# 17

# Construction, Cleanup, Copy, and Move

*Ignorance more frequently begets confidence*
*than does knowledge.*
*– Charles Darwin*



## 17.1 Introduction

This chapter focuses on technical aspects of an object's ''life cycle'': How do we create an object, how do we copy it, how do we move it around, and how do we clean up after it when it goes away? What are proper definitions of ''copy'' and ''move''? For example:

```
string ident(string arg)        // string passed by value (copied into arg)
{
    return arg;                 // return string (move the value of arg out of ident() to a caller)
}

int main ()
{
    string s1 {"Adams"};        // initialize string (construct in s1).
    s1 = indet(s1);             // copy s1 into ident()
                                // move the result of ident(s1) into s1;
                                // s1's value is "Adams".
    string s2 {"Pratchett"};    // initialize string (construct in s2)
    s1 = s2;                    // copy the value of s2 into s1
                                // both s1 and s2 have the value "Pratchett".
}
```

Clearly, after the call of **ident()**, the value of **s1** ought to be **"Adams"**. We copy the value of **s1** into the argument **arg**, then we move the value of **arg** out of the function call and (back) into **s1**. Next, we construct **s2** with the value **"Prachett"** and copy it into **s1**. Finally, at the exit from **main()** we destroy the variables **s1** and **s2.** The difference between *move* and *copy* is that after a copy two objects must have the same value, whereas after a move the source of the move is not required to have its original value. Moves can be used when the source object will not be used again. They are particularly useful for implementing the notion of moving a resource (§3.2.1.2, §5.2).

Several functions are used here:

- A constructor initializing a **string** with a string literal (used for **s1** and **s2**)
- A copy constructor copying a **string** (into the function argument **arg**)
- A move constructor moving the value of a **string** (from **arg** out of **ident()** into a temporary variable holding the result of **ident(s1)**)
- A move assignment moving the value of a **string** (from the temporary variable holding the result of **ident(s1)** into **s1**)
- A copy assignment copying a **string** (from **s2** into **s1**)
- A destructor releasing the resources owned by **s1**, **s2**, and the temporary variable holding the result of **ident(s1)**

An optimizer can eliminate some of this work. For example, in this simple example the temporary variable is typically eliminated. However, in principle, these operations are executed.

Constructors, copy and move assignment operations, and destructors directly support a view of lifetime and resource management. An object is considered an object of its type after its constructor completes, and it remains an object of its type until its destructor starts executing. The interaction between object lifetime and errors is explored further in §13.2 and §13.3. In particular, this chapter doesn't discuss the issue of half-constructed and half-destroyed objects.

Construction of objects plays a key role in many designs. This wide variety of uses is reflected in the range and flexibility of the language features supporting initialization.

Constructors, destructors, and copy and move operations for a type are not logically separate. We must define them as a matched set or suffer logical or performance problems. If a class **X** has a destructor that performs a nontrivial task, such as free-store deallocation or lock release, the class is likely to need the full complement of functions:

```
class X {
        X(Sometype);            // "ordinary constructor": create an object
        X();                    // default constructor
        X(const X&);            // copy constructor
        X(X&&);                 // move constructor
        X& operator=(const X&); // copy assignment: clean up target and copy
        X& operator=(X&&);      // move assignment: clean up target and move
        ~X();                   // destructor: clean up
        // ...
};
```

There are five situations in which an object is copied or moved:

- As the source of an assignment
- As an object initializer
- As a function argument
- As a function return value
- As an exception

In all cases, the copy or move constructor will be applied (unless it can be optimized away).

In addition to the initialization of named objects and objects on the free store, constructors are used to initialize temporary objects (§6.4.2) and to implement explicit type conversion (§11.5).

Except for the "ordinary constructor," these special member functions can be generated by the compiler; see §17.6.

This chapter is full of rules and technicalities. Those are necessary for a full understanding, but most people just learn the general rules from examples.

## 17.2 Constructors and Destructors

We can specify how an object of a class is to be initialized by defining a constructor (§16.2.5, §17.3). To complement constructors, we can define a destructor to ensure "cleanup" at the point of destruction of an object (e.g., when it goes out of scope). Some of the most effective techniques for resource management in C++ rely on constructor/destructor pairs. So do other techniques relying on a pair of actions, such as do/undo, start/stop, before/after, etc. For example:

```
struct Tracer {
        string mess;
        Tracer(const string& s) :mess{s} { clog << mess; }
        ~Tracer() {clog << "~" << mess; }
};

void f(const vector<int>& v)
{
        Tracer tr {"in f()\n"};
        for (auto x : v) {
                Tracer tr {string{"v loop "}+to<string>(x)+'\n'}; // §25.2.5.1
                // ...
        }
}
```

We could try a call:

```
f({2,3,5});
```

This would print to the logging stream:

```
in_f()
v loop 2
~v loop 2
v loop 3
~v loop 3
v loop 5
~v loop 5
~in_f()
```

### 17.2.1 Constructors and Invariants

A member with the same name as its class is called a *constructor*. For example:

```
class Vector {
public:
    Vector(int s);
    // ...
};
```

A constructor declaration specifies an argument list (exactly as for a function) but has no return type. The name of a class cannot be used for an ordinary member function, data member, member type, etc., within the class. For example:

```
struct S {
    S();                    // fine
    void S(int);            // error: no type can be specified for a constructor
    int S;                  // error: the class name must denote a constructor
    enum S { foo, bar };    // error: the class name must denote a constructor
};
```

A constructor's job is to initialize an object of its class. Often, that initialization must establish a *class invariant*, that is, something that must hold whenever a member function is called (from outside the class). Consider:

```
class Vector {
public:
    Vector(int s);
    // ...
private:
    double* elem; // elem points to an array of sz doubles
    int sz;       // sz is non-negative
};
```

Here (as is often the case), the invariant is stated as comments: "`elem` points to an array of `sz` doubles" and "`sz` is non-negative." The constructor must make that true. For example:

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

This constructor tries to establish the invariant and if it cannot, it throws an exception. If the constructor cannot establish the invariant, no object is created and the constructor must ensure that no resources are leaked (§5.2, §13.3). A resource is anything we need to acquire and eventually (explicitly or implicitly) give back (release) once we are finished with it. Examples of resources are memory (§3.2.1.2), locks (§5.3.4), file handles (§13.3), and thread handles (§5.3.1).

Why would you define an invariant?

- To focus the design effort for the class (§2.4.3.2)
- To clarify the behavior of the class (e.g., under error conditions; §13.2)
- To simplify the definition of member functions (§2.4.3.2, §16.3.1)
- To clarify the class's management of resources (§13.3)
- To simplify the documentation of the class

On average, the effort to define an invariant ends up saving work.

## 17.2.2 Destructors and Resources

A constructor initializes an object. In other words, it creates the environment in which the member functions operate. Sometimes, creating that environment involves acquiring a resource – such as a file, a lock, or some memory – that must be released after use (§5.2, §13.3). Thus, some classes need a function that is guaranteed to be invoked when an object is destroyed in a manner similar to the way a constructor is guaranteed to be invoked when an object is created. Inevitably, such a function is called a *destructor*. The name of a destructor is ~ followed by the class name, for example `~Vector()`. One meaning of ~ is "complement" (§11.1.2), and a destructor for a class complements its constructors. A destructor does not take an argument, and a class can have only one destructor. Destructors are called implicitly when an automatic variable goes out of scope, an object on the free store is deleted, etc. Only in very rare circumstances does the user need to call a destructor explicitly (§17.2.4).

Destructors typically clean up and release resources. For example:

```
class Vector {
public:
    Vector(int s) :elem{new double[s]}, sz{s} { };          // constructor: acquire memory
    ~Vector() { delete[] elem; }                            // destructor: release memory
    // ...
private:
    double* elem;  // elem points to an array of sz doubles
    int sz;        // sz is non-negative
};
```

For example:

```
Vector* f(int s)
{
        Vector v1(s);
        // ...
        return new Vector(s+s);
}

void g(int ss)
{
        Vector* p = f(ss);
        // ...
        delete p;
}
```

Here, the **Vector v1** is destroyed upon exit from **f()**. Also, the **Vector** created on the free store by **f()** using **new** is destroyed by the call of **delete**. In both cases, **Vector**'s destructor is invoked to free (deallocate) the memory allocated by the constructor.

What if the constructor failed to acquire enough memory? For example, **s*sizeof(double)** or **(s+s)*sizeof(double)** may be larger than the amount of available memory (measured in bytes). In that case, an exception **std::bad_alloc** (§11.2.3) is thrown by **new** and the exception-handling mechanism invokes the appropriate destructors so that all memory that has been acquired (and only that) is freed (§13.5.1).

This style of constructor/destructor-based resource management is called *Resource Acquisition Is Initialization* or simply *RAII* (§5.2, §13.3).

A matching constructor/destructor pair is the usual mechanism for implementing the notion of a variably sized object in C++. Standard-library containers, such as **vector** and **unordered_map**, use variants of this technique for providing storage for their elements.

A type that has no destructor declared, such as a built-in type, is considered to have a destructor that does nothing.

A programmer who declares a destructor for a class must also decide if objects of that class can be copied or moved (§17.6).

### 17.2.3 Base and Member Destructors

Constructors and destructors interact correctly with class hierarchies (§3.2.4, Chapter 20). A constructor builds a class object "from the bottom up":

[1] first, the constructor invokes its base class constructors,
[2] then, it invokes the member constructors, and
[3] finally, it executes its own body.

A destructor "tears down" an object in the reverse order:

[1] first, the destructor executes its own body,
[2] then, it invokes its member destructors, and
[3] finally, it invokes its base class destructors.

In particular, a **virtual** base is constructed before any base that might use it and destroyed after all such bases (§21.3.5.1). This ordering ensures that a base or a member is not used before it has been initialized or used after it has been destroyed. The programmer can defeat this simple and

essential rule, but only through deliberate circumvention involving passing pointers to uninitialized variables as arguments. Doing so violates language rules and the results are usually disastrous.

Constructors execute member and base constructors in declaration order (not the order of initializers): if two constructors used a different order, the destructor could not (without serious overhead) guarantee to destroy in the reverse order of construction. See also §17.4.

