8

Namespaces and Exceptions

The year is 787!
A.D.?
– Monty Python

No rule is so general,
which admits not some exception.
– Robert Burton

Modularity, interfaces, and exceptions — namespaces — ***using*** — ***using namespace*** — avoiding name clashes — name lookup — namespace composition — namespace aliases — namespaces and C code — exceptions — ***throw*** and ***catch*** — exceptions and program structure — advice — exercises.

8.1 Modularization and Interfaces [name.module]

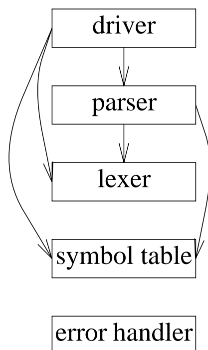
Any realistic program consists of a number of separate parts. For example, even the simple ''Hello, world!'' program involves at least two parts: the user code requests ***Hello, world!*** to be printed, and the I/O system does the printing.

Consider the desk calculator example from §6.1. It can be viewed as being composed of five parts:

[1] The parser, doing syntax analysis
[2] The lexer, composing tokens out of characters
[3] The symbol table, holding (string,value) pairs
[4] The driver, ***main()***
[5] The error handler

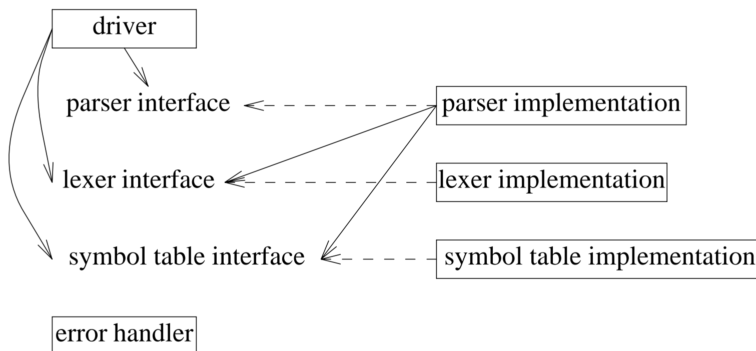
This can be represented graphically:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.



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:



Dashed lines 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 sections show how the logical structure of the desk calculator program can be made clear, and §9.3 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, §9.1) to the extent I do here. It is simply used to present techniques useful for larger programs without our drowning in code. In real programs, each ‘‘module’’ represented by a separate namespace will often have hundreds of functions, classes, templates, etc.

To demonstrate a variety of techniques and language features, I develop the modularization of

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

the calculator in stages. In ‘‘real life,’’ a program is unlikely to grow through all of these stages. An experienced programmer might pick a design that is ‘‘about right’’ from the start. However, as a program evolves over the years, dramatic structural changes are not uncommon.

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. Therefore, the discussion of how to represent modules as namespaces (§8.2) is followed by a demonstration of how we can use exceptions to further improve modularity (§8.3).

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 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.

8.2 Namespaces [name.namespace]

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. For example, the declarations of the parser from the desk calculator (§6.1.1) 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 §6.1.1.

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

```
namespace Lexer {
      enum Token_value {
            NAME,          NUMBER,          END,
            PLUS='+',      MINUS='-',       MUL='*',      DIV='/',
            PRINT=';',     ASSIGN='=',      LP='(',       RP=')'
      };

      Token_value curr_tok;
      double number_value;
      string string_value;

      Token_value get_token() { /* ... */ }
}
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

This use of namespaces makes it reasonably obvious what the lexer and the parser provide to a user. However, had I included the source code for the functions, this structure would have been obscured. If function bodies are included in the declaration of a realistically-sized namespace, you typically have to wade through pages or screenfuls of information to find what services are offered, that is, to find the interface.

An alternative to relying on separately specified interfaces is to provide a tool that extracts an interface from a module that includes implementation details. I don't consider that a good solution. Specifying interfaces is a fundamental design activity (see §23.4.3.4), a module can provide different interfaces to different users, and often an interface is designed long before the implementation details are made concrete.

Here is a version of the ***Parser*** with the interface separated from the implementation:

```
namespace Parser {
    double prim(bool);
    double term(bool);
    double expr(bool);
}

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

Note that as a result of separating the implementation of the interface, each function now has exactly one declaration and one definition. Users will see only the interface containing declarations. The implementation – in this case, the function bodies – will be placed ''somewhere else'' where a user need not look.

As shown, a member can be declared within a namespace definition and defined later using the *namespace-name*::*member-name* notation.

Members of a namespace must be introduced using this notation:

```
namespace namespace-name {
    // declaration and definitions
}
```

We cannot declare a new member of a namespace outside a namespace definition using the qualifier syntax. For example:

```
void Parser::logical(bool);     // error: no logical() in Parser
```

The idea is to make it reasonably easy to find all names in a namespace declaration and also to catch errors such as misspellings and type mismatches. For example:

```
double Parser::trem(bool);      // error: no trem() in Parser
double Parser::prim(int);       // error: Parser::prim() takes a bool argument
```

A namespace is a scope. Thus, ''namespace'' is a very fundamental and relatively simple concept. The larger a program is, the more useful namespaces are to express logical separations of its parts. Ordinary local scopes, global scopes, and classes are namespaces (§C.10.3).

Ideally, every entity in a program belongs to some recognizable logical unit (''module''). Therefore, every declaration in a nontrivial program should ideally be in some namespace named to

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

indicate its logical role in the program. The exception is ***main*()**, which must be global in order for the run-time environment to recognize it as special (§8.3.3).

