Introduction to metaprogramming

Nicolas Burrus

Revision 1.1, December 2003

This report aims at simplifying the discovery of the static C++ world. Mostly relying on Todd Veldhuisen *Techniques for scientific C++* and Andrei Alexandrescu *Modern C++ Design*, we try to give to the novice metaprogrammer most of the basics notions he should learn, in a didactic way. The goal is also to make the reader work and think by himself before discovering already existing solutions, in order to facilitate the understanding.

FIXME

Keywords
c++, template, metaprogramming

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
lrde@epita.fr – http://www.lrde.epita.fr

Copying this document

Contents

Introduction

Intent

This report has been written as a tutorial. It does not pretend to be a reference manual neither the only document to read. The goal is just to give an overview of useful tools and techniques one should learn before working on projects using advanced C++ static constructs. Most common caveats are briefly explained, so that the reader will remind them if he encounters the problem one day.

To achieve this goal, a tutorial approach has been chosen, with several exercise to make the reader think by himself about problems, thereby increasing his level of comprehension.

How to read this document

This document is mostly self-contained for people who already have a good knowledge of the C++ language. Most of the material can be found in (Veldhuizen, 1999) and (Alexandrescu, 2001) with much more details and explanations.

You should start be reading this report, play with the given code if it is not that simple for you, working on the given exercises, looking at the solutions if you can't find them by yourself.

After reading this document, you are highly advised to have a look at (Veldhuizen, 1999) and (Alexandrescu, 2001). Some chapters are essential to read carefully, other are not. Most of the important chapters (for our concern!) are referenced in the corresponding sections of this tutorial.

Note about C++ compilers

C++ code involving templates generally leads to cryptic compiler error messages. Some compilers provide better messages than others, so you should try your code with other compilers if you can't understand the given messages.

Besides, templates are a quite recent and tricky addition to the C++ standard, so almost all compilers still have many bugs with templates. Comparing the results with other compilers may help you tracking down compilers bugs.

(Veldhuizen, 1999) gives a good (even if a little outdated) introduction to C++ compilers functionalities. From our experience, here are the use we make of the different compilers we currently have:

- **G++ 2.95** is deprecated, we don't use it anymore. It is too loose and suffers from many

critical bugs, but has very good performance, both about compile-time and generated code. His favorite error message is "Internal error."

- **G++ 3.3** is fine. The standard is more respected, compile-time and generated code are correct. This is our default compiler.
- **G++ 3.4** has a new, completely rewritten parser. Therefore, error messages are sometimes better, but as the time of the writing this version is still experimental. The standard is much more respected than in previous version, so it is worth compiling with G++ 3.4 is you want to make sure that your code is standard compliant.
- **Intel C++ 8** compiles generally faster than Gcc, but the generated code is often slower (on Linux). The standard is usually respected, and error messages are fine (EDG front-end). It is a good idea to test your code with it.
- **Comeau 4.3** is the most standard compliant compiler. Thus, you should try your code with Comeau is you have a doubt about the C++ standard, it will generally give the good answer. The error messages are similar to ICC ones since they share the same front-end. However, the generated code is very slow compared to other compilers.

Outline

The first chapter deals with generalities C++ templates. The goal of this chapter is to familiarize the reader with the template possibilities, and to gives some tips and usual caveats C++ template programmers generally encounter.

Relying of this knowledge of templates, the second chapter presents some basic metaprogramming tools the reader will certainly need one day. It should also give a preview of the complicated things we can ask the compiler to do.

Finally, the third chapter summarizes more advanced techniques, which are not necessarily fundamental but which should be seen at least one time by every static C++ programmers.

Chapter 1

Short review of templates in C++

The goal of this chapter is to give an overview of the low level template construction and behavior in C++. You may want to skip some sections if you have a good knowledge of templates in C++. This chapter requires some basic knowledge of templates in C++.

1.1 The static world

In the following, the static world represent all the computations which can be performed at compile-time. In C++, thanks to the **template** keyword, many constructions are made available.

1.1.1 Template constructions as functions

Class templates can be seen as functions taking some parameters and returning a type. For example, when writing:

```
template <class T>
struct Foo
{
};
```

Foo is a function, taking a class T as parameter and returning a new type. On the contrary, Foo<**int**> is a concrete type, result of the function Foo applied to T.

