) )
3 {
4   topIndex++;
5   arr[topIndex] = i;
6 }
```

The line `template<typename T>` tells the C++ compiler that the following is a template where all instances of `T` should be changed with the specified type in any template instantiation. In the implementation of the Push method the `T` can be seen used in line 2 where the `StackTemplate<T>` specifies that this implementation of `Push` belongs to the template of the stack containing one type parameter. The T used as the argument specifies that the argument needs to be of the same type as the type used for the stack.

With the first version of the template of a stack in place, the process of "lifting" can continue. Lifting is the act of taking an implementation from a concrete level to a general level while maintaining high performance[1, p. 700].

4 Template arguments

The basic use of templates is to extend an implementation with type parameters. Templates can also use other parameters, expanding the possibilities of templates [1, p. 722]. One of these possibilities is to use a value as a template parameter. This can be used to define sizes of some aspect of the template. [1, p. 724]

In the initial implementation of the stack I chose not to make it dynamically expand when adding elements. This means that its size is fixed to 16 elements. This is of course inefficient if you only insert 2 elements, and insufficient if you need to insert 20 elements. A possible solution is to use a value parameter to specify how big the stack needs to be.

```
1 template<typename T, int entries>
```

```
StackTemplateValueArgument<T, entries>::
    StackTemplateValueArgument() : topIndex{ -1 }
{
  arr = new T[entries];
}
```

The implementation of the stack with value parameters can be found in *"StackTemplateValueArgument.h"* . It still uses the `T` as type parameter but now also has an `entries` parameter that specifies the size of the stack. Shown is the constructor where the array is created, where the value parameter is used to create the array of type `T` with `entries` elements

This implementation will be efficient in cases where the user knows how many elements needs to be in the stack.

This feature is not present at all in C# generics which is restricted to only type parameters making it impossible to reach a similar result[4].

5 Specialization

Looking at the templates written for the stack it seems at this point to be a very generic implementation of the simple data structure, but it can be lifted even more.

Consider a stack of integers and a stack of boolean values. In principle the same thing should happen. Push an element on the stack, pop an element, the same thing happens independently of the data type. The problem is that a boolean is stored in a byte even though it can be expressed as a single bit.

With this issue in mind I created a `bool` specialization of the stack. It is contained in *"StackTemplateBoolSpecialization.h"* . Here I implemented the stack to store the boolean values as a single byte instead of in a byte for each.

```
bool StackTemplateBoolSpecialization<bool>::Top()
{
  int positionInRepr = topIndex / 8;
  int positionInByte = topIndex % 8;

  char entry = arr[positionInRepr] <<(positionInByte);
  entry = entry >> 7;
  bool poppedEntry = (bool)entry;

  return poppedEntry;
}
```

Presented is a the `Top()` method from the boolean stack specialization. As this is the boolean specialization of the stack the class name is qualified with `bool`. The current top entry is found by shifting the bits first to the left, removing the bits of the rest of the stack and then 7 to the right for the boolean

value to settle at the rightmost bit, making it possible to read the boolean value from it.

A simple experiment shows that the normal stack implementation with space for 16 boolean take up 20 bytes of space, while the specialization only uses 8 bytes. This experiment can be found in the `StackTemplateBoolSpecializationTest()` test method that can be found in the *"Eksamensprojekt.cpp"* file.

Specialization is a feature in C++ template programming that makes it possible to make specialized implementations for certain arguments. It is possible to partially specialize for a subset of types, like pointers, and it is possible to fully specialize to a type like bool as has been demonstrated in the above [1, p. 730].

Again this is a feature that C# simply does not have. In C# generics you can provide one implementation, and that implementation will be used for every type parameter that will be provided[5].

6 Concepts

In order to explore templates even further I implemented a binary search tree as it is described in Cormen et al. [3, 286]. A binary search tree a still a simple data structure, but it has some requirements of its type arguments. In order to insert elements in the tree it is necessary that the type used as argument can be compared. Otherwise it simply does not make sense to build the tree.

In C# there is a construct in generics that can apply such a constraint on the generic class. In C# it is called `where` and a class declaration like `public class MyGenericClass<T> where T : IComparable` tells the C# compiler that any type used as an argument to `MyGenericClass` needs to implement the `IComparable` interface[6]. This interface contains a `CompareTo` method which can then be used for the comparison in the tree. If the class is instantiated with a type argument that does not satisfy this requirement the compiler will output an error.

In C++ this is not as easy, but it is possible to achieve the same effect. The technique is known as concepts and uses static asserts to make sure that some assertions on the type argument holds. [1, p. 708] In order to make these concepts a small framework had to be set up. This framework is described in Stroustrup [1, p. 800] and implemented in *"Concepts.cpp"* .

This small framework uses the rule known as SFINAE (substitution failure is not an error). The rule is that when the compiler tries to instantiate a template with an argument that would result in a compilation error, that template is simply not used and alternatives are considered[1, p. 692].

The implementation of a concept that requires the less than operator (<) is shown below.

```
1 //using SFINAE to achieve concept
2 template<typename T>
3 struct get_less_than_result
```

```
4  {
5    private:
6    template<typename X>
7    static auto check(X const& x) -> decltype(x < x);
8    static substitution_failure check(...);
9
10   public:
11   using type = decltype(check(std::declval<T>()));
12 };
13
14 //The concept usable in a static assert
15 template<typename T>
16 constexpr bool Has_Less_Than()
17 {
18   return has_less_than<T>::value;
19 }
```

Here the SFINAE rule is used to choose between two implementations of the `check()` method. In line 7 the method is define to have the return type of the expression `x < x` where x is of the template argument type. If this operation results in a compiler error, this definition will simply not be used, and the definition in the line below is used instead. This definition returns the type `substitution_failure`, which has been defined in the framework to have the value false. This concept can now be used to assert if a template argument has the less than operator defined.

```
1 template<typename T>
2 class BinarySearchTree
3 {
4
5   public:
6   static_assert(Has_Less_Than<T>(), "template argument
        type must implement the less than operator, <,
      in order to compare the inserted elements");
7 ...
8 }
```

This assertion is used in the implementation of the binary search tree and can be seen in the code fragment of the binary tree header above.

In order to test this construct I have created two very simple classes, `Data` and `DataIncompatible`. They both hold an integer, and should as such be possible to insert in a binary search tree. The problem with `DataIncompatible` is that it does not contain a definition of the less than operator, which makes it impossible for the C++ compiler to instantiate a template of the tree where the Insert method is meaningful.

Without the assertions just mentioned the Visual Studio C++ compiler gives the following error message

```
1 ''error C2678: binary '<' : no operator found which takes
      a left-hand operand of type 'DataIncompatible' (or
    there is no acceptable conversion)''
```

This error message is in this case not that bad, but that may be because I am the implementer of the tree. If the template gets just a little more complex, and another programmer is using it, this type of error message can very quickly get very hard to debug. Instead I can use my static assertion and create my own error message. As seen in the above code sample, I used the error "template argument type must implement the less than operator, <, in order to compare the inserted elements". I can tell the programmer exactly what is missing and why it is a problem. It is then much easier for him to make an informed decision on how to proceed with his work, without needing to debug deep into a template that he has not implemented.

While the C# `where` construct is easier and faster to set up, the concepts of C++ are not dependent on being a class or interface. The checks made in the concept can be almost anything. The standard library header `<type-traits>` contains predefined predicates such as `is_default_constructible<X>` and `is_assignable<X>`. The possibilities are much broader than just adhering to a class or interface.

7 Calling of arithmetic operators

The last aspect of templates that I want to explore is the final testament that C++ templates are more flexible than the C# counterpart. In C# you rely on the type safety provided by the where clauses used in the declaration. This makes the usage of the generic class safe, but prevents you from using the basic arithmetic operators. In C++ the templates are generated when instantiated, and type checked at that moment. This makes a method as this one legal

```
1 template<typename T>
2 T BinarySearchTree<T>::AddArbitrary(T k)
3 {
4   return root->key + k;
5 }
```

Even though there is no guarantee that this operator is present on the type argument, the code is legal. It will produce an error if instantiated with a type that does not implement the + operator, as discussed above. The techniques used in the previous section can of course be used to statically check for the + operator and to provide more meaningful error messages.

8 Conclusion

In this report I have explored the possibilities of templates in C++. I have consulted the literature in order to find and understand the features that templates provide which C# does not provide in generic types.

Templates and generics are tools that try to achieve many of the same goals. But because of the type safe nature and the fact that it is instantiated at runtime there are some aspects where generics are simply not as powerful as templates.

In the work I have done several very useful features turned up. The ability to use value types as a type parameter proves useful when you need to specify some size or maximum value for the template. Also the ability to specialize proves useful when optimization is an objective. Finally the concepts of C++ is a powerful way of expressing requirements of the type argument.

References

[1] Bjarne Stroustrup. The c++ programming language. 2013.

[2] Generics in the runtime. https://msdn.microsoft.com/en-us/library/f4a6ta2h.aspx. Visited: 2015-03-31.

[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[4] C# generics. https://msdn.microsoft.com/en-us/library/512aeb7t.aspx. Visited: 2015-03-31.

[5] Differences between c++ templates and c# generics. https://msdn.microsoft.com/en-us/library/c6cyy67b.aspx. Visited: 2015-03-31.

[6] where (generic type constraint). https://msdn.microsoft.com/en-us/library/bb384067.aspx. Visited: 2015-03-31.