Student Paper

A profound reflection on C++ template mechanism

Author(s):
Baumann, Rainer

Publication Date:
2003

Permanent Link:
https://doi.org/10.3929/ethz-a-004770832 →

A profound reflection on C++ template mechanism

Rainer Baumann, ETH Zurich 2002/03
baumann@hypert.net, http://hypert.net/education

Contents

Preface

This paper provides a profound reflection on C++ template mechanism and is not a tutorial. The reader should be familiar with the basic constructs of C++ template mechanism as well as with the fundamental principles of STL as containers and interators. This paper is ISO standard compliant and covers mostly all template topics in the standard.

This paper originates from a semester work at the Swiss Federal Institute of Technology Zurich ETH (www.ethz.ch), Department of Computer Science (www.inf.ethz.ch), Computer Systems Institute, Group Prof. Juerg Gutknecht, Ph.D. Eugene Zueff.

I'd like to thank my sponsor and tutor Ph.D. E. Zueff for his support. I'm grateful that he has given me the opportunity to write this work.

1. Common aspects

Template mechanism is provided for functions and classes. Notice that there are three kinds of classes, ordinary classes, structs and unions. There exists no restriction for there use concerning templates. Remember that in structs and unions all members are public.

A class template shall not have the same name as anything else in the namespace. Function template names are not unique due to overloading.

Be warned, because of the complexity of templates for compiler writers the ISO standard often states that no diagnostics is required. That means that your compiler does not have to recognize some errors or analyse the reason of errors and that the program behaviour can be undefined.

2. Template parameter and argument

There are three different kinds of template parameters and arguments: type, non-type and template. All of them have their own special characteristics, which are discussed in the next paragraphs. First, let's have a look at some general terms.

2.0.1 Specialties and technical details

A template parameter can be given a default value. That allows simplifying template instantiation. But there are some roles to take care on when using them. After a parameter with default value it is not allowed to declare any further parameters with no ones. So you always have to declare the non-default ones before. Default parameters may never be declared twice, for example in a header file and an implementation or in a specialization.

```
template<class T, class U = int> class X;              // ok
template<class M = int, class N> class Y;              // error: parameter with no default value
                                                       // declared after one with default value
template<class T, class U = int> class X { … };        // error: default value declared twice
```

When a template is completely specified by default arguments, the argument list can also be empty but must be written out. Note that this is no argument deduction where the brackets cannot be omitted.

```
template<typename T = int> struct S { ... };
S<> s1;         // ok
S s2;           // error
```

2.1 Type

Template type parameters are the most often used once. For example you can build a generic container taking advantage of them.

```
template<typename type>
class Container {
            struct node { type val; node* next; };
            node* root;
      public:
            void push(type val) { node temp; temp.next = root; root = *temp; temp.val=val; }
            type push() { type temp = &root.val; root = root.next;}
};
```

Example t01.cc

The use of the STL container vector is demonstrated in *Example t02.cc.*

2.1.1 Specialties and technical details

Template type parameters are indicated with the keywords "class" or "typename" which can be used interchangeable.

```
class A { };
struct B { };
union C { };

template<class U> class X { };
template<typename U> class Y { };

main() {
      X<A> x1;        Y<A> y1;
      X<B> x2;        Y<B> y2;
      X<C> x3;        Y<C> y3;
      X<int> x4;      Y<int> y4;
}
```

Example t03.cc

The only nasty pitfall is to use a local unlinked or unnamed type as parameter.

```
template <class T> class X { … };
void foo() {
        struct S { ... };
        X<S> x1;                        // error: local type with no linkage used as parameter
        X<struct { … }> x2;             // error: unnamed parameter
}
```

If you overload a type template with another non-type one, any ambiguities are resolved by taking the type parameter. For Example

```
template<class I> void f() { cout << "classI"; };
template<int I>     void f() { cout << "int I"; };
main {
        f<int()>();             // prints class I
}
```

2.1.2 Chosen applications

Here a nice function template converter using type parameters. It converts any printable type to a string and backwards.

```
template<typename T> T fromString(const std::string& s);
template<typename T> std::string toString(const T& t);
```

Example t04.cc

Type parameters are often used in design patterns, e.g. in a factory as provided in *Example t05.cc.* A factory defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

2.2 Non-Type

Non-type parameters can only be of the following types: pointer and references to objects and functions, enumerations, booleans, characters and integers. Especially you shall not use floating points or voids.
Neither it is allowed to change or to assign to non-reference non-type parameters nor is it possible to take an address of such a parameter because they are statically compiled in.

```
template<typename type, int max>
struct S { type a [max]; };

main () {
      S<int,10> s1;
}
```
Example t06.cc

2.2.1 Specialties and technical details

As with type parameters it is not allowed to use local unlinked or unnamed values as well as addresses of array elements and non-static class members.

```
template<char *> class X { … };
struct S { int m; static int s; };
void foo() {
        X<”forbidden”> x3;          // error: unnamed parameter
        X<&s.m> x4;                 // error: address to non-static member
        X<&S::s> x5;                // ok
}
```

2.2.2 Type conversion

Non-type parameters can be explicitly adjusted what sometimes change the outcome to something unintended. Here is a list of the most common conversions.

Argument type	*Converted type*
numeric type	matching numeric type
array of T	pointer to T
function returning T	pointer to function returning T
pointer to class T	pointer to base class of T
reference to T	never converted

2.2.3 Chosen applications

Non-type parameters are mostly used for class templates. Than in functions a non-type value can be passed as argument. But there are still some reasons why passing values to functions as template parameters. Using template parameter, some calculations can be done during compile time what speeds up the execution of a program. If a function is often called with same parameters the list of them can be shortened.

Here a nice random generator, which uses a non-type parameter in a template class.

```
template<int upperBound>
class Urand {
                int used[upperBound];
                bool recycle;
        public:
                Urand(bool recycle = false);
                int operator()();                       // The "generator" function
};
```
Example t07.cc

2.3 Template

A template template parameter is a template taking as parameter a class template. This technique is still very rarely used but very powerful. Ordinary templates provide an abstraction of implementation detail, local data and data types. Template template parameters give the possibility to introduce additional level of abstractions. The largest example is the C++ Standard Template Library (STL).