The same reasoning process stands for template functions, which are functions waiting for some static parameters and returning a concrete function.

This way of considering template constructs will help the understanding of the next chapters.

1.1.2 Possible kinds of template parameters

A template parameter can be either a type, an integral constant (predefined types: int, char, etc. excepted floats) or an unbounded template (non instantiated, still a function returning types).

For example:

```
// Type parameter
template <class T>
struct FooT {};

// Constant parameter
template <char c>
struct FooChar {};

// Unbounded template parameter
template <template <class> class F>
struct FooUnbound
{
  F<int> myfirstmember;
  F<char> mysecondmember;
};

struct FooT<int>;
struct FooChar<'a'>;
struct FooUnbound<FooT>;

// this is also valid since a qualified template is a type (FooChar<'c'>).
struct FooT<FooChar<'c'> >;
```

Note that non POD types cannot be used as template parameters:

```
struct Foo {};

template <Foo f> // invalid, Foo is not an integral type
struct Bar;
```

1.1.3 Possible template arguments

The main rule is (with many exceptions, we are in C++): whatever can be known at compile-time can be a template argument.

Let us take the example of a Vector class having an integer template parameter representing the dimension, one can write:

```
template <unsigned Dim>
struct Vector
{
  // ...
};

struct InfoEnum
{
  enum { dim = 2 };
};

struct InfoStaticConst
{
  static const unsigned dim = 2;
};

char func();
```

```
// This syntax can be used to explicitly instantiate template structures.
template struct Vector<2>;
template struct Vector<1 + 1>;
template struct Vector<InfoEnum::dim>;
template struct Vector<InfoStaticConst::dim>;
template struct Vector<sizeof(char) + 1>;
template struct Vector<sizeof(func()) + 1>;
```

Note that **sizeof** can give the size of an expression without actually evaluating it. Here, func is never called.

The same kind of indirections are possible with typedefs, as in the following example:

```
template <class T>
struct Foo
{};

struct Bar
{
  typedef int param_type;
};

template <class U>
struct Baz
{
  typedef U param_type;
};

template <class Ref>
struct Qux
{
  // Look for the param_type defined in Ref
  typedef typename Ref::param_type param_type;
};

// These 3 declarations are identical
template struct Foo<int>;
template struct Foo<Bar<int>::param_type>;
template struct Foo<Qux<Bar<int> >::param_type >;
```

static_cast can also be used without losing the static properties of arguments (this is obviously false with **dynamic_cast**).

Static members can also be accessed at compile time. Their value, however, is only available statically is the member is constant. Example:

```
#include <iostream>

template <unsigned I>
struct Bar
{};

struct Foo
{
  static void
  print()
```

```
  {
    std::cout << "Foo::print()" << std::endl;
  }

  static unsigned dim_not_const;
  static const unsigned dim_const = 5;
};

unsigned Foo::dim_not_const = 5;

template <class T>
void foo()
{
  // May be inlined, the method call is resolved statically.
  // No instance is required.
  T::print();

  // Its address can be used statically also
  std::cout << T::dim_not_const << std::endl;
}

template struct Bar<Foo::dim_const>; // allowed, dim_const is const
template struct Bar<Foo::dim_not_const>; // obviously forbidden
```

What is interesting to notice in this section is that all the computation required to resolve the indirections and find the actual types or values to use are performed by the compiler. Thus, no runtime overhead should be observed if your C++ compiler is intelligent enough (this is generally the case, fortunately).

1.1.4 Template evaluation

Template constructions are functions waiting for arguments to return something. So until they get concrete arguments they cannot be completely evaluated. Some compilers may detect code which will never work whatever the arguments are, but most compilers will just ignore the template code until it is instantiated. So be careful when writing code with templates, if they are not instantiated, you cannot be certain that your code is correct, even at the syntax level.

Template evaluation is lazy, templates are instantiated when they are used somewhere in the code or if they are explicitly instantiated (as in the previous examples).