If a class is used so that a default constructor is needed, and if the class does not have other constructors, the compiler will try to generate a default constructor. For example:

```
struct S1 {
        string s;
};

S1 x;       // OK: x.s is initialized to ""
```

Similarly, memberwise initialization can be used if initializers are needed. For example:

```
struct X { X(int); };

struct S2 {
        X x;
};

S2 x1;      // error:
S2 x2 {1}; // OK: x2.x is initialized with 1
```

See also §17.3.1.

## 17.2.4 Calling Constructors and Destructors

A destructor is invoked implicitly upon exit from a scope or by **delete**. It is typically not only unnecessary to explicitly call a destructor; doing so would lead to nasty errors. However, there are rare (but important) cases where a destructor must be called explicitly. Consider a container that (like **std::vector**) maintains a pool of memory into which it can grow and shrink (e.g., using **push_back()** and **pop_back()**). When we add an element, the container must invoke its constructor for a specific address:

```
void C::push_back(const X& a)
{
        // ...
        new(p) X{a};    // copy construct an X with the value a in address p
        // ...
}
```

This use of a constructor is known as "placement **new**" (§11.2.4).

Conversely, when we remove an element, the container needs to invoke its destructor:

```
void C::pop_back()
{
        // ...
        p->~X();   // destroy the X in address p
}
```

The `p->~X()` notation invokes `X`'s destructor for `*p`. That notation should never be used for an object that is destroyed in the normal way (by its object going out of scope or being `delete`d).

For a more complete example of explicit management of objects in a memory area, see §13.6.1.

If declared for a class `X`, a destructor will be implicitly invoked whenever an `X` goes out of scope or is `delete`d. This implies that we can prevent destruction of an `X` by declaring its destructor `=delete` (§17.6.4) or `private`.

Of the two alternatives, using `private` is the more flexible. For example, we can create a class for which objects can be explicitly destroyed, but not implicitly:

```
class Nonlocal {
public:
        // ...
        void destroy() { this->~Nonlocal(); }       // explicit destruction
private:
        // ...
        ~Nonlocal();                                    // don't destroy implicitly
};

void user()
{
        Nonlocal x;                 // error: cannot destroy a Nonlocal
        X* p = new Nonlocal;        // OK
        // ...
        delete p;                   // error: cannot destroy a Nonlocal
        p.destroy();                // OK
}
```

### 17.2.5 `virtual` Destructors

A destructor can be declared to be `virtual`, and usually should be for a class with a virtual function. For example:

```
class Shape {
public:
        // ...
        virtual void draw() = 0;
        virtual ~Shape();
};

class Circle {
public:
        // ...
        void draw();
        ~Circle();          // overrides ~Shape()
        // ...
};
```

The reason we need a `virtual` destructor is that an object usually manipulated through the interface provided by a base class is often also `delete`d through that interface:

```
void user(Shape* p)
{
    p->draw();      // invoke the appropriate draw()
    // ...
    delete p;       // invoke the appropriate destructor
};
```

Had `Shape`'s destructor not been `virtual` that `delete` would have failed to invoke the appropriate derived class destructor (e.g., `~Circle()`). That failure would cause the resources owned by the deleted object (if any) to be leaked.

## 17.3 Class Object Initialization

This section discusses how to initialize objects of a class with and without constructors. It also shows how to define constructors to accept arbitrarily sized homogeneous initializer lists (such as `{1,2,3}` and `{1,2,3,4,5,6}`).

### 17.3.1 Initialization Without Constructors

We cannot define a constructor for a built-in type, yet we can initialize it with a value of suitable type. For example:

```
int a {1};
char* p {nullptr};
```

Similarly, we can initialize objects of a class for which we have not defined a constructor using

- memberwise initialization,
- copy initialization, or
- default initialization (without an initializer or with an empty initializer list).

For example:

```
struct Work {
    string author;
    string name;
    int year;
};

Work s9 { "Beethoven",
          "Symphony No. 9 in D minor, Op. 125; Choral",
          1824
        };                          // memberwise initialization

Work currently_playing { s9 };      // copy initialization
Work none {};                       // default initialization
```

The three members of `currently_playing` are copies of those of `s9`.

The default initialization of using `{}` is defined as initialization of each member by `{}`. So, `none` is initialized to `{{},{},{}}`, which is `{"","",0}` (§17.3.3).

Where no constructor requiring arguments is declared, it is also possible to leave out the initializer completely. For example:

```
Work alpha;

void f()
{
    Work beta;
    // ...
}
```

For this, the rules are not as clean as we might like. For statically allocated objects (§6.4.2), the rules are exactly as if you had used **{}**, so the value of **alpha** is **{"","",0}**. However, for local variables and free-store objects, the default initialization is done only for members of class type, and members of built-in type are left uninitialized, so the value of **beta** is **{"","",unknown}**.

The reason for this complication is to improve performance in rare critical cases. For example:

```
struct Buf {
    int count;
    char buf[16*1024];
};
```

You can use a **Buf** as a local variable without initializing it before using it as a target for an input operation. Most local variable initializations are not performance critical, and uninitialized local variables are a major source of errors. If you want guaranteed initialization or simply dislike surprises, supply an initializer, such as **{}**. For example:

```
Buf buf0;       // statically allocated, so initialized by default

void f()
{
    Buf buf1;           // leave elements uninitialized
    Buf buf2 {};        // I really want to zero out those elements

    int* p1 = new int;  // *p1 is uninitialized
    int* p2 = new int{};// *p2 == 0
    int* p3 = new int{7};// *p3 == 7
    // ...
}
```

Naturally, memberwise initialization works only if we can access the members. For example:

```
template<class T>
class Checked_pointer { // control access to T* member
public:
    T& operator*();     // check for nullptr and return value
// ...
};

Checked_pointer<int> p {new int{7}};    // error: can't access p.p
```

If a class has a private non-**static** data member, it needs a constructor to initialize it.

## 17.3.2 Initialization Using Constructors

Where memberwise copy is not sufficient or desirable, a constructor can be defined to initialize an object. In particular, a constructor is often used to establish an invariant for its class and to acquire resources necessary to do that (§17.2.1).

If a constructor is declared for a class, some constructor will be used for every object. It is an error to try to create an object without a proper initializer as required by the constructors. For example:

```
struct X {
    X(int);
};

X x0;          // error: no initializer
X x1 {};       // error: empty initializer
X x2 {2};      // OK
X x3 {"two"};  // error: wrong initializer type
X x4 {1,2};    // error: wrong number of initializers
X x5 {x4};     // OK: a copy constructor is implicitly defined (§17.6)
```

Note that the default constructor (§17.3.3) disappears when you define a constructor requiring arguments; after all, `X(int)` states that an `int` is required to construct an `X`. However, the copy constructor does not disappear (§17.3.3); the assumption is that an object can be copied (once properly constructed). Where the latter might cause problems (§3.3.1), you can specifically disallow copying (§17.6.4).

I used the `{}` notation to make explicit the fact that I am initializing. I am not (just) assigning a value, calling a function, or declaring a function. The `{}` notation for initialization can be used to provide arguments to a constructor wherever an object can be constructed. For example:

```
struct Y : X {
    X m {0};                    // provide default initializer for member m
    Y(int a) :X{a}, m{a} { };   // initialize base and member (§17.4)
    Y() : X{0} { };             // initialize base and member
};

X g {1};   // initialize global variable

void f(int a)
{
    X def {};                   // error: no default value for X
    Y de2 {};                   // OK: use default constructor
    X* p {nullptr};
    X var {2};                  // initialize local variable
    p = new X{4};               // initialize object on free store
    X a[] {1,2,3};              // initialize array elements
    vector<X> v {1,2,3,4};      // initialize vector elements
}
```

For this reason, {} initialization is sometimes referred to as *universal* initialization: the notation can be used everywhere. In addition, {} initialization is *uniform*: wherever you initialize an object of type X with a value v using the {v} notation, the same value of type X (X{v}) is created.

The = and () notations for initialization (§6.3.5) are not universal. For example:

```
struct Y : X {
        X m;
        Y(int a) : X(a), m=a { };     // syntax error: can't use = for member initialization
};

X g(1);     // initialize global variable

void f(int a)
{
        X def();                      // function returning an X (surprise!?)
        X* p {nullptr};
        X var = 2;                    // initialize local variable
        p = new X=4;                  // syntax error: can't use = for new
        X a[](1,2,3);                 // error: can't use () for array initialization
        vector<X> v(1,2,3,4);         // error: can't use () for list elements
}
```

The = and () notations for initialization are not uniform either, but fortunately the examples of that are obscure. If you insist on using = or () initialization, you have to remember where they are allowed and what they mean.

The usual overload resolution rules (§12.3) apply for constructors. For example:

```
struct S {
        S(const char*);
        S(double*);
};

S s1 {"Napier"};              // S::S(const char*)
S s2 {new double{1.0}};       // S::S(double*);
S s3 {nullptr};               // ambiguous: S::S(const char*) or S::S(double*)?
```

Note that the {}-initializer notation does not allow narrowing (§2.2.2). That is another reason to prefer the {} style over () or =.