It is important to realize that a template template parameter is an uninstantiated template.

```
template < typename T> class A;
template < template < typename T> > class B;
main() {
        B<A> x1;                // A is past as uninstanitated template
        B<A<int> > x1;          // A<int> is not a template, it is already a complete defined type
}
```

2.3.1 Chosen applications

Template template parameters can be very useful to give developers the facility to adapt the predefined behaviour of classes and functions according to their needs. For example you want to build an application where you log all calls to a sequence container as provided in the STL. The containers have to be interchangeable, so that also custom ones can be used. In consequence you have to use the corresponding iterator for each container.

```
template < typename val_t,
        template <typename T, typename A> class cont_t = std::vector>
class log_cont
{
                typedef std::allocator<val_t> alloc_t; //  can also be passed as template parameter
                cont_t<val_t, alloc_t> m_cont;          //instantiate template template parameter
        public:
                typedef typename cont_t<val_t, alloc_t<val_t> >::iterator iterator;
                iterator begin () { iterator i = m_cont.begin ();  /* write to log */  return i; }
                // more delegated methods...
};

main () {
      log_cont<char *, list> cont;                // create list container
      cont.push_back("cool");                     // push strings to vector
}
```

Example t08.cc

The first template parameter val_t is the type of the objects to be kept inside the store. Cont_t, the second one, is the template template parameter, which we are interested in. The declaration states that cont_t expects two template parameters T and A, therefore any standard conforming sequence container is applicable. We also provide a default value for the template template parameter, the standard container vector. When working with template template parameters, one has to get used to the fact that one provides a real class template as template argument, not an instantiation. The container's allocator alloc_t defaults to the standard allocator. Adding a further template parameter would allow to pass a non default container.

Template template parameters allow parameterising with incomplete types. That’s why they are often used for generic programming to achieve a high flexible in design. This is

a kind of *structural* abstraction compared to the abstraction over simple types achieved with usual template parameters.

Lets try to apply a corresponding abstraction to generic functions as above to generic containers. In *example t08*.cc class users were given a convenient way to customize a complex data structure according to their application contexts. Transferring this abstraction to generic functions, provides functions whose behavior is modifiable by their template template arguments.
Let's have a look at a generic printer for our container, which writes its content out in a customisable way:

```
using namespace std;

template <template <typename iter_t> class mutator, typename cont_t>
void printer(ostream& os, cont cont_t) {
        typedef typename cont_t::iterator iterator;
        mutator<iterator> mut;
        mut.do(cont.begin(),cont.end());
        copy(contbegin(), cont.end(), ostream_iterator<val_t>(os, " "));
};
```

Here, mutator is the template template parameter, it is expected to have an iterator type as its template parameter. The mutator changes the order of the elements that are delimited by the two iterator arguments and then prints the changed sequence.

A large field of applications are algorithms. You can build a generic quick sort in a way that you never have to code another one in your life. The type parameter T abstracts the element type. The comparator class with type abstraction U is responsible for comparisons between two elements of the type U. So this quick sort can be used with every type and every comparator.

```
template<typename T, template<typename T> class container,
           template<typename U> class comparator>
class quicksort {
                typedef container<T> cont;
                typedef typename container<T>::iterator iterat;
                void qs(iterat l, iterat r);
        public:
                void sort(cont& c) { qs(c.begin(), --c.end()); }
};

template<typename T> class greatorC;
template<typename T> class lessC;


main() {
        int a1 [] = {1,7,4,3,8,5};
        vector<int> v1;
        for (int i=0; i<6; i++) v1.push_back(a1[i]);
        quicksort<int, vector, greatorC> qsInt;
        qsInt.sort(v1);
}
```

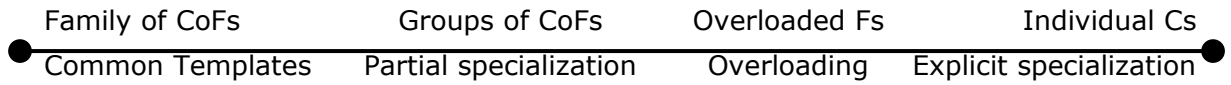
Example t10.cc

A bit a simpler quick sort implementation for arrays is provided in *Example t09*.cc.

Two further examples are provided. *Example t11.cc* is a rudimentarily implemented library for arbitrary precision arithmetic. And *Example t12.cc* is a function invocator. It is abandoned to the interested reader to study them.

3. Template specialization and overloading

A general template works very well for the majority of applications but in some special cases there is a problem. A special handling is required for these ones. The template specialization and overloading mechanisms lets you differ such cases. Lets first have a look at the levels of specialization: Templates form families of classes or functions. Partial specialization enables one to define more specialized code for subgroups of such families. Explicit specialization lets one define individual code for single members. For function templates overloading is provided instead of partial specialization, because overloading is a very well known concept. Overloading provides one with nearly the same possibilities as partial specialization but in its nature it is closer to explicit specialization.



Legend: CoF := Class or Function; F := Functions; C := Class

In this section the term of "primary template" is often used. A primary template is a general template with no specialization.

3.0.1 Meaning of "specialization"

While reading the ISO standard or a related document it is very important to understand that the word "specialization" has two meanings. A specialization is an explicit or partial specialization, redefining code for special cases. But a specialization also happens during instantiation of a template. That means that the template code is specialized by the compiler before using it. Both cases are denominated the same.

3.1 Explicit specialization

Explicit specialization is a complete specialization. Therewith implementations can be customised for specific template parameters. It may be declared for a function template, class template, a member or a class template or a member template. It is introduced by "template <>".

```
template<class T> void bar() { };       // primary function template
template<> void bar<int>() { };         // specialization for T == int


template<class T> class A { };          // primary class template
template<> class A<int> { };            // specialization for T == int


template<class T> class B {
            void foo() { };                     // general member
};
template<> void B<int>::foo() { };              // specialization of member for T == int


class C {
      template<class T> void zac() { };         // general member template
};
template<> void C::zac<int>() { };              // specialization of member template for T == int
```

Example t13.cc

Notice that in a class template specialization no content is derived from the primary template. So every thing has to be recoded.

3.1.1 Specialties and technical details

Explicit specialization of nested templates can be done as expected.

```
template<class T> class D {
      template<class U> class E {
            void muk();
      };
};

template<> template<> void D<int>::E<int>::muk() { };
        // ok, total specialization of muk()
template<> template<class U> void D<int>::E<U>::muk() { };
        // ok, specilization of muk() for D<int>
template<class T> template<> void D<T>::E<int>::muk() { };
        // error, specialization of a member template without specialization of enclosing template
```

Example t13.cc

It should be obvious that a template cannot be specialized before the primary template is even declared. The specialization should also be declared in the same namespace or for member templates in the namespace of the enclosing class. A specialization must be declared before the first use or an explicit instantiation.

```
template<> class A<int>;      // error, specialization before primary template declaration

template<class T> class A;

A<char> a;

template<> class A<char>;    // error, specialization after explicit instantiation
```

Argument deducing can also be used for explicit template specialization. *(More about argument deducing can be found in a separate paragraph.)*

```
template<class T> class Array;
template<class T> void sort(Array<T>& a);

template void sort(Array<int>&);              //ok, explicit specialization with argument deducing
```

Notice that the place of explicit specialization combined with partial specialization can affect the program behaviour. The order of files for compilation can affect exported templates.

No template shall be instantiated more than once.
No template shall be specialized with the same set of template parameters more than once.
It is not allowed to specify any function default argument in any specialization.