1.2 Template specialization

C++ templates can be specialized, completely or partially. This means we can write dedicated code for some particular values of the parameters. However, template specialization is only allowed at the namespace level. This means neither template member functions cannot be specialized, neither inner template classes:

```
template <class T>
struct Foo
{
  template <class U>
  struct Bar {};
```

```
#if 0
  template <>
  struct Bar<int>; // forbidden, we are not at the namespace level
#endif

  template <class U>
  void foo();

#if 0
  template <>
  void foo<int>(); // forbidden, we are not at the namespace level
#endif
};

// This is ok
template <>
struct Foo<int>
{};

template <class U>
void foo();

template <>
void foo<int>(); // ok
```

The first property to keep in mind is the exact matching of template parameters, which is not always natural, as in the following example:

```
// Default version
template <class T>
struct Dim
{
  enum { dim = 51; }
};

struct Image2d
{};

struct SquareImage2d : public Image2d
{};

// Specialization for A
template <>
struct Dim<Image2d>
{
  enum { dim = 2; }
};

int main()
{
  // Prints 51
  std::cout << Dim<SquareImage2d>::dim << std::endl;
}
```

Since the template matching mechanism performs an exact matching, the version of Dim specialized for Image2d is not analyzed when the argument is SquareImage2d, even if a SquareImage2d

"is" (in the sense of inheritance) an Image2d. The same rules are applied with function overloading when templates are involved, the default version is much more greedy:

```
#include <iostream>

template <class T>
void foo(T)
{
  std::cout << "foo(T)" << std::endl;
}

struct Image2d {};
struct SquareImage2d : public Image2d {};

void foo(Image2d&)
{
  std::cout << "foo(Image2d)" << std::endl;
}

int main()
{
  // Prints "foo(T)".
  foo(SquareImage2d());
}
```

1.2.1 Partial specialization

Partial specialization consists in bounding only a subset of the template parameters. Template classes can be partially specialized, however template functions cannot. Note that partial specialization if not restricted to the namespace level, unlike total specialization (I guess you are now beginning to love C++).

// Exo: specialization of an inner struct

```
template <class T, class U>
void foo();

// Partial specialization of functions is forbidden.
#if 0
template <class T>
void foo<T, int>();
#endif

template <class T, class U>
struct Foo
{};

// Ok, partial specialization
template <class U>
struct Foo<int, U>
{};

struct Bar
{
  template <class T, class U>
  struct Baz;
```

```
  // Ok, partial specialization is possible in structs.
  template <class T>
  struct Baz<T, int>;
};
```

Exercise 1.1

The goal of this exercise is to find a workaround to allow total specialization of inner structs. The solution should have the same behavior as this code:

```
#include <iostream>

struct A
{
  template <class T>
  struct B
  {
    static void foo()
    {
      std::cout << "B<T>::foo()" << std::endl;
    }
  };

  template <>
  struct B<int>
  {
    static void foo()
    {
      std::cout << "B<int>::foo()" << std::endl;
    }
  };
};

int main()
{
  A::B<int>::foo(); // "B<int>::foo()"
}
```

Hint: template parameters can have default values.

Exercise 1.2

Find a solution to have partial specialization of static template member functions.

```
#include <iostream>

template <class V>
struct A
{
  template <class T, class U>
  static void foo()
  {
```

```
    std::cout << "foo<T, U>, sizeof(V): " << sizeof(V) << std::endl;
  }

  template <class T>
  static void foo<T, int>()
  {
    std::cout << "foo<T, int>, sizeof(V):" << sizeof(V) << std::endl;
  }
};

int main()
{
  A<int>::foo<char, int>(); // should print "foo<T, int>, sizeof(V): 4"
}
```

1.3 Symbol lookup

By symbol lookup, we regroup function lookup, variable lookup, etc. Namespaces and overloading rules are quite tricky in C++, so this section tries to show a few common difficulties.

1.3.1 Koenig lookup

Symbols do not always need to be prefixed by their namespace. By default, symbols will be searched into the current namespace, into the namespace included within the current one with the **using** keyword.

