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:

```
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 std::bad_alloc. We can now write

```
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 (§13.5).

Often a function has to do some work before completing its assigned task after an exception is thrown. This work is called "handling" an exception simply means doing some minimal local cleanup and rethrowing the exception (§13.5.2.1).

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

- It helps us to understand precisely what we want.
- It forces us to be specific; that gives us better chance of getting our code correct (after debugging and testing).

More concretely, the notion of invariants underlies C++'s notions of resource management supported by constructors (§2.3.2) and destructors (§3.2.1.2, §5.2). See also §13.4, §16.3 and §17.2.

2.4.3.3 Static Assertions [tour1.assert]

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:

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

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. Such statements of expectations are called *assertions*.

The `static_assert` mechanism can be used for anything that can be expressed in terms of constant expressions (§2.2.3, §10.4). For example:

```
constexpr double C = 299792.458;          // km/s

void f(double speed)
{
    const double local_max = 160*60*60;           // 160 km/h

    static_assert(speed<C,"can't go that fast");      // error: speed must be a constant
    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_cast` come when we to make assertions about types used as parameters in generic programming (§5.4.2, §24.3).

2.5 Postscript [tour1.postscript]

The topics covered in this chapter roughly correspond to the contents of Part II (Chapters 5-15). Those are the parts of C++ that underlie all programming techniques and styles supported by C++. Experienced C and C++ programmers, please note that this foundation does not closely correspond to the C or C++98 subsets of C++ (that is, C++11).

2.6 Advice [tour1.advice]

[1] Don't panic! All will become clear in time; §2.1.
[2] You don't have to know every detail of C++ to write good programs; §1.3.1.
[3] Focus on programming techniques, not on language features; §2.1.