10

Classes

Those types are not "abstract";
they are as real as `int` *and* `float`.
– Doug McIlroy

Concepts and classes — class members — access control — constructors — ***static*** members — default copy — ***const*** member functions — ***this*** — ***struct***s — in-class function definition — concrete classes — member functions and helper functions — overloaded operators — use of concrete classes — destructors — default construction — local variables — user-defined copy — ***new*** and ***delete*** — member objects — arrays — static storage — temporary variables — unions — advice — exercises.

10.1 Introduction [class.intro]

The aim of the C++ class concept is to provide the programmer with a tool for creating new types that can be used as conveniently as the built-in types. In addition, derived classes (Chapter 12) and templates (Chapter 13) provide ways of organizing related classes that allow the programmer to take advantage of their relationships.

A type is a concrete representation of a concept. For example, the C++ built-in type ***float*** with its operations `+`, `-`, `*`, etc., provides a concrete approximation of the mathematical concept of a real number. A class is a user-defined type. We design a new type to provide a definition of a concept that has no direct counterpart among the built-in types. For example, we might provide a type ***Trunk_line*** in a program dealing with telephony, a type ***Explosion*** for a videogame, or a type ***list<Paragraph>*** for a text-processing program. A program that provides types that closely match the concepts of the application tends to be easier to understand and easier to modify than a program that does not. A well-chosen set of user-defined types makes a program more concise. In addition, it makes many sorts of code analysis feasible. In particular, it enables the compiler to detect illegal uses of objects that would otherwise remain undetected until the program is thoroughly tested.

The fundamental idea in defining a new type is to separate the incidental details of the implementation (e.g., the layout of the data used to store an object of the type) from the properties essential to the correct use of it (e.g., the complete list of functions that can access the data). Such a separation is best expressed by channeling all uses of the data structure and internal housekeeping routines through a specific interface.

This chapter focuses on relatively simple ''concrete'' user-defined types that logically don't differ much from built-in types. Ideally, such types should not differ from built-in types in the way they are used, only in the way they are created.

10.2 Classes [class.class]

A *class* is a user-defined type. This section introduces the basic facilities for defining a class, creating objects of a class, and manipulating such objects.

10.2.1 Member Functions [class.member]

Consider implementing the concept of a date using a ***struct*** to define the representation of a ***Date*** and a set of functions for manipulating variables of this type:

```
struct Date {          // representation
    int d, m, y;
};

void init_date(Date& d, int, int, int);     // initialize d
void add_year(Date& d, int n);              // add n years to d
void add_month(Date& d, int n);             // add n months to d
void add_day(Date& d, int n);               // add n days to d
```

There is no explicit connection between the data type and these functions. Such a connection can be established by declaring the functions as members:

```
struct Date {
    int d, m, y;

    void init(int dd, int mm, int yy);   // initialize
    void add_year(int n);                // add n years
    void add_month(int n);               // add n months
    void add_day(int n);                 // add n days
};
```

Functions declared within a class definition (a ***struct*** is a kind of class; §10.2.8) are called member functions and can be invoked only for a specific variable of the appropriate type using the standard syntax for structure member access. For example:

```
Date my_birthday;

void f()
{
    Date today;
```

```
        today.init(16,10,1996);
        my_birthday.init(30,12,1950);

        Date tomorrow = today;
        tomorrow.add_day(1);
        // ...
}
```

Because different structures can have member functions with the same name, we must specify the structure name when defining a member function:

```
void Date::init(int dd, int mm, int yy)
{
        d = dd;
        m = mm;
        y = yy;
}
```

In a member function, member names can be used without explicit reference to an object. In that case, the name refers to that member of the object for which the function was invoked. For example, when ***Date::init()*** is invoked for ***today***, ***m=mm*** assigns to ***today.m***. On the other hand, when ***Date::init()*** is invoked for ***my_birthday***, ***m=mm*** assigns to ***my_birthday.m***. A class member function always ''knows'' for which object it was invoked.

The construct

```
class X { ... };
```

is called a *class definition* because it defines a new type. For historical reasons, a class definition is often referred to as a *class declaration*. Also, like declarations that are not definitions, a class definition can be replicated in different source files using ***#include*** without violating the one-definition rule (§9.2.3).

10.2.2 Access Control [class.access]

The declaration of ***Date*** in the previous subsection provides a set of functions for manipulating a ***Date***. However, it does not specify that those functions should be the only ones to depend directly on ***Date***'s representation and the only ones to directly access objects of class ***Date***. This restriction can be expressed by using a ***class*** instead of a ***struct***:

```
class Date {
        int d, m, y;
public:
        void init(int dd, int mm, int yy);    // initialize

        void add_year(int n);                 // add n years
        void add_month(int n);                // add n months
        void add_day(int n);                  // add n days
};
```

The ***public*** label separates the class body into two parts. The names in the first, *private*, part can be used only by member functions. The second, *public*, part constitutes the public interface to objects

of the class. A ***struct*** is simply a ***class*** whose members are public by default (§10.2.8); member functions can be defined and used exactly as before. For example:

```
inline void Date::add_year(int n)
{
    y += n;
}
```

However, nonmember functions are barred from using private members. For example:

```
void timewarp(Date& d)
{
    d.y -= 200;     // error: Date::y is private
}
```

There are several benefits to be obtained from restricting access to a data structure to an explicitly declared list of functions. For example, any error causing a ***Date*** to take on an illegal value (for example, December 36, 1985) must be caused by code in a member function. This implies that the first stage of debugging – localization – is completed before the program is even run. This is a special case of the general observation that any change to the behavior of the type ***Date*** can and must be effected by changes to its members. In particular, if we change the representation of a class, we need only change the member functions to take advantage of the new representation. User code directly depends only on the public interface and need not be rewritten (although it may need to be recompiled). Another advantage is that a potential user need examine only the definition of the member functions in order to learn to use a class.

