A matrix represented as a multidimensional array is passed as a pointer (rather than copied; §7.4). The first dimension of an array is irrelevant to finding the location of an element; it simply states how many elements (here, `3`) of the appropriate type (here, `int[5]`) are present. For example, look at the layout of `ma` above and note that by knowing only that the second dimension is `5`, we can locate `ma[i][5]` for any `i`. The first dimension can therefore be passed as an argument:

```
void print_mi5(int m[][5], int dim1)
{
        for (int i = 0; i!=dim1; i++) {
                for (int j = 0; j!=5; j++)
                        cout << m[i][j] << '\t';
                cout << '\n';
        }
}
```

When both dimensions need to be passed, the ''obvious solution'' does not work:

```
void print_mij(int m[][], int dim1, int dim2)      // doesn't behave as most people would think
{
        for (int i = 0; i!=dim1; i++) {
                for (int j = 0; j!=dim2; j++)
                        cout << m[i][j] << '\t';       // surprise!
                cout << '\n';
        }
}
```

Fortunately, the argument declaration `m[][]` is illegal because the second dimension of a multidimensional array must be known in order to find the location of an element. However, the expression `m[i][j]` is (correctly) interpreted as `*(*(m+i)+j)`, although that is unlikely to be what the programmer intended. A correct solution is:

```
void print_mij(int* m, int dim1, int dim2)
{
        for (int i = 0; i!=dim1; i++) {
                for (int j = 0; j!=dim2; j++)
                        cout << m[i*dim2+j] << '\t'; // obscure
                cout << '\n';
        }
}
```

The expression used for accessing the members in `print_mij()` is equivalent to the one the compiler generates when it knows the last dimension.

To call this function, we pass a matrix as an ordinary pointer:

```
int test()
{
        int v[3][5] = {
                {0,1,2,3,4}, {10,11,12,13,14}, {20,21,22,23,24}
        };
```

```
        print_m35(v);
        print_mi5(v,3);
        print_mij(&v[0][0],3,5);
}
```

Note the use of `&v[0][0]` for the last call; `v[0]` would do because it is equivalent, but `v` would be a type error. This kind of subtle and messy code is best hidden. If you must deal directly with multi-dimensional arrays, consider encapsulating the code relying on it. In that way, you might ease the task of the next programmer to touch the code. Providing a multidimensional array type with a proper subscripting operator saves most users from having to worry about the layout of the data in the array (§29.2.2, §40.5.2).

The standard `vector` (§31.4) doesn't suffer from these problems.

7.5 Pointers and const

C++ offers two related meanings of ''constant'':

- `constexpr`: Evaluate at compile time (§2.2.3, §10.4).
- `const`: Do not modify in this scope (§2.2.3).

Basically, `constexpr`'s role is to enable and ensure compile-time evaluation, whereas `const`'s primary role is to specify immutability in interfaces. This section is primarily concerned with the second role: interface specification.

Many objects don't have their values changed after initialization:

- Symbolic constants lead to more maintainable code than using literals directly in code.
- Many pointers are often read through but never written through.
- Most function parameters are read but not written to.

To express this notion of immutability after initialization, we can add `const` to the definition of an object. For example:

```
const int model = 90;           // model is a const
const int v[] = { 1, 2, 3, 4 }; // v[i] is a const
const int x;                    // error: no initializer
```

Because an object declared `const` cannot be assigned to, it must be initialized.

Declaring something `const` ensures that its value will not change within its scope:

```
void f()
{
        model = 200;    // error
        v[2] = 3;       // error
}
```

Note that `const` modifies a type; it restricts the ways in which an object can be used, rather than specifying how the constant is to be allocated. For example:

```
void g(const X* p)
{
        // can't modify *p here
}
```

```
void h()
{
        X val;      // val can be modified here
        g(&val);
        // ...
}
```

When using a pointer, two objects are involved: the pointer itself and the object pointed to. ''Prefixing'' a declaration of a pointer with **const** makes the object, but not the pointer, a constant. To declare a pointer itself, rather than the object pointed to, to be a constant, we use the declarator operator ***const** instead of plain *. For example:

```
void f1(char* p)
{
        char s[] = "Gorm";

        const char* pc = s;             // pointer to constant
        pc[3] = 'g';                    // error: pc points to constant
        pc = p;                         // OK

        char *const cp = s;             // constant pointer
        cp[3] = 'a';                    // OK
        cp = p;                         // error: cp is constant

        const char *const cpc = s;      // const pointer to const
        cpc[3] = 'a';                   // error: cpc points to constant
        cpc = p;                        // error: cpc is constant
}
```

The declarator operator that makes a pointer constant is ***const**. There is no **const*** declarator operator, so a **const** appearing before the * is taken to be part of the base type. For example:

```
char *const cp;         // const pointer to char
char const* pc;         // pointer to const char
const char* pc2;        // pointer to const char
```

Some people find it helpful to read such declarations right-to-left, for example, ''**cp** is a **const** pointer to a **char**'' and ''**pc2** is a pointer to a **char const**.''

An object that is a constant when accessed through one pointer may be variable when accessed in other ways. This is particularly useful for function arguments. By declaring a pointer argument **const**, the function is prohibited from modifying the object pointed to. For example:

```
const char* strchr(const char* p, char c);      // find first occurrence of c in p
char* strchr(char* p, char c);                  // find first occurrence of c in p
```

The first version is used for strings where the elements mustn't be modified and returns a pointer to **const** that does not allow modification. The second version is used for mutable strings.

You can assign the address of a non-**const** variable to a pointer to constant because no harm can come from that. However, the address of a constant cannot be assigned to an unrestricted pointer because this would allow the object's value to be changed. For example:

```
void f4()
{
    int a = 1;
    const int c = 2;
    const int* p1 = &c;  // OK
    const int* p2 = &a;  // OK
    int* p3 = &c;        // error: initialization of int* with const int*
    *p3 = 7;             // try to change the value of c
}
```

It is possible, but typically unwise, to explicitly remove the restrictions on a pointer to **const** by explicit type conversion (§16.2.9, §11.5).

7.6 Pointers and Ownership

A resource is something that has to be acquired and later released (§5.2). Memory acquired by **new** and released by **delete** (§11.2) and files opened by **fopen()** and closed by **fclose()** (§43.2) are examples of resources where the most direct handle to the resource is a pointer. This can be most confusing because a pointer is easily passed around in a program, and there is nothing in the type system that distinguishes a pointer that owns a resource from one that does not. Consider:

```
void confused(int* p)
{
    // delete p?
}

int global {7};

void f()
{
    X* pn = new int{7};
    int i {7};
    int q = &i;
    confused(pn);
    confused(q);
    confused(&global);
}
```

If **confused()** **delete**s **p** the program will seriously misbehave for the second two calls because we may not **delete** objects not allocated by **new** (§11.2). If **confused()** does not **delete** **p** the program leaks (§11.2.1). In this case, obviously **f()** must manage the lifetime of the object it creates on the free store, but in general keeping track of what needs to be **delete**d in a large program requires a simple and consistent strategy.

It is usually a good idea to immediately place a pointer that represents ownership in a resource handle class, such as **vector**, **string**, and **unique_ptr**. That way, we can assume that every pointer that is not within a resource handle is not an owner and must not be **delete**d. Chapter 13 discusses resource management in greater detail.

7.7 References

A pointer allows us to pass potentially large amounts of data around at low cost: instead of copying the data we simply pass its address as a pointer value. The type of the pointer determines what can be done to the data through the pointer. Using a pointer differs from using the name of an object in a few ways:

- We use a different syntax, for example, `*p` instead of `obj` and `p->m` rather than `obj.m`.
- We can make a pointer point to different objects at different times.
- We must be more careful when using pointers than when using an object directly: a pointer may be a `nullptr` or point to an object that wasn't the one we expected.

These differences can be annoying; for example, some programmers find `f(&x)` ugly compared to `f(x)`. Worse, managing pointer variables with varying values and protecting code against the possibility of `nullptr` can be a significant burden. Finally, when we want to overload an operator, say `+`, we want to write `x+y` rather than `&x+&y`. The language mechanism addressing these problems is called a *reference*. Like a pointer, a *reference* is an alias for an object, is usually implemented to hold a machine address of an object, and does not impose performance overhead compared to pointers, but it differs from a pointer in that:

- You access a reference with exactly the same syntax as the name of an object.
- A reference always refers to the object to which it was initialized.
- There is no ''null reference,'' and we may assume that a reference refers to an object (§7.7.4).

A reference is an alternative name for an object, an alias. The main use of references is for specifying arguments and return values for functions in general and for overloaded operators (Chapter 18) in particular. For example:

```
template<class T>
class vector {
        T* elem;
        // ...
public:
        T& operator[](int i) { return elem[i]; }                // return reference to element
        const T& operator[](int i) const { return elem[i]; }    // return reference to const element

        void push_back(const T& a);                             // pass element to be added by reference
        // ...
};

void f(const vector<double>& v)
{
        double d1 = v[1];       // copy the value of the double referred to by v.operator[](1) into d1
        v[2] = 7;               // place 7 in the double referred to by the result of v.operator[](2)

        v.push_back(d1);        // give push_back() a reference to d1 to work with
}
```

The idea of passing function arguments by reference is as old as high-level programming languages (the first version of Fortran used that).

To reflect the lvalue/rvalue and **const**/non-**const** distinctions, there are three kinds of references:

- *lvalue references*: to refer to objects whose value we want to change
- **const** *references*: to refer to objects whose value we do not want to change (e.g., a constant)
- *rvalue references*: to refer to objects whose value we do not need to preserve after we have used it (e.g., a temporary)

Collectively, they are called *references*. The first two are both called *lvalue references*.

7.7.1 Lvalue References

In a type name, the notation **X&** means "reference to **X**." It is used for references to lvalues, so it is often called an *lvalue reference*. For example:

```
void f()
{
        int var = 1;
        int& r {var};       // r and var now refer to the same int
        int x = r;          // x becomes 1

        r = 2;              // var becomes 2
}
```

To ensure that a reference is a name for something (that is, that it is bound to an object), we must initialize the reference. For example:

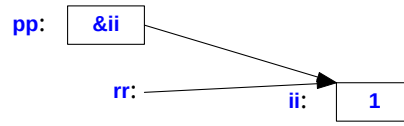
```
int var = 1;
int& r1 {var};          // OK: r1 initialized
int& r2;                // error: initializer missing
extern int& r3;         // OK: r3 initialized elsewhere
```

Initialization of a reference is something quite different from assignment to it. Despite appearances, no operator operates on a reference. For example:

```
void g()
{
        int var = 0;
        int& rr {var};
        ++rr;               // var is incremented to 1
        int* pp = &rr;      // pp points to var
}
```

Here, **++rr** does not increment the reference **rr**; rather, **++** is applied to the **int** to which **rr** refers, that is, to **var**. Consequently, the value of a reference cannot be changed after initialization; it always refers to the object it was initialized to denote. To get a pointer to the object denoted by a reference **rr**, we can write **&rr**. Thus, we cannot have a pointer to a reference. Furthermore, we cannot define an array of references. In that sense, a reference is not an object.

The obvious implementation of a reference is as a (constant) pointer that is dereferenced each time it is used. It doesn't do much harm to think about references that way, as long as one remembers that a reference isn't an object that can be manipulated the way a pointer is:



In some cases, the compiler can optimize away a reference so that there is no object representing that reference at run time.

Initialization of a reference is trivial when the initializer is an lvalue (an object whose address you can take; see §6.4). The initializer for a "plain" **T&** must be an lvalue of type **T**.

The initializer for a **const T&** need not be an lvalue or even of type **T**. In such cases:

[1] First, implicit type conversion to **T** is applied if necessary (see §10.5).

[2] Then, the resulting value is placed in a temporary variable of type **T**.

[3] Finally, this temporary variable is used as the value of the initializer.

Consider:

```
double& dr = 1;              // error: lvalue needed
const double& cdr {1};       // OK
```

The interpretation of this last initialization might be:

```
double temp = double{1};     // first create a temporary with the right value
const double& cdr {temp};    // then use the temporary as the initializer for cdr
```

A temporary created to hold a reference initializer persists until the end of its reference's scope.

References to variables and references to constants are distinguished because introducing a temporary for a variable would have been highly error-prone; an assignment to the variable would become an assignment to the – soon-to-disappear – temporary. No such problem exists for references to constants, and references to constants are often important as function arguments (§18.2.4).

A reference can be used to specify a function argument so that the function can change the value of an object passed to it. For example:

```
void increment(int& aa)
{
    ++aa;
}

void f()
{
    int x = 1;
    increment(x);        // x = 2
}
```

The semantics of argument passing are defined to be those of initialization, so when called, **increment**'s argument **aa** became another name for **x**. To keep a program readable, it is often best to avoid functions that modify their arguments. Instead, you can return a value from the function explicitly:

```
int next(int p) { return p+1; }

void g()
{
        int x = 1;
        increment(x);           // x = 2
        x = next(x);            // x = 3
}
```

The `increment(x)` notation doesn't give a clue to the reader that `x`'s value is being modified, the way `x=next(x)` does. Consequently, "plain" reference arguments should be used only where the name of the function gives a strong hint that the reference argument is modified.

References can also be used as return types. This is mostly used to define functions that can be used on both the left-hand and right-hand sides of an assignment. A `Map` is a good example. For example:

```
template<class K, class V>
class Map {             // a simple map class
public:
        V& operator[](const K& v);      // return the value corresponding to the key v

        pair<K,V>* begin() { return &elem[0]; }
        pair<K,V>* end() { return &elem[0]+elem.size(); }
private:
        vector<pair<K,V>> elem;         // {key,value} pairs
};
```

The standard-library `map` (§4.4.3, §31.4.3) is typically implemented as a red-black tree, but to avoid distracting implementation details, I'll just show an implementation based on linear search for a key match:

```
template<class K, class V>
V& Map<K,V>::operator[](const K& k)
{
        for (auto& x : elem)
                if (k == x.first)
                        return x.second;

        elem.push_back({k,V{}});        // add pair at end (§4.4.2)
        return elem.back().second;      // return the (default) value of the new element
}
```

I pass the key argument, `k`, by reference because it might be of a type that is expensive to copy. Similarly, I return the value by reference because it too might be of a type that is expensive to copy. I use a `const` reference for `k` because I don't want to modify it and because I might want to use a literal or a temporary object as an argument. I return the result by non-`const` reference because the user of a `Map` might very well want to modify the found value. For example:

```
int main() // count the number of occurrences of each word on input
{
        Map<string,int> buf;

        for (string s; cin>>s;) ++buf[s];

        for (const auto& x : buf)
                cout << x.first << ": " << x.second << '\n';
}
```

Each time around, the input loop reads one word from the standard input stream **cin** into the string **s** (§4.3.2) and then updates the counter associated with it. Finally, the resulting table of different words in the input, each with its number of occurrences, is printed. For example, given the input

```
aa bb bb aa aa bb aa aa
```

this program will produce

```
aa: 5
bb: 3
```

The range- **for** loop works for this because **Map** defined **begin()** and **end()**, just as is done for the standard-library **map**.

7.7.2 Rvalue References

The basic idea of having more than one kind of reference is to support different uses of objects:

- A non-**const** lvalue reference refers to an object, to which the user of the reference can write.
- A **const** lvalue reference refers to a constant, which is immutable from the point of view of the user of the reference.
- An rvalue reference refers to a temporary object, which the user of the reference can (and typically will) modify, assuming that the object will never be used again.

We want to know if a reference refers to a temporary, because if it does, we can sometimes turn an expensive copy operation into a cheap move operation (§3.3.2, §17.1, §17.5.2). An object (such as a **string** or a **list**) that is represented by a small descriptor pointing to a potentially huge amount of information can be simply and cheaply moved if we know that the source isn't going to be used again. The classic example is a return value where the compiler knows that a local variable returned will never again be used (§3.3.2).

An rvalue reference can bind to an rvalue, but not to an lvalue. In that, an rvalue reference is exactly opposite to an lvalue reference. For example:

```
string var {"Cambridge"};
string f();

string& r1 {var};          // lvalue reference, bind r1 to var (an lvalue)
string& r2 {f()};          // lvalue reference, error: f() is an rvalue
string& r3 {"Princeton"};  // lvalue reference, error: cannot bind to temporary
```

```
string&& rr1 {f()};          // rvalue reference, fine: bind rr1 to rvalue (a temporary)
string&& rr2 {var};          // rvalue reference, error: var is an lvalue
string&& rr3 {"Oxford"};     // rr3 refers to a temporary holding "Oxford"

const string cr1& {"Harvard"};// OK: make temporary and bind to cr1
```

The **&&** declarator operator means ''rvalue reference.'' We do *not* use **const** rvalue references; most of the benefits from using rvalue references involve writing to the object to which it refers. Both a **const** lvalue reference and an rvalue reference can bind to an rvalue. However, the purposes will be fundamentally different:

- We use rvalue references to implement a ''destructive read'' for optimization of what would otherwise have required a copy.
- We use a **const** lvalue reference to prevent modification of an argument.

An object referred to by an rvalue reference is accessed exactly like an object referred to by an lvalue reference or an ordinary variable name. For example:

```
string f(string&& s)
{
    if (s.size())
        s[0] = toupper(s[0]);
    return s;
}
```

Sometimes, a programmer knows that an object won't be used again, even though the compiler does not. Consider:

```
template<class T>
swap(T& a, T& b)          // "old-style swap"
{
    T tmp {a};// now we have two copies of a
    a = b;    // now we have two copies of b
    b = tmp;  // now we have two copies of tmp (aka a)
}
```

If **T** is a type for which it can be expensive to copy elements, such as **string** and **vector**, this **swap()** becomes an expensive operation. Note something curious: we didn't want any copies at all; we just wanted to move the values of **a**, **b**, and **tmp** around. We can tell that to the compiler:

```
template<class T>
void swap(T& a, T& b)     // "perfect swap" (almost)
{
    T tmp {static_cast<T&&>(a)};  // the initialization may write to a
    a = static_cast<T&&>(b);      // the assignment may write to b
    b = static_cast<T&&>(tmp);    // the assignment may write to tmp
}
```

The result value of **static_cast<T&&>(x)** is an rvalue of type **T&&** for **x**. An operation that is optimized for rvalues can now use its optimization for **x**. In particular, if a type **T** has a move constructor (§3.3.2, §17.5.2) or a move assignment, it will be used. Consider **vector**:

```
template<class T> class vector {
    // ...
    vector(const vector& r);  // copy constructor (copy r's representation)
    vector(vector&& r);       // move constructor ("steal" representation from r)
};

vector<string> s;
vector<string> s2 {s};              // s is an lvalue, so use copy constructor
vector<string> s3 {s+"tail"};       // s+"tail" is an rvalue so pick move constructor
```

The use of **static_cast** in **swap()** is a bit verbose and slightly prone to mistyping, so the standard library provides a **move()** function: **move(x)** means **static_cast<X&&>(x)** where **X** is the type of **x**. Given that, we can clean up the definition of **swap()** a bit:

```
template<class T>
void swap(T& a, T& b)     // "perfect swap" (almost)
{
    T tmp {move(a)};      // move from a
    a = move(b);          // move from b
    b = move(tmp);        // move from tmp
}
```

In contrast to the original **swap()**, this latest version need not make any copies; it will use move operations whenever possible.

Since **move(x)** does not move **x** (it simply produces an rvalue reference to **x**), it would have been better if **move()** had been called **rval()**, but by now **move()** has been used for years.

I deemed this **swap()** ''almost perfect'' because it will swap only lvalues. Consider:

```
void f(vector<int>& v)
{
    swap(v,vector<int>{1,2,3});     // replace v's elements with 1,2,3
    // ...
}
```

It is not uncommon to want to replace the contents of a container with some sort of default value, but this particular **swap()** cannot do that. A solution is to augment it by two overloads:

```
template<class T> void swap(T&& a, T& b);
template<class T> void swap(T& a, T&& b)
```

Our example will be handled by that last version of **swap()**. The standard library takes a different approach by defining **shrink_to_fit()** and **clear()** for **vector**, **string**, etc. (§31.3.3) to handle the most common cases of rvalue arguments to **swap()**:

```
void f(string& s, vector<int>& v)
{
    s.shrink_to_fit();          // make s.capacity()==s.size()
    swap(s,string{s});          // make s.capacity()==s.size()
```

```
    v.clear();                  // make v empty
    swap(v.vector<int>{});      // make v empty
    v = {};                     // make v empty
}
```

Rvalue references can also be used to provide perfect forwarding (§23.5.2.1, §35.5.1).

All standard-library containers provide move constructors and move assignment (§31.3.2). Also, their operations that insert new elements, such as **insert()** and **push_back()**, have versions that take rvalue references.

7.7.3 References to References

It you take a reference to a reference to a type, you get a reference to that type, rather than some kind of special reference to reference type. But what kind of reference? Lvalue reference or rvalue reference? Consider:

```
using rr_i = int&&;
using lr_i = int&;
using rr_rr_i = rr_i&&;     // "int && &&" is an int&&
using lr_rr_i = rr_i&;      // "int && &" is an int&
using rr_lr_i = lr_i&&;     // "int & &&" is an int&
using lr_lr_i = lr_i&;      // "int & &" is an int&
```

In other words, lvalue reference always wins. This makes sense: nothing we can do with types can change the fact that an lvalue reference refers to an lvalue. This is sometimes known as *reference collapse*.

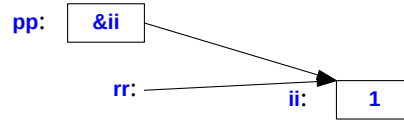
The syntax does not allow

```
int && & r = i;
```

Reference to reference can only happen as the result of an alias (§3.4.5, §6.5) or a template type argument (§23.5.2.1).

7.7.4 Pointers and References

Pointers and references are two mechanisms for referring to an object from different places in a program without copying. We can show this similarity graphically:



Each has its strengths and weaknesses.

If you need to change which object to refer to, use a pointer. You can use **=**, **+=**, **–=**, **++**, and **––** to change the value of a pointer variable (§11.1.4). For example:

```
void fp(char* p)
{
    while (*p)
        cout << ++*p;
}

void fr(char& r)
{
    while (r)
        cout << ++r;    // oops: increments the char referred to, not the reference
                        // near-infinite loop!
}

void fr2(char& r)
{
    char* p = &r;       // get a pointer to the object referred to
    while (*p)
        cout << ++*p;
}
```

Conversely, if you want to be sure that a name always refers to the same object, use a reference. For example:

```
template<class T> class Proxy {          // Proxy refers to the object with which it is initialized
    T& m;
public:
    Proxy(T& mm) :m{mm} {}
    // ...
};

template<class T> class Handle {    // Handle refers to its current object
    T* m;
public:
    Proxy(T* mm) :m{mm} {}
    void rebind(T* mm) { m = mm; }
    // ...
};
```

If you want to use a user-defined (overloaded) operator (§18.1) on something that refers to an object, use a reference:

```
Matrix operator+(const Matrix&, const Matrix&);    // OK
Matrix operator–(const Matrix*, const Matrix*);    // error: no user-defined type argument

Matrix y, z;
// ...
Matrix x = y+z;        // OK
Matrix x2 = &y–&z;     // error and ugly
```

It is not possible to (re)define an operator for a pair of built-in types, such as pointers (§18.2.3).

If you want a collection of something that refers to an object, you must use a pointer:

```
int x, y;
string& a1[] = {x, y};              // error: array of references
string* a2[] = {&x, &y};            // OK
vector<string&> s1 = {x , y};       // error: vector of references
vector<string*> s2 = {&x, &y};      // OK
```

Once we leave the cases where C++ leaves no choice for the programmer, we enter the domain of aesthetics. Ideally, we will make our choices so as to minimize the probability of error and in particular to maximize readability of code.

If you need a notion of "no value," pointers offer **nullptr**. There is no equivalent "null reference," so if you need a "no value," using a pointer may be most appropriate. For example:

```
void fp(X* p)
{
    if (p == nullptr) {
        // no value
    }
    else {
        // use *p
    }
}

void fr(X& r)    // common style
{
    // assume that r is valid and use it
}
```

If you really want to, you can construct and check for a "null reference" for a particular type:

```
void fr2(X& r)
{
    if (&r == &nullX) {    // or maybe r==nullX
        // no value
    }
    else {
        // use r
    }
}
```

Obviously, you need to have suitably defined **nullX**. The style is not idiomatic and I don't recommend it. A programmer is allowed to assume that a reference is valid. It is possible to create an invalid reference, but you have to go out of your way to do so. For example:

```
char* ident(char * p) { return p; }

char& r {*ident(nullptr)}; // invalid code
```

This code is not valid C++ code. Don't write such code even if your current implementation doesn't catch it.

7.8 Advice

[1] Keep use of pointers simple and straightforward; §7.4.1.
[2] Avoid nontrivial pointer arithmetic; §7.4.
[3] Take care not to write beyond the bounds of an array; §7.4.1.
[4] Avoid multidimensional arrays; define suitable containers instead; §7.4.2.
[5] Use **nullptr** rather than **0** or **NULL**; §7.2.2.
[6] Use containers (e.g., **vector**, **array**, and **valarray**) rather than built-in (C-style) arrays; §7.4.1.
[7] Use **string** rather than zero-terminated arrays of **char**; §7.4.
[8] Use raw strings for string literals with complicated uses of backslash; §7.3.2.1.
[9] Prefer **const** reference arguments to plain reference arguments; §7.7.3.
[10] Use rvalue references (only) for forwarding and move semantics; §7.7.2.
[11] Keep pointers that represent ownership inside handle classes; §7.6.
[12] Avoid **void*** except in low-level code; §7.2.1.
[13] Use **const** pointers and **const** references to express immutability in interfaces; §7.5.
[14] Prefer references to pointers as arguments, except where ‘‘no object’’ is a reasonable option; §7.7.4.

This page intentionally left blank

8

Structures, Unions, and Enumerations

Form a more perfect Union.
– The people

- Introduction
- Structures
  struct Layout; **struct** Names; Structures and Classes; Structures and Arrays; Type Equivalence; Plain Old Data; Fields
- Unions
  Unions and Classes; Anonymous **union**s
- Enumerations
  enum classes; Plain **enum**s; Unnamed **enum**s
- Advice

8.1 Introduction

The key to effective use of C++ is the definition and use of user-defined types. This chapter introduces the three most primitive variants of the notion of a user-defined type:

- A **struct** (a structure) is a sequence of elements (called *members*) of arbitrary types.
- A **union** is a **struct** that holds the value of just one of its elements at any one time.
- An **enum** (an enumeration) is a type with a set of named constants (called enumerators).
- **enum class** (a scoped enumeration) is an **enum** where the enumerators are within the scope of the enumeration and no implicit conversions to other types are provided.

Variants of these kinds of simple types have existed since the earliest days of C++. They are primarily focused on the representation of data and are the backbone of most C-style programming. The notion of a **struct** as described here is a simple form of a **class** (§3.2, Chapter 16).

8.2 Structures

An array is an aggregate of elements of the same type. In its simplest form, a **struct** is an aggregate of elements of arbitrary types. For example:

```
struct Address {
        const char* name;       // "Jim Dandy"
        int number;             // 61
        const char* street;     // "South St"
        const char* town;       // "New Providence"
        char state[2];          // 'N' 'J'
        const char* zip;        // "07974"
};
```

This defines a type called **Address** consisting of the items you need in order to send mail to someone within the USA. Note the terminating semicolon.

Variables of type **Address** can be declared exactly like other variables, and the individual *members* can be accessed using the **.** (dot) operator. For example:

```
void f()
{
        Address jd;
        jd.name = "Jim Dandy";
        jd.number = 61;
}
```

Variables of **struct** types can be initialized using the **{}** notation (§6.3.5). For example:

```
Address jd = {
        "Jim Dandy",
        61, "South St",
        "New Providence",
        {'N','J'}, "07974"
};
```

Note that **jd.state** could not be initialized by the string **"NJ"**. Strings are terminated by a zero character, **'\0'**, so **"NJ"** has three characters – one more than will fit into **jd.state**. I deliberately use rather low-level types for the members to illustrate how that can be done and what kinds of problems it can cause.

Structures are often accessed through pointers using the **–>** (**struct** pointer dereference) operator. For example:

```
void print_addr(Address* p)
{
        cout << p–>name << '\n'
                << p–>number << ' ' << p–>street << '\n'
                << p–>town << '\n'
                << p–>state[0] << p–>state[1] << ' ' << p–>zip << '\n';
}
```

When **p** is a pointer, **p–>m** is equivalent to **(*p).m**.

Alternatively, a **struct** can be passed by reference and accessed using the **.** (**struct** member access) operator:

```
void print_addr2(const Address& r)
{
        cout << r.name << '\n'
                << r.number << ' ' << r.street << '\n'
                << r.town << '\n'
                << r.state[0] << r.state[1] << ' ' << r.zip << '\n';
}
```

Argument passing is discussed in §12.2.

Objects of structure types can be assigned, passed as function arguments, and returned as the result from a function. For example:

```
Address current;

Address set_current(Address next)
{
        address prev = current;
        current = next;
        return prev;
}
```

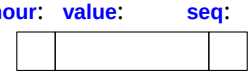
Other plausible operations, such as comparison (**==** and **!=**), are not available by default. However, the user can define such operators (§3.2.1.1, Chapter 18).

8.2.1 **struct** Layout

An object of a **struct** holds its members in the order they are declared. For example, we might store primitive equipment readout in a structure like this:

```
struct Readout {
        char hour;      // [0:23]
        int value;
        char seq;       // sequence mark ['a':'z']
};
```

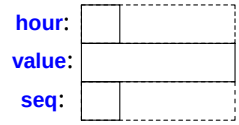
You could imagine the members of a **Readout** object laid out in memory like this:



Members are allocated in memory in declaration order, so the address of **hour** must be less than the address of **value**. See also §8.2.6.

However, the size of an object of a **struct** is not necessarily the sum of the sizes of its members. This is because many machines require objects of certain types to be allocated on architecture-dependent boundaries or handle such objects much more efficiently if they are. For example, integers are often allocated on word boundaries. On such machines, objects are said to have to be properly *aligned* (§6.2.9). This leads to ‘‘holes’’ in the structures. A more realistic layout of a

Readout on a machine with 4-byte **int** would be:

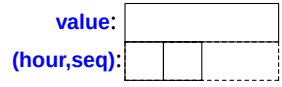


In this case, as on many machines, **sizeof(Readout)** is **12**, and not **6** as one would naively expect from simply adding the sizes of the individual members.

You can minimize wasted space by simply ordering members by size (largest member first). For example:

```
struct Readout {
        int value;
        char hour;      // [0:23]
        char seq;       // sequence mark ['a':'z']
};
```

This would give us:



Note that this still leaves a 2-byte ''hole'' (unused space) in a **Readout** and **sizeof(Readout)==8**. The reason is that we need to maintain alignment when we put two objects next to each other, say, in an array of **Readout**s. The size of an array of 10 **Readout** objects is **10*sizeof(Readout)**.

It is usually best to order members for readability and sort them by size only if there is a demonstrated need to optimize.

Use of multiple access specifiers (i.e., **public**, **private**, or **protected**) can affect layout (§20.5).

8.2.2 struct Names

The name of a type becomes available for use immediately after it has been encountered and not just after the complete declaration has been seen. For example:

```
struct Link {
        Link* previous;
        Link* successor;
};
```

However, it is not possible to declare new objects of a **struct** until its complete declaration has been seen. For example:

```
struct No_good {
        No_good member;  // error: recursive definition
};
```

This is an error because the compiler is not able to determine the size of **No_good**. To allow two (or

more) **struct**s to refer to each other, we can declare a name to be the name of a **struct**. For example:

```
struct List;            // struct name declaration: List to be defined later

struct Link {
        Link* pre;
        Link* suc;
        List* member_of;
        int data;
};

struct List {
        Link* head;
};
```

Without the first declaration of **List**, use of the pointer type **List*** in the declaration of **Link** would have been a syntax error.

The name of a **struct** can be used before the type is defined as long as that use does not require the name of a member or the size of the structure to be known. However, until the completion of the declaration of a **struct**, that **struct** is an incomplete type. For example:

```
struct S;  // "S" is the name of some type

extern S a;
S f();
void g(S);
S* h(S*);
```

However, many such declarations cannot be used unless the type **S** is defined:

```
void k(S* p)
{
        S a;            // error: S not defined; size needed to allocate

        f();            // error: S not defined; size needed to return value
        g(a);           // error: S not defined; size needed to pass argument
        p->m = 7;       // error: S not defined; member name not known

        S* q = h(p);    // ok: pointers can be allocated and passed
        q->m = 7;       // error: S not defined; member name not known
}
```

For reasons that reach into the prehistory of C, it is possible to declare a **struct** and a non-**struct** with the same name in the same scope. For example:

```
struct stat { /* ... */ };
int stat(char* name, struct stat* buf);
```

In that case, the plain name (**stat**) is the name of the non-**struct**, and the **struct** must be referred to with the prefix **struct**. Similarly, the keywords **class**, **union** (§8.3), and **enum** (§8.4) can be used as prefixes for disambiguation. However, it is best not to overload names to make such explicit disambiguation necessary.

8.2.3 Structures and Classes

A **struct** is simply a **class** where the members are **public** by default. So, a **struct** can have member functions (§2.3.2, Chapter 16). In particular, a **struct** can have constructors. For example:

```
struct Points {
    vector<Point> elem;// must contain at least one Point
    Points(Point p0) { elem.push_back(p0);}
    Points(Point p0, Point p1) { elem.push_back(p0); elem.push_back(p1); }
    // ...
};

Points x0;                          // error: no default constructor
Points x1{ {100,200} };             // one Point
Points x1{ {100,200}, {300,400} };  // two Points
```

You do not need to define a constructor simply to initialize members in order. For example:

```
struct Point {
    int x, y;
};

Point p0;       // danger: uninitialized if in local scope (§6.3.5.1)
Point p1 {};    // default construction: {{},{}}; that is {0.0}
Point p2 {1};   // the second member is default constructed: {1,{}}; that is {1,0}
Point p3 {1,2}; // {1,2}
```

Constructors are needed if you need to reorder arguments, validate arguments, modify arguments, establish invariants (§2.4.3.2, §13.4), etc. For example:

```
struct Address {
    string name;        // "Jim Dandy"
    int number;         // 61
    string street;      // "South St"
    string town;        // "New Providence"
    char state[2];      // 'N' 'J'
    char zip[5];        // 07974

    Address(const string n, int nu, const string& s, const string& t, const string& st, int z);
};
```

Here, I added a constructor to ensure that every member was initialized and to allow me to use a **string** and an **int** for the postal code, rather than fiddling with individual characters. For example:

```
Address jd = {
    "Jim Dandy",
    61, "South St",
    "New Providence",
    "NJ", 7974              // (07974 would be octal; §6.2.4.1)
};
```

The **Address** constructor might be defined like this:

```
Address::Address(const string& n, int nu, const string& s, const string& t, const string& st, int z)
                        // validate postal code
    :name{n},
     number{nu},
     street{s},
     town{t}
{
    if (st.size()!=2)
        error("State abbreviation should be two characters")
    state = {st[0],st[1]};      // store postal code as characters
    ostringstream ost;          // an output string stream; see §38.4.2
    ost << z;                   // extract characters from int
    string zi {ost.str()};
    switch (zi.size()) {
    case 5:
        zip = {zi[0], zi[1], zi[2], zi[3], zi[4]};
        break;
    case 4:    // starts with '0'
        zip = {'0', zi[0], zi[1], zi[2], zi[3]};
        break;
    default:
        error("unexpected ZIP code format");
    }
    // ... check that the code makes sense ...
}
```

8.2.4 Structures and Arrays

Naturally, we can have arrays of **struct**s and **struct**s containing arrays. For example:

```
struct Point {
    int x,y
};

Point points[3] {{1,2},{3,4},{5,6}};
int x2 = points[2].x;

struct Array {
    Point elem[3];
};

Array points2 {{1,2},{3,4},{5,6}};
int y2 = points2.elem[2].y;
```

Placing a built-in array in a **struct** allows us to treat that array as an object: we can copy the **struct** containing it in initialization (including argument passing and function return) and assignment. For example:

```
Array shift(Array a, Point p)
{
        for (int i=0; i!=3; ++i) {
                a.elem[i].x += p.x;
                a.elem[i].y += p.y;
        }
        return a;
}

Array ax = shift(points2,{10,20});
```

The notation for **Array** is a bit primitive: Why **i!=3**? Why keep repeating **.elem[i]**? Why just elements of type **Point**? The standard library provides **std::array** (§34.2.1) as a more complete and elegant development of the idea of a fixed-size array as a **struct**:

```
template<typename T, size_t N >
struct array {    // simplified (see §34.2.1)
        T elem[N];

        T* begin() noexcept { return elem; }
        const T* begin() const noexcept {return elem; }
        T* end() noexcept { return elem+N; }
        const T* end() const noexcept { return elem+N; }

        constexpr size_t size() noexcept;

        T& operator[](size_t n) { return elem[n]; }
        const T& operator[](size_type n) const { return elem[n]; }

        T * data() noexcept { return elem; }
        const T * data() const noexcept { return elem; }

        // ...
};
```

This **array** is a template to allow arbitrary numbers of elements of arbitrary types. It also deals directly with the possibility of exceptions (§13.5.1.1) and **const** objects (§16.2.9.1). Using **array**, we can now write:

```
struct Point {
        int x,y
};

using Array = array<Point,3>; // array of 3 Points

Array points {{1,2},{3,4},{5,6}};
int x2 = points[2].x;
int y2 = points[2].y;
```

```
Array shift(Array a, Point p)
{
        for (int i=0; i!=a.size(); ++i) {
                a[i].x += p.x;
                a[i].y += p.y;
        }
        return a;
}

Array ax = shift(points,{10,20});
```

The main advantages of **std::array** over a built-in array are that it is a proper object type (has assignment, etc.) and does not implicitly convert to a pointer to an individual element:

```
ostream& operator<<(ostream& os, Point p)
{
        cout << '{' << p[i].x << ',' << p[i].y << '}';
}

void print(Point a[],int s)  // must specify number of elements
{
        for (int i=0; i!=s; ++i)
                cout << a[i] << '\n';
}

template<typename T, int N>
void print(array<T,N>& a)
{
        for (int i=0; i!=a.size(); ++i)
                cout << a[i] << '\n';
}

Point point1[] = {{1,2},{3,4},{5,6}};               // 3 elements
array<Point,3> point2 = {{1,2},{3,4},{5,6}};        // 3 elements

void f()
{
        print(point1,4);        // 4 is a bad error
        print(point2);
}
```

The disadvantage of **std::array** compared to a built-in array is that we can't deduce the number of elements from the length of the initializer:

```
Point point1[] = {{1,2},{3,4},{5,6}};               // 3 elements
array<Point,3> point2 = {{1,2},{3,4},{5,6}};        // 3 elements
array<Point> point3 = {{1,2},{3,4},{5,6}};          // error: number of elements not given
```

8.2.5 Type Equivalence

Two **struct**s are different types even when they have the same members. For example:

```
struct S1 { int a; };
struct S2 { int a; };
```

S1 and **S2** are two different types, so:

```
S1 x;
S2 y = x;  // error: type mismatch
```

A **struct** is also a different type from a type used as a member. For example:

```
S1 x;
int i = x;  // error: type mismatch
```

Every **struct** must have a unique definition in a program (§15.2.3).

8.2.6 Plain Old Data

Sometimes, we want to treat an object as just ''plain old data'' (a contiguous sequence of bytes in memory) and not worry about more advanced semantic notions, such as run-time polymorphism (§3.2.3, §20.3.2), user-defined copy semantics (§3.3, §17.5), etc. Often, the reason for doing so is to be able to move objects around in the most efficient way the hardware is capable of. For example, copying a 100-element array using 100 calls of a copy constructor is unlikely to be as fast as calling **std::memcpy()**, which typically simply uses a block-move machine instruction. Even if the constructor is inlined, it could be hard for an optimizer to discover this optimization. Such ''tricks'' are not uncommon, and are important, in implementations of containers, such as **vector**, and in low-level I/O routines. They are unnecessary and should be avoided in higher-level code.

So, a *POD* (''Plain Old Data'') is an object that can be manipulated as ''just data'' without worrying about complications of class layouts or user-defined semantics for construction, copy, and move. For example:

```
struct S0 { };                                  // a POD
struct S1 { int a; };                           // a POD
struct S2 { int a; S2(int aa) : a(aa) { } };    // not a POD (no default constructor)
struct S3 { int a; S3(int aa) : a(aa) { } S3() {} };    // a POD (user-defined default constructor)
struct S4 { int a; S4(int aa) : a(aa) { } S4() = default; };    // a POD
struct S5 { virtual void f(); /* ... */ };      // not a POD (has a virtual function)

struct S6 : S1 { };             // a POD
struct S7 : S0 { int b; };      // a POD
struct S8 : S1 { int b; };      // not a POD (data in both S1 and S8)
struct S9 : S0, S1 {};          // a POD
```

For us to manipulate an object as ''just data'' (as a POD), the object must

- not have a complicated layout (e.g., with a **vptr**; (§3.2.3, §20.3.2),
- not have nonstandard (user-defined) copy semantics, and
- have a trivial default constructor.

Obviously, we need to be precise about the definition of POD so that we only use such

optimizations where they don't break any language guarantees. Formally (§iso.3.9, §iso.9), a POD object must be of

- a *standard layout type*, and
- a *trivially copyable type*,
- a type with a trivial default constructor.

A related concept is a *trivial type*, which is a type with

- a trivial default constructor and
- trivial copy and move operations

Informally, a default constructor is trivial if it does not need to do any work (use **=default** if you need to define one §17.6.1).

A type has standard layout unless it

- has a non-**static** member or a base that is not standard layout,
- has a **virtual** function (§3.2.3, §20.3.2),
- has a **virtual** base (§21.3.5),
- has a member that is a reference (§7.7),
- has multiple access specifiers for non-static data members (§20.5), or
- prevents important layout optimizations
  - by having non-**static** data members in more than one base class or in both the derived class and a base, or
  - by having a base class of the same type as the first non-**static** data member.

Basically, a standard layout type is one that has a layout with an obvious equivalent in C and is in the union of what common C++ Application Binary Interfaces (ABIs) can handle.

A type is trivially copyable unless it has a nontrivial copy operation, move operation, or destructor (§3.2.1.2, §17.6). Informally, a copy operation is trivial if it can be implemented as a bitwise copy. So, what makes a copy, move, or destructor nontrivial?

- It is user-defined.
- Its class has a **virtual** function.
- Its class has a **virtual** base.
- Its class has a base or a member that is not trivial.

An object of built-in type is trivially copyable, and has standard layout. Also, an array of trivially copyable objects is trivially copyable and an array of standard layout objects has standard layout. Consider an example:

```
template<typename T>
    void mycopy(T* to, const T* from, int count);
```

I'd like to optimize the simple case where **T** is a POD. I could do that by only calling **mycopy()** for PODs, but that's error-prone: if I use **mycopy()** can I rely on a maintainer of the code to remember never to call **mycopy()** for non-PODs? Realistically, I cannot. Alternatively, I could call **std::copy()**, which is most likely implemented with the necessary optimization. Anyway, here is the general and optimized code:

```
template<typename T>
void mycopy(T* to, const T* from, int count)
{
        if (is_pod<T>::value)
                memcpy(to,from,count*sizeof(T));
        else
                for (int i=0; i!=count; ++i)
                        to[i]=from[i];
}
```

The `is_pod` is a standard-library type property predicate (§35.4.1) defined in `<type_traits>` allowing us to ask the question ''Is `T` a POD?'' in our code. The best thing about `is_pod<T>` is that it saves us from remembering the exact rules for what a POD is.

Note that adding or subtracting non-default constructors does not affect layout or performance (that was not true in C++98).

If you feel an urge to become a language lawyer, study the layout and triviality concepts in the standard (§iso.3.9, §iso.9) and try to think about their implications to programmers and compiler writers. Doing so might cure you of the urge before it has consumed too much of your time.