But this is not the only rule. When a function is called, candidates are looked for in the namespace of the arguments and even in the namespace of the template parameters of the arguments. This is illustrated in the following example:

```
namespace nsp_foo {

  struct Foo
  {};

  template <class T>
  void foo(T)
  {}

}

namespace nsp_bar {

  template <class T>
  struct Bar
  {};

  void bar()
  {
    // foo will be found in nsp_foo because its argument
    // is in namespace nsp_foo.
    foo (nsp_foo::Foo());
```

```
    // foo will be found in nsp_foo because its argument has a
    // template parameter in namespace nsp_foo.
    foo (Bar<nsp_foo::Foo>());
  }

}
```

1.3.2 Overloading and inheritance

Be careful with overloading and inheritance, if a function is redefined in a subclass, versions of the function taking other kinds of parameters are not considered. An **using** directive is required:

```
struct A
{
  static void foo (int)
  {}
};

struct B : public A
{
  static void foo (double)
  {}
};

struct C : public A
{
  // Import A version of foo
  using A::foo;

  static void foo (double)
  {}
};

int main()
{
  B::foo(5); // will not call A::foo(int)
  C::foo(5); // will call A::foo(int)
}
```

1.3.3 Template functions argument deduction

Template parameters can be implicitly deduced if they are used as parameters of the function. The deduction can be intelligent:

```
template <class T>
struct A {};

template <class T>
void foo(A<T>)
{}

int main()
{
```

```
  A<int> a;
  foo(a); // ok, T will be identified as int
}
```

1.4 Common troubles

1.4.1 How to use the typename keyword

The **typename** keyword tells the compiler that the following expression is a type. It is often useful to disambiguate template expression, for example in **typedef** T::foo bar;, with T a template parameter, whether foo is a type, a constant or a function cannot be known by the compiler.

The exact rules are quite complicated and tricky (once again), so the general rule to apply is: put a **typename** whenever you use type template expression within a template, and test with several compilers.

1.4.2 The template keyword for disambiguation

The **template** keyword may be necessary in some weird situations to tells the compiler that the following expression is a template, see
`http://www.lrde.epita.fr/cgi-bin/twiki/view/Know/TemplateKeywordForDesambiguation`
for more details. You might need it with standard compliant compilers.

Chapter 2

A few metaprogramming tools

Introduction

With the introduction of templates, people initially wanted to bring genericity to the C++. Actually, template constructs are much more powerful than initially though. A complete meta programming level is available, which has been proved to be Turing complete! This means we can make the compiler execute real programs, manipulating C++ code (essentially types). (Veldhuizen, 1999) gives some example of complex programs which can be written in meta C++. This chapter gives a very short overview of the tasks which can be achieved at compile-time. The goal is to write basic tools which will be useful very often.

2.1 Static conditions

Conditionals can be written in meta C++ using template specialization. The goal of our meta_if is to choose between 2 types, depending on a condition. Example:

```
template <class T>
void foo()
{
  typedef meta_if(sizeof(T) == 4, int, float) my_type;
  my_type tmp;
}
```

This code should make my_type an **int** if **sizeof**(T) == 4, or a **float** otherwise. Or course, the condition will have to be static. This code can be easily implemented using template specialization. The code is left as an exercise.

Exercise 2.1

Write the meta_if structure enabling this code to work:

```
#include <iostream>

// meta_if
```

```
template <class T>
void foo()
{
  typedef typename meta_if<sizeof(T) == 1, char, float>::ret my_type;
  std::cout << sizeof(my_type) << std::endl;
}

int main()
{
  foo<char>(); // prints "1"
  foo<float>(); // prints sizeof(float), generally "4"
}
```

2.2 Static assertions

(Alexandrescu, 2001) has written a complete chapter about static assertions. This section only introduces the topic.

As seen in Chapter 1, template evaluation is lazy. This means we can write template code which may fail to compile if the given parameters do not satisfies our constraints, the only thing we must not do is to actually instantiate the templates with the wrong parameters. Let us consider the following example:

```
template <bool b>
struct static_assert
{
  static void ensure() {};
};

template <>
struct static_assert<false>
{
  // no ensure()
};

template <class T>
void foo()
{
  static_assert<sizeof(T) == 1>::ensure();
}

int main()
{
  foo<char>(); // ok
  foo<int>(); // does not compile, sizeof(int) != 1
}
```

This assertion will be evaluated statically, preventing the instantiation of invalid code.

Exercise 2.2

In the following example:

```
template <class T>
struct Foo
{};

template struct Foo<char>;
template struct Foo<int>; // should not compile
```

You have to ensure that the condition of the previous section is respected: **sizeof**(T) must be equal to 1. Instantiation of Foo with a wrong type should fail at soon at possible. The check code should be reusable.

2.3 Loops

Loop can be executed by the compiler. This can seem surprising, but creates many opportunities to beat the compiler hard (it is a good solution to relax after a few hours of C++ programming).

