

Static Metaprogramming in C++



Michael Pisula

University of Passau
Department of Mathematics and Computer Science

Chair for Programming, Prof. Lengauer

Summer course 2006

Contents

1	Introduction	2
2	Programming with templates	2
2.1	Example	3
2.2	Techniques	6
2.2.1	Partial specialization	6
2.2.2	If-Construct	6
2.2.3	Partial evaluation	7
2.2.4	Template template parameters	8
2.2.5	Loop unrolling	9
2.3	Functional elements	9
3	Real-life examples	9
3.1	Loki library	10
3.1.1	Typelists	10
3.1.2	Abstract Factory	12
3.1.3	Loki summary	14
3.2	Other libraries	14
4	Review and Summary	14
A	Code	16
A.1	Factorial example	16
A.1.1	iterative implementation	16
A.1.2	recursive implementation	16
A.1.3	template implementation	16
A.2	partial specialization	16
A.3	if-construct	17
A.4	partial evaluation	17
A.5	Template template parameters	19
A.6	Loop unrolling	19

1 Introduction

Since Bjarne Stroustrup designed C++ in 1979 the language kept evolving. Over the years many new features were added. In this text, we will deal with one of the later additions.

Originally the problem was to create flexible libraries, which would provide code for many different types, without having to specify those types. For this reason *templates* were introduced. Templates were designed to be constructs which could be defined without specifying their parameter and return types. If these templates were used in a program, the programmer would specify the types when making a call to the template and the compiler would insert these types into the template and thus create a normal class or function out of it (there are two types of templates, *class templates* and *function templates*). This was the basic idea behind the concept of templates. In short, templates should add *generic programming* to C++. As it turned out, they did a lot more.

In 1994, at the San Diego Standardization Meeting, one of the committee members, Erwin Unruh, discovered a new way of using templates. He wondered if it would be possible to make calculations using templates. Bjarne Stroustrup did not think this was possible, and so Unruh was eager to prove Stroustrup wrong. After a day of programming he managed to write a small program which would calculate prime numbers and was completely evaluated by the compiler. Unruh even made the compiler return the output by using error messages. Using templates, he found out, it was possible to write programs which would be evaluated at compile-time. Not only that: he proved that programming with templates was Turing-complete, which makes template metaprogramming basically a programming language. Thus C++ is a two-level language: you have template metaprogramming at the static level and the classic C++ at the dynamic level. But Unruh did not realize what he had just discovered; in his opinion this was only an interesting side-effect of templates, without much use.

In 1995 Todd Veldhuizen wrote an article demonstrating the power of template metaprogramming ([3]). He used templates to implement the Blitz++ library. His work showed that it is possible to create complex programs using template metaprogramming. From then on, more and more people began working with *template metaprogramming*.

By now, template metaprogramming slowly emerges from the depths of esoteric programming techniques and is put to use in several libraries.

2 Programming with templates

Programming with templates is a new experience for the common C++ programmer. Template Metaprogramming is very much a programming lan-

guage of its own, and things which are trivial in iterative programs can be very complicated in the world of templates. On the other side, template metaprogramming can provide elegant solutions to problems which would be complicated to solve with regular techniques.

Mainly template programming uses class templates, but we will also see function templates used in some of the example code. However function templates are only suitable for some applications, because they are not completely evaluated by the compiler. Class templates are, as indicated by the name, templates for classes. That means that their syntax is very similar to classes. You can define members (basically the fields and methods of a class), and even say which of them are public, and which are private. As for the members we have to make sure that the compiler can evaluate them statically. Therefore our members will be `enums` and `typedefs`. With `enum` we can assign a certain value to a constant, this assignment is done statically. With `typedef` we assign a type to a constant. Class templates can also define member functions, typically *inline functions* will be used as member functions. The compiler copies the code of functions which are defined as `inline` to the point of the call, if possible. This definition leaves the compiler the freedom to decide if it is better to make the call or copy the code. To access a templates member, we use the `::` operator. This operator has to be between the class and the member we want to call. As the members we will use are constants, changing the value is not possible.