8.2.1 Qualified Names [name.qualified]

A namespace is a scope. The usual scope rules hold for namespaces, so if a name is previously declared in the namespace or in an enclosing scope, it can be used without further fuss. A name from another namespace can be used when qualified by the name of its namespace. For example:

```
double Parser::term(bool get)           // note Parser:: qualification
{
    double left = prim(get);            // no qualification needed

    for (;;)
        switch (Lexer::curr_tok) {      // note Lexer:: qualification
        case Lexer::MUL:                // note Lexer:: qualification
            left *= prim(true);         // no qualification needed
        // ...
        }
    // ...
}
```

The ***Parser*** qualifier is necessary to state that this ***term*()** is the one declared in ***Parser*** and not some unrelated global function. Because ***term*()** is a member of ***Parser***, it need not use a qualifier for ***prim*()**. However, had the ***Lexer*** qualifier not been present, ***curr_tok*** would have been considered undeclared because the members of namespace ***Lexer*** are not in scope from within the ***Parser*** namespace.

8.2.2 Using Declarations [name.using.dcl]

When a name is frequently used outside its namespace, it can be a bother to repeatedly qualify it with its namespace name. Consider:

```
double Parser::prim(bool get)       // handle primaries
{
    if (get) Lexer::get_token();

    switch (Lexer::curr_tok) {
    case Lexer::NUMBER:             // floating-point constant
        Lexer::get_token();
        return Lexer::number_value;

    case Lexer::NAME:
    {   double& v = table[Lexer::string_value];
        if (Lexer::get_token() == Lexer::ASSIGN) v = expr(true);
        return v;
    }

    case Lexer::MINUS:              // unary minus
        return -prim(true);
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
        case Lexer::LP:
        {       double e = expr(true);
                if (Lexer::curr_tok != Lexer::RP) return Error::error(") expected");
                Lexer::get_token();     // eat ')'
                return e;
        }
        case Lexer::END:
                return 1;
        default:
                return Error::error("primary expected");
        }
}
```

The repeated qualification *Lexer* is tedious and distracting. This redundancy can be eliminated by a *using-declaration* to state in one place that the *get_token* used in this scope is *Lexer*'s *get_token*. For example:

```
double Parser::prim(bool get)          // handle primaries
{
        using Lexer::get_token;    // use Lexer's get_token
        using Lexer::curr_tok;     // use Lexer's curr_tok
        using Error::error;        // use Error's error

        if (get) get_token();

        switch (curr_tok) {
        case Lexer::NUMBER:             // floating-point constant
                get_token();
                return Lexer::number_value;
        case Lexer::NAME:
        {       double& v = table[Lexer::string_value];
                if (get_token() == Lexer::ASSIGN) v = expr(true);
                return v;
        }
        case Lexer::MINUS:              // unary minus
                return -prim(true);
        case Lexer::LP:
        {       double e = expr(true);
                if (curr_tok != Lexer::RP) return error(") expected");
                get_token();            // eat ')'
                return e;
        }
        case Lexer::END:
                return 1;
        default:
                return error("primary expected");
        }
}
```

A *using-declaration* introduces a local synonym.

It is often a good idea to keep local synonyms as local as possible to avoid confusion.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

However, all parser functions use similar sets of names from other modules. We can therefore place the *using-declarations* in the ***Parser***'s namespace definition:

```
namespace Parser {
      double prim(bool);
      double term(bool);
      double expr(bool);

      using Lexer::get_token;     // use Lexer's get_token
      using Lexer::curr_tok;      // use Lexer's curr_tok
      using Error::error;         // use Error's error
}
```

This allows us to simplify the ***Parser*** functions almost to our original version (§6.1.1):

```
double Parser::term(bool get)         // multiply and divide
{
      double left = prim(get);

      for (;;)
            switch (curr_tok) {
            case Lexer::MUL:
                  left *= prim(true);
                  break;

            case Lexer::DIV:
                  if (double d = prim(true)) {
                        left /= d;
                        break;
                  }
                  return error("divide by 0");

            default:
                  return left;
            }
}
```

I could have introduced the token names into the ***Parser***'s namespace. However, I left them explicitly qualified as a reminder of ***Parser***'s dependency on ***Lexer***.

8.2.3 Using Directives [name.using.dir]

What if our aim were to simplify the ***Parser*** functions to be *exactly* our original versions? This would be a reasonable aim for a large program that was being converted to using namespaces from a previous version with less explicit modularity.

A *using-directive* makes names from a namespace available almost as if they had been declared outside their namespace (§8.2.8). For example:

```
namespace Parser {
      double prim(bool);
      double term(bool);
      double expr(bool);
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
        using namespace Lexer;    // make all names from Lexer available
        using namespace Error;    // make all names from Error available
}
```

This allows us to write *Parser*'s functions exactly as we originally did (§6.1.1):

```
double Parser::term(bool get)          // multiply and divide
{
        double left = prim(get);

        for (;;)
                switch (curr_tok) {                  // Lexer's curr_tok
                case MUL:                            // Lexer's MUL
                        left *= prim(true);
                        break;
                case DIV:                            // Lexer's DIV
                        if (double d = prim(true)) {
                                left /= d;
                                break;
                        }
                        return error("divide by 0"); // Error's error
                default:
                        return left;
                }
}
```

Global *using-directive*s are a tool for transition (§8.2.9) and are otherwise best avoided. In a namespace, a ***using-directive*** is a tool for namespace composition (§8.2.8). In a function (only), a ***using-directive*** can be safely used as a notational convenience (§8.3.3.1).

