Updated for C++ 20

Discovering Modern C++

An Intensive Course for Scientists, Engineers, and Programmers

Peter Gottschling

SECOND EDITION

C++ In-Depth Series   Bjarne Stroustrup

Discovering Modern C++

Second Edition

```
        template <typename LinOp>
        void initialize(LinOp& A) { /* ... */ }

        matrix operator+(const matrix& A, const matrix& B)
        {
            matrix C;
            initialize(C); // not qualified, same namespace
            add(A, B, C);
            return C;
        }
    }
```

Every time we use the function `initialize` (which is supposedly being only defined in `rocketscience`), we can omit the qualification for all classes in namespace `rocketscience`:

```
int main ()
{
    rocketscience::matrix A, B, C, D;
    rocketscience::initialize(B); // qualified
    initialize(C);                // rely on ADL

    chez_herbert::matrix E, F, G;
    rocketscience::initialize(E); // qualification needed
    initialize(F);                // Error: initialize not found
}
```

Operators are also subject to ADL:

```
A= B + C + D;
```

Imagine the previous expression without ADL:

```
A= rocketscience::operator+(rocketscience::operator+(B, C), D);
```

Similarly ugly and even more cumbersome is streaming I/O when the namespace must be qualified. Since user code should not be in namespace `std`, the `operator<<` for a class is preferably defined in that class's namespace. This allows ADL to find the right overload for each type, e.g.:

```
std::cout << A << E << B << F << std::endl;
```

Without ADL we would need to qualify the namespace of each operator in its verbose notation. This would turn the previous expression into:

```
std::operator<<(chez_herbert::operator<<(
    rocketscience::operator<<(chez_herbert::operator<<(
        rocketscience::operator<<(std::cout, A), E), B),
    F), std::endl);
```

The ADL mechanism can also be used to select the right function template overload when the classes are distributed over multiple namespaces. The L_1 norm (also known as the Manhattan norm) from linear algebra is defined for both matrices and vectors, and we want to provide a template implementation for both:

```
template <typename Matrix>
double one_norm(const Matrix& A) { ... }

template <typename Vector>
double one_norm(const Vector& x) { ... }
```

How can the compiler know which overload we prefer for each type? One possible solution is to introduce a namespace for matrices and one for vectors so that the correct overload is selected by ADL:

```
namespace rocketscience {
    namespace mat {
        struct sparse_matrix {};
        struct dense_matrix {};
        struct über_matrix[6] {};

        template <typename Matrix>
        double one_norm(const Matrix& A) { ... }
    }
    namespace vec {
        struct sparse_vector {};
        struct dense_vector {};
        struct über_vector {};

        template <typename Vector>
        double one_norm(const Vector& x) { ... }
    }
}
```

The ADL mechanism searches functions only in the namespaces of the arguments' type declarations but not in their respective parent namespaces:

```
namespace rocketscience {
    ...
    namespace vec {
        struct sparse_vector {};
        struct dense_vector {};
        struct über_vector {};
    }
    template <typename Vector>
    double one_norm(const Vector& x) { ... }
}

int main ()
{
    rocketscience::vec::über_vector x;
    double norm_x= one_norm(x);      // Error: not found by ADL
}
```

6. Of course, we use the original German spelling of *uber*—sometimes even seen in American papers. Please note that some compilers (e.g., `g++-9`) have problems with non-ASCII characters.

Also, when we import a name into another namespace, the functions in that namespace are not considered by ADL either:

```
namespace rocketscience {
    ...
    using vec::über_vector;

    template <typename Vector>
    double one_norm(const Vector& x) { ... }
}

int main ()
{
    rocketscience::über_vector x;
    double norm_x= one_norm(x);      // Error: not found by ADL
}
```

Relying on ADL only for selecting the right overload has its limitations. When we use a third-party library, we may find functions and operators that we also implemented in our namespace. Such ambiguities can be reduced (but not entirely avoided) by importing only single functions with `using` instead of entire namespaces.

The probability of ambiguities rises further with multi-argument functions, especially when parameter types come from different namespaces, e.g.:

```
namespace rocketscience {
    namespace mat {
        ...
        template <typename Scalar, typename Matrix>
        Matrix operator*(const Scalar& a, const Matrix& A) { ... }
    }
    namespace vec {
        ...
        template <typename Scalar, typename Vector>
        Vector operator*(const Scalar& a, const Vector& x) { ... }

        template <typename Matrix, typename Vector>
        Vector operator*(const Matrix& A, const Vector& x) { ... }
    }
}
int main (int argc, char* argv[])
{
    rocketscience::mat::über_matrix A;
    rocketscience::vec::über_vector x, y;
    y= A * x;                    // which overload is selected?
}
```

Here the intention is clear. Well, to human readers. For the compiler it is less so. The type of `A` is defined in `rocketscience::mat` and that of `x` in `rocketscience::vec` so that `operator*` is sought in both namespaces. Thus, all three template overloads are available and none of them is a better match than the others (although probably only one of them would compile).