2.1 Example

To demonstrate the use of template programming, we will implement the factorial function with templates. It is fairly simple, yet introduces important techniques of template metaprogramming.

Most C++ programmers would choose an iterative implementation. It is short, fast and easy to implement. But iterative programming does not match with templates. We would need a loop, but there are no build-in loops we could use statically. What you basically do in a loop is to iterate over a variable whose value you increment (or decrement). As you cannot change the value of a member in template metaprogramming once it was assigned, loops are out of question for static metaprogramming in C++ (we will see later that it is possible to create constructs which simulate loops with templates). The answer to this problem is *recursion*, a technique which most C++ programmers will usually shy away from. In template metaprogramming, however, it is the way of choice. The compiler can (and does) resolve recursive calls, because a call instantiates the template with another value, and the compiler has to evaluate all template instantiations.

Next we will need to evaluate a condition, in order to terminate the recursion. As `if` is also dynamically evaluated, we need to find a static way to do this.

Templates were originally created to allow the creation of code which works for several types. But the need may arise to have a different implementation for a single type, for which the general implementation does not work. Because of this, templates can be defined with a specific value for the parameter. This technique is called *template specialization* (or just *specialization*). The compiler will then choose the most specific template it can find for a given template call. Another possibility for conditions is the `?:` operator. It is also evaluated by the compiler, if no variables are involved.

With all problems sorted out, we can implement the factorial function. First we will create the general implementation. Using the normal syntax for templates we define a class template with one `int` parameter and the name `Factorial`. We need to provide our result as a return value, however classes are not meant to return something. But we can define a public member field, and store the result in it. To get the result we recursively calculate the factorial of `n-1` and multiply it by `n`. Templates can only be called with constants, this might seem like a contradiction to the call to `Factorial<N-1>`. However, `N` is a constant in this context; you cannot change a template parameter.

```
template<int N>
class Factorial {
public:
    enum {
        value = N * Factorial<N-1>::value
    };
};
```

The specialization is even easier. We omit the parameter in the template definition and specify it in the class definition. Again we will need to create a public member with the same name as in the general implementation, so that the compiler can find it while evaluating the recursive call. This specialization will only return 1 as result.

```
template<>
class Factorial<0> {
public:
    enum {value = 1 };
};
```

With the function implemented, the question is how to call the function. It is done the same way the recursive call was made. Again, only constants are allowed as parameters. To be precise, we instantiate the class template with the parameter, as it is no function we cannot call it. However, this instantiation will work as a call at compile-time, because the compiler will evaluate the template and calculate the value of the member `value`. As we access the member the calculated value will then be inserted directly into the assembly code.

```
int main() {
    cout << "5! ist " << Factorial<5>::value << endl;
    cout << "7! ist " << Factorial<7>::value << endl;
}
```

Now we can execute the program, the result will be as shown in figure 1.

```
[michi@kopernic HS]$ g++ -o fact fact.cpp
[michi@kopernic HS]$ ./fact
5! ist 120
7! ist 5040
[michi@kopernic HS]$ █
```

Figure 1: Execution of the Factorial code

Obviously the program returns the correct results, but the question arises if the results are computed at compile-time. We can check this, by letting the compiler create only the assembly code, and inspect the assembly file.

```
.LCFI11:
movl    %esp, %ebp
.LCFI12:
pushl   %ecx
.LCFI13:
subl    $36, %esp
.LCFI14:
movl    $120, -8(%ebp)
movl    $.LC0, 4(%esp)
movl    $0, _ZSt4cout, (%esp)
call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
movl    %eax, %edx
movl    -8(%ebp), %eax
movl    %eax, 4(%esp)
movl    %edx, (%esp)
call    _ZNSolsEi
movl    $_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_, 4(%esp)
movl    %eax, (%esp)
call    _ZNSolsEPFRSoSE
movl    $5040, -8(%ebp)
movl    $.LC1, 4(%esp)
movl    $_ZSt4cout, (%esp)
call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
movl    %eax, %edx
movl    -8(%ebp), %eax
movl    %eax, 4(%esp)
-----XEmacs: fact.s (Assembler Font)-----L75--24%
```

Figure 2: Assembly code of fact.cpp

As we see in figure 2, the compiler has computed the result and inserted it into the `cout` operation.