The protection of private data relies on restriction of the use of the class member names. It can therefore be circumvented by address manipulation and explicit type conversion. But this, of course, is cheating. C++ protects against accident rather than deliberate circumvention (fraud). Only hardware can protect against malicious use of a general-purpose language, and even that is hard to do in realistic systems.

The ***init()*** function was added partially because it is generally useful to have a function that sets the value of an object and partly because making the data private forces us to provide it.

10.2.3 Constructors [class.ctor]

The use of functions such as ***init()*** to provide initialization for class objects is inelegant and error-prone. Because it is nowhere stated that an object must be initialized, a programmer can forget to do so – or do so twice (often with equally disastrous results). A better approach is to allow the programmer to declare a function with the explicit purpose of initializing objects. Because such a function constructs values of a given type, it is called a *constructor*. A constructor is recognized by having the same name as the class itself. For example:

```
class Date {
    // ...
    Date(int, int, int);        // constructor
};
```

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

```
Date today = Date(23,6,1983);
Date xmas(25,12,1990);          // abbreviated form
Date my_birthday;               // error: initializer missing
Date release1_0(10,12);         // error: 3rd argument missing
```

It is often nice to provide several ways of initializing a class object. This can be done by providing several constructors. For example:

```
class Date {
    int d, m, y;
public:
    // ...
    Date(int, int, int);        // day, month, year
    Date(int, int);             // day, month, today's year
    Date(int);                  // day, today's month and year
    Date();                     // default Date: today
    Date(const char*);          // date in string representation
};
```

Constructors obey the same overloading rules as do other functions (§7.4). As long as the constructors differ sufficiently in their argument types, the compiler can select the correct one for each use:

```
Date today(4);
Date july4("July 4, 1983");
Date guy("5 Nov");
Date now;                       // default initialized as today
```

The proliferation of constructors in the ***Date*** example is typical. When designing a class, a programmer is always tempted to add features just because somebody might want them. It takes more thought to carefully decide what features are really needed and to include only those. However, that extra thought typically leads to smaller and more comprehensible programs. One way of reducing the number of related functions is to use default arguments (§7.5). In the ***Date***, each argument can be given a default value interpreted as ''pick the default: ***today***.''

```
class Date {
    int d, m, y;
public:
    Date(int dd=0, int mm=0, int yy=0);
    // ...
};

Date::Date(int dd, int mm, int yy)
{
    d = dd ? dd : today.d;
    m = mm ? mm : today.m;
    y = yy ? yy : today.y;

    // check that the Date is valid
}
```

When an argument value is used to indicate ''pick the default,'' the value chosen must be outside the set of possible values for the argument. For ***day*** and ***month***, this is clearly so, but for ***year***, zero

may not be an obvious choice. Fortunately, there is no year zero on the European calendar; 1AD (*year==1*) comes immediately after 1BC (*year==-1*).

10.2.4 Static Members [class.static]

The convenience of a default value for ***Date***s was bought at the cost of a significant hidden problem. Our ***Date*** class became dependent on the global variable ***today***. This ***Date*** class can be used only in a context in which ***today*** is defined and correctly used by every piece of code. This is the kind of constraint that causes a class to be useless outside the context in which it was first written. Users get too many unpleasant surprises trying to use such context-dependent classes, and maintenance becomes messy. Maybe ''just one little global variable'' isn't too unmanageable, but that style leads to code that is useless except to its original programmer. It should be avoided.

Fortunately, we can get the convenience without the encumbrance of a publicly accessible global variable. A variable that is part of a class, yet is not part of an object of that class, is called a ***static*** member. There is exactly one copy of a ***static*** member instead of one copy per object, as for ordinary non-***static*** members. Similarly, a function that needs access to members of a class, yet doesn't need to be invoked for a particular object, is called a ***static*** member function.

Here is a redesign that preserves the semantics of default constructor values for ***Date*** without the problems stemming from reliance on a global:

```
class Date {
        int d, m, y;
        static Date default_date;
public:
        Date(int dd=0, int mm=0, int yy=0);
        // ...
        static void set_default(int, int, int);
};
```

We can now define the ***Date*** constructor like this:

```
Date::Date(int dd, int mm, int yy)
{
        d = dd ? dd : default_date.d;
        m = mm ? mm : default_date.m;
        y = yy ? yy : default_date.y;

        // check that the Date is valid
}
```

We can change the default date when appropriate. A static member can be referred to like any other member. In addition, a static member can be referred to without mentioning an object. Instead, its name is qualified by the name of its class. For example:

```
void f()
{
        Date::set_default(4,5,1945);
}
```

Static members – both function and data members – must be defined somewhere. For example:

```
Date Date::default_date(16,12,1770);

void Date::set_default(int d, int m, int y)
{
    Date::default_date = Date(d,m,y);
}
```

Now the default value is Beethoven's birth date – until someone decides otherwise.

Note that ***Date()*** serves as a notation for the value of ***Date::default_date***. For example:

```
Date copy_of_default_date = Date();
```

Consequently, we don't need a separate function for reading the default date.

