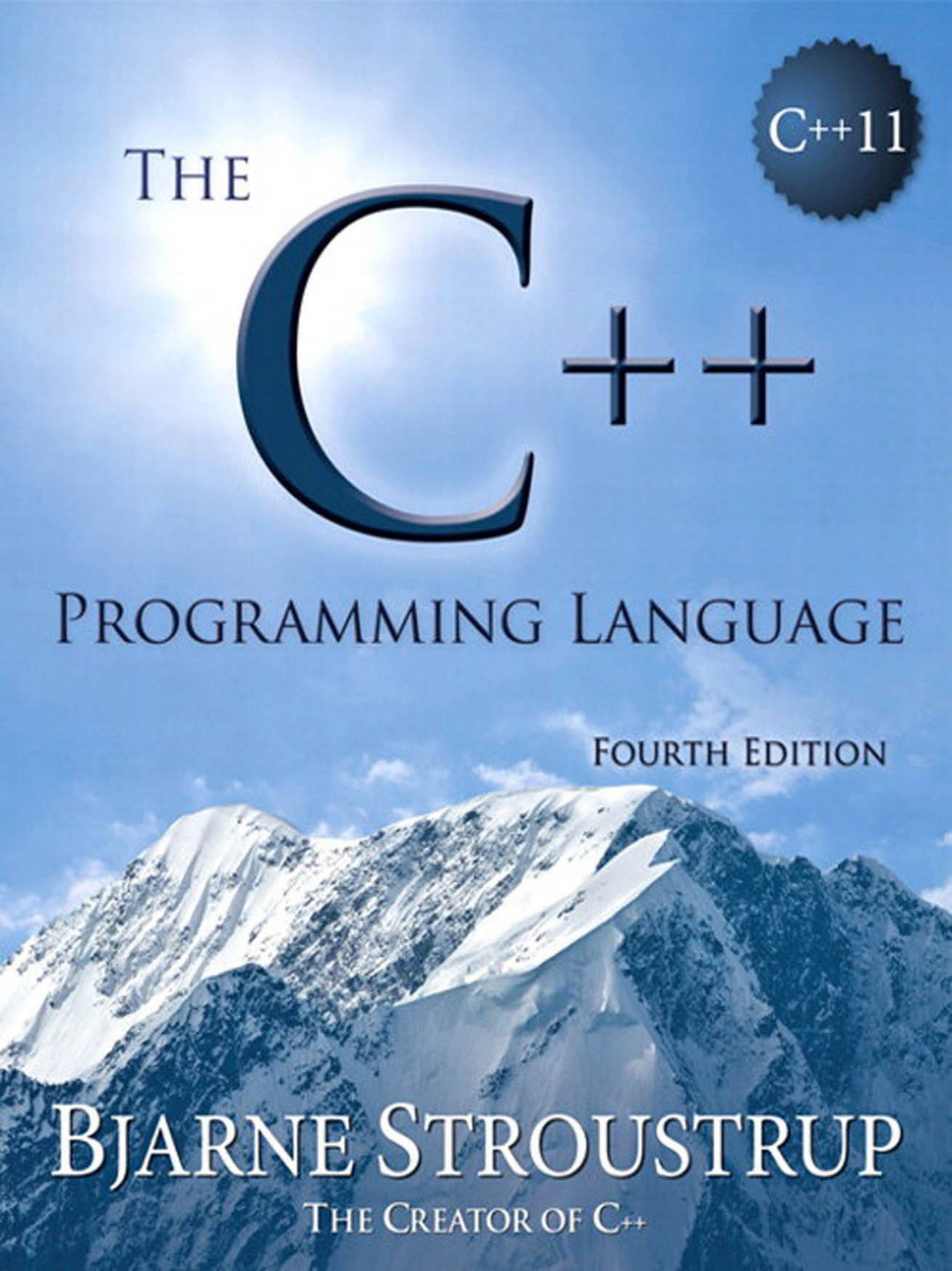

C++11
THE
C++
PROGRAMMING LANGUAGE
FOURTH EDITION
BJARNE STROUSTRUP
THE CREATOR OF C++

The C++ Programming Language

Fourth Edition

Bjarne Stroustrup

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Within a function, a **using**-directive can be safely used as a notational convenience, but care should be taken with global **using**-directives because overuse can lead to exactly the name clashes that namespaces were introduced to avoid. For example:

```
namespace Graph_lib {
        class Shape { /* ... */ };
        class Line : public Shape { /* ... */ };
        class Poly_line : public Shape { /* ... */ };// connected sequence of lines
        class Text : public Shape { /* ... */ };        // text label

        Shape operator+(const Shape&, const Shape&);    // compose

        Graph_reader open(const char*);   // open file of Shapes
}

namespace Text_lib {
        class Glyph { /* ... */ };
        class Word { /* ... */ };      // sequence of Glyphs
        class Line { /* ... */ };              // sequence of Words
        class Text { /* ... */ };              // sequence of Lines

        File* open(const char*);  // open text file

        Word operator+(const Line&, const Line&);    // concatenate
}

using namespace Graph_lib;
using namespace Text_lib;

Glyph gl;                   // Text_lib::Glyph
vector<Shape*> vs;          // Graph_lib::Shape
```

So far, so good. In particular, we can use names that do not clash, such as **Glyph** and **Shape**. However, name clashes now occur as soon as we use one of the names that clash – exactly as if we had not used namespaces. For example:

```
Text txt;                                   // error: ambiguous
File* fp = open("my_precious_data");        // error: ambiguous
```

Consequently, we must be careful with **using**-directives in the global scope. In particular, don't place a **using**-directive in the global scope in a header file except in very specialized circumstances (e.g., to aid transition) because you never know where a header might be **#include**d.