### 17.3.2.1 Initialization by Constructors

Using the () notation, you can request to use a constructor in an initialization. That is, you can ensure that for a class, you will get initialization by constructor and not get the memberwise initialization or initializer-list initialization (§17.3.4) that the {} notation also offers. For example:

```
struct S1 {
        int a,b;                              // no constructor
};
```

```
struct S2 {
        int a,b;
        S2(int a = 0, int b = 0) : a(aa), b(bb) {}          // constructor
};

S1 x11(1,2);    // error: no constructor
S1 x12 {1,2};   // OK: memberwise initialization

S1 x13(1);      // error: no constructor
S1 x14 {1};     // OK: x14.b becomes 0

S2 x21(1,2);    // OK: use constructor
S2 x22 {1,2};   // OK: use constructor

S2 x23(1);      // OK: use constructor and one default argument
S2 x24 {1};     // OK: use constructor and one default argument
```

The uniform use of **{}** initialization only became possible in C++11, so older C++ code uses **()** and **=** initialization. Consequently, the **()** and **=** may be more familiar to you. However, I don't know any logical reason to prefer the **()** notation except in the rare case where you need to distinguish between initialization with a list of elements and a list of constructor arguments. For example:

```
vector<int> v1 {77};        // one element with the value 77
vector<int> v2(77);         // 77 elements with the default value 0
```

This problem – and the need to choose – can occur when a type with an initializer-list constructor (§17.3.4), typically a container, also has an "ordinary constructor" accepting arguments of the element type. In particular, we occasionally must use **()**initialization for **vector**s of integers and floating-point numbers but never need to for **vector**s of strings of pointers:

```
vector<string> v1 {77};         // 77 elements with the default value ""
                                // (vector<string>(std::initializer_list<string>) doesn't accept {77})
vector<string> v2(77);          // 77 elements with the default value ""

vector<string> v3 {"Booh!"};    // one element with the value "Booh!"
vector<string> v4("Booh!");     // error: no constructor takes a string argument

vector<int*> v5 {100,0};        // 100 int*s initialized to nullptr (100 is not an int*)

vector<int*> v6 {0,0};          // 2 int*s initialized to nullptr
vector<int*> v7(0,0);           // empty vector (v7.size()==0)
vector<int*> v8;                // empty vector (v7.size()==0)
```

The **v6** and **v7** examples are only of interest to language lawyers and testers.

## 17.3.3 Default Constructors

A constructor that can be invoked without an argument is called a *default constructor*. Default constructors are very common. For example:

```
class Vector {
public:
    Vector();  // default constructor: no elements
    // ...
};
```

A default constructor is used if no arguments are specified or if an empty initializer list is provided:

```
Vector v1;    // OK
Vector v2 {};  // OK
```

A default argument (§12.2.5) can make a constructor that takes arguments into a default constructor. For example:

```
class String {
public:
    String(const char* p = "");    // default constructor: empty string
    // ...
};

String s1;    // OK
String s2 {};  // OK
```

The standard-library **vector** and **string** have such default constructors (§36.3.2, §31.3.2).

The built-in types are considered to have default and copy constructors. However, for a built-in type the default constructor is not invoked for uninitialized non-**static** variables (§17.3). The default value of a built-in type is **0** for integers, **0.0** for floating-point types, and **nullptr** for pointers. For example:

```
void f()
{
    int a0;              // uninitialized
    int a1();            // function declaration (intended?)

    int a {};            // a becomes 0
    double d {};         // d becomes 0.0
    char* p {};          // p becomes nullptr

    int* p1 = new int;   // uninitialized int
    int* p2 = new int{}; // the int is initialized to 0
}
```

Constructors for built-in types are most often used for template arguments. For example:

```
template<class T>
struct Handle {
    T* p;
    Handle(T* pp = new T{}) :p{pp} { }
    // ...
};

Handle<int> px;    // will generate int{}
```

The generated **int** will be initialized to **0**.

References and **const**s must be initialized (§7.7, §7.5). Therefore, a class containing such members cannot be default constructed unless the programmer supplies in-class member initializers (§17.4.4) or defines a default constructor that initializes them (§17.4.1). For example:

```
int glob {9};

struct X {
        const int a1 {7};      // OK
        const int a2;          // error: requires a user-defined constructor
        const int& r {9};      // OK
        int& r1 {glob};        // OK
        int& r2;               // error: requires a user-defined constructor
};

X x;        // error: no default constructor for X
```

An array, a standard-library **vector**, and similar containers can be declared to allocate a number of default-initialized elements. In such cases, a default constructor is obviously required for a class used as the element type of a **vector** or array. For example:

```
struct S1 { S1(); };                    // has default constructor
struct S2 { S2(string); };              // no default constructor

S1 a1[10];                              // OK: 10 default elements
S2 a2[10];                              // error: cannot initialize elements
S2 a3[] { "alpha", "beta" };            // OK: two elements: S2{"alpha"}, S2{"beta"}

vector<S1> v1(10);                      // OK: 10 default elements
vector<S2> v2(10);                      // error: cannot initialize elements
vector<S2> v3 { "alpha", "beta" };      // OK: two elements: S2{"alpha"}, S2{"beta"}

vector<S2> v2(10,"");                   // OK: 10 elements each initialized to S2{""}
vector<S2> v4;                          // OK: no elements
```

When should a class have a default constructor? A simple-minded technical answer is "when you use it as the element type for an array, etc." However, a better question is "For what types does it make sense to have a default value?" or even "Does this type have a 'special' value we can 'naturally' use as a default?" String has the empty string, **""**, containers have the empty set, **{}**, and numeric values have zero. The trouble with deciding on a default **Date** (§16.3) arose because there is no "natural" default date (the Big Bang is too far in the past and not precisely associated with our everyday dates). It is a good idea not to be too clever when inventing default values. For example, the problem with containers of elements without default values is often best solved by not allocating elements until you have proper values for them (e.g., using **push_back()**).

## 17.3.4 Initializer-List Constructors

A constructor that takes a single argument of type **std::initializer_list** is called an *initializer-list constructor*. An initializer-list constructor is used to construct objects using a **{}**-list as its initializer

value. Standard-library containers (e.g., **vector** and **map**) have initializer-list constructors, assignments, etc. (§31.3.2, §31.4.3). Consider:

```
vector<double> v = { 1, 2, 3.456, 99.99 };

list<pair<string,string>> languages = {
        {"Nygaard","Simula"}, {"Richards","BCPL"}, {"Ritchie","C"}
};

map<vector<string>,vector<int>> years = {
        { {"Maurice","Vincent", "Wilkes"},{1913, 1945, 1951, 1967, 2000} },
        { {"Martin", "Richards"} {1982, 2003, 2007} },
        { {"David", "John", "Wheeler"}, {1927, 1947, 1951, 2004} }
};
```

The mechanism for accepting a **{}**-list is a function (often a constructor) taking an argument of type **std::initializer_list<T>**. For example:

```
void f(initializer_list<int>);

f({1,2});
f({23,345,4567,56789});
f({});      // the empty list

f{1,2};     // error: function call () missing

years.insert({{"Bjarne","Stroustrup"},{1950, 1975, 1985}});
```

The initializer list can be of arbitrary length but must be homogeneous. That is, all elements must be of the template argument type, **T**, or implicitly convertible to **T**.

### 17.3.4.1 initializer_list Constructor Disambiguation

When you have several constructors for a class, the usual overload resolution rules (§12.3) are used to select the right one for a given set of arguments. For selecting a constructor, default and initializer lists take precedence. Consider:

```
struct X {
        X(initializer_list<int>);
        X();
        X(int);
};

X x0 {};    // empty list: default constructor or initializer-list constructor? (the default constructor)
X x1 {1};   // one integer: an int argument or a list of one element? (the  initializer-list constructor)
```

The rules are:

- If either a default constructor or an initializer-list constructor could be invoked, prefer the default constructor.
- If both an initializer-list constructor and an ‘‘ordinary constructor’’ could be invoked, prefer the initializer-list constructor.

The first rule, "prefer the default constructor," is basically common sense: pick the simplest constructor when you can. Furthermore, if you define an initializer-list constructor to do something with an empty list that differs from what the default constructor does, you probably have a design error on your hands.

The second rule, "prefer the initializer-list constructor," is necessary to avoid different resolutions based on different numbers of elements. Consider **std::vector** (§31.4):

```
vector<int> v1 {1};        // one element
vector<int> v2 {1,2};      // two elements
vector<int> v3 {1,2,3};    // three elements

vector<string> vs1 {"one"};
vector<string> vs2 {"one", "two"};
vector<string> vs3 {"one", "two", "three"};
```

In every case, the initializer-list constructor is used. If we really want to invoke the constructor taking one or two integer arguments, we must use the **()** notation:

```
vector<int> v1(1);   // one element with the default value (0)
vector<int> v2(1,2); // one element with the value 2
```

### 17.3.4.2 Use of initializer_lists

A function with an **initializer_list<T>** argument can access it as a sequence using the member functions **begin()**, **end()**, and **size()**. For example:

```
void f(initializer_list<int> args)
{
    for (int i = 0; i!=args.size(); ++i)
        cout << args.begin()[i] << "\n";
}
```

Unfortunately, **initializer_list** doesn't provide subscripting.

An **initializer_list<T>** is passed by value. That is required by the overload resolution rules (§12.3) and does not impose overhead because an **initializer_list<T>** object is just a small handle (typically two words) to an array of **T**s.

That loop could equivalently have been written:

```
void f(initializer_list<int> args)
{
    for (auto p=args.begin(); p!=args.end(); ++p)
        cout << *p << "\n";
}
```

or:

```
void f(initializer_list<int> args)
{
    for (auto x : args)
        cout << x << "\n";
}
```

To explicitly use an initializer_list you must #include the header file in which it is defined: <initializer_list>. However, since vector, map, etc., use initializer_lists, their headers (<vector>, <map>, etc.) already #include <initializer_list>, so you rarely have to do so directly.

The elements of an initializer_list are immutable. Don't even think about trying to modify their values. For example:

```
int f(std::initializer_list<int> x, int val)
{
    *x.begin() = val;          // error: attempt to change the value of an initializer-list element
    return *x.begin();    // OK
}

void g()
{
    for (int i=0; i!=10; ++i)
        cout << f({1,2,3},i) << '\n';
}
```

Had the assignment in f() succeeded, it would have appeared that the value of 1 (in {1,2,3}) could change. That would have done serious damage to some of our most fundamental concepts. Because initializer_list elements are immutable, we cannot apply a move constructor (§3.3.2, §17.5.2) to them.

A container might implement an initializer-list constructor like this:

```
template<class E>
class Vector {
public:
    Vector(std::initializer_list<E> s); // initializer-list constructor
    // ...
private:
    int sz;
    E* elem;
};

template<class E>
Vector::Vector(std::initializer_list<E> s)
    :sz{s.size()}                                  // set vector size
{
    reserve(sz);                                   // get the right amount of space
    uninitialized_copy(s.begin(), s.end(), elem);      // initialize elements in elem[0:s.size())
}
```

The initializer lists are part of the universal and uniform initialization design (§17.3).