Compared to the few lines of code the dynamic implementations needed, we can already see that template metaprograms tend to be more complicated

than their dynamic counterparts. But on the other side, we have a program which can compute the factorial of a number at compile-time. Of course only if said number is a numerical constant known at compile-time.

2.2 Techniques

Template specialization and recursion are the two most important constructs in template metaprogramming, but there are some other interesting ones. Some of them are defined by the language, some have to be implemented, but all of them can help in making better template metaprograms.

2.2.1 Partial specialization

Specialization gave us the possibility of evaluating a condition, but what can we do, if our template takes more than one parameter? We can use *partial specialization*, meaning that we define all the parameters in the same way as in the general implementation and only specify the parameter we need to. In our example we have a template which will calculate the *m*th power of *n*. Again we will have to use recursion, and then specialize the template, but this time only the second parameter, the exponent will be specialized.

```
template<int n, int m>
struct power {
    enum { RET = power<n,m-1>::RET * n };
};

template<int n>
struct power<n,0> {
    enum { RET = 1 };
};
```

This can save a lot of work, because we do not need to spend our time to design templates with one parameter for the conditions we have. However, not all compilers will let you do this. Especially the older ones will not. Now that template metaprogramming becomes more and more popular, compiler builders pay more attention to enable partial specialization.

2.2.2 If-Construct

As discussed, `if` is dynamic and cannot be used in static programs. But, of course, sometimes it can be desirable to have an *if-construct*. We can build a template version of `if`, using partial specialization. Our first parameter will be the condition, and the second and third are types which will be returned if the condition does or does not hold. We will again need to return something, the general implementation will return the `ThenType`. Then we use partial specialization, set the first parameter to false, and set the return member to the `ElseType`. With these few lines of code, we can now use an if-construct.

```

template<bool cond, class ThenType, class ElseType>
struct IF {
    typedef ThenType RET;
};

template<class ThenType, class ElseType>
struct IF<false, ThenType, ElseType> {
    typedef ElseType RET;
};

```

Then we can specify a value or function in those types with the same name, and make a call to this member of the return type of `IF`. A possible use of the `IF`-template can be found in the appendix on page 17. One thing to note is, that one not should access members of the `Then` and `ElseTypes`, but access the member of the type returned by `IF`. Compilers evaluate templates lazily, as long as a template is not instantiated or a member is called, it will not be evaluated, it just has to exist. If one of the types passed to the `IF` template is not valid for a condition, the compiler would quit with an error, if an access to one of this types members was made.

It is possible to build an `IF`-construct without partial specialization as well; however this solution is much more complicated.

Clearly `if` is not the only construct which can be implemented. Template implementations of `switch..case`, `for`, `while` and `do..loop` can be made without much effort.

2.2.3 Partial evaluation

The compiler can evaluate template programs only when all values are static. You cannot call a class template with variables. Template instantiation works only with constants, and as there is no other way to pass a class values, class templates have to be static. Using function templates we can at least do some pre-evaluation, because we can pass dynamic values in the function call. The instantiation still has to be static though. For instance, if we have a function which calculates the n th power of a number m , but we know that our program will use this function to calculate only the third power of a number, we can use template metaprogramming to eliminate some overhead. Instead of three calls to the power function, the code results in $m * m * m$.