8.2.7 Fields

It seems extravagant to use a whole byte (a `char` or a `bool`) to represent a binary variable – for example, an on/off switch – but a `char` is the smallest object that can be independently allocated and addressed in C++ (§7.2). It is possible, however, to bundle several such tiny variables together as *fields* in a `struct`. A field is often called a *bit-field*. A member is defined to be a field by specifying the number of bits it is to occupy. Unnamed fields are allowed. They do not affect the meaning of the named fields, but they can be used to make the layout better in some machine-dependent way:

```
struct PPN {            // R6000 Physical Page Number
        unsigned int PFN : 22;    // Page Frame Number
        int : 3;                  // unused
        unsigned int CCA : 3;           // Cache Coherency Algorithm
        bool nonreachable : 1;
        bool dirty : 1;
        bool valid : 1;
        bool global : 1;
};
```

This example also illustrates the other main use of fields: to name parts of an externally imposed layout. A field must be of an integral or enumeration type (§6.2.1). It is not possible to take the address of a field. Apart from that, however, it can be used exactly like other variables. Note that a `bool` field really can be represented by a single bit. In an operating system kernel or in a debugger, the type `PPN` might be used like this:

```
void part_of_VM_system(PPN* p)
{
        // ...
```

```
        if (p–>dirty) {   // contents changed
            // copy to disk
            p–>dirty = 0;
        }
}
```

Surprisingly, using fields to pack several variables into a single byte does not necessarily save space. It saves data space, but the size of the code needed to manipulate these variables increases on most machines. Programs have been known to shrink significantly when binary variables were converted from bit-fields to characters! Furthermore, it is typically much faster to access a **char** or an **int** than to access a field. Fields are simply a convenient shorthand for using bitwise logical operators (§11.1.1) to extract information from and insert information into part of a word.

8.3 Unions

A **union** is a **struct** in which all members are allocated at the same address so that the **union** occupies only as much space as its largest member. Naturally, a **union** can hold a value for only one member at a time. For example, consider a symbol table entry that holds a name and a value:

```
enum Type { str, num };

struct Entry {
        char* name;
        Type t;
        char* s;    // use s if t==str
        int i;      // use i if t==num
};

void f(Entry* p)
{
        if (p–>t == str)
            cout << p–>s;
        // ...
}
```

The members **s** and **i** can never be used at the same time, so space is wasted. It can be easily recovered by specifying that both should be members of a **union**, like this:

```
union Value {
        char* s;
        int i;
};
```

The language doesn't keep track of which kind of value is held by a **union**, so the programmer must do that:

```
struct Entry {
        char* name;
        Type t;
        Value v;    // use v.s if t==str; use v.i if t==num
};

void f(Entry* p)
{
        if (p->t == str)
                cout << p->v.s;
        // ...
}
```

To avoid errors, one can encapsulate a **union** so that the correspondence between a type field and access to the **union** members can be guaranteed (§8.3.2).

Unions are sometimes misused for ''type conversion.'' This misuse is practiced mainly by programmers trained in languages that do not have explicit type conversion facilities, so that cheating is necessary. For example, the following ''converts'' an **int** to an **int*** simply by assuming bitwise equivalence:

```
union Fudge {
        int i;
        int* p;
};

int* cheat(int i)
{
        Fudge a;
        a.i = i;
        return a.p;        // bad use
}
```

This is not really a conversion at all. On some machines, an **int** and an **int*** do not occupy the same amount of space, while on others, no integer can have an odd address. Such use of a **union** is dangerous and nonportable. If you need such an inherently ugly conversion, use an explicit type conversion operator (§11.5.2) so that the reader can see what is going on. For example:

```
int* cheat2(int i)
{
        return reinterpret_cast<int*>(i);      // obviously ugly and dangerous
}
```

Here, at least the compiler has a chance to warn you if the sizes of objects are different and such code stands out like the sore thumb it is.

Use of **union**s can be essential for compactness of data and through that for performance. However, most programs don't improve much from the use of **union**s and **union**s are rather error-prone. Consequently, I consider **union**s an overused feature; avoid them when you can.

8.3.1 Unions and Classes

Many nontrivial **union**s have a member that is much larger than the most frequently used members. Because the size of a **union** is at least as large as its largest member, space is wasted. This waste can often be eliminated by using a set of derived classes (§3.2.2, Chapter 20) instead of a **union**.

Technically, a **union** is a kind of a **struct** (§8.2) which in turn is a kind of a **class** (Chapter 16). However, many of the facilities provided for classes are not relevant for unions, so some restrictions are imposed on **union**s:

[1] A **union** cannot have virtual functions.
[2] A **union** cannot have members of reference type.
[3] A **union** cannot have base classes.
[4] If a **union** has a member with a user-defined constructor, a copy operation, a move operation, or a destructor, then that special function is **delete**d (§3.3.4, §17.6.4) for that **union**; that is, it cannot be used for an object of the **union** type.
[5] At most one member of a **union** can have an in-class initializer (§17.4.4).
[6] A **union** cannot be used as a base class.

These restrictions prevent many subtle errors and simplify the implementation of **union**s. The latter is important because the use of **union**s is often an optimization and we won't want "hidden costs" imposed to compromise that.

The rule that **delete**s constructors (etc.) from a **union** with a member that has a constructor (etc.) keeps simple **union**s simple and forces the programmer to provide complicated operations if they are needed. For example, since **Entry** has no member with constructors, destructors, or assignments, we can create and copy **Entry**s freely. For example:

```
void f(Entry a)
{
    Entry b = a;
};
```

Doing so with a more complicated **union** would cause implementation difficulties or errors:

```
union U {
    int m1;
    complex<double> m2;    // complex has a constructor
    string m3;             // string has a constructor (maintaining a serious invariant)
};
```

To copy a **U** we would have to decide which copy operation to use. For example:

```
void f2(U x)
{
    U u;                   // error: which default constructor?
    U u2 = x;              // error: which copy constructor?
    u.m1 = 1;              // assign to int member
    string s = u.m3;       // disaster: read from string member
    return;                // error: which destructors are called for x, u, and u2?
}
```

It's illegal to write one member and then read another, but people do that nevertheless (usually by mistake). In this case, the **string** copy constructor would be called with an invalid argument. It is

fortunate that `U` won't compile. When needed, a user can define a class containing a `union` that properly handles `union` members with constructors, destructors, and assignments (§8.3.2). If desired, such a class can also prevent the error of writing one member and then reading another.

It is possible to specify an in-class initializer for at most one member. If so, this initializer will be used for default initialization. For example:

```
union U2 {
        int a;
        const char* p {""};
};

U2 x1;          // default initialized to x1.p == ""
U2 x2 {7};      // x2.a == 7
```

8.3.2 Anonymous unions

To see how we can write a class that overcomes the problems with misuse of a `union`, consider a variant of `Entry` (§8.3):

```
class Entry2 {  // two alternative representations represented as a union
private:
        enum class Tag { number, text };
        Tag type;  // discriminant

        union {    // representation
                int i;
                string s;   // string has default constructor, copy operations, and destructor
        };
public:
        struct Bad_entry { };       // used for exceptions

        string name;

        ~Entry2();
        Entry2& operator=(const Entry2&);       // necessary because of the string variant
        Entry2(const Entry2&);
        // ...

        int number() const;
        string text() const;

        void set_number(int n);
        void set_text(const string&);
        // ...
};
```

I'm not a fan of get/set functions, but in this case we really need to perform a nontrivial user-specified action on each access. I chose to name the "get" function after the value and use the `set_` prefix for the "set" function. That happens to be my favorite among the many naming conventions.

The read-access functions can be defined like this:

```
int Entry2::number() const
{
    if (type!=Tag::number) throw Bad_entry{};
    return i;
};

string Entry2::text() const
{
    if (type!=Tag::text) throw Bad_entry{};
    return s;
};
```

These access functions check the **type** tag, and if it is the one that correctly corresponds to the access we want, it returns a reference to the value; otherwise, it throws an exception. Such a **union** is often called a *tagged union* or a *discriminated union*.

The write-access functions basically do the same checking of the **type** tag, but note how setting a new value must take the previous value into account:

```
void Entry2::set_number(int n)
{
    if (type==Tag::text) {
        s.~string();                // explicitly destroy string (§11.2.4)
        type = Tag::number;
    }
    i = n;
}

void Entry2::set_text(const string& ss)
{
    if (type==Tag::text)
        s = ss;
    else {
        new(&s) string{ss};         // placement new: explicitly construct string (§11.2.4)
        type = Tag::text;
    }
}
```

The use of a **union** forces us to use otherwise obscure and low-level language facilities (explicit construction and destruction) to manage the lifetime of the **union** elements. This is another reason to be wary of using **unions**.

Note that the **union** in the declaration of **Entry2** is not named. That makes it an *anonymous union*. An anonymous **union** is an object, not a type, and its members can be accessed without mentioning an object name. That means that we can use members of an anonymous **union** exactly as we use other members of a class – as long as we remember that **union** members really can be used only one at a time.

Entry2 has a member of a type with a user-defined assignment operator, **string**, so **Entry2**'s assignment operator is **delete**d (§3.3.4, §17.6.4). If we want to assign **Entry2**s, we have to define

Entry2::operator=(). Assignment combines the complexities of reading and writing but is otherwise logically similar to the access functions:

```
Entry2& Entry2::operator=(const Entry2& e)  // necessary because of  the string variant
{
    if (type==Tag::text && e.type==Tag::text) {
        s = e.s;            // usual string assignment
        return *this;
    }

    if (type==Tag::text) s.~string(); // explicit destroy (§11.2.4)

    switch (e.type) {
    case Tag::number:
        i = e.i;
        break;
    case Tag::text:
        new(&s)(e.s);   // placement new: explicit construct (§11.2.4)
        type = e.type;
    }

    return *this;
}
```

Constructors and a move assignment can be defined similarly as needed. We need at least a constructor or two to establish the correspondence between the **type** tag and a value. The destructor must handle the **string** case:

```
Entry2::~Entry2()
{
    if (type==Tag::text) s.~string(); // explicit destroy (§11.2.4)
}
```

8.4 Enumerations

An *enumeration* is a type that can hold a set of integer values specified by the user (§iso.7.2). Some of an enumeration's possible values are named and called *enumerators*. For example:

```
enum class Color { red, green, blue };
```

This defines an enumeration called **Color** with the enumerators **red**, **green**, and **blue**. ''An enumeration'' is colloquially shortened to ''an **enum**.''

There are two kinds of enumerations:

[1] **enum class**es, for which the enumerator names (e.g., **red**) are local to the **enum** and their values do not implicitly convert to other types

[2] ''Plain **enum**s,'' for which the enumerator names are in the same scope as the **enum** and their values implicitly convert to integers

In general, prefer the **enum class**es because they cause fewer surprises.

8.4.1 enum classes

An **enum class** is a scoped and strongly typed enumeration. For example:

```
enum class Traffic_light { red, yellow, green };
enum class Warning { green, yellow, orange, red };// fire alert levels

Warning a1 = 7;                 // error: no int->Warning conversion
int a2 = green;                 // error: green not in scope
int a3 = Warning::green;        // error: no Warning->int conversion
Warning a4 = Warning::green;    // OK

void f(Traffic_light x)
{
    if (x == 9) { /* ... */ }                   // error: 9 is not a Traffic_light
    if (x == red) { /* ... */ }                 // error: no red in scope
    if (x == Warning::red)  { /* ... */ }       // error: x is not a Warning
    if (x == Traffic_light::red) { /* ... */ }  // OK
}
```

Note that the enumerators present in both **enum**s do not clash because each is in the scope of its own **enum class**.

An enumeration is represented by some integer type and each enumerator by some integer value. We call the type used to represent an enumeration its *underlying type*. The underlying type must be one of the signed or unsigned integer types (§6.2.4); the default is **int**. We could be explicit about that:

```
enum class Warning : int { green, yellow, orange, red }; // sizeof(Warning)==sizeof(int)
```

If we considered that too wasteful of space, we could instead use a **char**:

```
enum class Warning : char { green, yellow, orange, red };      // sizeof(Warning)==1
```

By default, enumerator values are assigned increasing from **0**. Here, we get:

```
static_cast<int>(Warning::green)==0
static_cast<int>(Warning::yellow)==1
static_cast<int>(Warning::orange)==2
static_cast<int>(Warning::red)==3
```

Declaring a variable **Warning** instead of plain **int** can give both the user and the compiler a hint as to the intended use. For example:

```
void f(Warning key)
{
    switch (key) {
    case Warning::green:
        // do something
        break;
    case Warning::orange:
        // do something
        break;
```

```
        case Warning::red:
            // do something
            break;
        }
}
```

A human might notice that `yellow` was missing, and a compiler might issue a warning because only three out of four `Warning` values are handled.

An enumerator can be initialized by a constant expression (§10.4) of integral type (§6.2.1). For example:

```
enum class Printer_flags {
        acknowledge=1,
        paper_empty=2,
        busy=4,
        out_of_black=8,
        out_of_color=16,
        //
};
```

The values for the `Printer_flags` enumerators are chosen so that they can be combined by bitwise operations. An `enum` is a user-defined type, so we can define the `|` and `&` operators for it (§3.2.1.1, Chapter 18). For example:

```
constexpr Printer_flags operator|(Printer_flags a, Printer_flags b)
{
        return static_cast<Printer_flags>(static_cast<int>(a))|static_cast<int>(b));
}

constexpr Printer_flags operator&(Printer_flags a, Printer_flags b)
{
        return static_cast<Printer_flags>(static_cast<int>(a))&static_cast<int>(b));
}
```

The explicit conversions are necessary because a `class enum` does not support implicit conversions. Given these definitions of `|` and `&` for `Printer_flags`, we can write:

```
void try_to_print(Printer_flags x)
{
        if (x&Printer_flags::acknowledge) {
            // ...
        }
        else if (x&Printer_flags::busy) {
            // ...
        }
        else if (x&(Printer_flags::out_of_black|Printer_flags::out_of_color)) {
            // either we are out of black or we are out of color
            // ...
        }
        // ...
}
```

I defined **operator|()** and **operator&()** to be **constexpr** functions (§10.4, §12.1.6) because someone might want to use those operators in constant expressions. For example:

```
void g(Printer_flags x)
{
    switch (x) {
    case Printer_flags::acknowledge:
        // ...
        break;
    case Printer_flags::busy:
        // ...
        break;
    case Printer_flags::out_of_black:
        // ...
        break;
    case Printer_flags::out_of_color:
        // ...
        break;
    case Printer_flags::out_of_black&Printer_flags::out_of_color:
        // we are out of black *and* out of color
        // ...
        break;
    }

    // ...
}
```

It is possible to declare an **enum class** without defining it (§6.3) until later. For example:

```
enum class Color_code : char;       // declaration
void foobar(Color_code* p);         // use of declaration
// ...
enum class Color_code : char {      // definition
    red, yellow, green, blue
};
```

A value of integral type may be explicitly converted to an enumeration type. The result of such a conversion is undefined unless the value is within the range of the enumeration's underlying type. For example:

```
enum class Flag : char{ x=1, y=2, z=4, e=8 };

Flag f0 {};                         // f0 gets the default value 0
Flag f1 = 5;                        // type error: 5 is not of type Flag
Flag f2 = Flag{5};                  // error: no narrowing conversion to an enum class
Flag f3 = static_cast<Flag>(5);     // brute force
Flag f4 = static_cast<Flag>(999);   // error: 999 is not a char value (maybe not caught)
```

The last assignments show why there is no implicit conversion from an integer to an enumeration; most integer values do not have a representation in a particular enumeration.

Each enumerator has an integer value. We can extract that value explicitly. For example:

```
int i = static_cast<int>(Flag::y);          // i becomes 2
char c = static_cast<char>(Flag::e);        // c becomes 8
```

The notion of a range of values for an enumeration differs from the enumeration notion in the Pascal family of languages. However, bit-manipulation examples that require values outside the set of enumerators to be well defined (e.g., the **Printer_flags** example) have a long history in C and C++.

The **sizeof** an **enum class** is the **sizeof** of its underlying type. In particular, if the underlying type is not explicitly specified, the size is **sizeof(int)**.

8.4.2 Plain enums

A ''plain **enum**'' is roughly what C++ offered before the **enum class**es were introduced, so you'll find them in lots of C and C++98-style code. The enumerators of a plain **enum** are exported into the **enum**'s scope, and they implicitly convert to values of some integer type. Consider the examples from §8.4.1 with the ''**class**'' removed:

```
enum Traffic_light { red, yellow, green };
enum Warning { green, yellow, orange, red }; // fire alert levels

// error: two definitions of yellow (to the same value)
// error: two definitions of red (to different values)

Warning a1 = 7;                 // error: no int->Warning conversion
int a2 = green;                 // OK: green is in scope and converts to int
int a3 = Warning::green;        // OK: Warning->int conversion
Warning a4 = Warning::green;    // OK

void f(Traffic_light x)
{
    if (x == 9) { /* ... */ }                   // OK (but Traffic_light doesn't have a 9)
    if (x == red) { /* ... */ }                 // error: two reds in scope
    if (x == Warning::red)  { /* ... */ }       // OK (Ouch!)
    if (x == Traffic_light::red) { /* ... */ }  // OK
}
```

We were ''lucky'' that defining **red** in two plain enumerations in a single scope saved us from hard-to-spot errors. Consider ''cleaning up'' the plain **enum**s by disambiguating the enumerators (as is easily done in a small program but can be done only with great difficulty in a large one):

```
enum Traffic_light { tl_red, tl_yellow, tl_green };
enum Warning { green, yellow, orange, red }; // fire alert levels
```

```
void f(Traffic_light x)
{
    if (x == red) { /* ... */ }                    // OK (ouch!)
    if (x == Warning::red)  { /* ... */ }          // OK (ouch!)
    if (x == Traffic_light::red) { /* ... */ }     // error: red is not a Traffic_light value
}
```

The compiler accepts the `x==red`, which is almost certainly a bug. The injection of names into an enclosing scope (as `enum`s, but not `enum class`es or `class`es, do) is *namespace pollution* and can be a major problem in larger programs (Chapter 14).

You can specify the underlying type of a plain enumeration, just as you can for `enum class`es. If you do, you can declare the enumerations without defining them until later. For example:

```
enum Traffic_light : char { tl_red, tl_yellow, tl_green };   // underlying type is char

enum Color_code : char;       // declaration
void foobar(Color_code* p);   // use of declaration
// ...
enum Color_code : char { red, yellow, green, blue }; // definition
```

If you don't specify the underlying type, you can't declare the `enum` without defining it, and its underlying type is determined by a relatively complicated algorithm: when all enumerators are non-negative, the range of the enumeration is $[0{:}2^k{-}1]$ where 2^k is the smallest power of 2 for which all enumerators are within the range. If there are negative enumerators, the range is $[-2^k{:}2^k{-}1]$. This defines the smallest bit-field capable of holding the enumerator values using the conventional two's complement representation. For example:

```
enum E1 { dark, light };                    // range 0:1
enum E2 { a = 3, b = 9 };                   // range 0:15
enum E3 { min = –10, max = 1000000 };       // range -1048576:1048575
```

The rule for explicit conversion of an integer to a plain `enum` is the same as for the `class enum` except that when there is no explicit underlying type, the result of such a conversion is undefined unless the value is within the range of the enumeration. For example:

```
enum Flag { x=1, y=2, z=4, e=8 };    // range 0:15

Flag f0 {};                          // f0 gets the default value 0
Flag f1 = 5;                         // type error: 5 is not of type Flag
Flag f2 = Flag{5};                   // error: no explicit conversion from int to Flag
Flag f2 = static_cast<Flag>(5);      // OK: 5 is within the range of Flag
Flag f3 = static_cast<Flag>(z|e);    // OK: 12 is within the range of Flag
Flag f4 = static_cast<Flag>(99);     // undefined: 99 is not within the range of Flag
```

Because there is an implicit conversion from a plain `enum` to its underlying type, we don't need to define `|` to make this example work: `z` and `e` are converted to `int` so that `z|e` can be evaluated. The `sizeof` an enumeration is the `sizeof` its underlying type. If the underlying type isn't explicitly specified, it is some integral type that can hold its range and not larger than `sizeof(int)`, unless an enumerator cannot be represented as an `int` or as an `unsigned int`. For example, `sizeof(e1)` could be `1` or maybe `4` but not `8` on a machine where `sizeof(int)==4`.

8.4.3 Unnamed enums

A plain **enum** can be unnamed. For example:

```
enum { arrow_up=1, arrow_down, arrow_sideways };
```

We use that when all we need is a set of integer constants, rather than a type to use for variables.

8.5 Advice

[1] When compactness of data is important, lay out structure data members with larger members before smaller ones; §8.2.1.
[2] Use bit-fields to represent hardware-imposed data layouts; §8.2.7.
[3] Don't naively try to optimize memory consumption by packing several values into a single byte; §8.2.7.
[4] Use **union**s to save space (represent alternatives) and never for type conversion; §8.3.
[5] Use enumerations to represent sets of named constants; §8.4.
[6] Prefer **class enum**s over "plain" **enum**s to minimize surprises; §8.4.
[7] Define operations on enumerations for safe and simple use; §8.4.1.

9

Statements

A programmer is a machine
for turning caffeine into code.
– A programmer

- Introduction
- Statement Summary
- Declarations as Statements
- Selection Statements
  if Statements; **switch** Statements; Declarations in Conditions
- Iteration Statements
  Range-**for** Statements; **for** Statements; **while** Statements; **do** Statements; Loop exit
- **goto** Statements
- Comments and Indentation
- Advice

9.1 Introduction

C++ offers a conventional and flexible set of statements. Basically all that is either interesting or complicated is found in expressions and declarations. Note that a declaration is a statement and that an expression becomes a statement when you add a semicolon at its end.

Unlike an expression, a statement does not have a value. Instead, statements are used to specify the order of execution. For example:

```
a = b+c;        // expression statement
if (a==7)       // if-statement
    b = 9;      // execute if and only if a==9
```

Logically, **a=b+c** is executed before the **if**, as everyone would expect. A compiler may reorder code to improve performance as long as the result is identical to that of the simple order of execution.

9.2 Statement Summary

Here is a summary of C++ statements:

statement:
declaration
expression$_{opt}$ **;**
{ *statement-list*$_{opt}$ **}**
try { *statement-list*$_{opt}$ **}** *handler-list*

case *constant-expression* **:** *statement*
default : *statement*
break ;
continue ;

return *expression*$_{opt}$ **;**

goto *identifier* **;**
identifier **:** *statement*

selection-statement
iteration-statement

selection-statement:
if (*condition* **)** *statement*
if (*condition* **)** *statement* **else** *statement*
switch (*condition* **)** *statement*

iteration-statement:
while (*condition* **)** *statement*
do *statement* **while (** *expression* **) ;**
for (*for-init-statement condition*$_{opt}$ **;** *expression*$_{opt}$ **)** *statement*
for (*for-init-declaration* **:** *expression* **)** *statement*

statement-list:
statement statement-list$_{opt}$

condition:
expression
type-specifier declarator **=** *expression*
type-specifier declarator **{** *expression* **}**

handler-list:
handler handler-list$_{opt}$

handler:
catch (*exception-declaration* **) {** *statement-list*$_{opt}$ **}**

A semicolon is by itself a statement, the *empty statement*.

A (possibly empty) sequence of statements within ''curly braces'' (i.e., { and }) is called a *block* or a *compound statement*. A name declared in a block goes out of scope at the end of its block (§6.3.4).

A *declaration* is a statement and there is no assignment statement or procedure-call statement; assignments and function calls are expressions.

A *for-init-statement* must be either a declaration or an *expression-statement*. Note that both end with a semicolon.

A *for-init-declaration* must be the declaration of a single uninitialized variable.

The statements for handling exceptions, *try-block*s, are described in §13.5.

9.3 Declarations as Statements

A declaration is a statement. Unless a variable is declared **static**, its initializer is executed whenever the thread of control passes through the declaration (see also §6.4.2). The reason for allowing declarations wherever a statement can be used (and a few other places; §9.4.3, §9.5.2) is to enable the programmer to minimize the errors caused by uninitialized variables and to allow better locality in code. There is rarely a reason to introduce a variable before there is a value for it to hold. For example:

```
void f(vector<string>& v, int i, const char* p)
{
        if (p==nullptr) return;
        if (i<0 || v.size()<=i)
                error("bad index");
        string s = v[i];
        if (s == p) {
                // ...
        }
        // ...
}
```

The ability to place declarations after executable code is essential for many constants and for single-assignment styles of programming where a value of an object is not changed after initialization. For user-defined types, postponing the definition of a variable until a suitable initializer is available can also lead to better performance. For example:

```
void use()
{
        string s1;
        s1 = "The best is the enemy of the good.";
        // ...
}
```

This requests a default initialization (to the empty string) followed by an assignment. This can be slower than a simple initialization to the desired value:

```
        string s2 {"Voltaire"};
```

The most common reason to declare a variable without an initializer is that it requires a statement

to give it its desired value. Input variables are among the few reasonable examples of that:

```
void input()
{
        int buf[max];
        int count = 0;
        for (int i; cin>>i;) {
                if (i<0) error("unexpected negative value");
                if (count==max) error("buffer overflow");
                buf[count++] = i;
        }
        // ...
}
```

I assume that `error()` does not return; if it does, this code may cause a buffer overflow. Often, `push_back()` (§3.2.1.3, §13.6, §31.3.6) provides a better solution to such examples.

9.4 Selection Statements

A value can be tested by either an `if`-statement or a `switch`-statement:

```
if ( condition ) statement
if ( condition ) statement else statement
switch ( condition ) statement
```

A *condition* is either an expression or a declaration (§9.4.3).

9.4.1 `if` Statements

In an `if`-statement, the first (or only) statement is executed if the condition is `true` and the second statement (if it is specified) is executed otherwise. If a condition evaluates to something different from a Boolean, it is – if possible – implicitly converted to a `bool`. This implies that any arithmetic or pointer expression can be used as a condition. For example, if `x` is an integer, then

```
if (x) // ...
```

means

```
if (x != 0) // ...
```

For a pointer `p`,

```
if (p) // ...
```

is a direct statement of the test ‘‘Does `p` point to a valid object (assuming proper initialization)?’’ and is equivalent to

```
if (p != nullptr) // ...
```

Note that a ‘‘plain’’ `enum` can be implicitly converted to an integer and then to a `bool`, whereas an `enum class` cannot (§8.4.1). For example:

```
enum E1 { a, b };
enum class E2 { a, b };

void f(E1 x, E2 y)
{
    if (x)              // OK
        // ...
    if (y)              // error: no conversion to bool
        // ...
    if (y==E2::a)       // OK
        // ...
}
```

The logical operators

```
&&   ||   !
```

are most commonly used in conditions. The operators `&&` and `||` will not evaluate their second argument unless doing so is necessary. For example,

```
if (p && 1<p–>count) // ...
```

This tests `1<p–>count` only if `p` is not `nullptr`.

For choosing between two alternatives each of which produces a value, a conditional expression (§11.1.3) is a more direct expression of intent than an `if`-statement. For example:

```
int max(int a, int b)
{
    return (a>b)?a:b;    // return the larger of a and b
}
```

A name can only be used within the scope in which it is declared. In particular, it cannot be used on another branch of an `if`-statement. For example:

```
void f2(int i)
{
    if (i) {
        int x = i+2;
        ++x;
        // ...
    }
    else {
        ++x; // error: x is not in scope
    }
    ++x;        // error: x is not in scope
}
```

A branch of an `if`-statement cannot be just a declaration. If we need to introduce a name in a branch, it must be enclosed in a block (§9.2). For example:

```
void f1(int i)
{
        if (i)
                int x = i+2;            // error: declaration of if-statement branch
}
```

9.4.2 switch Statements

A **switch**-statement selects among a set of alternatives (**case**-labels). The expression in the **case** labels must be a constant expression of integral or enumeration type. A value may not be used more than once for **case**-labels in a **switch**-statement. For example:

```
void f(int i)
{
        switch (i) {
        case 2.7:  // error: floating point uses for case
                // ...
        case 2:
                // ...
        case 4–2: // error: 2 used twice in case labels
                // ...
};
```

A **switch**-statement can alternatively be written as a set of **if**-statements. For example:

```
switch (val) {
case 1:
        f();
        break;
case 2:
        g();
        break;
default:
        h();
        break;
}
```

This could be expressed as:

```
if (val == 1)
        f();
else if (val == 2)
        g();
else
        h();
```

The meaning is the same, but the first (**switch**) version is preferred because the nature of the operation (testing a single value against a set of constants) is explicit. This makes the **switch**-statement easier to read for nontrivial examples. It typically also leads to the generation of better code because there is no reason to repeatedly check individual values. Instead, a jump table can be used.

Beware that a case of a switch must be terminated somehow unless you want to carry on executing the next case. Consider:

```
switch (val) {              // beware
case 1:
        cout << "case 1\n";
case 2:
        cout << "case 2\n";
default:
        cout << "default: case not found\n";
}
```

Invoked with `val==1`, the output will greatly surprise the uninitiated:

```
case 1
case 2
default: case not found
```

It is a good idea to comment the (rare) cases in which a fall-through is intentional so that an uncommented fall-through can be assumed to be an error. For example:

```
switch (action) {               // handle (action,value) pair
case do_and_print:
        act(value);
        // no break: fall through to print
case print:
        print(value);
        break;
// ...
}
```

A `break` is the most common way of terminating a case, but a `return` is often useful (§10.2.1).

When should a `switch`-statement have a `default`? There is no single answer that covers all situations. One use is for the `default` to handle the most common case. Another common use is the exact opposite: the `default`: action is simply a way to catch errors; every valid alternative is covered by the `case`s. However, there is one case where a `default` should not be used: if a `switch` is intended to have one case for each enumerator of an enumeration. If so, leaving out the `default` gives the compiler a chance to warn against a set of `case`s that almost but not quite match the set of enumerators. For example, this is almost certainly an error:

```
enum class Vessel { cup, glass, goblet, chalice };

void problematic(Vessel v)
{
        switch (v) {
        case Vessel::cup:       /* ... */       break;
        case Vessel::glass:     /* ... */       break;
        case Vessel::goblet:    /* ... */       break;
        }
}
```

Such a mistake can easily occur when a new enumerator is added during maintenance.

Testing for an ''impossible'' enumerator value is best done separately.

9.4.2.1 Declarations in Cases

It is possible, and common, to declare variables within the block of a **switch**-statement. However, it is not possible to bypass an initialization. For example:

```
void f(int i)
{
    switch (i) {
    case 0:
        int x;          // uninitialized
        int y = 3;      // error: declaration can be bypassed (explicitly initialized)
        string s;       // error: declaration can be bypassed (implicitly initialized)
    case 1:
        ++x;            // error: use of uninitialized object
        ++y;
        s = "nasty!";
    }
}
```

Here, if **i==1**, the thread of execution would bypass the initializations of **y** and **s**, so **f()** will not compile. Unfortunately, because an **int** needn't be initialized, the declaration of **x** is not an error. However, its use is an error: we read an uninitialized variable. Unfortunately, compilers often give just a warning for the use of an uninitialized variable and cannot reliably catch all such misuses. As usual, avoid uninitialized variables (§6.3.5.1).

If we need a variable within a **switch**-statement, we can limit its scope by enclosing its declaration and its use in a block. For an example, see **prim()** in §10.2.1.

9.4.3 Declarations in Conditions

To avoid accidental misuse of a variable, it is usually a good idea to introduce the variable into the smallest scope possible. In particular, it is usually best to delay the definition of a local variable until one can give it an initial value. That way, one cannot get into trouble by using the variable before its initial value is assigned.

One of the most elegant applications of these two principles is to declare a variable in a condition. Consider:

```
if (double d = prim(true)) {
    left /= d;
    break;
}
```

Here, **d** is declared and initialized and the value of **d** after initialization is tested as the value of the condition. The scope of **d** extends from its point of declaration to the end of the statement that the condition controls. For example, had there been an **else**-branch to the **if**-statement, **d** would be in scope on both branches.

The obvious and traditional alternative is to declare **d** before the condition. However, this opens the scope (literally) for the use of **d** before its initialization or after its intended useful life:

```
double d;
// ...
d2 = d;     // oops!
// ...
if (d = prim(true)) {
        left /= d;
        break;
}
// ...
d = 2.0;    // two unrelated uses of d
```

In addition to the logical benefits of declaring variables in conditions, doing so also yields the most compact source code.

A declaration in a condition must declare and initialize a single variable or **const**.

9.5 Iteration Statements

A loop can be expressed as a **for**-, **while**-, or **do**-statement:

while (*condition* **)** *statement*
do *statement* **while (** *expression* **) ;**
for (*for-init-statement condition*$_{opt}$ **;** *expression*$_{opt}$ **)** *statement*
for (*for-declaration* **:** *expression* **)** *statement*

A *for-init-statement* must be either a declaration or an *expression-statement*. Note that both end with a semicolon.

The statement of a **for**-statement (called the *controlled statement* or the *loop body*) is executed repeatedly until the condition becomes **false** or the programmer breaks out of the loop some other way (such as a **break**, a **return**, a **throw**, or a **goto**).

More complicated loops can be expressed as an algorithm plus a lambda expression (§11.4.2).

9.5.1 Range-for Statements

The simplest loop is a range-**for**-statement; it simply gives the programmer access to each element of a range. For example:

```
int sum(vector<int>& v)
{
        int s = 0;
        for (int x : v)
                s+=x;
        return s;
}
```

The **for (int x : v)** can be read as "for each element **x** in the range **v**" or just "for each **x** in **v**." The elements of **v** are visited in order from the first to the last.

The scope of the variable naming the element (here, **x**) is the **for**-statement.

The expression after the colon must denote a sequence (a range); that is, it must yield a value for which we can call **v.begin()** and **v.end()** or **begin(v)** and **end(v)** to obtain an iterators (§4.5):

[1] the compiler first looks for members **begin** and **end** and tries to use those. If a **begin** or an **end** is found that cannot be used as a range (e.g., because a member **begin** is a variable rather than a function), the range-**for** is an error.

[2] Otherwise, the compiler looks for a **begin**/**end** member pair in the enclosing scope. If none is found or if what is found cannot be used (e.g., because the **begin** did not take an argument of the sequence's type), the range-**for** is an error.

The compiler uses **v** and **v+N** as **begin(v)** and **end(v)** for a built-in array **T v[N]**. The **<iterator>** header provides **begin(c)** and **end(c)** for built-in arrays and for all standard-library containers. For sequences of our own design, we can define **begin()** and **end()** in the same way as it is done for standard-library containers (§4.4.5).

The controlled variable, **x** in the example, that refers to the current element is equivalent to ***p** when using an equivalent **for**-statement:

```
int sum2(vector<int>& v)
{
    int s = 0;
    for (auto p = begin(v); p!=end(v); ++p)
        s+=*p;
    return s;
}
```

If you need to modify an element in a range-**for** loop, the element variable should be a reference. For example, we can increment each element of a **vector** like this:

```
void incr(vector<int>& v)
{
    for (int& x : v)
        ++x;
}
```

References are also appropriate for elements that might be large, so that copying them to the element value could be costly. For example:

```
template<class T> T accum(vector<T>& v)
{
    T sum = 0;
    for (const T& x : v)
        sum += x;
    return sum;
}
```

Note that a range-**for** loop is a deliberately simple construct. For example, using it you can't touch two elements at the same time and can't effectively traverse two ranges simultaneously. For that we need a general **for**-statement.

9.5.2 for Statements

There is also a more general **for**-statement allowing greater control of the iteration. The loop variable, the termination condition, and the expression that updates the loop variable are explicitly presented ''up front'' on a single line. For example:

```
void f(int v[], int max)
{
    for (int i = 0; i!=max; ++i)
        v[i] = i*i;
}
```

This is equivalent to

```
void f(int v[], int max)
{
    int i = 0;          // introduce loop variable
    while (i!=max) {    // test termination condition
        v[i] = i*i;     // execute the loop body
        ++i;            // increment loop variable
    }
}
```

A variable can be declared in the initializer part of a **for**-statement. If that initializer is a declaration, the variable (or variables) it introduced is in scope until the end of the **for**-statement.

It is not always obvious what is the right type to use for a controlled variable in a *for* loop, so **auto** often comes in handy:

```
for (auto p = begin(c); c!=end(c); ++p) {
    // ... use iterator p for elements in container c ...
}
```

If the final value of an index needs to be known after exit from a **for**-loop, the index variable must be declared outside the **for**-loop (e.g., see §9.6).

If no initialization is needed, the initializing statement can be empty.

If the expression that is supposed to increment the loop variable is omitted, we must update some form of loop variable elsewhere, typically in the body of the loop. If the loop isn't of the simple ''introduce a loop variable, test the condition, update the loop variable'' variety, it is often better expressed as a **while**-statement. However, consider this elegant variant:

```
for (string s; cin>>s;)
    v.push_back(s);
```

Here, the reading and testing for termination and combined in **cin>>s**, so we don't need an explicit loop variable. On the other hand, the use of **for**, rather than **while**, allows us to limit the scope of the ''current element,'' **s**, to the loop itself (the **for**-statement).

A **for**-statement is also useful for expressing a loop without an explicit termination condition:

```
for (;;) {    // "forever"
    // ...
}
```

However, many consider this idiom obscure and prefer to use:

```
while(true) {    // "forever"
    // ...
}
```

9.5.3 while Statements

A **while**-statement executes its controlled statement until its condition becomes **false**. For example:

```
template<class Iter, class Value>
Iter find(Iter first, Iter last, Value val)
{
    while (first!=last && *first!=val)
        ++first;
    return first;
}
```

I tend to prefer **while**-statements over **for**-statements when there isn't an obvious loop variable or where the update of a loop variable naturally comes in the middle of the loop body.

A **for**-statement (§9.5.2) is easily rewritten into an equivalent **while**-statement and vice versa.

9.5.4 do Statements

A **do**-statement is similar to a **while**-statement except that the condition comes after the body. For example:

```
void print_backwards(char a[], int i)    // i must be positive
{
    cout << '{';
    do {
        cout << a[--i];
    } while (i);
    cout << '}';
}
```

This might be called like this: **print_backwards(s,strlen(s))**; but it is all too easy to make a horrible mistake. For example, what if **s** was the empty string?

In my experience, the **do**-statement is a source of errors and confusion. The reason is that its body is always executed once before the condition is evaluated. However, for the body to work correctly, something very much like the condition must hold even the first time through. More often than I would have guessed, I have found that condition not to hold as expected either when the program was first written and tested or later after the code preceding it has been modified. I also prefer the condition "up front where I can see it." Consequently, I recommend avoiding **do**-statements.

9.5.5 Loop Exit

If the *condition* of an iteration statement (a **for**-, **while**-, or **do**-statement) is omitted, the loop will not terminate unless the user explicitly exits it by a **break**, **return** (§12.1.4), **goto** (§9.6), **throw** (§13.5), or some less obvious way such as a call of **exit()** (§15.4.3). A **break** "breaks out of" the

nearest enclosing *switch-statement* (§9.4.2) or *iteration-statement*. For example:

```
void f(vector<string>& v, string terminator)
{
        char c;
        string s;
        while (cin>>c) {
                // ...
                if (c == '\n') break;
                // ...
        }
}
```

We use a **break** when we need to leave the loop body "in the middle." Unless it warps the logic of a loop (e.g., requires the introduction of an extra varible), it is usually better to have the complete exit condition as the condition of a **while**-statement or a **for**-statement.

Sometimes, we don't want to exit the loop completely, we just want to get to the end of the loop body. A **continue** skips the rest of the body of an *iteration-statement*. For example:

```
void find_prime(vector<string>& v)
{
        for (int i = 0; i!=v.size(); ++i) {
                if (!prime(v[i]) continue;
                return v[i];
        }
}
```

After a **continue**, the increment part of the loop (if any) is executed, followed by the loop condition (if any). So **find_prime()** could equivalently have been written as:

```
void find_prime(vector<string>& v)
{
        for (int i = 0; i!=v.size(); ++i) {
                if (!prime(v[i]) {
                        return v[i];
                }
        }
}
```

9.6 goto Statements

C++ possesses the infamous **goto**:

goto *identifier* **;**
identifier **:** *statement*

The **goto** has few uses in general high-level programming, but it can be very useful when C++ code is generated by a program rather than written directly by a person; for example, **goto**s can be used in a parser generated from a grammar by a parser generator.

The scope of a label is the function it is in (§6.3.4). This implies that you can use **goto** to jump both into and out of blocks. The only restriction is that you cannot jump past an initializer or into an exception handler (§13.5).

One of the few sensible uses of **goto** in ordinary code is to break out from a nested loop or **switch**-statement (a **break** breaks out of only the innermost enclosing loop or **switch**-statement). For example:

```
void do_something(int i, int j)
    // do something to a two-dimensional matrix called mn
{
    for (i = 0; i!=n; ++i)
        for (j = 0; j!=m; ++j)
            if (nm[i][j] == a)
                goto found;
    // not found
    // ...
found:
    // nm[i][j] == a
}
```

Note that this **goto** just jumps forward to exit its loop. It does not introduce a new loop or enter a new scope. That makes it the least troublesome and least confusing use of a **goto**.

9.7 Comments and Indentation

Judicious use of comments and consistent use of indentation can make the task of reading and understanding a program much more pleasant. Several different consistent styles of indentation are in use. I see no fundamental reason to prefer one over another (although, like most programmers, I have my preferences, and this book reflects them). The same applies to styles of comments.

Comments can be misused in ways that seriously affect the readability of a program. The compiler does not understand the contents of a comment, so it has no way of ensuring that a comment

- is meaningful,
- describes the program, and
- is up to date.

Most programs contain comments that are incomprehensible, ambiguous, and just plain wrong. Bad comments can be worse than no comments.

If something can be stated *in the language itself*, it should be, and not just mentioned in a comment. This remark is aimed at comments such as these:

```
// variable "v" must be initialized

// variable "v" must be used only by function "f()"

// call function "init()" before calling any other function in this file

// call function "cleanup()" at the end of your program
```

```
// don't use function "weird()"

// function "f(int ...)" takes two or three arguments
```

Such comments can typically be rendered unnecessary by proper use of C++.

Once something has been stated clearly in the language, it should not be mentioned a second time in a comment. For example:

```
a = b+c;   // a becomes b+c
count++;  // increment the counter
```

Such comments are worse than simply redundant. They increase the amount of text the reader has to look at, they often obscure the structure of the program, and they may be wrong. Note, however, that such comments are used extensively for teaching purposes in programming language textbooks such as this. This is one of the many ways a program in a textbook differs from a real program.

A good comment states what a piece of code is supposed to do (the intent of the code), whereas the code (only) states what it does (in terms of how it does it). Preferably, a comment is expressed at a suitably high level of abstraction so that it is easy for a human to understand without delving into minute details.

My preference is for:

- A comment for each source file stating what the declarations in it have in common, references to manuals, the name of the programmer, general hints for maintenance, etc.
- A comment for each class, template, and namespace
- A comment for each nontrivial function stating its purpose, the algorithm used (unless it is obvious), and maybe something about the assumptions it makes about its environment
- A comment for each global and namespace variable and constant
- A few comments where the code is nonobvious and/or nonportable
- Very little else

For example:

```
//      tbl.c: Implementation of the symbol table.

/*
        Gaussian elimination with partial pivoting.
        See Ralston: "A first course ..." pg 411.
*/

//      scan(p,n,c) requires that p points to an array of at least n elements

// sort(p,q) sorts the elements of the sequence [p:q) using < for comparison.

// Revised to handle invalid dates. Bjarne Stroustrup, Feb 29 2013
```

A well-chosen and well-written set of comments is an essential part of a good program. Writing good comments can be as difficult as writing the program itself. It is an art well worth cultivating.

Note that /* */ style comments do not nest. For example:

```
/*
        remove expensive check
        if (check(p,q)) error("bad p q") /* should never happen */
*/
```

This nesting should give an error for an unmatched final */.

9.8 Advice

[1] Don't declare a variable until you have a value to initialize it with; §9.3, §9.4.3, §9.5.2.
[2] Prefer a **switch**-statement to an **if**-statement when there is a choice; §9.4.2.
[3] Prefer a range-**for**-statement to a **for**-statement when there is a choice; §9.5.1.
[4] Prefer a **for**-statement to a **while**-statement when there is an obvious loop variable; §9.5.2.
[5] Prefer a **while**-statement to a **for**-statement when there is no obvious loop variable; §9.5.3.
[6] Avoid **do**-statements; §9.5.
[7] Avoid **goto**; §9.6.
[8] Keep comments crisp; §9.7.
[9] Don't say in comments what can be clearly stated in code; §9.7.
[10] State intent in comments; §9.7.
[11] Maintain a consistent indentation style; §9.7.

10

Expressions

Programming is like sex:
It may give some concrete results,
but that is not why we do it.
– apologies to Richard Feynman

- Introduction
- A Desk Calculator
  The Parser; Input; Low-Level Input; Error Handling; The Driver; Headers; Command-Line Arguments; A Note on Style
- Operator Summary
  Results; Order of Evaluation; Operator Precedence; Temporary Objects
- Constant Expressions
  Symbolic Constants; **const**s in Constant Expressions; Literal Types; Reference Arguments; Address Constant Expressions
- Implicit Type Conversion
  Promotions; Conversions; Usual Arithmetic Conversions
- Advice

10.1 Introduction

This chapter discusses expressions in some detail. In C++, an assignment is an expression, a function call is an expression, the construction of an object is an expression, and so are many other operations that go beyond conventional arithmetic expression evaluation. To give an impression of how expressions are used and to show them in context, I first present a small complete program, a simple ‘‘desk calculator.’’ Next, the complete set of operators is listed and their meaning for built-in types is briefly outlined. The operators that require more extensive explanation are discussed in Chapter 11.