The trick comes from two main observations:

- arithmetic operations on static arguments are computed statically;
- templates can be specialized implicitly.

To illustrates the possibilities of loops in metaprograms, we will take the classical example of the factorial function (also in (Veldhuizen, 1999)). It is possible to make the compiler compute the factorial of a number:

```
#include <iostream>

template <int n>
struct fact
{
  enum { res = n * fact<n - 1>::res };
}

template <>
struct fact<0>
{
  enum { res = 1 };
};

int main()
{
  std::cout << fact<10>::res << std::endl;
}
```

Why does this work ? The multiplication n * fact<n − 1>::res can be done statically, since both n, a template parameter, and fact<n - 1>::res, an **enum** value, can be statically computed.

This technique leads to good loop unrolling possibilities, as shown in (Veldhuizen, 1999) with a dot product optimization example.

2.4 Traits

Let us start with a simple example:

```
template <class T>
struct type_traits;

template <>
struct type_traits<unsigned char>
{
  typedef unsigned short bigger_type;
  // ... other associated properties
};

template <>
struct type_traits<signed int>
{
  typedef signed long bigger_type;
  // ...
};

// ... other specializations of type_traits

template <class T>
type_traits<T>::bigger_type
compute_sum (T data[5])
{
  type_traits<T>::bigger_type sum = 0;
  for (unsigned i = 0; i < 5; ++i)
    sum += data[i];
  return sum;
}
```

The compute_sum functions computes the sum of five elements of type T. If the result is stored in a variable of type T, an overflow will certainly occur. We would like to store the result if a bigger type than T to decrease the probability to get an overflow. The bigger type to use depends on T. Template classes, which can be specialized, are a good way to associate static properties (types, values, static functions, ...) to a particular type, or more generally to a particular set of static parameters (template classes can have several parameters).

This is often useful when writing generic algorithms or generic classes when one need to access to some properties of the template parameter. Do not forget that template specialization are just classes, and can inherit from other classes. So traits of one type can inherit from traits of another type for example.

Exercise 2.3

Implement a generic function plus(T, U) which returns the result of the addition of the two parameters. The return type depends on the of the input parameters, with the following rules:

- T + T $\Rightarrow$ T
- **char** + **int** = **int** + **char** $\Rightarrow$ int

- any type + **float** = **float** + any type ⇒ float

Code redundancy should be minimized.

Example of test code:

```
#include <cassert>

// ...

int main()
{
  assert( sizeof(plus(1, 2)) == sizeof(int) );
  assert( sizeof(plus(0.5f, 2)) == sizeof(float) );
  assert( sizeof(plus(2, 0.5f)) == sizeof(float) );
  assert( sizeof(plus(true, 0.5f)) == sizeof(float) );
  assert( sizeof(plus('c', 5)) == sizeof(int) );
  assert( sizeof(plus(true, false)) == sizeof(bool) );
}
```

Exercise 2.4

Find a solution to implement commutative traits automatically, that is: traits <T, U>::ret should be the same as traits <U, T> without explicitly defined both specialization. Example:

```
template <class T, class U>
struct my_traits;

template <>
struct my_traits<int, char>
{
  typedef int ret;
};

int main()
{
  assert( sizeof(my_traits<int, char>::ret) == sizeof(int) );
  assert( sizeof(my_traits<char, int>::ret) == sizeof(int) );
};
```

Hint 1: do not actually use my_traits in the user code, but a wrapper which performs the required additional computations.

Hint 2: You can use meta_if.

Hint 3: Hey, you already have enough hints!

2.5 Recommended readings

Todd chapter on metaprogramming is interesting, you should especially read the section about dot product unrolling. However, the FFT example is complicated, and not that interesting from a didactic point of view.

Chapter 3

Useful techniques

Now you should be quite familiar with template constructs and the expression power one can use to make the compiler execute programs. This chapter focuses on more specific techniques, which can be helpful in some applications. In all cases, it is worth at least knowing that they exist, and they gives some concrete application of the weird and complicated C++ you have started to learn.

3.1 Static lists

(Alexandrescu, 2001) has a complete chapter about static lists. It is worth reading it to get a big complete panel of the possibilities, but this section should be enough to get only an overview of the main principles. Anyway, it may be better to read and work on the exercise of this section before discovering more details in the book of Andrei Alexandrescu.