10.2.5 Copying Class Objects [class.default.copy]

By default, class objects can be copied. In particular, a class object can be initialized with a copy of another object of the same class. This can be done even where constructors have been declared. For example:

```
Date d = today;        // initialization by copy
```

By default, the copy of a class object is a copy of each member. If that default is not the behavior wanted for a class ***X***, a more appropriate behavior can be provided by defining a copy constructor, ***X::X(const X&)***. This is discussed further in §10.4.4.1.

Similarly, class objects can by default be copied by assignment. For example:

```
void f(Date& d)
{
    d = today;
}
```

Again, the default semantics is memberwise copy. If that is not the right choice for a class ***X***, the user can define an appropriate assignment operator (§10.4.4.1).

10.2.6 Constant Member Functions [class.constmem]

The ***Date*** defined so far provides member functions for giving a ***Date*** a value and changing it. Unfortunately, we didn't provide a way of examining the value of a ***Date***. This problem can easily be remedied by adding functions for reading the day, month, and year:

```
class Date {
    int d, m, y;
public:
    int day() const { return d; }
    int month() const { return m; }
    int year() const;
    // ...
};
```

Note the ***const*** after the (empty) argument list in the function declarations. It indicates that these functions do not modify the state of a ***Date***.

Naturally, the compiler will catch accidental attempts to violate this promise. For example:

```
inline int Date::year() const
{
    return y++;    // error: attempt to change member value in const function
}
```

When a ***const*** member function is defined outside its class, the ***const*** suffix is required:

```
inline int Date::year() const        // correct
{
    return y;
}

inline int Date::year()    // error: const missing in member function type
{
    return y;
}
```

In other words, the ***const*** is part of the type of ***Date::day()*** and ***Date::year()***.

A ***const*** member function can be invoked for both ***const*** and non-***const*** objects, whereas a non-***const*** member function can be invoked only for non-***const*** objects. For example:

```
void f(Date& d, const Date& cd)
{
    int i = d.year();      // ok
    d.add_year(1);         // ok

    int j = cd.year();     // ok
    cd.add_year(1);        // error: cannot change value of const cd
}
```

10.2.7 Self-Reference [class.this]

The state update functions ***add_year()***, ***add_month()***, and ***add_day()*** were defined not to return values. For such a set of related update functions, it is often useful to return a reference to the updated object so that the operations can be chained. For example, we would like to write

```
void f(Date& d)
{
    // ...
    d.add_day(1).add_month(1).add_year(1);
    // ...
}
```

to add a day, a month, and a year to ***d***. To do this, each function must be declared to return a reference to a ***Date***:

```
class Date {
    // ...
```

```
        Date& add_year(int n);    // add n years
        Date& add_month(int n);   // add n months
        Date& add_day(int n);     // add n days
};
```

Each (nonstatic) member function knows what object it was invoked for and can explictly refer to it. For example:

```
Date& Date::add_year(int n)
{
        if (d==29 && m==2 && !leapyear(y+n)) { // beware of February 29
                d = 1;
                m = 3;
        }
        y += n;
        return *this;
}
```

The expression ***this*** refers to the object for which a member function is invoked. It is equivalent to Simula's ***THIS*** and Smalltalk's ***self***.

In a nonstatic member function, the keyword ***this*** is a pointer to the object for which the function was invoked. In a non-***const*** member function of class ***X***, the type of ***this*** is ***X *const***. The ***const*** makes it clear that the user is not supposed to change the value of ***this***. In a ***const*** member function of class ***X***, the type of ***this*** is ***const X *const*** to prevent modification of the object itself (see also §5.4.1).

Most uses of ***this*** are implicit. In particular, every reference to a nonstatic member from within a class relies on an implicit use of ***this*** to get the member of the appropriate object. For example, the ***add_year*** function could equivalently, but tediously, have been defined like this:

```
Date& Date::add_year(int n)
{
        if (this->d==29 && this->m==2 && !leapyear(this->y+n)) {
                this->d = 1;
                this->m = 3;
        }
        this->y += n;
        return *this;
}
```

One common explicit use of ***this*** is in linked-list manipulation (e.g., §24.3.7.4).

10.2.7.1 Physical and Logical Constness [class.const]

Occasionally, a member function is logically ***const***, but it still needs to change the value of a member. To a user, the function appears not to change the state of its object. However, some detail that the user cannot directly observe is updated. This is often called *logical constness*. For example, the ***Date*** class might have a function returning a string representation that a user could use for output. Constructing this representation could be a relatively expensive operation. Therefore, it would make sense to keep a copy so that repeated requests would simply return the copy, unless the

Date's value had been changed. Caching values like that is more common for more complicated data structures, but let's see how it can be achieved for a *Date*:

```
class Date {
        bool cache_valid;
        string cache;
        void compute_cache_value();    // fill cache
        // ...
public:
        // ...
        string string_rep() const;        // string representation
};
```

From a user's point of view, *string_rep* doesn't change the state of its *Date*, so it clearly should be a *const* member function. On the other hand, the cache needs to be filled before it can be used. This can be achieved through brute force:

```
string Date::string_rep() const
{
        if (cache_valid == false) {
                Date* th = const_cast<Date*>(this);  // cast away const
                th->compute_cache_value();
                th->cache_valid = true;
        }
        return cache;
}
```

That is, the *const_cast* operator (§15.4.2.1) is used to obtain a pointer of type *Date** to *this*. This is hardly elegant, and it is not guaranteed to work when applied to an object that was originally declared as a *const*. For example:

```
Date d1;
const Date d2;
string s1 = d1.string_rep();
string s2 = d2.string_rep();      // undefined behavior
```

In the case of *d1*, *string_rep*() simply casts back to *d1*'s original type so that the call will work. However, *d2* was defined as a *const* and the implementation could have applied some form of memory protection to ensure that its value wasn't corrupted. Consequently, *d2.string_rep*() is not guaranteed to give a single predictable result on all implementations.