14.2.4 Argument-Dependent Lookup

A function taking an argument of user-defined type **X** is more often than not defined in the same namespace as **X**. Consequently, if a function isn't found in the context of its use, we look in the namespaces of its arguments. For example:

```
namespace Chrono {
    class Date { /* ... */ };

    bool operator==(const Date&, const std::string&);

    std::string format(const Date&);    // make string representation
    // ...
}

void f(Chrono::Date d, int i)
{
    std::string s = format(d);      // Chrono::format()
    std::string t = format(i);      // error: no format() in scope
}
```

This lookup rule (called *argument-dependent lookup* or simply ADL) saves the programmer a lot of typing compared to using explicit qualification, yet it doesn't pollute the namespace the way a **using**-directive (§14.2.3) can. It is especially useful for operator operands (§18.2.5) and template arguments (§26.3.5), where explicit qualification can be quite cumbersome.

Note that the namespace itself needs to be in scope and the function must be declared before it can be found and used.

Naturally, a function can take arguments from more than one namespace. For example:

```
void f(Chrono::Date d, std::string s)
{
    if (d == s) {
        // ...
    }
    else if (d == "August 4, 1914") {
        // ...
    }
}
```

In such cases, we look for the function in the scope of the call (as ever) and in the namespaces of every argument (including each argument's class and base classes) and do the usual overload resolution (§12.3) of all functions we find. In particular, for the call **d==s**, we look for **operator==** in the scope surrounding **f()**, in the **std** namespace (where **==** is defined for **string**), and in the **Chrono** namespace. There is a **std::operator==()**, but it doesn't take a **Date** argument, so we use **Chrono::operator==()**, which does. See also §18.2.5.

