PREV
2. User-Defined Types

NEXT
4. Classes

3. Modularity

Don't interrupt me while I'm interrupting.

– Winston S. Churchill

3.1. INTRODUCTION

A C++ program consists of many separately developed parts, such as functions (§1.3), user-defined types (Chapter 2), class hierarchies (§4.5), and templates (Chapter 5). The key to managing this is to clearly define the interactions among those parts. The first and most important step is to distinguish between the interface to a part and its implementation. At the language level, C++ represents interfaces by declarations. A *declaration* specifies all that's needed to use a function or a type. For example:

Click here to view code image

```
double sqrt(double);      // the square root function takes a double

class Vector {
public:
        Vector(int s);
        double& operator[](int i);
        int size();
private:
        double* elem; // elem points to an array of sz doubles
        int sz;
};
```

The key point here is that the function bodies, the function *definitions*, are "elsewhere." For this example, we might like for the representation of `Vector` to be "elsewhere" also, but we will deal with that later (abstract types; §4.3). The definition of `sqrt()` will look like this:

Click here to view code image

```
double sqrt(double d)                    // definition of sqrt()
{
      // ... algorithm as found in math textbook ...
}
```

For `Vector`, we need to define all three member functions:

Click here to view code image

```
Vector::Vector(int s)                          // definition of th
        :elem{new double[s]}, sz{s}            // initialize membe
{
}

double& Vector::operator[](int i)              // definition of subscr
{
        return elem[i];
}
```

```
int Vector::size()                              // definition of siz
{
        return sz;
}
```

We must define Vector's functions, but not sqrt() because it is part of the standard library. However, that makes no real difference: a library is simply some "other code we happen to use" written with the same language facilities as we use.

3.2. SEPARATE COMPILATION

C++ supports a notion of separate compilation where user code sees only declarations of the types and functions used. The definitions of those types and functions are in separate source files and compiled separately. This can be used to organize a program into a set of semi-independent code fragments. Such separation can be used to minimize compilation times and to strictly enforce separation of logically distinct parts of a program (thus minimizing the chance of errors). A library is often a collection of separately compiled code fragments (e.g., functions).

Typically, we place the declarations that specify the interface to a module in a file with a name indicating its intended use. For example:

Click here to view code image

```
// Vector.h:

class Vector {
public:
        Vector(int s);
        double& operator[](int i);
        int size();
private:
        double* elem;       // elem points to an array of sz doubles
        int sz;
};
```

This declaration would be placed in a file Vector.h, and users will *include* that file, called a *header file*, to access that interface. For example:

Click here to view code image

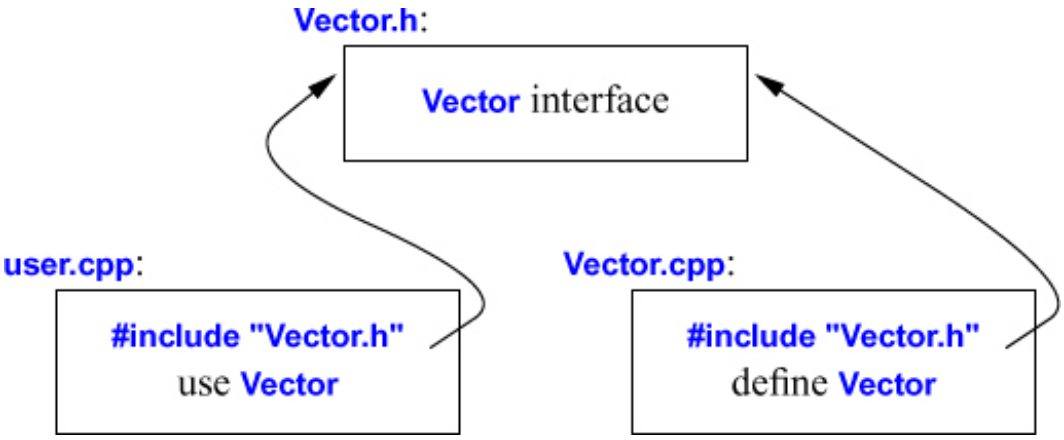
```
// user.cpp:

#include "Vector.h"        // get Vector's interface
#include <cmath>           // get the the standard-library math fur

using namespace std;       // make std members visible (§3.3)

double sqrt_sum(Vector& v)
{
        double sum = 0;
        for (int i=0; i!=v.size(); ++i)
                sum+=sqrt(v[i]);                    // sum of square roots
        return sum;
}
```

To help the compiler ensure consistency, the **.cpp** file providing the implementation of **Vector** will also include the **.h** file providing its interface:

Click here to view code image

```
// Vector.cpp:

#include "Vector.h" // get the interface

Vector::Vector(int s)
        :elem{new double[s]}, sz{s}                 // initialize member
{
}

double& Vector::operator[](int i)
{
        return elem[i];
}

int Vector::size()
{
        return sz;
}
```

The code in user.cpp and Vector.cpp shares the Vector interface information presented in Vector.h, but the two files are otherwise independent and can be separately compiled. Graphically, the program fragments can be represented like this:



Strictly speaking, using separate compilation isn't a language issue; it is an issue of how best to take advantage of a particular language implementation. However, it is of great practical importance. The best approach is to maximize modularity, represent that modularity logically through language features, and then exploit the modularity physically through files for effective separate compilation.