An ambiguity can occur with overloading functions and explicit specialization. *(For more detail look at the section "Ambiguity".)*

3.1.2 Chosen applications

Every experienced programmer realizes the helpfulness of specialization. Here are two nice applications.

You can use it for recursive compile time computation, e.g. to compute the Fibonacci numbers.

```
template<unsigned i>
struct fibonacci {
	static const unsigned result = fibonacci<i-1>::result + fibonacci<i-2>::result;
};

template<>
struct fibonacci<0> {
	static const unsigned result = 1;
};

template<>
struct fibonacci<1> {
	static const unsigned result = 1;
};

main() {
	cout << fibonacci<9>::result;
}
```

Example t14.cc

Note that this code snipped looks very close to the mathematical definition:
f(i) = f(i-1) + f(i-2)
f(1) = 1
f(0) = 1

Specialization is very helpful for design patterns, e.g. adapters. In the example below, we want to store a new integer in a list or in a container. Lists provide the function add and containers the function insert to store an integer. This is very bothering, so we always have to differ which we have to call. The template class adapter solves this problem translating a request in its corresponding request to the specific call.

```
class list {
	public:
		void add(int val);
};

class container {
	public:
		void insert(int val);
};


template<class T>
class adapter {
		void add(T cont, int val) { cont.add(val); };
};

template<> void adapter<container>::add(container cont, int val) { cont.insert(val); };
```

Example t15.cc

3.2 Partial specialization of class template

Partial specialization gives the ability to provide specialized class templates for some specific parameters. The declaration of partial specialized class template looks something between a declaration of a normal template and a declaration of an explicit specialization. With empty brackets after template it would be an explicit specialization and without brackets after the class name a common template.

Partial specialization is mostly used for type parameters because of its possibility for individual stepwise specialization, e.g. type X specialised to pointer to type X. But it also can be used for explicit specialization of individual non-type parameters. For simplicity the mechanism of partial specialization is first demonstrated with non-type parameters.

```
template<int l, int w, class T> class image;              // primary template
template<int w, class T> class image<320, w, T>;          // partial specialized template for l == 320
template<int a, class T> class image<a, a, T>;            // partial specialized template for l == w
template<int l, int w, class T> class image<a, w, T*>;  // partial specialized template, T as pointer
```

Notice that partial specialization only works with class templates. For function template overloading provides nearly the same possibilities. If this is not enough a rapper class has to be used.

3.2.1 Specialties and technical details

It should be obvious that a primary template must be declared before it could be specialized. A partial specialization must be declared before the first use or explicit instantiation.

A partial specialization must be a real specialization and not just a rewriting of the primary template. For non-type parameters, no expressions are allowed in template arguments. A non-type parameter which type depends on a template type parameter shall not be specialized without specialization of the corresponding type.

```
template<int l, int w>  class image;                  // primary template
template<int l, int w>  class<l, w> image;            // error, no real specialization
template<int a, class T> class image<2*a, a+3>;       // error, expression in non-type argument

template<class T, T t> class A;                       // primary template
template<class T> class A<T, 1>;                      // error, dependant
                                                      // non-type parameter specialization
```

It is not allowed to specify template default parameters in partial specializations.

An ambiguity can occur with partial specialization. *(Details can be found in the section "Ambiguity".)*

If you want to specialize just a part of a template class, not the whole template, and explicit specialization is not what you want, than a derived partial specialization is what you need as shown below.

```
template<class U> struct A {
        int i;
        void print();
};

template<class U> struct A<U*> : public A<U> {
        void print();
};
```

Example t16.cc

3.2.2 Chosen applications

Very often a template has to handle objects and pointers to objects. It's evident that a template cannot treat them in the same way. So a partial specialization is the right tool. The example below provides a sort algorithm, which accepts objects and pointer to objects.

```
template<class T> struct Sorted : public std::vector<T> {
        void sort();
};

template<class T> struct Sorted<T*> : public std::vector<T*> {
        void sort();
};
```

Example t17.cc

This template demonstrates the power of partial specialization to perform a non-trivial computations on types. The template computes the number of dimensions of a type.

```
template<class T> struct Rank {
        static const int value = 0;
};

template<class T, int N> struct Rank<T[N]> {
        static const int value = 1+Rank<T>::value;
};

main() {
        typedef float a;                    // dim 0
        typedef float b[10];                // dim 1
        typedef float c[20][30];            // dim 2
        printf( "%d %d %d\n", Rank<a>::value, Rank<b>::value, Rank<c>::value ); // out: 0 1 2
}
```

Example t18.cc

Here an if then else construct returning the appropriate type.

```
template<bool condition, class Then, class Else>
struct IF { typedef Then RET; };

template<class Then, class Else>
struct IF<false, Then, Else> { typedef Else RET; };

main() {
      cout << sizeof(IF<(1+2>4), short, int>::RET);
}
```

Example t19.cc

Below an accumulator for classes is presented.

```
struct Square{ template<int n> struct Apply; };
template<int n, class F> struct;
template<class F> struct Accumulate<0,F>;
template<> struct Accumulate<0,Square>;
```

Example t20.cc

3.3 Overloading template function

It is not possible to specialize partially function templates. But overloading is provided for them in nearly the same way as for ordinary functions. This means you can define multiple functions and template functions with the same name, but different signature. The signature of a template function consists of its function signature, its return type and its template parameter list.

```
void foo(int*);                          // ordinary function
template<class T> void foo(T);           // template (overloading foo)
template<class T> void foo(T*);          // template (overloading foo and foo<T>
template<class U> void foo(U);           // error, template with same signature already exists
```

3.3.1 Specialties and technical details

As stated, the signature also embeds the return value. In the example below the return value between line one and three is different. E.g. f<2,1>(…) has A<3> or A<1> as return value.

```
template<int i, int j>   A<i+j>   f(A<i>, A<j>);       // #1
template<int l, int m> A<l+m> f(A<l>, A<m>);       // error, same signature as #1
template<int i, int j>   A<i-j>   f(A<i>, A<j>);       // ok, different return value
```

It is allowed to declare a function template several times. If they have the same signature (#1,#2) and look the same they were linked together. If they have different signatures (#1,#3) they are recognized as overloading. But if they have the same signature but don't look the same the program is ill formed and the compiler does not have to detect this error.

```
template<int i> void f(A<i+5>);        // #1
template<int i> void f(A<i+5>);        // #2, same declaration as #1
template<int i> void f(A<i+7>);        // #3, different declarations #1
template<int i> void f(A<i+2+3>);      // #4, wrong, but not detected
```

3.3.2 Chosen applications

In a generic quick sort with median of three a specific comparator can be specified (#1). An overloading function template (#2) is given, using common comparison. In (#3) even an instance of the comparator is passed.

```
template<typename T, typename Compare> (T &) median (T &a, T &b, T &c); // #1
template<typename T> (T &) median (T &a, T &b, T &c);                   // #2
template<typename T> (T &) median (T &a, T &b, T &c, Compare comp);     // #3
```

Example t21.cc

4. Instantiation

Instantiation is the act of creating real useable code from templates for specific template parameters. So it should be logic that a template must be defined and not only declared before instantiating.