Again we use recursion to solve the problem, and we have two partial specializations, one for 1 and one for 0.

```

template<int n>
inline int power(const int& m) {
    return power<n-1>(m) * m;
}

template<>

```

```

inline int power<1>(const int& m) {
return m;
}

template<>
inline int power<0>(const int& m) {
return 1;
}

```

Note that we use inline functions here. An implementation which contains the code to this section, but does several things which go beyond the scope of this work is explained in the appendix on page 17.

2.2.4 Template template parameters

Template template parameters are perhaps the most powerful technique template metaprogramming has to offer. Using this technique we can define constructs very reminiscent of the high level functions known from functional programming. We can pass a template as a parameter to another template. Other parameters may be passed to this template, e.g. to implement a function which applies a given function to a set of values and sums up the results.

The accumulation template works very similar to what we saw until now. We also use partial specialization in this example. The specialized case applies 0 to a given function F, which is specified to be a template which takes one `int` parameter. In the general case we add the result of the recursive call, $F(n)$ and store this value in the `RET` member.

```

template<int n, template<int> class F>
struct Accumulate {
    enum {
        RET = Accumulate<n-1,F>::RET + F<n>::RET
    };
};

template<template<int> class F>
struct Accumulate<0,F> {
    enum { RET = F<0>::RET };
};

```

To define the template, we can use every function that works on one `int` parameter. In this example, we use the square function. In the accumulation template, we accessed the `RET` member of `F`, so we have to make sure that any template we want to use as `F`, has a public member `RET` defined.

```

template<int n>
struct Square {
    enum { RET = n*n };
};

```

We call the accumulation template the same way as any other template, the second parameter is simply the name of the template we want to use.

```
int void() {  
    cout << Accumulate<3, Square>::RET << endl;  
}
```

Of course, we could now define some more functions to apply to accumulate, and at the point when `Accumulate` is implemented, the only thing we need to do to extend the functionality is to create new templates to apply.

2.2.5 Loop unrolling

This technique uses the possibility of generating code with template metaprogramming. We saw a way of unrolling loops when we were talking about partial evaluation already. You can, however, let the template generate real code, for example output. The compiler will evaluate this, and put the generated code at the place from which the template was called. An example for this is in the appendix on page 19.

2.3 Functional elements

As we saw before, template metaprogramming has some similarities to functional programming languages. The necessity of recursion, no multiple value assignments, specialization as a conditional construct and the possibility of passing templates as parameters. Bearing in mind that template metaprogramming is not a designed language, but rather one which came into existence by accident, this is a very interesting fact.

3 Real-life examples

Until now we saw which tools template metaprogramming has to offer, and how to use them. But another important aspect is if there are applications of template metaprogramming beyond some proof-of-concept examples. Template metaprogramming still is far from being a standard programming technique, but there are some applications of it. Template metaprogramming is mostly used in libraries, e.g. to enhance the speed of mathematical computations through precalculated results, or to make using design patterns easier.

For our discussion we pick the *Loki* library, which was designed by *Andrei Alexandrescu* for his book *Modern C++ Design*. *Loki* is a generic design library which makes extensive use of sophisticated metaprogramming, which goes well beyond what we saw up until now.

Furthermore we will take a brief look at other libraries which use template metaprogramming.

3.1 Loki library

In his book, Alexandrescu describes what can be done with template metaprogramming. His main objective is to help C++ programmers to use Design Patterns without having to put up a heavy effort. Loki provides patterns like *Visitor*, *Singleton*, *Command* and *Factories*. The library is not implemented using only template metaprogramming, Alexandrescu makes use of other techniques like generic programming as well, and this is one of the interesting parts, seeing template metaprogramming in combination with other tools to create a great library.

To get an idea of what Loki can do, we will take a look at one of the most interesting concept Alexandrescu implements, the *typelist*. We will also see how Loki uses typelists for the Abstract Factory Pattern.