### 17.3.4.3 Direct and Copy Initialization

The distinction between direct initialization and copy initialization (§16.2.6) is maintained for {} initialization. For a container, this implies that the distinction is applied to both the container and its elements:

- The container's initializer-list constructor can be **explicit** or not.
- The constructor of the element type of the initializer list can be **explicit** or not.

For a **vector<vector<double>>**, we can see the direct initialization vs. copy initialization distinction applied to elements. For example:

```
vector<vector<double>> vs = {
        {10,11,12,13,14},               // OK: vector of five elements
        {10},                           // OK: vector of one element
        10,                             // error: vector<double>(int) is explicit

        vector<double>{10,11,12,13},    // OK: vector of five elements
        vector<double>{10},             // OK: vector of one element with value 10.0
        vector<double>(10),             // OK:  vector of 10 elements with value 0.0
};
```

A container can have some constructors explicit and some not. The standard-library **vector** is an example of that. For example, **std::vector<int>(int)** is **explicit**, but **std::vector<int>(initialize_list<int>)** is not:

```
vector<double> v1(7);   // OK: v1 has 7 elements; note: uses () rather than {}
vector<double> v2 = 9;  // error: no conversion from int to vector

void f(const vector<double>&);
void g()
{
        v1 = 9;         // error: no conversion from int to vector
        f(9);           // error: no conversion from int to vector
}
```

By replacing **()** with **{}** we get:

```
vector<double> v1 {7};          // OK: v1 has one element (with the value 7)
vector<double> v2 = {9};        // OK: v2 has one element (with the value 9)

void f(const vector<double>&);
void g()
{
        v1 = {9};       // OK: v1 now has one element (with the value 9)
        f({9});         // OK: f is called with the list {9}
}
```

Obviously, the results are dramatically different.

This example was carefully crafted to give an example of the most confusing cases. Note that the apparent ambiguities (in the eyes of the human reader but not the compiler) do not emerge for longer lists. For example:

```
vector<double> v1 {7,8,9};          // OK: v1 has three elements with values {7,8,9}
vector<double> v2 = {9,8,7};        // OK: v2 has three elements with values {9,8,7}
```

```
void f(const vector<double>&);
void g()
{
        v1 = {9,10,11};         // OK: v1 now has three elements with values {9,10,11}
        f({9,8,7,6,5,4});       // OK: f is called with the list {9,8,7,6,5,4}
}
```

Similarly, the potential ambiguities do not occur for lists of elements of nonintegral types:

```
vector<string> v1 { "Anya"};          // OK: v1 has one element (with the value "Anya")
vector<string> v2 = {"Courtney"};     // OK: v2 has one element (with the value "Courtney")

void f(const vector<string>&);
void g()
{
        v1 = {"Gavin"};         // OK: v1 now has one element (with the value "Gavin")
        f({"Norah"});           // OK: f is called with the list {"Norah"}
}
```

## 17.4 Member and Base Initialization

Constructors can establish invariants and acquire resources. Generally, they do that by initializing class members and base classes.

### 17.4.1 Member Initialization

Consider a class that might be used to hold information for a small organization:

```
class Club {
        string name;
        vector<string> members;
        vector<string> officers;
        Date founded;
        // ...
        Club(const string& n, Date fd);
};
```

The `Club`'s constructor takes the name of the club and its founding date as arguments. Arguments for a member's constructor are specified in a *member initializer list* in the definition of the constructor of the containing class. For example:

```
Club::Club(const string& n, Date fd)
        : name{n}, members{}, officers{}, founded{fd}
{
        // ...
}
```

The member initializer list starts with a colon, and the individual member initializers are separated by commas.

The members' constructors are called before the body of the containing class's own constructor is executed (§17.2.3). The constructors are called in the order in which the members are declared in the class rather than the order in which the members appear in the initializer list. To avoid confusion, it is best to specify the initializers in the member declaration order. Hope for a compiler warning if you don't get the order right. The member destructors are called in the reverse order of construction after the body of the class's own destructor has been executed.

If a member constructor needs no arguments, the member need not be mentioned in the member initializer list. For example:

```
Club::Club(const string& n, Date fd)
        : name{n}, founded{fd}
{
        // ...
}
```

This constructor is equivalent to the previous version. In each case, **Club::officers** and **Club::members** are initialized to a **vector** with no elements.

It is usually a good idea to be explicit about initializing members. Note that an "implicitly initialized" member of a built-in type is left uninitialized (§17.3.1).

A constructor can initialize members and bases of its class, but not members or bases of its members or bases. For example:

```
struct B { B(int); /* ... */};
struct BB : B { /* ... */ };
struct BBB : BB {
        BBB(int i) : B(i) { };  // error: trying to initialize base's base
        // ...
};
```

### 17.4.1.1 Member Initialization and Assignment

Member initializers are essential for types for which the meaning of initialization differs from that of assignment. For example:

```
class X {
        const int i;
        Club cl;
        Club& rc;
        // ...
        X(int ii, const string& n, Date d, Club& c) : i{ii}, cl{n,d}, rc{c} { }
};
```

A reference member or a **const** member must be initialized (§7.5, §7.7, §17.3.3). However, for most types the programmer has a choice between using an initializer and using an assignment. In that case, I usually prefer to use the member initializer syntax to make it explicit that initialization is being done. Often, there also is an efficiency advantage to using the initializer syntax (compared to using an assignment). For example:

```
class Person {
        string name;
        string address;
        // ...
        Person(const Person&);
        Person(const string& n, const string& a);
};

Person::Person(const string& n, const string& a)
        : name{n}
{
        address = a;
}
```

Here **name** is initialized with a copy of **n**. On the other hand, **address** is first initialized to the empty string and then a copy of **a** is assigned.

### 17.4.2 Base Initializers

Bases of a derived class are initialized in the same way non-data members are. That is, if a base requires an initializer, it must be provided as a base initializer in a constructor. If we want to, we can explicitly specify default construction. For example:

```
class B1 { B1(); };    // has default constructor
class B2 { B2(int); } // no default constructor

struct D1 : B1, B2 {
        D1(int i) :B1{}, B2{i} {}
};

struct D2 : B1, B2 {
        D2(int i) :B2{i} {}          // B1{} is used implicitly
};

struct D1 : B1, B2 {
        D1(int i) { }                // error: B2 requires an int initializer
};
```

As with members, the order of initialization is the declaration order, and it is recommended to specify base initializers in that order. Bases are initialized before members and destroyed after members (§17.2.3).

### 17.4.3 Delegating Constructors

If you want two constructors to do the same action, you can repeat yourself or define "an **init()** function" to perform the common action. Both "solutions" are common (because older versions of C++ didn't offer anything better). For example:

```
class X {
        int a;
        validate(int x) { if (0<x && x<=max) a=x; else throw Bad_X(x); }
public:
        X(int x) { validate(x); }
        X() { validate(42); }
        X(string s) { int x = to<int>(s); validate(x); }     // §25.2.5.1
        // ...
};
```

Verbosity hinders readability and repetition is error-prone. Both get in the way of maintainability. The alternative is to define one constructor in terms of another:

```
class X {
        int a;
public:
        X(int x) { if (0<x && x<=max) a=x; else throw Bad_X(x); }
        X() :X{42} { }
        X(string s) :X{to<int>(s)} { }          // §25.2.5.1
        // ...
};
```

That is, a member-style initializer using the class's own name (its constructor name) calls another constructor as part of the construction. Such a constructor is called a *delegating constructor* (and occasionally a *forwarding constructor*).

You cannot both delegate and explicitly initialize a member. For example:

```
class X {
        int a;
public:
        X(int x) { if (0<x && x<=max) a=x; else throw Bad_X(x); }
        X() :X{42}, a{56} { }          // error
        // ...
};
```

Delegating by calling another constructor in a constructor's member and base initializer list is very different from explicitly calling a constructor in the body of a constructor. Consider:

```
class X {
        int a;
public:
        X(int x) { if (0<x && x<=max) a=x; else throw Bad_X(x); }
        X() { X{42}; }     // likely error
        // ...
};
```

The `X{42}` simply creates a new unnamed object (a temporary) and does nothing with it. Such use is more often than not a bug. Hope for a compiler warning.

An object is not considered constructed until its constructor completes (§6.4.2). When using a delegating constructor, the object is not considered constructed until the delegating constructor completes – just completing the delegated-to constructor is not sufficient. A destructor will not be

called for an object unless its original constructor completed.

If all you need is to set a member to a default value (that doesn't depend on a constructor argument), a member initializer (§17.4.4) may be simpler.

### 17.4.4 In-Class Initializers

We can specify an initializer for a non-**static** data member in the class declaration. For example:

```
class A {
public:
    int a {7};
    int b = 77;
};
```

For pretty obscure technical reasons related to parsing and name lookup, the **{}** and **=** initializer notations can be used for in-class member initializers, but the **()** notation cannot.

By default, a constructor will use such an in-class initializer, so that example is equivalent to:

```
class A {
public:
    int a;
    int b;
    A() : a{7}, b{77} {}
};
```

Such use of in-class initializers can save a bit of typing, but the real benefits come in more complicated classes with multiple constructors. Often, several constructors use the same initializer for a member. For example:

```
class A {
public:
    A() :a{7}, b{5}, algorithm{"MD5"}, state{"Constructor run"} {}
    A(int a_val) :a{a_val}, b{5}, algorithm{"MD5"}, state{"Constructor run"} {}
    A(D d) :a{7}, b{g(d)}, algorithm{"MD5"}, state{"Constructor run"} {}
    // ...
private:
    int a, b;
    HashFunction algorithm;     // cryptographic hash to be applied to all As
    string state;               // string indicating state in object life cycle
};
```

The fact that **algorithm** and **state** have the same value in all constructors is lost in the mess of code and can easily become a maintenance problem. To make the common values explicit, we can factor out the unique initializer for data members:

```
class A {
public:
    A() :a{7}, b{5} {}
    A(int a_val) :a{a_val}, b{5} {}
    A(D d) :a{7}, b{g(d)} {}
    // ...
```

```
private:
        int a, b;
        HashFunction algorithm {"MD5"};      // cryptographic hash to be applied to all As
        string state {"Constructor run"};    // string indicating state in object life cycle
};
```

If a member is initialized by both an in-class initializer and a constructor, only the constructor's initialization is done (it "overrides" the default). So we can simplify further:

```
class A {
public:
        A() {}
        A(int a_val) :a{a_val} {}
        A(D d) :b{g(d)} {}
        // ...
private:
        int a {7};                           // the meaning of 7 for a is ...
        int b {5};                           // the meaning of 5 for b is ...
        HashFunction algorithm {"MD5"};      // Cryptographic hash to be applied to all As
        string state {"Constructor run"};    // String indicating state in object lifecycle
};
```

As shown, default in-class initializers provide an opportunity for documentation of common cases.