3.3. NAMESPACES

In addition to functions (§1.4), classes (§2.3), and enumerations (§2.5), C++ offers *namespaces* as a mechanism for expressing that some declarations belong together and that their names shouldn't clash with other names. For example, I might want to experiment with my own complex number type (§4.2.1, §12.4):

Click here to view code image

```
namespace My_code {
    class complex {

            // ...
    };

    complex sqrt(complex);
    // ...

    int main();
}

int My_code::main()
```

```
{
        complex z {1,2};
        auto z2 = sqrt(z);
        std::cout << '{' << z2.real() << ',' << z2.imag() << "}\n";
        // ...
};

int main()
{
        return My_code::main();
}
```

By putting my code into the namespace My_code, I make sure that my names do not conflict with the standard-library names in namespace std (§3.3). The precaution is wise, because the standard library does provide support for complex arithmetic (§4.2.1, §12.4).

The simplest way to access a name in another namespace is to qualify it with the namespace name (e.g., std::cout and My_code::main). The "real main() " is defined in the global namespace, that is, not local to a defined namespace, class, or function. To gain access to names in the standard-library namespace, we can use a using-directive:

```
using namespace std;
```

A using-directive makes names from the named namespace accessible as if they were local to the scope in which we placed the directive. So after the using-directive for std, we can simply write cout rather than std::std.

Namespaces are primarily used to organize larger program components, such as libraries. They simplify the composition of a program out of separately developed parts.

3.4. ERROR HANDLING

Error handling is a large and complex topic with concerns and ramifications that go far beyond language facilities into programming techniques and tools. However, C++ provides a few features to help. The major tool is the type system itself. Instead of painstakingly building up our applications from the built-in types (e.g., char, int, and double) and statements (e.g., if, while, and for), we build more types that are appropriate for our applications (e.g., string, map, and regex) and algorithms (e.g., sort(), find_if(), and draw_all()). Such higher-level constructs simplify our programming, limit our opportunities for mistakes (e.g., you are unlikely to try to apply a tree traversal to a dialog box), and increase the compiler's chances of catching such errors. The majority of C++ constructs are dedicated to the design and implementation of elegant and efficient abstractions (e.g., user-defined types and algorithms using them). One effect of this modularity and abstraction (in particular, the use of libraries) is that the point where a run-time error can be detected is separated from the point where it can be handled. As programs grow, and especially when libraries are used extensively, standards for handling errors become important. It is a good idea to design and articulate a strategy for error handling early on in the development of a program.

3.4.1. Exceptions

Consider again the Vector example. What *ought* to be done when we try to access an element that is out of range for the vector from §2.3?

• The writer of Vector doesn't know what the user would like to have done in this case (the writer of Vector typically doesn't even know in which program the vector will be running).

• The user of Vector cannot consistently detect the problem (if the user could, the out-of-range access wouldn't happen in the first place).

The solution is for the Vector implementer to detect the attempted out-of-range access and then tell the user about it. The user can then take appropriate action. For example, Vector::operator[]() can detect an attempted out-of-range access and throw an out_of_range exception:

Click here to view code image

```
double& Vector::operator[](int i)
{
        if (i<0 || size()<=i)
                throw out_of_range{"Vector::operator[]"};
```

```
        return elem[i];
    }
```

The throw transfers control to a handler for exceptions of type out_of_range in some function that directly or indirectly called Vector::operator[](). To do that, the implementation will *unwind* the function call stack as needed to get back to the context of that caller. That is, the exception handling mechanism will exit scopes and function as needed to get back to a caller that has expressed interest in handling that kind of exception, invoking destructors (§4.2.2) along the way as needed. For example:

Click here to view code image

```
void f(Vector& v)
{
        // ...
        try { // exceptions here are handled by the handler defined b

                v[v.size()] = 7; // try to access beyond the end of v
        }
        catch (out_of_range) {          // oops: out_of_range error
                // ... handle range error ...
        }
        // ...
}
```

We put code for which we are interested in handling exceptions into a try-block. That attempted assignment to v[v.size()] will fail. Therefore, the catch-clause providing a handler for out_of_range will be entered. The out_of_range type is defined in the standard library (in <stdexcept>) and is in fact used by some standard-library container access functions.

Use of the exception-handling mechanisms can make error handling simpler, more systematic, and more readable. To achieve that don't overuse try-statements. The main technique for making error handling simple and systematic (called *Resource Aquisition Is Initialization*) is explained in §4.2.2.

A function that should never throw an exception can be declared noexcept. For example:

Click here to view code image

```
void user(int sz) noexcept
{
        Vector v(sz);
        iota(&v[0],&v[sz],1); //  fill v with 1,2,3,4...
        //  ...
}
```

If all good intent and planning fails, so that user() still throws, the standard-library function terminate() is called to immediately terminate the program.