3.1.1 Typelists

Typelists are a very good example of how to implement data structures with template metaprogramming. As we did before we will again take a look at functional programming and how lists are implemented there. We have a basic element and a constructor, using that we have everything necessary to implement lists. A list could look like this: `Cons(4, Cons(2, Empty))` Keeping this in mind implementing a list with templates is not a difficult task.

```
template<class U, class T>
struct List {
    typedef U Head;
    typedef T Tail;
}
```

With this definition we can now create lists like this: `List<2, List<3, 4> >`. One problem of this implementation in contrast to the functional example is that we lack a basic element. Loki defines a `NullType` type for this use. However the user has to think about using it. Should she forget it, algorithms working on the list might fail. The List from above would have to be constructed as `List<2, List<3, List<4, NullType> > >`. It is important to remember the spaces between the `>`, else the compiler will misinterpret it as the stream input operator `»`.

Loki solves this problems by defining macros for lists up to a length of 50 elements. Using this macros a typelist definition is also much simpler: `TYPELIST_3(int, double, char)`. This example already shows the difference between a normal list and a typelist: a typelist takes types as values. These typelists are completely static; they do not exist at runtime.

Of course, only the typelist would be of no interest without functions to go with it. Loki provides functions starting with a simple length function up to indexed access and searching. We will discuss the length function here, as

it is the simplest, but still shows how to implement an algorithm that works with a list.

Alexandrescu shows what interesting things you can do with specialization in this example. He defines a template `Length` with one parameter, but as this template is not defined, the compiler will return an error when instantiating this template.

```
template <class TList> struct Length;
```

The second template will step in if the recursion reaches our artificial basic element `NullType`. The thing to notice is, that although we will call `Length` on a list, this specialization looks for `NullType`, which is only a part of a list, not the list itself. The third implementation shows the reason for this.

```
template <> struct Length<NullType> {
    enum {value = 0};
};
```

This is the second specialization of `Length`; this one, however, takes two parameters, but the definition also specifies that both parameters have to be inside a typelist template call. So when `Length` is instantiated with a typelist, the compiler will first try to match the parameters against the most specific implementation. This specialization is very specific, as we define which format the parameters need to have. So for every typelist, this template will be instantiated. Should `Length` be called with any other argument than `NullType` or a typelist, this will lead to a compiler error, because the general implementation is not defined. So, this little trick helps us to control that only the right parameters will be processed.

```
template <class T, class U>
struct Length<TypeList<T, U>> {
    enum { value = 1 + Length<U>::value };
};
```

In the body of this specialization, we call `Length` on the tail of the list. Looking on the way we defined our list, the tail of a list is either a `NullType` or a list again. This way the algorithm can process the whole list.

Adding, deleting, eliminating duplicates and replacing elements are also implemented. Up to this point the functions are interesting from an implementation point of view, but nothing extraordinary. A function which enforces a partial order on a typelist according to inheritance relationships of its types already provides more insight into the power of typelists. But the true power of typelists shows itself in three functions which create classes from typelists.

Let us assume you want to create a class with some fields. However, you want to be flexible which types these fields have. Loki defines two functions which create hierarchies from typelists. You pass a template to those functions along with the typelist, and they will return an object which

has an instantiation of the template with each of the types from the typelist. A possible template could be:

```
template<class T>
struct Holder {
    T value_;
};
```

Every type in the typelist will be applied to this template, so you will end up with an object with a value field for every type in your typelist. Loki has three functions of this kind. One generates a scattered hierarchy, the types end up in leaves of a tree and are therefore scattered. Another is similar to the scattered hierarchy, but gives direct access to the fields, without having to use its name. It is meant to be used for templates like the one above (i.e. with only one member). The third function creates a linear hierarchy, to maximize space efficiency. Of course, these functions can be used with much more complicated templates as the one used here, as long as they have one parameter. One could imagine creating a class with virtual functions for several types in a very elegant way. Should changes occur to the types you need the functions for, you just have to change types in the typelist accordingly and recompile.