An in-class member initializer can use names that are in scope at the point of their use in the member declaration. Consider the following headache-inducing technical example:

```
int count = 0;
int count2 = 0;

int f(int i) { return i+count; }

struct S {
        int m1 {count2};     // that is, ::count2
        int m2 {f(m1)};      // that is, this->m1+::count; that is, ::count2+::count
        S() { ++count2; }    // very odd constructor
};

int main()
{
        S s1;       // {0,0}
        ++count;
        S s2;       // {1,2}
}
```

Member initialization is done in declaration order (§17.2.3), so first **m1** is initialized to the value of a global variable **count2**. The value of the global variable is obtained at the point where the constructor for a new **S** object is run, so it can (and in this example does) change. Next, **m2** is initialized by a call to the global **f()**.

It is a bad idea to hide subtle dependencies on global data in member initializers.

### 17.4.5 static Member Initialization

A **static** class member is statically allocated rather than part of each object of the class. Generally, the **static** member declaration acts as a declaration for a definition outside the class. For example:

```
class Node {
    // ...
    static int node_count;          // declaration
};

int Node::node_count = 0;           // definition
```

However, for a few simple special cases, it is possible to initialize a **static** member in the class declaration. The **static** member must be a **const** of an integral or enumeration type, or a **constexpr** of a literal type (§10.4.3), and the initializer must be a *constant-expression*. For example:

```
class Curious {
public:
    static const int c1 = 7;          // OK
    static int c2 = 11;               // error: not const
    const int c3 = 13;                // OK, but not static (§17.4.4)
    static const int c4 = sqrt(9);    // error: in-class initializer not constant
    static const float c5 = 7.0;      // error: in-class not integral (use constexpr rather than const)
    // ...
};
```

If (and only if) you use an initialized member in a way that requires it to be stored as an object in memory, the member must be (uniquely) defined somewhere. The initializer may not be repeated:

```
const int Curious::c1;              // don't repeat initializer here
const int* p = &Curious::c1;        // OK: Curious::c1 has been defined
```

The main use of member constants is to provide symbolic names for constants needed elsewhere in the class declaration. For example:

```
template<class T, int N>
class Fixed {    // fixed-size array
public:
    static constexpr int max = N;
    // ...
private:
    T a[max];
};
```

For integers, enumerators (§8.4) offer an alternative for defining symbolic constants within a class declaration. For example:

```
class X {
    enum { c1 = 7, c2 = 11, c3 = 13, c4 = 17 };
    // ...
};
```

## 17.5 Copy and Move

When we need to transfer a value from **a** to **b**, we usually have two logically distinct options:

- *Copy* is the conventional meaning of **x=y**; that is, the effect is that the values of **x** and **y** are both equal to **y**'s value before the assignment.
- *Move* leaves **x** with **y**'s former value and **y** with some *moved-from state.* For the most interesting cases, containers, that moved-from state is "empty."

This simple logical distinction is confounded by tradition and by the fact that we use the same notation for both move and copy.

Typically, a move cannot throw, whereas a copy might (because it may need to acquire a resource), and a move is often more efficient than a copy. When you write a move operation, you should leave the source object in a valid but unspecified state because it will eventually be destroyed and the destructor cannot destroy an object left in an invalid state. Also, standard-library algorithms rely on being able to assign to (using move or copy) a moved-from object. So, design your moves not to throw, and to leave their source objects in a state that allows destruction and assignment.

To save us from tedious repetitive work, copy and move have default definitions (§17.6.2).

### 17.5.1 Copy

Copy for a class **X** is defined by two operations:

- Copy constructor: **X(const X&)**
- Copy assignment: **X& operator=(const X&)**

You can define these two operations with more adventurous argument types, such as **volatile X&**, but don't; you'll just confuse yourself and others. A copy constructor is supposed to make a copy of an object without modifying it. Similarly, you can use **const X&** as the return type of the copy assignment. My opinion is that doing so causes more confusion than it is worth, so my discussion of copy assumes that the two operations have the conventional types.

Consider a simple two-dimensional **Matrix**:

```
template<class T>
class Matrix {
    array<int,2> dim;    // two dimensions
    T* elem;             // pointer to dim[0]*dim[1] elements of type T
public:
    Matrix(int d1, int d2) :dim{d1,d2}, elem{new T[d1*d2]} {}      // simplified (no error handling)
    int size() const { return dim[0]*dim[1]; }

    Matrix(const Matrix&);                // copy constructor
    Matrix& operator=(const Matrix&);     // copy assignment

    Matrix(Matrix&&);                     // move constructor
    Matrix& operator=(Matrix&&);          // move assignment

    ~Matrix() { delete[] elem; }
    // ...
};
```

First we note that the default copy (copy the members) would be disastrously wrong: the **Matrix** elements would not be copied, the **Matrix** copy would have a pointer to the same elements as the source, and the **Matrix** destructor would delete the (shared) elements twice (§3.3.1).

However, the programmer can define any suitable meaning for these copy operations, and the conventional one for a container is to copy the contained elements:

```
template<class T>
Matrix:: Matrix(const Matrix& m)          // copy constructor
      : dim{m.dim},
        elem{new T[m.size()]}
{
      uninitialized_copy(m.elem,m.elem+m.size(),elem);       // copy elements
}

template<class T>
Matrix& Matrix::operator=(const Matrix& m)        // copy assignment
{
      if (dim[0]!=m.dim[0] || dim[1]!=m.dim[1])
            throw runtime_error("bad size in Matrix =");
      copy(m.elem,m.elem+m.size(),elem);     // copy elements
}
```

A copy constructor and a copy assignment differ in that a copy constructor initializes uninitialized memory, whereas the copy assignment operator must correctly deal with an object that has already been constructed and may own resources.

The **Matrix** copy assignment operator has the property that if a copy of an element throws an exception, the target of the assignment may be left with a mixture of its old value and the new. That is, that **Matrix** assignment provided the basic guarantee, but not the strong guarantee (§13.2). If that is not considered acceptable, we can avoid it by the fundamental technique of first making a copy and then swapping representations:

```
Matrix& Matrix::operator=(const Matrix& m)        // copy assignment
{
      Matrix tmp {m};          // make a copy
      swap(tmp,*this);         // swap tmp's representation with *this's
      return *this;
}
```

The **swap()** will be done only if the copy was successful. Obviously, this **operator=()** works only if the implementation **swap()** does not use assignment (**std::swap()** does not); see §17.5.2.

Usually a copy constructor must copy every non-**static** member (§17.4.1). If a copy constructor cannot copy an element (e.g., because it needs to acquire an unavailable resource to do so), it can throw an exception.

Note that I did not protect **Matrix**'s copy assignment against self-assignment, **m=m**. The reason I did not test is that self-assignment of the members is already safe: both my implementations of **Matrix**'s copy assignment will work correctly and reasonably efficiently for **m=m**. Also, self-assignment is rare, so test for self-assignment in a copy assignment only if you are sure that you need to.

### 17.5.1.1 Beware of Default Constructors

When writing a copy operation, be sure to copy every base and member. Consider:

```
class X {
        string s;
        string s2;
        vector<string> v;

        X(const X&)             // copy constructor
                :s{a.s}, v{a.v}  // probably sloppy and probably wrong
        {
        }
        // ...
};
```

Here, I "forgot" to copy `s2`, so it gets default initialized (to `""`). This is unlikely to be right. It is also unlikely that I would make this mistake for a simple class. However, for larger classes the chances of forgetting go up. Worse, when someone long after the initial design adds a member to a class, it is easy to forget to add it to the list of members to be copied. This is one reason to prefer the default (compiler-generated) copy operations (§17.6).

### 17.5.1.2 Copy of Bases

For the purposes of copying, a base is just a member: to copy an object of a derived class you have to copy its bases. For example:

```
struct B1 {
        B1();
        B1(const B1&);
        // ...
};

struct B2 {
        B2(int);
        B2(const B2&);
        // ...
};

struct D : B1, B2 {
        D(int i) :B1{}, B2{i}, m1{}, m2{2*i} {}
        D(const D& a) :B1{a}, B2{a}, m1{a.m1}, m2{a.m2} {}
        B1 m1;
        B2 m2;
};

D d {1};    // construct with int argument
D dd {d};  // copy construct
```

The order of initialization is the usual (base before member), but for copying the order had better not matter.

A **virtual** base (§21.3.5) may appear as a base of several classes in a hierarchy. A default copy constructor (§17.6) will correctly copy it. If you define your own copy constructor, the simplest technique is to repeatedly copy the **virtual** base. Where the base object is small and the **virtual** base occurs only a few times in a hierarchy, that can be more efficient than techniques for avoiding the replicated copies.

### 17.5.1.3 The Meaning of Copy

What does a copy constructor or copy assignment have to do to be considered "a proper copy operation"? In addition to be declared with a correct type, a copy operation must have the proper copy semantics. Consider a copy operation, **x=y**, of two objects of the same type. To be suitable for value-oriented programming in general (§16.3.4), and for use with the standard library in particular (§31.2.2), the operation must meet two criteria:

- *Equivalence*: After **x=y**, operations on **x** and **y** should give the same result. In particular, if **==** is defined for their type, we should have **x==y** and **f(x)==f(y)** for any function **f()** that depends only on the values of **x** and **y** (as opposed to having its behavior depend on the addresses of **x** and **y**).
- *Independence*: After **x=y**, operations on **x** should not implicitly change the state of **y**, that is **f(x)** does not change the value of **y** as long as **f(x)** doesn't refer to **y**.