8.2.4 Multiple Interfaces [name.multi]

It should be clear that the namespace definition we evolved for ***Parser*** is not the interface that the ***Parser*** presents to its users. Instead, it is the set of declarations that is needed to write the individual parser functions conveniently. The ***Parser***'s interface to its users should be far simpler:

```
namespace Parser {
        double expr(bool);
}
```

Fortunately, the two *namespace-definition*s for ***Parser*** can coexist so that each can be used where it is most appropriate. We see the namespace ***Parser*** used to provide two things:

[1] The common environment for the functions implementing the parser

[2] The external interface offered by the parser to its users

Thus, the driver code, ***main*()**, should see only:

```
namespace Parser {              // interface for users
        double expr(bool);
}
```

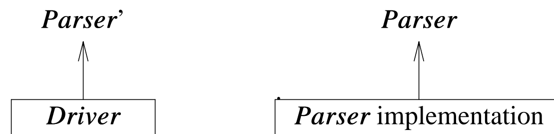
The functions implementing the parser should see whichever interface we decided on as the best for expressing those functions' shared environment. That is:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
namespace Parser {                  // interface for implementers
      double prim(bool);
      double term(bool);
      double expr(bool);

      using Lexer::get_token;       // use Lexer's get_token
      using Lexer::curr_tok;        // use Lexer's curr_tok
      using Error::error;           // use Error's error
}
```

or graphically:



The arrows represent ‘‘relies on the interface provided by’’ relations.

Parser´ is the small interface offered to users. The name ***Parser´*** (Parser prime) is not a C++ identifier. It was chosen deliberately to indicate that this interface doesn't have a separate name in the program. The lack of a separate name need not lead to confusion because programmers naturally invent different and obvious names for the different interfaces and because the physical layout of the program (see §9.3.2) naturally provides separate (file) names.

The interface offered to implementers is larger than the interface offered to users. Had this interface been for a realistically-sized module in a real system, it would change more often than the interface seen by users. It is important that the users of a module (in this case, ***main*()** using ***Parser***) are insulated from such changes.

We don't need to use two separate namespaces to express the two different interfaces, but if we wanted to, we could. Designing interfaces is one of the most fundamental design activities and one in which major benefits can be gained and lost. Consequently, it is worthwhile to consider what we are really trying to achieve and to discuss a number of alternatives.

Please keep in mind that the solution presented is the simplest of those we consider, and often the best. Its main weaknesses are that the two interfaces don't have separate names and that the compiler doesn't necessarily have sufficient information to check the consistency of the two definitions of the namespace. However, even though the compiler doesn't always get the opportunity to check the consistency, it usually does. Furthermore, the linker catches most errors missed by the compiler.

The solution presented here is the one I use for the discussion of physical modularity (§9.3) and the one I recommend in the absence of further logical constraints (see also §8.2.7).

8.2.4.1 Interface Design Alternatives [name.alternatives]

The purpose of interfaces is to minimize dependencies between different parts of a program. Minimal interfaces lead to systems that are easier to understand, have better data hiding properties, are easier to modify, and compile faster.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

When dependencies are considered, it is important to remember that compilers and programmers tend to take a somewhat simple-minded approach to them: ''If a definition is in scope at point X, then anything written at point X depends on anything stated in that definition.'' Typically, things are not really that bad because most definitions are irrelevant to most code. Given the definitions we have used, consider:

```
namespace Parser {          // interface for implementers
    // ...
    double expr(bool);
    // ...
}

int main()
{
    // ...
    Parser::expr(false);
    // ...
}
```

The function *main*() depends on *Parser*::*expr*() only, but it takes time, brain power, computation, etc., to figure that out. Consequently, for realistically-sized programs people and compilation systems often play it safe and assume that where there might be a dependency, there is one. This is typically a perfectly reasonable approach.

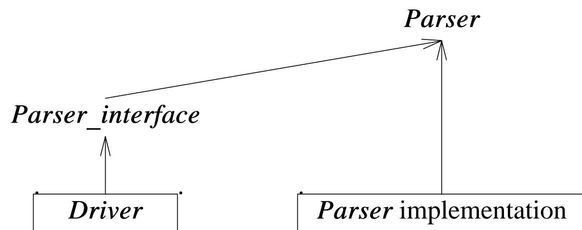
Thus, our aim is to express our program so that the set of potential dependencies is reduced to the set of actual dependencies.

First, we try the obvious: define a user interface to the parser in terms of the implementer interface we already have:

```
namespace Parser {                  // interface for implementers
    // ...
    double expr(bool);
    // ...
}

namespace Parser_interface {        // interface for users
    using Parser::expr;
}
```

Clearly, users of *Parser_interface* depend only, and indirectly, on *Parser*::*expr*(). However, a crude look at the dependency graph gives us this:



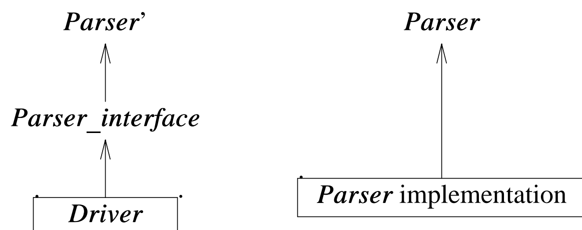
The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

Now the *driver* appears vulnerable to any change in the *Parser* interface from which it was supposed to be insulated. Even this appearance of a dependency is undesirable, so we explicitly restrict *Parser_interface*'s dependency on *Parser* by having only the relevant part of the implementer interface to parser (that was called *Parser´* earlier) in scope where we define *Parser_interface*:

```
namespace Parser {            // interface for users
    double expr(bool);
}

namespace Parser_interface {  // separately named interface for users
    using Parser::expr;
}
```

or graphically:



To ensure the consistency of *Parser* and *Parser´*, we again rely on the compilation system as a whole, rather than on just the compiler working on a single compilation unit. This solution differs from the one in §8.2.4 only by the extra namespace *Parser_interface*. If we wanted to, we could give *Parser_interface* a concrete representation by giving it its own *expr()* function:

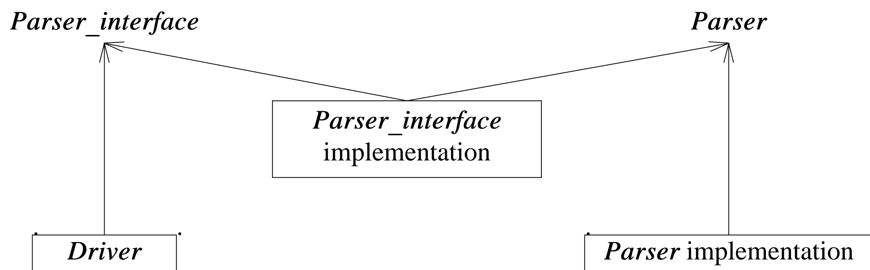
```
namespace Parser_interface {
    double expr(bool);
}
```

Now *Parser* need not be in scope in order to define *Parser_interface*. It needs to be in scope only where *Parser_interface::expr()* is defined:

```
double Parser_interface::expr(bool get)
{
    return Parser::expr(get);
}
```

This last variant can be represented graphically like this:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.



Now all dependencies are minimized. Everything is concrete and properly named. However, for most problems I face, this solution is also massive overkill.

8.2.5 Avoiding Name Clashes [name.clash]

Namespaces are intended to express logical structure. The simplest such structure is the distinction between code written by one person vs. code written by someone else. This simple distinction can be of great practical importance.

When we use only a single global scope, it is unnecessarily difficult to compose a program out of separate parts. The problem is that the supposedly-separate parts each define the same names. When combined into the same program, these names clash. Consider:

```
// my.h:
    char f(char);
    int f(int);
    class String { /* ... */ };

// your.h:
    char f(char);
    double f(double);
    class String { /* ... */ };
```

Given these definitions, a third party cannot easily use both *my.h* and *your.h*. The obvious solution is to wrap each set of declarations in its own namespace:

```
namespace My {
    char f(char);
    int f(int);
    class String { /* ... */ };
}

namespace Your {
    char f(char);
    double f(double);
    class String { /* ... */ };
}
```

Now we can use declarations from *My* and *Your* through explicit qualification (§8.2.1), *using-declarations* (§8.2.2), or *using-directives* (§8.2.3).

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

8.2.5.1 Unnamed Namespaces [name.unnamed]

It is often useful to wrap a set of declarations in a namespace simply to protect against the possibility of name clashes. That is, the aim is to preserve locality of code rather than to present an interface to users. For example:

```
#include "header.h"
namespace Mine {
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
```

Since we don't want the name *Mine* to be known outside a local context, it simply becomes a bother to invent a redundant global name that might accidentally clash with someone else's names. In that case, we can simply leave the namespace without a name:

```
#include "header.h"
namespace {
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
```

Clearly, there has to be some way of accessing members of an unnamed namespace from the outside. Consequently, an unnamed namespace has an implied *using-directive*. The previous declaration is equivalent to

```
namespace $$$ {
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
using namespace $$$;
```

where `$$$` is some name unique to the scope in which the namespace is defined. In particular, unnamed namespaces in different translation units are different. As desired, there is no way of naming a member of an unnamed namespace from another translation unit.

8.2.6 Name Lookup [name.koenig]

A function taking an argument of type ***T*** is more often than not defined in the same namespace as ***T***. 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&);
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
    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 saves the programmer a lot of typing compared to using explicit qualification, yet it doesn't pollute the namespace the way a *using-directive* (§8.2.3) can. It is especially useful for operator operands (§11.2.4) and template arguments (§C.13.8.4), 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 (§7.4) 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 §11.2.4.

When a class member invokes a function, other members of the same class and its base classes are preferred over functions potentially found based on the argument types (§11.2.4).

8.2.7 Namespace Aliases [name.alias]

If users give their namespaces short names, the names of different namespaces will clash:

```
namespace A { // short name, will clash (eventually)
    // ...
}

A::String s1 = "Grieg";
A::String s2 = "Nielsen";
```

However, long namespace names can be impractical in real code:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
namespace American_Telephone_and_Telegraph {    // too long
    // ...
}

American_Telephone_and_Telegraph::String s3 = "Grieg";
American_Telephone_and_Telegraph::String s4 = "Nielsen";
```

This dilemma can be resolved by providing a short alias for a longer namespace name:

```
// use namespace alias to shorten names:

namespace ATT = American_Telephone_and_Telegraph;

ATT::String s3 = "Grieg";
ATT::String s4 = "Nielsen";
```

Namespace aliases also allow a user to refer to ''the library'' and have a single declaration defining what library that really is. For example:

```
namespace Lib = Foundation_library_v2r11;

// ...

Lib::set s;
Lib::String s5 = "Sibelius";
```

This can immensely simplify the task of replacing one version of a library with another. By using *Lib* rather than *Foundation_library_v2r11* directly, you can update to version ''v3r02'' by changing the initialization of the alias *Lib* and recompiling. The recompile will catch source level incompatibilities. On the other hand, overuse of aliases (of any kind) can lead to confusion.

8.2.8 Namespace Composition [name.compose]

Often, we want to compose an interface out of existing interfaces. For example:

```
namespace His_string {
    class String { /* ... */ };
    String operator+(const String&, const String&);
    String operator+(const String&, const char*);
    void fill(char);
    // ...
}

namespace Her_vector {
    template<class T> class Vector { /* ... */ };
    // ...
}

namespace My_lib {
    using namespace His_string;
    using namespace Her_vector;
    void my_fct(String&);
}
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

Given this, we can now write the program in terms of ***My_lib***:

```
void f()
{
	My_lib::String s = "Byron";	// finds My_lib::His_string::String
	// ...
}

using namespace My_lib;

void g(Vector<String>& vs)
{
	// ...
	my_fct(vs[5]);
	// ...
}
```

If an explicitly qualified name (such as ***My_lib::String***) isn't declared in the namespace mentioned, the compiler looks in namespaces mentioned in *using-directive*s (such as ***His_string***).

Only if we need to define something, do we need to know the real namespace of an entity:

```
void My_lib::fill()		// error: no fill() declared in My_lib
{
	// ...
}

void His_string::fill()	// ok: fill() declared in His_string
{
	// ...
}

void My_lib::my_fct(My_lib::Vector<My_lib::String>& v) // ok
{
	// ...
}
```

Ideally, a namespace should

[1] express a logically coherent set of features,
[2] not give users access to unrelated features, and
[3] not impose a significant notational burden on users.

The composition techniques presented here and in the following subsections – together with the #***include*** mechanism (§9.2.1) – provide strong support for this.

8.2.8.1 Selection [name.select]

Occasionally, we want access to only a few names from a namespace. We could do that by writing a namespace declaration containing only those names we want. For example, we could declare a version of ***His_string*** that provided the ***String*** itself and the concatenation operator only:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
namespace His_string {                    // part of His_string only
      class String { /* ... */ };
      String operator+(const String&, const String&);
      String operator+(const String&, const char*);
}
```

However, unless I am the designer or maintainer of ***His_string***, this can easily get messy. A change to the ‘‘real’’ definition of ***His_string*** will not be reflected in this declaration. Selection of features from a namespace is more explicitly made with *using-declaration*s:

```
namespace My_string {
      using His_string::String;
      using His_string::operator+;      // use any + from His_string
}
```

A *using-declaration* brings every declaration with a given name into scope. In particular, a single *using-declaration* can bring in every variant of an overloaded function.

In this way, if the maintainer of ***His_string*** adds a member function to ***String*** or an overloaded version of the concatenation operator, that change will automatically become available to users of ***My_string***. Conversely, if a feature is removed from ***His_string*** or has its interface changed, affected uses of ***My_string*** will be detected by the compiler (see also §15.2.2).

8.2.8.2 Composition and Selection [name.comp]

Combining composition (by *using-directive*s) with selection (by *using-declaration*s) yields the flexibility needed for most real-world examples. With these mechanisms, we can provide access to a variety of facilities in such a way that we resolve name clashes and ambiguities arising from their composition. For example:

```
namespace His_lib {
      class String { /* ... */ };
      template<class T> class Vector { /* ... */ };
      // ...
}

namespace Her_lib {
      template<class T> class Vector { /* ... */ };
      class String { /* ... */ };
      // ...
}

namespace My_lib {
      using namespace His_lib; // everything from His_lib
      using namespace Her_lib; // everything from Her_lib

      using His_lib::String;     // resolve potential clash in favor of His_lib
      using Her_lib::Vector;     // resolve potential clash in favor of Her_lib

      template<class T> class List { /* ... */ }; // additional stuff
      // ...
}
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

When looking into a namespace, names explicitly declared there (including names declared by *using-declarations*) take priority over names made accessible in another scope by a *using-directive* (see also §C.10.1). Consequently, a user of ***My_lib*** will see the name clashes for ***String*** and ***Vector*** resolved in favor of ***His_lib::String*** and ***Her_lib::Vector***. Also, ***My_lib::List*** will be used by default independently of whether ***His_lib*** or ***Her_lib*** are providing a ***List***.

Usually, I prefer to leave a name unchanged when including it into a new namespace. In that way, I don't have to remember two different names for the same entity. However, sometimes a new name is needed or simply nice to have. For example:

```
namespace Lib2 {
	using namespace His_lib; // everything from His_lib
	using namespace Her_lib; // everything from Her_lib

	using His_lib::String;       // resolve potential clash in favor of His_lib
	using Her_lib::Vector;       // resolve potential clash in favor of Her_lib

	typedef Her_lib::String Her_string;          // rename

	template<class T> class His_vec              // ``rename''
		: public His_lib::Vector<T> { /* ... */ };

	template<class T> class List { /* ... */ }; // additional stuff
	// ...
}
```

There is no specific language mechanism for renaming. Instead, the general mechanisms for defining new entities are used.

8.2.9 Namespaces and Old Code [name.get]

Millions of lines of C and C++ code rely on global names and existing libraries. How can we use namespaces to alleviate problems in such code? Redesigning existing code isn't always a viable option. Fortunately, it is possible to use C libraries as if they were defined in a namespace. However, this cannot be done for libraries written in C++ (§9.2.4). On the other hand, namespaces are designed so that they can be introduced with minimal disruption into an older C++ program.

8.2.9.1 Namespaces and C [name.c]

Consider the canonical first C program:

```
#include <stdio.h>

int main()
{
	printf("Hello, world!\n");
}
```

Breaking this program wouldn't be a good idea. Making standard libraries special cases isn't a good idea either. Consequently, the language rules for namespaces are designed to make it relatively easy to take a program written without namespaces and turn it into a more explicitly structured one using namespaces. In fact, the calculator program (§6.1) is an example of this.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

The *using-directive* is the key to achieving this. For example, the declarations of the standard C I/O facilities from the C header *stdio.h* are wrapped in a namespace like this:

```
// stdio.h:

    namespace std {
        // ...
        int printf(const char* ... );
        // ...
    }
    using namespace std;
```

This achieves backwards compatibility. Also, a new header file *cstdio* is defined for people who don't want the names implicitly available:

```
// cstdio:

    namespace std {
        // ...
        int printf(const char* ... );
        // ...
    }
```

C++ standard library implementers who worry about replication of declarations will, of course, define *stdio.h* by including *cstdio*:

```
// stdio.h:

    #include<cstdio>
    using namespace std;
```

I consider nonlocal *using-directive*s primarily a transition tool. Most code referring to names from other namespaces can be expressed more clearly with explicit qualification and *using-declaration*s.

The relationship between namespaces and linkage is described in §9.2.4.

8.2.9.2 Namespaces and Overloading [name.over]

Overloading (§7.4) works across namespaces. This is essential to allow us to migrate existing libraries to use namespaces with minimal source code changes. For example:

```
// old A.h:

    void f(int);
    // ...