3.1.2 Abstract Factory

The Abstract Factory pattern was introduced in the definitive book on design patterns *Design Patterns: Elements of Reusable Object-Oriented Software* by *Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides*. It is an expansion of the Factory Pattern. A Factory method is used to create an instance of a given object, initialize it and return it to the caller. The idea behind this is, that the factory method can be overwritten by a subclass. This way the superclass does not define which class will be loaded, but leaves this to subclasses. The Abstract Factory extends this principle to families of objects.

Given an application we might want to have a configurable appearance, however, we do not want all elements to have the same appearance. We might have Buttons and ScrollBars, now we want a classic and a modern appearance, but we do not want to have a ClassicButton at the same time as a ModernScrollBar. An Abstract Factory for creating the GUI elements would define virtual methods to load each element we have, concrete implementations of this Factory would then implement these functions to load the specific elements they need. In the application, the user would set the appearance he wants to have, and the code responsible for drawing the GUI would then call the abstract factory functions to get the elements. This way the programmer can make sure that the elements would not mix.

However, implementing this pattern can be tedious; you have to write the Abstract Factory with all the elements you want it to create, and each

concrete implementation of the factory, also with all elements. Should the need occur to change one of those elements, you have to change it in the abstract factory and every concrete factory. And this is the point where Loki comes in. With the power of typelists, creating Abstract Factories is suddenly a very easy task. You provide a template, a typelist and use the class generation function we mentioned in the previous section, and it is done. Loki even defines a template with the functions the Abstract Factory provides (one `DoCreate` function for every type to create and a destructor), so to create a Abstract Factory with a typelist which consists of a `Button` and a `ScrollBar`, all you have to do write is

```
typedef AbstractFactory <
    TYPELIST_2(Button, ScrollBar)
> AbstractAppFactory;
```

The `AbstractFactory` template is one of the more sophisticated templates, as it uses several techniques which were not introduced until now. For this reason it will not be discussed in detail. More insights into the details please can be found in [1]. With this code, we have an Abstract Factory with virtual functions to create a `Button` and a `ScrollBar`.

Creating the concrete factories is not much more work. For this Loki uses a technique called *policy*. Policies are small templates which define behavior. For example, if you want to write templates which dictate how new elements are created, one could implement several methods (one could use `malloc`, one could use `new` and so on). Now you can create a template which controls through which the elements will be created, and at initialization of this template provides the policy that should be used. Every time this element will be created through this template, the specified policy will be used to create it. In our example there is a default policy, which should fit in most cases (since every Concrete Factory has to implement the Abstract Factories create methods, there will usually be only a difference in the types used... and Loki takes care of the types automatically). So creating a concrete implementation is also just one call.

```
typedef ConcreteFactory <
    AbstractAppFactory,
    OpNewFactoryUnit,
    TYPELIST_2(ModernButton, ModernScrollBar)
> ModernAppFactory;
```

`OpNewFactoryUnit` is the policy. Again the `ConcreteFactory` and `OpNewFactoryUnit` templates are rather too sophisticated to be discussed in detail at this point. The interesting point about the Abstract Factory in Loki, is how Loki uses typelists to implement the Abstract Factory pattern. This shows how much potential template metaprogramming has.

3.1.3 Loki summary

As we saw, Loki can help programmers using design patterns without having to put up much extra effort. The powerful tools and techniques Loki offers translate into an easily usable library.

3.2 Other libraries

Blitz++ Blitz++ is a library for scientific computing. Its performance is on par with Fortran 77/90. Todd Veldhuizen created the core of Blitz++, by now several people are contributing to it.

BoostML BoostML is a general-purpose, high-level library. It was created by Aleksey Gurtovoy and David Abrahams in 2002 and is discussed in their book *C++ Template Metaprogramming*.

Spirit Spirit is a object-oriented recursive-decent parser generator library made by Joel de Guzman in 1998.

Pooma Pooma was developed at Los Alamos National Laboratory as a general purpose library for parallel scientific computations. It uses expression templates to achieve a performance comparable to Fortran 90.