Unfortunately, ADL is disabled whenever template parameters are explicitly given.[7] To overcome this issue, the function must be made visible by namespace qualification or imported via `using`.

Which function overload is called depends on the following, so-far-discussed rules on:

- Namespace nesting and qualification,
- Name hiding,
- ADL, and
- Overload resolution.

This non-trivial interplay must be understood for heavily overloaded functions to ascertain that no ambiguity occurs and the right overload is selected. Therefore, we give some examples in Appendix A.6.2. Feel free to postpone this discussion until you get baffled with unexpected overload resolutions or ambiguities when dealing with a larger code base.

3.2.3 Namespace Qualification or ADL

Many programmers do not want to get into the complicated rules of how a compiler picks an overload or runs into ambiguities. They qualify the namespace of the called function and know exactly which function overload is selected (assuming the overloads in that namespace are not ambiguous within the overload resolution). We do not blame them; the name lookup is anything but trivial.

When we plan to write good generic software containing function and class templates instantiatable with many types, we should consider ADL. We will demonstrate this with a very popular performance bug (especially in C++03) that many programmers have run into. The standard library contains a function template called `swap`. It swaps the content of two objects of the same type. The old default implementation used copies and a temporary:

```
template <typename T>
inline void swap(T& x, T& y)
{
    T tmp(x); x= y; y= tmp;
}
```

It works for all types with copy constructor and assignment. So far, so good. Say we have two vectors, each containing 1GB of data. Then we have to copy 3GB and also need a spare gigabyte of memory when we use the default implementation. Or we do something smarter: we switch the pointers referring to the data and the size information:

```
class vector
{
    ...
    friend inline void swap(vector& x, vector& y)
    { std::swap(x.my_size, y.my_size); std::swap(x.data, y.data); }
```

7. The problem is that ADL is performed too late in the compilation process and the opening angle bracket is already misinterpreted as less-than.

```
    private:
      unsigned my_size;
      double    *data;
  };
```

Assume we have to swap data of a parametric type in some generic function:

```
  template <typename T, typename U>
  inline void some_function(T& x, T& y, const U& z, int i)
  {
      ...
      std::swap(x, y); // can be expensive
      ...
  }
```

We played it safe and used the standard `swap` function which works with all copyable types. But we copied 3GB of data. It would be much faster and memory-efficient to use our implementation that only switches the pointers. This can be achieved with a small change in a generic manner:

```
  template <typename T, typename U>
  inline void some_function(T& x, T& y, const U& z, int i)
  {
      using std::swap;
      ...
      swap(x, y); // involves ADL
      ...
  }
```

With this implementation, both `swap` overloads are candidates but the one in our class is prioritized by overload resolution as its argument type is more specific than that of the standard implementation. More generally, any implementation for a user type is more specific than `std::swap`. In fact, `std::swap` is already overloaded for standard containers for the same reason. This is a general pattern:

Use `using`

Do not qualify namespaces of function templates for which user-type overloads might exist. Make the name visible instead and call the function unqualified.

C++11

As an addendum to the default `swap` implementation: Since C++11, the default is to move the values between the two arguments and the temporary if possible:

```
  template <typename T>
  inline void swap(T& x, T& y)
  {
      T tmp(move(x));
      x= move(y);
      y= move(tmp);
  }
```

As a result, types without user-defined `swap` can be swapped efficiently when they provide a fast move constructor and assignment. Only types without user implementation and move support are finally copied.

3.3 Class Templates

Before the namespaces, we described the use of templates to create generic functions. Templates can also be used to create generic classes. Analogous to generic function, class template is the correct term from the standard whereas template class (or templated class) is more frequently used in daily life. In these classes, the types of data members can be parameterized.

This is in particular useful for general-purpose container classes like vectors, matrices, and lists. We could also extend the complex class with a parametric value type. However, we have already spent so much time with this class that it seems more entertaining to look at something else.

3.3.1 A Container Example

⇒ c++11/vector_template.cpp

Let us, for example, write a generic vector class, in the sense of linear algebra not like an STL vector. First, we implement a class with the most fundamental operators only:

Listing 3–1: Template vector class

```
template <typename T>
class vector
{
  public:
    explicit vector(int size)
      : my_size{size}, data{new T[my_size]}
    {}

    vector(const vector& that)
      : my_size{that.my_size}, data{new T[my_size]}
    {
        std::copy(&that.data[0], &that.data[that.my_size], &data[0]);
    }

    int size() const { return my_size; }

    const T& operator[](int i) const
    {
        check_index(i);
        return data[i];
    }
    // ...

  private:
    int                   my_size;
    std::unique_ptr<T[]>  data;
};
```