// old B.h:

    void f(char);
    // ...

// old user.c:

    #include "A.h"
    #include "B.h"
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
void g()
{
    f('a');    // calls the f() from B.h
}
```

This program can be upgraded to a version using namespaces without changing the actual code:

```
// new A.h:

    namespace A {
        void f(int);
        // ...
    }

// new B.h:

    namespace B {
        void f(char);
        // ...
    }

// new user.c:

    #include "A.h"
    #include "B.h"

    using namespace A;
    using namespace B;

    void g()
    {
        f('a');    // calls the f() from B.h
    }
```

Had we wanted to keep *user.c* completely unchanged, we would have placed the *using-directive*s in the header files.

8.2.9.3 Namespaces Are Open [name.open]

A namespace is open; that is, you can add names to it from several 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()
}
```

In this way, we can support large program fragments within a single namespace the way an older library or application lives within the single global namespace. To do this, we must distribute the namespace definition over several header and source code files. As shown by the calculator example (§8.2.4), the openness of namespaces allows us to present different interfaces to different kinds

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

of users by presenting different parts of a namespace. This openness is also an aid to transition. For example,

```
// my header:
    void f();  // my function
    // ...
    #include<stdio.h>
    int g();  // my function
    // ...
```

can be rewritten without reordering of the declarations:

```
// my header:

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

    #include<stdio.h>

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

When writing new code, I prefer to use many smaller namespaces (see §8.2.8) 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.

When defining a previously declared member of a namespace, it is safer to use the ***Mine***`::` syntax than to re-open ***Mine***. For example:

```
void Mine::ff()       // error: no ff() declared in Mine
{
    // ...
}
```

A compiler catches this error. However, because new functions can be defined within a namespace, a compiler cannot catch the equivalent error in a re-opened namespace:

```
namespace Mine {  // re-opening Mine to define functions

    void ff()  // oops! no ff() declared in Mine; ff() is added to Mine by this definition
    {
        // ...
    }

    // ...
}
```

The compiler has no way of knowing that you didn't want that new ***ff***`()`.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

8.3 Exceptions [name.except]

When a program is composed of separate modules, and especially when those modules come from separately developed libraries, error handling needs to be separated into two distinct parts:

[1] The reporting of error conditions that cannot be resolved locally
[2] The handling of errors detected elsewhere

The author of a library can detect run-time errors but does not in general have any idea what to do about them. The user of a library may know how to cope with such errors but cannot detect them – or else they would be handled in the user's code and not left for the library to find.

In the calculator example, we bypassed this problem by designing the program as a whole. By doing that, we could fit error handling into our overall framework. However, when we separate the logical parts of the calculator into separate namespaces, we see that every namespace depends on namespace *Error* (§8.2.2) and that the error handling in *Error* relies on every module behaving appropriately after an error. Let's assume that we don't have the freedom to design the calculator as a whole and don't want the tight coupling between *Error* and all other modules. Instead, assume that the parser, etc., are written without knowledge of how a driver might like to handle errors.

Even though *error()* was very simple, it embodied a strategy for error handling:

```
namespace Error {
    int no_of_errors;

    double error(const char* s)
    {
        std::cerr << "error: " << s << '\n';
        no_of_errors++;
        return 1;
    }
}
```

The *error()* function writes out an error message, supplies a default value that allows its caller to continue a computation, and keeps track of a simple error state. Importantly, every part of the program knows that *error()* exists, how to call it, and what to expect from it. For a program composed of separately-developed libraries, that would be too much to assume.

Exceptions are C++'s means of separating error reporting from error handling. In this section, exceptions are briefly described in the context of their use in the calculator example. Chapter 14 provides a more extensive discussion of exceptions and their uses.

8.3.1 Throw and Catch [name.throw]

The notion of an *exception* is provided to help deal with error reporting. For example:

```
struct Range_error {
    int i;
    Range_error(int ii) { i = ii; }   // constructor (§2.5.2, §10.2.3)
};
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
char to_char(int i)
{
    if (i<numeric_limits<char>::min() || numeric_limits<char>::max()<i) // see §22.2
        throw Range_Error();
    return c;
}
```

The *to_char*() function either returns the *char* with the numeric value *i* or throws a *Range_error*. The fundamental idea is that a function that finds a problem it cannot cope with *throws* an exception, hoping that its (direct or indirect) caller can handle the problem. A function that wants to handle a problem can indicate that it is willing to *catch* exceptions of the type used to report the problem. For example, to call *to_char*() and catch the exception it might throw, we could write:

```
void g(int i)
{
    try {
        char c = to_char(i);
        // ...
    }
    catch (Range_error) {
        cerr << "oops\n";
    }
}
```

The construct

```
    catch ( /* ... */ ) {
        // ...
    }
```

is called an *exception handler*. It can be used only immediately after a block prefixed with the keyword *try* or immediately after another exception handler; *catch* is also a keyword. The parentheses contain a declaration that is used in a way similar to how a function argument declaration is used. That is, it specifies the type of the objects that can be caught by this handler and optionally names the object caught. For example, if we wanted to know the value of the *Range_error* thrown, we would provide a name for the argument to *catch* exactly the way we name function arguments. For example:

```
void h(int i)
{
    try {
        char c = to_char(i);
        // ...
    }
    catch (Range_error x) {
        cerr << "oops: to_char(" << x.i << ")\n";
    }
}
```

If any code in a *try-block* – or called from it – throws an exception, the try-block's handlers will be

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

examined. If the exception thrown is of a type specified for a handler, that handler is executed. If not, the exception handlers are ignored and the *try-block* acts just like an ordinary block.

Basically, C++ exception handling is a way to transfer control to designated code in a calling function. Where needed, some information about the error can be passed along to the caller. C programmers can think of exception handling as a well-behaved mechanism replacing ***setjmp***/***longjmp*** (§16.1.2). The important interaction between exception handling and classes is described in Chapter 14.

8.3.2 Discrimination of Exceptions [name.discrimination]

Typically, a program will have several different possible run-time errors. Such errors can be mapped into exceptions with distinct names. I prefer to define types with no other purpose than exception handling. This minimizes confusion about their purpose. In particular, I never use a built-in type, such as ***int***, as an exception. In a large program, I would have no effective way to find unrelated uses of ***int*** exceptions. Thus, I could never be sure that such other uses didn't interfere with my use.

Our calculator (§6.1) must handle two kinds of run-time errors: syntax errors and attempts to divide by zero. No values need to be passed to a handler from the code that detects an attempt to divide by zero, so zero divide can be represented by a simple empty type:

```
struct Zero_divide { };
```

On the other hand, a handler would most likely prefer to get an indication of what kind of syntax error occurred. Here, we pass a string along:

```
struct Syntax_error {
	const char* p;
	Syntax_error(const char* q) { p = q; }
};
```

For notational convenience, I added a constructor (§2.5.2, §10.2.3) to the ***struct***.

A user of the parser can discriminate between the two exceptions by adding handlers for both to a ***try*** block. Where needed, the appropriate handler will be entered. If we ''fall through the bottom'' of a handler, the execution continues at the end of the list of handlers:

```
try {
	// ...
	expr(false);
	// we get here if and only if expr() didn't cause an exception
	// ...
}

catch (Syntax_error) {
	// handle syntax error
}
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
    catch (Zero_divide) {
        // handle divide by zero
    }
    // we get here if expr didn't cause an exception or if a Syntax_error
    // or Zero_divide exception was caught (and its handler didn't return,
    // throw an exception, or in some other way alter the flow of control).
```

A list of handlers looks a bit like a ***switch*** statement, but there is no need for ***break*** statements. The syntax of a list of handlers differs from the syntax of a list of cases partly for that reason and partly to indicate that each handler is a scope (§4.9.4).

A function need not catch all possible exceptions. For example, the previous *try-block* didn't try to catch exceptions potentially generated by the parser's input operations. Those exceptions simply ''pass through,'' searching for a caller with an appropriate handler.

From the language's point of view, an exception is considered handled immediately upon entry into its handler so that any exceptions thrown while executing a handler must be dealt with by the callers of the *try-block*. For example, this does not cause an infinite loop:

```
class input_overflow { /* ... */ };

void f()
{
    try {
        // ...
    }
    catch (input_overflow) {
        // ...
        throw input_overflow();
    }
}
```

Exception handlers can be nested. For example:

```
class XXII { /* ... */ };

void f()
{
    // ...
    try {
        // ...
    }
    catch (XXII) {
        try {
            // something complicated
        }
        catch (XXII) {
            // complicated handler code failed
        }
    }
    // ...
}
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

However, such nesting is rare in human-written code and is more often than not an indication of poor style.

8.3.3 Exceptions in the Calculator [name.calc]

Given the basic exception-handling mechanism, we can rework the calculator example from §6.1 to separate the handling of errors found at run-time from the main logic of the calculator. This will result in an organization of the program that more realistically matches what is found in programs built from separate, loosely connected parts.

First, *error*() can be eliminated. Instead, the parser functions know only the types used to signal errors:

```
namespace Error {
	struct Zero_divide { };

	struct Syntax_error {
		const char* p;
		Syntax_error(const char* q) { p = q; }
	};
}
```

The parser detects three syntax errors:

```
Token_value Lexer::get_token()
{
	using namespace std;	// to use cin, isalpha(), etc.

		// ...

		default:		// NAME, NAME =, or error
			if (isalpha(ch)) {
				cin.putback(ch);
				cin >> string_value;
				return curr_tok=NAME;
			}
			throw Error::Syntax_error("bad token");
	}
}

double Parser::prim(bool get)		// handle primaries
{
	// ...

	case Lexer::LP:
	{	double e = expr(true);
		if (curr_tok != Lexer::RP) throw Error::Syntax_error("`)´ expected");
		get_token();		// eat ')'
		return e;
	}
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
        case Lexer::END:
            return 1;
        default:
            throw Error::Syntax_error("primary expected");
        }
}
```

When a syntax error is detected, *throw* is used to transfer control to a handler defined in some (direct or indirect) caller. The *throw* operator also passes a value to the handler. For example,

```
throw Syntax_error("primary expected");
```

passes a *Syntax_error* object containing a pointer to the string *primary expected* to the handler.

Reporting a divide-by-zero error doesn't require any data to be passed along:

```
double Parser::term(bool get)        // multiply and divide
{
        // ...
        case Lexer::DIV:
            if (double d = prim(true)) {
                left /= d;
                break;
            }
            throw Error::Zero_divide();

        // ...
}
```

The driver can now be defined to handle *Zero_divide* and *Syntax_error* exceptions. For example:

```
int main(int argc, char* argv[])
{
        // ...
        while (*input) {
            try {
                Lexer::get_token();
                if (Lexer::curr_tok == Lexer::END) break;
                if (Lexer::curr_tok == Lexer::PRINT) continue;
                cout << Parser::expr(false) << '\n';
            }
            catch(Error::Zero_divide) {
                cerr << "attempt to divide by zero\n";
                skip();
            }
            catch(Error::Syntax_error e) {
                cerr << "syntax error:" << e.p << "\n";
                skip();
            }
        }

        if (input != &cin) delete input;
        return no_of_errors;
}
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

The function *skip*() tries to bring the parser into a well-defined state after an error by skipping tokens until it finds an end-of-line or a semicolon. It, *no_of_errors*, and *input* are obvious candidates for a *Driver* namespace:

```
namespace Driver {
    int no_of_errors;
    std::istream* input;
    void skip();
}

void Driver::skip()
{
    no_of_errors++;

    while (*input) {
        char ch;
        input->get(ch);

        switch (ch) {
        case '\n':
        case ';':
            input->get(ch);
            return;
        }
    }
}
```

The code for *skip*() is deliberately written at a lower level of abstraction than the parser code so as to avoid being caught by exceptions from the parser while handling parser exceptions.

I retained the idea of counting the number of errors and reporting that number as the program's return value. It is often useful to know if a program encountered an error even if it was able to recover from it.

I did not put *main*() in the *Driver* namespace. The global *main*() is the initial function of a program (§3.2); a *main*() in another namespace has no special meaning.

8.3.3.1 Alternative Error-Handling Strategies [name.strategy]

The original error-handling code was shorter and more elegant than the version using exceptions. However, it achieved that elegance by tightly coupling all parts of the program. That approach doesn't scale well to programs composed of separately developed libraries.

We could consider eliminating the separate error-handling function *skip*() by introducing a state variable in *main*(). For example:

```
int main(int argc, char* argv[])    // example of poor style
{
    // ...

    bool in_error = false;
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
        while (*Driver::input) {
            try {
                Lexer::get_token();
                if (Lexer::curr_tok == Lexer::END) break;
                if (Lexer::curr_tok == Lexer::PRINT) {
                    in_error = false;
                    continue;
                }
                if (in_error == false) cout << Parser::expr(false) << ´\n´;
            }

            catch(Error::Zero_divide) {
                cerr << "attempt to divide by zero\n";
                in_error = true;
            }
            catch(Error::Syntax_error e) {
                cerr << "syntax error:" << e.p << "\n";
                in_error = true;
            }
        }

        if (Driver::input != std::cin) delete Driver::input;
        return Driver::no_of_errors;
    }
```

I consider this a bad idea for several reasons:

[1] State variables are a common source of confusion and errors, especially if they are allowed to proliferate and affect larger sections of a program. In particular, I consider the version of ***main()*** using ***in_error*** less readable than the version using ***skip()***.

[2] It is generally a good strategy to keep error handling and ‘‘normal’’ code separate.

[3] Doing error handling using the same level of abstraction as the code that caused the error is hazardous; the error-handling code might repeat the same error that triggered the error handling in the first place. I leave it as an exercise to find how that can happen for the version of ***main()*** using ***in_error*** (§8.5[7]).

[4] It is more work to modify the ‘‘normal’’ code to add error-handling code than to add separate error-handling routines.

Exception handling is intended for dealing with nonlocal problems. If an error can be handled locally, it almost always should be. For example, there is no reason to use an exception to handle the too-many-arguments error:

```
    int main(int argc, char* argv[])
    {
        using namespace std;
        using namespace Driver;

        switch (argc) {
        case 1:                                 // read from standard input
            input = &cin;
            break;
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
        case 2:                                    // read argument string
                input = new istringstream(argv[1]);
                break;
        default:
                cerr << "too many arguments\n";
                return 1;
        }

        // as before
}
```

Exceptions are discussed further in Chapter 14.

8.4 Advice [name.advice]

[1] Use namespaces to express logical structure; §8.2.
[2] Place every nonlocal name, except *main*(), in some namespace; §8.2.
[3] Design a namespace so that you can conveniently use it without accidentally gaining access to unrelated namespaces; §8.2.4.
[4] Avoid very short names for namespaces; §8.2.7.
[5] If necessary, use namespace aliases to abbreviate long namespaces names; §8.2.7.
[6] Avoid placing heavy notational burdens on users of your namespaces; §8.2.2, §8.2.3.
[7] Use the *Namespace*::*member* notation when defining namespace members; §8.2.8.
[8] Use *using namespace* only for transition or within a local scope; §8.2.9.
[9] Use exceptions to decouple the treatment of ‘‘errors’’ from the code dealing with the ordinary processing; §8.3.3.
[10] Use user-defined rather than built-in types as exceptions; §8.3.2.
[11] Don’t use exceptions when local control structures are sufficient; §8.3.3.1.

8.5 Exercises [name.exercises]

1. (∗2.5) Write a doubly-linked list of *string* module in the style of the *Stack* module from §2.4. Exercise it by creating a list of names of programming languages. Provide a *sort*() function for that list, and provide a function that reverses the order of the strings in it.
2. (∗2) Take some not-too-large program that uses at least one library that does not use namespaces and modify it to use a namespace for that library. Hint: §8.2.9.
3. (∗2) Modify the desk calculator program into a module in the style of §2.4 using namespaces. Don’t use any global *using-directive*s. Keep a record of the mistakes you made. Suggest ways of avoiding such mistakes in the future.
4. (∗1) Write a program that throws an exception in one function and catches it in another.
5. (∗2) Write a program consisting of functions calling each other to a calling depth of 10. Give each function an argument that determines at which level an exception is thrown. Have *main*() catch these exceptions and print out which exception is caught. Don’t forget the case in which an exception is caught in the function that throws it.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

6. (∗2) Modify the program from §8.5[5] to measure if there is a difference in the cost of catching exceptions depending on where in a class stack the exception is thrown. Add a string object to each function and measure again.
7. (∗1) Find the error in the first version of ***main***`()` in §8.3.3.1.
8. (∗2) Write a function that either returns a value or that throws that value based on an argument. Measure the difference in run-time between the two ways.
9. (∗2) Modify the calculator version from §8.5[3] to use exceptions. Keep a record of the mistakes you make. Suggest ways of avoiding such mistakes in the future.
10. (∗2.5) Write ***plus***`()`, ***minus***`()`, ***multiply***`()`, and ***divide***`()` functions that check for possible overflow and underflow and that throw exceptions if such errors happen.
11. (∗2) Modify the calculator to use the functions from §8.5[10].

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.