3.4.2. Invariants

The use of exceptions to signal out-of-range access is an example of a function checking its argument and refusing to act because a basic assumption, a *precondition*, didn't hold. Had we formally specified Vector's subscript operator, we would have said something like "the index must be in the [0:size()) range," and that was in fact what we tested in our operator[](). The [a:b) notation specifies a half-open range, meaning that a is part of the range, but b is not. Whenever we define a function, we should consider what its preconditions are and if feasible test them.

However, operator[]() operates on objects of type Vector and nothing it does makes any sense unless the members of Vector have "reasonable" values. In particular, we did say "elem points to an array of sz doubles" but we only said that in a comment. Such a statement of what is assumed to be true for a class is called a *class invariant*, or simply an *invariant*. It is the job of a constructor to establish the invariant for its class (so that the member functions can rely on it) and for the member functions to make sure that the invariant holds when they exit. Unfortunately, our Vector constructor only partially did its job. It properly initialized the Vector members, but it failed to check that the arguments passed to it made sense. Consider:

```
Vector v(–27);
```

This is likely to cause chaos.

Here is a more appropriate definition:

Click here to view code image

```
Vector::Vector(int s)
{
        if (s<0)
                throw length_error{};
        elem = new double[s];
        sz = s;
}
```

I use the standard-library exception `length_error` to report a non-positive number of elements because some standard-library operations use that exception to report problems of this kind. If operator `new` can't find memory to allocate, it throws a `std::bad_alloc`. We can now write:

Click here to view code image

```
void test()
{
        try {
                Vector v(–27);
        }
        catch (std::length_error) {
                // handle negative size
        }
        catch (std::bad_alloc) {
                // handle memory exhaustion
        }
}
```

You can define your own classes to be used as exceptions and have them carry arbitrary information from a point where an error is detected to a point where it can be handled (§3.4.1).

Often, a function has no way of completing its assigned task after an exception is thrown. Then, "handling" an exception simply means doing some minimal local cleanup and rethrowing the exception. To throw (*rethrow*) the exception caught in an exception handler, we simply write `throw;`. For example:

Click here to view code image

```
void test()
{
        try {
                Vector v(-27);
        }
        catch (std::length_error) {
                cout << "test failed: length error\n";
                throw;        // rethrow
        }
        catch (std::bad_alloc) {
                // Ouch! test() is not designed to handle memory exha
                std::terminate();            // terminate the program
        }
}
```

The notion of invariants is central to the design of classes, and preconditions serve a similar role in the design of functions. Invariants

• helps us to understand precisely what we want

• forces us to be specific; that gives us a better chance of getting our code correct (after debugging and testing).

The notion of invariants underlies C++'s notions of resource management supported by constructors (Chapter 4) and destructors (§4.2.2, §11.2).

3.4.3. Static Assertions

Exceptions report errors found at run time. If an error can be found at compile time, it is usually preferable to do so. That's what much of the type system and the facilities for specifying the interfaces to user-defined types are for. However, we can also perform simple checks on other properties that are known at compile time and report failures as compiler error messages. For example:

Click here to view code image

```
static_assert(4<=sizeof(int), "integers are too small"); // check int
```

This will write `integers are too small` if `4<=sizeof(int)` does not hold, that is, if an `int` on this system does not have at least 4 bytes. We call such statements of expectations *assertions*.

The static_assert mechanism can be used for anything that can be expressed in terms of constant expressions (§1.7). For example:

Click here to view code image

```
constexpr double C = 299792.458;                        // k

void f(double speed)
{
        const double local_max = 160.0/(60*60);         // 160
        static_asser t(speed<C,"can't go that fast");   // error: sp
        static_assert(local_max<C,"can't go that fast"); // OK

        // ...
}
```

In general, static_assert(A,S) prints S as a compiler error message if A is not true.

The most important uses of static_assert come when we make assertions about types used as parameters in generic programming (§5.4, §11.6).

For runtime-checked assertions, use exceptions.

3.5. ADVICE

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapters 13-15 of [Stroustrup,2013].

[2] Distinguish between declarations (used as interfaces) and definitions (used as implementations); §3.1.

[3] Use header files to represent interfaces and to emphasize logical structure; §3.2.

[4] #include a header in the source file that implements its functions; §3.2.

[5] Avoid non-inline function definitions in headers; §3.2.

[6] Use namespaces to express logical structure; §3.3.

[7] Use using-directives for transition, for foundational libraries (such as std), or within a local scope; §3.3.

[8] Don't put a using-directive in a header file; §3.3.

[9] Throw an exception to indicate that you cannot perform an assigned task; §3.4.

[10] Use exceptions for error handling; §3.4.

[11] Develop an error-handling strategy early in a design; §3.4.

[12] Use purpose-designed user-defined types as exceptions (not built-in types); §3.4.1.

[13] Don't try to catch every exception in every function; §3.4.

[14] If your function may not throw, declare it noexcept; §3.4.

[15] Let a constructor establish an invariant, and throw if it cannot; §3.4.2.

[16] Design your error-handling strategy around invariants; §3.4.2.

[17] What can be checked at compile time is usually best checked at compile time (using static_assert); §3.4.3.