10.2 A Desk Calculator

Consider a simple desk calculator program that provides the four standard arithmetic operations as infix operators on floating-point numbers. The user can also define variables. For example, given the input

```
r = 2.5
area = pi * r * r
```

(pi is predefined) the calculator program will write

```
2.5
19.635
```

where 2.5 is the result of the first line of input and 19.635 is the result of the second.

The calculator consists of four main parts: a parser, an input function, a symbol table, and a driver. Actually, it is a miniature compiler in which the parser does the syntactic analysis, the input function handles input and lexical analysis, the symbol table holds permanent information, and the driver handles initialization, output, and errors. We could add many features to this calculator to make it more useful, but the code is long enough as it is, and most features would just add code without providing additional insight into the use of C++.

10.2.1 The Parser

Here is a grammar for the language accepted by the calculator:

```
program:
        end                          // end is end-of-input
        expr_list end

expr_list:
        expression print             // print is newline or semicolon
        expression print expr_list

expression:
        expression + term
        expression – term
        term

term:
        term / primary
        term * primary
        primary

primary:
        number                       // number is a floating-point literal
        name                         // name is an identifier
        name = expression
        – primary
        ( expression )
```

In other words, a program is a sequence of expressions separated by semicolons. The basic units of an expression are numbers, names, and the operators `*`, `/`, `+`, `–` (both unary and binary), and `=` (assignment). Names need not be declared before use.

I use a style of syntax analysis called *recursive descent*; it is a popular and straightforward top-down technique. In a language such as C++, in which function calls are relatively cheap, it is also efficient. For each production in the grammar, there is a function that calls other functions. Terminal symbols (for example, `end`, `number`, `+`, and `–`) are recognized by a lexical analyzer and nonterminal symbols are recognized by the syntax analyzer functions, `expr()`, `term()`, and `prim()`. As soon as both operands of a (sub)expression are known, the expression is evaluated; in a real compiler, code could be generated at this point.

For input, the parser uses a `Token_stream` that encapsulates the reading of characters and their composition into `Token`s. That is, a `Token_stream` ''tokenizes'': it turns streams of characters, such as `123.45`, into `Token`s. A `Token` is a {kind-of-token,value} pair, such as `{number,123.45}`, where the `123.45` has been turned into a floating point value. The main parts of the parser need only to know the name of the `Token_stream`, `ts`, and how to get `Token`s from it. To read the next `Token`, it calls `ts.get()`. To get the most recently read `Token` (the ''current token''), it calls `ts.current()`. In addition to providing tokenizing, the `Token_stream` hides the actual source of the characters. We'll see that they can come directly from a user typing to `cin`, from a program command line, or from any other input stream (§10.2.7).

The definition of `Token` looks like this:

```
enum class Kind : char {
    name, number, end,
    plus='+', minus='–', mul='*', div='/', print=';', assign='=', lp='(', rp=')'
};

struct Token {
    Kind kind;
    string string_value;
    double number_value;
};
```

Representing each token by the integer value of its character is convenient and efficient and can be a help to people using debuggers. This works as long as no character used as input has a value used as an enumerator – and no current character set I know of has a printing character with a single-digit integer value.

The interface to `Token_stream` looks like this:

```
class Token_stream {
public:
    Token get();                // read and return next token
    const Token& current();     // most recently read token
    // ...
};
```

The implementation is presented in §10.2.2.

Each parser function takes a `bool` (§6.2.2) argument, called `get`, indicating whether the function needs to call `Token_stream::get()` to get the next token. Each parser function evaluates ''its''

expression and returns the value. The function **expr()** handles addition and subtraction. It consists of a single loop that looks for terms to add or subtract:

```
double expr(bool get)           // add and subtract
{
        double left = term(get);

        for (;;) {                          // "forever"
                switch (ts.current().kind) {
                case Kind::plus:
                        left += term(true);
                        break;
                case Kind::minus:
                        left -= term(true);
                        break;
                default:
                        return left;
                }
        }
}
```

This function really does not do much itself. In a manner typical of higher-level functions in a large program, it calls other functions to do the work.

The **switch**-statement (§2.2.4, §9.4.2) tests the value of its condition, which is supplied in parentheses after the **switch** keyword, against a set of constants. The **break**-statements are used to exit the **switch**-statement. If the value tested does not match any **case** label, the **default** is chosen. The programmer need not provide a **default**.

Note that an expression such as **2-3+4** is evaluated as **(2-3)+4**, as specified in the grammar.

The curious notation **for(;;)** is a way to specify an infinite loop; you could pronounce it "forever" (§9.5); **while(true)** is an alternative. The **switch**-statement is executed repeatedly until something different from **+** and **-** is found, and then the **return**-statement in the default case is executed.

The operators **+=** and **-=** are used to handle the addition and subtraction; **left=left+term(true)** and **left=left-term(true)** could have been used without changing the meaning of the program. However, **left+=term(true)** and **left-=term(true)** are not only shorter but also express the intended operation directly. Each assignment operator is a separate lexical token, so **a + = 1;** is a syntax error because of the space between the **+** and the **=**.

C++ provides assignment operators for the binary operators:

```
+    -    *    /    %    &    |    ^    <<   >>
```

so that the following assignment operators are possible:

```
=    +=   -=   *=   /=   %=   &=   |=   ^=   <<=  >>=
```

The **%** is the modulo, or remainder, operator; **&**, **|**, and **^** are the bitwise logical operators and, or, and exclusive or; **<<** and **>>** are the left shift and right shift operators; §10.3 summarizes the operators and their meanings. For a binary operator **@** applied to operands of built-in types, an expression **x@=y** means **x=x@y**, except that **x** is evaluated once only.

The function **term()** handles multiplication and division in the same way **expr()** handles addition and subtraction:

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

    for (;;) {
        switch (ts.current().kind) {
        case Kind::mul:
            left *= prim(true);
            break;
        case Kind::div:
            if (auto d = prim(true)) {
                left /= d;
                break;
            }
            return error("divide by 0");
        default:
            return left;
        }
    }
}
```

The result of dividing by zero is undefined and usually disastrous. We therefore test for **0** before dividing and call **error()** if we detect a zero divisor. The function **error()** is described in §10.2.4.

The variable **d** is introduced into the program exactly where it is needed and initialized immediately. The scope of a name introduced in a condition is the statement controlled by that condition, and the resulting value is the value of the condition (§9.4.3). Consequently, the division and assignment **left/=d** are done if and only if **d** is nonzero.

The function **prim()** handling a *primary* is much like **expr()** and **term()**, except that because we are getting lower in the call hierarchy a bit of real work is being done and no loop is necessary:

```
double prim(bool get)           // handle primaries
{
    if (get) ts.get(); // read next token

    switch (ts.current().kind) {
    case Kind::number:      // floating-point constant
    {   double v = ts.current().number_value;
        ts.get();
        return v;
    }
    case Kind::name:
    {   double& v = table[ts.current().string_value];      // find the corresponding
        if (ts.get().kind == Kind::assign) v = expr(true);  // '=' seen: assignment
        return v;
    }
```

```
        case Kind::minus:          // unary minus
            return –prim(true);
        case Kind::lp:
        {   auto e = expr(true);
            if (ts.current().kind != Kind::rp) return error("')' expected");
            ts.get();          // eat ')'
            return e;
        }
        default:
            return error("primary expected");
        }
    }
```

When a `Token` that is a `number` (that is, an integer or floating-point literal) is seen, its value is placed in its `number_value`. Similarly, when a `Token` that is a `name` (however defined; see §10.2.2 and §10.2.3) is seen, its value is placed in its `string_value`.

Note that `prim()` always reads one more `Token` than it uses to analyze its primary expression. The reason is that it *must* do that in some cases (e.g., to see if a name is assigned to), so for consistency it must do it in all cases. In the cases where a parser function simply wants to move ahead to the next `Token`, it doesn't use the return value from `ts.get()`. That's fine because we can get the result from `ts.current()`. Had ignoring the return value of `get()` bothered me, I'd have either added a `read()` function that just updated `current()` without returning a value or explicitly ''thrown away'' the result: `void(ts.get())`.

Before doing anything to a name, the calculator must first look ahead to see if it is being assigned to or simply read. In both cases, the symbol table is consulted. The symbol table is a `map` (§4.4.3, §31.4.3):

```
map<string,double> table;
```

That is, when `table` is indexed by a `string`, the resulting value is the `double` corresponding to the `string`. For example, if the user enters

```
radius = 6378.388;
```

the calculator will reach `case Kind::name` and execute

```
double& v = table["radius"];
// ... expr() calculates the value to be assigned ...
v = 6378.388;
```

The reference `v` is used to hold on to the `double` associated with `radius` while `expr()` calculates the value `6378.388` from the input characters.

Chapter 14 and Chapter 15 discuss how to organize a program as a set of modules. However, with one exception, the declarations for this calculator example can be ordered so that everything is declared exactly once and before it is used. The exception is `expr()`, which calls `term()`, which calls `prim()`, which in turn calls `expr()`. This loop of calls must be broken somehow. A declaration

```
double expr(bool);
```

before the definition of `prim()` will do nicely.

10.2.2 Input

Reading input is often the messiest part of a program. To communicate with a person, the program must cope with that person's whims, conventions, and seemingly random errors. Trying to force the person to behave in a manner more suitable for the machine is often (rightly) considered offensive. The task of a low-level input routine is to read characters and compose higher-level tokens from them. These tokens are then the units of input for higher-level routines. Here, low-level input is done by **ts.get()**. Writing a low-level input routine need not be an everyday task. Many systems provide standard functions for this.

First we need to see the complete definition of **Token_stream**:

```
class Token_stream {
public:
        Token_stream(istream& s) : ip{&s}, owns{false} { }
        Token_stream(istream* p) : ip{p}, owns{true} { }

        ~Token_stream() { close(); }

        Token get();            // read and return next token
        Token& current();       // most recently read token

        void set_input(istream& s) { close(); ip = &s; owns=false; }
        void set_input(istream* p) { close(); ip = p; owns = true; }

private:
        void close() { if (owns) delete ip; }

        istream* ip;                    // pointer to an input stream
        bool owns;                      // does the Token_stream own the istream?
        Token ct {Kind::end} ;          // current token
};
```

We initialize a **Token_stream** with an input stream (§4.3.2, Chapter 38) from which it gets its characters. The **Token_stream** implements the convention that it owns (and eventually deletes; §3.2.1.2, §11.2) an **istream** passed as a pointer, but not an **istream** passed as a reference. This may be a bit elaborate for this simple program, but it is a useful and general technique for classes that hold a pointer to a resource requiring destruction.

A **Token_stream** holds three values: a pointer to its input stream (**ip**), a Boolean (**owns**), indicating ownership of the input stream, and the current token (**ct**).

I gave **ct** a default value because it seemed sloppy not to. People should not call **current()** before **get()**, but if they do, they get a well-defined **Token**. I chose **Kind::end** as the initial value for **ct** so that a program that misuses **current()** will not get a value that wasn't on the input stream.

I present **Token_stream::get()** in two stages. First, I provide a deceptively simple version that imposes a burden on the user. Next, I modify it into a slightly less elegant, but much easier to use, version. The idea for **get()** is to read a character, use that character to decide what kind of token needs to be composed, read more characters when needed, and then return a **Token** representing the characters read.

The initial statements read the first non-whitespace character from `*ip` (the stream pointed to by `ip`) into `ch` and check that the read operation succeeded:

```
Token Token_stream::get()
{
    char ch = 0;
    *ip>>ch;

    switch (ch) {
    case 0:
        return ct={Kind::end};    // assign and return
```

By default, operator `>>` skips whitespace (that is, spaces, tabs, newlines, etc.) and leaves the value of `ch` unchanged if the input operation failed. Consequently, `ch==0` indicates end-of-input.

Assignment is an operator, and the result of the assignment is the value of the variable assigned to. This allows me to assign the value `Kind::end` to `curr_tok` and return it in the same statement. Having a single statement rather than two is useful in maintenance. If the assignment and the `return` became separated in the code, a programmer might update the one and forget to update the other.

Note also how the `{}`-list notation (§3.2.1.3, §11.3) is used on the right-hand side of an assignment. That is, it is an expression. I could have written that `return`-statement as:

```
ct.kind = Kind::end; // assign
return ct;           // return
```

However, I think that assigning a complete object `{Kind::end}` is clearer than dealing with individual members of `ct`. The `{Kind::end}` is equivalent to `{Kind::end,0,0}`. That's good if we care about the last two members of the `Token` and not so good if we are worried about performance. Neither is the case here, but in general dealing with complete objects is clearer and less error-prone than manipulating data members individually. The cases below give examples of the other strategy.

Consider some of the cases separately before considering the complete function. The expression terminator, `';'`, the parentheses, and the operators are handled simply by returning their values:

```
    case ';':    // end of expression; print
    case '*':
    case '/':
    case '+':
    case '-':
    case '(':
    case ')':
    case '=':
        return ct={static_cast<Kind>(ch)};
```

The `static_cast` (§11.5.2) is needed because there is no implicit conversion from `char` to `Kind` (§8.4.1); only some characters correspond to `Kind` values, so we have to "certify" that in this case `ch` does.

Numbers are handled like this:

```
    case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9':
    case '.':
```

```
        ip–>putback(ch);          // put the first digit (or .) back into the input stream
        *ip >> ct.number_value;   // read the number into ct
        ct.kind=Kind::number;
        return ct;
```

Stacking `case` labels horizontally rather than vertically is generally not a good idea because this arrangement is harder to read. However, having one line for each digit is tedious. Because operator `>>` is already defined for reading floating-point values into a `double`, the code is trivial. First the initial character (a digit or a dot) is put back into `cin`. Then, the floating-point value can be read into `ct.number_value`.

If the token is not the end of input, an operator, a punctuation character, or a number, it must be a name. A name is handled similarly to a number:

```
    default:                // name, name =, or error
        if (isalpha(ch)) {
            ip–>putback(ch);            // put the first character back into the input stream
            *ip>>ct.string_value;       // read the string into ct
            ct.kind=Kind::name;
            return ct;
        }
```

Finally, we may simply have an error. The simple-minded, but reasonably effective way to deal with an error is the write call an `error()` function and then return a `print` token if `error()` returns:

```
    error("bad token");
    return ct={Kind::print};
```

The standard-library function `isalpha()` (§36.2.1) is used to avoid listing every character as a separate `case` label. Operator `>>` applied to a string (in this case, `string_value`) reads until it hits white-space. Consequently, a user must terminate a name by a space before an operator using the name as an operand. This is less than ideal, so we will return to this problem in §10.2.3.

Here, finally, is the complete input function:

```
Token Token_stream::get()
{
    char ch = 0;
    *ip>>ch;

    switch (ch) {
    case 0:
        return ct={Kind::end};          // assign and return
    case ';':   // end of expression; print
    case '*':
    case '/':
    case '+':
    case '–':
    case '(':
    case ')':
    case '=':
        return ct=={static_cast<Kind>(ch)};
```

```
        case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9':
        case '.':
                ip->putback(ch);                // put the first digit (or .) back into the input stream
                *ip >> ct.number_value;         // read number into ct
                ct.kind=Kind::number;
                return ct;
        default:                // name, name =, or error
                if (isalpha(ch)) {
                        ip->putback(ch);        // put the first character back into the input stream
                        *ip>>ct.string_value;   // read string into ct
                        ct.kind=Kind::name;
                        return ct;
                }

                error("bad token");
                return ct={Kind::print};
        }
}
```

The conversion of an operator to its `Token` value is trivial because the `kind` of an operator was defined as the integer value of the operator (§10.2.1).

10.2.3 Low-Level Input

Using the calculator as defined so far reveals a few inconveniences. It is tedious to remember to add a semicolon after an expression in order to get its value printed, and having a name terminated by whitespace only is a real nuisance. For example, `x=7` is an identifier – rather than the identifier `x` followed by the operator `=` and the number `7`. To get what we (usually) want, we would have to add whitespace after `x`: `x =7`. Both problems are solved by replacing the type-oriented default input operations in `get()` with code that reads individual characters.

First, we'll make a newline equivalent to the semicolon used to mark the end-of-expression:

```
Token Token_stream::get()
{
        char ch;

        do {  // skip whitespace except '\n'
                if (!ip->get(ch)) return ct={Kind::end};
        } while (ch!='\n' && isspace(ch));

        switch (ch) {
        case ';':
        case '\n':
                return ct={Kind::print};
```

Here, I use a `do`-statement; it is equivalent to a `while`-statement except that the controlled statement is always executed at least once. The call `ip->get(ch)` reads a single character from the input stream `*ip` into `ch`. By default, `get()` does not skip whitespace the way `>>` does. The test `if (!ip->get(ch))` succeeds if no character can be read from `cin`; in this case, `Kind::end` is returned to terminate the calculator session. The operator `!` (not) is used because `get()` returns `true` in case of success.

The standard-library function **isspace()** provides the standard test for whitespace (§36.2.1); **isspace(c)** returns a nonzero value if **c** is a whitespace character and zero otherwise. The test is implemented as a table lookup, so using **isspace()** is much faster than testing for the individual whitespace characters. Similar functions test if a character is a digit (**isdigit()**), a letter (**isalpha()**), or a digit or letter (**isalnum()**).

After whitespace has been skipped, the next character is used to determine what kind of lexical token is coming.

The problem caused by **>>** reading into a string until whitespace is encountered is solved by reading one character at a time until a character that is not a letter or a digit is found:

```
default:                // NAME, NAME=, or error
    if (isalpha(ch)) {
        string_value = ch;
        while (ip->get(ch) && isalnum(ch))
            string_value += ch; // append ch to end of string_value
        ip->putback(ch);
        return ct={Kind::name};
    }
```

Fortunately, these two improvements could both be implemented by modifying a single local section of code. Constructing programs so that improvements can be implemented through local modifications only is an important design aim.

You might worry that adding characters to the end of a **string** one by one would be inefficient. It would be for very long **string**s, but all modern **string** implementations provide the ''small string optimization'' (§19.3.3). That means that handling the kind of strings we are likely to use as names in a calculator (or even in a compiler) doesn't involve any inefficient operations. In particular, using a short **string** doesn't require any use of free store. The maximum number of characters for a short **string** is implementation-dependent, but 14 would be a good guess.

10.2.4 Error Handling

It is always important to detect and report errors. However, for this program, a simple error handling strategy suffices. The **error()** function simply counts the errors, writes out an error message, and returns:

```
int no_of_errors;

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

The stream **cerr** is an unbuffered output stream usually used to report errors (§38.1).

The reason for returning a value is that errors typically occur in the middle of the evaluation of an expression, so we should either abort that evaluation entirely or return a value that is unlikely to cause subsequent errors. The latter is adequate for this simple calculator. Had **Token_stream::get()**

kept track of the line numbers, `error()` could have informed the user approximately where the error occurred. This would be useful when the calculator is used noninteractively.

A more stylized and general error-handling strategy would separate error detection from error recovery. This can be implemented using exceptions (see §2.4.3.1, Chapter 13), but what we have here is quite suitable for a 180-line calculator.

10.2.5 The Driver

With all the pieces of the program in place, we need only a driver to start things. I decided on two functions: `main()` to do setup and error reporting and `calculate()` to handle the actual calculation:

```
Token_stream ts {cin};    // use input from cin

void calculate()
{
    for (;;) {
        ts.get();
        if (ts.current().kind == Kind::end) break;
        if (ts.current().kind == Kind::print) continue;
        cout << expr(false) << '\n';
    }
}

int main()
{
    table["pi"] = 3.1415926535897932385;    // insert predefined names
    table["e"] = 2.7182818284590452354;

    calculate();

    return no_of_errors;
}
```

Conventionally, `main()` returns zero if the program terminates normally and nonzero otherwise (§2.2.1). Returning the number of errors accomplishes this nicely. As it happens, the only initialization needed is to insert the predefined names into the symbol table.

The primary task of the main loop (in `calculate()`) is to read expressions and write out the answer. This is achieved by the line:

```
    cout << expr(false) << '\n';
```

The argument `false` tells `expr()` that it does not need to call `ts.get()` to read a token on which to work.

Testing for `Kind::end` ensures that the loop is correctly exited when `ts.get()` encounters an input error or an end-of-file. A `break`-statement exits its nearest enclosing `switch`-statement or loop (§9.5). Testing for `Kind::print` (that is, for `'\n'` and `';'`) relieves `expr()` of the responsibility for handling empty expressions. A `continue`-statement is equivalent to going to the very end of a loop.

10.2.6 Headers

The calculator uses standard-library facilities. Therefore, appropriate headers must be **#include**d to complete the program:

```
#include<iostream> // I/O
#include<string>   // strings
#include<map>      // map
#include<cctype>   // isalpha(), etc.
```

All of these headers provide facilities in the **std** namespace, so to use the names they provide we must either use explicit qualification with **std::** or bring the names into the global namespace by

```
using namespace std;
```

To avoid confusing the discussion of expressions with modularity issues, I did the latter. Chapter 14 and Chapter 15 discuss ways of organizing this calculator into modules using namespaces and how to organize it into source files.

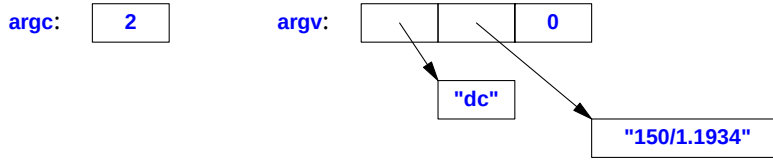
10.2.7 Command-Line Arguments

After the program was written and tested, I found it a bother to first start the program, then type the expressions, and finally quit. My most common use was to evaluate a single expression. If that expression could be presented as a command-line argument, a few keystrokes could be avoided.

A program starts by calling **main()** (§2.2.1, §15.4). When this is done, **main()** is given two arguments specifying the number of arguments, conventionally called **argc**, and an array of arguments, conventionally called **argv**. The arguments are C-style character strings (§2.2.5, §7.3), so the type of **argv** is **char*[argc+1]**. The name of the program (as it occurs on the command line) is passed as **argv[0]**, so **argc** is always at least **1**. The list of arguments is zero-terminated; that is, **argv[argc]==0**. For example, for the command

```
dc 150/1.1934
```

the arguments have these values:



Because the conventions for calling **main()** are shared with C, C-style arrays and strings are used.

The idea is to read from the command string in the same way that we read from the input stream. A stream that reads from a string is unsurprisingly called an **istringstream** (§38.2.2). So to calculate expressions presented on the command line, we simply have to get our **Token_stream** to read from an appropriate **istringstream**:

```
Token_stream ts {cin};

int main(int argc, char* argv[])
{
        switch (argc) {
        case 1:                                         // read from standard input
                break;
        case 2:                                         // read from argument string
                ts.set_input(new istringstream{argv[1]});
                break;
        default:
                error("too many arguments");
                return 1;
        }

        table["pi"] = 3.1415926535897932385;            // insert predefined names
        table["e"] = 2.7182818284590452354;

        calculate();

        return no_of_errors;
}
```

To use an `istringstream`, include `<sstream>`.

It would be easy to modify `main()` to accept several command-line arguments, but this does not appear to be necessary, especially as several expressions can be passed as a single argument:

```
dc "rate=1.1934;150/rate;19.75/rate;217/rate"
```

I use quotes because `;` is the command separator on my UNIX systems. Other systems have different conventions for supplying arguments to a program on startup.

Simple as they are, `argc` and `argv` are still a source of minor, yet annoying, bugs. To avoid those and especially to make it easier to pass around the program arguments, I tend to use a simple function to create a `vector<string>`:

```
vector<string> arguments(int argc, char* argv[])
{
        vector<string> res;
        for (int i = 0; i!=argc; ++i)
                res.push_back(argv[i]);
        return res;
}
```

More elaborate argument parsing functions are not uncommon.

10.2.8 A Note on Style

To programmers unacquainted with associative arrays, the use of the standard-library `map` as the symbol table seems almost like cheating. It is not. The standard library and other libraries are meant to be used. Often, a library has received more care in its design and implementation than a

programmer could afford for a handcrafted piece of code to be used in just one program.

Looking at the code for the calculator, especially at the first version, we can see that there isn't much traditional C-style, low-level code presented. Many of the traditional tricky details have been replaced by uses of standard-library classes such as **ostream**, **string**, and **map** (§4.3.1, §4.2, §4.4.3, §31.4, Chapter 36, Chapter 38).

Note the relative scarcity of loops, arithmetic, and assignments. This is the way things ought to be in code that doesn't manipulate hardware directly or implement low-level abstractions.

10.3 Operator Summary

This section presents a summary of expressions and some examples. Each operator is followed by one or more names commonly used for it and an example of its use. In these tables:

- A *name* is an identifier (e.g., **sum** and **map**), an operator name (e.g., **operator int**, **operator+**, and **operator"" km**), or the name of a template specialization (e.g., **sort<Record>** and **array<int,10>**), possibly qualified using **::** (e.g., **std::vector** and **vector<T>::operator[]**).
- A *class-name* is the name of a class (including **decltype(expr)** where **expr** denotes a class).
- A *member* is a member name (including the name of a destructor or a member template).
- An *object* is an expression yielding a class object.
- A *pointer* is an expression yielding a pointer (including **this** and an object of that type that supports the pointer operation).
- An *expr* is an expression, including a literal (e.g., **17**, **"mouse"**, and **true**)
- An *expr-list* is a (possibly empty) list of expressions.
- An *lvalue* is an expression denoting a modifiable object (§6.4.1).
- A *type* can be a fully general type name (with *****, **()**, etc.) only when it appears in parentheses; elsewhere, there are restrictions (§iso.A).
- A *lambda-declarator* is a (possibly empty, comma-separated) list of parameters optionally followed by the **mutable** specifier, optionally followed by a **noexcept** specifier, optionally followed by a return type (§11.4).
- A *capture-list* is a (possibly empty) list specifying context dependencies (§11.4).
- A *stmt-list* is a (possibly empty) list of statements (§2.2.4, Chapter 9).

The syntax of expressions is independent of operand types. The meanings presented here apply when the operands are of built-in types (§6.2.1). In addition, you can define meanings for operators applied to operands of user-defined types (§2.3, Chapter 18).

A table can only approximate the rules of the grammar. For details, see §iso.5 and §iso.A.

Operator Summary (continues) (§iso.5.1)		
Parenthesized expression	**(** *expr* **)**	
Lambda	**[** *capture-list* **]** *lambda-declarator* **{** *stmt-List* **}**	§11.4
Scope resolution	*class-name* **::** *member*	§16.2.3
Scope resolution	*namespace-name* **::** *member*	§14.2.1
Global	**::** *name*	§14.2.1

Each box holds operators with the same precedence. Operators in higher boxes have higher precedence. For example, **N::x.m** means **(N::m).m** rather than the illegal **N::(x.m)**.

Operator Summary (continued, continues)		
Member selection	*object* **.** *member*	§16.2.3
Member selection	*pointer* **–>** *member*	§16.2.3
Subscripting	*pointer* **[** *expr* **]**	§7.3
Function call	*expr* **(** *expr-list* **)**	§12.2
Value construction	*type* **{** *expr-list* **}**	§11.3.2
Function-style type conversion	*type* **(** *expr-list* **)**	§11.5.4
Post increment	*lvalue* **++**	§11.1.4
Post decrement	*lvalue* **––**	§11.1.4
Type identification	**typeid (** *type* **)**	§22.5
Run-time type identification	**typeid (** *expr* **)**	§22.5
Run-time checked conversion	**dynamic_cast <** *type* **> (** *expr* **)**	§22.2.1
Compile-time checked conversion	**static_cast <** *type* **> (** *expr* **)**	§11.5.2
Unchecked conversion	**reinterpret_cast <** *type* **> (** *expr* **)**	§11.5.2
const conversion	**const_cast <** *type* **> (** *expr* **)**	§11.5.2
Size of object	**sizeof** *expr*	§6.2.8
Size of type	**sizeof (** *type* **)**	§6.2.8
Size of parameter pack	**sizeof...** *name*	§28.6.2
Alignment of type	**alignof (** *type* **)**	§6.2.9
Pre increment	**++** *lvalue*	§11.1.4
Pre decrement	**––** *lvalue*	§11.1.4
Complement	**~** *expr*	§11.1.2
Not	**!** *expr*	§11.1.1
Unary minus	**–** *expr*	§2.2.2
Unary plus	**+** *expr*	§2.2.2
Address of	**&** *lvalue*	§7.2
Dereference	***** *expr*	§7.2
Create (allocate)	**new** *type*	§11.2
Create (allocate and initialize)	**new** *type* **(** *expr-list* **)**	§11.2
Create (allocate and initialize)	**new** *type* **{** *expr-list* **}**	§11.2
Create (place)	**new (** *expr-list* **)** *type*	§11.2.4
Create (place and initialize)	**new (** *expr-list* **)** *type* **(** *expr-list* **)**	§11.2.4
Create (place and initialize)	**new (** *expr-list* **)** *type* **{** *expr-list* **}**	§11.2.4
Destroy (deallocate)	**delete** *pointer*	§11.2
Destroy array	**delete []** *pointer*	§11.2.2
Can expression throw?	**noexcept (** *expr* **)**	§13.5.1.2
Cast (type conversion)	**(** *type* **)** *expr*	§11.5.3
Member selection	*object* **.*** *pointer-to-member*	§20.6
Member selection	*pointer* **–>*** *pointer-to-member*	§20.6

For example, postfix **++** has higher precedence than unary *****, so ***p++** means ***(p++)**, *not* **(*p)++**.

Operator Summary (continued)		
Multiply	*expr* * *expr*	§10.2.1
Divide	*expr* / *expr*	§10.2.1
Modulo (remainder)	*expr* % *expr*	§10.2.1
Add (plus)	*expr* + *expr*	§10.2.1
Subtract (minus)	*expr* – *expr*	§10.2.1
Shift left	*expr* << *expr*	§11.1.2
Shift right	*expr* >> *expr*	§11.1.2
Less than	*expr* < *expr*	§2.2.2
Less than or equal	*expr* <= *expr*	§2.2.2
Greater than	*expr* > *expr*	§2.2.2
Greater than or equal	*expr* >= *expr*	§2.2.2
Equal	*expr* == *expr*	§2.2.2
Not equal	*expr* != *expr*	§2.2.2
Bitwise and	*expr* & *expr*	§11.1.2
Bitwise exclusive-or	*expr* ^ *expr*	§11.1.2
Bitwise inclusive-or	*expr* | *expr*	§11.1.2
Logical and	*expr* && *expr*	§11.1.1
Logical inclusive or	*expr* || *expr*	§11.1.1
Conditional expression	*expr* ? *expr* : *expr*	§11.1.3
List	{ *expr-list* }	§11.3
Throw exception	throw *expr*	§13.5
Simple assignment	*lvalue* = *expr*	§10.2.1
Multiply and assign	*lvalue* *= *expr*	§10.2.1
Divide and assign	*lvalue* /= *expr*	§10.2.1
Modulo and assign	*lvalue* %= *expr*	§10.2.1
Add and assign	*lvalue* += *expr*	§10.2.1
Subtract and assign	*lvalue* –= *expr*	§10.2.1
Shift left and assign	*lvalue* <<= *expr*	§10.2.1
Shift right and assign	*lvalue* >>= *expr*	§10.2.1
Bitwise and and assign	*lvalue* &= *expr*	§10.2.1
Bitwise inclusive-or and assign	*lvalue* |= *expr*	§10.2.1
Bitwise exclusive-or and assign	*lvalue* ^= *expr*	§10.2.1
comma (sequencing)	*expr* , *expr*	§10.3.2

For example: `a+b*c` means `a+(b*c)` rather than `(a+b)*c` because * has higher precedence than `+`.

Unary operators and assignment operators are right-associative; all others are left-associative. For example, `a=b=c` means `a=(b=c)` whereas `a+b+c` means `(a+b)+c`.

A few grammar rules cannot be expressed in terms of precedence (also known as binding strength) and associativity. For example, `a=b<c?d=e:f=g` means `a=((b<c)?(d=e):(f=g))`, but you need to look at the grammar (§iso.A) to determine that.

Before applying the grammar rules, lexical tokens are composed from characters. The longest possible sequence of characters is chosen to make a token. For example, `&&` is a single operator, rather than two `&` operators, and `a+++1` means `(a ++) + 1`. This is sometimes called the *Max Munch rule*.

Token Summary (§iso.2.7)		
Token Class	Examples	Reference
Identifier	`vector`, `foo_bar`, `x3`	§6.3.3
Keyword	`int`, `for`, `virtual`	§6.3.3.1
Character literal	`'x'`, `'\n'`, `'U'\UFADEFADE'`	§6.2.3.2
Integer literal	`12`, `012`, `0x12`	§6.2.4.1
Floating-point literal	`1.2`, `1.2e–3`, `1.2L`	§6.2.5.1
String literal	`"Hello!"`, `R"("World"!)"`	§7.3.2
Operator	`+=`, `%`, `<<`	§10.3
Punctuation	`;`, `,`, `{`, `}`, `(`, `)`	
Preprocessor notation	`#`, `##`	§12.6

Whitespace characters (e.g., space, tab, and newline) can be token separators (e.g., `int count` is a keyword followed by an identifier, rather than `intcount`) but are otherwise ignored.

Some characters from the basic source character set (§6.1.2), such as `|`, are not convenient to type on some keywords. Also, some programmers find it odd to use of symbols, such as `&&` and `~`, for basic logical operations. Consequently, a set of alternative representation are provided as keywords:

Alternative Representation (§iso.2.12)										
`and`	`and_eq`	`bitand`	`bitor`	`compl`	`not`	`not_eq`	`or`	`or_eq`	`xor`	`xor_eq`
`&`	`&=`	`&`	`|`	`~`	`!`	`!=`	`|`	`|=`	`^`	`^=`

For example

```
bool b =  not (x or y) and z;
int x4 = ~ (x1 bitor x2) bitand x3;
```

is equivalent to

```
bool b = !(x || y) && z;
int x4 = ~(x1 | x2) & x3;
```

Note that `and=` is not equivalent to `&=`; if you prefer keywords, you must write `and_eq`.

10.3.1 Results

The result types of arithmetic operators are determined by a set of rules known as ‘‘the usual arithmetic conversions’’ (§10.5.3). The overall aim is to produce a result of the ‘‘largest’’ operand type. For example, if a binary operator has a floating-point operand, the computation is done using floating-point arithmetic and the result is a floating-point value. Similarly, if it has a `long` operand, the computation is done using long integer arithmetic, and the result is a `long`. Operands that are smaller than an `int` (such as `bool` and `char`) are converted to `int` before the operator is applied.

The relational operators, **==**, **<=**, etc., produce Boolean results. The meaning and result type of user-defined operators are determined by their declarations (§18.2).

Where logically feasible, the result of an operator that takes an lvalue operand is an lvalue denoting that lvalue operand. For example:

```
void f(int x, int y)
{
    int j = x = y;          // the value of x=y is the value of x after the assignment
    int* p = &++x;          // p points to x
    int* q = &(x++);        // error: x++ is not an lvalue (it is not the value stored in x)
    int* p2 = &(x>y?x:y);   // address of the int with the larger value
    int& r = (x<y)?x:1;     // error: 1 is not an lvalue
}
```

If both the second and third operands of **?:** are lvalues and have the same type, the result is of that type and is an lvalue. Preserving lvalues in this way allows greater flexibility in using operators. This is particularly useful when writing code that needs to work uniformly and efficiently with both built-in and user-defined types (e.g., when writing templates or programs that generate C++ code).

The result of **sizeof** is of an unsigned integral type called **size_t** defined in **<cstddef>**. The result of pointer subtraction is of a signed integral type called **ptrdiff_t** defined in **<cstddef>**.

Implementations do not have to check for arithmetic overflow and hardly any do. For example:

```
void f()
{
    int i = 1;
    while (0 < i) ++i;
    cout << "i has become negative!" << i << '\n';
}
```

This will (eventually) try to increase **i** past the largest integer. What happens then is undefined, but typically the value ''wraps around'' to a negative number (on my machine **–2147483648**). Similarly, the effect of dividing by zero is undefined, but doing so usually causes abrupt termination of the program. In particular, underflow, overflow, and division by zero do not throw standard exceptions (§30.4.1.1).

10.3.2 Order of Evaluation

The order of evaluation of subexpressions within an expression is undefined. In particular, you cannot assume that the expression is evaluated left-to-right. For example:

```
int x = f(2)+g(3);          // undefined whether f() or g() is called first
```

Better code can be generated in the absence of restrictions on expression evaluation order. However, the absence of restrictions on evaluation order can lead to undefined results. For example:

```
int i = 1;
v[i] = i++; // undefined result
```

The assignment may be evaluated as either **v[1]=1** or **v[2]=1** or may cause some even stranger behavior. Compilers can warn about such ambiguities. Unfortunately, most do not, so be careful not to write an expression that reads or writes an object more than once, unless it does so using a single

operator that makes it well defined, such as `++` and `+=`, or explicitly express sequencing using `,` (comma), `&&`, or `||`.

The operators `,` (comma), `&&` (logical and), and `||` (logical or) guarantee that their left-hand operand is evaluated before their right-hand operand. For example, `b=(a=2,a+1)` assigns `3` to `b`. Examples of the use of `||` and `&&` can be found in §10.3.3. For built-in types, the second operand of `&&` is evaluated only if its first operand is `true`, and the second operand of `||` is evaluated only if its first operand is `false`; this is sometimes called *short-circuit evaluation*. Note that the sequencing operator `,` (comma) is logically different from the comma used to separate arguments in a function call. For example:

```
f1(v[i],i++);          // two arguments
f2( (v[i],i++) );      // one argument
```

The call of `f1` has two arguments, `v[i]` and `i++`, and the order of evaluation of the argument expressions is undefined. So it should be avoided. Order dependence of argument expressions is very poor style and has undefined behavior. The call of `f2` has only one argument, the comma expression `(v[i],i++)`, which is equivalent to `i++`. That is confusing, so that too should be avoided.

Parentheses can be used to force grouping. For example, `a*b/c` means `(a*b)/c`, so parentheses must be used to get `a*(b/c)`; `a*(b/c)` may be evaluated as `(a*b)/c` only if the user cannot tell the difference. In particular, for many floating-point computations `a*(b/c)` and `(a*b)/c` are significantly different, so a compiler will evaluate such expressions exactly as written.

10.3.3 Operator Precedence

Precedence levels and associativity rules reflect the most common usage. For example:

```
if (i<=0 || max<i) // ...
```

means ''if `i` is less than or equal to `0` or if `max` is less than `i`.'' That is, it is equivalent to

```
if ( (i<=0) || (max<i) ) // ...
```

and not the legal but nonsensical

```
if (i <= (0||max) < i) // ...
```

However, parentheses should be used whenever a programmer is in doubt about those rules. Use of parentheses becomes more common as the subexpressions become more complicated, but complicated subexpressions are a source of errors. Therefore, if you start feeling the need for parentheses, you might consider breaking up the expression by using an extra variable.

There are cases when the operator precedence does not result in the ''obvious'' interpretation. For example:

```
if (i&mask == 0)      // oops! == expression as operand for &
```

This does not apply a mask to `i` and then test if the result is zero. Because `==` has higher precedence than `&`, the expression is interpreted as `i&(mask==0)`. Fortunately, it is easy enough for a compiler to warn about most such mistakes. In this case, parentheses are important:

```
if ((i&mask) == 0) // ...
```

It is worth noting that the following does not work the way a mathematician might expect:

```
if (0 <= x <= 99) // ...
```

This is legal, but it is interpreted as `(0<=x)<=99`, where the result of the first comparison is either `true` or `false`. This Boolean value is then implicitly converted to `1` or `0`, which is then compared to `99`, yielding `true`. To test whether `x` is in the range `0..99`, we might use

```
if (0<=x && x<=99) // ...
```

A common mistake for novices is to use `=` (assignment) instead of `==` (equals) in a condition:

```
if (a = 7)   // oops! constant assignment in condition
```

This is natural because `=` means ''equals'' in many languages. Again, it is easy for a compiler to warn about most such mistakes – and many do. I do not recommend warping your style to compensate for compilers with weak warnings. In particular, I don't consider this style worthwhile:

```
if (7 == a) // try to protect against misuse of =; not recommended
```

10.3.4 Temporary Objects

Often, the compiler must introduce an object to hold an intermediate result of an expression. For example, for `v=x+y*z` the result of `y*z` has to be put somewhere before it is added to `x`. For built-in types, this is all handled so that a *temporary object* (often referred to as just a *temporary*) is invisible to the user. However, for a user-defined type that holds a resource knowing the lifetime of a temporary can be important. Unless bound to a reference or used to initialize a named object, a temporary object is destroyed at the end of the full expression in which it was created. A *full expression* is an expression that is not a subexpression of some other expression.

The standard-library `string` has a member `c_str()` (§36.3) that returns a C-style pointer to a zero-terminated array of characters (§2.2.5, §43.4). Also, the operator `+` is defined to mean string concatenation. These are useful facilities for `string`s. However, in combination they can cause obscure problems. For example:

```
void f(string& s1, string& s2, string& s3)
{
    const char* cs = (s1+s2).c_str();
    cout << cs;
    if (strlen(cs=(s2+s3).c_str())<8 && cs[0]=='a') {
        // cs used here
    }
}
```

Probably, your first reaction is ''But don't do that!'' and I agree. However, such code does get written, so it is worth knowing how it is interpreted.

A temporary `string` object is created to hold `s1+s2`. Next, a pointer to a C-style string is extracted from that object. Then – at the end of the expression – the temporary object is deleted. However, the C-style string returned by `c_str()` was allocated as part of the temporary object holding `s1+s2`, and that storage is not guaranteed to exist after that temporary is destroyed. Consequently, `cs` points to deallocated storage. The output operation `cout<<cs` might work as expected, but that would be sheer luck. A compiler can detect and warn against many variants of this problem.

The problem with the **if**-statement is a bit more subtle. The condition will work as expected because the full expression in which the temporary holding **s2+s3** is created is the condition itself. However, that temporary is destroyed before the controlled statement is entered, so any use of **cs** there is not guaranteed to work.

Please note that in this case, as in many others, the problems with temporaries arose from using a high-level data type in a low-level way. A cleaner programming style yields a more understandable program fragment and avoids the problems with temporaries completely. For example:

```
void f(string& s1, string& s2, string& s3)
{
        cout << s1+s2;
        string s = s2+s3;
        if (s.length()<8 && s[0]=='a') {
                // use s here
        }
}
```

A temporary can be used as an initializer for a **const** reference or a named object. For example:

```
void g(const string&, const string&);

void h(string& s1, string& s2)
{
        const string& s = s1+s2;
        string ss = s1+s2;

        g(s,ss);    // we can use s and ss here
}
```

This is fine. The temporary is destroyed when ''its'' reference or named object goes out of scope. Remember that returning a reference to a local variable is an error (§12.1.4) and that a temporary object cannot be bound to a non-**const** lvalue reference (§7.7).

A temporary object can also be created explicitly in an expression by invoking a constructor (§11.5.1). For example:

```
void f(Shape& s, int n, char ch)
{
        s.move(string{n,ch});      // construct a string with n copies of ch to pass to Shape::move()
        // ...
}
```

Such temporaries are destroyed in exactly the same way as the implicitly generated temporaries.

10.4 Constant Expressions

C++ offers two related meanings of ''constant'':

- **constexpr**: Evaluate at compile time (§2.2.3).
- **const**: Do not modify in this scope (§2.2.3, §7.5).

Basically, **constexpr**'s role is to enable and ensure compile-time evaluation, whereas **const**'s

primary role is to specify immutability in interfaces. This section is primarily concerned with the first role: compile-time evaluation.

A *constant expression* is an expression that a compiler can evaluate. It cannot use values that are not known at compile time and it cannot have side effects. Ultimately, a constant expression must start out with an integral value (§6.2.1), a floating-point value (§6.2.5), or an enumerator (§8.4), and we can combine those using operators and **constexpr** functions that in turn produce values. In addition, some addresses can be used in some forms of constant expressions. For simplicity, I discuss those separately in §10.4.5.

There are a variety of reasons why someone might want a named constant rather than a literal or a value stored in a variable:

[1] Named constants make the code easier to understand and maintain.

[2] A variable might be changed (so we have to be more careful in our reasoning than for a constant).

[3] The language requires constant expressions for array sizes, **case** labels, and **template** value arguments.

[4] Embedded systems programmers like to put immutable data into read-only memory because read-only memory is cheaper than dynamic memory (in terms of cost and energy consumption), and often more plentiful. Also, data in read-only memory is immune to most system crashes.