On one hand, template classes can take type arguments. On the other hand, template classes are themselves types when their arguments are specified. This way, recursive templates can be constructed:

```
struct End;

template <class T>
struct Recursive
{};

template struct Recursive< Recursive< Recursive<End> > >;
```

Recursive takes one type parameter, which is itself of type Recursive, etc. This code is not really interesting, but opens a new field of opportunities to torture the C++ compiler. If Recursive takes one additional parameter, an element, and keeps the second parameter to continue the recursion, we get linked lists:

```
struct End;

template <class Element, class Next>
struct List
{};
```

```
template struct List<int, List<char, List<float, End> > >;
```

Each List class holds an element and the next elements. What is interesting here is that we have a list of types, which can be handled at compile-time as we want.

Exercise 3.1

Store the size of static lists (statically of course). Sample test code:

```
#include <cassert>

struct End;

template <class Element, class Next>
struct List
{
  // ...
};

int main()
{
  typedef List<int, List<char, List<float, End> > > mylist;
  assert( mylist::size == 3 );
}
```

List of types can be useful is many applications, refer to (Alexandrescu, 2001) for more examples. Here we will detail just one idiom where lists can be useful: inheritance from a list of types. We will consider a generic class Aggregator which takes a list of types and inherit from each one. The code is almost straight forward using type lists:

```
struct End;

template <class Element, class Next>
struct List
{};

struct Foo
{
  void foo() {}
};

struct Bar
{
  void bar() {}
};

template <class T>
struct Aggregator
{};

template <class Element, class Next>
struct Aggregator<List<Element, Next> >
  : public Element, // Inherit from the current type
    public Aggregator<Next> // Inherit from the remaining types indirectly
```

```
{};

int main()
{
  Aggregator< List <Foo, List <Bar, End > > > agg;
  agg.foo();
  agg.bar();
}
```

Exercise 3.2

The goal of this exercise is to implement a meta_switch, in the same spirit as the meta_if. Basically, a switch is an expression (of type integer so simplify the exercise), followed by a list of cases. Each case is a pair (value of the expression, result). You should implement the code of the meta_switch which allows the following code to work:

```
#include <cassert>

// ... meta_switch

template <unsigned expr>
unsigned foo()
{
  typedef typename meta_switch<expr,
                               meta_case<1, char,
                               meta_case<4, int,
                               meta_case<8, float,
                               meta_default<bool>
                    > > > >::ret result_type;
  return sizeof(result_type);
}

int main()
{
  assert( foo<1>()  == sizeof(char) );
  assert( foo<4>()  == sizeof(int) );
  assert( foo<8>()  == sizeof(float) );
  assert( foo<5>()  == sizeof(bool) );
  assert( foo<12>() == sizeof(bool) );
}
```

Hint: you already have meta_if. Hint: the recursion is represented by meta_case.

3.2 Static polymorphism

Chapter 1.3 of (Veldhuizen, 1999) gives a good introduction of static polymorphism. You should read and understand it (especially Section 1.3.3) before continuing to read this section.

In the following, we will consider this example:

```
template <class Exact>
struct AbstractImage
```

```
{
  Exact& exact()
  {
    return static_cast<Exact&>(*this);
  }

  unsigned size()
  {
    // Dispatch on the exact type
    exact().size();
  }
};

struct MyImage2d : public AbstractImage<MyImage2d>
{
  unsigned size();
  unsigned nb_row();
  unsigned nb_col();
};

template <class Exact>
void foo (AbstractImage<Exact>& ima)
{
  // ...
}
```

This is a direct application of the Barton and Nackman trick. For a theoretical discussion about why we want foo to take an AbstractImage and not directly a template parameter T, thereby solving the polymorphism issues, see (Lesage, 2003). The main idea is that we want to keep strong both function signature (to keep overloading possibilities) and abstraction power.