This is the behavior that **int** and **vector** offer. Copy operations that provide equivalence and independence lead to simpler and more maintainable code. This is worth stating because code that violate these simple rules is not uncommon, and programmers don't always realize that such violations are the root cause of some of their nastier problems. A copy that provides equivalence and independence is part of the notion of a regular type (§24.3.1).

First consider the requirement of equivalence. People rarely violate this requirement deliberately, and the default copy operations do not violate it; they do memberwise copy (§17.3.1, §17.6.2). However, tricks, such as having the meaning of copy depend on "options," occasionally appear and typically cause confusion. Also, it is not uncommon for an object to contain members that are not considered part of its value. For example, a copy of a standard container does not copy its allocator because the allocator is considered part of the container, rather than part of its value. Similarly, counters for statistics gathering and cached values are sometimes not simply copied. Such "non-value" parts of an object's state should not affect the result of comparison operators. In particular, **x=y** should imply **x==y**. Furthermore, slicing (§17.5.1.4) can lead to "copies" that behave differently, and is most often a bad mistake.

Now consider the requirement of independence. Most of the problems related to (lack of) independence have to do with objects that contain pointers. The default meaning of copy is memberwise copy. A default copy operation copies a pointer member, but does not copy the object (if any) that it points to. For example:

```
struct S {
    int* p;     // a pointer
};

S x {new int{0}};
```

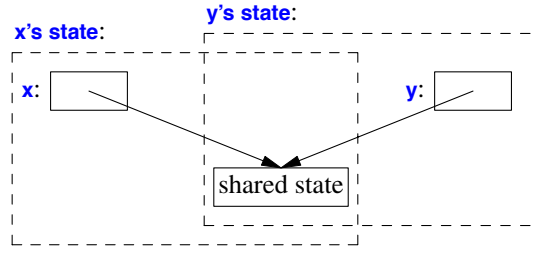
```
void f()
{
    S y {x};            // "copy" x

    *y.p = 1;           // change y; affects x
    *x.p = 2;           // change x; affects y
    delete y.p;         // affects x and y
    y.p = new int{3};   // OK: change y; does not affect x
    *x.p = 4;           // oops: write to deallocated memory
}
```

Here I violated the rule of independence. After the "copy" of **x** into **y**, we can manipulate part of **x**'s state through **y**. This is sometimes called *shallow copy* and (too) often praised for "efficiency." The obvious alternative of copying the complete state of an object is called *deep copy*. Often, the better alternative to deep copy is not a shallow copy, but a move operation, which minimizes copying without adding complexity (§3.3.2, §17.5.2).

A shallow copy leaves two objects (here, **x** and **y**) with a *shared state*, and has a huge potential for confusion and errors. We say that the objects **x** and **y** have become *entangled* when the requirement of independence have been violated. It is not possible to reason about an entangled object in isolation. For example, it is not obvious from the source code that the two assignments to ***x.p** can have dramatically different effects.

We can represent two entangled objects graphically:




Note that entanglement can arise in a variety of ways. Often, it is not obvious that entanglement has happened until probems arise. For example, a type like **S** may incautiously be used as a member of an otherwise well-behaved class. The original author of **S** may be aware of the entanglement and prepared to cope with it, but someone naively assuming that copying an **S** meant copying its complete value could be surprised, and someone who finds an **S** deeply nested in other classes could be very surprised.

We can address problems related to the lifetime of a shared subobject by introducing a form of garbage collection. For example:

```
struct S2 {
    shared_ptr<int> p;
};

S2 x {new int{0}};
```

```
void f()
{
        S2 y {x};                   // "copy" x

        *y.p = 1;                   // change y, affects x
        *x.p = 2;                   // change x; affects y
        y.p.reset(new int{3});      // change y; affects x
        *x.p = 4;                   // change x; affects y
}
```

In fact, shallow copy and such entangled objects are among the sources of demands for garbage collection. Entangled objects lead to code that is very hard to manage without some form of garbage collection (e.g., **shared_ptr**s).

However, a **shared_ptr** is still a pointer, so we cannot consider objects containing a **shared_ptr** in isolation. Who can update the pointed-to object? How? When? If we are running in a multi-threaded system, is synchronization needed for access to the shared data? How can we be sure? Entangled objects (here, resulting from a shallow copy) is a source of complexity and errors that is at best partially solved by garbage collection (in any form).

Note that an immutable shared state is not a problem. Unless we compare addresses, we cannot tell whether two equal values happen to be represented as one or two copies. This is a useful observation because many copies are never modified. For example, objects passed by value are rarely written to. This observation leads to the notion of *copy-on-write*. The idea is that a copy doesn't actually need independence until a shared state is written to, so we can delay the copying of the shared state until just before the first write to it. Consider:

```
class Image {
public:
        // ...
        Image(const Image& a);          // copy constructor
        // ...
        void write_block(Descriptor);
        // ...
private:
        Representation* clone();        // copy *rep
        Representation* rep;
        bool shared;
};
```

Assume that a **Representation** can be huge and that a **write_block()** is expensive compared to testing a **bool**. Then, depending on the use of **Image**s, it can make sense to implement the copy constructor as a shallow copy:

```
Image::Image(const Image& a)        // do shallow copy and prepare for copy-on-write
        :rep{a.rep},
         shared{true}
{
}
```

We protect the argument to that copy constructor by copying the **Representation** before a write:

```
void write_block(Descriptor d)
{
        if (shared) {
                rep =clone();           // make a copy of *rep
                shared = false;         // no more sharing
        }
        // ... now we can safely write to our own copy of rep ...
}
```

Like any other technique, copy-on-write is not a panacea, but it can be an effective combination of the simplicity of true copy and the efficiency of shallow copy.

### 17.5.1.4 Slicing

A pointer to a derived class implicitly converts to a pointer to its public base class. When applied to a copy operation, this simple and necessary rule (§3.2.4, §20.2) leads to a trap for the unwary. Consider:

```
struct Base {
        int b;
        Base(const Base&);
        // ...
};

struct Derived : Base {
        int d;
        Derived(const Derived&);
        // ...
};

void naive(Base* p)
{
        B b2 = *p;      // may slice: invokes Base::Base(const Base&)
        // ...
}

void user()
{
        Derived d;
        naive(&d);
        Base bb = d;    // slices: invokes Base::Base(const Base&), not Derived::Derived(const Derived&)
        // ...
}
```

The variables **b2** and **bb** contain copies of the **Base** part of **d**, that is, a copy of **d.b**. The member **d.d** is not copied. This phenomenon is called *slicing*. It may be exactly what you intended (e.g., see the copy constructor for **D** in §17.5.1.2 where we pass selected information to a base class), but typically it is a subtle bug. If you don't want slicing, you have two major tools to prevent it:

[1] Prohibit copying of the base class: **delete** the copy operations (§17.6.4).
[2] Prevent conversion of a pointer to a derived to a pointer to a base: make the base class a **private** or **protected** base (§20.5).

The former would make the initializations of **b2** and **bb** errors; the latter would make the call of **naive()** and the initialization of **bb** errors.

## 17.5.2 Move

The traditional way of getting a value from **a** to **b** is to copy it. For an integer in a computer's memory, that's just about the only thing that makes sense: that's what the hardware can do with a single instruction. However, from a general and logical point of view that's not so. Consider the obvious implementation of **swap()** exchanging the value of two objects:

```
template<class T>
void swap(T& a, T& b)
{
    const T tmp = a;    // put a copy of a into tmp
    a = b;              // put a copy of b into a
    b = tmp;            // put a copy of tmp into b
};
```

After the initialization of **tmp**, we have two copies of **a**'s value. After the assignment to **tmp**, we have two copies of **b**'s value. After the assignment to **b**, we have two copies of **tmp**'s value (that is, the original value of **a**). Then we destroy **tmp**. That sounds like a lot of work, and it can be. For example:

```
void f(string& s1, string& s2,
    vector<string>& vs1, vector<string>& vs2,
    Matrix& m1, Matrix& m2)
{
    swap(s1,s2);
    swap(vs1.vs2);
    swap(m1,m2);
}
```

What if **s1** has a thousand characters? What if **vs2** has a thousand elements each of a thousand characters? What if **m1** is a 1000∗1000 matrix of **double**s? The cost of copying those data structures could be significant. In fact, the standard-library **swap()** has always been carefully designed to avoid such overhead for **string** and **vector**. That is, effort has been made to avoid copying (taking advantage of the fact that **string** and **vector** objects really are just handles to their elements). Similar work must be done to avoid a serious performance problem for **swap()** of **Matrix**es. If the only operation we have is copy, similar work must be done for huge numbers of functions and data structures that are not part of the standard.

The fundamental problem is that we really didn't want to do any copying at all: we just wanted to exchange pairs of values.

We can also look at the issue of copying from a completely different point of view: we don't usually copy physical things unless we absolutely have to. If you want to borrow my phone, I pass my phone to you rather than making you your own copy. If I lend you my car, I give you a key and

you drive away in my car, rather than in your freshly made copy of my car. Once I have given you an object, you have it and I no longer do. Consequently, we talk about "giving away," "handing over," "transferring ownership of," and "moving" physical objects. Many objects in a computer resemble physical objects (which we don't copy without need and only at considerable cost) more than integer values (which we typically copy because that's easier and cheaper than alternatives). Examples are locks, sockets, file handles, threads, long strings, and large vectors.

To allow the user to avoid the logical and performance problems of copying, C++ directly supports the notion of *moving* as well as the notion of *copying*. In particular, we can define *move constructors* and *move assignments* to move rather than copy their argument. Consider again the simple two-dimensional **Matrix** from §17.5.1:

```
template<class T>
class Matrix {
    std::array<int,2> dim;
    T* elem;  // pointer to sz elements of type T

    Matrix(int d1, int d2) :dim{d1,d2}, elem{new T[d1*d2]} {}
    int size() const { return dim[0]*dim[1]; }

    Matrix(const Matrix&);              // copy constructor
    Matrix(Matrix&&);                   // move constructor

    Matrix& operator=(const Matrix&);  // copy assignment
    Matrix& operator=(Matrix&&);        // move assignment

    ~Matrix(); // destructor
    // ...
};
```

The **&&** indicates an rvalue reference (§7.7.2).

The idea behind a move assignment is to handle lvalues separately from rvalues: copy assignment and copy constructors take lvalues whereas move assignment and move constructors take rvalues. For a **return** value, the move constructor is chosen.

We can define **Matrix**'s move constructor to simply take the representation from its source and replace it with an empty **Matrix** (which is cheap to destroy). For example:

```
template<class T>
Matrix<T>::Matrix(Matrix&& a)       // move constructor
    :dim{a.dim}, elem{a.elem}       // grab a's representation
{
    a.dim = {0,0};                  // clear a's representation
    a.elem = nullptr;
}
```

For the move assignment, we can simply do a swap. The idea behind using a swap to implement a move assignment is that the source is just about to be destroyed, so we can just let the destructor for the source do the necessary cleanup work for us:

```
template<class T>
Matrix<T>& Matrix<T>::operator=(Matrix&& a)          // move assignment
{
    swap(dim,a.dim);              // swap representations
    swap(elem,a.elem);
    return *this;
}
```

Move constructors and move assignments take non-**const** (rvalue) reference arguments: they can, and usually do, write to their argument. However, the argument of a move operation must always be left in a state that the destructor can cope with (and preferably deal with very cheaply and easily).

For resource handles, move operations tend to be significantly simpler and more efficient than copy operations. In particular, move operations typically do not throw exceptions; they don't acquire resources or do complicated operations, so they don't need to. In this, they differ from many copy operations (§17.5).

How does the compiler know when it can use a move operation rather than a copy operation? In a few cases, such as for a return value, the language rules say that it can (because the next action is defined to destroy the element). However, in general we have to tell it by giving an rvalue reference argument. For example:

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

The **move()** is a standard-library function returning an rvalue reference to its argument (§35.5.1): **move(x)** means "give me an rvalue reference to **x**." That is, **std::move(x)** does not move anything; instead, it allows a user to move **x**. It would have been better if **move()** had been called **rval()**, but the name **move()** has been used for this operation for years.

Standard-library containers have move operations (§3.3.2, §35.5.1) and so have other standard-library types, such as **pair** (§5.4.3, §34.2.4.1) and **unique_ptr** (§5.2.1, §34.3.1). Furthermore, operations that insert new elements into standard-library containers, such as **insert()** and **push_back()**, have versions that take rvalue references (§7.7.2). The net result is that the standard containers and algorithms deliver better performance than they would have been able to if they had to copy.

What if we try to swap objects of a type that does not have a move constructor? We copy and pay the price. In general, a programmer is responsible for avoiding excessive copying. It is not the compiler's job to decide what is excessive and what is necessary. To get the copy-to-move optimization for your own data structures, you have to provide move operations (either explicitly or implicitly; see §17.6).

Built-in types, such as **int** and **double***, are considered to have move operations that simply copy. As usual, you have to be careful about data structures containing pointers (§3.3.1). In particular, don't assume that a moved-from pointer is set to **nullptr**.

Having move operations affects the idiom for returning large objects from functions. Consider:

```
Matrix operator+(const Matrix& a, const Matrix& b)
    // res[i][j] = a[i][j]+b[i][j] for each i and j
{
    if (a.dim[0]!=b.dim[0] || a.dim[1]!=b.dim[1])
        throw std::runtime_error("unequal Matrix sizes in +");

    Matrix res{a.dim[0],a.dim[1]};
    constexpr auto n = a.size();
    for (int i = 0; i!=n; ++i)
        res.elem[i] = a.elem[i]+b.elem[i];
    return res;
}
```

**Matrix** has a move constructor so that "return by value" is simple and efficient as well as "natural." Without move operations, we have performance problems and must resort to workarounds. We might have considered:

```
Matrix& operator+(const Matrix& a, const Matrix& b)    // beware!
{
    Matrix& res = *new Matrix;    // allocate on free store
    // res[i][j] = a[i][j]+b[i][j] for each i and j
    return res;
}
```

The use of **new** within **operator+()** is not obvious and forces the user of **+** to deal with tricky memory management issues:

- How does the object created by **new** get **delete**d?
- Do we need a garbage collector?
- Should we use a pool of **Matrix**es rather than the general **new**?
- Do we need use-counted **Matrix** representations?
- Should we redesign the interface of our **Matrix** addition?
- Must the caller of **operator+()** remember to **delete** the result?
- What happens to the newly allocated memory if the computation throws an exception?

None of the alternatives are elegant or general.

## 17.6 Generating Default Operations

Writing conventional operations, such as a copy and a destructor, can be tedious and error-prone, so the compiler can generate them for us as needed. By default, a class provides:

- A default constructor: **X()**
- A copy constructor: **X(const X&)**
- A copy assignment: **X& operator=(const X&)**
- A move constructor: **X(X&&)**
- A move assignment: **X& operator=(X&&)**
- A destructor: **~X()**

By default, the compiler generates each of these operations if a program uses it. However, if the programmer takes control by defining one or more of those operations, the generation of related operations is suppressed:

- If the programmer declares any constructor for a class, the default constructor is not generated for that class.
- If the programmer declares a copy operation, a move operation, or a destructor for a class, no copy operation, move operation, or destructor is generated for that class.

Unfortunately, the second rule is only incompletely enforced: for backward compatibility, copy constructors and copy assignments are generated even if a destructor is defined. However, that generation is deprecated in the ISO standard (§iso.D), and you should expect a modern compiler to warn against it.

If necessary, we can be explicit about which functions are generated (§17.6.1) and which are not (§17.6.4).

## 17.6.1 Explicit Defaults

Since the generation of otherwise default operations can be suppressed, there has to be a way of getting back a default. Also, some people prefer to see a complete list of operations in the program text even if that complete list is not needed. For example, we can write:

```
class gslice {
        valarray<size_t>  size;
        valarray<size_t> stride;
        valarray<size_t> d1;
public:
        gslice() = default;
        ~gslice() = default;
        gslice(const gslice&) = default;
        gslice(gslice&&) = default;
        gslice& operator=(const gslice&) = default;
        gslice& operator=(gslice&&) = default;
        // ...
};
```

This fragment of the implementation of **std::gslice** (§40.5.6) is equivalent to:

```
class gslice {
        valarray<size_t> size;
        valarray<size_t> stride;
        valarray<size_t> d1;
public:
        // ...
};
```

I prefer the latter, but I can see the point of using the former in code bases maintained by less experienced C++ programmers: what you don't see, you might forget about.

Using **=default** is always better than writing your own implementation of the default semantics. Someone assuming that it is better to write something, rather than nothing, might write:

```
class gslice {
        valarray<size_t> size;
        valarray<size_t> stride;
        valarray<size_t> d1;
public:
        // ...
        gslice(const gslice& a);
};

gslice::gslice(const gslice& a)
        : size{a.size },
          stride{a.stride},
          d1{a.d1}
{
}
```

This is not only verbose, making it harder to read the definition of **gslice**, but also opens the opportunity for making mistakes. For example, I might forget to copy one of the members and get it default initialized (rather than copied). Also, when the user provides a function, the compiler no longer knows the semantics of that function and some optimizations become inhibited. For the default operations, those optimizations can be significant.

## 17.6.2 Default Operations

The default meaning of each generated operation, as implemented when the compiler generates it, is to apply the operation to each base and non-**static** data member of the class. That is, we get memberwise copy, memberwise default construction, etc. For example:

```
struct S {
        string a;
        int b;
};

S f(S arg)
{
        S s0 {};    // default construction: {"",0}
        S s1 {s0}; // copy construction
        s1 = arg;  // copy assignment
        return s1; // move construction
}
```

The copy construction of **s1** copies **s0.a** and **s0.b**. The **return** of **s1** moves **s1.a** and **s1.b**, leaving **s1.a** as the empty string and **s1.b** unchanged.

Note that the value of a moved-from object of a built-in type is unchanged. That's the simplest and fastest thing for the compiler to do. If we want something else done for a member of a class, we have to write our move operations for that class.

The default moved-from state is one for which the default destructor and default copy assignment work correctly. It is not guaranteed (or required) that an arbitrary operation on a moved-from object will work correctly. If you need stronger guarantees, write your own operations.

## 17.6.3 Using Default Operations

This section presents a few examples demonstrating how copy, move, and destructors are logically linked. If they were not linked, errors that are obvious when you think about them would not be caught by the compiler.

### 17.6.3.1 Default Constructors

Consider:

```
struct X {
    X(int);     // require an int to initialize an X
};
```

By declaring a constructor that requires an integer argument, the programmer clearly states that a user needs to provide an `int` to initialize an `X`. Had we allowed the default constructor to be generated, that simple rule would have been violated. We have:

```
X a {1};    // OK
X b {};     // error: no default constructor
```

If we also want the default constructor, we can define one or declare that we want the default generated by the compiler. For example:

```
struct Y {
    string s;
    int n;
    Y(const string& s);  // initialize Y with a string
    Y() = default;       // allow default initialization with the default meaning
};
```

The default (i.e., generated) default constructor default constructs each member. Here, `Y()` sets `s` to the empty string. The "default initialization" of a built-in member leaves that member uninitialized. Sigh! Hope for a compiler warning.

### 17.6.3.2 Maintaining Invariants

Often, a class has an invariant. If so, we want copy and move operations to maintain it and the destructor to free any resources involved. Unfortunately, the compiler cannot in every case know what a programmer considers an invariant. Consider a somewhat far-fetched example:

```
struct Z { // invariant:
           // my_favorite is the index of my favorite element of elem
           // largest points to the element with the highest value in elem
    vector<int> elem;
    int my_favorite;
    int* largest;
};
```

The programmer stated an invariant in the comment, but the compiler doesn't read comments. Furthermore, the programmer did not leave a hint about how that invariant is to be established and maintained. In particular, there are no constructors or assignments declared. That invariant is

implicit. The result is that a **Z** can be copied and moved using the default operations:

```
Z v0;                       // no initialization (oops! possibility of undefined values)
Z val {{1,2,3},1,&val[2]};  // OK, but ugly and error-prone
Z v2 = val;                 // copies: v2.largest points into val
Z v3 = move(val);           // moves: val.elem becomes empty; v3.my_favorite is out of range
```

This is a mess. The root problem is that **Z** is badly designed because critical information is "hidden" in a comment or completely missing. The rules for the generation of default operations are heuristic intended to catch common mistakes and to encourage a systematic approach to construction, copy, move, and destruction. Wherever possible

[1] Establish an invariant in a constructor (including possibly resource acquisition).
[2] Maintain the invariant with copy and move operations (with the usual names and types).
[3] Do any needed cleanup in the destructor (incl. possibly resource release).