[5] If initialization is done at compile time, there can be no data races on that object in a multi-threaded system.

[6] Sometimes, evaluating something once (at compile time) gives significantly better performance than doing so a million times at run time.

Note that reasons [1], [2], [5], and (partly) [4] are logical. We don't just use constant expressions because of an obsession with performance. Often, the reason is that a constant expression is a more direct representation of our system requirements.

As part of the definition of a data item (here, I deliberately avoid the word "variable"), **constexpr** expresses the need for compile-time evaluation. If the initializer for a **constexpr** can't be evaluated at compile time, the compiler will give an error. For example:

```
int x1 = 7;
constexpr int x2 = 7;

constexpr int x3 = x1;          // error: initializer is not a constant expression
constexpr int x4 = x2;          // OK

void f()
{
    constexpr int y3 = x1;      // error: initializer is not a constant expression
    constexpr int y4 = x2;      // OK
    // ...
}
```

A clever compiler could deduce that the value of **x1** in the initializer for **x3** was **7**. However, we prefer not to rely on degrees of cleverness in compilers. In a large program, determining the values of variables at compile time is typically either very difficult or impossible.

The expressive power of constant expressions is great. We can use integer, floating-point, and enumeration values. We can use any operator that doesn't modify state (e.g., **+**, **?:**, and **[]**, but not **=** or **++**). We can use **constexpr** functions (§12.1.6) and literal types (§10.4.3) to provide a significant level of type safety and expressive power. It is almost unfair to compare this to what is commonly done with macros (§12.6).

The conditional-expression operator **?:** is the means of selection in a constant expression. For example, we can compute an integer square root at compile time:

```
constexpr int isqrt_helper(int sq, int d, int a)
{
    return sq <= a ? isqrt_helper(sq+d,d+2,a) : d;
}

constexpr int isqrt(int x)
{
    return isqrt_helper(1,3,x)/2 – 1;
}

constexpr int s1 = isqrt(9);          // s1 becomes 3
constexpr int s2 = isqrt(1234);
```

The condition of a **?:** is evaluated and then the selected alternative is evaluated. The alternative not selected is not evaluated and might even not be a constant expression. Similarly, operands of **&&** and **||** that are not evaluated need not be constant expressions. This feature is primarily useful in **constexpr** functions that are sometimes used as constant expressions and sometimes not.

10.4.1 Symbolic Constants

The most important single use of constants (**constexpr** or **const** values) is simply to provide symbolic names for values. Symbolic names should be used systematically to avoid ''magic numbers'' in code. Literal values scattered freely around in code is one of the nastiest maintenance hazards. If a numeric constant, such as an array bound, is repeated in code, it becomes hard to revise that code because every occurrence of that constant must be changed to update the code correctly. Using a symbolic name instead localizes information. Usually, a numeric constant represents an assumption about the program. For example, **4** may represent the number of bytes in an integer, **128** the number of characters needed to buffer input, and **6.24** the exchange factor between Danish kroner and U.S. dollars. Left as numeric constants in the code, these values are hard for a maintainer to spot and understand. Also, many such values need to change over time. Often, such numeric values go unnoticed and become errors when a program is ported or when some other change violates the assumptions they represent. Representing assumptions as well-commented named (symbolic) constants minimizes such maintenance problems.

10.4.2 **const**s in Constant Expressions

A **const** is primarily used to express interfaces (§7.5). However, **const** can also be used to express constant values. For example:

```
const int x = 7;
const string s = "asdf";
const int y = sqrt(x);
```

A **const** initialized with a constant expression can be used in a constant expression. A **const** differs from a **constexpr** in that it can be initialized by something that is not a constant expression; in that case, the **const** cannot be used as a constant expression. For example:

```
constexpr int xx = x;          // OK
constexpr string ss = s;       // error: s is not a constant expression
constexpr int yy = y;          // error: sqrt(x) is not a constant expression
```

The reasons for the errors are that **string** is not a literal type (§10.4.3) and **sqrt()** is not a **constexpr** function (§12.1.6).

Usually, **constexpr** is a better choice than **const** for defining simple constants, but **constexpr** is new in C++11, so older code tends to use **const**. In many cases, enumerators (§8.4) are another alternative to **const**s.

10.4.3 Literal Types

A sufficiently simple user-defined type can be used in a constant expression. For example:

```
struct Point {
    int x,y,z;
    constexpr Point up(int d) { return {x,y,z+d}; }
    constexpr Point move(int dx, int dy) { return {x+dx,y+dy}; }
    // ...
};
```

A class with a **constexpr** constructor is called a *literal type*. To be simple enough to be **constexpr**, a constructor must have an empty body and all members must be initialized by potentially constant expressions. For example:

```
constexpr Point origo {0,0};
constexpr int z = origo.x;

constexpr Point a[] = {
    origo, Point{1,1}, Point{2,2}, origo.move(3,3)
};
constexpr int x = a[1].x;              // x becomes 1

constexpr Point xy{0,sqrt(2)};         // error: sqrt(2) is not a constant expression
```

Note that we can have **constexpr** arrays and also access array elements and object members.

Naturally, we can define **constexpr** functions to take arguments of literal types. For example:

```
constexpr int square(int x)
{
    return x*x;
}
```

```
constexpr int radial_distance(Point p)
{
    return isqrt(square(p.x)+square(p.y)+square(p.z));
}

constexpr Point p1 {10,20,30};          // the default constructor is constexpr
constexpr p2 {p1.up(20)};               // Point::up() is constexpr
constexpr int dist = radial_distance(p2);
```

I used **int** rather than **double** just because I didn't have a **constexpr** floating-point square root function handy.

For a member function **constexpr** implies **const**, so I did not have to write:

```
constexpr Point move(int dx, int dy) const { return {x+dx,y+dy}; }
```

10.4.4 Reference Arguments

When working with **constexpr**, the key thing to remember is that **constexpr** is all about values. There are no objects that can change values or side effects here: **constexpr** provides a miniature compile-time functional programming language. That said, you might guess that **constexpr** cannot deal with references, but that's only partially true because **const** references refer to values and can therefore be used. Consider the specialization of the general **complex<T>** to a **complex<double>** from the standard library:

```
template<> class complex<double> {
public:
    constexpr complex(double re = 0.0, double im = 0.0);
    constexpr complex(const complex<float>&);
    explicit constexpr complex(const complex<long double>&);

    constexpr double real();        // read the real part
    void real(double);              // set the real part
    constexpr double imag();        // read the imaginary part
    void imag(double);              // set the imaginary part

    complex<double>& operator= (double);
    complex<double>& operator+=(double);
    // ...
};
```

Obviously, operations, such as **=** and **+=**, that modify an object cannot be **constexpr**. Conversely, operations that simply read an object, such as **real()** and **imag()**, can be **constexpr** and be evaluated at compile time given a constant expression. The interesting member is the template constructor from another **complex** type. Consider:

```
constexpr complex<float> z1 {1,2};          // note: <float> not <double>
constexpr double re = z1.real();
constexpr double im = z1.imag();
constexpr complex<double> z2 {re,im};       // z2 becomes a copy of z1
constexpr complex<double> z3 {z1};          // z3 becomes a copy of z1
```

The copy constructor works because the compiler recognizes that the reference (the **const complex<float>&**) refers to a constant value and we just use that value (rather than trying anything advanced or silly with references or pointers).

Literal types allow for type-rich compile-time programming. Traditionally, C++ compile-time evaluation has been restricted to using integer values (and without functions). This has resulted in code that was unnecessarily complicated and error-prone, as people encoded every kind of information as integers. Some uses of template metaprogramming (Chapter 28) are examples of that. Other programmers have simply preferred run-time evaluation to avoid the difficulties of writing in an impoverished language.

10.4.5 Address Constant Expressions

The address of a statically allocated object (§6.4.2), such as a global variable, is a constant. However, its value is assigned by the linker, rather than the compiler, so the compiler cannot know the value of such an address constant. That limits the range of constant expressions of pointer and reference type. For example:

```
constexpr const char* p1 = "asdf";
constexpr const char* p2 = p1;          // OK
constexpr const char* p2 = p1+2;        // error: the compiler does not know the value of p1
constexpr char c = p1[2];               // OK, c=='d'; the compiler knows the value pointed to by p1
```

10.5 Implicit Type Conversion

Integral and floating-point types (§6.2.1) can be mixed freely in assignments and expressions. Wherever possible, values are converted so as not to lose information. Unfortunately, some value-destroying (''narrowing'') conversions are also performed implicitly. A conversion is value-preserving if you can convert a value and then convert the result back to its original type and get the original value. If a conversion cannot do that, it is a *narrowing conversion* (§10.5.2.6). This section provides a description of conversion rules, conversion problems, and their resolution.

10.5.1 Promotions

The implicit conversions that preserve values are commonly referred to as *promotions*. Before an arithmetic operation is performed, *integral promotion* is used to create **int**s out of shorter integer types. Similarly, *floating-point promotion* is used to create **double**s out of **float**s. Note that these promotions will *not* promote to **long** (unless the operand is a **char16_t**, **char32_t**, **wchar_t**, or a plain enumeration that is already larger than an **int**) or **long double**. This reflects the original purpose of these promotions in C: to bring operands to the ''natural'' size for arithmetic operations.

The integral promotions are:

- A **char**, **signed char**, **unsigned char**, **short int**, or **unsigned short int** is converted to an **int** if **int** can represent all the values of the source type; otherwise, it is converted to an **unsigned int**.
- A **char16_t**, **char32_t**, **wchar_t** (§6.2.3), or a plain enumeration type (§8.4.2) is converted to the first of the following types that can represent all the values of its underlying type: **int**, **unsigned int**, **long**, **unsigned long**, or **unsigned long long**.

- A bit-field (§8.2.7) is converted to an **int** if **int** can represent all the values of the bit-field; otherwise, it is converted to **unsigned int** if **unsigned int** can represent all the values of the bit-field. Otherwise, no integral promotion applies to it.
- A **bool** is converted to an **int**; **false** becomes **0** and **true** becomes **1**.

Promotions are used as part of the usual arithmetic conversions (§10.5.3).

10.5.2 Conversions

The fundamental types can be implicitly converted into each other in a bewildering number of ways (§iso.4). In my opinion, too many conversions are allowed. For example:

```
void f(double d)
{
    char c = d;          // beware: double-precision floating-point to char conversion
}
```

When writing code, you should always aim to avoid undefined behavior and conversions that quietly throw away information (''narrowing conversions'').

A compiler can warn about many questionable conversions. Fortunately, many compilers do.

The **{}**-initializer syntax prevents narrowing (§6.3.5). For example:

```
void f(double d)
{
    char c {d};          // error: double-precision floating-point to char conversion
}
```

If potentially narrowing conversions are unavoidable, consider using some form of run-time checked conversion function, such as **narrow_cast<>()** (§11.5).

10.5.2.1 Integral Conversions

An integer can be converted to another integer type. A plain enumeration value can be converted to an integer type (§8.4.2) .

If the destination type is **unsigned**, the resulting value is simply as many bits from the source as will fit in the destination (high-order bits are thrown away if necessary). More precisely, the result is the least unsigned integer congruent to the source integer modulo **2** to the **n**th, where **n** is the number of bits used to represent the unsigned type. For example:

```
unsigned char uc = 1023;// binary 1111111111: uc becomes binary 11111111, that is, 255
```

If the destination type is **signed**, the value is unchanged if it can be represented in the destination type; otherwise, the value is implementation-defined:

```
signed char sc = 1023;    // implementation-defined
```

Plausible results are **127** and **–1** (§6.2.3).

A Boolean or plain enumeration value can be implicitly converted to its integer equivalent (§6.2.2, §8.4).

10.5.2.2 Floating-Point Conversions

A floating-point value can be converted to another floating-point type. If the source value can be exactly represented in the destination type, the result is the original numeric value. If the source value is between two adjacent destination values, the result is one of those values. Otherwise, the behavior is undefined. For example:

```
float f = FLT_MAX;          // largest float value
double d = f;               // OK: d == f

double d2 = DBL_MAX;        // largest double value
float f2 = d2;              // undefined if FLT_MAX<DBL_MAX

long double ld = d2;        // OK: ld = d3
long double ld2 = numeric_limits<long double>::max();
double d3 = ld2;            // undefined if sizeof(long double)>sizeof(double)
```

DBL_MAX and **FLT_MAX** are defined in **<climits>**; **numeric_limits** is defined in **<limits>** (§40.2).

10.5.2.3 Pointer and Reference Conversions

Any pointer to an object type can be implicitly converted to a **void*** (§7.2.1). A pointer (reference) to a derived class can be implicitly converted to a pointer (reference) to an accessible and unambiguous base (§20.2). Note that a pointer to function or a pointer to member cannot be implicitly converted to a **void***.

A constant expression (§10.4) that evaluates to **0** can be implicitly converted to a null pointer of any pointer type. Similarly, a constant expression that evaluates to **0** can be implicitly converted to a pointer-to-member type (§20.6). For example:

```
int* p = (1+2)*(2*(1–1));   // OK, but weird
```

Prefer **nullptr** (§7.2.2).

A **T*** can be implicitly converted to a **const T*** (§7.5). Similarly, a **T&** can be implicitly converted to a **const T&**.

10.5.2.4 Pointer-to-Member Conversions

Pointers and references to members can be implicitly converted as described in §20.6.3.

10.5.2.5 Boolean Conversions

Pointer, integral, and floating-point values can be implicitly converted to **bool** (§6.2.2). A nonzero value converts to **true**; a zero value converts to **false**. For example:

```
void f(int* p, int i)
{
    bool is_not_zero = p;       // true if p!=0
    bool b2 = i;                // true if i!=0
    // ...
}
```

The pointer-to-**bool** conversion is useful in conditions, but confusing elsewhere:

```
void fi(int);
void fb(bool);

void ff(int* p, int* q)
{
    if (p) do_something(*p);          // OK
    if (q!=nullptr) do_something(*q); // OK, but verbose
    // ...
    fi(p);                            // error: no pointer to int conversion
    fb(p);                            // OK: pointer to bool conversion (surprise!?)
}
```

Hope for a compiler warning for **fb(p)**.

10.5.2.6 Floating-Integral Conversions

When a floating-point value is converted to an integer value, the fractional part is discarded. In other words, conversion from a floating-point type to an integer type truncates. For example, the value of **int(1.6)** is **1**. The behavior is undefined if the truncated value cannot be represented in the destination type. For example:

```
int i = 2.7;        // i becomes 2
char b = 2000.7;    // undefined for 8-bit chars: 2000 cannot be represented as an 8-bit char
```

Conversions from integer to floating types are as mathematically correct as the hardware allows. Loss of precision occurs if an integral value cannot be represented exactly as a value of the floating type. For example:

```
int i = float(1234567890);
```

On a machine where both **int**s and **float**s are represented using 32 bits, the value of **i** is **1234567936**.

Clearly, it is best to avoid potentially value-destroying implicit conversions. In fact, compilers can detect and warn against some obviously dangerous conversions, such as floating to integral and **long int** to **char**. However, general compile-time detection is impractical, so the programmer must be careful. When ''being careful'' isn't enough, the programmer can insert explicit checks. For example:

```
char checked_cast(int i)
{
    char c = i;     // warning: not portable (§10.5.2.1)
    if (i != c) throw std::runtime_error{"int–to–char check failed"};
    return c;
}

void my_code(int i)
{
    char c = checked_cast(i);
    // ...
}
```

A more general technique for expressing checked conversions is presented in §25.2.5.1.

To truncate in a way that is guaranteed to be portable requires the use of **numeric_limits** (§40.2). In initializations, truncation can be avoided by using the **{}**-initializer notation (§6.3.5).

10.5.3 Usual Arithmetic Conversions

These conversions are performed on the operands of a binary operator to bring them to a common type, which is then used as the type of the result:

[1] If either operand is of type **long double**, the other is converted to **long double**.
- Otherwise, if either operand is **double**, the other is converted to **double**.
- Otherwise, if either operand is **float**, the other is converted to **float**.
- Otherwise, integral promotions (§10.5.1) are performed on both operands.

[2] Otherwise, if either operand is **unsigned long long**, the other is converted to **unsigned long long**.
- Otherwise, if one operand is a **long long int** and the other is an **unsigned long int**, then if a **long long int** can represent all the values of an **unsigned long int**, the **unsigned long int** is converted to a **long long int**; otherwise, both operands are converted to **unsigned long long int**. Otherwise, if either operand is **unsigned long long**, the other is converted to **unsigned long long**.
- Otherwise, if one operand is a **long int** and the other is an **unsigned int**, then if a **long int** can represent all the values of an **unsigned int**, the **unsigned int** is converted to a **long int**; otherwise, both operands are converted to **unsigned long int**.
- Otherwise, if either operand is **long**, the other is converted to **long**.
- Otherwise, if either operand is **unsigned**, the other is converted to **unsigned**.
- Otherwise, both operands are **int**.

These rules make the result of converting an unsigned integer to a signed one of possibly larger size implementation-defined. That is yet another reason to avoid mixing unsigned and signed integers.

10.6 Advice

[1] Prefer the standard library to other libraries and to ‘‘handcrafted code’’; §10.2.8.
[2] Use character-level input only when you have to; §10.2.3.
[3] When reading, always consider ill-formed input; §10.2.3.
[4] Prefer suitable abstractions (classes, algorithms, etc.) to direct use of language features (e.g., **int**s, statements); §10.2.8.
[5] Avoid complicated expressions; §10.3.3.
[6] If in doubt about operator precedence, parenthesize; §10.3.3.
[7] Avoid expressions with undefined order of evaluation; §10.3.2.
[8] Avoid narrowing conversions; §10.5.2.
[9] Define symbolic constants to avoid ‘‘magic constants’’; §10.4.1.
[10] Avoid narrowing conversions; §10.5.2.

This page intentionally left blank

11

Select Operations

When someone says
"I want a programming language in which
I need only say what I wish done,"
give him a lollipop.
– Alan Perlis

- Etc. Operators
  Logical Operators; Bitwise Logical Operators; Conditional Expressions; Increment and Decrement
- Free Store
  Memory Management; Arrays; Getting Memory Space; Overloading **new**
- Lists
  Implementation Model; Qualified Lists; Unqualified Lists
- Lambda Expressions
  Implementation Model; Alternatives to Lambdas; Capture; Call and Return; The Type of a Lambda
- Explicit Type Conversion
  Construction; Named Casts; C-Style Cast; Function-Style Cast
- Advice

11.1 Etc. Operators

This section examines a mixed bag of simple operators: logical operators (**&&**, **||**, and **!**), bitwise logical operators (**&**, **|**, **~**, **<<**, and **>>**), conditional expressions (**?:**), and increment and decrement operators (**++** and **--**). They have little in common beyond their details not fitting elsewhere in the discussions of operators.

11.1.1 Logical Operators

The logical operators **&&** (and), **||** (or), and **!** (not) take operands of arithmetic and pointer types, convert them to **bool**, and return a **bool** result. The **&&** and **||** operators evaluate their second argument only if necessary, so they can be used to control evaluation order (§10.3.2). For example:

```
while (p && !whitespace(*p)) ++p;
```

Here, **p** is not dereferenced if it is the **nullptr**.

11.1.2 Bitwise Logical Operators

The bitwise logical operators **&** (and), **|** (or), **^** (exclusive or, xor), **~** (complement), **>>** (right shift), and **<<** (left shift) are applied to objects of integral types – that is, **char**, **short**, **int**, **long**, **long long** and their **unsigned** counterparts, and **bool**, **wchar_t**, **char16_t**, and **char32_t**. A plain **enum** (but not an **enum class**) can be implicitly converted to an integer type and used as an operand to bitwise logical operations. The usual arithmetic conversions (§10.5.3) determine the type of the result.

A typical use of bitwise logical operators is to implement the notion of a small set (a bit vector). In this case, each bit of an unsigned integer represents one member of the set, and the number of bits limits the number of members. The binary operator **&** is interpreted as intersection, **|** as union, **^** as symmetric difference, and **~** as complement. An enumeration can be used to name the members of such a set. Here is a small example borrowed from an implementation of **ostream**:

```
enum ios_base::iostate {
        goodbit=0, eofbit=1, failbit=2, badbit=4
};
```

The implementation of a stream can set and test its state like this:

```
state = goodbit;
// ...
if (state&(badbit|failbit)) // stream not good
```

The extra parentheses are necessary because **&** has higher precedence than **|** (§10.3).

A function that reaches the end-of-input might report it like this:

```
state |= eofbit;
```

The **|=** operator is used to add to the state. A simple assignment, **state=eofbit**, would have cleared all other bits.

These stream state flags are observable from outside the stream implementation. For example, we could see how the states of two streams differ like this:

```
int old = cin.rdstate();      // rdstate() returns the state
// ... use cin ...
if (cin.rdstate()^old) {      // has anything changed?
        // ...
}
```

Computing differences of stream states is not common. For other similar types, computing differences is essential. For example, consider comparing a bit vector that represents the set of interrupts being handled with another that represents the set of interrupts waiting to be handled.

Please note that this bit fiddling is taken from the implementation of iostreams rather than from the user interface. Convenient bit manipulation can be very important, but for reliability, maintainability, portability, etc., it should be kept at low levels of a system. For more general notions of a set, see the standard-library **set** (§31.4.3) and **bitset** (§34.2.2).

Bitwise logical operations can be used to extract bit-fields from a word. For example, one could extract the middle 16 bits of a 32-bit **int** like this:

```
constexpr unsigned short middle(int a)
{
        static_assert(sizeof(int)==4,"unexpected int size");
        static_assert(sizeof(short)==2,"unexpected short size");
        return (a>>8)&0xFFFF;
}

int x = 0xFF00FF00; // assume sizeof(int)==4
short y = middle(x); // y = 0x00FF
```

Using fields (§8.2.7) is a convenient shorthand for such shifting and masking.

Do not confuse the bitwise logical operators with the logical operators: **&&**, **||**, and **!**. The latter return **true** or **false**, and they are primarily useful for writing the test in an **if**-, **while**-, or **for**-statement (§9.4, §9.5). For example, **!0** (not zero) is the value **true**, which converts to **1**, whereas **~0** (complement of zero) is the bit pattern all-ones, which in two's complement representation is the value **–1**.

11.1.3 Conditional Expressions

Some **if**-statements can conveniently be replaced by *conditional-expression*s. For example:

```
if (a <= b)
        max = b;
else
        max = a;
```

This is more directly expressed like this:

```
max = (a<=b) ? b : a;
```

The parentheses around the condition are not necessary, but I find the code easier to read when they are used.

Conditional expressions are important in that they can be used in constant expressions (§10.4).

A pair of expressions **e1** and **e2** can be used as alternatives in a conditional expression, **c?e1:e2**, if they are of the same type or if there is a common type **T**, to which they can both be implicitly converted. For arithmetic types, the usual arithmetic conversions (§10.5.3) are used to find that common type. For other types, either **e1** must be implicitly convertible to **e2**'s type or vice versa. In addition, one branch may be a **throw**-expression (§13.5.1). For example:

```
void fct(int* p)
{
        int i = (p) ? *p : std::runtime_error{"unexpected nullptr};
        // ...
}
```

11.1.4 Increment and Decrement

The ++ operator is used to express incrementing directly, rather than expressing it indirectly using a combination of an addition and an assignment. Provided lvalue has no side effects, ++lvalue means lvalue+=1, which again means lvalue=lvalue+1. The expression denoting the object to be incremented is evaluated once (only). Decrementing is similarly expressed by the -- operator.

The operators ++ and -- can be used as both prefix and postfix operators. The value of ++x is the new (that is, incremented) value of x. For example, y=++x is equivalent to y=(x=x+1). The value of x++, however, is the old value of x. For example, y=x++ is equivalent to y=(t=x,x=x+1,t), where t is a variable of the same type as x.

Like adding an int to a pointer, or subtracting it, ++ and -- on a pointer operate in terms of elements of the array into which the pointer points; p++ makes p point to the next element (§7.4.1).

The ++ and -- operators are particularly useful for incrementing and decrementing variables in loops. For example, one can copy a zero-terminated C-style string like this:

```
void cpy(char* p, const char* q)
{
        while (*p++ = *q++) ;
}
```

Like C, C++ is both loved and hated for enabling such terse, expression-oriented coding. Consider:

```
while (*p++ = *q++) ;
```

This is more than a little obscure to non-C programmers, but because the style of coding is not uncommon, it is worth examining more closely. Consider first a more traditional way of copying an array of characters:

```
int length = strlen(q);
for (int i = 0; i<=length; i++)
        p[i] = q[i];
```

This is wasteful. The length of a zero-terminated string is found by reading the string looking for the terminating zero. Thus, we read the string twice: once to find its length and once to copy it. So we try this instead:

```
int i;
for (i = 0; q[i]!=0 ; i++)
        p[i] = q[i];
p[i] = 0;           // terminating zero
```

The variable i used for indexing can be eliminated because p and q are pointers:

```
while (*q != 0) {
        *p = *q;
        p++;        // point to next character
        q++;        // point to next character
}
*p = 0;             // terminating zero
```

Because the post-increment operation allows us first to use the value and then to increment it, we can rewrite the loop like this:

```
while (*q != 0) {
        *p++ = *q++;
}
*p = 0; // terminating zero
```

The value of `*p++ = *q++` is `*q`. We can therefore rewrite the example like this:

```
while ((*p++ = *q++) != 0) { }
```

In this case, we don't notice that `*q` is zero until we already have copied it into `*p` and incremented `p`. Consequently, we can eliminate the final assignment of the terminating zero. Finally, we can reduce the example further by observing that we don't need the empty block and that the `!=0` is redundant because the result of an integral condition is always compared to zero anyway. Thus, we get the version we set out to discover:

```
while (*p++ = *q++) ;
```

Is this version less readable than the previous versions? Not to an experienced C or C++ programmer. Is this version more efficient in time or space than the previous versions? Except for the first version that called `strlen()`, not really; the performance will be equivalent and often identical code will be generated.

The most efficient way of copying a zero-terminated character string is typically the standard C-style string copy function:

```
char* strcpy(char*, const char*);    // from <string.h>
```

For more general copying, the standard `copy` algorithm (§4.5, §32.5) can be used. Whenever possible, use standard-library facilities in preference to fiddling with pointers and bytes. Standard-library functions may be inlined (§12.1.3) or even implemented using specialized machine instructions. Therefore, you should measure carefully before believing that some piece of handcrafted code outperforms library functions. Even if it does, the advantage may not exist on some other handware+compiler combination, and your alternative may give a maintainer a headache.

11.2 Free Store

A named object has its lifetime determined by its scope (§6.3.4). However, it is often useful to create an object that exists independently of the scope in which it was created. For example, it is common to create objects that can be used after returning from the function in which they were created. The operator `new` creates such objects, and the operator `delete` can be used to destroy them. Objects allocated by `new` are said to be ''on the *free store*'' (also, ''on the *heap*'' or ''in *dynamic memory*'').

Consider how we might write a compiler in the style used for the desk calculator (§10.2). The syntax analysis functions might build a tree of the expressions for use by the code generator:

```
struct Enode {
        Token_value oper;
        Enode* left;
        Enode* right;
        // ...
};
```

```
Enode* expr(bool get)
{
    Enode* left = term(get);

    for (;;) {
        switch (ts.current().kind) {
        case Kind::plus:
        case Kind::minus:
            left = new Enode {ts.current().kind,left,term(true)};
            break;
        default:
            return left;                    // return node
        }
    }
}
```

In cases `Kind::plus` and `Kind::minus`, a new `Enode` is created on the free store and initialized by the value `{ts.current().kind,left,term(true)}`. The resulting pointer is assigned to `left` and eventually returned from `expr()`.

I used the `{}`-list notation for specifying arguments. Alternatively, I could have used the old-style `()`-list notation to specify an initializer. However, trying the `=` notation for initializing an object created using `new` results in an error:

```
int* p = new int = 7; // error
```

If a type has a default constructor, we can leave out the initializer, but built-in types are by default uninitialized. For example:

```
auto pc = new complex<double>;    // the complex is initialized to {0,0}
auto pi = new int;                // the int is uninitialized
```

This can be confusing. To be sure to get default initialization, use `{}`. For example:

```
auto pc = new complex<double>{}; // the complex is initialized to {0,0}
auto pi = new int{};             // the int is initialized to 0
```

A code generator could use the `Enode`s created by `expr()` and delete them:

```
void generate(Enode* n)
{
    switch (n–>oper) {
    case Kind::plus:
        // use n
        delete n;  // delete an Enode from the free store
    }
}
```

An object created by `new` exists until it is explicitly destroyed by `delete`. Then, the space it occupied can be reused by `new`. A C++ implementation does not guarantee the presence of a ‘‘garbage collector’’ that looks out for unreferenced objects and makes them available to `new` for reuse. Consequently, I will assume that objects created by `new` are manually freed using `delete`.

The **delete** operator may be applied only to a pointer returned by **new** or to the **nullptr**. Applying **delete** to the **nullptr** has no effect.

If the deleted object is of a class with a destructor (§3.2.1.2, §17.2), that destructor is called by **delete** before the object's memory is released for reuse.

11.2.1 Memory Management

The main problems with free store are:

- *Leaked objects*: People use **new** and then forget to **delete** the allocated object.
- *Premature deletion*: People **delete** an object that they have some other pointer to and later use that other pointer.
- *Double deletion*: An object is deleted twice, invoking its destructor (if any) twice.

Leaked objects are potentially a bad problem because they can cause a program to run out of space. Premature deletion is almost always a nasty problem because the pointer to the ''deleted object'' no longer points to a valid object (so reading it may give bad results) and may indeed point to memory that has been reused for another object (so writing to it may corrupt an unrelated object). Consider this example of very bad code:

```
int* p1 = new int{99};
int* p2 = p1;                   // potential trouble
delete p1;                      // now p2 doesn't point to a valid object
p1 = nullptr;                   // gives a false sense of safety
char* p3 = new char{'x'};       // p3 may now point to the memory pointed to by p2
*p2 = 999;                      // this may cause trouble
cout << *p3 << '\n';            // may not print x
```

Double deletion is a problem because resource managers typically cannot track what code owns a resource. Consider:

```
void sloppy()   // very bad code
{
    int* p = new int[1000];     // acquire memory
    // ... use *p ...
    delete[] p;                 // release memory

    // ... wait a while ...

    delete[] p;                 // but sloppy() does not own *p
}
```

By the second **delete[]**, the memory pointed to by ***p** may have been reallocated for some other use and the allocator may get corrupted. Replace **int** with **string** in that example, and we'll see **string**'s destructor trying to read memory that has been reallocated and maybe overwritten by other code, and using what it read to try to **delete** memory. In general, a double deletion is undefined behavior and the results are unpredictable and usually disastrous.

The reason people make these mistakes is typically not maliciousness and often not even simple sloppiness; it is genuinely hard to consistently deallocate every allocated object in a large program (once and at exactly the right point in a computation). For starters, analysis of a localized part of a program will not detect these problems because an error usually involves several separate parts.

As alternatives to using ''naked'' **new**s and **delete**s, I can recommend two general approaches to resource management that avoid such problems:

[1] Don't put objects on the free store if you don't have to; prefer scoped variables.

[2] When you construct an object on the free store, place its pointer into a *manager object* (sometimes called a *handle*) with a destructor that will destroy it. Examples are **string**, **vector** and all the other standard-library containers, **unique_ptr** (§5.2.1, §34.3.1), and **shared_ptr** (§5.2.1, §34.3.2). Wherever possible, have that manager object be a scoped variable. Many classical uses of free store can be eliminated by using move semantics (§3.3, §17.5.2) to return large objects represented as manager objects from functions.

This rule [2] is often referred to as RAII (''Resource Acquisition Is Initialization''; §5.2, §13.3) and is the basic technique for avoiding resource leaks and making error handling using exceptions simple and safe.

The standard-library **vector** is an example of these techniques:

```
void f(const string& s)
{
    vector<char> v;
    for (auto c : s)
        v.push_back(c);
    // ...
}
```

The **vector** keeps its elements on the free store, but it handles all allocations and deallocations itself. In this example, **push_back()** does **new**s to acquire space for its elements and **delete**s to free space that it no longer needs. However, the users of **vector** need not know about those implementation details and will just rely on **vector** not leaking.

The **Token_stream** from the calculator example is an even simpler example (§10.2.2). There, a user can use **new** and hand the resulting pointer to a **Token_stream** to manage:

```
Token_stream ts{new istringstream{some_string}};
```

We do not need to use the free store just to get a large object out of a function. For example:

```
string reverse(const string& s)
{
    string ss;
    for (int i=s.size()–1; 0<=i; ––i)
        ss.push_back(s[i]);
    return ss;
}
```

Like **vector**, a **string** is really a handle to its elements. So, we simply *move* the **ss** out of **reverse()** rather than copying any elements (§3.3.2).

The resource management ''smart pointers'' (e.g., **unique_ptr** and **smart_ptr**) are a further example of these ideas (§5.2.1, §34.3.1). For example:

```
void f(int n)
{
    int* p1 = new int[n];                    // potential trouble
    unique_ptr<int[]> p2 {new int[n]};
```

```
        // ...
        if (n%2) throw runtime_error("odd");
        delete[] p1;                        // we may never get here
}
```

For `f(3)` the memory pointed to by `p1` is leaked, but the memory pointed to by `p2` is correctly and implicitly deallocated.

My rule of thumb for the use of `new` and `delete` is ''no naked `new`s''; that is, `new` belongs in constructors and similar operations, `delete` belongs in destructors, and together they provide a coherent memory management strategy. In addition, `new` is often used in arguments to resource handles.

If everything else fails (e.g., if someone has a lot of old code with lots of undisciplined use of `new`), C++ offers a standard interface to a garbage collector (§34.5).

11.2.2 Arrays

Arrays of objects can also be created using `new`. For example:

```
char* save_string(const char* p)
{
        char* s = new char[strlen(p)+1];
        strcpy(s,p);            // copy from p to s
        return s;
}

int main(int argc, char* argv[])
{
        if (argc < 2) exit(1);
        char* p = save_string(argv[1]);
        // ...
        delete[] p;
}
```

The ''plain'' operator `delete` is used to delete individual objects; `delete[]` is used to delete arrays.

Unless you really must use a `char*` directly, the standard-library `string` can be used to simplify the `save_string()`:

```
string save_string(const char* p)
{
        return string{p};
}

int main(int argc, char* argv[])
{
        if (argc < 2) exit(1);
        string s = save_string(argv[1]);
        // ...
}
```

In particular, the `new[]` and the `delete[]` vanished.

To deallocate space allocated by **new**, **delete** and **delete[]** must be able to determine the size of the object allocated. This implies that an object allocated using the standard implementation of **new** will occupy slightly more space than a static object. At a minimum, space is needed to hold the object's size. Usually two or more words per allocation are used for free-store management. Most modern machines use 8-byte words. This overhead is not significant when we allocate many objects or large objects, but it can matter if we allocate lots of small objects (e.g., **int**s or **Point**s) on the free store.

Note that a **vector** (§4.4.1, §31.4) is a proper object and can therefore be allocated and deallocated using plain **new** and **delete**. For example:

```
void f(int n)
{
    vector<int>* p = new vector<int>(n);    // individual object
    int* q = new int[n];                    // array
    // ...
    delete p;
    delete[] q;
}
```

The **delete[]** operator may be applied only to a pointer to an array returned by **new** of an array or to the null pointer (§7.2.2). Applying **delete[]** to the null pointer has no effect.

However, do not use **new** to create local objects. For example:

```
void f1()
{
    X* p =new X;
    // ... use *p ...
    delete p;
}
```

That's verbose, inefficient, and error-prone (§13.3). In particular, a **return** or an exception thrown before the **delete** will cause a memory leak (unless even more code is added). Instead, use a local variable:

```
void f2()
{
    X x;
    // ... use x ...
}
```

The local variable **x** is implicitly destroyed upon exit from **f2**.

11.2.3 Getting Memory Space

The free-store operators **new**, **delete**, **new[]**, and **delete[]** are implemented using functions presented in the **<new>** header:

```
void* operator new(size_t);          // allocate space for individual object
void operator delete(void* p);       // if (p) deallocate space allocated using operator new()
```

```
void* operator new[](size_t);          // allocate space for array
void operator delete[](void* p);       // if (p) deallocate space allocated using operator new[]()
```

When operator **new** needs to allocate space for an object, it calls **operator new()** to allocate a suitable number of bytes. Similarly, when operator **new** needs to allocate space for an array, it calls **operator new[]()**.

The standard implementations of **operator new()** and **operator new[]()** do not initialize the memory returned.

The allocation and deallocation functions deal in untyped and uninitialized memory (often called ''raw memory''), as opposed to typed objects. Consequently, they take arguments or return values of type **void***. The operators **new** and **delete** handle the mapping between this untyped-memory layer and the typed-object layer.

What happens when **new** can find no store to allocate? By default, the allocator throws a standard-library **bad_alloc** exception (for an alternative, see §11.2.4.1). For example:

```
void f()
{
    vector<char*> v;
    try {
        for (;;) {
            char * p = new char[10000];   // acquire some memory
            v.push_back(p);               // make sure the new memory is referenced
            p[0] = 'x';                   // use the new memory
        }
    }
    catch(bad_alloc) {
        cerr << "Memory exhausted!\n";
    }
}
```

However much memory we have available, this will eventually invoke the **bad_alloc** handler. Please be careful: the **new** operator is not guaranteed to throw when you run out of physical main memory. So, on a system with virtual memory, this program can consume a lot of disk space and take a long time doing so before the exception is thrown.

We can specify what **new** should do upon memory exhaustion; see §30.4.1.3.

In addition to the functions defined in **<new>**, a user can define **operator new()**, etc., for a specific class (§19.2.5). Class members **operator new()**, etc., are found and used in preference to the ones from **<new>** according to the usual scope rules.

11.2.4 Overloading new

By default, operator **new** creates its object on the free store. What if we wanted the object allocated elsewhere? Consider a simple class:

```
class X {
public:
    X(int);
    // ...
};
```

We can place objects anywhere by providing an allocator function (§11.2.3) with extra arguments and then supplying such extra arguments when using **new**:

```
void* operator new(size_t, void* p) { return p; }     // explicit placement operator

void* buf = reinterpret_cast<void*>(0xF00F);          // significant address
X* p2 = new(buf) X;                                   // construct an X at buf;
                                                      // invokes: operator new(sizeof(X),buf)
```

Because of this usage, the **new(buf) X** syntax for supplying extra arguments to **operator new()** is known as the *placement syntax*. Note that every **operator new()** takes a size as its first argument and that the size of the object allocated is implicitly supplied (§19.2.5). The **operator new()** used by the **new** operator is chosen by the usual argument matching rules (§12.3); every **operator new()** has a **size_t** as its first argument.

The ''placement'' **operator new()** is the simplest such allocator. It is defined in the standard header **<new>**:

```
void* operator new (size_t sz, void* p) noexcept;       // place object of size sz at p
void* operator new[](size_t sz, void* p) noexcept;      // place object of size sz at p

void operator delete (void* p, void*) noexcept;         // if (p) make *p invalid
void operator delete[](void* p, void*) noexcept;        // if (p) make *p invalid
```

The ''placement **delete**'' operators do nothing except possibly inform a garbage collector that the **delete**d pointer is no longer safely derived (§34.5).

The placement **new** construct can also be used to allocate memory from a specific arena:

```
class Arena {
public:
        virtual void* alloc(size_t) =0;
        virtual void free(void*) =0;
        // ...
};

void* operator new(size_t sz, Arena* a)
{
        return a–>alloc(sz);
}
```

Now objects of arbitrary types can be allocated from different **Arena**s as needed. For example:

```
extern Arena* Persistent;
extern Arena* Shared;

void g(int i)
{
        X* p = new(Persistent) X(i);    // X in persistent storage
        X* q = new(Shared) X(i);        // X in shared memory
        // ...
}
```

Placing an object in an area that is not (directly) controlled by the standard free-store manager implies that some care is required when destroying the object. The basic mechanism for that is an explicit call of a destructor:

```
void destroy(X* p, Arena* a)
{
    p->~X();        // call destructor
    a->free(p);     // free memory
}
```

Note that explicit calls of destructors should be avoided except in the implementation of resource management classes. Even most resource handles can be written using **new** and **delete**. However, it would be hard to implement an efficient general container along the lines of the standard-library **vector** (§4.4.1, §31.3.3) without using explicit destructor calls. A novice should think thrice before calling a destructor explicitly and also should ask a more experienced colleague before doing so.

See §13.6.1 for an example of how placement **new** can interact with exception handling.

There is no special syntax for placement of arrays. Nor need there be, since arbitrary types can be allocated by placement **new**. However, an **operator delete()** can be defined for arrays (§11.2.3).

11.2.4.1 nothrow new

In programs where exceptions must be avoided (§13.1.5), we can use **nothrow** versions of **new** and **delete**. For example:

```
void f(int n)
{
    int* p = new(nothrow) int[n];       // allocate n ints on the free store
    if (p==nullptr) {// no memory available
        // ... handle allocation error ...
    }
    // ...
    operator delete(nothrow,p);         // deallocate *p
}
```

That **nothrow** is the name of an object of the standard-library type **nothrow_t** that is used for disambiguation; **nothrow** and **nothrow_t** are declared in **<new>**.

The functions implementing this are found in **<new>**:

```
void* operator new(size_t sz, const nothrow_t&) noexcept;   // allocate sz bytes;
                                                            // return nullptr if allocation failed
void operator delete(void* p, const nothrow_t&) noexcept;   // deallocate space allocated by new

void* operator new[](size_t sz, const nothrow_t&) noexcept; // allocate sz bytes;
                                                            // return nullptr if allocation failed
void operator delete[](void* p, const nothrow_t&) noexcept; // deallocate space allocated by new
```

These **operator new** functions return **nullptr**, rather than throwing **bad_alloc**, if there is not sufficient memory to allocate.

11.3 Lists

In addition to their use for initializing named variables (§6.3.5.2), {}-lists can be used as expressions in many (but not all) places. They can appear in two forms:

[1] Qualified by a type, T{...}, meaning "create an object of type T initialized by T{...}"; §11.3.2

[2] Unqualified {...}, for which the the type must be determined from the context of use; §11.3.3

For example:

```
struct S { int a, b; };
struct SS { double a, b; };

void f(S);      // f() takes an S

void g(S);
void g(SS);     // g() is overloaded

void h()
{
    f({1,2});       // OK: call f(S{1,2})

    g({1,2});       // error: ambiguous
    g(S{1,2});      // OK: call g(S)
    g(SS{1,2});     // OK: call g(SS)
}
```

As in their use for initializing named variables (§6.3.5), lists can have zero, one, or more elements. A {}-list is used to construct an object of some type, so the number of elements and their types must be what is required to construct an object of that type.

11.3.1 Implementation Model

The implementation model for {}-lists comes in three parts:

- If the {}-list is used as constructor arguments, the implementation is just as if you had used a ()-list. List elements are not copied except as by-value constructor arguments.
- If the {}-list is used to initialize the elements of an aggregate (an array or a class without a constructor), each list element initializes an element of the aggregate. List elements are not copied except as by-value arguments to aggregate element constructors.
- If the {}-list is used to construct an initializer_list object each list element is used to initialize an element of the *underlying array* of the initializer_list. Elements are typically copied from the initializer_list to wherever we use them.

Note that this is the general model that we can use to understand the semantics of a {}-list; a compiler may apply clever optimizations as long as the meaning is preserved.

Consider:

```
vector<double> v = {1, 2, 3.14};
```

The standard-library vector has an initializer-list constructor (§17.3.4), so the initializer list

`{1,2,3.14}` is interpreted as a temporary constructed and used like this:

```
const double temp[] = {double{1}, double{2}, 3.14 } ;
const initializer_list<double> tmp(temp,sizeof(temp)/sizeof(double));
vector<double> v(tmp);
```

That is, the compiler constructs an array containing the initializers converted to the desired type (here, `double`). This array is passed to `vector`s initializer-list constructor as an `initializer_list`. The initializer-list constructor then copies the values from the array into its own data structure for elements. Note that an `initializer_list` is a small object (probably two words), so passing it by value makes sense.

The underlying array is immutable, so there is no way (within the standard's rules) that the meaning of a `{}`-list can change between two uses. Consider:

```
void f()
{
    initializer_list<int> lst {1,2,3};
    cout << *lst.begin() << '\n';
    *lst.begin() = 2;                   // error: lst is immutable
    cout << *lst.begin() << '\n';
}
```

In particular, having a `{}`-list be immutable implies that a container taking elements from it must use a copy operation, rather than a move operation.

The lifetime of a `{}`-list (and its underlying array) is determined by the scope in which it is used (§6.4.2). When used to initialize a variable of type `initializer_list<T>`, the list lives as long as the variable. When used in an expression (including as an initializer to a variable of some other type, such as `vector<T>`), the list is destroyed at the end of its full expression.