This topic is very wide and mostly interesting for compiler writers. The important compiler writing aspects are covered in the section "Linkage, compilation and exporting templates".

4.1 Implicit instantiation

A template is implicitly instantiated by the compiler when it really needs the corresponding code. (E.g. for a pointer to a template no code is needed.) That also means that for class templates not the whole template is instantiated, only the parts required.
Function templates are instantiated when they are used or their addresses are taken. For function templates arguments can be deduced. More about argument deduction can be found in the section "Argument deducing and matching".
Class templates are instantiated when an object of the template is defined. For class templates no argument deduction exists. So the template parameters must be explicitly specified, except the one having a default value.

```
template<class T> class X { };

main() {
        X<int> x;       // implicit instantiation, object declaration
}
```

4.2 Explicit instantiation

A template can also be instantiated explicitly. An explicit instantiation of a template instantiates explicitly all its members not yet instantiated explicitly or specialized explicitly. (For explicitly specialized templates, instantiation makes no sense because this is already concrete usable code.) Explicit instantiation looks similar to declaration, always starting with the keyword "template".

```
template<class T> class X { };                          // class template definition
template<class T, class U> T foo(T *tp, int n) { };     // function template definition

template class X<int>;                                  // explicit instantiation
template long foo<long, double>(long *, int);           // explicit instantiation
```

It should be logic that a declaration must be given first before explicit instantiating a template. A template should never be instantiated explicitly more than once.

This mechanism is required in several cases:
- With explicit instantiation a programmer can decide in which object file a specific template instance will be stored. Like this linkage between object files can be reduced.
- When a program is assembled from several compilation units, one compilation unit could need a specific template instance of another compilation unit, which doesn't instantiate this specific template instance. With explicit instantiation this specific template instance can be provided.
  As a consequence explicit instantiation is often used for building Libraries.
  Don't mix up exporting with explicit instantiation. In exporting the non-instantiated template is exported.

5. Argument deducing and matching

When a template is referenced two things have to be done. If more than one template could be chosen the best one has to be found. If not all template parameters are specified they have to be deduced.

5.1 Referring to function templates

A template argument list may be specified when referring to a function template. Trailing template arguments, which can be deduced or have default values can be omitted. If the arguments brackets are empty they can be omitted them self.

```
template<class V, class U> void convert(V v, U u) { };

void foo(int i, double d) {
        convert<int, double>(i, d);     // all parameter explicit specified
        convert<int>(i, d);             // U deduced from d
        convert<>(i, d);                // U & V deduced
        convert(i, d);                  // U & V deduced, brackets omitted
}                                       // all calls refer to convert<int, double>
```

But be warned, just omitting the template arguments can cause deduction compilation problems.

```
template<typename T> void f(T,T);

main() {
      char *s ;
      char const *cs ;

      f(s, cs);                   // error, no matching function for call to `f(char*&, const char*&)'
      f<char const*>(s, cs);      // ok
}
```

Example t22.cc

5.1.1 Explicit argument specification

In common function calls implicit conversion is performed on a function argument to convert it to the type of the corresponding function parameter. The same happens when a type is explicit specified with a template argument.

```
template<class T> void bar(T t) { };

void foo(double d) {
        bar<int>(d);            // converts double -> int : bar(int(1.0)) is called
};

main() {
        foo(1);                 // converts int -> double : foo(double(1)) is called
}
```

When types are specified explicitly in template arguments, the function argument types must be compatible with them otherwise argument deduction will fail. There are some other rules with the objective of preventing illegal constructs to be deduced.

```
template<class T> void bar(T t) { };

main() {
        bar<int>("hallo");      // error, invalid conversion from `const char*' to `int'
}
```

5.1.2 Argument deducing

Argument deducing is not so easy and the ISO standard needs about eight pages to describe it. Let's try to understand it without every finical detail.

All arguments of type array or function are converted to pointers before starting argument deducing (standard conversion).
In turn, the template parameters are identified in the argument of the called function. For each parameter found, the type, value or template is deduced from the template function's argument. If a parameter is found several times, the deduced parameter must be the same.

There are three kinds of conversions used for argument deducing: lvalue transformations qualification conversions, conversion to a base class.

Note that the type of the return value is not considered in deducing.

5.1.2.1 Lvalue transformation

There are tree types of lvalue transformations:

Lvalue-to-rvalue conversions: Simply stated, a lvalue is an expression that may be used to the left of an assignment operator. It is an object whose address may be determined, and which contains a value. In contrast, a rvalue is an expression that may be used to the right of an assignment operator: it represents a value that does not have an address and that cannot be modified. In a statement like x = y; (in which x and y are variables of comparable types), the value of y is determined. Then this value is assigned to x. Determining the value of y is called a lvalue-to-rvalue conversion. A lvalue-to-rvalue conversion takes place in situations where the value of a lvalue expression is required. This also happens when a variable is used as argument to a function having a value parameter.

Array-to-pointer conversions: An array-to-pointer conversion occurs when the name of an array is assigned to a pointer variable. This is frequently seen with functions using parameters that are pointer variables. When calling such functions, an array is often specified as argument to the function. The address of the array is then assigned to the pointer-parameter. This is called an array-to-pointer conversion.

Function-to-pointer conversions: This conversion is most often seen with functions defining a parameter, which is a pointer to a function. When calling such a function the name of a function may be specified for the parameter, which is a pointer to a function. The address of the function is then assigned to the pointer-parameter. This is called a function-to-pointer conversion.

5.1.2.2 Qualification conversions

A qualification conversion adds const or volatile qualifications to pointers. This can be used to prevent any modifications of an argument

```
template<class T> void f(T const *t) {
	int i = t[2];		// ok, just reading
	t[4] = 6;		// error, assignment of read-only location
};

main() {
	int ia[5];
	f(ia);			// qualification conversion
}
```

Example t23.cc

5.1.2.3 Conversion to a base class

Like common classes, template classes can be derived. Argument deducing accepts a derived class instead of the base class.

```
template<class T> class Base { };
template<class T> class Derived : public Base<T> { };

template<class T> void foo(Base<T> b) { };

main() {
	Base<int> b;
	foo(b);
	Derived<int> d;
	foo(d);
}
```

Example t23.cc

5.1.3 Chosen applications

As known it is not possible to determine the length of an array during runtime. But during compile time a compiler sometimes knows it and it can be deduced. The (&a) prevents the compiler to convert the array to a pointer as described in (4.1.2).

```
template<typename T, int n> void foo(T (&a) [n]) { };

main() {
	int a[10];
	foo(a);
}
```

Example t24.cc

5.2 Matching of overloaded (template) functions

When a call to an overloaded function is performed a set of candidate function is constructed. When the template brackets are used, even empty, only template functions are considered.

All corresponding non-template functions with compatible types are added to the set. For all corresponding templates argument deduction is tried and if successful the deducted specialization is added as well. If there is more than one candidate than the partial ordering algorithm has to decide which one is the most specialist and this one is chosen. If there are a template and a non-template candidate, which both matches the call exactly, the non-template function is taken.

```
template<class T> inf f(T);     // #1
inf f(int);                     // #2

int  k = f(1);          // uses #2, 1 and 2 matches exactly so the non-template function is taken
int  l = f<>(1);        // uses #1, only templates considered because of <>
```

5.2.1 Partial ordering of function templates

Partial ordering is, as described in the ISO standard, under specified and suffers from a number of problems. There is wide discussion of interpretation and revision. Anyway let's have a look at it but with no guarantees.

Partial ordering decides which of two function templates is more specialized than the other by transforming each template in turn (see next paragraph) and performing template argument deduction using the function parameter types, or in the case of a conversion function the return type. If the deduction succeeds, the transformed function is said to be at least as specialized as the other. If one function is at least as specialized as the other, and the other is not at least as specialized as the first, then the first is more specialized.

To produce the transformed template, for each type, non-type, or template template parameter synthesize a unique type, value, or class template respectively and substitute that for each occurrence of that parameter in the function type of the template. This is all very confusing on a first read but with an example it is much clearer.

```
template <class T> void foo (T *);      //a
template <class T> void foo (T &);      //b

main() {
        int* p;
        foo (p);
}

a: foo (T *) -> foo (X *), using foo (T &) deduce T to be X *, deduction succeeds.
b: foo (T &) -> foo (X &), using foo (T *) deduction fails

so b is less specialized than a, and void foo(T*) [with T = int] should be selected
```

5.3 Matching of class template partial specializations

When a class template is instantiated, it is necessary to detect where it should be generated from, based on the template arguments.
- If exactly one matching specialization is found. This one is used.
- If more than one matching specialization is found than the partial ordering algorithm has to decide which one is the most specialist and this one is chosen.
- If no matching specialization is found, the primary template is used.

A template arguments of a specialization matches if they can be deduced from the argument list. *(More about deducing is provided in a separate paragraph.)*

5.3.1 Partial ordering of class template specialization

Partial ordering is defined between two templates. So it is decidable which is more specialized. Partial ordering for class templates is derived from partial function template ordering in the following way. The class templates are converted to relating function template. The class template relating to the more specialized function is the more specialized one.

```
template<int i, int j, class T>   class X;                  // primary template
template<int i, int j>            class X<i, j, int>;       // #1
template<int i>                   class X<i, i, int>;       // #2

template<int i, int j>    void f(X<i, j, int>);             // #A, relating to #1
template<int i>           void f(X<i, i, int>);             // #B, relating to #2
```

5.4 Compiler support

Example t25.cc provides a simple compiler checker for this topic.

6. Statics in template

A template class can contain static members (data and functions) as in non-template classes. In non-template classes only one instance of that member exists, shared by all instances of the class. But in template classes there exists an unique member for every instantiated combination of template parameters, shared by all instances with the same template parameters. Static members must be initialised outside the class declaration.

```
template<class T>
class X {
      public:
            static T s;
};

template<class T> T X<T>::s = 1;
template<char> char X<char>::s = 'a';
template<> double X<double>::s = 2.0;              // with argument deducing

X<int> x1;              // x1.s = 1
X<char> x2;             // x2.s = a
X<double> x3;           // x3.s = 2.0
X<int> x4;              // x4.s = 1
```

Example t26.cc

7. Member templates

A template declared within a class or class template is called a member template. Every function template or class template can be a member template. They can be defined in- or outside the containing class or class template.

```
template<class T>
class Outer {
            template<class U> class Inner1;
            template<class V> class Inner2 { /* ... */ };
            template<class W> void foo1(W);
            template<class X> void foo2(X) { /* ... */ };
};

template <class T> template <class U> class Outer<T>::Inner1<U> { /* ... */ };
template<> template<> void A<int>::foo1<>(int i) { };
```

Example t27.cc

Even a constructor is allowed to be a member template but a destructor shall never be one.

```
template<class T>
class Outer {
      public:
            template<class Y> Outer(int i) { /* ... */ };        // ok
            template<class Z> ~Outer() { /* ... */ };            // error, destructor as template
};
```

Example t28.cc

Member templates shall never be virtual.

```
template<class T>
class Outer {
//            template<int i> virtual void bar();     // error, virtual member template
};
```

Example t28.cc

Every class declared in a function definition is called a local class. Such classes shall not have member templates.

```
void foo() {
        class A {
                template<class Y> class X;      // error, template in local class
        };
};
```

Example t28.cc

It does not make any sense to code member templates in functions or function templates. This would just be artificial. All required information can and must be passed with the function call. So no member template ever can be instantiated with other information from outside.

7.1 Nested templates

Nested templates are class templates with member templates. First its very important to understand when there is a real need for nested templates. Mostly the template parameters of the member template are already known at instantiation time of the enclosing template class. So there is no need for nested templates and all required information can already be passed with the instantiation of the enclosing template class.

7.1.1 Chosen applications

Nested templates can be very useful to change the behaviour of class members. Lets assume we have a template taking any container as template parameter. Now we want to print the content of such a container in different orders. We could provide a different print function for each of them. But this is defective. Every time a new order is introduced, a new function has to be added. A much cleverer way is to pass an iterator to the print function to define the order.

```
template<typename cont_t>
struct C {
        cont_t cont;
        template<typename iter_t> void print();
};
```

Nested templates can also nicely be used with constructors. The example below is an adaptation of a class found in the Standard Template Library. Note that an object of class Pair<long, long double> is constructed from an object of class Pair<short, float>. By using a template constructor it is possible to construct a Pair from any other Pair, assuming that conversion from T to A and U to B are supported. Without the availability of template constructors one could only declare constructors with fixed types like "Pair(int)" or else use the template arguments to Pair itself, as in "Pair(A, B)".

```
template <class A, class B> struct Pair {
        A a;
        B b;
        Pair(const A& ax, const B& bx) : a(ax), b(bx) {}
        template <class T, class U> Pair(const Pair<T,U>& p) : a(p.a), b(p.b) {}
};

main() {
        Pair<short, float> x(37, 12.34);
        Pair<long, long double> y(x);
        printf("%ld %Lg\n", y.a, y.b);
}
```

Example t29.cc

8. Friends

Elements of a class can have the access levels private, protected or public. The one with private or protected can normally not be accessed from outside the same class at which they are declared. The keyword "friend" provides a possibility to bypass this restriction.

```
class A {
      int i;                                    // private
      friend class X;
      friend void foo();
};

A a;
class X { public: void set() {a.i=5;}; };       // ok
class Y { public: void set() {a.i=6;}; };       // error
void foo() { a.i=7; };                          // ok
```

8.1 Friend of template

Template classes can have friends as common classes but it is a bit trickier because of specialization. But this is very logic. The following parts are a bit drawn-out because nearly all possibilities are covered.