10.2.7.2 Mutable [class.mutable]

The explicit type conversion ''casting away *const*'' and its consequent implementation-dependent behavior can be avoided by declaring the data involved in the cache management to be *mutable*:

```
class Date {
        mutable bool cache_valid;
        mutable string cache;
        void compute_cache_value() const;   // fill (mutable) cache
        // ...
public:
        // ...
        string string_rep() const;          // string representation
};
```

The storage specifier ***mutable*** specifies that a member should be stored in a way that allows updating – even when it is a member of a ***const*** object. In other words, ***mutable*** means ''can never be ***const***.'' This can be used to simplify the definition of ***string_rep()***:

```
string Date::string_rep() const
{
        if (!cache_valid) {
                compute_cache_value();
                cache_valid = true;
        }
        return cache;
}
```

and makes reasonable uses of ***string_rep()*** valid. For example:

```
Date d3;
const Date d4;
string s3 = d3.string_rep();
string s4 = d4.string_rep();      // ok!
```

Declaring members ***mutable*** is most appropriate when (only) part of a representation is allowed to change. If most of an object changes while the object remains logically ***const***, it is often better to place the changing data in a separate object and access it indirectly. If that technique is used, the string-with-cache example becomes:

```
struct cache {
        bool valid;
        string rep;
};

class Date {
        cache* c;                           // initialize in constructor (§10.4.6)
        void compute_cache_value() const;   // fill what cache refers to
        // ...
public:
        // ...
        string string_rep() const;          // string representation
};
```

```
string Date::string_rep() const
{
    if (!c->valid) {
        compute_cache_value();
        c->valid = true;
    }
    return c->rep;
}
```

The programming techniques that support a cache generalize to various forms of lazy evaluation.

10.2.8 Structures and Classes [class.struct]

By definition, a ***struct*** is a class in which members are by default public; that is,

```
struct s { ...
```

is simply shorthand for

```
class s { public: ...
```

The access specifier ***private***: can be used to say that the members following are private, just as ***public***: says that the members following are public. Except for the different names, the following declarations are equivalent:

```
class Date1 {
    int d, m, y;
public:
    Date1(int dd, int mm, int yy);

    void add_year(int n);     // add n years
};

struct Date2 {
private:
    int d, m, y;
public:
    Date2(int dd, int mm, int yy);

    void add_year(int n);     // add n years
};
```

Which style you use depends on circumstances and taste. I usually prefer to use ***struct*** for classes that have all data public. I think of such classes as ''not quite proper types, just data structures.'' Constructors and access functions can be quite useful even for such structures, but as a shorthand rather than guarantors of properties of the type (invariants, see §24.3.7.1).

It is not a requirement to declare data first in a class. In fact, it often makes sense to place data members last to emphasize the functions providing the public user interface. For example:

```
class Date3 {
public:
    Date3(int dd, int mm, int yy);
```

```
        void add_year(int n);      // add n years
    private:
        int d, m, y;
    };
```

In real code, where both the public interface and the implementation details typically are more extensive than in tutorial examples, I usually prefer the style used for ***Date3***.

Access specifiers can be used many times in a single class declaration. For example:

```
    class Date4 {
    public:
        Date4(int dd, int mm, int yy);
    private:
        int d, m, y;
    public:
        void add_year(int n);      // add n years
    };
```

Having more than one public section, as in ***Date4***, tends to be messy. So does having more than one private section. However, allowing many access specifiers in a class is useful for machine-generated code.

10.2.9 In-Class Function Definitions [class.inline]

A member function defined within the class definition – rather than simply declared there – is taken to be an inline member function. That is, in-class definition of member functions is for small, frequently-used functions. Like the class definition it is part of, a member function defined in-class can be replicated in several translation units using ***#include***. Like the class itself, its meaning must be the same wherever it is used (§9.2.3).

The style of placing the definition of data members last in a class can lead to a minor problem with public inline functions that refer to the representation. Consider:

```
    class Date {     // potentially confusing
    public:
        int day() const { return d; }    // return Date::d
        // ...
    private:
        int d, m, y;
    };
```

This is perfectly good C++ code because a member function declared within a class can refer to every member of the class as if the class were completely defined before the member function bodies were considered. However, this can confuse human readers.

Consequently, I usually either place the data first or define the inline member functions after the class itself. For example:

```
class Date {
public:
    int day() const;
    // ...
private:
    int d, m, y;
};

inline int Date::day() const { return d; }
```

10.3 Efficient User-Defined Types [class.concrete]

The previous section discussed bits and pieces of the design of a ***Date*** class in the context of introducing the basic language features for defining classes. Here, I reverse the emphasis and discuss the design of a simple and efficient ***Date*** class and show how the language features support this design.

Small, heavily-used abstractions are common in many applications. Examples are Latin characters, Chinese characters, integers, floating-point numbers, complex numbers, points, pointers, coordinates, transforms, (*pointer,offset*) pairs, dates, times, ranges, links, associations, nodes, (*value,unit*) pairs, disk locations, source code locations, ***BCD*** characters, currencies, lines, rectangles, scaled fixed-point numbers, numbers with fractions, character strings, vectors, and arrays. Every application uses several of these. Often, a few of these simple concrete types are used heavily. A typical application uses a few directly and many more indirectly from libraries.