11.3.2 Qualified Lists

The basic idea of initializer lists as expressions is that if you can initialize a variable `x` using the notation

```
T x {v};
```

then you can create an object with the same value as an expression using `T{v}` or `new T{v}`. Using `new` places the object on the free store and returns a pointer to it, whereas "plain `T{v}`" makes a temporary object in the local scope (§6.4.2). For example:

```
struct S { int a, b; };

void f()
{
    S v {7,8};          // direct initialization of a variable
    v = S{7,8};         // assign using qualified list
    S* p = new S{7,8};  // construct on free store using qualified list
}
```

The rules constructing an object using a qualified list are those of direct initialization (§16.2.6).

One way of looking at a qualified initializer list with one element is as a conversion from one type to another. For example:

```
template<class T>
T square(T x)
{
        return x*x;
}

void f(int i)
{
        double d = square(double{i});
        complex<double> z = square(complex<double>{i});
}
```

That idea is explored further in §11.5.1.

11.3.3 Unqualified Lists

A unqualified list is used where an expected type is unambiguously known. It can be used as an expression only as:

- A function argument
- A return value
- The right-hand operand of an assignment operator (=, +=, *=, etc.)
- A subscript

For example:

```
int f(double d, Matrix& m)
{
        int v {7};              // initializer (direct initialization)
        int v2 = {7};           // initializer (copy initialization)
        int v3 = m[{2,3}];      // assume m takes value pairs as subscripts

        v = {8};                // right-hand operand of assignment
        v += {88};              // right-hand operand of assignment
        {v} = 9;                // error: not left-hand operand of assignment
        v = 7+{10};             // error: not an operand of a non-assignment operator
        f({10.0});              // function argument
        return {11};            // return value
}
```

The reason that an unqualified list is not allowed on the left-hand side of assignments is primarily that the C++ grammar allows { in that position for compound statements (blocks), so that readability would be a problem for humans and ambiguity resolution would be tricky for compilers. This is not an insurmountable problem, but it was decided not to extend C++ in that direction.

When used as the initializer for a named object without the use of a = (as for v above), an unqualified {}-list performs direct initialization (§16.2.6). In all other cases, it performs copy initialization (§16.2.6). In particular, the otherwise redundant = in an initializer restricts the set of initializations that can be performed with a given {}-list.

The standard-library type `initializer_list<T>` is used to handle variable-length `{}`-lists (§12.2.3). Its most obvious use is to allow initializer lists for user-defined containers (§3.2.1.3), but it can also be used directly; for example:

```
int high_value(initializer_list<int> val)
{
    int high = numeric_traits<int>lowest();
    if (val.size()==0) return high;

    for (auto x : val)
        if (x>high) high = x;

    return high;
}

int v1 = high_value({1,2,3,4,5,6,7});
int v2 = high_value({–1,2,v1,4,–9,20,v1});
```

A `{}`-list is the simplest way of dealing with homogeneous lists of varying lengths. However, beware that zero elements can be a special case. If so, that case should be handled by a default constructor (§17.3.3).

The type of a `{}`-list can be deduced (only) if all elements are of the same type. For example:

```
auto x0 = {};       // error (no element type)
auto x1 = {1};      // initializer_list<int>
auto x2 = {1,2};    // initializer_list<int>
auto x3 = {1,2,3};  // initializer_list<int>
auto x4 = {1,2.0};  // error: nonhomogeneous list
```

Unfortunately, we do not deduce the type of an unqualified list for a plain template argument. For example:

```
template<typename T>
void f(T);

f({});          // error: type of initializer is unknown
f({1});         // error: an unqualified list does not match “plain T”
f({1,2});       // error: an unqualified list does not match “plain T”
f({1,2,3});     // error: an unqualified list does not match “plain T”
```

I say ‘‘unfortunately’’ because this is a language restriction, rather than a fundamental rule. It would be technically possible to deduce the type of those `{}`-lists as `initializer_list<int>`, just like we do for `auto` initializers.

Similarly, we do not deduce the element type of a container represented as a template. For example:

```
template<class T>
void f2(const vector<T>&);

f2({1,2,3});                // error: cannot deduce T
f2({"Kona","Sidney"});      // error: cannot deduce T
```

This too is unfortunate, but it is a bit more understandable from a language-technical point of view: nowhere in those calls does it say **vector**. To deduce **T** the compiler would first have to decide that the user really wanted a **vector** and then look into the definition of **vector** to see if it has a constructor that accepts **{1,2,3}**. In general, that would require an instantiation of **vector** (§26.2). It would be possible to handle that, but it could be costly in compile time, and the opportunities for ambiguities and confusion if there were many overloaded versions of **f2()** are reasons for caution. To call **f2()**, be more specific:

```
f2(vector<int>{1,2,3});                 // OK
f2(vector<string>{"Kona","Sidney"});    // OK
```

11.4 Lambda Expressions

A *lambda expression*, sometimes also referred to as a *lambda function* or (strictly speaking incorrectly, but colloquially) as a *lambda*, is a simplified notation for defining and using an anonymous function object. Instead of defining a named class with an **operator()**, later making an object of that class, and finally invoking it, we can use a shorthand. This is particularly useful when we want to pass an operation as an argument to an algorithm. In the context of graphical user interfaces (and elsewhere), such operations are often referred to as *callbacks*. This section focuses on technical aspects of lambdas; examples and techniques for the use of lambdas can be found elsewhere (§3.4.3, §32.4, §33.5.2).

A lambda expression consists of a sequence of parts:

- A possibly empty *capture list*, specifying what names from the definition environment can be used in the lambda expression's body, and whether those are copied or accessed by reference. The capture list is delimited by **[]** (§11.4.3).
- An optional *parameter list*, specifying what arguments the lambda expression requires. The parameter list is delimited by **()** (§11.4.4).
- An optional **mutable** specifier, indicating that the lambda expression's body may modify the state of the lambda (i.e., change the lambda's copies of variables captured by value) (§11.4.3.4).
- An optional **noexcept** specifier.
- An optional return type declaration of the form **–>** *type* (§11.4.4).
- A *body*, specifying the code to be executed. The body is delimited by **{}** (§11.4.3).

The details of passing arguments, returning results, and specifying the body are those of functions and are presented in Chapter 12. The notion of ''capture'' of local variables is not provided for functions. This implies that a lambda can act as a local function even though a function cannot.

11.4.1 Implementation Model

Lambda expressions can be implemented in a variety of ways, and there are some rather effective ways of optimizing them. However, I find it useful to understand the semantics of a lambda by considering it a shorthand for defining and using a function object. Consider a relatively simple example:

```
void print_modulo(const vector<int>& v, ostream& os, int m)
    // output v[i] to os if v[i]%m==0
{
    for_each(begin(v),end(v),
        [&os,m](int x) { if (x%m==0) os << x << '\n'; }
    );
}
```

To see what this means, we can define the equivalent function object:

```
class Modulo_print {
    ostream& os;   // members to hold the capture list
    int m;
public:
    Modulo_print(ostream& s, int mm) :os(s), m(mm) {}       // capture
    void operator()(int x) const
        { if (x%m==0) os << x << '\n'; }
};
```

The capture list, **[&os,m]**, becomes two member variables and a constructor to initialize them. The **&** before **os** means that we should store a reference, and the absence of a **&** for **m** means that we should store a copy. This use of **&** mirrors its use in function argument declarations.

The body of the lambda simply becomes the body of the **operator()()**. Since the lambda doesn't return a value, the **operator()()** is **void**. By default, **operator()()** is **const**, so that the lambda body doesn't modify the captured variables. That's by far the most common case. Should you want to modify the state of a lambda from its body, the lambda can be declared **mutable** (§11.4.3.4). This corresponds to an **operator()()** *not* being declared **const**.

An object of a class generated from a lambda is called a *closure object* (or simply a *closure*). We can now write the original function like this:

```
void print_modulo(const vector<int>& v, ostream& os, int m)
    // output v[i] to os if v[i]%m==0
{
    for_each(begin(v),end(v),Modulo_print{os,m});
}
```

If a lambda potentially captures every local variable by reference (using the capture list **[&]**), the closure may be optimized to simply contain a pointer to the enclosing stack frame.

11.4.2 Alternatives to Lambdas

That final version of **print_modulo()** is actually quite attractive, and naming nontrivial operations is generally a good idea. A separately defined class also leaves more room for comments than does a lambda embedded in some argument list.

However, many lambdas are small and used only once. For such uses, the realistic equivalent involves a local class defined immediately before its (only) use. For example:

```
void print_modulo(const vector<int>& v, ostream& os, int m)
    // output v[i] to os if v[i]%m==0
{
    class Modulo_print {
        ostream& os;   // members to hold the capture list
        int m;
    public:
        Modulo_print (ostream& s, int mm) :os(s), m(mm) {}      // capture
        void operator()(int x) const
            { if (x%m==0) os << x << '\n'; }
    };

    for_each(begin(v),end(v),Modulo_print{os,m});
}
```

Compared to that, the version using the lambda is a clear winner. If we really want a name, we can just name the lambda:

```
void print_modulo(const vector<int>& v, ostream& os, int m)
    // output v[i] to os if v[i]%m==0
{
    auto Modulo_print = [&os,m] (int x) { if (x%m==0) os << x << '\n'; };

    for_each(begin(v),end(v),Modulo_print);
}
```

Naming the lambda is often a good idea. Doing so forces us to consider the design of the operation a bit more carefully. It also simplifies code layout and allows for recursion (§11.4.5).

Writing a **for**-loop is an alternative to using a lambda with a **for_each()**. Consider:

```
void print_modulo(const vector<int>& v, ostream& os, int m)
    // output v[i] to os if v[i]%m==0
{
    for (auto x : v)
        if (x%m==0) os << x << '\n';
}
```

Many would find this version much clearer than any of the lambda versions. However, **for_each** is a rather special algorithm, and **vector<int>** is a very specific container. Consider generalizing **print_modulo()** to handle arbitrary containers:

```
template<class C>
void print_modulo(const C& v, ostream& os, int m)
    // output v[i] to os if v[i]%m==0
{
    for (auto x : v)
        if (x%m==0) os << x << '\n';
}
```

This version works nicely for a **map**. The C++ range-**for**-statement specifically caters to the special case of traversing a sequence from its beginning to its end. The STL containers make such

traversals easy and general. For example, using a **for**-statement to traverse a **map** gives a depth-first traversal. How would we do a breadth-first traversal? The **for**-loop version of **print_modulo()** is not amenable to change, so we have to rewrite it to an algorithm. For example:

```
template<class C>
void print_modulo(const C& v, ostream& os, int m)
    // output v[i] to os if v[i]%m==0
{
    breadth_first(begin(v),end(v),
        [&os,m](int x) { if (x%m==0) os << x << '\n'; }
    );
}
```

Thus, a lambda can be used as ''the body'' for a generalized loop/traversal construct represented as an algorithm. Using **for_each** rather than **breadth_first** would give depth-first traversal.

The performance of a lambda as an argument to a traversal algorithm is equivalent (typically identical) to that of the equivalent loop. I have found that to be quite consistent across implementations and platforms. The implication is that we have to base our choice between ''algorithm plus lambda'' and ''**for**-statement with body'' on stylistic grounds and on estimates of extensibility and maintainability.

11.4.3 Capture

The main use of lambdas is for specifying code to be passed as arguments. Lambdas allow that to be done ''inline'' without having to name a function (or function object) and use it elsewhere. Some lambdas require no access to their local environment. Such lambdas are defined with the empty lambda introducer **[]**. For example:

```
void algo(vector<int>& v)
{
    sort(v.begin(),v.end());    // sort values
    // ...
    sort(v.begin(),v.end(),[](int x, int y) { return abs(x)<abs(y); });        // sort absolute values
    // ...
}
```

If we want to access local names, we have to say so or get an error:

```
void f(vector<int>& v)
{
    bool sensitive = true;
    // ...
    sort(v.begin(),v.end(),
        [](int x, int y) { return sensitive ? x<y : abs(x)<abs(y); }    // error: can't access sensitive
    );
}
```

I used the *lambda introducer* **[]**. This is the simplest lambda introducer and does not allow the lambda to refer to names in the calling environment. The first character of a lambda expression is always **[**. A lambda introducer can take various forms:

- **[]**: an empty capture list. This implies that no local names from the surrounding context can be used in the lambda body. For such lambda expressions, data is obtained from arguments or from nonlocal variables.
- **[&]**: implicitly capture by reference. All local names can be used. All local variables are accessed by reference.
- **[=]**: implicitly capture by value. All local names can be used. All names refer to copies of the local variables taken at the point of call of the lambda expression.
- **[***capture-list***]**: explicit capture; the *capture-list* is the list of names of local variables to be captured (i.e., stored in the object) by reference or by value. Variables with names preceded by **&** are captured by reference. Other variables are captured by value. A capture list can also contain **this** and names followed by **...** as elements.
- **[&,** *capture-list***]**: implicitly capture by reference all local variables with names not mentioned in the list. The capture list can contain **this**. Listed names cannot be preceded by **&**. Variables named in the capture list are captured by value.
- **[=,** *capture-list***]**: implicitly capture by value all local variables with names not mentioned in the list. The capture list cannot contain **this**. The listed names must be preceded by **&**. Variables named in the capture list are captured by reference.

Note that a local name preceded by **&** is always captured by reference and a local name not preceded by **&** is always captured by value. Only capture by reference allows modification of variables in the calling environment.

The *capture-list* cases are used for fine-grained control over what names from the call environment are used and how. For example:

```
void f(vector<int>& v)
{
    bool sensitive = true;
    // ...
    sort(v.begin(),v.end()
        [sensitive](int x, int y) { return sensitive ? x<y : abs(x)<abs(y); }
    );
}
```

By mentioning **sensitive** in the capture list, we make it accessible from within the lambda. By not specifying otherwise, we ensure that the capture of **sensitive** is done ''by value''; just as for argument passing, passing a copy is the default. Had we wanted to capture **sensitive** ''by reference,'' we could have said so by adding a **&** before **sensitive** in the capture list: **[&sensitive]**.

The choice between capturing by value and by reference is basically the same as the choice for function arguments (§12.2). We use a reference if we need to write to the captured object or if it is large. However, for lambdas, there is the added concern that a lambda might outlive its caller (§11.4.3.1). When passing a lambda to another thread, capturing by value (**[=]**) is typically best: accessing another thread's stack through a reference or a pointer can be most disruptive (to performance or correctness), and trying to access the stack of a terminated thread can lead to extremely difficult-to-find errors.

If you need to capture a variadic template (§28.6) argument, use **...**. For example:

```
template<typename... Var>
void algo(int s, Var... v)
{
    auto helper = [&s,&v...] { return s*(h1(v...)+h2(v...)); }
    // ...
}
```

Beware that is it easy to get too clever about capture. Often, there is a choice between capture and argument passing. When that's the case, capture is usually the least typing but has the greatest potential for confusion.

11.4.3.1 Lambda and Lifetime

A lambda might outlive its caller. This can happen if we pass a lambda to a different thread or if the callee stores away the lambda for later use. For example:

```
void setup(Menu& m)
{
    // ...
    Point p1, p2, p3;
    // compute positions of p1, p2, and p3
    m.add("draw triangle",[&]{ m.draw(p1,p2,p3); });    // probable disaster
    // ...
}
```

Assuming that `add()` is an operation that adds a (name,action) pair to a menu and that the `draw()` operation makes sense, we are left with a time bomb: the `setup()` completes and later – maybe minutes later – a user presses the `draw triangle` button and the lambda tries to access the long-gone local variables. A lambda that wrote to a variable caught by reference would be even worse in that situation.

If a lambda might outlive its caller, we must make sure that all local information (if any) is copied into the closure object and that values are returned through the `return` mechanism (§12.1.4) or through suitable arguments. For the `setup()` example, that is easily done:

```
m.add("draw triangle",[=]{ m.draw(p1,p2,p3); });
```

Think of the capture list as the initializer list for the closure object and `[=]` and `[&]` as short-hand notation (§11.4.1).

11.4.3.2 Namespace Names

We don't need to ''capture'' namespace variables (including global variables) because they are always accessible (provided they are in scope). For example:

```
template<typename U, typename V>
ostream& operator<<(ostream& os, const pair<U,V>& p)
{
    return os << '{' << p.first << ',' << p.second << '}';
}
```

```
void print_all(const map<string,int>& m, const string& label)
{
    cout << label << ":\n{\n";
    for_each(m.begin(),m.end(),
        [](const pair<string,int>& p) { cout << p << '\n'; }
    );
    cout << "}\n";
}
```

Here, we don't need to capture **cout** or the output operator for **pair**.

11.4.3.3 Lambda and this

How do we access members of a class object from a lambda used in a member function? We can include class members in the set of names potentially captured by adding **this** to the capture list. This is used when we want to use a lambda in the implementation of a member function. For example, we might have a class for building up requests and retrieving results:

```
class Request {
    function<map<string,string>(const map<string,string>&)> oper;    // operation
    map<string,string> values;          // arguments
    map<string,string> results;         // targets
public:
    Request(const string& s);           // parse and store request

    void execute()
    {
        [this]() { results=oper(values); }    // do oper to values yielding results
    }
};
```

Members are always captured by reference. That is, **[this]** implies that members are accessed through **this** rather than copied into the lambda. Unfortunately, **[this]** and **[=]** are incompatible. This implies that incautious use can lead to race conditions in multi-threaded programs (§42.4.6).

11.4.3.4 mutable Lambdas

Usually, we don't want to modify the state of the function object (the closure), so by default we can't. That is, the **operator()()** for the generated function object (§11.4.1) is a **const** member function. In the unlikely event that we want to modify the state (as opposed to modifying the state of some variable captured by reference; §11.4.3), we can declare the lambda **mutable**. For example:

```
void algo(vector<int>& v)
{
    int count = v.size();
    std::generate(v.begin(),v.end(),
        [count]()mutable{ return --count; }
    );
}
```

The **--count** decrements the copy of **v**'s size stored in the closure.

11.4.4 Call and Return

The rules for passing arguments to a lambda are the same as for a function (§12.2), and so are the rules for returning results (§12.1.4). In fact, with the exception of the rules for capture (§11.4.3) most rules for lambdas are borrowed from the rules for functions and classes. However, two irregularities should be noted:

[1] If a lambda expression does not take any arguments, the argument list can be omitted. Thus, the minimal lambda expression is `[]{}`.

[2] A lambda expression's return type can be deduced from its body. Unfortunately, that is not also done for a function.

If a lambda body does not have a `return`-statement, the lambda's return type is `void`. If a lambda body consists of just a single `return`-statement, the lambda's return type is the type of the `return`'s expression. If neither is the case, we have to explicitly supply a return type. For example:

```
void g(double y)
{
    [&]{ f(y); }                                  // return type is void
    auto z1 = [=](int x){ return x+y; }           // return type is double
    auto z2 = [=,y]{ if (y) return 1; else return 2; }  // error: body too complicated
                                                  // for return type deduction
    auto z3 =[y]() { return 1 : 2; }              // return type is int
    auto z4 = [=,y]()->int { if (y) return 1; else return 2; }  // OK: explicit return type
}
```

When the suffix return type notation is used, we cannot omit the argument list.

11.4.5 The Type of a Lambda

To allow for optimized versions of lambda expressions, the type of a lambda expression is not defined. However, it is defined to be the type of a function object in the style presented in §11.4.1. This type, called the *closure type*, is unique to the lambda, so no two lambdas have the same type. Had two lambdas had the same type, the template instantiation mechanism might have gotten confused. A lambda is of a local class type with a constructor and a `const` member function `operator()()`. In addition to using a lambda as an argument, we can use it to initialize a variable declared `auto` or `std::function<R(AL)>` where `R` is the lambda's return type and `AL` is its argument list of types (§33.5.3).

For example, I might try to write a lambda to reverse the characters in a C-style string:

```
auto rev = [&rev](char* b, char* e)
                { if (1<e–b) { swap(*b,*––e); rev(++b,e); } };          // error
```

However, that's not possible because I cannot use an `auto` variable before its type has been deduced. Instead, I can introduce a name and then use it:

```
void f(string& s1, string& s2)
{
    function<void(char* b, char* e)> rev =
        [&](char* b, char* e) { if (1<e–b) { swap(*b,*––e); rev(++b,e); } };
```

```
        rev(&s1[0],&s1[0]+s1.size());
        rev(&s2[0],&s2[0]+s2.size());
}
```

Now, the type of **rev** is specified before it is used.

If we just want to name a lambda, rather than using it recursively, **auto** can simplify things:

```
void g(vector<string>& vs1, vector<string>& vs2)
{
        auto rev = [&](char* b, char* e) { while (1<e–b) swap(*b++,*––e); };

        rev(&s1[0],&s1[0]+s1.size());
        rev(&s2[0],&s2[0]+s2.size());
}
```

A lambda that captures nothing can be assigned to a pointer to function of an appropriate type. For example:

```
double (*p1)(double) = [](double a) { return sqrt(a); };
double (*p2)(double) = [&](double a) { return sqrt(a); };      // error: the lambda captures
double (*p3)(int) = [](int a) { return sqrt(a); };             // error: argument types do not match
```

11.5 Explicit Type Conversion

Sometimes, we have to convert a value of one type into a value of another. Many (arguably too many) such conversions are done implicitly according to the language rules (§2.2.2, §10.5). For example:

```
double d = 1234567890;  // integer to floating-point
int i = d;              // floating-point to integer
```

In other cases, we have to be explicit.

For logical and historical reasons, C++ offers explicit type conversion operations of varying convenience and safety:

- Construction, using the **{}** notation, providing type-safe construction of new values (§11.5.1)
- Named conversions, providing conversions of various degrees of nastiness:
  - **const_cast** for getting write access to something declared **const** (§7.5)
  - **static_cast** for reversing a well-defined implicit conversion (§11.5.2)
  - **reinterpret_cast** for changing the meaning of bit patterns (§11.5.2)
  - **dynamic_cast** for dynamically checked class hierarchy navigation (§22.2.1)
- C-style casts, providing any of the named conversions and some combinations of those (§11.5.3)
- Functional notation, providing a different notation for C-style casts (§11.5.4)

I have ordered these conversions in my order of preference and safety of use.

Except for the **{}** construction notation, I can't say I like any of those, but at least **dynamic_cast** is run-time checked. For conversion between two scalar numeric types, I tend to use a homemade explicit conversion function, **narrow_cast**, where a value might be narrowed:

```
template<class Target, class Source>
Target narrow_cast(Source v)
{
    auto r = static_cast<Target>(v);        // convert the value to the target type
    if (static_cast<Source>(r)!=v)
        throw runtime_error("narrow_cast<>() failed");
    return r;
}
```

That is, if I can convert a value to the target type, convert the result back to the source type, and get back the original value, I'm happy with the result. That is a generalization of the rule the language applies to values in `{}` initialization (§6.3.5.2). For example:

```
void test(double d, int i, char* p)
{
    auto c1 = narrow_cast<char>(64);
    auto c2 = narrow_cast<char>(–64);       // will throw if chars are unsigned
    auto c3 = narrow_cast<char>(264);       // will throw if chars are 8-bit and signed

    auto d1 = narrow_cast<double>(1/3.0F); // OK
    auto f1 = narrow_cast<float>(1/3.0);    // will probably throw

    auto c4 = narrow_cast<char>(i);         // may throw
    auto f2 = narrow_cast<float>(d);        // may throw

    auto p1 = narrow_cast<char*>(i);        // compile-time error
    auto i1 = narrow_cast<int>(p);          // compile-time error

    auto d2 = narrow_cast<double>(i);       // may throw (but probably will not)
    auto i2 = narrow_cast<int>(d);          // may throw
}
```

Depending on your use of floating-point numbers, it may be worthwhile to use a range test for floating-point conversions, rather than `!=`. That is easily done using specializations (§25.3.4.1) or type traits (§35.4.1).

11.5.1 Construction

The construction of a value of type `T` from a value `e` can be expressed by the notation `T{e}` (§iso.8.5.4). For example:

```
auto d1 = double{2};        // d1==2.0
double d2 {double{2}/4};    // d1==0.5
```

Part of the attraction of the `T{v}` notation is that it will perform only "well-behaved" conversions. For example:

```
void f(int);
void f(double);

void g(int i, double d)
{
    f(i);                           // call f(int)
    f(double{i});                   // error: {} doesn't do int to floating conversion

    f(d);                           // call f(double)
    f(int{d});                      // error: {} doesn't truncate
    f(static_cast<int>(d));         // call f(int) with a truncated value

    f(round(d));                    // call f(double) with a rounded value
    f(static_cast<int>(lround(d))); // call f(int) with a rounded value
                                    // if the d is overflows the int, this still truncates
}
```

I don't consider truncation of floating-point numbers (e.g., **7.9** to **7**) ''well behaved,'' so having to be explicit when you want it is a good thing. If rounding is desirable, we can use the standard-library function **round()**; it performs ''conventional 4/5 rounding,'' such as **7.9** to **8** and **7.4** to **7**.

It sometimes comes as a surprise that **{}**-construction doesn't allow **int** to **double** conversion, but if (as is not uncommon) the size of an **int** is the same as the size of a **double**, then some such conversions must lose information. Consider:

```
static_assert(sizeof(int)==sizeof(double),"unexpected sizes");

int x = numeric_limits<int>::max(); // largest possible integer
double d = x;
int y = x;
```

We will not get **x==y**. However, we can still initialize a **double** with an integer literal that can be represented exactly. For example:

```
double d { 1234 };    // fine
```

Explicit qualification with the desired type does not enable ill-behaved conversions. For example:

```
void g2(char* p)
{
    int x = int{p};         // error: no char* to int conversion
    using Pint = int*;
    int* p2 = Pint{p};      // error: no char* to int* conversion
    // ...
}
```

For **T{v}**, ''reasonably well behaved'' is defined as having a ''non-narrowing'' (§10.5) conversion from **v** to **T** or having an appropriate constructor for **T** (§17.3).

The constructor notation **T{}** is used to express the default value of type **T**. For example:

```
template<class T> void f(const T&);

void g3()
{
        f(int{});                       // default int value
        f(complex<double>{});   // default complex value
        // ...
}
```

The value of an explicit use of the constructor for a built-in type is `0` converted to that type (§6.3.5). Thus, `int{}` is another way of writing `0`. For a user-defined type `T`, `T{}` is defined by the default constructor (§3.2.1.1, §17.6), if any, otherwise by default construction, `MT{}`, of each member.

Explicitly constructed unnamed objects are temporary objects, and (unless bound to a reference) their lifetime is limited to the full expression in which they are used (§6.4.2). In this, they differ from unnamed objects created using `new` (§11.2).

11.5.2 Named Casts

Some type conversions are not well behaved or easy to type check; they are not simple constructions of values from a well-defined set of argument values. For example:

```
IO_device* d1 = reinterpret_cast<IO_device*>(0Xff00);  // device at 0Xff00
```

There is no way a compiler can know whether the integer `0Xff00` is a valid address (of an I/O device register). Consequently, the correctness of the conversions is completely in the hands of the programmer. Explicit type conversion, often called *casting*, is occasionally essential. However, traditionally it is seriously overused and a major source of errors.

Another classical example of the need for explicit type conversion is dealing with ''raw memory,'' that is, memory that holds or will hold objects of a type not known to the compiler. For example, a memory allocator (such as `operator new()`; §11.2.3) may return a `void*` pointing to newly allocated memory:

```
void* my_allocator(size_t);

void f()
{
        int* p = static_cast<int*>(my_allocator(100));          // new allocation used as ints
        // ...
}
```

A compiler does not know the type of the object pointed to by the `void*`.

The fundamental idea behind the named casts is to make type conversion more visible and to allow the programmer to express the intent of a cast:

- `static_cast` converts between related types such as one pointer type to another in the same class hierarchy, an integral type to an enumeration, or a floating-point type to an integral type. It also does conversions defined by constructors (§16.2.6, §18.3.3, §iso.5.2.9) and conversion operators (§18.4).

- **reinterpret_cast** handles conversions between unrelated types such as an integer to a pointer or a pointer to an unrelated pointer type (§iso.5.2.10).
- **const_cast** converts between types that differ only in **const** and **volatile** qualifiers (§iso.5.2.11).
- **dynamic_cast** does run-time checked conversion of pointers and references into a class hierarchy (§22.2.1, §iso.5.2.7).

These distinctions among the named casts allow the compiler to apply some minimal type checking and make it easier for a programmer to find the more dangerous conversions represented as **reinterpret_cast**s. Some **static_cast**s are portable, but few **reinterpret_cast**s are. Hardly any guarantees are made for **reinterpret_cast**, but generally it produces a value of a new type that has the same bit pattern as its argument. If the target has at least as many bits as the original value, we can **reinterpret_cast** the result back to its original type and use it. The result of a **reinterpret_cast** is guaranteed to be usable only if its result is converted back to the exact original type. Note that **reinterpret_cast** is the kind of conversion that must be used for pointers to functions (§12.5). Consider:

```
char x = 'a';
int* p1 = &x;                          // error: no implicit char* to int* conversion
int* p2 = static_cast<int*>(&x);       // error: no implicit char* to int* conversion
int* p3 = reinterpret_cast<int*>(&x);  // OK: on your head be it

struct B { /* ... */ };
struct D : B { /* ... */ };            // see §3.2.2 and §20.5.2

B* pb = new D;                         // OK: implicit conversion from D* to B*
D* pd = pb;                            // error: no implicit conversion from B* to D*
D* pd = static_cast<D*>(pb);           // OK
```

Conversions among class pointers and among class reference types are discussed in §22.2.

If you feel tempted to use an explicit type conversion, take the time to consider if it is *really* necessary. In C++, explicit type conversion is unnecessary in most cases when C needs it (§1.3.3) and also in many cases in which earlier versions of C++ needed it (§1.3.2, §44.2.3). In many programs, explicit type conversion can be completely avoided; in others, its use can be localized to a few routines.

11.5.3 C-Style Cast

From C, C++ inherited the notation **(T)e**, which performs any conversion that can be expressed as a combination of **static_cast**s, **reinterpret_cast**s, **const_cast**s to make a value of type **T** from the expression **e** (§44.2.3). Unfortunately, the C-style cast can also cast from a pointer to a class to a pointer to a private base of that class. Never do that, and hope for a warning from the compiler if you do it by mistake. This C-style cast is far more dangerous than the named conversion operators because the notation is harder to spot in a large program and the kind of conversion intended by the programmer is not explicit. That is, **(T)e** might be doing a portable conversion between related types, a nonportable conversion between unrelated types, or removing the **const** modifier from a pointer type. Without knowing the exact types of **T** and **e**, you cannot tell.

11.5.4 Function-Style Cast

The construction of a value of type `T` from a value `e` can be expressed by the functional notation `T(e)`. For example:

```
void f(double d)
{
    int i = int(d);                 // truncate d
    complex z = complex(d);         // make a complex from d
    // ...
}
```

The `T(e)` construct is sometimes referred to as a *function-style cast*. Unfortunately, for a built-in type `T`, `T(e)` is equivalent to `(T)e` (§11.5.3). This implies that for many built-in types `T(e)` is not safe.

```
void f(double d, char* p)
{
    int a = int(d);    // truncates
    int b = int(p);    // not portable
    // ...
}
```

Even explicit conversion of a longer integer type to a shorter (such as `long` to `char`) can result in nonportable implementation-defined behavior.

Prefer `T{v}` conversions for well-behaved construction and the named casts (e.g., `static_cast`) for other conversions.

11.6 Advice

[1] Prefer prefix `++` over suffix `++`; §11.1.4.
[2] Use resource handles to avoid leaks, premature deletion, and double deletion; §11.2.1.
[3] Don't put objects on the free store if you don't have to; prefer scoped variables; §11.2.1.
[4] Avoid "naked `new`" and "naked `delete`"; §11.2.1.
[5] Use RAII; §11.2.1.
[6] Prefer a named function object to a lambda if the operation requires comments; §11.4.2.
[7] Prefer a named function object to a lambda if the operation is generally useful; §11.4.2.
[8] Keep lambdas short; §11.4.2.
[9] For maintainability and correctness, be careful about capture by reference; §11.4.3.1.
[10] Let the compiler deduce the return type of a lambda; §11.4.4.
[11] Use the `T{e}` notation for construction; §11.5.1.
[12] Avoid explicit type conversion (casts); §11.5.
[13] When explicit type conversion is necessary, prefer a named cast; §11.5.
[14] Consider using a run-time checked cast, such as `narrow_cast<>()`, for conversion between numeric types; §11.5.

This page intentionally left blank

12

Functions

Death to all fanatics!
– Paradox

- Function Declarations
  Why Functions?; Parts of a Function Declaration; Function Definitions; Returning Values; **inline** Functions; **constexpr** Functions; **[[noreturn]]** Functions; Local Variables
- Argument Passing
  Reference Arguments; Array Arguments; List Arguments; Unspecified Number of Arguments; Default Arguments
- Overloaded Functions
  Automatic Overload Resolution; Overloading and Return Type; Overloading and Scope; Resolution for Multiple Arguments; Manual Overload Resolution
- Pre- and Postconditions
- Pointer to Function
- Macros
  Conditional Compilation; Predefined Macros; Pragmas
- Advice

12.1 Function Declarations

The main way of getting something done in a C++ program is to call a function to do it. Defining a function is the way you specify how an operation is to be done. A function cannot be called unless it has been previously declared.

A function declaration gives the name of the function, the type of the value returned (if any), and the number and types of the arguments that must be supplied in a call. For example:

```
Elem* next_elem();      // no argument; return an Elem*
void exit(int);         // int argument; return nothing
double sqrt(double);    // double argument; return a double
```

The semantics of argument passing are identical to the semantics of copy initialization (§16.2.6). Argument types are checked and implicit argument type conversion takes place when necessary. For example:

```
double s2 = sqrt(2);          // call sqrt() with the argument double{2}
double s3 = sqrt("three");    // error: sqrt() requires an argument of type double
```

The value of such checking and type conversion should not be underestimated.

A function declaration may contain argument names. This can be a help to the reader of a program, but unless the declaration is also a function definition, the compiler simply ignores such names. As a return type, **void** means that the function does not return a value (§6.2.7).

The type of a function consists of the return type and the argument types. For class member functions (§2.3.2, §16.2), the name of the class is also part of the function type. For example:

```
double f(int i, const Info&);          // type: double(int,const Info&)
char& String::operator[](int);         // type: char& String::(int)
```

12.1.1 Why Functions?

There is a long and disreputable tradition of writing very long functions – hundreds of lines long. I once encountered a single (handwritten) function with more than 32,768 lines of code. Writers of such functions seem to fail to appreciate one of the primary purposes of functions: to break up complicated computations into meaningful chunks and name them. We want our code to be comprehensible, because that is the first step on the way to maintainability. The first step to comprehensibility is to break computational tasks into comprehensible chunks (represented as functions and classes) and name those. Such functions then provide the basic vocabulary of computation, just as the types (built-in and user-defined) provide the basic vocabulary of data. The C++ standard algorithms (e.g., **find**, **sort**, and **iota**) provide a good start (Chapter 32). Next, we can compose functions representing common or specialized tasks into larger computations.

The number of errors in code correlates strongly with the amount of code and the complexity of the code. Both problems can be addressed by using more and shorter functions. Using a function to do a specific task often saves us from writing a specific piece of code in the middle of other code; making it a function forces us to name the activity and document its dependencies. Also, function call and return saves us from using error-prone control structures, such as **goto**s (§9.6) and **continue**s (§9.5.5). Unless they are very regular in structure, nested loops are an avoidable source of errors (e.g., use a dot product to express a matrix algorithm rather than nesting loops; §40.6).

The most basic advice is to keep a function of a size so that you can look at it in total on a screen. Bugs tend to creep in when we can view only part of an algorithm at a time. For many programmers that puts a limit of about 40 lines on a function. My ideal is a much smaller size still, maybe an average of 7 lines.

In essentially all cases, the cost of a function call is not a significant factor. Where that cost could be significant (e.g., for frequently used access functions, such as vector subscripting) inlining can eliminate it (§12.1.5). Use functions as a structuring mechanism.

12.1.2 Parts of a Function Declaration

In addition to specifying a name, a set of arguments, and a return type, a function declaration can contain a variety of specifiers and modifiers. In all we can have:

- The name of the function; required
- The argument list, which may be empty **()**; required
- The return type, which may be **void** and which may be prefix or suffix (using **auto**); required
- **inline**, indicating a desire to have function calls implemented by inlining the function body (§12.1.5)
- **constexpr**, indicating that it should be possible to evaluate the function at compile time if given constant expressions as arguments (§12.1.6)
- **noexcept**, indicating that the function may not throw an exception (§13.5.1.1)
- A linkage specification, for example, **static** (§15.2)
- **[[noreturn]]**, indicating that the function will not return using the normal call/return mechanism (§12.1.4)

In addition, a member function may be specified as:

- **virtual**, indicating that it can be overridden in a derived class (§20.3.2)
- **override**, indicating that it must be overriding a virtual function from a base class (§20.3.4.1)
- **final**, indicating that it cannot be overriden in a derived class (§20.3.4.2)
- **static**, indicating that it is not associated with a particular object (§16.2.12)
- **const**, indicating that it may not modify its object (§3.2.1.1, §16.2.9.1)

If you feel inclined to give readers a headache, you may write something like:

```
struct S {
    [[noreturn]] virtual inline auto f(const unsigned long int *const) –> void const noexcept;
};
```

12.1.3 Function Definitions

Every function that is called must be defined somewhere (once only; §15.2.3). A function definition is a function declaration in which the body of the function is presented. For example:

```
void swap(int*, int*);          // a declaration

void swap(int* p, int* q)       // a definition
{
    int t = *p;
    *p = *q;
    *q = t;
}
```

The definition and all declarations for a function must specify the same type. Unfortunately, to preserve C compatibility, a **const** is ignored at the highest level of an argument type. For example, this is two declarations of the same function:

```
void f(int);          // type is void(int)
void f(const int);    // type is void(int)
```

That function, **f()**, could be defined as:

```
void f(int x) { /*we can modify x here*/ }
```

Alternatively, we could define **f()** as:

```
void f(const int x) { /*we cannot modify x here*/ }
```

In either case, the argument that **f()** can or cannot modify is a copy of what a caller provided, so there is no danger of an obscure modification of the calling context.

Function argument names are not part of the function type and need not be identical in different declarations. For example:

```
int& max(int& a, int& b, int& c); // return a reference to the larger of a, b, and c

int& max(int& x1, int& x2, int& x3)
{
    return (x1>x2)? ((x1>x3)?x1:x3) : ((x2>x3)?x2:x3);
}
```

Naming arguments in declarations that are not definitions is optional and commonly used to simplify documentation. Conversely, we can indicate that an argument is unused in a function definition by not naming it. For example:

```
void search(table* t, const char* key, const char*)
{
    // no use of the third argument
}
```

Typically, unnamed arguments arise from the simplification of code or from planning ahead for extensions. In both cases, leaving the argument in place, although unused, ensures that callers are not affected by the change.

In addition to functions, there are a few other things that we can call; these follow most rules defined for functions, such as the rules for argument passing (§12.2):

- *Constructors* (§2.3.2, §16.2.5) are technicallly not functions; in particular, they don't return a value, can initialize bases and members (§17.4), and can't have their address taken.
- *Destructors* (§3.2.1.2, §17.2) can't be overloaded and can't have their address taken.
- *Function objects* (§3.4.3, §19.2.2) are not functions (they are objects) and can't be overloaded, but their **operator()**s are functions.
- *Lambda expressions* (§3.4.3, §11.4) are basically a shorthand for defining function objects.

12.1.4 Returning Values

Every function declaration contains a specification of the function's *return type* (except for constructors and type conversion functions). Traditionally, in C and C++, the return type comes first in a function declaration (before the name of the function). However, a function declaration can also be written using a syntax that places the return type after the argument list. For example, the following two declarations are equivalent:

```
string to_string(int a);          // prefix return type
auto to_string(int a) –> string;  // suffix return type
```

That is, a prefix **auto** indicates that the return type is placed after the argument list. The suffix return type is preceded by **–>**.

The essential use for a suffix return type comes in function template declarations in which the return type depends on the arguments. For example:

```
template<class T, class U>
auto product(const vector<T>& x, const vector<U>& y) –> decltype(x*y);
```

However, the suffix return syntax can be used for any function. There is an obvious similarity between the suffix return syntax for a function and the lambda expression syntax (§3.4.3, §11.4); it is a pity those two constructs are not identical.

A function that does not return a value has a ''return type'' of **void**.

A value must be returned from a function that is not declared **void** (however, **main()** is special; see §2.2.1). Conversely, a value cannot be returned from a **void** function. For example:

```
int f1() { }                // error: no value returned
void f2() { }               // OK

int f3() { return 1; }      // OK
void f4() { return 1; }     // error: return value in void function

int f5() { return; }        // error: return value missing
void f6() { return; }       // OK
```

A return value is specified by a **return**-statement. For example:

```
int fac(int n)
{
    return (n>1) ? n*fac(n–1) : 1;
}
```

A function that calls itself is said to be *recursive*.

There can be more than one **return**-statement in a function:

```
int fac2(int n)
{
    if (n > 1)
        return n*fac2(n–1);
    return 1;
}
```

Like the semantics of argument passing, the semantics of function value return are identical to the semantics of copy initialization (§16.2.6). A **return**-statement initializes a variable of the returned type. The type of a return expression is checked against the type of the returned type, and all standard and user-defined type conversions are performed. For example:

```
double f() { return 1; }        // 1 is implicitly converted to double{1}
```

Each time a function is called, a new copy of its arguments and local (automatic) variables is created. The store is reused after the function returns, so a pointer to a local non-**static** variable should never be returned. The contents of the location pointed to will change unpredictably:

```
int* fp()
{
    int local = 1;
    // ...
    return &local;  // bad
}
```

An equivalent error can occur when using references:

```
int& fr()
{
    int local = 1;
    // ...
    return local;    // bad
}
```

Fortunately, a compiler can easily warn about returning references to local variables (and most do).

There are no **void** values. However, a call of a **void** function may be used as the return value of a **void** function. For example:

```
void g(int* p);

void h(int* p)
{
    // ...
    return g(p);    // OK: equivalent to "g(p); return;"
}
```

This form of return is useful to avoid special cases when writing template functions where the return type is a template parameter.

A **return**-statement is one of five ways of exiting a function:

- Executing a **return**-statement.
- ''Falling off the end'' of a function; that is, simply reaching the end of the function body. This is allowed only in functions that are not declared to return a value (i.e., **void** functions) and in **main()**, where falling off the end indicates successful completion (§12.1.4).
- Throwing an exception that isn't caught locally (§13.5).
- Terminating because an exception was thrown and not caught locally in a **noexcept** function (§13.5.1.1).
- Directly or indirectly invoking a system function that doesn't return (e.g., **exit()**; §15.4).

A function that does not return normally (i.e., through a **return** or ''falling off the end'') can be marked **[[noreturn]]** (§12.1.7).

12.1.5 inline Functions

A function can be defined to be **inline**. For example:

```
inline int fac(int n)
{
    return (n<2) ? 1 : n*fac(n–1);
}
```

The inline specifier is a hint to the compiler that it should attempt to generate code for a call of fac() inline rather than laying down the code for the function once and then calling through the usual function call mechanism. A clever compiler can generate the constant 720 for a call fac(6). The possibility of mutually recursive inline functions, inline functions that recurse or not depending on input, etc., makes it impossible to guarantee that every call of an inline function is actually inlined. The degree of cleverness of a compiler cannot be legislated, so one compiler might generate 720, another 6*fac(5), and yet another an un-inlined call fac(6). If you want a guarantee that a value is computed at compile time, declare it constexpr and make sure that all functions used in its evaluation are constexpr (§12.1.6).

To make inlining possible in the absence of unusually clever compilation and linking facilities, the definition – and not just the declaration – of an inline function must be in scope (§15.2). An inline specifier does not affect the semantics of a function. In particular, an inline function still has a unique address, and so do static variables (§12.1.8) of an inline function.

If an inline function is defined in more than one translation unit (e.g., typically because it was defined in a header; §15.2.2), its definition in the different translation units must be identical (§15.2.3).