But in advance some remarks:
As with non-template friends every friend must be declared in the same name scope.
Never declare any partial specialization in a friend declaration.

8.1.2 Friend class of template

Every template and non-template class can be a friend of a template class. Worth mentioning is that in a common class all instantiations are friends of each other but not in template classes. The instances of X<int> are not automatically friends of X<char>.

```
class A;
template<class P> class B;
template<class P> class C;

template<class T>
class X {
      int i;
      friend class A                            // ok
      template<class P> friend class B;         // ok, every instance of B is a friend
      friend class C<T>;                        // ok, only instances of C with the same template
                                                   parameter as for X are friends
//     friend class X<T>;                       // error, makes no sense, X<T> is always a friend
                                                   of every other X<T>
      friend class X<char>;                     // ok, every X<T> has also X<char> as friend
};
                                                                          Example t30.cc
```

In template classes it is also allowed to declare and define member friend non-template classes. But it is not allowed to define member friend template classes. Only the declaration of them is fine.

```
template<class T>
class X {                                          // could also be a non-template class
      friend class A1;                             // ok, declaration of member class
      friend class A2 { int i; };                  // ok, definition of member class
      template<class U> friend class B1;           // ok, declaration of member template class
      template<class U> friend class B2 { T t; };  // error, not allowed definition of
                                                      template member class
};
```

Example t31.cc

8.1.2 Friend function of template

Take a template class X and its specialization X<int>. X has a friend function foo, so foo is a friend of all specializations of X, which are not defined separately e.g. X<char>. But foo is not a friend of X<int> because it is not declared to be a friend in the specialized definition of X<int>. But bar is declared to be a friend of X<int>.

```
template<class T>
class X {
      int i;
      friend void foo();
};

template<>
class X<int> {
      int i;
      friend void bar();
};


void foo() {
      X<char> xc;     xc.i = 2;     // ok
//     X<int>  xi;      xi.i = 3;     // error, not a friend of the partial specialization X<int>
};

void bar() {
//     X<char> xc;  xc.i = 4;       // error, not a friend of the primary template X
      X<int>  xi;     xi.i = 5;       // ok
};
```

Example t31.cc

Certainly is it allowed to give friend functions any parameters you like.

```
template<class T>
class X {
      int i;
      friend void mik(X<T>);
};

void mik(X<char> x) { x.i= 7; };       // ok
```

Example t31.cc

Also template functions can be friends of template classes. In the example below the function sod is always a friend of the given instance of X. But ral is only a friend of instances with type X<bool>.

```
template<class T>
class X {
	int i;
	friend void sod<T>(X<T>);
	friend void ral<bool>(X<bool>);
};

template<class T> void sod(X<T> x) { x.i= 6; };
template<class T> void ral(X<T> x)  { x.i= 8; };

main() {
	X<char> xc;
	X<bool> xb;

	sod(xc);		// ok
	sod(xb);		// ok
//	 ral(xc);		// error, ral is only a friend of X<bool>
	ral(xb);		// ok
}
```

Example t31.cc

It is also allowed in template classes to declare and define member friend templates and non-template functions. But be aware that a member template function you define is well formed for all used instances of X<T> even when the function is never used.

```
template<class T>
class X {                                                  // could also be a non-template class
	friend void foo1();                                    // ok, declaration of member function
	friend void foo2() { /* */ };                          // ok, definition of member function
	template<class U> friend void bar1();                  // ok, declaration of member
	                                                          template function
	template<class U> friend void bar2() { /* */ };        // ok, definition of member
	                                                          template function
};
```

Example t30.cc

8.2 Template as friend

Specialized or non-specialized template classes or functions can be declared being a friend of a non-template class as common ones. *(Details of usage can be found in the part above about "Friend of template".)*

Anyhow some remarks. A member of a class template can be declared being a friend as in non-template classes.

```
template<class T> class A {
	public:
		void foo();
};

class B {
	template<class T> friend void A<T>::foo();
};
```

Example t32.cc

9. Names

In templates names and their lookups are a bit different because of the template parameters.

9.1 Typename

The keyword "typename" is only used within templates. When using a template parameter or a template name in a nested name as a type. This is because the compiler does not recognize them as types because they are not real types, they are rather a family of types.

```
template<class T> class A  {
        class B;
        typename A::B b1;
        typename A<T>::B b2;
        typename T::s s1;
}
```

9.2 Local declared names

A template parameter hides, from the place of its declaration, any entity with the same name until the end of its template. It shall not be redeclared within its scope.

```
struct T { };

template<class T, int v> struct B {
        T t;                            // B's T
        char v;                         // error, template parameter redeclared
};
```


```
struct T { };

template<class T, int v> struct B {
        T t;                            // B's T
        char v;                         // error, template parameter redeclared
};
```


```
class T { … }
int i;
template<class T, T I> void f() {
        T t1 = i;                       // template namespace T & i
        ::T t2 = ::i;                   // global namespace T & i
}
```

In a template-definition the brackets for self-referencing can be omitted.

```
template<class T> class A {
      A* x;                     // reference to A<T>
};

template<> class A<int> {
      A* x;                     // reference to A<int>
      A<bool>* x;               // reference to A<bool>
};
```

A template parameter can be used from the point of where it is declared till the end of the template, even in the declaration, but not before.

```
template<class T, T* p> class X { … };        // ok
template<T* p, class T> class X { … };        // error: T used before it is declared
```

9.3 Dependent names

Names, which do not depend on a template parameter, are looked up in the definition and instantiation context, dependent names only in the definition context.
If a member is defined outside of its containing class the names defined in the containing class are not considered till instantiation.

10. Ambiguities

There are two kinds of ambiguities. Grammatical ones, coming form the abstract definition of tokens and a resolving ones, provoked by the programmer.