Now imagine that we want to define an abstraction for 2d images, say AbstractImage2d. We need to apply the Barton and Nackman trick to a 3-level hierarchy. This can be done easily by propagating the exact type all over the hierarchy:

```
template <class Exact>
struct AbstractImage
{
  Exact& exact()
  {
    return static_cast<Exact&>(*this);
  }

  unsigned size()
  {
    // Dispatch on the exact type
    exact().size();
  }
};

template <class Exact>
struct AbstractImage2d
  : public AbstractImage<Exact> // Propagation of the Exact parameter
{
  unsigned nb_row()
  {
    return exact().nb_row(); // exact() is inherited for AbstractImage
```

```
  }

  unsigned nb_col()
  {
    return exact().nb_col();
  }
};

struct MyImage2d : public AbstractImage2d<MyImage2d>
{
  unsigned size();
  unsigned nb_row();
  unsigned nb_col();
};

template <class Exact>
void foo2d (AbstractImage2d<Exact>& ima2d)
{
  ima2d.nb_row();
  // ...
}

int main()
{
  MyImage2d ima;
  foo2d(ima); // ok
}
```

This way, hierarchies with an arbitrary number of levels can be defined. However, only leafs classes (here MyImage2d) can be instantiated. If we want to have classes containing virtual methods but which can still be instantiated (with default code for the virtual methods), the problem becomes more complicated. (Lesage, 2003) studies of how complex hierarchies, much more similar to classical hierarchies can be defined.

3.3 Expression templates

(Veldhuizen, 1999) has a comprehensive and interesting Chapter about expression templates. It is a useful method to keep somewhere in your mind, so you are highly advised to read it (Chapter 1.9).

Conclusion

You should now be prepared to read more specific and advanced material. Do not forget to read (at least quickly) the given references, which are really essential.

Appendix A

Solutions to exercises

A.1 Exercise 1.1

```
#include <iostream>

struct A
{

  // The default argument will not change the call in the user code.
  template <class T, class Bogus = void>
  struct B
  {
    static void foo()
    {
      std::cout << "B<T>::foo()" << std::endl;
    }
  };

  // partial specialization is accepted
  template <class Bogus>
  struct B<int, Bogus>
  {
    static void foo()
    {
      std::cout << "B<int>::foo()" << std::endl;
    }
  };
};

int main()
{
  A::B<char>::foo(); // "B<T>::foo()"
  A::B<int>::foo(); // "B<int>::foo()"
}
```

The trick is to simulate partial specialization, which is possible for inner structs. Yes, this is tricky, but the C++ is a tricky language!

A.2 Exercise 1.2

```
#include <iostream>

template <class V>
struct A
{
  template <class T, class U>
  static void foo()
  {
    foo_struct<T, U>::foo();
  }

  template <class T, class U>
  struct foo_struct
  {
    static void foo()
    {
      std::cout << "foo<T, U>: " << sizeof(V) << std::endl;
    }
  };

  template <class T>
  struct foo_struct<T, int>
  {
    static void foo()
    {
      std::cout << "foo<T, int>: " << sizeof(V) << std::endl;
    }
  };
};

int main()
{
  A<int>::foo<char, int>(); // should print "foo<T, int>: 4"
}
```

Static functions can be enclosed in structs. Since structs partial specialization is possible, partial specialization of member functions is also possible!

A.3 Exercise 2.1

```
#include <iostream>

template <bool cond, class T, class U>
struct meta_if
{
  typedef T ret;
};

template <class T, class U>
struct meta_if<false, T, U>
{
  typedef U ret;
```

```
};

template <class T>
void foo()
{
  typedef typename meta_if<sizeof(T) == 1, char, float>::ret my_type;
  std::cout << sizeof(my_type) << std::endl;
}

int main()
{
  foo<char>(); // prints sizeof(char)
  foo<int>(); // prints sizeof(float)
}
```

The code is really simple. meta_if returns the first type is the condition is true, the second one otherwise.

A.4 Exercise 2.2

```
template <bool b>
struct static_assert
{
  typedef void ensure_type;
};

template <>
struct static_assert<false>
{};

template <class T>
struct Foo
{
  typedef typename static_assert<sizeof(T) == 1>::ensure_type ensure_type;
};

template struct Foo<char>;
template struct Foo<int>; // does not compile
```

If the boolean is false, ensure_type is not found, and the class cannot be instantiated.