12.1.6 constexpr Functions

In general, a function cannot be evaluated at compile time and therefore cannot be called in a constant expression (§2.2.3, §10.4). By specifying a function constexpr, we indicate that we want it to be usable in constant expressions if given constant expressions as arguments. For example:

```
constexpr int fac(int n)
{
    return (n>1) ? n*fac(n–1) : 1;
}

constexpr int f9 = fac(9);          // must be evaluated at compile time
```

When constexpr is used in a function definition, it means "should be usable in a constant expression when given constant expressions as arguments." When used in an object definition, it means "evaluate the initializer at compile time." For example:

```
void f(int n)
{
    int f5 = fac(5);                // may be evaluated at compile time
    int fn = fac(n);                // evaluated at run time (n is a variable)

    constexpr int f6 = fac(6);      // must be evaluated at compile time
    constexpr int fnn = fac(n);     // error: can't guarantee compile-time evaluation (n is a variable)

    char a[fac(4)];                 // OK: array bounds must be constants and fac() is constexpr
    char a2[fac(n)];                // error: array bounds must be constants and n is a variable

    // ...
}
```

To be evaluated at compile time, a function must be suitably simple: a constexpr function must

consist of a single **return**-statement; no loops and no local variables are allowed. Also, a **constexpr** function may not have side effects. That is, a **constexpr** function is a pure function. For example:

```
int glob;

constexpr void bad1(int a)      // error: constexpr function cannot be void
{
    glob = a;                   // error: side effect in constexpr function
}

constexpr int bad2(int a)
{
    if (a>=0) return a; else return –a;     // error: if-statement in constexpr function
}

constexpr int bad3(int a)
{
    sum = 0;                                // error: local variable in constexpr function
    for (int i=0; i<a; +=i) sum +=fac(i);   // error: loop in constexpr function
    return sum;
}
```

The rules for a **constexpr** constructor are suitably different (§10.4.3); there, only simple initialization of members is allowed.

A **constexpr** function allows recursion and conditional expressions. This implies that you can express just about anything as a **constexpr** function if you really want to. However, you'll find the debugging gets unnecessarily difficult and compile times longer than you would like unless you restrict the use of **constexpr** functions to the relatively simple tasks for which they are intended.

By using literal types (§10.4.3), **constexpr** functions can be defined to use user-defined types.

Like inline functions, **constexpr** functions obey the ODR (''one-definition rule''), so that definitions in the different translation units must be identical (§15.2.3). You can think of **constexpr** functions as a restricted form of inline functions (§12.1.5).

12.1.6.1 constexpr and References

A **constexpr** function cannot have side effects, so writing to nonlocal objects is not possible. However, a **constexpr** function can refer to nonlocal objects as long as it does not write to them.

```
constexpr int ftbl[] { 1, 2, 3, 5, 8, 13 };

constexpr int fib(int n)
{
    return (n<sizeof(ftbl)/sizeof(*ftbl)) ? ftbl[n] : fib(n);
}
```

A **constexpr** function can take reference arguments. Of course, it cannot write through such references, but **const** reference parameters are as useful as ever. For example, in the standard library (§40.4) we find:

```
template<> class complex<float> {
public:
// ...
    explicit constexpr complex(const complex<double>&);
    // ...
};
```

This allows us to write:

```
constexpr complex<float> z {2.0};
```

The temporary variable that is logically constructed to hold the **const** reference argument simply becomes a value internal to the compiler.

It is possible for a **constexpr** function to return a reference or a pointer. For example:

```
constexpr const int* addr(const int& r) { return &r; }     // OK
```

However, doing so brings us away from the fundamental role of **constexpr** functions as parts of constant expression evaluation. In particular, it can be quite tricky to determine whether the result of such a function is a constant expression. Consider:

```
static const int x = 5;
constexpr const int* p1 = addr(x);      // OK
constexpr int xx = *p1;                 // OK

static int y;
constexpr const int* p2 = addr(y);      // OK
constexpr int yy = *y;                  // error: attempt to read a variable

constexpr const int* tp = addr(5);      // error: address of temporary
```

12.1.6.2 Conditional Evaluation

A branch of a conditional expression that is not taken in a **constexpr** function is not evaluated. This implies that a branch not taken can require run-time evaluation. For example:

```
constexpr int check(int i)
{
    return (low<=i && i<high) ? i : throw out_of_range();
}

constexpr int low = 0;
constexpr int  high = 99;

// ...
constexpr int val = check(f(x,y,z));
```

You might imagine **low** and **high** to be configuration parameters that are known at compile time, but not at design time, and that **f(x,y,z)** computes some implementation-dependent value.

12.1.7 [[noreturn]] Functions

A construct [[...]] is called an *attribute* and can be placed just about anywhere in the C++ syntax. In general, an attribute specifies some implementation-dependent property about the syntactic entity that precedes it. In addition, an attribute can be placed in front of a declaration. There are only two standard attributes (§iso.7.6), and [[noreturn]] is one of them. The other is [[carries_dependency]] (§41.3).

Placing [[noreturn]] at the start of a function declaration indicates that the function is not expected to return. For example:

```
[[noreturn]] void exit(int);      // exit will never return
```

Knowing that a function does not return is useful for both comprehension and code generation. What happens if the function returns despite a [[noreturn]] attribute is undefined.

12.1.8 Local Variables

A name defined in a function is commonly referred to as a *local name*. A local variable or constant is initialized when a thread of execution reaches its definition. Unless declared static, each invocation of the function has its own copy of the variable. If a local variable is declared static, a single, statically allocated object (§6.4.2) will be used to represent that variable in all calls of the function. It will be initialized only the first time a thread of execution reaches its definition. For example:

```
void f(int a)
{
    while (a--) {
        static int n = 0;      // initialized once
        int x = 0;             // initialized 'a' times in each call of f()

        cout << "n == " << n++ << ", x == " << x++ << '\n';
    }
}

int main()
{
    f(3);
}
```

This prints:

```
n == 0, x == 0
n == 1, x == 0
n == 2, x == 0
```

A static local variable allows the function to preserve information between calls without introducing a global variable that might be accessed and corrupted by other functions (see also §16.2.12).

Initialization of a static local variable does not lead to a data race (§5.3.1) unless you enter the function containing it recursively or a deadlock occurs (§iso.6.7). That is, the C++ implementation must guard the initialization of a local static variable with some kind of lock-free construct (e.g., a call_once; §42.3.3). The effect of initializing a local static recursively is undefined. For example:

```
int fn(int n)
{
        static int n1 = n;              // OK
        static int n2 = fn(n–1)+1;      // undefined
        return n;
}
```

A **static** local variable is useful for avoiding order dependencies among nonlocal variables (§15.4.1).

There are no local functions; if you feel you need one, use a function object or a lambda expression (§3.4.3, §11.4).

The scope of a label (§9.6), should you be foolhardy enough to use one, is the complete function, independent of which nested scope it may be in.

12.2 Argument Passing

When a function is called (using the suffix **()**, known as the *call operator* or *application operator*), store is set aside for its *formal arguments* (also known as its *parameters*), and each formal argument is initialized by its corresponding actual argument. The semantics of argument passing are identical to the semantics of initialization (copy initialization, to be precise; §16.2.6). In particular, the type of an actual argument is checked against the type of the corresponding formal argument, and all standard and user-defined type conversions are performed. Unless a formal argument (parameter) is a reference, a copy of the actual argument is passed to the function. For example:

```
int* find(int* first, int* last, int v)     // find x in [first:last)
{
        while (first!=last && *first!=v)
                ++first;
        return first;
}

void g(int* p, int* q)
{
        int* pp = find(p,q,'x');
        // ...
}
```

Here, the caller's copy of the argument, **p**, is not modified by the operations on **find()**'s copy, called **first**. The pointer is passed by value.

There are special rules for passing arrays (§12.2.2), a facility for passing unchecked arguments (§12.2.4), and a facility for specifying default arguments (§12.2.5). The use of initializer lists is described in §12.2.3 and the ways of passing arguments to template functions in §23.5.2 and §28.6.2.

12.2.1 Reference Arguments

Consider:

```
void f(int val, int& ref)
{
    ++val;
    ++ref;
}
```

When `f()` is called, `++val` increments a local copy of the first actual argument, whereas `++ref` increments the second actual argument. Consider:

```
void g()
{
    int i = 1;
    int j = 1;
    f(i,j);
}
```

The call `f(i,j)` will increment `j` but not `i`. The first argument, `i`, is passed *by value*; the second argument, `j`, is passed *by reference*. As mentioned in §7.7, functions that modify call-by-reference arguments can make programs hard to read and should most often be avoided (but see §18.2.5). It can, however, be noticeably more efficient to pass a large object by reference than to pass it by value. In that case, the argument might be declared a `const` reference to indicate that the reference is used for efficiency reasons only and not to enable the called function to change the value of the object:

```
void f(const Large& arg)
{
    // the value of "arg" cannot be changed
    // (except by using explicit type conversion; §11.5)
}
```

The absence of `const` in the declaration of a reference argument is taken as a statement of intent to modify the variable:

```
void g(Large& arg);      // assume that g() modifies arg
```

Similarly, declaring a pointer argument `const` tells readers that the value of an object pointed to by that argument is not changed by the function. For example:

```
int strlen(const char*);                    // number of characters in a C-style string
char* strcpy(char* to, const char* from);   // copy a C-style string
int strcmp(const char*, const char*);       // compare C-style strings
```

The importance of using `const` arguments increases with the size of a program.

Note that the semantics of argument passing are different from the semantics of assignment. This is important for `const` arguments, reference arguments, and arguments of some user-defined types.

Following the rules for reference initialization, a literal, a constant, and an argument that requires conversion can be passed as a `const T&` argument, but not as a plain (non-`const`) `T&` argument. Allowing conversions for a `const T&` argument ensures that such an argument can be given

exactly the same set of values as a `T` argument by passing the value in a temporary, if necessary. For example:

```
float fsqrt(const float&);  // Fortran-style sqrt taking a reference argument

void g(double d)
{
    float r = fsqrt(2.0f);      // pass reference to temp holding 2.0f
    r = fsqrt(r);               // pass reference to r
    r = fsqrt(d);               // pass reference to temp holding static_cast<float>(d)
}
```

Disallowing conversions for non-`const` reference arguments (§7.7) avoids the possibility of silly mistakes arising from the introduction of temporaries. For example:

```
void update(float& i);

void g(double d, float r)
{
    update(2.0f);       // error: const argument
    update(r);          // pass reference to r
    update(d);          // error: type conversion required
}
```

Had these calls been allowed, `update()` would quietly have updated temporaries that immediately were deleted. Usually, that would come as an unpleasant surprise to the programmer.

If we wanted to be precise, pass-by-reference would be pass-by-lvalue-reference because a function can also take rvalue references. As described in §7.7, an rvalue can be bound to an rvalue reference (but not to an lvalue reference) and an lvalue can be bound to an lvalue reference (but not to an rvalue reference). For example:

```
void f(vector<int>&);               // (non-const) lvalue reference argument
void f(const vector<int>&);         // const lvalue reference argument
void f(vector<int>&&);              // rvalue reference argument

void g(vector<int>& vi, const vector<int>& cvi)
{
    f(vi);                  // call f(vector<int>&)
    f(vci);                 // call f(const vector<int>&)
    f(vector<int>{1,2,3,4});    // call f(vector<int>&&);
}
```

We must assume that a function will modify an rvalue argument, leaving it good only for destruction or reassignment (§17.5). The most obvious use of rvalue references is to define move constructors and move assignments (§3.3.2, §17.5.2). I'm sure someone will find a clever use for `const`-rvalue-reference arguments, but so far, I have not seen a genuine use case.

Please note that for a template argument `T`, the template argument type deduction rules give `T&&` a significantly different meaning from `X&&` for a type `X` (§23.5.2.1). For template arguments, an rvalue reference is most often used to implement "perfect forwarding" (§23.5.2.1, §28.6.3).

How do we choose among the ways of passing arguments? My rules of thumb are:

[1] Use pass-by-value for small objects.
[2] Use pass-by-**const**-reference to pass large values that you don't need to modify.
[3] Return a result as a **return** value rather than modifying an object through an argument.
[4] Use rvalue references to implement move (§3.3.2, §17.5.2) and forwarding (§23.5.2.1).
[5] Pass a pointer if "no object" is a valid alternative (and represent "no object" by **nullptr**).
[6] Use pass-by-reference only if you have to.

The "when you have to" in the last rule of thumb refers to the observation that passing pointers is often a less obscure mechanism for dealing with objects that need modification (§7.7.1, §7.7.4) than using references.

12.2.2 Array Arguments

If an array is used as a function argument, a pointer to its initial element is passed. For example:

```
int strlen(const char*);

void f()
{
    char v[] = "Annemarie";
    int i = strlen(v);
    int j = strlen("Nicholas");
}
```

That is, an argument of type **T[]** will be converted to a **T*** when passed as an argument. This implies that an assignment to an element of an array argument changes the value of an element of the argument array. In other words, arrays differ from other types in that an array is not passed by value. Instead, a pointer is passed (by value).

A parameter of array type is equivalent to a parameter of pointer type. For example:

```
void odd(int* p);
void odd(int a[]);
void odd(int buf[1020]);
```

These three declarations are equivalent and declare the same function. As usual, the argument names do not affect the type of the function (§12.1.3). The rules and techniques for passing multi-dimensional arrays can be found in §7.4.3.

The size of an array is not available to the called function. This is a major source of errors, but there are several ways of circumventing this problem. C-style strings are zero-terminated, so their size can be computed (e.g., by a potentially expensive call of **strlen()**; §43.4). For other arrays, a second argument specifying the size can be passed. For example:

```
void compute1(int* vec_ptr, int vec_size);      // one way
```

At best, this is a workaround. It is usually preferable to pass a reference to some container, such as **vector** (§4.4.1, §31.4), **array** (§34.2.1), or **map** (§4.4.3, §31.4.3).

If you really want to pass an array, rather than a container or a pointer to the first element of an array, you can declare a parameter of type reference to array. For example:

```
void f(int(&r)[4]);

void g()
{
        int a1[] = {1,2,3,4};
        int a2[] = {1,2};

        f(a1);      // OK
        f(a2);      // error : wrong number of elements
}
```

Note that the number of elements is part of a reference-to-array type. That makes such references far less flexible than pointers and containers (such as **vector**). The main use of references to arrays is in templates, where the number of elements is then deduced. For example:

```
template<class T, int N> void f(T(&r)[N])
{
        // ...
}

int a1[10];
double a2[100];

void g()
{
        f(a1);          // T is int; N is 10
        f(a2);          // T is double; N is 100
}
```

This typically gives rise to as many function definitions as there are calls to **f()** with distinct array types.

Multidimensional arrays are tricky (see §7.3), but often arrays of pointers can be used instead, and they need no special treatment. For example:

```
const char* day[] = {
        "mon", "tue", "wed", "thu", "fri", "sat", "sun"
};
```

As ever, **vector** and similar types are alternatives to the built-in, low-level arrays and pointers.

12.2.3 List Arguments

A **{}**-delimited list can be used as an argument to a parameter of:

[1] Type **std::initializer_list<T>**, where the values of the list can be implicitly converted to **T**

[2] A type that can be initialized with the values provided in the list

[3] A reference to an array of **T**, where the values of the list can be implicitly converted to **T**

Technically, case [2] covers all examples, but I find it easier to think of the three cases separately. Consider:

```
template<class T>
void f1(initializer_list<T>);

struct S {
        int a;
        string s;
};
void f2(S);

template<class T, int N>
void f3(T (&r)[N]);

void f4(int);

void g()
{
        f1({1,2,3,4});     // T is int and the initializer_list has size() 4
        f2({1,"MKS"});     // f2(S{1,"MKS"})
        f3({1,2,3,4});     // T is int and N is 4
        f4({1});           // f4(int{1});
}
```

If there is a possible ambiguity, an **initializer_list** parameter takes priority. For example:

```
template<class T>
void f(initializer_list<T>);

struct S {
        int a;
        string s;
};
void f(S);

template<class T, int N>
void f(T (&r)[N]);

void f(int);

void g()
{
        f({1,2,3,4});      // T is int and the initializer_list has size() 4
        f({1,"MKS"});      // calls f(S)
        f({1});            // T is int and the initializer_list has size() 1
}
```

The reason that a function with an **initializer_list** argument take priority is that it could be very confusing if different functions were chosen based on the number of elements of a list. It is not possible to eliminate every form of confusion in overload resolution (for example, see §4.4, §17.3.4.1), but giving **initializer_list** parameters priority for **{}**-list arguments seems to minimize confusion.

If there is a function with an initializer-list argument in scope, but the argument list isn't a match for that, another function can be chosen. The call `f({1,"MKS"})` was an example of that.

Note that these rules apply to `std::initializer_list<T>` arguments only. There are no special rules for `std::initializer_list<T>&` or for other types that just happen to be called `initializer_list` (in some other scope).

12.2.4 Unspecified Number of Arguments

For some functions, it is not possible to specify the number and type of all arguments expected in a call. To implement such interfaces, we have three choices:

[1] Use a variadic template (§28.6): this allows us to handle an arbitrary number of arbitrary types in a type-safe manner by writing a small template metaprogram that interprets the argument list to determine its meaning and take appropriate actions.

[2] Use an `initializer_list` as the argument type (§12.2.3). This allows us to handle an arbitrary number of arguments of a single type in a type-safe manner. In many contexts, such homogeneous lists are the most common and important case.

[3] Terminate the argument list with the ellipsis (`...`), which means "and maybe some more arguments." This allows us to handle an arbitrary number of (almost) arbitrary types by using some macros from `<cstdarg>`. This solution is *not* inherently type-safe and can be hard to use with sophisticated user-defined types. However, this mechanism has been used from the earliest days of C.

The first two mechanisms are described elsewhere, so I describe only the third mechanism (even though I consider it inferior to the others for most uses). For example:

```
int printf(const char* ...);
```

This specifies that a call of the standard-library function `printf()` (§43.3) must have at least one argument, a C-style string, but may or may not have others. For example:

```
printf("Hello, world!\n");
printf("My name is %s %s\n", first_name, second_name);
printf("%d + %d = %d\n",2,3,5);
```

Such a function must rely on information not available to the compiler when interpreting its argument list. In the case of `printf()`, the first argument is a format string containing special character sequences that allow `printf()` to handle other arguments correctly; `%s` means "expect a `char*` argument" and `%d` means "expect an `int` argument." However, the compiler cannot in general ensure that the expected arguments are really provided in a call or that an argument is of the expected type. For example:

```
#include <cstdio>

int main()
{
    std::printf("My name is %s %s\n",2);
}
```

This is not valid code, but most compilers will not catch this error. At best, it will produce some strange-looking output (try it!).

Clearly, if an argument has not been declared, the compiler does not have the information needed to perform the standard type checking and type conversion for it. In that case, a **char** or a **short** is passed as an **int** and a **float** is passed as a **double**. This is not necessarily what the programmer expects.

A well-designed program needs at most a few functions for which the argument types are not completely specified. Overloaded functions, functions using default arguments, functions taking **initializer_list** arguments, and variadic templates can be used to take care of type checking in most cases when one would otherwise consider leaving argument types unspecified. Only when both the number of arguments *and* the types of arguments vary *and* a variadic template solution is deemed undesirable is the ellipsis necessary.

The most common use of the ellipsis is to specify an interface to C library functions that were defined before C++ provided alternatives:

```
int fprintf(FILE*, const char* ...);        // from <cstdio>
int execl(const char* ...);                 // from UNIX header
```

A standard set of macros for accessing the unspecified arguments in such functions can be found in **<cstdarg>**. Consider writing an error function that takes one integer argument indicating the severity of the error followed by an arbitrary number of strings. The idea is to compose the error message by passing each word as a separate C-style string argument. The list of string arguments should be terminated by the null pointer:

```
extern void error(int ...);
extern char* itoa(int, char[]);   // int to alpha

int main(int argc, char* argv[])
{
        switch (argc) {
        case 1:
                error(0,argv[0],nullptr);
                break;
        case 2:
                error(0,argv[0],argv[1],nullptr);
                break;
        default:
                char buffer[8];
                error(1,argv[0],"with",itoa(argc-1,buffer),"arguments",nullptr);
        }
        // ...
}
```

The function **itoa()** returns a C-style string representing its **int** argument. It is popular in C, but not part of the C standard.

I always pass **argv[0]** because that, conventionally, is the name of the program.

Note that using the integer **0** as the terminator would not have been portable: on some implementations, the integer 0 and the null pointer do not have the same representation (§6.2.8). This illustrates the subtleties and extra work that face the programmer once type checking has been suppressed using the ellipsis.

The error() function could be defined like this:

```
#include <cstdarg>

void error(int severity ...) // "severity" followed by a zero-terminated list of char*s
{
        va_list ap;
        va_start(ap,severity);      // arg startup

        for (;;) {
                char* p = va_arg(ap,char*);
                if (p == nullptr) break;
                cerr << p << ' ';
        }

        va_end(ap);                 // arg cleanup

        cerr << '\n';
        if (severity) exit(severity);
}
```

First, a va_list is defined and initialized by a call of va_start(). The macro va_start takes the name of the va_list and the name of the last formal argument as arguments. The macro va_arg() is used to pick the unnamed arguments in order. In each call, the programmer must supply a type; va_arg() assumes that an actual argument of that type has been passed, but it typically has no way of ensuring that. Before returning from a function in which va_start() has been used, va_end() must be called. The reason is that va_start() may modify the stack in such a way that a return cannot successfully be done; va_end() undoes any such modifications.

Alternatively, error() could have been defined using a standard-library initializer_list:

```
void error(int severity, initializer_list<string> err)
{
        for (auto& s : err)
                cerr << s << ' ';
        cerr << '\n';
        if (severity) exit(severity);
}
```

It would then have to be called using the list notation. For example:

```
        switch (argc) {
        case 1:
                error(0,{argv[0]});
                break;
        case 2:
                error(0,{argv[0],argv[1]});
                break;
        default:
                error(1,{argv[0],"with",to_string(argc−1),"arguments"});
        }
```

The int-to-string conversion function to_string() is provided by the standard library (§36.3.5).

If I didn't have to mimic C style, I would further simplify the code by passing a container as a single argument:

```
void error(int severity, const vector<string>& err)  // almost as before
{
        for (auto& s : err)
                cerr << s << ' ';
        cerr << '\n';
        if (severity) exit(severity);
}

vector<string> arguments(int argc, char* argv[]) // package arguments
{
        vector<string> res;
        for (int i = 0; i!=argc; ++i)
                res.push_back(argv[i]);
        return res
}

int main(int argc, char* argv[])
{
        auto args = arguments(argc,argv);
        error((args.size()<2)?0:1,args);
        // ...
}
```

The helper function, arguments(), is trivial, and main() and error() are simple. The interface between main() and error() is more general in that it now passes all arguments. That would allow later improvements of error(). The use of the vector<string> is far less error-prone than any use of an unspecified number of arguments.

12.2.5 Default Arguments

A general function often needs more arguments than are necessary to handle simple cases. In particular, functions that construct objects (§16.2.5) often provide several options for flexibility. Consider class complex from §3.2.1.1:

```
class complex {
        double re, im;
public:
        complex(double r, double i) :re{r}, im{i} {}     // construct complex from two scalars
        complex(double r) :re{r}, im{0} {}               // construct complex from one scalar
        complex() :re{0}, im{0} {}
// default complex: {0,0}
        // ...
};
```

The actions of complex's constructors are quite trivial, but logically there is something odd about having three functions (here, constructors) doing essentially the same task. Also, for many classes,

constructors do more work and the repetitiveness is common. We could deal with the repetitiveness by considering one of the constructors ''the real one'' and forward to that (§17.4.3):

```
complex(double r, double i) :re{r}, im{i} {}     // construct complex from two scalars
complex(double r) :complex{2,0} {}                // construct complex from one scalar
complex() :complex{0,0} {}                        // default complex: {0,0}
```

Say we wanted to add some debugging, tracing, or statistics-gathering code to **complex**; we now have a single place to do so. However, this can be abbreviated further:

```
complex(double r ={}, double i ={}) :re{r}, im{i} {}   // construct complex from two scalars
```

This makes it clear that if a user supplies fewer than the two arguments needed, the default is used. The intent of having a single constructor plus some shorthand notation is now explicit.

A default argument is type checked at the time of the function declaration and evaluated at the time of the call. For example:

```
class X {
public:
        static int def_arg;
        void f(int =def_arg);
        // ...
};

int X::def_arg = 7;

void g(X& a)
{
        a.f();          // maybe f(7)
        a.def_arg = 9;
        a.f();          // f(9)
}
```

Default arguments that can change value are most often best avoided because they introduce subtle context dependencies.

Default arguments may be provided for trailing arguments only. For example:

```
int f(int, int =0, char* =nullptr);// OK
int g(int =0, int =0, char*);          // error
int h(int =0, int, char* =nullptr);    // error
```

Note that the space between the * and the = is significant (*= is an assignment operator; §10.3):

```
int nasty(char*=nullptr);              // syntax error
```

A default argument cannot be repeated or changed in a subsequent declaration in the same scope. For example:

```
void f(int x = 7);
void f(int = 7);            // error: cannot repeat default argument
void f(int = 8);            // error: different default arguments
```

```
void g()
{
    void f(int x = 9);      // OK: this declaration hides the outer one
    // ...
}
```

Declaring a name in a nested scope so that the name hides a declaration of the same name in an outer scope is error-prone.

12.3 Overloaded Functions

Most often, it is a good idea to give different functions different names, but when different functions conceptually perform the same task on objects of different types, it can be more convenient to give them the same name. Using the same name for operations on different types is called *overloading*. The technique is already used for the basic operations in C++. That is, there is only one name for addition, **+**, yet it can be used to add values of integer and floating-point types and combinations of such types. This idea is easily extended to functions defined by the programmer. For example:

```
void print(int);            // print an int
void print(const char*);    // print a C-style string
```

As far as the compiler is concerned, the only thing functions of the same name have in common is that name. Presumably, the functions are in some sense similar, but the language does not constrain or aid the programmer. Thus, overloaded function names are primarily a notational convenience. This convenience is significant for functions with conventional names such as **sqrt**, **print**, and **open**. When a name is semantically significant, this convenience becomes essential. This happens, for example, with operators such as **+**, *, and **<<**, in the case of constructors (§16.2.5, §17.1), and in generic programming (§4.5, Chapter 32).

Templates provide a systematic way of defining sets of overloaded functions (§23.5).

12.3.1 Automatic Overload Resolution

When a function **fct** is called, the compiler must determine which of the functions named **fct** to invoke. This is done by comparing the types of the actual arguments with the types of the parameters of all functions in scope called **fct**. The idea is to invoke the function that is the best match to the arguments and give a compile-time error if no function is the best match. For example:

```
void print(double);
void print(long);

void f()
{
    print(1L);      // print(long)
    print(1.0);     // print(double)
    print(1);       // error, ambiguous: print(long(1)) or print(double(1))?
}
```

To approximate our notions of what is reasonable, a series of criteria are tried in order:

[1] Exact match; that is, match using no or only trivial conversions (for example, array name to pointer, function name to pointer to function, and **T** to **const T**)

[2] Match using promotions; that is, integral promotions (**bool** to **int**, **char** to **int**, **short** to **int**, and their **unsigned** counterparts; §10.5.1) and **float** to **double**

[3] Match using standard conversions (e.g., **int** to **double**, **double** to **int**, **double** to **long double**, **Derived*** to **Base*** (§20.2), **T*** to **void*** (§7.2.1), **int** to **unsigned int** (§10.5))

[4] Match using user-defined conversions (e.g., **double** to **complex<double>**; §18.4)

[5] Match using the ellipsis **...** in a function declaration (§12.2.4)

If two matches are found at the highest level where a match is found, the call is rejected as ambiguous. The resolution rules are this elaborate primarily to take into account the elaborate C and C++ rules for built-in numeric types (§10.5). For example:

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);

void h(char c, int i, short s, float f)
{
        print(c);       // exact match: invoke print(char)
        print(i);       // exact match: invoke print(int)
        print(s);       // integral promotion: invoke print(int)
        print(f);       // float to double promotion: print(double)

        print('a');     // exact match: invoke print(char)
        print(49);      // exact match: invoke print(int)
        print(0);       // exact match: invoke print(int)
        print("a");     // exact match: invoke print(const char*)
        print(nullptr); // nullptr_t to const char* promotion: invoke print(cost char*)
}
```

The call **print(0)** invokes **print(int)** because **0** is an **int**. The call **print('a')** invokes **print(char)** because **'a'** is a **char** (§6.2.3.2). The reason to distinguish between conversions and promotions is that we want to prefer safe promotions, such as **char** to **int**, over unsafe conversions, such as **int** to **char**. See also §12.3.5.

Overload resolution is independent of the order of declaration of the functions considered.

Function templates are handled by applying the overload resolution rules to the result of specialization based on a set of arguments (§23.5.3). There are separate rules for overloading when a **{}**-list is used (initializer lists take priority; §12.2.3, §17.3.4.1) and for rvalue reference template arguments (§23.5.2.1).

Overloading relies on a relatively complicated set of rules, and occasionally a programmer will be surprised which function is called. So, why bother? Consider the alternative to overloading. Often, we need similar operations performed on objects of several types. Without overloading, we must define several functions with different names:

```
void print_int(int);
void print_char(char);
void print_string(const char*);      // C-style string

void g(int i, char c, const char* p, double d)
{
        print_int(i);           // OK
        print_char(c);          // OK
        print_string(p);        // OK

        print_int(c);           // OK? calls print_int(int(c)), prints a number
        print_char(i);          // OK? calls print_char(char(i)), narrowing
        print_string(i);        // error
        print_int(d);           // OK? calls print_int(int(d)), narrowing
}
```

Compared to the overloaded **print()**, we have to remember several names and remember to use those correctly. This can be tedious, defeats attempts to do generic programming (§4.5), and generally encourages the programmer to focus on relatively low-level type issues. Because there is no overloading, all standard conversions apply to arguments to these functions. It can also lead to errors. In the previous example, this implies that only one of the four calls with doubtful semantics is caught by the compiler. In particular, two calls rely on error-prone narrowing (§2.2.2, §10.5). Thus, overloading can increase the chances that an unsuitable argument will be rejected by the compiler.

12.3.2 Overloading and Return Type

Return types are not considered in overload resolution. The reason is to keep resolution for an individual operator (§18.2.1, §18.2.5) or function call context-independent. Consider:

```
float sqrt(float);
double sqrt(double);

void f(double da, float fla)
{
        float fl = sqrt(da);    // call sqrt(double)
        double d = sqrt(da);    // call sqrt(double)
        fl = sqrt(fla);         // call sqrt(float)
        d = sqrt(fla);          // call sqrt(float)
}
```

If the return type were taken into account, it would no longer be possible to look at a call of **sqrt()** in isolation and determine which function was called.

12.3.3 Overloading and Scope

Overloading takes place among the members of an overload set. By default, that means the functions of a single scope; functions declared in different non-namespace scopes do not overload. For example:

```
void f(int);

void g()
{
    void f(double);
    f(1);           // call f(double)
}
```

Clearly, `f(int)` would have been the best match for `f(1)`, but only `f(double)` is in scope. In such cases, local declarations can be added or subtracted to get the desired behavior. As always, intentional hiding can be a useful technique, but unintentional hiding is a source of surprises.

A base class and a derived class provide different scopes so that overloading between a base class function and a derived class function doesn't happen by default. For example:

```
struct Base {
    void f(int);
};

struct Derived : Base {
    void f(double);
};

void g(Derived& d)
{
    d.f(1);     // call Derived::f(double);
}
```

When overloading across class scopes (§20.3.5) or namespace scopes (§14.4.5) is wanted, `using`-declarations or `using`-directives can be used (§14.2.2). Argument-dependent lookup (§14.2.4) can also lead to overloading across namespaces.

12.3.4 Resolution for Multiple Arguments

We can use the overload resolution rules to select the most appropriate function when the efficiency or precision of computations differs significantly among types. For example:

```
int pow(int, int);
double pow(double, double);
complex pow(double, complex);
complex pow(complex, int);
complex pow(complex, complex);

void k(complex z)
{
    int i = pow(2,2);           // invoke pow(int,int)
    double d = pow(2.0,2.0);    // invoke pow(double,double)
    complex z2 = pow(2,z);      // invoke pow(double,complex)
    complex z3 = pow(z,2);      // invoke pow(complex,int)
    complex z4 = pow(z,z);      // invoke pow(complex,complex)
}
```

In the process of choosing among overloaded functions with two or more arguments, a best match is found for each argument using the rules from §12.3. A function that is the best match for one argument and a better or equal match for all other arguments is called. If no such function exists, the call is rejected as ambiguous. For example:

```
void g()
{
    double d = pow(2.0,2);    // error: pow(int(2.0),2) or pow(2.0,double(2))?
}
```

The call is ambiguous because `2.0` is the best match for the first argument of `pow(double,double)` and `2` is the best match for the second argument of `pow(int,int)`.

12.3.5 Manual Overload Resolution

Declaring too few (or too many) overloaded versions of a function can lead to ambiguities. For example:

```
void f1(char);
void f1(long);

void f2(char*);
void f2(int*);

void k(int i)
{
    f1(i);      // ambiguous: f1(char) or f1(long)?
    f2(0);      // ambiguous: f2(char*) or f2(int*)?
}
```

Where possible, consider the set of overloaded versions of a function as a whole and see if it makes sense according to the semantics of the function. Often the problem can be solved by adding a version that resolves ambiguities. For example, adding

```
inline void f1(int n) { f1(long(n)); }
```

would resolve all ambiguities similar to `f1(i)` in favor of the larger type `long int`.

One can also add an explicit type conversion to resolve a specific call. For example:

```
f2(static_cast<int*>(0));
```

However, this is most often simply an ugly stopgap. Soon another similar call will be made and have to be dealt with.

Some C++ novices get irritated by the ambiguity errors reported by the compiler. More experienced programmers appreciate these error messages as useful indicators of design errors.

12.4 Pre- and Postconditions

Every function has some expectations on its arguments. Some of these expectations are expressed in the argument types, but others depend on the actual values passed and on relationships among

argument values. The compiler and linker can ensure that arguments are of the right types, but it is up to the programmer to decide what to do about ''bad'' argument values. We call logical criteria that are supposed to hold when a function is called *preconditions*, and logical criteria that are supposed to hold when a function returns its *postconditions*. For example:

```
int area(int len, int wid)
/*
        calculate the area of a rectangle

        precondition: len and wid are positive

        postcondition: the return value is positive

        postcondition: the return value is the area of a rectange with sides len and wid
*/
{
        return len*wid;
}
```

Here, the statements of the pre- and postconditions are longer than the function body. This may seem excessive, but the information provided is useful to the implementer, to the users of **area()**, and to testers. For example, we learn that **0** and **–12** are not considered valid arguments. Furthermore, we note that we could pass a couple of huge values without violating the precondition, but if **len*wid** overflows either or both of the postconditions are not met.

What should we do about a call **area(numeric_limits<int>::max(),2)**?

[1] Is it the caller's task to avoid it? Yes, but what if the caller doesn't?

[2] Is it the implementer's task to avoid it? If so, how is an error to be handled?

There are several possible answers to these questions. It is easy for a caller to make a mistake and fail to establish a precondition. It is also difficult for an implementer to cheaply, efficiently, and completely check preconditions. We would like to rely on the caller to get the preconditions right, but we need a way to test for correctness. For now, just note that some pre- and postconditions are easy to check (e.g., **len** is positive and **len*wid** is positive). Others are semantic in nature and hard to test directly. For example, how do we test ''the return value is the area of a rectangle with sides **len** and **wid**''? This is a semantic constraint because we have to know the meaning of ''area of a rectangle,'' and just trying to multiply **len** and **wid** again with a precision that precluded overflow could be costly.

It seems that writing out the pre- and postconditions for **area()** uncovered a subtle problem with this very simple function. This is not uncommon. Writing out pre- and postconditions is a great design tool and provides good documentation. Mechanisms for documenting and enforcing conditions are discussed in §13.4.

If a function depends only on its arguments, its preconditions are on its arguments only. However, we have to be careful about functions that depend on non-local values (e.g., a member function that depends on the state of its object). In essence, we have to consider every nonlocal value read as an implicit argument to a function. Similarly, the postcondition of a function without side effects simply states that a value is correctly computed, but if a function writes to nonlocal objects, its effect must be considered and documented.

The writer of a function has several alternatives, including:

[1] Make sure that every input has a valid result (so that we don't have a precondition).
[2] Assume that the precondition holds (rely on the caller not to make mistakes).
[3] Check that the precondition holds and throw an exception if it does not.
[4] Check that the precondition holds and terminate the program if it does not.

If a postconditon fails, there was either an unchecked precondition or a programming error. §13.4 discusses ways to represent alternative strategies for checking.

12.5 Pointer to Function

Like a (data) object, the code generated for a function body is placed in memory somewhere, so it has an address. We can have a pointer to a function just as we can have a pointer to an object. However, for a variety of reasons – some related to machine architecture and others to system design – a pointer to function does not allow the code to be modified. There are only two things one can do to a function: call it and take its address. The pointer obtained by taking the address of a function can then be used to call the function. For example:

```
void error(string s) { /* ... */ }

void (*efct)(string);       // pointer to function taking a string argument and returning nothing

void f()
{
    efct = &error;          // efct points to error
    efct("error");          // call error through efct
}
```

The compiler will discover that **efct** is a pointer and call the function pointed to. That is, dereferencing a pointer to function using * is optional. Similarly, using **&** to get the address of a function is optional:

```
void (*f1)(string) = &error;    // OK: same as = error
void (*f2)(string) = error;     // OK: same as = &error

void g()
{
    f1("Vasa");                 // OK: same as (*f1)("Vasa")
    (*f1)("Mary Rose");         // OK: as f1("Mary Rose")
}
```

Pointers to functions have argument types declared just like the functions themselves. In pointer assignments, the complete function type must match exactly. For example:

```
void (*pf)(string);     // pointer to void(string)
void f1(string);        // void(string)
int f2(string);         // int(string)
void f3(int*);          // void(int*)
```

```
void f()
{
    pf = &f1;           // OK
    pf = &f2;           // error: bad return type
    pf = &f3;           // error: bad argument type

    pf("Hera");         // OK
    pf(1);              // error: bad argument type

    int i = pf("Zeus"); // error: void assigned to int
}
```

The rules for argument passing are the same for calls directly to a function and for calls to a function through a pointer.

You can convert a pointer to function to a different pointer-to-function type, but you must cast the resulting pointer back to its original type or strange things may happen:

```
using P1 = int(*)(int*);
using P2 = void(*)(void);

void f(P1 pf)
{
    P2 pf2 = reinterpret_cast<P2>(pf)
    pf2();                                  // likely serious problem
    P1 pf1 = reinterpret_cast<P1>(pf2);     // convert pf2 "back again"
    int x = 7;
    int y = pf1(&x);                        // OK
    // ...
}
```

We need the nastiest of casts, **reinterpret_cast**, to do conversion of pointer-to-function types. The reason is that the result of using a pointer to function of the wrong type is so unpredictable and system-dependent. For example, in the example above, the called function may write to the object pointed to by its argument, but the call **pf2()** didn't supply any argument!

Pointers to functions provide a way of parameterizing algorithms. Because C does not have function objects (§3.4.3) or lambda expressions (§11.4), pointers to functions are widely used as function arguments in C-style code. For example, we can provide the comparison operation needed by a sorting function as a pointer to function:

```
using CFT = int(const void*, const void*);

void ssort(void* base, size_t n, size_t sz, CFT cmp)
/*
    Sort the "n" elements of vector "base" into increasing order
    using the comparison function pointed to by "cmp".
    The elements are of size "sz".