10.1 Grammatical ambiguities

The ISO standard allows using any "expression" or "type_name" as template argument. This use of expression is where the most ambiguities in templates come from. You can protect yourself from annoying compiler errors when you follow two simple roles:

1. Always put brackets around expressions in template parameters
2. Always leave a space between the ">" when it is used in a template arguments or parameters

```
template<bool x> class A;
template<class Y> class B;

main() {
        A<2>3> x1;          // error, first ">" interpreted as closing bracket
        A<(2>3)> x1;        // ok, rule 1
        B<A<1>> x1;         // error, ">>" interpreted as shift operator
        B<A<1> > x1;        // ok, rule 2
}
```

There is also another ambiguity problem with member templates. After ".", "->" and "::" the "<" is interpreted as less than. To solve this, the keyword "template" has to be used.

```
class C{
      public:
            template<int i> void foo() {};
            template<int i> static void bar() {};
};

main() {
      C c1;
      C* c2;
      c1.foo<3>();                  // error, "<" means less than
      c1.template foo<3>();         // ok
      c2->foo<3>();                 // error, "<" means less than
      c2->template foo<3>();        // ok
      C::bar<3>();                  // error, "<" means less than
      C::template bar<3>();         // ok
}
```

t33.cc

10.2 Resolving ambiguities

Most of template specific ambiguities are solved due to ordering and other special rules.

An ambiguity can occur with overloading functions and explicit specialization.

```
template<class T> void foo(T);
template<class T> void foo(T*);

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

Another ambiguity occurs with partial specialization.

```
template<int l, int w> class image { };          // primary template
template<int i> class image <i,200> { };         // partial specialization for w == 200
template<int w> class image <300,w> { };         // partial specialization for l == 300

main() {
      image<3,2> x1;            // ok, using primary template
      image<3,200> x2;          // ok, using primary partial specialization for w == 200
      image<300,2> x3;          // ok, using primary partial specialization for l == 300
      image<300,200> x4;        // error, ambiguous
}
```

11. Linkage, compilation and exporting templates

A Deep C++ article of Microsoft starts with: "export and Exported Templates
The horror. The horror." and in the ISO standard committee a holy war had been fought for a long time about template compilation model. Although this topic is still hot in today C++ compiler development, the topic is well understood and not so difficult.

For a compiler the processing of template classes and methods differ totally from processing common ones. A template is a description of a pattern for a family of related classes and functions. So a template definition cannot be translated because the compiler lacks of the template parameters. But a compiler will be capable to do the translation for concrete template instantiations. But there are some important points to look at:

- It is desirable to end up with only one copy of each instantiated entity in a program.
- It is essential always to take the most specialized template definition.
- A not referenced template shall never be compiled because it could be ill formed.

Now let's have a look at the two different compilation models.

11.1 Inclusion model with duplicate elimination

The main idea of the inclusion model is just to include the code for a template instance everywhere it is referenced. Then it uses a smart linker, which removes all duplicate code of instantiated functions. This straightforward approach works very well. But it tends to blow up object files.

Now, let's explore the usage of this model. If just one code file with no header file is used, there is nothing to bee considered and the compilation works well. Typically for C++ is to split interface and implementation using header files for interface declarations. This header files can then be included in several implementation files. For a template to be accessible in several implementation files they are also included in the header files. But in difference to classes and other objects with external linkage the definition of templates have to be provided within this header files.

```
file1.h:
        template<class T> class X { /*  */ };
caller1.cc:
        #include "file1.h"
        X<int> x;
```

A possibility to separate declaration and implementation is the following.

```
file2.h:
        template<class T> class X;
        #include "file1.cc"
file2.C:
        template<class T> class X { /*  */ };
caller2.C:
        #include "file2.h"
        X<int> x;
```

Some compilers permit to leave away the include in the header file and, if needed, automatically include the associated file with the same base name.

This model is relatively simple to understand. But it pretends a not existing separation of name space between caller.cc and file1.cc. So every thing in both files belong to the

same name space if not declared explicitly elsewhere. Also don't forget that macros stated in caller.cc will act on file.cc.

Some compiler using this model even restricts template only to be declared and defined in header files.

The ISO standard fully supports this model and used it for its Standard Template Library.

11.2 Separate compilation model with trial linking

In the separate compilation model no templates are instantiated during normal compilation. All information about the compiled files is stored in a separate directory. During linkage the undefined references of templates are discovered. Then the linker consults for each of them the information directory to find the source of the entity, compiles it and finally links it with the normal object code.

11.2.1 The historical realization

The historical realization is the following. All templates must be declared in ".h" header files and the corresponding definition must be in a ".C" file with the same base name. So the linker just has to change the suffix to find the implementation. In some compiler the suffix can be changed using compiler directives.

```
file3.h:
        template<class T> class X;
file3.C:
        template<class T> class X { /*  */ };
caller3.C:
        #include "file3.h"
        X<int> x;
```

This historical realization is not ISO standard compliant. With an ISO standard compliant compiler the example above would be compiled using the inclusion model with duplicate elimination.

11.2.2 ISO's way

The ISO standard committee agreed on using a new keyword "export" to mark the usage of the separate compilation model. The "export" essentially indicates to the compiler that the template should be made available for use outside the current translation unit.

```
file4.h:
        template<class T> class X;
file4.C:
        export template<class T> class X { /*  */ };
caller4.C:
        #include "file4.h"
        X<int> x;
```

The standard grants for higher flexibility that the keyword "export" could be used in front of a template definition or declaration with the same effect.

The standard does not specify how the compiler finds the corresponding definition for a template. The compiler can find it in a magic way. Possible solutions to this are explicit

compilation of the template definition file before compiling the main program, manual instruction to the compiler or a similar one as in the historical realization.

Now let's have a look at some details.
The behaviour of the compiler is undefined if two exported templates with the same signature exist.
A template function which is declared inline and export is just inline.
When a specialization is not visible in the instantiation context and would affect the instantiation the program is ill formed but the compiler does not have to realize this.

```
file5.h:
        template<class T> class X { /*  */ };
        template<class T> void foo(T);
file5.C:
        export template<class T> void foo(t) { X<T> x; };
caller5.C:
        #include "file5.h"
        template<> class A<int> { /*  */ };
        main { f<int>(1); }
```

This model solves the major problems having with the inline model but also introduces new ones. The most important is the non-intermediate context limitation.

```
ic1.h:
      export template <class T>
      void g(const T&);
ic1.C:
      export template <class T>
      void g(const T& t)
      {
            length(t);                    // how does length get found?
      }
ic2.h:
      export template <class T> void f(T);
ic2.C:
      #include "ic1.h"

      template <class T>
      class Container { ... };

      export template <class T>
      int length (const Container<T>&) { ... }

      export template <class T> void f(T t) {
            Container<T> s;
            g(s);
      }
ic3.C:
      #include "ic2.h"

      class A { ... };