C++ and other programming languages directly support a few of these abstractions. However, most are not, and cannot be, supported directly because there are too many of them. Furthermore, the designer of a general-purpose programming language cannot foresee the detailed needs of every application. Consequently, mechanisms must be provided for the user to define small concrete types. Such types are called concrete types or concrete classes to distinguish them from abstract classes (§12.3) and classes in class hierarchies (§12.2.4, §12.4).

It was an explicit aim of C++ to support the definition and efficient use of such user-defined data types very well. They are a foundation of elegant programming. As usual, the simple and mundane is statistically far more significant than the complicated and sophisticated.

In this light, let us build a better date class:

```
class Date {
public:          // public interface:
    enum Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

    class Bad_date { };  // exception class

    Date(int dd =0, Month mm =Month(0), int yy =0);  // 0 means ''pick a default''

// functions for examining the Date:
    int day() const;
    Month month() const;
    int year() const;
```

```
        string string_rep() const;          // string representation
        void char_rep(char s[]) const;      // C-style string representation

        static void set_default(int, Month, int);

    // functions for changing the Date:
        Date& add_year(int n);              // add n years
        Date& add_month(int n);             // add n months
        Date& add_day(int n);               // add n days
    private:
        int d, m, y;                        // representation
        static Date default_date;
    };
```

This set of operations is fairly typical for a user-defined type:

[1] A constructor specifying how objects/variables of the type are to be initialized.

[2] A set of functions allowing a user to examine a ***Date***. These functions are marked ***const*** to indicate that they don't modify the state of the object/variable for which they are called.

[3] A set of functions allowing the user to manipulate ***Date***s without actually having to know the details of the representation or fiddle with the intricacies of the semantics.

[4] A set of implicitly defined operations to allow ***Date***s to be freely copied.

[5] A class, ***Bad_date***, to be used for reporting errors as exceptions.

I defined a ***Month*** type to cope with the problem of remembering, for example, whether the 7th of June is written ***Date(6,7)*** (American style) or ***Date(7,6)*** (European style). I also added a mechanism for dealing with default arguments.

I considered introducing separate types ***Day*** and ***Year*** to cope with possible confusion of ***Date(1995,jul,27)*** and ***Date(27,jul,1995)***. However, these types would not be as useful as the ***Month*** type. Almost all such errors are caught at run-time anyway – the 26th of July year 27 is not a common date in my work. How to deal with historical dates before year 1800 or so is a tricky issue best left to expert historians. Furthermore, the day of the month can't be properly checked in isolation from its month and year. See §11.7.1 for a way of defining a convenient ***Year*** type.

The default date must be defined as a valid ***Date*** somewhere. For example:

```
Date Date::default_date(22,jan,1901);
```

I omitted the cache technique from §10.2.7.1 as unnecessary for a type this simple. If needed, it can be added as an implementation detail without affecting the user interface.

Here is a small – and contrived – example of how ***Date***s can be used:

```
void f(Date& d)
{
    Date lvb_day = Date(16,Date::dec,d.year());

    if (d.day()==29 && d.month()==Date::feb) {
        // ...
    }

    if (midnight()) d.add_day(1);

    cout << "day after:" << d+1 << '\n';
}
```

This assumes that the output operator << and the addition operator + have been declared for ***Date***s. I do that in §10.3.3.

Note the ***Date*****::*****feb*** notation. The function ***f*****( )** is not a member of ***Date***, so it must specify that it is referring to ***Date***'s ***feb*** and not to some other entity.

Why is it worthwhile to define a specific type for something as simple as a date? After all, we could define a structure:

```
struct Date {
    int day, month, year;
};
```

and let programmers decide what to do with it. If we did that, though, every user would either have to manipulate the components of ***Date***s directly or provide separate functions for doing so. In effect, the notion of a date would be scattered throughout the system, which would make it hard to understand, document, or change. Inevitably, providing a concept as only a simple structure causes extra work for every user of the structure.

Also, even though the ***Date*** type seems simple, it takes some thought to get right. For example, incrementing a ***Date*** must deal with leap years, with the fact that months are of different lengths, and so on (note: §10.6[1]). Also, the day-month-and-year representation is rather poor for many applications. If we decided to change it, we would need to modify only a designated set of functions. For example, to represent a ***Date*** as the number of days before or after January 1, 1970, we would need to change only ***Date***'s member functions (§10.6[2]).

10.3.1 Member Functions [class.memfct]

Naturally, an implementation for each member function must be provided somewhere. For example, here is the definition of ***Date***'s constructor:

```
Date::Date(int dd, Month mm, int yy)
{
    if (yy == 0) yy = default_date.year();
    if (mm == 0) mm = default_date.month();
    if (dd == 0) dd = default_date.day();

    int max;

    switch (mm) {
    case feb:
        max = 28+leapyear(yy);
        break;
    case apr: case jun: case sep: case nov:
        max = 30;
        break;
    case jan: case mar: case may: case jul: case aug: case oct: case dec:
        max = 31;
        break;
    default:
        throw Bad_date();   // someone cheated
    }
```

```
        if (dd<1 || max<dd) throw Bad_date();

        y = yy;
        m = mm;
        d = dd;
    }
```

The constructor checks that the data supplied denotes a valid *Date*. If not, say for *Date(30,Date::feb,1994)*, it throws an exception (§8.3, Chapter 14), which indicates that something went wrong in a way that cannot be ignored. If the data supplied is acceptable, the obvious initialization is done. Initialization is a relatively complicated operation because it involves data validation. This is fairly typical. On the other hand, once a *Date* has been created, it can be used and copied without further checking. In other words, the constructor establishes the invariant for the class (in this case, that it denotes a valid date). Other member functions can rely on that invariant and must maintain it. This design technique can simplify code immensely (see §24.3.7.1).

I'm using the value *Month(0)* – which doesn't represent a month – to represent ''pick the default month.'' I could have defined an enumerator in *Month* specifically to represent that. But I decided that it was better to use an obviously anomalous value to represent ''pick the default month'' rather than give the appearance that there were 13 months in a year. Note that *0* can be used because it is within the range guaranteed for the enumeration *Month* (§4.8).

I considered factoring out the data validation in a separate function *is_date()*. However, I found the resulting user code more complicated and less robust than code relying on catching the exception. For example, assuming that >> is defined for *Date*:

```
void fill(vector<Date>& aa)
{
    while (cin) {
        Date d;
        try {
            cin >> d;
        }

        catch (Date::Bad_date) {
            // my error handling
            continue;
        }
        aa.push_back(d);    // see §3.7.3
    }
}
```

As is common for such simple concrete types, the definitions of member functions vary between the trivial and the not-too-complicated. For example:

```
inline int Date::day() const
{
    return d;
}
```

```
Date& Date::add_month(int n)
{
    if (n==0) return *this;

    if (n>0) {
        int delta_y = n/12;
        int mm = m+n%12;
        if (12 < mm) {  // note: int(dec)==12
            delta_y++;
            mm -= 12;
        }

        // handle the cases where Month(mm) doesn't have day d

        y += delta_y;
        m = Month(mm);
        return *this;
    }

    // handle negative n

    return *this;
}
```

10.3.2 Helper Functions [class.helper]

Typically, a class has a number of functions associated with it that need not be defined in the class itself because they don't need direct access to the representation. For example:

```
int diff(Date a, Date b); // number of days in the range [a,b) or [b,a)
bool leapyear(int y);
Date next_weekday(Date d);
Date next_saturday(Date d);
```

Defining such functions in the class itself would complicate the class interface and increase the number of functions that would potentially need to be examined when a change to the representation was considered.

How are such functions ''associated'' with class ***Date***? Traditionally, their declarations were simply placed in the same file as the declaration of class ***Date***, and users who needed ***Date***s would make them all available by including the file that defined the interface (§9.2.1). For example:

```
#include "Date.h"
```

In addition to using a specific ***Date.h*** header, or as an alternative, we can make the association explicit by enclosing the class and its helper functions in a namespace (§8.2):

```
namespace Chrono {      // facilities for dealing with time
    class Date { /* ... */};
```

```
    int diff(Date a, Date b);
    bool leapyear(int y);
    Date next_weekday(Date d);
    Date next_saturday(Date d);
    // ...
}
```

The ***Chrono*** namespace would naturally also contain related classes, such as ***Time*** and ***Stopwatch***, and their helper functions. Using a namespace to hold a single class is usually an over-elaboration that leads to inconvenience.

10.3.3 Overloaded Operators [class.over]

It is often useful to add functions to enable conventional notation. For example, the ***operator***== function defines the equality operator == to work for ***Date***s:

```
inline bool operator==(Date a, Date b) // equality
{
    return a.day()==b.day() && a.month()==b.month() && a.year()==b.year();
}
```

Other obvious candidates are:

```
bool operator!=(Date, Date);          // inequality
bool operator<(Date, Date);           // less than
bool operator>(Date, Date);           // greater than
// ...

Date& operator++(Date& d);            // increase Date by one day
Date& operator--(Date& d);            // decrease Date by one day

Date& operator+=(Date& d, int n);     // add n days
Date& operator-=(Date& d, int n);     // subtract n days

Date operator+(Date d, int n);        // add n days
Date operator-(Date d, int n);        // subtract n days

ostream& operator<<(ostream&, Date d); // output d
istream& operator>>(istream&, Date& d); // read into d
```

For ***Date***, these operators can be seen as mere conveniences. However, for many types – such as complex numbers (§11.3), vectors (§3.7.1), and function-like objects (§18.4) – the use of conventional operators is so firmly entrenched in people's minds that their definition is almost mandatory. Operator overloading is discussed in Chapter 11.

10.3.4 The Significance of Concrete Classes [class.significance]

I call simple user-defined types, such as ***Date***, *concrete types* to distinguish them from abstract classes (§2.5.4) and class hierarchies (§12.3) and also to emphasize their similarity to built-in types such as ***int*** and ***char***. They have also been called *value types*, and their use *value-oriented programming*. Their model of use and the ''philosophy'' behind their design are quite different from what is often advertised as object-oriented programming (§2.6.2).

The intent of a concrete type is to do a single, relatively small thing well and efficiently. It is not usually the aim to provide the user with facilities to modify the behavior of a concrete type. In particular, concrete types are not intended to display polymorphic behavior (see §2.5.5, §12.2.6).

If you don't like some detail of a concrete type, you build a new one with the desired behavior. If you want to ''reuse'' a concrete type, you use it in the implementation of your new type exactly as you would have used an ***int***. For example:

```
class Date_and_time {
private:
    Date d;
    Time t;
public:
    Date_and_time(Date d, Time t);
    Date_and_time(int d, Date::Month m, int y, Time t);
    // ...
};
```

The derived class mechanism discussed in Chapter 12 can be used to define new types from a concrete class by describing the desired differences. The definition of ***Vec*** from ***vector*** (§3.7.2) is an example of this.