When a class member invokes a named function, other members of the same class and its base classes are preferred over functions potentially found based on the argument types (operators follow a different rule; §18.2.1, §18.2.5). For example:

```
namespace N {
    struct S { int i; };
    void f(S);
    void g(S);
    void h(int);
}
```

```
struct Base {
    void f(N::S);
};

struct D : Base {
    void mf(N::S);

    void g(N::S x)
    {
        f(x);       // call Base::f()
        mf(x);      // call D::mf()
        h(1);       // error: no h(int) available
    }
};
```

In the standard, the rules for argument-dependent lookup are phrased in terms of *associated namespaces* (§iso.3.4.2). Basically:

- If an argument is a class member, the associated namespaces are the class itself (including its base classes) and the class's enclosing namespaces.
- If an argument is a member of a namespace, the associated namespaces are the enclosing namespaces.
- If an argument is a built-in type, there are no associated namespaces.

Argument-dependent lookup can save a lot of tedious and distracting typing, but occasionally it can give surprising results. For example, the search for a declaration of a function **f()** does not have a preference for functions in a **namespace** in which **f()** is called (the way it does for functions in a **class** in which **f()** is called):

```
namespace N {
    template<typename T>
        void f(T, int);    // N::f()
    class X { };
}

namespace N2 {
    N::X x;

    void f(N::X, unsigned);

    void g()
    {
        f(x,1);      // calls N::f(X,int)
    }
}
```

It may seem obvious to choose **N2::f()**, but that is not done. Overload resolution is applied and the best match is found: **N::f()** is the best match for **f(x,1)** because **1** is an **int** rather than an **unsigned**. Conversely, examples have been seen where a function in the caller's namespace is chosen but the programmer expected a better function from a known namespace to be used (e.g., a standard-library function from **std**). This can be most confusing. See also §26.3.6.

14.2.5 Namespaces Are Open

A namespace is open; that is, you can add names to it from several separate namespace declarations. For example:

```
namespace A {
    int f();    // now A has member f()
}

namespace A {
    int g();    // now A has two members, f() and g()
}
```

That way, the members of a namespace need not be placed contiguously in a single file. This can be important when converting older programs to use namespaces. For example, consider a header file written without the use of namespaces:

```
// my header:

    void mf();      // my function
    void yf();      // your function
    int mg();       // my function
    // ...
```

Here, we have (unwisely) just added the declarations needed without concerns of modularity. This can be rewritten without reordering the declarations:

```
// my header:

    namespace Mine {
        void mf();      // my function
        // ...
    }

    void yf();          // your function (not yet put into a namespace)

    namespace Mine {
        int mg();       // my function
        // ...
    }
```

When writing new code, I prefer to use many smaller namespaces (see §14.4) rather than putting really major pieces of code into a single namespace. However, that is often impractical when converting major pieces of software to use namespaces.

Another reason to define the members of a namespace in several separate namespace declarations is that sometimes we want to distinguish parts of a namespace used as an interface from parts used to support easy implementation; §14.3 provides an example.

A namespace alias (§14.4.2) cannot be used to re-open a namespace.

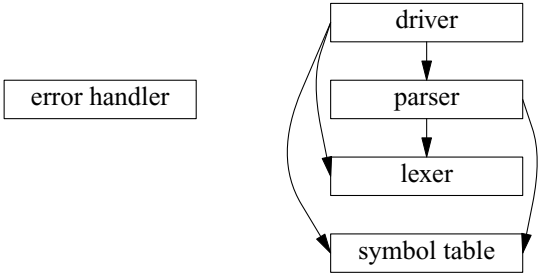
14.3 Modularization and Interfaces

Any realistic program consists of a number of separate parts. For example, even the simple "Hello, world!" program involves at least two parts: the user code requests **Hello, world!** to be printed, and the I/O system does the printing.