A.5 Exercise 2.3

```
#include <cassert>

// Default

template <class T, class U>
struct plus_traits;

// T + T
```

```
template <class T>
struct plus_traits<T, T>
{
  typedef T ret;
};

// char + int ; int + char

template <>
struct plus_traits<char, int>
{
  typedef int ret;
};

template <>
struct plus_traits<int, char> : public plus_traits<char, int> {};

// float + T ; T + float

template <class T>
struct plus_traits<float, T>
{
  typedef float ret;
};

template <class T>
struct plus_traits<T, float> : public plus_traits<float, T> {};

// plus

template <class T, class U>
typename plus_traits<T, U>::ret
plus(T t, U u)
{
  return t + u;
}

int main()
{
  assert( sizeof(plus(1, 2))        == sizeof(int) );
  assert( sizeof(plus(0.5f, 2))     == sizeof(float) );
  assert( sizeof(plus(2, 0.5f))     == sizeof(float) );
  assert( sizeof(plus(true, 0.5f))  == sizeof(float) );
  assert( sizeof(plus('c', 5))      == sizeof(int) );
  assert( sizeof(plus(true, false)) == sizeof(bool) );
}
```

Note the use of inheritance between traits to factorize the code.

A.6 Exercise 2.4

```
#include <cassert>

// meta_if
```

```
template <bool b, class T, class U>
struct meta_if
{
  typedef T ret;
};

template <class T, class U>
struct meta_if<false, T, U>
{
  typedef U ret;
};

struct undefined;

template <class T>
struct is_defined
{
  enum { ret = true };
};

template <>
struct is_defined<undefined>
{
  enum { ret = false };
};

// Default my_traits

template <class T, class U>
struct my_traits_core
{
  typedef undefined ret;
};

// Try my_traits_core<T, U>, if it is undefined, return my_traits_core<U, T>

template <class T, class U>
struct my_traits
{
  typedef typename
  meta_if<is_defined<typename my_traits_core<T, U>::ret>::ret,
          typename my_traits_core<T, U>::ret,
          typename my_traits_core<U, T>::ret>::ret ret;
};

template <>
struct my_traits_core<int, char>
{
  typedef int ret;
};

int main()
{
  assert( sizeof(my_traits<int, char>::ret) == sizeof(int) );
  assert( sizeof(my_traits<char, int>::ret) == sizeof(int) );
};
```

The real traits are defined in my_traits_core. my_traits first tries the normal order, but if no specialization exists it tries the inverse order.

A.7 Exercise 3.1

```
#include <cassert>

struct End
{
  enum { size = 0 };
};

template <class Element, class Next>
struct List
{
  enum { size = 1 + Next::size };
};

int main()
{
  typedef List<int, List<char, List<float, End> > > mylist;
  assert( mylist::size == 3 );
}
```

No comment, this code is simple, isn't it?

A.8 Exercise 3.2

```
#include <cassert>

template <bool cond, class T, class U>
struct meta_if
{
  typedef T ret;
};

template <class T, class U>
struct meta_if<false, T, U>
{
  typedef U ret;
};

template <int Value, class Res, class Next>
struct meta_case;

template <class Res>
struct meta_default;

template <int Expr, class Cases>
struct meta_switch;

// Try one case
```

```
template <int Expr, int Value, class Res, class Next>
struct meta_switch<Expr, meta_case<Value, Res, Next> >
{
  // If Expr == value, return the value associated to the case.
  // Otherwise try next cases.
  typedef typename
  meta_if<Expr == Value,
          Res,
          typename meta_switch<Expr, Next>::ret>::ret ret;
};

// Default case
template <int Expr, class Res>
struct meta_switch<Expr, meta_default<Res> >
{
  typedef Res ret;
};

template <int expr>
unsigned foo()
{
  typedef typename meta_switch<expr,
                               meta_case<1, char,
                               meta_case<4, int,
                               meta_case<8, float,
                               meta_default<bool>
                    > > > >::ret result_type;
  return sizeof(result_type);
}

int main()
{
  assert( foo<1>()  == sizeof(char) );
  assert( foo<4>()  == sizeof(int) );
  assert( foo<8>()  == sizeof(float) );
  assert( foo<5>()  == sizeof(bool) );
  assert( foo<12>() == sizeof(bool) );
}
```

meta_switch<meta_case> analyzes the current case, in the value is not good, it continues on the next cases. If meta_default is encountered, the recursion stops and the corresponding type is returned.

(?)

Bibliography

Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison Wesley Professional.

Lesage, D. (2003). Scoop tutorial.

Veldhuizen, T. L. (1999). Techniques for scientific C++.