MTL The Matrix Template Library provides high-performance linear algebra functionality for a wide variety of matrix formats.

4 Review and Summary

Template Metaprogramming is still relatively unknown. However it gains popularity, and finds more and more applications. It has many advantages, the most important being that it is evaluated at compile-time, providing high-performance (like we saw before, several libraries managed to get a performance as good as Fortran 90 using template metaprogramming). With template metaprogramming, dynamic code can be generated, which saves the programmer a lot of time, and helps with the maintenance of the program. With template metaprogramming the programmer can create generic code, as the compiler will fill the necessary types for him. This way the programmer can focus on the design of the program, and eventually raise the quality of his product. But, of course, there are also some drawbacks, and most of them origin in the fact that template metaprogramming was more or less born by accident. The code itself tends to be esoteric and hard to understand for someone not used to template metaprogramming. Not all compilers can cope with template metaprogramming, and if they do, the results may still differ. This can lead to portability issues, as a compiler which can deal with template metaprogramming might not be available on

all platforms. Debugging programs which make use of template metaprogramming are also an important issue. Most debugging tools were made with common C++ programs in mind, so templates can lead to problems here. Debugging templates is again another issue, as the debuggers work at runtime, and are of no use for debugging template metaprograms. Another issue is a side effect of the speed-up at runtime. This speed-up comes at the cost of a slow-down during compilation. Depending on how big a project is and how often it needs to be recompiled, this can be an issue.

However, the compiler compliance issue shows that some of the problems will be remedied when template metaprogramming gains more popularity. While pioneers like Unruh and Veldhuizen had not much of a choice regarding their compiler, nowadays there are many compilers on most important platforms (e.g. gcc) which work just fine with template metaprograms.

In conclusion, template metaprogramming is an interesting new way of programming, which gains most of its power out of the fact that it can work with normal C++ code, and generate or manipulate this dynamic code. However, it is not something which the C++ programmer would need in his daily routine. Its use is more in the field of high-performance or generic libraries.

A Code

A.1 Factorial example

A.1.1 iterative implementation

```
int fac(int n) {
    int x = 1;
    for(int i=1;i<=n;i++) {
        x *= i;
    }
    return x;
}
```

A.1.2 recursive implementation

```
int fac(int n) {
    if(n == 0) return 1;
    return n*fac(n-1);
}
```

A.1.3 template implementation

```
template<int N>
class Factorial {
public:
    enum {
        value = N * Factorial<N-1>::value
    };
};
template<>
class Factorial<1> {
public:
    enum { value = 1 };
};
template<>
class Factorial<0> {
public:
    enum {value = 1 };
};

int main() {
    int f = Factorial<5>::value;
    cout << "5! ist " << f << endl;
    f = Factorial<7>::value;
    cout << "7! ist " << f << endl;
}
```

A.2 partial specialization

```

template<int n, int m>
struct power {
    enum { RET = power<n,m-1>::RET * n};
};

template<int n>
struct power<n,0> {
    enum { RET = 1 };
};

int main() {
    cout << "2 hoch 3 ist: " << power<2,3>::RET << endl;
    cout << "3 hoch 3 ist: " << power<3,3>::RET << endl;
}

```

A.3 **if-construct**

```

template<bool cond, class ThenType, class ElseType>
struct IF {
    typedef ThenType RET;
};

template<class ThenType, class ElseType>
struct IF<false, ThenType, ElseType> {
    typedef ElseType RET;
};

struct Type1 {
    static void print() {
        cout << "Type1" << endl;
    }
};

struct Type2 {
    static void print() {
        cout << "Type2" << endl;
    }
};

int main() {
    IF<(1<2), Type1, Type2>::RET::print();
    IF<(4%2 == 0), Type1, Type2>::RET::print();
    IF<(5%2 == 0), Type1, Type2>::RET::print();
}