Consider the desk calculator example from §10.2. It can be viewed as composed of five parts:

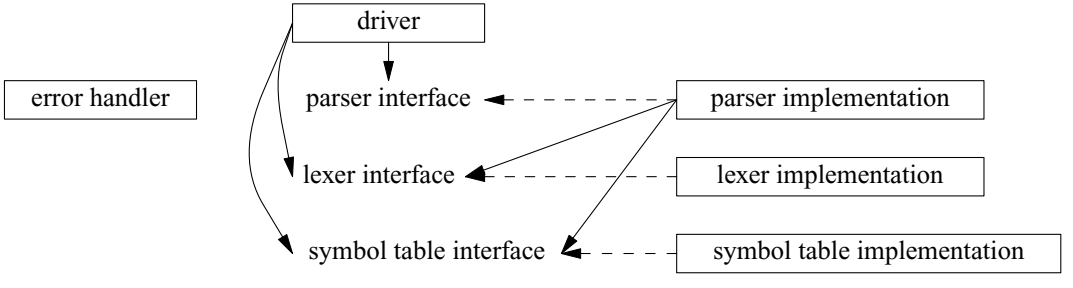
[1] The parser, doing syntax analysis: **expr()**, **term()**, and **prim()**
[2] The lexer, composing tokens out of characters: **Kind**, **Token**, **Token_stream**, and **ts**
[3] The symbol table, holding (string,value) pairs: **table**
[4] The driver: **main()** and **calculate()**
[5] The error handler: **error()** and **number_of_errors**

This can be represented graphically:



where an arrow means "using." To simplify the picture, I have not represented the fact that every part relies on error handling. In fact, the calculator was conceived as three parts, with the driver and error handler added for completeness.

When one module uses another, it doesn't need to know everything about the module used. Ideally, most of the details of a module are unknown to its users. Consequently, we make a distinction between a module and its interface. For example, the parser directly relies on the lexer's interface (only), rather than on the complete lexer. The lexer simply implements the services advertised in its interface. This can be presented graphically like this:



A dashed line means "implements." I consider this to be the real structure of the program, and our job as programmers is to represent this faithfully in code. That done, the code will be simple, efficient, comprehensible, maintainable, etc., because it will directly reflect our fundamental design.

The following subsections show how the logical structure of the desk calculator program can be made clear, and §15.3 shows how the program source text can be physically organized to take advantage of it. The calculator is a tiny program, so in "real life" I wouldn't bother using namespaces and separate compilation (§2.4.1, §15.1) to the extent done here. Making the structure of the calculator explicit is simply an illustration of techniques useful for larger programs without drowning in code. In real programs, each "module" represented by a separate namespace will often have hundreds of functions, classes, templates, etc.

Error handling permeates the structure of a program. When breaking up a program into modules or (conversely) when composing a program out of modules, we must take care to minimize dependencies between modules caused by error handling. C++ provides exceptions to decouple the detection and reporting of errors from the handling of errors (§2.4.3.1, Chapter 13).

There are many more notions of modularity than the ones discussed in this chapter and the next. For example, we might use concurrently executing and communicating tasks (§5.3, Chapter 41) or processes to represent important aspects of modularity. Similarly, the use of separate address spaces and the communication of information between address spaces are important topics not discussed here. I consider these notions of modularity largely independent and orthogonal. Interestingly, in each case, separating a system into modules is easy. The hard problem is to provide safe, convenient, and efficient communication across module boundaries.

14.3.1 Namespaces as Modules

A namespace is a mechanism for expressing logical grouping. That is, if some declarations logically belong together according to some criteria, they can be put in a common namespace to express that fact. So we can use namespaces to express the logical structure of our calculator. For example, the declarations of the parser from the desk calculator (§10.2.1) may be placed in a namespace **Parser**:

```
namespace Parser {
        double expr(bool);
        double prim(bool get) { /* ... */ }
        double term(bool get) { /* ... */ }
        double expr(bool get) { /* ... */ }
}
```

The function **expr()** must be declared first and then later defined to break the dependency loop described in §10.2.1.

The input part of the desk calculator could also be placed in its own namespace:

```
namespace Lexer {
        enum class Kind : char { /* ... */ };
        class Token { /* ... */ };
        class Token_stream { /* ... */ };

        Token_stream ts;
}
```

The symbol table is extremely simple: