1. （p392）

class Date

{

int d, m, y;

public:

Date(int dd, int mm, int yy); // constructor

// ...

};

When a class has a constructor, all objects of that class will be initialized by a constructor call. If the constructor requires arguments, these arguments must be supplied:

Date today = Date(23,6,1983);

Date xmas(25,12,1990); // abbreviated for m

Date my\_bir thday; // error : initializer missing

Date release1\_0(10,12); // error : third argument missing

Since a constructor defines initialization for a class, we can use the {} -initializer notation:

Date today = Date {23,6,1983};

Date xmas {25,12,1990}; // abbreviated for m

Date release1\_0 {10,12}; // error : third argument missing

I recommend the {} notation over the () notation for initialization because it is explicit about what is being done (initialization), avoids some potential mistakes, and can be used consistently (§2.2.2,§6.3.5). There are cases where () notation must be used (§4.4.1, §17.3.2.1), but they are rare.

1. （p394）默认情况下，应该将单参数的额构造函数声明为explicit。除非你又很好的理由。

如果一个构造函数声明为ecplicit且定义在类外，则在定义中不能重复explicit：

class Data

{

int d, m, y;

public:

exlpicit Data( int dd );

};

Data::Data( int d ) { /\*…\*/ } //正确

explicit Data::Data( int dd ) { /\*…\*/ } //错误

1. （p402）在多线程代码中，static数据成员需要某种锁机制或访问规则来避免竞争条件（见5.3.4节和41.2.4节）。
2. 成员类型（p403）

类型和类型别名也可以用作类的成员。例如：

template<typename T>

class Tree

{

using value\_type = T; // member alias

enum Policy { rb, splay, treeps }; // member enum

class Node // member class

{

Node∗ right;

Node∗ left;

value\_type value;

public:

void f(Tree∗);

};

Node∗ top;

public:

void g(const T&);

// ...

};

成员类（member class，通常也成为嵌套类，nested class）可以引用其所属类的类型和static成员。当给定所属类的一个对象时，只能引用非static成员。为了避免陷入复杂的二叉树结构，我只使用“f()”和“g()”风格的例子。

嵌套类可以访问其所属类的成员（甚至是private成员，这方面与成员函数类似），但他没有当前类对象的概念。例如：

template<typename T>

void Tree::Node::f(Tree∗ p)

{

top = right; // error : no object of type Tree specified

p−>top = right; // OK

value\_type v = left−>value; // OK: value\_type is not associated with an object

}

相反，一个类并没有任何特殊权限能访问其嵌入类的成员。例如:

template<typename T>

void Tree::g(Tree::Node∗ p)

{

value\_type val = right−>value; // error : no object of type Tree::Node

value\_type v = p−>right−>value; // error : Node::r ight is private

p−>f(this); // OK

}

1. （p417）基于构造函数/析构函数的资源管理风格被称为资源获取即初始化（Resource Acquisition Is Initialization）或简称RAII（见5.2节和13.3节）。 如智能指针
2. 基类和成员析构函数（p417）

构造函数会“自顶向下”地创建一个类对象：

1. 首先，构造函数调用其基类的构造函数。
2. 然后，它调用成员的构造函数，
3. 最后，他执行自身的函数体。

析构函数则按相反顺序“拆除”一个对象：

1. 首先，析构函数执行自身的函数体，
2. 然后，它调用其成员的析构函数，
3. 最后，它调用其基类的析构函数。

特别是，一个virtual基类必须在任何可能使用它的基类之前构造，并在他们之后销毁（见21.3.5.1节）。这种顺序保证了一个基类或一个成员不会初始化完成之前或已销毁之后使用。

构造函数按声明顺序（而非初始化器的顺序）执行成员和基类的构造函数：如果两个构函数使用了不同的顺序，析构函数不能保证（即使能保证也会有严重的额外开销）按构造的相反顺序进行销毁。

1. （p418）当对象退出作用域或被delete释放时，析构函数会被隐式调用。显示调用析构函数通常是不必要的，而且会导致严重的错误。但是，在极少数（但很重要的）情况下，我们必须显式调用析构函数。考虑一个容器（如std::vector）维护一个可增长和缩减（例如使用push\_back()和pop\_back()）的内存池。当我们添加一个元素时，容器必须对一个特定地址前。

void C::push\_back(const X& a)

{

// ...

new(p) X{a}; //在地址p调用值a拷贝构造一个X

// ...

}

构造函数的这种用法被称为“放置式new”（ 见11.2.4节）。

相反地，当我们删除一个元素时，容器需要调用其析构函数：

void C::pop\_back()

{

// ...

p−>˜X(); // 销毁地址p中的X

}

语法p−>˜X()对\*p调用X的析构函数。对正常方式销毁的对象（离开其作用域或用delete释放）绝不能使用这种语法。

1. （17.3.2节）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);
2. （17.3.4.1节）

When you have sev eral 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.

1. 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}

{

// ...

}

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 : tr ying to initialize base’s base

// ...

};

1. 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.

1. 17.4.2 Base Initializers

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).

1. 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).

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; // cr yptographic 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"}; // cr yptographic 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"}; // Cr yptographic 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 };

// ...

};

#### 17.5.1 Copy

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 constr uctor

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

Matrix(Matrix&&); // move constr uctor

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

˜Matrix() { delete[] elem; }

// ...

};

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 constr uctor

: dim{m.dim},

elem{new T[m.siz e()]}

{

uninitialized\_copy(m.elem,m.elem+m.siz e(),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.siz e(),elem); // copy elements

}

注：copy 是依次调用重载的运算符=, uninitialized\_copy是依次调用拷贝构造函数。如果目标区间是未初始化的，应该用uninitialized\_copy，否则用copy

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.

#### 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 constr uct

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 .

#### 17.5.2 Move

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.

### 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<siz e\_t> siz e;

valarray<siz e\_t> stride;

valarray<siz e\_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<siz e\_t> siz e;

valarray<siz e\_t> stride;

valarray<siz e\_t> d1;

public:

// ...

};

I prefer the latter, but I can see the point of using the former in code bases maintained by less expe-

rienced 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<siz e\_t> siz e;

valarray<siz e\_t> stride;

valarray<siz e\_t> d1;

public:

// ...

gslice(const gslice& a);

};

gslice::gslice(const gslice& a)

: siz e{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 constr uction

s1 = arg; // copy assignment

return s1; // move constr uction

}

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.

#### 17.6.4 delete d 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 constr uctor

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(siz e\_t) = delete;

};

You can’t hav e 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 =delete d 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 de-

structor 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, giv e 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.