      void m() {
            A a;
            f(a);                         // this starts the instantiations
      }
```

Here the compiler looks up for the function length in the definition context of ic1.C as well as in the instantiation context of ic3.C but not in the intermediate one of ic2.C. The only way to make this work is a rearrangement of the file structure.

11.3 Historical

The earliest compiler supporting templates was the cfront from AT&T/USL/NOVEL/SCO. It supported the historical model of separate compilation, was easily confused and slow. So when PC vendors started providing C++ template compiler they developed the inline model for better performance. During standardization by ANSI and later ISO hot fights were carried out about the two models, with the result that the ANSI standard does not support the separate compilation model. In the newer ISO standard the "export" compromise had been chosen, worked out from Silicon Graphics.

12. Compiler

Today's new compilers support template mechanism very well. Some still have several issues. The lack of full support for both ISO compilation model is still the greatest weakness of them but no problem for common use.
For a software engineer it is clear that also full standard compliant compiler can have bugs as every other software, but this is no restriction for full standard compliancy.

12.1 GCC

GNU GCC (g++), Version 3.2, Released 14.8.2002

Standard compliance issues
- The export keyword is not supported on templates. No separate compilation is possible.
- Two stage lookup in templates is not implemented.
- Probably some more but not known exactly at the time of writing. For more actual information check: http://gcc.gnu.org/cgi-bin/cvsweb.cgi/gcc/gcc/doc/standards.texi

12.2 VC6

Microsoft Visual C++ 6.0 Enterprise Edition, Released 1998

Standard compliance issues
- Class template partial specialization is not supported.
- Partial ordering of function templates is not supported.
- The compiler does not support out-of-class definition of member template functions and classes. So they must be declared and defined within the class or template.
- The export keyword is not supported on templates. No separate compilation is possible.
- Template template-parameters are not supported.
- If not all template parameters are used in the arguments or the return type of a template function the template function is not overloaded correctly.

12.3 .NET

Microsoft Development Environment, Version 7.0, Released 2002
Microsoft .NET Framework 1.0, Released 2001
Microsoft Visual C++ .NET, Released 2002

Standard compliance issues
- Class template partial specialization is not supported.
- Partial ordering of function templates is not supported.
- The compiler does not support out-of-class definition of member template functions and classes. So they must be declared and defined within the class or template.
- The export keyword is not supported on templates. No separate compilation is possible.

12.4 EDG

EDG C++ front, Version 2.45.2, Released 14.2.2001

Fully standard compliant

12.5 Borland

Borland® C++ Builder™ 6, Released 2002

Fully standard compliant

12.6 Conclusion

Borland's C++ Builder is very nice and a good choice for today's C++ template projects. Microsoft's C++ compilers are disappointing. Even the newest version (.NET) lacks of some very important templates features and can not be used for professional application. GCC's greatest weakness is that even the project members don't know yet the standard compliance issues. So one should be very careful not use it for important projects.
It can be expected that a lot of professional C++ compilers will be fully standard compliant till end of 2003 due to their great efforts on this topic.

13. References

13.1 Official Documents

1. ANSI/ISO Standard, Programming language C++, ISO/IEC 14882, 1998, http://std.dkuug.dk/jtc1/sc22/wg21/
2. Revisions to Partial Ordering Rules (issue 214), 2002, http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2002/n1393.html

13.2 Textbooks

3. Linkers and Loaders, John R. Levine, Morgan Kaufmann Publishers 1999, ISBN 1-55860-496-0
4. Thinking in C++, 2nd edition, Volume 2,Revision 4.0, Bruce Eckel & Chuck Allison, http://www.camtp.uni-mb.si/books/Thinking-in-C++/Frames.html
5. C++ Annotations Version 5.1.1a, Frank B. Brokken, ISBN 90 367 0470 7, http://www.icce.rug.nl/documents/cplusplus
6. Design Patterns, Gamma Helm Johnson Vlissides, Addison-Wessley 1995, ISBN 0201633612
7. Generative Programming: Methods, Tools, and Applications, Krysztof Czarnecki Ulrich Eisenecker, Addison Wesley Professional 2000, ISBN: 0-201-30977-7

13.3 Papers

8. C++ Templates as Partial Evaluation, Todd L. Veldhuizen, http://www.osl.iu.edu/~tveldhui/papers/pepm99/
9. Template Argument Matching, Robert Schmidt, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndeepc/html/deep05042000.asp
10. Roland Weiss und Volker Simonis, Exploring Template Template Parameters, University Tübingen, 2001, http://www.progdoc.de/papers/ttp/psi-ttp.pdf
11. "Export" vs. "Extern" in the Template Compilation Model, Bill Gibbons, http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/1996/N1034.pdf

13.4 Archives, Discussion Groups, Guides

12. CUED Talk: C++ and the STL (Standard Template Library), http://www-h.eng.cam.ac.uk/help/tpl/talks/C++.html
13. GotW Archive for C++, http://www.gotw.ca/gotw/
14. C++ Standard Template Library, Dr. Mark J. Sebern, http://www.msoe.edu/eecs/ce/courseinfo/stl
15. C++ User's Guide, http://www.ictp.trieste.it/~manuals/programming/sun/c-plusplus/c++_ug
16. Partial ordering of function templates underspecified, Martin v. Lövis, http://www.informatik.hu-berlin.de/~loewis/cxxdefects.html
17. Standard Compliance Issues in Visual C++ in .NET, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclang/html/vclrf14exportkeywordontemplate.asp
18. Template Compilation Model, Jonathan Schilling, http://www.glenmccl.com/016.htm, http://www.glenmccl.com/017.htm, http://www.glenmccl.com/ansi_026.htm, http://www.glenmccl.com/tmpl_cmp.htm

19. Separate compilation and exporting template definitions, http://www.corfield.org/cplusplus.phtml?cpp=ptexp
20. Open UNIX,Instantiating C++ templates, ttp://docsrv.caldera.com/SDK_cprog/CTOC-InstantiatCTemplates.html
21. C++ User's Guide, Forte Developer 7, http://docs.sun.com/source/816-2460/
22. ANSI/ISO C++ Professional Programmer's Handbook, http://documentation.captis.com/files/c++/handbook/ch09/ch09.htm

13.5 Tutorials

23. C++ Standard Core Language Issue Table of Contents, Rev 22, http://www.comeaucomputing.com/iso/cwg_toc.html
24. The cplusplus.com, http://www.cplusplus.com/
25. Introduction to C++ Templates, Anthony Williams, 2001, http://web.onetel.net.uk/~anthony_w/cplusplus/intrototemplates.pdf
26. C++ Template Tutorial, Microsoft, http://babbage.cs.qc.edu/STL_Docs/templates.htm
27. Microsoft, STL Tutorial, http://www.staff.fh-vorarlberg.ac.at/hv/stl/Visual%20C++stl.pdf
28. OOOPWeb Tutorial, http://oopweb.com/CPP/Documents/CPPAnnotations/VolumeFrames.html?/CPP/Documents/CPPAnnotations/Volume/cplusplus16.html
29. Standard Template Library Programmer's Guide, Silicon Graphics, http://www.sgi.com/tech/stl/
30. An introduction to C++ templates, Kais Dukes, http://www.codeguru.com/atl/KD062002.html
31. Techniques for Scientific C++, Todd Veldhuizen, http://osl.iu.edu/~tveldhui/papers/techniques/