```

A.4 **partial evaluation**

This code does a bit more than just partial evaluation. It implements three different versions of the Power function, as well as a StaticInt wrapper for integer constants. The three different versions of the power functions are

defined for different mixtures of static and dynamic parameters. One is for calls with 2 static ints, it is the version we looked at when we talked about partial specialization. The second is the one described in the section about partial evaluation, which takes one dynamic and one static parameter. The third is normal dynamic inline function. This example shows quite nicely how static and dynamic code can interact to make speed-ups where possible without putting the task on the programmer to think about how to call each of the different versions. A template will call the right implementation for the given parameters, thanks to specialization. However this techniques go a bit over the scope of this works, that is why they are not discussed in detail.

```

inline int power(const int& m, int n) {
    int r = 1;
    for (; n>0; --n) r *= m;
    return r;
}

template<int m, int n>
struct Power {
    enum { RET = Power<m, n-1>::RET * m };
};
template<int m>
struct Power<m,0> {
    enum { RET = 1 };
};

template<int n>
inline int power(const int& m) {
    return power<n-1>(m) * m;
}

template<>
inline int power<1>(const int& m) {
    return m;
}

template<>
inline int power<0>(const int& m) {
    return 1;
}

template<int n>
struct StaticInt {
    enum { RET = n };
    operator const int() const {return n;}
};

template<int m, int n>
inline StaticInt<Power<m,n>::RET>

```

```

    power (const StaticInt<m>&, const StaticInt<n>&) {
return StaticInt<Power<m,n>::RET>();
}

template<int n>
inline int power(const int& m, const StaticInt<n>&) {
return power<n>(m);
}

StaticInt<2> c2;
StaticInt<3> c3;

int main() {

cout << power(2,3) << endl;      // dynamic version
cout << power(c2,3) << endl;     // dynamic version (the exponent is dynamic
cout << power(2,c3) << endl;     // partial evaluation
cout << power(c2,c3) << endl;    // static version
cout << power(c2,power(c3,c3)) << endl; // static version
}

```

A.5 Template template parameters

```

template<int n, template<int> class F>
struct Accumulate {
    enum {
        RET = Accumulate<n-1,F>::RET + F<n>::RET
    };
};

template<template<int> class F>
struct Accumulate<0,F> {
    enum { RET = F<0>::RET };
};

template<int n>
struct Square {
    enum { RET = n*n };
};

int void() {
    cout << Accumulate<3,Square>::RET << endl;
}

```

A.6 Loop unrolling

```

template< int i >
class LOOP{
public:

```

```

        static inline void EXEC(){
            cout << "A-" << i << " ";
            LOOP< i-1 >::EXEC();
            cout << "B-" << i << " ";
        }
};
template<>
class LOOP< 0 >{
public:
    static inline void EXEC(){
        cout << "A-0";
        cout << "\n";
        cout << "B-0 ";
    }
};

int main() {
    LOOP<8>::EXEC();
    cout << "\n";
}

```

Output:

```

A-8 A-7 A-6 A-5 A-4 A-3 A-2 A-1 A-0
B-0 B-1 B-2 B-3 B-4 B-5 B-6 B-7 B-8

```

References

- [1] Andrei Alexandrescu: *Modern C++ Design* Addison-Wesley, 2001
- [2] Krzysztof Czarnecki, Ulrich W. Eisenecker: *Generative Programming* Addison-Wesley, 2000
- [3] Todd Veldhuizen: *Template Metaprograms*
<http://osl.iu.edu/~tveldhui/papers/Template-Metaprograms/meta-art.html>
- [4] Wikipedia: *Template Metaprogramming*
- [5] The Code Project: *Template Metaprogramming*
http://www.codeproject.com/cpp/crc_meta.asp
- [6] Blitz++ Homepage <http://www.oonumerics.org/blitz/papers>
- [7] BoostML Homepage <http://www.boost.org/libs/mpl/doc>
- [8] Spirit Homepage <http://www.boost.org/libs/spirit>