    Shell sort (Knuth, Vol3, pg84)
*/
```

```
{
    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i!=n; i++)
            for (int j=i–gap; 0<=j; j–=gap) {
                char* b = static_cast<char*>(base);      // necessary cast
                char* pj = b+j*sz;                       // &base[j]
                char* pjg = b+(j+gap)*sz;                // &base[j+gap]
                if (cmp(pjg,pj)<0) {                     // swap base[j] and base[j+gap]:
                    for (int k=0; k!=sz; k++) {
                        char temp = pj[k];
                        pj[k] = pjg[k];
                        pjg[k] = temp;
                    }
                }
            }
}
```

The **ssort()** routine does not know the type of the objects it sorts, only the number of elements (the array size), the size of each element, and the function to call to perform a comparison. The type of **ssort()** was chosen to be the same as the type of the standard C library sort routine, **qsort()**. Real programs use **qsort()**, the C++ standard-library algorithm **sort** (§32.6), or a specialized sort routine. This style of code is common in C, but it is not the most elegant way of expressing this algorithm in C++ (see §23.5, §25.3.4.1).

Such a sort function could be used to sort a table such as this:

```
struct User {
    const char* name;
    const char* id;
    int dept;
};

vector<User> heads = {
    "Ritchie D.M.",     "dmr",    11271,
    "Sethi R.",         "ravi",   11272,
    "Szymanski T.G.",   "tgs",    11273,
    "Schryer N.L.",     "nls",    11274,
    "Schryer N.L.",     "nls",    11275,
    "Kernighan B.W.",   "bwk",    11276
};

void print_id(vector<User>& v)
{
    for (auto& x : v)
        cout << x.name << '\t' << x.id << '\t' << x.dept << '\n';
}
```

To be able to sort, we must first define appropriate comparison functions. A comparison function must return a negative value if its first argument is less than the second, zero if the arguments are equal, and a positive number otherwise:

```
int cmp1(const void* p, const void* q)    // Compare name strings
{
    return strcmp(static_cast<const User*>(p)->name,static_cast<const User*>(q)->name);
}

int cmp2(const void* p, const void* q)    // Compare dept numbers
{
    return static_cast<const User*>(p)->dept – static_cast<const User*>(q)->dept;
}
```

There is no implicit conversion of argument or return types when pointers to functions are assigned or initialized. This means that you cannot avoid the ugly and error-prone casts by writing:

```
int cmp3(const User* p, const User* q)  // Compare ids
{
    return strcmp(p->id,q->id);
}
```

The reason is that accepting **cmp3** as an argument to **ssort()** would violate the guarantee that **cmp3** will be called with arguments of type **const User*** (see also §15.2.6).

This program sorts and prints:

```
int main()
{
    cout << "Heads in alphabetical order:\n";
    ssort(heads,6,sizeof(User),cmp1);
    print_id(heads);
    cout << '\n';

    cout << "Heads in order of department number:\n";
    ssort(heads,6,sizeof(User),cmp2);
    print_id(heads);
}
```

To compare, we can equivalently write:

```
int main()
{
    cout << "Heads in alphabetical order:\n";
    sort(heads.begin(), head.end(),
        [](const User& x, const User& y) { return x.name<y.name; }
    );
    print_id(heads);
    cout << '\n';

    cout << "Heads in order of department number:\n";
    sort(heads.begin(), head.end(),
        [](const User& x, const User& y) { return x.dept<y.dept; }
    );
    print_id(heads);
}
```

No mention of sizes is needed nor any helper functions. If the explicit use of **begin()** and **end()** is annoying, it can be eliminated by using a version of **sort()** that takes a container (§14.4.5):

```
sort(heads,[](const User& x, const User& y) { return x.name<y.name; });
```

You can take the address of an overloaded function by assigning to or initializing a pointer to function. In that case, the type of the target is used to select from the set of overloaded functions. For example:

```
void f(int);
int f(char);

void (*pf1)(int) = &f;      // void f(int)
int (*pf2)(char) = &f;      // int f(char)
void (*pf3)(char) = &f;     // error: no void f(char)
```

It is also possible to take the address of member functions (§20.6), but a pointer to member function is quite different from a pointer to (nonmember) function.

A pointer to a **noexcept** function can be declared **noexcept**. For example:

```
void f(int) noexcept;
void g(int);

void (*p1)(int) = f;                // OK: but we throw away useful information
void (*p2)(int) noexcept = f;       // OK: we preserve the noexcept information
void (*p3)(int) noexcept = g;       // error: we don't know that g doesn't throw
```

A pointer to function must reflect the linkage of a function (§15.2.6). Neither linkage specification nor **noexcept** may appear in type aliases:

```
using Pc = extern "C" void(int);    // error: linkage specification in alias
using Pn = void(int) noexcept;      // error: noexcept in alias
```

12.6 Macros

Macros are very important in C but have far fewer uses in C++. The first rule about macros is: don't use them unless you have to. Almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer. Because they rearrange the program text before the compiler proper sees it, macros are also a major problem for many programming support tools. So when you use macros, you should expect inferior service from tools such as debuggers, cross-reference tools, and profilers. If you must use macros, please read the reference manual for your own implementation of the C++ preprocessor carefully and try not to be too clever. Also, to warn readers, follow the convention to name macros using lots of capital letters. The syntax of macros is presented in §iso.16.3.

I recommend using macros only for conditional compilation (§12.6.1) and in particular for include guards (§15.3.3).

A simple macro is defined like this:

```
#define NAME rest of line
```

Where **NAME** is encountered as a token, it is replaced by **rest of line**. For example:

```
named = NAME
```

will expand into

```
named = rest of line
```

A macro can also be defined to take arguments. For example:

```
#define MAC(x,y) argument1: x argument2: y
```

When **MAC** is used, two argument strings must be presented. They will replace **x** and **y** when **MAC()** is expanded. For example:

```
expanded = MAC(foo bar, yuk yuk)
```

will be expanded into

```
expanded = argument1: foo bar argument2: yuk yuk
```

Macro names cannot be overloaded, and the macro preprocessor cannot handle recursive calls:

```
#define PRINT(a,b) cout<<(a)<<(b)
#define PRINT(a,b,c) cout<<(a)<<(b)<<(c)      /* trouble?: redefines, does not overload */

#define FAC(n) (n>1)?n*FAC(n–1):1             /* trouble: recursive macro */
```

Macros manipulate character strings and know little about C++ syntax and nothing about C++ types or scope rules. Only the expanded form of a macro is seen by the compiler, so an error in a macro will be reported when the macro is expanded, not when it is defined. This leads to very obscure error messages.

Here are some plausible macros:

```
#define CASE break;case
#define FOREVER for(;;)
```

Here are some completely unnecessary macros:

```
#define PI 3.141593
#define BEGIN {
#define END }
```

Here are some dangerous macros:

```
#define SQUARE(a) a*a
#define INCR_xx (xx)++
```

To see why they are dangerous, try expanding this:

```
int xx = 0;       // global counter

void f(int xx)
{
    int y = SQUARE(xx+2);    // y=xx+2*xx+2; that is, y=xx+(2*xx)+2
    INCR_xx;                 // increments argument xx (not the global xx)
}
```

If you must use a macro, use the scope resolution operator, ::, when referring to global names (§6.3.4) and enclose occurrences of a macro argument name in parentheses whenever possible. For example:

```
#define MIN(a,b) (((a)<(b))?(a):(b))
```

This handles the simpler syntax problems (which are often caught by compilers), but not the problems with side effects. For example:

```
int x = 1;
int y = 10;
int z = MIN(x++,y++);          // x becomes 3; y becomes 11
```

If you must write macros complicated enough to require comments, it is wise to use /* */ comments because old C preprocessors that do not know about // comments are sometimes used as part of C++ tools. For example:

```
#define M2(a) something(a)     /* thoughtful comment */
```

Using macros, you can design your own private language. Even if you prefer this ''enhanced language'' to plain C++, it will be incomprehensible to most C++ programmers. Furthermore, the preprocessor is a very simple-minded macro processor. When you try to do something nontrivial, you are likely to find it either impossible or unnecessarily hard to do. The **auto**, **constexpr**, **const**, **decltype**, **enum**, **inline**, lambda expressions, **namespace**, and **template** mechanisms can be used as better-behaved alternatives to many traditional uses of preprocessor constructs. For example:

```
const int answer = 42;

template<class T>
inline const T& min(const T& a, const T& b)
{
      return (a<b)?a:b;
}
```

When writing a macro, it is not unusual to need a new name for something. A string can be created by concatenating two strings using the ## macro operator. For example:

```
#define NAME2(a,b) a##b

int NAME2(hack,cah)();
```

will produce

```
int hackcah();
```

A single # before a parameter name in a replacement string means a string containing the macro argument. For example:

```
#define printx(x) cout << #x " = " << x << '\n';

int a = 7;
string str = "asdf";
```

```
void f()
{
        printx(a);      // cout << "a" << " = " << a << '\n';
        printx(str);    // cout << "str" << " = " << str << '\n';
}
```

Writing `#x " = "` rather than `#x << " = "` is obscure ‘‘clever code’’ rather than an error. Adjacent string literals are concatenated (§7.3.2).

The directive

```
#undef X
```

ensures that no macro called `X` is defined – whether or not one was before the directive. This affords some protection against undesired macros. However, it is not always easy to know what the effects of `X` on a piece of code were supposed to be.

The argument list (‘‘replacement list’’) of a macro can be empty:

```
#define EMPTY() std::cout<<"empty\n"
EMPTY();        // print "empty\n"
EMPTY;          // error: macro replacement list missing
```

I have a hard time thinking of uses of an empty macro argument list that are not error-prone or malicious.

Macros can even be variadic. For example:

```
#define err_print(...) fprintf(stderr,"error: %s %d\n", __VA_ARGS__)
err_print("The answer",54);
```

The ellipsis (`...`) means that `__VA_ARGS__` represents the arguments actually passed as a string, so the output is:

```
error: The answer 54
```

12.6.1 Conditional Compilation

One use of macros is almost impossible to avoid. The directive

```
#ifdef IDENTIFIER
```

does nothing if `IDENTIFIER` is defined, but if it is not, the directive causes all input to be ignored until a `#endif` directive is seen. For example:

```
int f(int a
#ifdef arg_two
,int b
#endif
);
```

Unless a macro called `arg_two` has been `#define`d , this produces:

```
int f(int a
);
```

This example confuses tools that assume sane behavior from the programmer.

Most uses of #ifdef are less bizarre, and when used with restraint, #ifdef and its complement #ifndef do little harm. See also §15.3.3.

Names of the macros used to control #ifdef should be chosen carefully so that they don't clash with ordinary identifiers. For example:

```
struct Call_info {
        Node* arg_one;
        Node* arg_two;
        // ...
};
```

This innocent-looking source text will cause some confusion should someone write:

```
#define arg_two x
```

Unfortunately, common and unavoidable headers contain many dangerous and unnecessary macros.

12.6.2 Predefined Macros

A few macros are predefined by the compiler (§iso.16.8, §iso.8.4.1):

- __cplusplus: defined in a C++ compilation (and not in a C compilation). Its value is 201103L in a C++11 program; previous C++ standards have lower values.
- __DATE__: date in "yyyy:mm:dd" format.
- __TIME__: time in "hh:mm:ss" format.
- __FILE__: name of current source file.
- __LINE__: source line number within the current source file.
- __FUNC__: an implementation-defined C-style string naming the current function.
- __STDC_HOSTED__: 1 if the implementation is hosted (§6.1.1); otherwise 0.

In addition, a few macros are conditionally defined by the implementation:

- __STDC__: defined in a C compilation (and not in a C++ compilation)
- __STDC_MB_MIGHT_NEQ_WC__: 1 if, in the encoding for wchar_t, a member of the basic character set (§6.1) might have a code value that differs from its value as an ordinary character literal
- __STDCPP_STRICT_POINTER_SAFETY__: 1 if the implementation has strict pointer safety (§34.5); otherwise undefined.
- __STDCPP_THREADS__: 1 if a program can have more than one thread of execution; otherwise undefined.

For example:

```
cout << __FUNC__ << "() in file " << __FILE__ << " on line " << __LINE__ << "\n";
```

In addition, most C++ implementations allow a user to define arbitrary macros on the command line or in some other form of compile-time environment. For example, NDEBUG is defined unless the compilation is done in (some implementation-specific) "debug mode" and is used by the assert() macro (§13.4). This can be useful, but it does imply that you can't be sure of the meaning of a program just by reading its source text.

12.6.3 Pragmas

Implementations often provide facilities that differ from or go beyond what the standard offers. Obviously, the standard cannot specify how such facilities are provided, but one standard syntax is a line of tokens prefixed with the preprocessor directive **#pragma**. For example:

```
#pragma foo bar 666 foobar
```

If possible, **#pragma**s are best avoided.

12.7 Advice

[1] ‘‘Package’’ meaningful operations as carefully named functions; §12.1.
[2] A function should perform a single logical operation; §12.1.
[3] Keep functions short; §12.1.
[4] Don’t return pointers or references to local variables; §12.1.4.
[5] If a function may have to be evaluated at compile time, declare it **constexpr**; §12.1.6.
[6] If a function cannot return, mark it **[[noreturn]]**; §12.1.7.
[7] Use pass-by-value for small objects; §12.2.1.
[8] Use pass-by-**const**-reference to pass large values that you don’t need to modify; §12.2.1.
[9] Return a result as a **return** value rather than modifying an object through an argument; §12.2.1.
[10] Use rvalue references to implement move and forwarding; §12.2.1.
[11] Pass a pointer if ‘‘no object’’ is a valid alternative (and represent ‘‘no object’’ by **nullptr**); §12.2.1.
[12] Use pass-by-non-**const**-reference only if you have to; §12.2.1.
[13] Use **const** extensively and consistently; §12.2.1.
[14] Assume that a **char*** or a **const char*** argument points to a C-style string; §12.2.2.
[15] Avoid passing arrays as pointers; §12.2.2.
[16] Pass a homogeneous list of unknown length as an **initializer_list<T>** (or as some other container); §12.2.3.
[17] Avoid unspecified numbers of arguments (**...**); §12.2.4.
[18] Use overloading when functions perform conceptually the same task on different types; §12.3.
[19] When overloading on integers, provide functions to eliminate common ambiguities; §12.3.5.
[20] Specify preconditions and postconditions for your functions; §12.4.
[21] Prefer function objects (including lambdas) and virtual functions to pointers to functions; §12.5.
[22] Avoid macros; §12.6.
[23] If you must use macros, use ugly names with lots of capital letters; §12.6.

This page intentionally left blank

13

Exception Handling

Don't interrupt me
while I'm interrupting.
– Winston S. Churchill

- Error Handling
  Exceptions; Traditional Error Handling; Muddling Through; Alternative Views of Exceptions; When You Can't Use Exceptions; Hierarchical Error Handling; Exceptions and Efficiency
- Exception Guarantees
- Resource Management
  Finally
- Enforcing Invariants
- Throwing and Catching Exceptions
  Throwing Exceptions; Catching Exceptions; Exceptions and Threads
- A **vector** Implementation
  A Simple **vector**; Representing Memory Explicitly; Assignment; Changing Size
- Advice

13.1 Error Handling

This chapter presents error handling using exceptions. For effective error handling, the language mechanisms must be used based on a strategy. Consequently, this chapter presents the *exception-safety guarantees* that are central to recovery from run-time errors and the *Resource Acquisition Is Initialization* (RAII) technique for resource management using constructors and destructors. Both the exception-safety guarantees and RAII depend on the specification of *invariants*, so mechanisms for enforcement of assertions are presented.

The language facilities and techniques presented here address problems related to the handling of errors in software; the handling of asynchronous events is a different topic.

The discussion of errors focuses on errors that cannot be handled locally (within a single small function), so that they require separation of error-handling activities into different parts of a program. Such parts of a program are often separately developed. Consequently, I often refer to a part of a program that is invoked to perform a task as ''a library.'' A library is just ordinary code, but in the context of a discussion of error handling it is worth remembering that a library designer often cannot even know what kind of programs the library will become part of:

- The author of a library can detect a run-time error but does not in general have any idea what to do about it.
- The user of a library may know how to cope with a run-time error but cannot easily detect it (or else it would have been handled in the user's code and not left for the library to find).

The discussion of exceptions focuses on problems that need to be handled in long-running systems, systems with stringent reliability requirements, and libraries. Different kinds of programs have different requirements, and the amount of care and effort we expend should reflect that. For example, I would not apply every technique recommended here to a two-page program written just for myself. However, many of the techniques presented here simplify code, so I would use those.

13.1.1 Exceptions

The notion of an *exception* is provided to help get information from the point where an error is detected to a point where it can be handled. A function that cannot cope with a problem *throws* an exception, hoping that its (direct or indirect) caller can handle the problem. A function that wants to handle a kind of problem indicates that by *catch*ing the corresponding exception (§2.4.3.1):

- A calling component indicates the kinds of failures that it is willing to handle by specifying those exceptions in a **catch**-clause of a **try**-block.
- A called component that cannot complete its assigned task reports its failure to do so by throwing an exception using a **throw**-expression.

Consider a simplified and stylized example:

```
void taskmaster()
{
    try {
        auto result = do_task();
        // use result
    }
    catch (Some_error) {
        // failure to do_task: handle problem
    }
}

int do_task()
{
    // ...
    if (/* could perform the task */)
        return result;
    else
        throw Some_error{};
}
```

The taskmaster() asks do_task() to do a job. If do_task() can do that job and return a correct result, all is fine. Otherwise, do_task() must report a failure by throwing some exception. The taskmaster() is prepared to handle a Some_error, but some other kind of exception may be thrown. For example, do_task() may call other functions to do a lot of subtasks, and one of those may throw because it can't do its assigned subtask. An exception different from Some_error indicates a failure of taskmaster() to do its job and must be handled by whatever code invoked taskmaster().

A called function cannot just return with an indication that an error happened. If the program is to continue working (and not just print an error message and terminate), the returning function must leave the program in a good state and not leak any resources. The exception-handling mechanism is integrated with the constructor/destructor mechanisms and the concurrency mechanisms to help ensure that (§5.2). The exception-handling mechanism:

- Is an alternative to the traditional techniques when they are insufficient, inelegant, or error-prone
- Is complete; it can be used to handle all errors detected by ordinary code
- Allows the programmer to explicitly separate error-handling code from ''ordinary code,'' thus making the program more readable and more amenable to tools
- Supports a more regular style of error handling, thus simplifying cooperation between separately written program fragments

An exception is an object thrown to represent the occurrence of an error. It can be of any type that can be copied, but it is strongly recommended to use only user-defined types specifically defined for that purpose. That way, we minimize the chances of two unrelated libraries using the same value, say 17, to represent different errors, thereby throwing our recovery code into chaos.

An exception is caught by code that has expressed interest in handling a particular type of exception (a catch-clause). Thus, the simplest way of defining an exception is to define a class specifically for a kind of error and throw that. For example:

```
struct Range_error {};

void f(int n)
{
    if (n<0 || max<n) throw Range_error {};
    // ...
}
```

If that gets tedious, the standard library defines a small hierarchy of exception classes (§13.5.2).

An exception can carry information about the error it represents. Its type represents the kind of error, and whatever data it holds represents the particular occurrence of that error. For example, the standard-library exceptions contain a string value, which can be used to transmit information such as the location of the throw (§13.5.2).

13.1.2 Traditional Error Handling

Consider the alternatives to exceptions for a function detecting a problem that cannot be handled locally (e.g., an out-of-range access) so that an error must be reported to a caller. Each conventional approach has problems, and none are general:

- *Terminate the program.* This is a pretty drastic approach. For example:

```
if (something_wrong) exit(1);
```

  For most errors, we can and must do better. For example, in most situations we should at least write out a decent error message or log the error before terminating. In particular, a library that doesn't know about the purpose and general strategy of the program in which it is embedded cannot simply **exit()** or **abort()**. A library that unconditionally terminates cannot be used in a program that cannot afford to crash.
- *Return an error value.* This is not always feasible because there is often no acceptable "error value." For example:

```
int get_int();    // get next integer from input
```

  For this input function, *every* **int** is a possible result, so there can be no integer value representing an input failure. At a minimum, we would have to modify **get_int()** to return a pair of values. Even where this approach is feasible, it is often inconvenient because every call must be checked for the error value. This can easily double the size of a program (§13.1.7). Also, callers often ignore the possibility of errors or simply forget to test a return value. Consequently, this approach is rarely used systematically enough to detect all errors. For example, **printf()** (§43.3) returns a negative value if an output or encoding error occurred, but programmers essentially never test for that. Finally, some operations simply do not have return values; a constructor is the obvious example.
- *Return a legal value and leave the program in an "error state."* This has the problem that the calling function may not notice that the program has been put in an error state. For example, many standard C library functions set the nonlocal variable **errno** to indicate an error (§43.4, §40.3):

```
double d = sqrt(–1.0);
```

  Here, the value of **d** is meaningless and **errno** is set to indicate that **–1.0** isn't an acceptable argument for a floating-point square root function. However, programs typically fail to set and test **errno** and similar nonlocal state consistently enough to avoid consequential errors caused by values returned from failed calls. Furthermore, the use of nonlocal variables for recording error conditions doesn't work well in the presence of concurrency.
- *Call an error-handler function.* For example:

```
if (something_wrong) something_handler();    // and possibly continue here
```

  This must be some other approach in disguise because the problem immediately becomes "What does the error-handling function do?" Unless the error-handling function can completely resolve the problem, the error-handling function must in turn either terminate the program, return with some indication that an error had occurred, set an error state, or throw an exception. Also, if the error-handling function can handle the problem without bothering the ultimate caller, why do we consider it an error?

Traditionally, an unsystematic combination of these approached co-exists in a program.

13.1.3 Muddling Through

One aspect of the exception-handling scheme that will appear novel to some programmers is that the ultimate response to an unhandled error (an uncaught exception) is to terminate the program. The traditional response has been to muddle through and hope for the best. Thus, exception handling makes programs more ''brittle'' in the sense that more care and effort must be taken to get a program to run acceptably. This is preferable, though, to getting wrong results later in the development process – or after the development process is considered complete and the program is handed over to innocent users. Where termination is unacceptable, we can catch all exceptions (§13.5.2.2). Thus, an exception terminates a program only if a programmer allows it to terminate. Typically, this is preferable to the unconditional termination that happens when a traditional incomplete recovery leads to a catastrophic error. Where termination is an acceptable response, an uncaught exception will achieve that because it turns into a call of **terminate()** (§13.5.2.5). Also, a **noexcept** specifier (§13.5.1.1) can make that desire explicit.

Sometimes, people try to alleviate the unattractive aspects of ''muddling through'' by writing out error messages, putting up dialog boxes asking the user for help, etc. Such approaches are primarily useful in debugging situations in which the user is a programmer familiar with the structure of the program. In the hands of nondevelopers, a library that asks the (possibly absent) user/operator for help is unacceptable. A good library doesn't ''blabber'' in this way. If a user has to be informed, an exception handler can compose a suitable message (e.g., in Finnish for Finnish users or in XML for an error-logging system). Exceptions provide a way for code that detects a problem from which it cannot recover to pass the problem on to a part of the system that might be able to recover. Only a part of the system that has some idea of the context in which the program runs has any chance of composing a meaningful error message.

Please recognize that error handling will remain a difficult task and that the exception-handling mechanism – although more formalized than the techniques it replaces – is still relatively unstructured compared with language features involving only local control flow. The C++ exception-handling mechanism provides the programmer with a way of handling errors where they are most naturally handled, given the structure of a system. Exceptions make the complexity of error handling visible. However, exceptions are not the cause of that complexity. Be careful not to blame the messenger for bad news.

13.1.4 Alternative Views of Exceptions

''Exception'' is one of those words that means different things to different people. The C++ exception-handling mechanism is designed to support handling of errors that cannot be handled locally (''exceptional conditions''). In particular, it is intended to support error handling in programs composed of independently developed components. Given that there is nothing particularly exceptional about a part of a program being unable to perform its given task, the word ''exception'' may be considered a bit misleading. Can an event that happens most times a program is run be considered exceptional? Can an event that is planned for and handled be considered an error? The answer to both questions is ''yes.'' ''Exceptional'' does not mean ''almost never happens'' or ''disastrous.''

13.1.4.1 Asynchronous Events

The mechanism is designed to handle only synchronous exceptions, such as array range checks and I/O errors. Asynchronous events, such as keyboard interrupts and power failures, are not necessarily exceptional and are not handled directly by this mechanism. Asynchronous events require mechanisms fundamentally different from exceptions (as defined here) to handle them cleanly and efficiently. Many systems offer mechanisms, such as signals, to deal with asynchrony, but because these tend to be system-dependent, they are not described here.

13.1.4.2 Exceptions That Are Not Errors

Think of an exception as meaning ''some part of the system couldn't do what it was asked to do'' (§13.1.1, §13.2).

Exception **throw**s should be infrequent compared to function calls or the structure of the system has been obscured. However, we should expect most large programs to **throw** and **catch** at least some exceptions in the course of a normal and successful run.

If an exception is expected and caught so that it has no bad effects on the behavior of the program, then how can it be an error? Only because the programmer thinks of it as an error and of the exception-handling mechanisms as tools for handling errors. Alternatively, one might think of the exception-handling mechanisms as simply another control structure, an alternative way of returning a value to a caller. Consider a binary tree search function:

```
void fnd(Tree* p, const string& s)
{
        if (s == p->str) throw p;          // found s
        if (p->left) fnd(p->left,s);
        if (p->right) fnd(p->right,s);
}

Tree* find(Tree* p, const string& s)
{
        try {
                fnd(p,s);
        }
        catch (Tree* q) {       // q->str==s
                return q;
        }
        return 0;
}
```

This actually has some charm, but it should be avoided because it is likely to cause confusion and inefficiencies. When at all possible, stick to the ''exception handling is error handling'' view. When this is done, code is clearly separated into two categories: ordinary code and error-handling code. This makes code more comprehensible. Furthermore, the implementations of the exception mechanisms are optimized based on the assumption that this simple model underlies the use of exceptions.

Error handling is inherently difficult. Anything that helps preserve a clear model of what is an error and how it is handled should be treasured.

13.1.5 When You Can't Use Exceptions

Use of exceptions is the only fully general and systematic way of dealing with errors in a C++ program. However, we must reluctantly conclude that there are programs that for practical and historical reasons cannot use exceptions. For example:

- A time-critical component of an embedded system where an operation must be guaranteed to complete in a specific maximum time. In the absence of tools that can accurately estimate the maximum time for an exception to propagate from a **throw** to a **catch**, alternative error-handling methods must be used.
- A large old program in which resource management is an ad hoc mess (e.g., free store is unsystematically "managed" using "naked" pointers, **new**s, and **delete**s), rather than relying on some systematic scheme, such as resource handles (e.g., **string** and **vector**; §4.2, §4.4).

In such cases, we are thrown back onto "traditional" (pre-exception) techniques. Because such programs arise in a great variety of historical contexts and in response to a variety of constraints, I cannot give a general recommendation for how to handle them. However, I can point to two popular techniques:

- To mimic RAII, give every class with a constructor an **invalid()** operation that returns some **error_code**. A useful convention is for **error_code==0** to represent success. If the constructor fails to establish the class invariant, it ensures that no resource is leaked and **invalid()** returns a nonzero **error_code**. This solves the problem of how to get an error condition out of a constructor. A user can then systematically test **invalid()** after each construction of an object and engage in suitable error handling in case of failure. For example:

```
void f(int n)
{
    my_vector<int> x(n);
    if (x.invalid()) {
        // ... deal with error ...
    }
    // ...
}
```

- To mimic a function either returning a value or throwing an exception, a function can return a **pair<Value,Error_code>** (§5.4.3). A user can then systematically test the **error_code** after each function call and engage in suitable error handling in case of failure. For example:

```
void g(int n)
{
    auto v = make_vector(n); // return a pair
    if (v.second) {
        // ... deal with error ...
    }
    auto val = v.first;
    // ...
}
```

Variations of this scheme have been reasonably successful, but they are clumsy compared to using exceptions in a systematic manner.

13.1.6 Hierarchical Error Handling

The purpose of the exception-handling mechanisms is to provide a means for one part of a program to inform another part that a requested task could not be performed (that an ''exceptional circumstance'' has been detected). The assumption is that the two parts of the program are written independently and that the part of the program that handles the exception often can do something sensible about the error.

To use handlers effectively in a program, we need an overall strategy. That is, the various parts of the program must agree on how exceptions are used and where errors are dealt with. The exception-handling mechanisms are inherently nonlocal, so adherence to an overall strategy is essential. This implies that the error-handling strategy is best considered in the earliest phases of a design. It also implies that the strategy must be simple (relative to the complexity of the total program) and explicit. Something complicated would not be consistently adhered to in an area as inherently tricky as error recovery.

Successful fault-tolerant systems are multilevel. Each level copes with as many errors as it can without getting too contorted and leaves the rest to higher levels. Exceptions support that view. Furthermore, **terminate()** supports this view by providing an escape if the exception-handling mechanism itself is corrupted or if it has been incompletely used, thus leaving exceptions uncaught. Similarly, **noexcept** provides a simple escape for errors where trying to recover seems infeasible.

Not every function should be a firewall. That is, not every function can test its preconditions well enough to ensure that no errors could possibly stop it from meeting its postcondition. The reasons that this will not work vary from program to program and from programmer to programmer. However, for larger programs:

[1] The amount of work needed to ensure this notion of ''reliability'' is too great to be done consistently.

[2] The overhead in time and space is too great for the system to run acceptably (there will be a tendency to check for the same errors, such as invalid arguments, over and over again).

[3] Functions written in other languages won't obey the rules.

[4] This purely local notion of ''reliability'' leads to complexities that actually become a burden to overall system reliability.

However, separating the program into distinct subsystems that either complete successfully or fail in well-defined ways is essential, feasible, and economical. Thus, major libraries, subsystems, and key interface functions should be designed in this way. Furthermore, in most systems, it is feasible to design every function to ensure that it always either completes successfully or fails in a well-defined manner.

Usually, we don't have the luxury of designing all of the code of a system from scratch. Therefore, to impose a general error-handling strategy on all parts of a program, we must take into account program fragments implemented using strategies different from ours. To do this we must address a variety of concerns relating to the way a program fragment manages resources and the state in which it leaves the system after an error. The aim is to have the program fragment appear to follow the general error-handling strategy even if it internally follows a different strategy.

Occasionally, it is necessary to convert from one style of error reporting to another. For example, we might check **errno** and possibly throw an exception after a call to a C library or, conversely, catch an exception and set **errno** before returning to a C program from a C++ library:

```
void callC()    // Call a C function from C++; convert errno to a throw
{
        errno = 0;
        c_function();
        if (errno) {
                // ... local cleanup, if possible and necessary ...
                throw C_blewit(errno);
        }
}

extern "C" void call_from_C() noexcept        // Call a C++ function from C; convert a throw to errno
{
        try {
                c_plus_plus_function();
        }
        catch (...) {
                // ... local cleanup, if possible and necessary ...
                errno = E_CPLPLFCTBLEWIT;
        }
}
```

In such cases, it is important to be systematic enough to ensure that the conversion of error-reporting styles is complete. Unfortunately, such conversions are often most desirable in ''messy code'' without a clear error-handling strategy and therefore difficult to be systematic about.

Error handling should be – as far as possible – hierarchical. If a function detects a run-time error, it should not ask its caller for help with recovery or resource acquisition. Such requests set up cycles in the system dependencies. That in turn makes the program hard to understand and introduces the possibility of infinite loops in the error-handling and recovery code.

13.1.7 Exceptions and Efficiency

In principle, exception handling can be implemented so that there is no run-time overhead when no exception is thrown. In addition, this can be done so that throwing an exception isn't all that expensive compared to calling a function. Doing so without adding significant memory overhead while maintaining compatibility with C calling sequences, debugger conventions, etc., is possible, but hard. However, please remember that the alternatives to exceptions are not free either. It is not unusual to find traditional systems in which half of the code is devoted to error handling.

Consider a simple function `f()` that appears to have nothing to do with exception handling:

```
void f()
{
        string buf;
        cin>>buf;
        // ...
        g(1);
        h(buf);
}
```

However, g() or h() may throw an exception, so f() must contain code ensuring that buf is destroyed correctly in case of an exception.

Had g() not thrown an exception, it would have had to report its error some other way. Consequently, the comparable code using ordinary code to handle errors instead of exceptions isn't the plain code above, but something like:

```
bool g(int);
bool h(const char*);
char* read_long_string();

bool f()
{
        char* s = read_long_string();
        // ...
        if (g(1)) {
                if (h(s)) {
                        free(s);
                        return true;
                }
                else {
                        free(s);
                        return false;
                }
        }
        else {
                free(s);
                return false;
        }
}
```

Using a local buffer for s would simplify the code by eliminating the calls to free(), but then we'd have range-checking code instead. Complexity tends to move around rather than just disappear.

People don't usually handle errors this systematically, though, and it is not always critical to do so. However, when careful and systematic handling of errors is necessary, such housekeeping is best left to a computer, that is, to the exception-handling mechanisms.

The noexcept specifier (§13.5.1.1) can be most helpful in improving generated code. Consider:

```
void g(int) noexcept;
void h(const string&) noexcept;
```

Now, the code generated for f() can possibly be improved.

No traditional C function throws an exception, so most C functions can be declared noexcept. In particular, a standard-library implementer knows that only a few standard C library functions (such as atexit() and qsort()) can throw, and can take advantage of that fact to generate better code.

Before declaring a "C function" noexcept, take a minute to consider if it could possibly throw an exception. For example, it might have been converted to use the C++ operator new, which can throw bad_alloc, or it might call a C++ library that throws an exception.

As ever, discussions about efficiency are meaningless in the absence of measurements.

13.2 Exception Guarantees

To recover from an error – that is, to catch an exception and continue executing a program – we need to know what can be assumed about the state of the program before and after the attempted recovery action. Only then can recovery be meaningful. Therefore, we call an operation *exception-safe* if that operation leaves the program in a valid state when the operation is terminated by throwing an exception. However, for that to be meaningful and useful, we have to be precise about what we mean by ''valid state.'' For practical design using exceptions, we must also break down the overly general ''exception-safe'' notion into a few specific guarantees.

When reasoning about objects, we assume that a class has a class invariant (§2.4.3.2, §17.2.1). We assume that this invariant is established by its constructor and maintained by all functions with access to the object's representation until the object is destroyed. So, by *valid state* we mean that a constructor has completed and the destructor has not yet been entered. For data that isn't easily viewed as an object, we must reason similarly. That is, if two pieces of nonlocal data are assumed to have a specific relationship, we must consider that an invariant and our recovery action must preserve it. For example:

```
namespace Points {      // (vx[i],vy[i]) is a point for all i
    vector<int> vx;
    vector<int> vy;
};
```

Here it is assumed that `vx.size()==vy.size()` is (always) true. However, that was only stated in a comment, and compilers do not read comments. Such implicit invariants can be very hard to discover and maintain.

Before a `throw`, a function must place all constructed objects in valid states. However, such a valid state may be one that doesn't suit the caller. For example, a `string` may be left as the empty string or a container may be left unsorted. Thus, for complete recovery, an error handler may have to produce values that are more appropriate/desirable for the application than the (valid) ones existing at the entry to a `catch`-clause.

The C++ standard library provides a generally useful conceptual framework for design for exception-safe program components. The library provides one of the following guarantees for every library operation:

- The *basic guarantee* for all operations: The basic invariants of all objects are maintained, and no resources, such as memory, are leaked. In particular, the basic invariants of every built-in and standard-library type guarantee that you can destroy an object or assign to it after every standard-library operation (§iso.17.6.3.1).
- The *strong guarantee* for key operations: in addition to providing the basic guarantee, either the operation succeeds, or it has no effect. This guarantee is provided for key operations, such as `push_back()`, single-element `insert()` on a `list`, and `uninitialized_copy()`.
- The *nothrow guarantee* for some operations: in addition to providing the basic guarantee, some operations are guaranteed not to throw an exception. This guarantee is provided for a few simple operations, such as `swap()` of two containers and `pop_back()`.

Both the basic guarantee and the strong guarantee are provided on the condition that

- user-supplied operations (such as assignments and **swap()** functions) do not leave container elements in invalid states,
- user-supplied operations do not leak resources, and
- destructors do not throw exceptions (§iso.17.6.5.12).

Violating a standard-library requirement, such as having a destructor exit by throwing an exception, is logically equivalent to violating a fundamental language rule, such as dereferencing a null pointer. The practical effects are also equivalent and often disastrous.

Both the basic guarantee and the strong guarantee require the absence of resource leaks. This is necessary for every system that cannot afford resource leaks. In particular, an operation that throws an exception must not only leave its operands in well-defined states but must also ensure that every resource that it acquired is (eventually) released. For example, at the point where an exception is thrown, all memory allocated must be either deallocated or owned by some object, which in turn must ensure that the memory is properly deallocated. For example:

```
void f(int i)
{
        int* p = new int[10];
        // ...
        if (i<0) {
                delete[] p;        // delete before the throw or leak
                throw Bad();
        }
        // ...
}
```

Remember that memory isn't the only kind of resource that can leak. I consider anything that has to be acquired from another part of the system and (explicitly or implicitly) given back to be a resource. Files, locks, network connections, and threads are examples of system resources. A function may have to release those or hand them over to some resource handler before throwing an exception.

The C++ language rules for partial construction and destruction ensure that exceptions thrown while constructing subobjects and members will be handled correctly without special attention from standard-library code (§17.2.3). This rule is an essential underpinning for all techniques dealing with exceptions.

In general, we must assume that every function that can throw an exception will throw one. This implies that we must structure our code so that we don't get lost in a rat's nest of complicated control structures and brittle data structures. When analyzing code for potential errors, simple, highly structured, ''stylized'' code is the ideal; §13.6 includes a realistic example of such code.

13.3 Resource Management

When a function acquires a resource – that is, it opens a file, allocates some memory from the free store, acquires a mutex, etc. – it is often essential for the future running of the system that the resource be properly released. Often that ''proper release'' is achieved by having the function that acquired it release it before returning to its caller. For example:

```
void use_file(const char* fn)   // naive code
{
        FILE* f = fopen(fn,"r");

        // ... use f ...

        fclose(f);
}
```

This looks plausible until you realize that if something goes wrong after the call of **fopen()** and before the call of **fclose()**, an exception may cause **use_file()** to be exited without **fclose()** being called. Exactly the same problem can occur in languages that do not support exception handling. For example, the standard C library function **longjmp()** can cause the same problem. Even an ordinary **return**-statement could exit **use_file** without closing **f**.

A first attempt to make **use_file()** fault-tolerant looks like this:

```
void use_file(const char* fn)   // clumsy code
{
        FILE* f = fopen(fn,"r");
        try {
                // ... use f ...
        }
        catch (...) {           // catch every possible exception
                fclose(f);
                throw;
        }
        fclose(f);
}
```

The code using the file is enclosed in a **try**-block that catches every exception, closes the file, and rethrows the exception.

The problem with this solution is that it is verbose, tedious, and potentially expensive. Worse still, such code becomes significantly more complex when several resources must be acquired and released. Fortunately, there is a more elegant solution. The general form of the problem looks like this:

```
void acquire()
{
        // acquire resource 1
        // ...
        // acquire resource n

        // ... use resources ...

        // release resource n
        // ...
        // release resource 1
}
```

It is typically important that resources are released in the reverse order of their acquisition. This

strongly resembles the behavior of local objects created by constructors and destroyed by destructors. Thus, we can handle such resource acquisition and release problems using objects of classes with constructors and destructors. For example, we can define a class **File_ptr** that acts like a **FILE***:

```
class File_ptr {
        FILE* p;
public:
        File_ptr(const char* n, const char* a)     // open file n
                : p{fopen(n,a)}
        {
                if (p==nullptr) throw runtime_error{"File_ptr: Can't open file"};
        }

        File_ptr(const string& n, const char* a)  // open file n
                :File_ptr{n.c_str(),a}
        { }

        explicit File_ptr(FILE* pp)                    // assume ownership of pp
                :p{pp}
        {
                if (p==nullptr) throw runtime_error("File_ptr: nullptr"};
        }

        // ... suitable move and copy operations ...

        ~File_ptr() { fclose(p); }

        operator FILE*() { return p; }
};
```

We can construct a **File_ptr** given either a **FILE*** or the arguments required for **fopen()**. In either case, a **File_ptr** will be destroyed at the end of its scope and its destructor will close the file. **File_ptr** throws an exception if it cannot open a file because otherwise every operation on the file handle would have to test for **nullptr**. Our function now shrinks to this minimum:

```
void use_file(const char* fn)
{
        File_ptr f(fn,"r");
        // ... use f ...
}
```

The destructor will be called independently of whether the function is exited normally or exited because an exception is thrown. That is, the exception-handling mechanisms enable us to remove the error-handling code from the main algorithm. The resulting code is simpler and less error-prone than its traditional counterpart.

This technique for managing resources using local objects is usually referred to as ‘‘Resource Acquisition Is Initialization’’ (RAII; §5.2). This is a general technique that relies on the properties of constructors and destructors and their interaction with exception handling.

It is often suggested that writing a ‘‘handle class’’ (a RAII class) is tedious so that providing a nicer syntax for the `catch(...)` action would provide a better solution. The problem with that approach is that you need to remember to ‘‘catch and correct’’ the problem wherever a resource is acquired in an undisciplined way (typically dozens or hundreds of places in a large program), whereas the handler class need be written only once.

An object is not considered constructed until its constructor has completed. Then and only then will stack unwinding (§13.5.1) call the destructor for the object. An object composed of subobjects is constructed to the extent that its subobjects have been constructed. An array is constructed to the extent that its elements have been constructed (and only fully constructed elements are destroyed during unwinding).

A constructor tries to ensure that its object is completely and correctly constructed. When that cannot be achieved, a well-written constructor restores – as far as possible – the state of the system to what it was before creation. Ideally, a well-designed constructor always achieves one of these alternatives and doesn’t leave its object in some ‘‘half-constructed’’ state. This can be simply achieved by applying the RAII technique to the members.

Consider a class `X` for which a constructor needs to acquire two resources: a file `x` and a mutex `y` (§5.3.4). This acquisition might fail and throw an exception. Class `X`’s constructor must never complete having acquired the file but not the mutex (or the mutex and not the file, or neither). Furthermore, this should be achieved without imposing a burden of complexity on the programmer. We use objects of two classes, `File_ptr` and `std::unique_lock` (§5.3.4), to represent the acquired resources. The acquisition of a resource is represented by the initialization of the local object that represents the resource:

```
class Locked_file_handle {
        File_ptr p;
        unique_lock<mutex> lck;
public:
        X(const char* file, mutex& m)
                : p{file,"rw"},         // acquire “file”
                  lck{m}                // acquire “m”
        {}
        // ...
};
```

Now, as in the local object case, the implementation takes care of all of the bookkeeping. The user doesn’t have to keep track at all. For example, if an exception occurs after `p` has been constructed but before `lck` has been, then the destructor for `p` but not for `lck` will be invoked.

This implies that where this simple model for acquisition of resources is adhered to, the author of the constructor need not write explicit exception-handling code.

The most common resource is memory, and `string`, `vector`, and the other standard containers use RAII to implicitly manage acquisition and release. Compared to ad hoc memory management using `new` (and possibly also `delete`), this saves lots of work and avoids lots of errors.

When a pointer to an object, rather than a local object, is needed, consider using the standard-library types `unique_ptr` and `shared_ptr` (§5.2.1, §34.3) to avoid leaks.

13.3.1 Finally

The discipline required to represent a resource as an object of a class with a destructor have bothered some. Again and again, people have invented ‘‘finally’’ language constructs for writing arbitrary code to clean up after an exception. Such techniques are generally inferior to RAII because they are ad hoc, but if you really want ad hoc, RAII can supply that also. First, we define a class that will execute an arbitrary action from its destructor.