### 17.6.3.3 Resource Invariants

Many of the most critical and obvious uses of invariants relate to resource management. Consider a simple **Handle**:

```
template<class T> class Handle {
    T* p;
public:
    Handle(T* pp) :p{pp} { }
    T& operator*() { return *p; }
    ~Handle() { delete p; }
};
```

The idea is that you construct a **Handle** given a pointer to an object allocated using **new**. The **Handle** provides access to the object pointed to and eventually **delete**s that object. For example:

```
void f1()
{
    Handle<int> h {new int{99}};
    // ...
}
```

**Handle** declares a constructor that takes an argument: this suppresses the generation of the default constructor. That's good because a default constructor could leave **Handle<T>::p** uninitialized:

```
void f2()
{
    Handle<int> h; // error: no default constructor
    // ...
}
```

The absence of a default constructor saves us from the possibility of a **delete** with a random memory address.

Also, **Handle** declares a destructor: this suppresses the generation of copy and move operations. Again, that saves us from a nasty problem. Consider:

```
void f3()
{
    Handle<int> h1 {new int{7}};
    Handle<int> h2 {h1};          // error: no copy constructor
    // ...
}
```

Had **Handle** had a default copy constructor, both **h1** and **h2** would have had a copy of the pointer and both would have **delete**d it. The results would be undefined and most likely disastrous (§3.3.1). Caveat: the generation of copy operations is only deprecated, not banned, so if you ignore warnings, you might get this example past the compiler. In general, if a class has a pointer member, the default copy and move operations should be considered suspicious. If that pointer member represents ownership, memberwise copy is wrong. If that pointer member does not represent ownership and memberwise copy *is* appropriate, explicit **=default** and a comment are most likely a good idea.

If we wanted copy construction, we could define something like:

```
template<class T>
class Handle {
    // ...
    Handle(const T& a) :p{new T{*a.p}} { }        // clone
};
```

### 17.6.3.4 Partially Specified Invariants

Troublesome examples that rely on invariants but only partially express them through constructors or destructors are rarer but not unheard of. Consider:

```
class Tic_tac_toe {
public:
    Tic_tac_toe(): pos(9) {}    // always 9 positions

    Tic_tac_toe& operator=(const Tic_tac_toe& arg)
    {
        for(int i = 0; i<9; ++i)
            pos.at(i) = arg.pos.at(i);
        return *this;
    }

    // ... other operations ...

    enum State { empty, nought, cross };
private:
    vector<State> pos;
};
```

This was reported to have been part of a real program. It uses the "magic number" **9** to implement a copy assignment that accesses its argument **arg** without checking that the argument actually has nine elements. Also, it explicitly implements the copy assignment, but not the copy constructor. This is not what I consider good code.

We defined copy assignment, so we must also define the destructor. That destructor can be =default because all it needs to do is to ensure that the member pos is destroyed, which is what would have been done anyway had the copy assignment not been defined. At this point, we notice that the user-defined copy assignment is essentially the one we would have gotten by default, so we can =default that also. Add a copy constructor for completeness and we get:

```
class Tic_tac_toe {
public:
    Tic_tac_toe(): pos(9) {}    // always 9 positions
    Tic_tac_toe(const Tic_tac_toe&) = default;
    Tic_tac_toe& operator=(const Tic_tac_toe& arg) = default;
    ~Tic_tac_toe() = default;

    // ... other operations ...

    enum State { empty, nought, cross };
private:
    vector<State> pos;
};
```

Looking at this, we realize that the net effect of these =defaults is just to eliminate move operations. Is that what we want? Probably not. When we made the copy assignment =default, we eliminated the nasty dependence on the magic constant 9. Unless other operations on Tic_tac_toe, not mentioned so far, are also "hardwired with magic numbers," we can safely add move operations. The simplest way to do that is to remove the explicit =defaults, and then we see that Tic_tac_toe is really a perfectly ordinary type:

```
class Tic_tac_toe {
public:
    // ... other operations ...
    enum State { empty, nought, cross };
private:
    vector<State> pos {Vector<State>(9)};        // always 9 positions
};
```

One conclusion that I draw from this and other examples where an "odd combination" of the default operations is defined is that we should be highly suspicious of such types: their irregularity often hides design flaws. For every class, we should ask:

[1] Is a default constructor needed (because the default one is not adequate or has been suppressed by another constructor)?

[2] Is a destructor needed (e.g., because some resource needs to be released)?

[3] Are copy operations needed (because the default copy semantics is not adequate, e.g., because the class is meant to be a base class or because it contains pointers to objects that must be deleted by the class)?

[4] Are move operations needed (because the default semantics is not adequate, e.g., because an empty object doesn't make sense)?

In particular, we should never just consider one of these operations in isolation.

### 17.6.4 deleted Functions

We can "delete" a function; that is, we can state that a function does not exist so that it is an error to try to use it (implicitly or explicitly). The most obvious use is to eliminate otherwise defaulted functions. For example, it is common to want to prevent the copying of classes used as bases because such copying easily leads to slicing (§17.5.1.4):

```
class Base {
        // ...
        Base& operator=(const Base&) = delete;// disallow copying
        Base(const Base&) = delete;

        Base& operator=(Base&&) = delete;           // disallow moving
        Base(Base&&) = delete;
};

Base x1;
Base x2 {x1};   // error: no copy constructor
```

Enabling and disabling copy and move is typically more conveniently done by saying what we want (using **=default**; §17.6.1) rather than saying what we don't want (using **=delete**). However, we can **delete** any function that we can declare. For example, we can eliminate a specialization from the set of possible specializations of a function template:

```
template<class T>
T* clone(T* p)  // return copy of *p
{
        return new T{*p};
};

Foo* clone(Foo*) = delete;          // don't try to clone a Foo

void f(Shape* ps, Foo* pf)
{
        Shape* ps2 = clone(ps);     // fine
        Foo* pf2 = clone(pf);       // error: clone(Foo*) deleted
}
```

Another application is to eliminate an undesired conversion. For example:

```
struct Z {
        // ...
        Z(double);          // can initialize with a double
        Z(int) = delete;    // but not with an integer
};

void f()
{
        Z z1 {1};       // error: Z(int) deleted
        Z z2 {1.0};     // OK
}
```

A further use is to control where a class can be allocated:

```
class Not_on_stack {
    // ...
    ~Not_on_stack() = delete;
};

class Not_on_free_store {
    // ...
    void* operator new(size_t) = delete;
};
```

You can't have a local variable that can't be destroyed (§17.2.2), and you can't allocate an object on the free store when you have =deleted its class's memory allocation operator (§19.2.5). For example:

```
void f()
{
    Not_on_stack v1;              // error: can't destroy
    Not_on_free_store v2;         // OK

    Not_on_stack* p1 = new Not_on_stack;              // OK
    Not_on_free_store* p2 = new Not_on_free_store;    // error: can't allocate
}
```

However, we can never delete that Not_on_stack object. The alternative technique of making the destructor private (§17.2.2) can address that problem.

Note the difference between a =deleted function and one that simply has not been declared. In the former case, the compiler notes that the programmer has tried to use the deleted function and gives an error. In the latter case, the compiler looks for alternatives, such as not invoking a destructor or using a global operator new().

## 17.7 Advice

[1] Design constructors, assignments, and the destructor as a matched set of operations; §17.1.
[2] Use a constructor to establish an invariant for a class; §17.2.1.
[3] If a constructor acquires a resource, its class needs a destructor to release the resource; §17.2.2.
[4] If a class has a virtual function, it needs a virtual destructor; §17.2.5.
[5] If a class does not have a constructor, it can be initialized by memberwise initialization; §17.3.1.
[6] Prefer {} initialization over = and () initialization; §17.3.2.
[7] Give a class a default constructor if and only if there is a "natural" default value; §17.3.3.
[8] If a class is a container, give it an initializer-list constructor; §17.3.4.
[9] Initialize members and bases in their order of declaration; §17.4.1.
[10] If a class has a reference member, it probably needs copy operations (copy constructor and copy assignment); §17.4.1.1.

[11] Prefer member initialization over assignment in a constructor; §17.4.1.1.
[12] Use in-class initializers to provide default values; §17.4.4.
[13] If a class is a resource handle, it probably needs copy and move operations; §17.5.
[14] When writing a copy constructor, be careful to copy every element that needs to be copied (beware of default initializers); §17.5.1.1.
[15] A copy operations should provide equivalence and independence; §17.5.1.3.
[16] Beware of entangled data structures; §17.5.1.3.
[17] Prefer move semantics and copy-on-write to shallow copy; §17.5.1.3.
[18] If a class is used as a base class, protect against slicing; §17.5.1.4.
[19] If a class needs a copy operation or a destructor, it probably needs a constructor, a destructor, a copy assignment, and a copy constructor; §17.6.
[20] If a class has a pointer member, it probably needs a destructor and non-default copy operations; §17.6.3.3.
[21] If a class is a resource handle, it needs a constructor, a destructor, and non-default copy operations; §17.6.3.3.
[22] If a default constructor, assignment, or destructor is appropriate, let the compiler generate it (don't rewrite it yourself); §17.6.
[23] Be explicit about your invariants; use constructors to establish them and assignments to maintain them; §17.6.3.2.
[24] Make sure that copy assignments are safe for self-assignment; §17.5.1.
[25] When adding a new member to a class, check to see if there are user-defined constructors that need to be updated to initialize the member; §17.5.1.