```
template<typename F>
struct Final_action {
        Final_action(F f): clean{f} {}
        ~Final_action() { clean(); }
        F clean;
};
```

The ‘‘finally action’’ is provided as an argument to the constructor.

Next, we define a function that conveniently deduces the type of an action:

```
template<class F>
Final_action<F> finally(F f)
{
        return Final_action<F>(f);
}
```

Finally, we can test `finally()`:

```
void test()
        // handle undiciplined resource acquisition
        // demonstrate that arbitrary actions are possible
{
        int* p = new int{7};                            // probably should use a unique_ptr (§5.2)
        int* buf = (int*)malloc(100*sizeof(int));       // C-style allocation

        auto act1 = finally([&]{    delete p;
                                    free(buf);          // C-style deallocation
                                    cout<< "Goodby, Cruel world!\n";
                                }
                            );

        int var = 0;
        cout << "var = " << var << '\n';

        // nested block:
        {
                var = 1;
                auto act2 = finally([&]{ cout<< "finally!\n"; var=7; });
                cout << "var = " << var << '\n';
        } // act2 is invoked here

        cout << "var = " << var << '\n';
} // act1 is invoked here
```

This produced:

```
var = 0
var = 1
finally!
var = 7
Goodby, Cruel world!
```

In addition, the memory allocated and pointed to by `p` and `buf` is appropriately `delete`d and `free()`d.

It is generally a good idea to place a guard close to the definition of whatever it is guarding. That way, we can at a glance see what is considered a resource (even if ad hoc) and what is to be done at the end of its scope. The connection between `finally()` actions and the resources they manipulate is still ad hoc and implicit compared to the use of RAII for resource handles, but using `finally()` is far better than scattering cleanup code around in a block.

Basically, `finally()` does for a block what the increment part of a `for`-statement does for the `for`-statement (§9.5.2): it specifies the final action at the top of a block where it is easy to be seen and where it logically belongs from a specification point of view. It says what is to be done upon exit from a scope, saving the programmer from trying to write code at each of the potentially many places from which the thread of control might exit the scope.

13.4 Enforcing Invariants

When a precondition for a function (§12.4) isn't met, the function cannot correctly perform its task. Similarly, when a constructor cannot establish its class invariant (§2.4.3.2, §17.2.1), the object is not usable. In those cases, I typically throw exceptions. However, there are programs for which throwing an exception is not an option (§13.1.5), and there are people with different views of how to deal with the failure of a precondition (and similar conditions):

- *Just don't do that:* It is the caller's job to meet preconditions, and if the caller doesn't do that, let bad results occur – eventually those errors will be eliminated from the system through improved design, debugging, and testing.
- *Terminate the program:* Violating a precondition is a serious design error, and the program must not proceed in the presence of such errors. Hopefully, the total system can recover from the failure of one component (that program) – eventually such failures may be eliminated from the system through improved design, debugging, and testing.

Why would anyone choose one of these alternatives? The first approach often relates to the need for performance: systematically checking preconditions can lead to repeated tests of logically unnecessary conditions (for example, if a caller has correctly validated data, millions of tests in thousands of called functions may be logically redundant). The cost in performance can be significant. It may be worthwhile to suffer repeated crashes during testing to gain that performance. Obviously, this assumes that you eventually get all critical precondition violations out of the system. For some systems, typically systems completely under the control of a single organization, that can be a realistic aim.

The second approach tends to be used in systems where complete and timely recovery from a precondition failure is considered infeasible. That is, making sure that recovery is complete imposes unacceptable complexity on the system design and implementation. On the other hand,

termination of a program is considered acceptable. For example, it is not unreasonable to consider program termination acceptable if it is easy to rerun the program with inputs and parameters that make repeated failure unlikely. Some distributed systems are like this (as long as the program that terminates is only a part of the complete system), and so are many of the small programs we write for our own consumption.

Realistically, many systems use a mix of exceptions and these two alternative approaches. All three share a common view that preconditions should be defined and obeyed; what differs is how enforcement is done and whether recovery is considered feasible. Program structure can be radically different depending on whether (localized) recovery is an aim. In most systems, some exceptions are thrown without real expectation of recovery. For example, I often throw an exception to ensure some error logging or to produce a decent error message before terminating or re-initializing a process (e.g., from a `catch(...)` in `main()`).

A variety of techniques are used to express checks of desired conditions and invariants. When we want to be neutral about the logical reason for the check, we typically use the word *assertion*, often abbreviated to an *assert*. An assertion is simply a logical expression that is assumed to be `true`. However, for an assertion to be more than a comment, we need a way of expressing what happens if it is `false`. Looking at a variety of systems, I see a variety of needs when it comes to expressing assertions:

- We need to choose between compile-time asserts (evaluated by the compiler) and run-time asserts (evaluated at run time).
- For run-time asserts we need a choice of throw, terminate, or ignore.
- No code should be generated unless some logical condition is `true`. For example, some run-time asserts should not be evaluated unless the logical condition is `true`. Usually, the logical condition is something like a debug flag, a level of checking, or a mask to select among asserts to enforce.
- Asserts should not be verbose or complicated to write (because they can be very common).

Not every system has a need for or supports every alternative.

The standard offers two simple mechanisms:

- In `<cassert>`, the standard library provides the `assert(A)` macro, which checks its assertion, `A`, at run time if and only if the macro `NDEBUG` (“not debugging”) is not defined (§12.6.2). If the assertion fails, the compiler writes out an error message containing the (failed) assertion, the source file name, and the source file line number and terminates the program.
- The language provides `static_assert(A,message)`, which unconditionally checks its assertion, `A`, at compile time (§2.4.3.3). If the assertion fails, the compiler writes out the `message` and the compilation fails.

Where `assert()` and `static_assert()` are insufficient, we could use ordinary code for checking. For example:

```
void f(int n)
    // n should be in [1:max)
{
    if (2<debug_level && (n<=0 || max<n)
        throw Assert_error("range problem");
    // ...
}
```

However, using such ‘‘ordinary code’’ tends to obscure what is being tested. Are we:

- Evaluating the conditions under which we test? (Yes, the `2<debug_level` part.)
- Evaluating a condition that is expected to be true for some calls and not for others? (No, because we are throwing an exception – unless someone is trying to use exceptions as simply another return mechanism; §13.1.4.2.)
- Checking a precondition which should never fail? (Yes, the exception is simply our chosen response.)

Worse, the precondition testing (or invariant testing) can easily get dispersed in other code and thus be harder to spot and easier to get wrong. What we would like is a recognizable mechanism for checking assertions. What follows here is a (possibly slightly overelaborate) mechanism for expressing a variety of assertions and a variety of responses to failures. First, I define mechanisms for deciding when to test and deciding what to do if an assertion fails:

```
namespace Assert {
    enum class Mode { throw_, terminate_, ignore_ };
    constexpr Mode current_mode = CURRENT_MODE;
    constexpr int current_level = CURRENT_LEVEL;
    constexpr int default_level = 1;

    constexpr bool level(int n)
        { return n<=current_level; }

    struct Error : runtime_error {
        Error(const string& p) :runtime_error(p) {}
    };

    // ...
}
```

The idea is to test whenever an assertion has a ‘‘level’’ lower than or equal to `current_level`. If an assertion fails, `current_mode` is used to choose among three alternatives. The `current_level` and `current_mode` are constants because the idea is to generate no code whatsoever for an assertion unless we have made a decision to do so. Imagine `CURRENT_MODE` and `CURRENT_LEVEL` to be set in the build environment for a program, possibly as compiler options.

The programmer will use `Assert::dynamic()` to make assertions:

```
namespace Assert {
    // ...

    string compose(const char* file, int line, const string& message)
        // compose message including file name and line number
    {
        ostringstream os ("(");
        os << file << "," << line << "):" << message;
        return os.str();
    }
```

```
        template<bool condition =level(default_level), class Except = Error>
        void dynamic(bool assertion, const string& message ="Assert::dynamic failed")
        {
            if (assertion)
                return;
            if (current_mode == Assert_mode::throw_)
                throw Except{message};
            if (current_mode == Assert_mode::terminate_)
                std::terminate();
        }

        template<>
        void dynamic<false,Error>(bool, const string&)     // do nothing
        {
        }

        void dynamic(bool b, const string& s)          // default action
        {
            dynamic<true,Error>(b,s);
        }

        void dynamic(bool b)                           // default message
        {
            dynamic<true,Error>(b);
        }
    }
```

I chose the name **Assert::dynamic** (meaning ''evaluate at run time'') to contrast with **static_assert** (meaning ''evaluate at compile time''; §2.4.3.3).

Further implementation trickery could be used to minimize the amount of code generated. Alternatively, we could do more of the testing at run time if more flexibility is needed. This **Assert** is not part of the standard and is presented primarily as an illustration of the problems and the implementation techniques. I suspect that the demands on an assertion mechanism vary too much for a single one to be used everywhere.

We can use **Assert::dynamic** like this:

```
void f(int n)
    // n should be in [1:max)
{
    Assert::dynamic<Assert::level(2),Assert::Error>(
        (n<=0 || max<n), Assert::compose(__FILE__,__LINE__,"range problem");
    // ...
}
```

The **__FILE__** and **__LINE__** are macros that expand at their point of appearance in the source code (§12.6.2). I can't hide them from the user's view by placing them inside the implementation of **Assert** where they belong.

Assert::Error is the default exception, so we need not mention it explicitly. Similarly, if we are willing to use the default assertion level, we don't need to mention the level explicitly:

```
void f(int n)
    // n should be in [1:max)
{
    Assert::dynamic((n<=0 || max<n),Assert::compose(__FILE__,__LINE__,"range problem");
    // ...
}
```

I do not recommend obsessing about the amount of text needed to express an assertion, but by using a namespace directive (§14.2.3) and the default message, we can get to a minimum:

```
void f(int n)
    // n should be in [1:max)
{
    dynamic(n<=0||max<n);
    // ...
}
```

It is possible to control the testing done and the response to testing through build options (e.g., controlling conditional compilation) and/or through options in the program code. That way, you can have a debug version of a system that tests extensively and enters the debugger and a production version that does hardly any testing.

I personally favor leaving at least some tests in the final (shipping) version of a program. For example, with **Assert** the obvious convention is that assertions marked as level zero will always be checked. We never find the last bug in a large program under continuous development and maintenance. Also, even if all else works perfectly, having a few ''sanity checks'' left to deal with hardware failures can be wise.

Only the builder of the final complete system can decide whether a failure is acceptable or not. The writer of a library or reusable component usually does not have the luxury of terminating unconditionally. I interpret that to mean that for general library code, reporting an error – preferably by throwing an exception – is essential.

As usual, destructors should not throw, so don't use a throwing **Assert()** in a destructor.

13.5 Throwing and Catching Exceptions

This section presents exceptions from a language-technical point of view.

13.5.1 Throwing Exceptions

We can **throw** an exception of any type that can be copied or moved. For example:

```
class No_copy {
    No_copy(const No_copy&) = delete;      // prohibit copying (§17.6.4)
};

class My_error {
    // ...
};
```

```
void f(int n)
{
        switch (n) {
        case 0:    throw My_error{};        // OK
        case 1:    throw No_copy{};         // error: can't copy a No_copy
        case 2:    throw My_error;          // error: My_error is a type, rather than an object
        }
}
```

The exception object caught (§13.5.2) is in principle a copy of the one thrown (though an optimizer is allowed to minimize copying); that is, a **throw x;** initializes a temporary variable of **x**'s type with **x**. This temporary may be further copied several times before it is caught: the exception is passed (back) from called function to calling function until a suitable handler is found. The type of the exception is used to select a handler in the **catch**-clause of some **try**-block. The data in the exception object – if any – is typically used to produce error messages or to help recovery. The process of passing the exception ''up the stack'' from the point of throw to a handler is called *stack unwinding*. In each scope exited, the destructors are invoked so that every fully constructed object is properly destroyed. For example:

```
void f()
{
        string name {"Byron"};
        try {
                string s = "in";
                g();
        }
        catch (My_error) {
                // ...
        }
}

void g()
{
        string s = "excess";
        {
                string s = "or";
                h();
        }
}

void h()
{
        string s = "not";
        throw My_error{};
        string s2 = "at all";
}
```

After the throw in **h()**, all the **string**s that were constructed are destroyed in the reverse order of their construction: **"not"**, **"or"**, **"excess"**, **"in"**, but not **"at all"**, which the thread of control never reached, and not **"Byron"**, which was unaffected.

Because an exception is potentially copied several times before it is caught, we don't usually put huge amounts of data in it. Exceptions containing a few words are very common. The semantics of exception propagation are those of initialization, so objects of types with move semantics (e.g., **string**s) are not expensive to throw. Some of the most common exceptions carry no information; the name of the type is sufficient to report the error. For example:

```
struct Some_error { };

void fct()
{
    // ...
    if (something_wrong)
        throw Some_error{};
}
```

There is a small standard-library hierarchy of exception types (§13.5.2) that can be used either directly or as base classes. For example:

```
struct My_error2 : std::runtime_error {
    const char* what() const noexcept { return "My_error2"; }
};
```

The standard-library exception classes, such as **runtime_error** and **out_of_range**, take a string argument as a constructor argument and have a virtual function **what()** that will regurgitate that string. For example:

```
void g(int n)    // throw some exception
{
    if (n)
        throw std::runtime_error{"I give up!"};
    else
        throw My_error2{};
}

void f(int n)    // see what exception g() throws
{
    try {
        void g(n);
    }
    catch (std::exception& e) {
        cerr << e.what() << '\n';
    }
}
```

13.5.1.1 noexcept Functions

Some functions don't throw exceptions and some really shouldn't. To indicate that, we can declare such a function **noexcept**. For example:

```
double compute(double) noexcept; // may not throw an exception
```

Now no exception will come out of compute().

Declaring a function noexcept can be most valuable for a programmer reasoning about a program and for a compiler optimizing a program. The programmer need not worry about providing try-clauses (for dealing with failures in a noexcept function) and an optimizer need not worry about control paths from exception handling.

However, noexcept is not completely checked by the compiler and linker. What happens if the programmer ''lied'' so that a noexcept function deliberately or accidentally threw an exception that wasn't caught before leaving the noexcept function? Consider:

```
double compute(double x) noexcept;
{
        string s = "Courtney and Anya";
        vector<double> tmp(10);
        // ...
}
```

The vector constructor may fail to acquire memory for its ten doubles and throw a std::bad_alloc. In that case, the program terminates. It terminates unconditionally by invoking std::terminate() (§30.4.1.3). It does not invoke destructors from calling functions. It is implementation-defined whether destructors from scopes between the throw and the noexcept (e.g., for s in compute()) are invoked. The program is just about to terminate, so we should not depend on any object anyway. By adding a noexcept specifier, we indicate that our code was not written to cope with a throw.

13.5.1.2 The noexcept Operator

It is possible to declare a function to be conditionally noexcept. For example:

```
template<typename T>
void my_fct(T& x) noexcept(Is_pod<T>());
```

The noexcept(Is_pod<T>()) means that My_fct may not throw if the predicate Is_pod<T>() is true but may throw if it is false. I may want to write this if my_fct() copies its argument. I know that copying a POD does not throw, whereas other types (e.g., a string or a vector) may.

The predicate in a noexcept() specification must be a constant expression. Plain noexcept means noexcept(true).

The standard library provides many type predicates that can be useful for expressing the conditions under which a function may throw an exception (§35.4).

What if the predicate we want to use isn't easily expressed using type predicates only? For example, what if the critical operation that may or may not throw is a function call f(x)? The noexcept() operator takes an expression as its argument and returns true if the compiler ''knows'' that it cannot throw and false otherwise. For example:

```
template<typename T>
void call_f(vector<T>& v) noexcept(noexcept(f(v[0]))
{
        for (auto x : v)
                f(x);
}
```

The double mention of **noexcept** looks a bit odd, but **noexcept** is not a common operator.

The operand of **noexcept()** is not evaluated, so in the example we do not get a run-time error if we pass **call_f()** with an empty **vector**.

A **noexcept(expr)** operator does not go to heroic lengths to determine whether **expr** can throw; it simply looks at every operation in **expr** and if they *all* have **noexcept** specifications that evaluate to **true**, it returns **true**. A **noexcept(expr)** does not look inside definitions of operations used in **expr**.

Conditional **noexcept** specifications and the **noexcept()** operator are common and important in standard-library operations that apply to containers. For example (§iso.20.2.2):

```
template<class T, size_t N>
void swap(T (&a)[N], T (&b)[N]) noexcept(noexcept(swap(*a, *b)));
```

13.5.1.3 Exception Specifications

In older C++ code, you may find *exception specifications*. For example:

```
void f(int) throw(Bad,Worse);   // may only throw Bad or Worse exceptions
void g(int) throw();            // may not throw
```

An empty exception specification **throw()** is defined to be equivalent to **noexcept** (§13.5.1.1). That is, if an exception is thrown, the program terminates.

The meaning of a nonempty exception specification, such as **throw(Bad,Worse)**, is that if the function (here **f()**) throws any exception that is not mentioned in the list or publicly derived from an exception mentioned there, an *unexpected handler* is called. The default effect of an unexpected exception is to terminate the program (§30.4.1.3). A nonempty **throw** specification is hard to use well and implies potentially expensive run-time checks to determine if the right exception is thrown. This feature has not been a success and is deprecated. Don't use it.

If you want to dynamically check which exceptions are thrown, use a **try**-block.

13.5.2 Catching Exceptions

Consider:

```
void f()
{
    try {
        throw E{};
    }
    catch(H) {
        // when do we get here?
    }
}
```

The handler is invoked:

[1] If **H** is the same type as **E**
[2] If **H** is an unambiguous public base of **E**
[3] If **H** and **E** are pointer types and [1] or [2] holds for the types to which they refer
[4] If **H** is a reference and [1] or [2] holds for the type to which **H** refers

In addition, we can add **const** to the type used to catch an exception in the same way that we can

add it to a function parameter. This doesn't change the set of exceptions we can catch; it only restricts us from modifying the exception caught.

In principle, an exception is copied when it is thrown (§13.5). The implementation may apply a wide variety of strategies for storing and transmitting exceptions. It is guaranteed, however, that there is sufficient memory to allow **new** to throw the standard out-of-memory exception, **bad_alloc** (§11.2.3).

Note the possibility of catching an exception by reference. Exception types are often defined as part of class hierarchies to reflect relationships among the kinds of errors they represent. For examples, see §13.5.2.3 and §30.4.1.1. The technique of organizing exception classes into hierarchies is common enough for some programmers to prefer to catch every exception by reference.

The **{}** in both the **try**-part and a **catch**-clause of a **try**-block are real scopes. Consequently, if a name is to be used in both parts of a **try**-block or outside it, that name must be declared outside the **try**-block. For example:

```
void g()
{
        int x1;

        try {
                int x2 = x1;
                // ...
        }
        catch (Error) {
                ++x1;       // OK
                ++x2;       // error: x2 not in scope
                int x3 = 7;
                // ...
        }
        catch(...) {
                ++x3;       // error: x3 not in scope
                // ...
        }

        ++x1;               // OK
        ++x2;               // error: x2 not in scope
        ++x3;               // error: x3 not in scope
}
```

The ''catch everything'' clause, **catch(...)**, is explained in §13.5.2.2.

13.5.2.1 Rethrow

Having caught an exception, it is common for a handler to decide that it can't completely handle the error. In that case, the handler typically does what can be done locally and then throws the exception again. Thus, an error can be handled where it is most appropriate. This is the case even when the information needed to best handle the error is not available in a single place, so that the recovery action is best distributed over several handlers. For example:

```
void h()
{
    try {
        // ... code that might throw an exception ...
    }
    catch (std::exception& err) {
        if (can_handle_it_completely) {
            // ... handle it ...
            return;
        }
        else {
            // ... do what can be done here ...
            throw;      // rethrow the exception
        }
    }
}
```

A rethrow is indicated by a **throw** without an operand. A rethrow may occur in a **catch**-clause or in a function called from a **catch**-clause. If a rethrow is attempted when there is no exception to rethrow, **std::terminate()** (§13.5.2.5) will be called. A compiler can detect and warn about some, but not all, such cases.

The exception rethrown is the original exception caught and not just the part of it that was accessible as an **exception**. For example, had an **out_of_range** been thrown, **h()** would catch it as a plain **exception**, but **throw;** would still rethrow it as an **out_of_range**. Had I written **throw err;** instead of the simpler **throw;**, the exception would have been sliced (§17.5.1.4) and **h()**'s caller could not have caught it as an **out_of_range**.

13.5.2.2 Catch Every Exception

In **<stdexcept>**, the standard library provides a small hierarchy of exception classes with a common base **exception** (§30.4.1.1). For example:

```
void m()
{
    try {
        // ... do something ...
    }
    catch (std::exception& err) {       // handle every standard-library exception
        // ... cleanup ...
        throw;
    }
}
```

This catches every standard-library exception. However, the standard-library exceptions are just one set of exception types. Consequently, you cannot catch every exception by catching **std::exception**. If someone (unwisely) threw an **int** or an exception from some application-specific hierarchy, it would not be caught by the handler for **std::exception&**.

However, we often need to deal with every kind of exception. For example, if **m()** is supposed to leave some pointers in the state in which it found them, then we can write code in the handler to

give them acceptable values. As for functions, the ellipsis, **...**, indicates ‘‘any argument’’ (§12.2.4), so **catch(...)** means ‘‘catch any exception.’’ For example:

```
void m()
{
    try {
        // ... something ...
    }
    catch (...) {           // handle every exception
        // ... cleanup ...
        throw;
    }
}
```

13.5.2.3 Multiple Handlers

A **try**-block may have multiple **catch**-clauses (handlers). Because a derived exception can be caught by handlers for more than one exception type, the order in which the handlers are written in a **try**-statement is significant. The handlers are tried in order. For example:

```
void f()
{
    try {
        // ...
    }
    catch (std::ios_base::failure) {
        // ... handle any iostream error (§30.4.1.1) ...
    }
    catch (std::exception& e) {
        // ... handle any standard-library exception (§30.4.1.1) ...
    }
    catch (...) {
        // ... handle any other exception (§13.5.2.2) ...
    }
}
```

The compiler knows the class hierarchy, so it can warn about many logical mistakes. For example:

```
void g()
{
    try {
        // ...
    }
    catch (...) {
        // ... handle every exception (§13.5.2.2) ...
    }
    catch (std::exception& e) {
        // ...handle any standard library exception (§30.4.1.1) ...
    }
```

```
        catch (std::bad_cast) {
            // ... handle dynamic_cast failure (§22.2.1) ...
        }
    }
```

Here, the **exception** is never considered. Even if we removed the ''catch-all'' handler, **bad_cast** wouldn't be considered because it is derived from **exception**. Matching exception types to **catch**-clauses is a (fast) run-time operation and is not as general as (compile-time) overload resolution.

13.5.2.4 Function try-Blocks

The body of a function can be a **try**-block. For example:

```
int main()
try
{
    // ... do something ...
}
catch (...) {
    // ... handle exception ...
}
```

For most functions, all we gain from using a function **try**-block is a bit of notational convenience. However, a **try**-block allows us to deal with exceptions thrown by base-or-member initializers in constructors (§17.4). By default, if an exception is thrown in a base-or-member initializer, the exception is passed on to whatever invoked the constructor for the member's class. However, the constructor itself can catch such exceptions by enclosing the complete function body – including the member initializer list – in a **try**-block. For example:

```
class X {
    vector<int> vi;
    vector<string> vs;

    // ...
public:
    X(int,int);
    // ...
};

X::X(int sz1, int sz2)
try
    :vi(sz1),   // construct vi with sz1 ints
     vs(sz2),   // construct vs with sz2 strings
{
    // ...
}
catch (std::exception& err) { // exceptions thrown for vi and vs are caught here
    // ...
}
```

So, we can catch exceptions thrown by member constructors. Similarly, we can catch exceptions thrown by member destructors in a destructor (though a destructor should never throw). However, we cannot ''repair'' the object and return normally as if the exception had not happened: an exception from a member constructor means that the member may not be in a valid state. Also, other member objects will either not be constructed or already have had their destructors invoked as part of the stack unwinding.

The best we can do in a **catch**-clause of a function **try**-block for a constructor or destructor is to throw an exception. The default action is to rethrow the original exception when we ''fall off the end'' of the **catch**-clause (§iso.15.3).

There are no such restrictions for the **try**-block of an ordinary function.

13.5.2.5 Termination

There are cases where exception handling must be abandoned for less subtle error-handling techniques. The guiding principles are:

- Don't throw an exception while handling an exception.
- Don't throw an exception that can't be caught.

If the exception-handling implementation catches you doing either, it will terminate your program.

If you managed to have two exceptions active at one time (in the same thread, which you can't), the system would have no idea which of the exceptions to try to handle: your new one or the one it was already trying to handle. Note that an exception is considered handled immediately upon entry into a **catch**-clause. Rethrowing an exception (§13.5.2.1) or throwing a new exception from within a **catch**-clause is considered a new throw done after the original exception has been handled. You can throw an exception from within a destructor (even during stack unwinding) as long as you catch it before it leaves the destructor.

The specific rules for calling **terminate()** are (§iso.15.5.1)

- When no suitable handler was found for a thrown exception
- When a **noexcept** function tries to exit with a **throw**
- When a destructor invoked during stack unwinding tries to exit with a **throw**
- When code invoked to propagate an exception (e.g., a copy constructor) tries to exit with a **throw**
- When someone tries to rethrow (**throw;**) when there is no current exception being handled
- When a destructor for a statically allocated or thread-local object tries to exit with a **throw**
- When an initializer for a statically allocated or thread-local object tries to exit with a **throw**
- When a function invoked as an **atexit()** function tries to exit with a **throw**

In such cases, the function **std::terminate()** is called. In addition, a user can call **terminate()** if less drastic approaches are infeasible.

By ''tries to exit with a **throw**,'' I mean that an exception is thrown somewhere and not caught so that the run-time system tries to propagate it from a function to its caller.

By default, **terminate()** will call **abort()** (§15.4.3). This default is the correct choice for most users – especially during debugging. If that is not acceptable, the user can provide a *terminate handler* function by a call **std::set_terminate()** from **<exception>**:

```
using terminate_handler = void(*)();      // from <exception>

[[noreturn]] void my_handler()            // a terminate handler cannot return
{
    // handle termination my way
}

void dangerous()    // very!
{
    terminate_handler old = set_terminate(my_handler);
    // ...
    set_terminate(old);  // restore the old terminate handler
}
```

The return value is the previous function given to **set_terminate()**.

For example, a terminate handler could be used to abort a process or maybe to re-initialize a system. The intent is for **terminate()** to be a drastic measure to be applied when the error recovery strategy implemented by the exception-handling mechanism has failed and it is time to go to another level of a fault tolerance strategy. If a terminate handler is entered, essentially nothing can be assumed about a program's data structures; they must be assumed to be corrupted. Even writing an error message using **cerr** must be assumed to be hazardous. Also, note that as **dangerous()** is written, it is not exception-safe. A **throw** or even a **return** before **set_terminate(old)** will leave **my_handler** in place when it wasn't meant to be. If you must mess with **terminate()**, at least use RAII (§13.3).

A terminate handler cannot return to its caller. If it tries to, **terminate()** will call **abort()**.

Note that **abort()** indicates abnormal exit from the program. The function **exit()** can be used to exit a program with a return value that indicates to the surrounding system whether the exit is normal or abnormal (§15.4.3).

It is implementation-defined whether destructors are invoked when a program is terminated because of an uncaught exception. On some systems, it is essential that the destructors are not called so that the program can be resumed from the debugger. On other systems, it is architecturally close to impossible *not* to invoke the destructors while searching for a handler.

If you want to ensure cleanup when an otherwise uncaught exception happens, you can add a catch-all handler (§13.5.2.2) to **main()** in addition to handlers for exceptions you really care about. For example:

```
int main()
try {
    // ...
}
catch (const My_error& err) {
    // ... handle my error ...
}
catch (const std::range_error&)
{
    cerr << "range error: Not again!\n";
}
```

```
catch (const std::bad_alloc&)
{
        cerr << "new ran out of memory\n";
}
catch (...) {
        // ...
}
```

This will catch every exception, except those thrown by construction and destruction of namespace and thread-local variables (§13.5.3). There is no way of catching exceptions thrown during initialization or destruction of namespace and thread-local variables. This is another reason to avoid global variables whenever possible.

When an exception is caught, the exact point where it was thrown is generally not known. This represents a loss of information compared to what a debugger might know about the state of a program. In some C++ development environments, for some programs, and for some people, it might therefore be preferable *not* to catch exceptions from which the program isn't designed to recover.

See **Assert** (§13.4) for an example of how one might encode the location of a **throw** into the thrown exception.

13.5.3 Exceptions and Threads

If an exception is not caught on a **thread** (§5.3.1, §42.2), **std::terminate()** (§13.5.2.5) is called. So, if we don't want an error in a thread to stop the whole program, we must catch all errors from which we would like to recover and somehow report them to a part of the program that is interested in the results of the thread. The "catch-all" construct **catch(...)** (§13.5.2.2) comes in handy for that.

We can transfer an exception thrown on one thread to a handler on another thread using the standard-library function **current_exception()** (§30.4.1.2). For example:

```
try {
        // ... do the work ...
}
catch(...) {
        prom.set_exception(current_exception());
}
```

This is the basic technique used by **packaged_task** to handle exceptions from user code (§5.3.5.2).

13.6 A vector Implementation

The standard **vector** provides splendid examples of techniques for writing exception-safe code: its implementation illustrates problems that occur in many contexts and solutions that apply widely.

Obviously, a **vector** implementation relies on many language facilities provided to support the implementation and use of classes. If you are not (yet) comfortable with C++'s classes and templates, you may prefer to delay studying this example until you have read Chapter 16, Chapter 25, and Chapter 26. However, a good understanding of the use of exceptions in C++ requires a more extensive example than the code fragments so far in this chapter.

The basic tools available for writing exception-safe code are:

- The try-block (§13.5).
- The support for the "Resource Acquisition Is Initialization" technique (§13.3).

The general principles to follow are to

- Never let go of a piece of information before its replacement is ready for use.
- Always leave objects in valid states when throwing or rethrowing an exception.

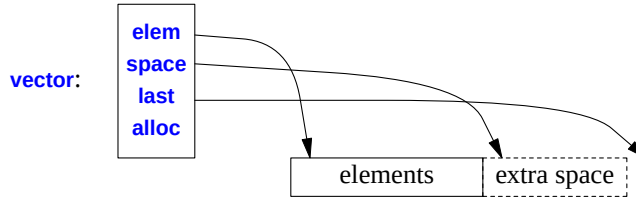
That way, we can always back out of an error situation. The practical difficulty in following these principles is that innocent-looking operations (such as <, =, and sort()) might throw exceptions. Knowing what to look for in an application takes experience.

When you are writing a library, the ideal is to aim at the strong exception-safety guarantee (§13.2) and always to provide the basic guarantee. When writing a specific program, there may be less concern for exception safety. For example, if I write a simple data analysis program for my own use, I'm usually quite willing to have the program terminate in the unlikely event of memory exhaustion.

Correctness and basic exception safety are closely related. In particular, the techniques for providing basic exception safety, such as defining and checking invariants (§13.4), are similar to the techniques that are useful to get a program small and correct. It follows that the overhead of providing the basic exception-safety guarantee (§13.2) – or even the strong guarantee – can be minimal or even insignificant.

13.6.1 A Simple vector

A typical implementation of vector (§4.4.1, §31.4) will consist of a handle holding pointers to the first element, one-past-the-last element, and one-past-the-last allocated space (§31.2.1) (or the equivalent information represented as a pointer plus offsets):



In addition, it holds an allocator (here, alloc), from which the vector can acquire memory for its elements. The default allocator (§34.4.1) uses new and delete to acquire and release memory.

Here is a declaration of vector simplified to present only what is needed to discuss exception safety and avoidance of resource leaks:

```
template<class T, class A = allocator<T>>
class vector {
private:
    T* elem;      // start of allocation
    T* space;     // end of element sequence, start of space allocated for possible expansion
    T* last;      // end of allocated space
    A alloc;      // allocator
```

```
public:
    using size_type = unsigned int;              // type used for vector sizes

    explicit vector(size_type n, const T& val = T(), const A& = A());

    vector(const vector& a);                     // copy constructor
    vector& operator=(const vector& a);          // copy assignment

    vector(vector&& a);                          // move constructor
    vector& operator=(vector&& a);               // move assignment

    ~vector();

    size_type size() const { return space–elem; }
    size_type capacity() const { return last–elem; }
    void reserve(size_type n);                   // increase capacity to n

    void resize(size_type n, const T& = {});     // increase size to n
    void push_back(const T&);                    // add an element at the end

    // ...
};
```

Consider first a naive implementation of the constructor that initializes a **vector** to **n** elements initialized to **val**:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)  // warning: naive implementation
    :alloc{a}                                // copy the allocator
{
    elem = alloc.allocate(n);                // get memory for elements (§34.4)
    space = last = elem+n;
    for (T* p = elem; p!=last; ++p)
        a.construct(p,val);                  // construct copy of val in *p (§34.4)
}
```

There are two potential sources of exceptions here:

[1] **allocate()** may throw an exception if no memory is available.

[2] **T**'s copy constructor may throw an exception if it can't copy **val**.

What about the copy of the allocator? We can imagine that it throws, but the standard specifically requires that it does not do that (§iso.17.6.3.5). Anyway, I have written the code so that it wouldn't matter if it did.

In both cases of a **throw**, no **vector** object is created, so **vector**'s destructor is not called (§13.3).

When **allocate()** fails, the **throw** will exit before any resources are acquired, so all is well.

When **T**'s copy constructor fails, we have acquired some memory that must be freed to avoid memory leaks. Worse still, the copy constructor for **T** might throw an exception after correctly constructing a few elements but before constructing them all. These **T** objects may own resources that then would be leaked.

To handle this problem, we could keep track of which elements have been constructed and destroy those (and only those) in case of an error:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)        // elaborate implementation
    :alloc{a}                                 // copy the allocator
{
    elem = alloc.allocate(n);                 // get memory for elements

    iterator p;

    try {
        iterator end = elem+n;
        for (p=elem; p!=end; ++p)
            alloc.construct(p,val);           // construct element (§34.4)
        last = space = p;
    }
    catch (...) {
        for (iterator q = elem; q!=p; ++q)
            alloc.destroy(q);                 // destroy constructed elements
        alloc.deallocate(elem,n);             // free memory
        throw;                                // rethrow
    }
}
```

Note that the declaration of `p` is outside the `try`-block; otherwise, we would not be able to access it in both the `try`-part and the `catch`-clause.

The overhead here is the overhead of the `try`-block. In a good C++ implementation, this overhead is negligible compared to the cost of allocating memory and initializing elements. For implementations where entering a `try`-block incurs a cost, it may be worthwhile to add a test `if (n)` before the `try` to explicitly handle the (very common) empty `vector` case.

The main part of this constructor is a repeat of the implementation of `std::uninitialized_fill()`:

```
template<class For, class T>
void uninitialized_fill(For beg, For end, const T& x)
{
    For p;
    try {
        for (p=beg; p!=end; ++p)
            ::new(static_cast<void*>(&*p)) T(x);      // construct copy of x in *p (§11.2.4)
    }
    catch (...) {
        for (For q = beg; q!=p; ++q)
            (&*q)->~T();                              // destroy element (§11.2.4)
        throw;                                        // rethrow (§13.5.2.1)
    }
}
```

The curious construct `&*p` takes care of iterators that are not pointers. In that case, we need to take the address of the element obtained by dereference to get a pointer. Together with the explicitly

global ::new, the explicit cast to void* ensures that the standard-library placement function (§17.2.4) is used to invoke the constructor, and not some user-defined operator new() for T*s. The calls to alloc.construct() in the vector constructors are simply syntactic sugar for this placement new. Similarly, the alloc.destroy() call simply hides explicit destruction (like (&*q)->~T()). This code is operating at a rather low level where writing truly general code can be difficult.

Fortunately, we don't have to invent or implement uninitialized_fill(), because the standard library provides it (§32.5.6). It is often essential to have initialization operations that either complete successfully, having initialized every element, or fail, leaving no constructed elements behind. Consequently, the standard library provides uninitialized_fill(), uninitialized_fill_n(), and uninitialized_copy() (§32.5.6), which offer the strong guarantee (§13.2).

The uninitialized_fill() algorithm does not protect against exceptions thrown by element destructors or iterator operations (§32.5.6). Doing so would be prohibitively expensive and probably impossible.

The uninitialized_fill() algorithm can be applied to many kinds of sequences. Consequently, it takes a forward iterator (§33.1.2) and cannot guarantee to destroy elements in the reverse order of their construction.

Using uninitialized_fill(), we can simplify our constructor:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)        // still a bit messy
    :alloc(a)                                   // copy the allocator
{
    elem = alloc.allocate(n);                   // get memory for elements
    try {
        uninitialized_fill(elem,elem+n,val);    // copy elements
        space = last = elem+n;
    }
    catch (...) {
        alloc.deallocate(elem,n);               // free memory
        throw;                                  // rethrow
    }
}
```

This is a significant improvement on the first version of this constructor, but the next section demonstrates how to further simplify it.

The constructor rethrows a caught exception. The intent is to make vector transparent to exceptions so that the user can determine the exact cause of a problem. All standard-library containers have this property. Exception transparency is often the best policy for templates and other "thin" layers of software. This is in contrast to major parts of a system ("modules") that generally need to take responsibility for all exceptions thrown. That is, the implementer of such a module must be able to list every exception that the module can throw. Achieving this may involve grouping exceptions into hierarchies (§13.5.2) and using catch(...) (§13.5.2.2).

13.6.2 Representing Memory Explicitly

Experience shows that writing correct exception-safe code using explicit try-blocks is more difficult than most people expect. In fact, it is unnecessarily difficult because there is an alternative: The

"Resource Acquisition Is Initialization" technique (§13.3) can be used to reduce the amount of code that must be written and to make the code more stylized. In this case, the key resource required by the **vector** is memory to hold its elements. By providing an auxiliary class to represent the notion of memory used by a **vector**, we can simplify the code and decrease the chances of accidentally forgetting to release it:

```
template<class T, class A = allocator<T> >
struct vector_base {                        // memory structure for vector
    A alloc;        // allocator
    T* elem;        // start of allocation
    T* space;       // end of element sequence, start of space allocated for possible expansion
    T* last;        // end of allocated space

    vector_base(const A& a, typename A::size_type n)
        : alloc{a}, elem{alloc.allocate(n)}, space{elem+n}, last{elem+n} { }
    ~vector_base() { alloc.deallocate(elem,last–elem); }

    vector_base(const vector_base&) = delete;           // no copy operations
    vector_base& operator=(const vector_base&) = delete;

    vector_base(vector_base&&);                         // move operations
    vector_base& operator=(vector_base&&);
};
```

As long as **elem** and **last** are correct, **vector_base** can be destroyed. Class **vector_base** deals with memory for a type **T**, not objects of type **T**. Consequently, a user of **vector_base** must construct all objects explicitly in the allocated space and later destroy all constructed objects in a **vector_base** before the **vector_base** itself is destroyed.

The **vector_base** is designed exclusively to be part of the implementation of **vector**. It is always hard to predict where and how a class will be used, so I made sure that a **vector_base** can't be copied and also that a move of a **vector_base** properly transfers ownership of the memory allocated for elements:

```
template<class T, class A>
vector_base<T,A>::vector_base(vector_base&& a)
    : alloc{a.alloc},
    elem{a.elem},
    space{a.space},
    last{a.space}
{
    a.elem = a.space = a.last = nullptr;  // no longer owns any memory
}

template<class T, class A>
vector_base<T,A>::& vector_base<T,A>::operator=(vector_base&& a)
{
    swap(*this,a);
    return *this;
}
```

This definition of the move assignment uses **swap()** to transfer ownership of any memory allocated for elements. There are no objects of type **T** to destroy: **vector_base** deals with memory and leaves concerns about objects of type **T** to **vector**.

Given **vector_base**, **vector** can be defined like this:

```
template<class T, class A = allocator<T> >
class vector {
        vector_base<T,A> vb;                    // the data is here
        void destroy_elements();
public:
        using size_type = unsigned int;

        explicit vector(size_type n, const T& val = T(), const A& = A());

        vector(const vector& a);                // copy constructor
        vector& operator=(const vector& a);     // copy assignment

        vector(vector&& a);                     // move constructor
        vector& operator=(vector&& a);          // move assignment

        ~vector() { destroy_elements(); }

        size_type size() const { return vb.space–vb.elem; }
        size_type capacity() const { return vb.last–vb.elem; }

        void reserve(size_type);                // increase capacity

        void resize(size_type, T = {});         // change the number of elements
        void clear() { resize(0); }             // make the vector empty
        void push_back(const T&);               // add an element at the end

        // ...
};

template<class T, class A>
void vector<T,A>::destroy_elements()
{
        for (T* p = vb.elem; p!=vb.space; ++p)
                p->~T();                        // destroy element (§17.2.4)
        vb.space=vb.elem;
}
```

The **vector** destructor explicitly invokes the **T** destructor for every element. This implies that if an element destructor throws an exception, the **vector** destruction fails. This can be a disaster if it happens during stack unwinding caused by an exception and **terminate()** is called (§13.5.2.5). In the case of normal destruction, throwing an exception from a destructor typically leads to resource leaks and unpredictable behavior of code relying on reasonable behavior of objects. There is no really good way to protect against exceptions thrown from destructors, so the library makes no guarantees if an element destructor throws (§13.2).

Now the constructor can be simply defined:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
    :vb{a,n}        // allocate space for n elements
{
    uninitialized_fill(vb.elem,vb.elem+n,val);// make n copies of val
}
```

The simplification achieved for this constructor carries over to every **vector** operation that deals with initialization or allocation. For example, the copy constructor differs mostly by using **uninitialized_copy()** instead of **uninitialized_fill()**:

```
template<class T, class A>
vector<T,A>::vector(const vector<T,A>& a)
    :vb{a.alloc,a.size()}
{
    uninitialized_copy(a.begin(),a.end(),vb.elem);
}
```

This style of constructor relies on the fundamental language rule that when an exception is thrown from a constructor, subobjects (including bases) that have already been completely constructed will be properly destroyed (§13.3). The **uninitialized_fill()** algorithm and its cousins (§13.6.1) provide the equivalent guarantee for partially constructed sequences.

The move operations are even simpler:

```
template<class T, class A>
vector<T,A>::vector(vector&& a)                 // move constructor
    :vb{move(a.vb)}         // transfer ownership
{
}
```

The **vector_base** move constructor will set the argument's representation to "empty."

For the move assignment, we must take care of the old value of the target:

```
template<class T, class A>
vector<T,A>::& vector<T,A>::operator=(vector&& a)       // move assignment
{
    clear();            // destroy elements
    swap(*this,a);      // transfer ownership
}
```

The **clear()** is strictly speaking redundant because I could assume that the rvalue **a** would be destroyed immediately after the assignment. However, I don't know if some programmer has been playing games with **std::move()**.

13.6.3 Assignment

As usual, assignment differs from construction in that an old value must be taken care of. First consider a straightforward implementation:

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)        // offers the strong guarantee (§13.2)
{
    vector_base<T,A> b(alloc,a.size());                     // get memory
    uninitialized_copy(a.begin(),a.end(),b.elem);           // copy elements
    destroy_elements();                                     // destroy old elements
    swap(vb,b);                                             // transfer ownership
    return *this;                                           // implicitly destroy the old value
}
```

This **vector** assignment provides the strong guarantee, but it repeats a lot of code from constructors and destructors. We can avoid repetition:

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)        // offers the strong guarantee (§13.2)
{
    vector temp {a};                // copy allocator
    std::swap(*this,temp);          // swap representations
    return *this;
}
```

The old elements are destroyed by **temp**'s destructor, and the memory used to hold them is deallocated by **temp**'s **vector_base**'s destructor.

The reason that the standard-library **swap()** (§35.5.2) works for **vector_base**s is that we defined **vector_base** move operations for **swap()** to use.

The performance of the two versions ought to be equivalent. Essentially, they are just two different ways of specifying the same set of operations. However, the second implementation is shorter and doesn't replicate code from related **vector** functions, so writing the assignment that way ought to be less error-prone and lead to simpler maintenance.

Note that I did not test for self-assignment, such as **v=v**. This implementation of **=** works by first constructing a copy and then swapping representations. This obviously handles self-assignment correctly. I decided that the efficiency gained from the test in the rare case of self-assignment was more than offset by its cost in the common case where a different **vector** is assigned.

In either case, two potentially significant optimizations are missing:

[1] If the capacity of the **vector** assigned to is large enough to hold the assigned **vector**, we don't need to allocate new memory.

[2] An element assignment may be more efficient than an element destruction followed by an element construction.

Implementing these optimizations, we get:

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)        // optimized, basic guarantee (§13.2) only
{
    if (capacity() < a.size()) {  // allocate new vector representation:
        vector temp {a};          // copy allocator
        swap(*this,temp);         // swap representations
        return *this;             // implicitly destroy the old value
    }
```

```
        if (this == &a) return *this;                           // optimize self assignment

        size_type sz = size();
        size_type asz = a.size();
        vb.alloc = a.vb.alloc;                                  // copy the allocator
        if (asz<=sz) {
            copy(a.begin(),a.begin()+asz,vb.elem);
            for (T* p = vb.elem+asz; p!=vb.space; ++p)          // destroy surplus elements (§16.2.6)
                p->~T();
        }
        else {
            copy(a.begin(),a.begin()+sz,vb.elem);
            uninitialized_copy(a.begin()+sz,a.end(),vb.space); // construct extra elements
        }
        vb.space = vb.elem+asz;
        return *this;
    }
```

These optimizations are not free. Obviously, the complexity of the code is far higher. Here, I also test for self-assignment. However, I do so mostly to show how it is done because here it is only an optimization.

The `copy()` algorithm (§32.5.1) does *not* offer the strong exception-safety guarantee. Thus, if `T::operator=()` throws an exception during `copy()`, the `vector` being assigned to need not be a copy of the `vector` being assigned, and it need not be unchanged. For example, the first five elements might be copies of elements of the assigned `vector` and the rest unchanged. It is also plausible that an element – the element that was being copied when `T::operator=()` threw an exception – ends up with a value that is neither the old value nor a copy of the corresponding element in the `vector` being assigned. However, if `T::operator=()` leaves its operands in valid states before it throws (as it should), the `vector` is still in a valid state – even if it wasn't the state we would have preferred.

The standard-library `vector` assignment offers the (weaker) basic exception-safety guarantee of this last implementation – and its potential performance advantages. If you need an assignment that leaves the `vector` unchanged if an exception is thrown, you must either use a library implementation that provides the strong guarantee or provide your own assignment operation. For example:

```
template<class T, class A>
void safe_assign(vector<T,A>& a, const vector<T,A>& b)      // simple a = b
{
    vector<T,A> temp{b};            // copy the elements of b into a temporary
    swap(a,temp);
}
```

Alternatively, we could simply use call-by-value (§12.2):

```
template<class T, class A>
void safe_assign(vector<T,A>& a, vector<T,A> b)         // simple a = b (note: b is passed by value)
{
    swap(a,b);
}
```

I never can decide if this last version is simply beautiful or too clever for real (maintainable) code.

13.6.4 Changing Size

One of the most useful aspects of **vector** is that we can change its size to suit our needs. The most popular functions for changing size are **v.push_back(x)**, which adds an **x** at the end of **v**, and **v.resize(s)**, which makes **s** the number of elements in **v**.

13.6.4.1 reserve()

The key to a simple implementation of such functions is **reserve()**, which adds free space at the end for the **vector** to grow into. In other words, **reserve()** increases the **capacity()** of a **vector**. If the new allocation is larger than the old, **reserve()** needs to allocate new memory and move the elements into it. We could try the trick from the unoptimized assignment (§13.6.3):

```
template<class T, class A>
void vector<T,A>::reserve(size_type newalloc)      // flawed first attempt
{
      if (newalloc<=capacity()) return;            // never decrease allocation
      vector<T,A> v(capacity());                   // make a vector with the new capacity
      copy(elem,elem+size(),v.begin())             // copy elements
      swap(*this,v);                               // install new value
} // implicitly release old value
```

This has the nice property of providing the strong guarantee. However, not all types have a default value, so this implementation is flawed. Furthermore, looping over the elements twice, first to default construct and then to copy, is a bit odd. So let us optimize:

```
template<class T, class A>
void vector<T,A>::reserve(size_type newalloc)
{
      if (newalloc<=capacity()) return;                 // never decrease allocation
      vector_base<T,A> b {vb.alloc,newalloc};           // get new space
      uninitialized_move(elem,elem+size(),b.elem);      // move elements
      swap(vb,b);                                       // install new base
} // implicitly release old space
```

The problem is that the standard library doesn't offer **uninitialized_move()**, so we have to write it:

```
template<typename In, typename Out>
Out uninitialized_move(In b, In e, Out oo)
{
      for (; b!=e; ++b,++oo) {
            new(static_cast<void*>(&*oo)) T{move(*b)};  // move construct
            b->~T();                                    // destroy
      }
      return b;
}
```

In general, there is no way of recovering the original state from a failed move, so I don't try to. This **uninitialized_move()** offers only the basic guarantee. However, it is simple and for the vast majority of cases it is fast. Also, the standard-library **reserve()** only offers the basic guarantee.

Whenever **reserve()** may have moved the elements, any iterators into the **vector** may have been invalidated (§31.3.3).

Remember that a move operation should not throw. In the rare cases where the obvious implementation of a move might throw, we typically go out of our way to avoid that. A throw from a move operation is rare, unexpected, and damaging to normal reasoning about code. If at all possible avoid it. The standard-library **move_if_noexcept()** operations may be of help here (§35.5.1).

The explicit use of **move()** is needed because the compiler doesn't know that **elem[i]** is just about to be destroyed.

13.6.4.2 resize()

The **vector** member function **resize()** changes the number of elements. Given **reserve()**, the implementation **resize()** is fairly simple. If the number of elements increases, we must construct the new elements. Conversely, if the number of elements decrease, we must destroy the surplus elements:

```
template<class T, class A>
void vector<T,A>::resize(size_type newsize, const T& val)
{
    reserve(newsize);
    if (size()<newsize)
        uninitialized_fill(elem+size(),elem+newsize,val);    // construct new elements: [size():newsize)
    else
        destroy(elem.size(),elem+newsize);                   // destroy surplus elements: [newsize:size())
    vb.space = vb.last = vb.elem+newsize;
}
```

There is no standard **destroy()**, but that easily written:

```
template<typename In>
void destroy(In b, In e)
{
    for (; b!=e; ++b)       // destroy [b:e)
        b->~T();
}
```

13.6.4.3 push_back()

From an exception-safety point of view, **push_back()** is similar to assignment in that we must take care that the **vector** remains unchanged if we fail to add a new element:

```
template< class T, class A>
void vector<T,A>::push_back(const T& x)
{
    if (capacity()==size())                  // no more free space; relocate:
        reserve(sz?2*sz:8);                  // grow or start with 8
    vb.alloc.construct(&vb.elem[size()],val);    // add val at end
    ++vb.space;                              // increment size
}
```

Naturally, the copy constructor used to initialize ***space** might throw an exception. If that happens,