With a reasonably good compiler, a concrete class such as ***Date*** incurs no hidden overhead in time or space. The size of a concrete type is known at compile time so that objects can be allocated on the run-time stack (that is, without free-store operations). The layout of each object is known at compile time so that inlining of operations is trivially achieved. Similarly, layout compatibility with other languages, such as C and Fortran, comes without special effort.

A good set of such types can provide a foundation for applications. Lack of suitable ''small efficient types'' in an application can lead to gross run-time and space inefficiencies when overly general and expensive classes are used. Alternatively, lack of concrete types can lead to obscure programs and time wasted when each programmer writes code to directly manipulate ''simple and frequently used'' data structures.

10.4 Objects [class.objects]

Objects can be created in several ways. Some are local variables, some are global variables, some are members of classes, etc. This section discusses these alternatives, the rules that govern them, the constructors used to initialize objects, and the destructors used to clean up objects before they become unusable.

10.4.1 Destructors [class.dtor]

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 (§14.4.7). 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 functions are called *destructors*. They typically clean up and release resources. Destructors are called

implicitly when an automatic variable goes out of scope, an object on the free store is deleted, etc. Only in very unusual circumstances does the user need to call a destructor explicitly (§10.4.11).

The most common use of a destructor is to release memory acquired in a constructor. Consider a simple table of elements of some type ***Name***. The constructor for ***Table*** must allocate memory to hold the elements. When the table is somehow deleted, we must ensure that this memory is reclaimed for further use elsewhere. We do this by providing a special function to complement the constructor:

```
class Name {
    const char* s;
    // ...
};

class Table {
    Name* p;
    size_t sz;
public:
    Table(size_t s = 15) { p = new Name[sz = s]; }// constructor

    ~Table() { delete[] p; }                      // destructor

    Name* lookup(const char *);
    bool insert(Name*);
};
```

The destructor notation ***~Table()*** uses the complement symbol ~ to hint at the destructor's relation to the ***Table()*** constructor.

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 ***map***, use a variant of this technique for providing storage for their elements, so the following discussion illustrates techniques you rely on every time you use a standard container (including a standard ***string***). The discussion applies to types without a destructor, also. Such types are seen simply as having a destructor that does nothing.

10.4.2 Default Constructors [class.default]

Similarly, most types can be considered to have a default constructor. A default constructor is a constructor that can be called without supplying an argument. Because of the default argument ***15***, ***Table::Table(size_t)*** is a default constructor. If a user has declared a default constructor, that one will be used; otherwise, the compiler will try to generate one if needed and if the user hasn't declared other constructors. A compiler-generated default constructor implicitly calls the default constructors for a class' members of class type and bases (§12.2.2). For example:

```
struct Tables {
    int i;
    int vi[10];
    Table t1;
    Table vt[10];
};
```

```
Tables tt;
```

Here, *tt* will be initialized using a generated default constructor that calls *Table(15)* for *tt.t1* and each element of *tt.vt*. On the other hand, *tt.i* and the elements of *tt.vi* are not initialized because those objects are not of a class type. The reasons for the dissimilar treatment of classes and built-in types are C compatibility and fear of run-time overhead.

Because ***const***s and references must be initialized (§5.5, §5.4), a class containing ***const*** or reference members cannot be default-constructed unless the programmer explicitly supplies a constructor (§10.4.6.1). For example:

```
struct X {
    const int a;
    const int& r;
};

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

Default constructors can be invoked explicitly (§10.4.10). Built-in types also have default constructors (§6.2.8).

10.4.3 Construction and Destruction [class.ctor.dtor]

Consider the different ways an object can be created and how it gets destroyed afterwards. An object can be created as:

- §10.4.4 A named automatic object, which is created each time its declaration is encountered in the execution of the program and destroyed each time the program exits the block in which it occurs
- §10.4.5 A free-store object, which is created using the ***new*** operator and destroyed using the ***delete*** operator
- §10.4.6 A nonstatic member object, which is created as a member of another class object and created and destroyed when the object of which it is a member is created and destroyed
- §10.4.7 An array element, which is created and destroyed when the array of which it is an element is created and destroyed
- §10.4.8 A local static object, which is created the first time its declaration is encountered in the execution of the program and destroyed once at the termination of the program
- §10.4.9 A global, namespace, or class static object, which is created once ''at the start of the program'' and destroyed once at the termination of the program
- §10.4.10 A temporary object, which is created as part of the evaluation of an expression and destroyed at the end of the full expression in which it occurs
- §10.4.11 An object placed in memory obtained from a user-supplied function guided by arguments supplied in the allocation operation
- §10.4.12 A ***union*** member, which may not have a constructor or a destructor

This list is roughly sorted in order of importance. The following subsections explain these various ways of creating objects and their uses.

10.4.4 Local Variables [class.local]

The constructor for a local variable is executed each time the thread of control passes through the declaration of the local variable. The destructor for a local variable is executed each time the local variable's block is exited. Destructors for local variables are executed in reverse order of their construction. For example:

```
void f(int i)
{
    Table aa;
    Table bb;
    if (i>0) {
        Table cc;
        // ...
    }
    Table dd;
    // ...
}
```

Here, ***aa***, ***bb***, and ***dd*** are constructed (in that order) each time ***f*( )** is called, and ***dd***, ***bb***, and ***aa*** are destroyed (in that order) each time we return from ***f*( )**. If ***i>0*** for a call, ***cc*** will be constructed after ***bb*** and destroyed before ***dd*** is constructed.

10.4.4.1 Copying Objects [class.copy]

If *t1* and *t2* are objects of a class ***Table***, *t2=t1* by default means a memberwise copy of *t1* into *t2* (§10.2.5). Having assignment interpreted this way can cause a surprising (and usually undesired) effect when used on objects of a class with pointer members. Memberwise copy is usually the wrong semantics for copying objects containing resources managed by a constructor/destructor pair. For example:

```
void h( )
{
    Table t1;
    Table t2 = t1;   // copy initialization: trouble
    Table t3;

    t3 = t2;         // copy assignment: trouble
}
```

Here, the ***Table*** default constructor is called twice: once each for *t1* and *t3*. It is not called for *t2* because that variable was initialized by copying. However, the ***Table*** destructor is called three times: once each for *t1*, *t2*, and *t3*! The default interpretation of assignment is memberwise copy, so *t1*, *t2*, and *t3* will, at the end of ***h*( )**, each contain a pointer to the array of names allocated on the free store when *t1* was created. No pointer to the array of names allocated when *t3* was created remains because it was overwritten by the *t3=t2* assignment. Thus, in the absence of automatic garbage collection (§10.4.5), its storage will be lost to the program forever. On the other hand, the array created for *t1* appears in *t1*, *t2*, and *t3*, so it will be deleted thrice. The result of that is undefined and probably disastrous.

Such anomalies can be avoided by defining what it means to copy a ***Table***:

```
class Table {
        // ...
        Table(const Table&);                    // copy constructor
        Table& operator=(const Table&);         // copy assignment
};
```

The programmer can define any suitable meaning for these copy operations, but the traditional one for this kind of container is to copy the contained elements (or at least to give the user of the container the appearance that a copy has been done; see §11.12). For example:

```
Table::Table(const Table& t)            // copy constructor
{
        p = new Name[sz=t.sz];
        for (int i = 0; i<sz; i++) p[i] = t.p[i];
}

Table& Table::operator=(const Table& t)         // assignment
{
        if (this != &t) {       // beware of self-assignment: t = t
                delete[] p;
                p = new Name[sz=t.sz];
                for (int i = 0; i<sz; i++) p[i] = t.p[i];
        }
        return *this;
}
```

As is almost always the case, the copy constructor and the copy assignment differ considerably. The fundamental reason is that a copy constructor initializes uninitialized memory, whereas the copy assignment operator must correctly deal with a well-constructed object.

Assignment can be optimized in some cases, but the general strategy for an assignment operator is simple: protect against self-assignment, delete old elements, initialize, and copy in new elements. Usually every nonstatic member must be copied (§10.4.6.3).

10.4.5 Free Store [class.free]

An object created on the free store has its constructor invoked by the ***new*** operator and exists until the ***delete*** operator is applied to a pointer to it. Consider:

```
int main()
{
        Table* p = new Table;
        Table* q = new Table;

        delete p;
        delete p; // probably causes run-time error
}
```

The constructor ***Table::Table()*** is called twice. So is the destructor ***Table::~Table()***. Unfortunately, the ***new***s and the ***delete***s in this example don't match, so the object pointed to by ***p*** is

deleted twice and the object pointed to by ***q*** not at all. Not deleting an object is typically not an error as far as the language is concerned; it is only a waste of space. However, in a program that is meant to run for a long time, such a memory leak is a serious and hard-to-find error. There are tools available for detecting such leaks. Deleting ***p*** twice is a serious error; the behavior is undefined and most likely disastrous.

Some C++ implementations automatically recycle the storage occupied by unreachable objects (garbage collecting implementations), but their behavior is not standardized. Even when a garbage collector is running, ***delete*** will invoke a destructor if one is defined, so it is still a serious error to delete an object twice. In many cases, that is only a minor inconvenience. In particular, where a garbage collector is known to exist, destructors that do memory management only can be eliminated. This simplification comes at the cost of portability and for some programs, a possible increase in run time and a loss of predictability of run-time behavior (§C.9.1).

After ***delete*** has been applied to an object, it is an error to access that object in any way. Unfortunately, implementations cannot reliably detect such errors.

The user can specify how ***new*** does allocation and how ***delete*** does deallocation (see §6.2.6.2 and §15.6). It is also possible to specify the way an allocation, initialization (construction), and exceptions interact (see §14.4.5 and §19.4.5). Arrays on the free store are discussed in §10.4.7.

10.4.6 Class Objects as Members [class.m]

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

```
class Club {
        string name;
        Table members;
        Table 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 initializers are preceded by a colon and the individual member initializers are separated by commas.

The members' constructors are called before the body of the containing class' own constructor is executed. The constructors are called in the order in which they are declared in the class rather than the order in which they appear in the initializer list. To avoid confusion, it is best to specify the initializers in declaration order. The member destructors are called in the reverse order of construction.

If a member constructor needs no arguments, the member need not be mentioned in the member initializer list, so

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

is equivalent to the previous version. In each case, ***Club::officers*** is constructed by ***Table::Table*** with the default argument ***15***.

When a class object containing class objects is destroyed, the body of that object's own destructor (if one is specified) is executed first and then the members' destructors are executed in reverse order of declaration. A constructor assembles the execution environment for the member functions for a class from the bottom up (members first). The destructor disassembles it from the top down (members last).

10.4.6.1 Necessary Member Initialization [class.ref.init]

Member initializers are essential for types for which initialization differs from assignment – that is, for member objects of classes without default constructors, for ***const*** members, and for reference members. For example:

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

There isn't any other way to initialize such members, and it is an error not to initialize objects of those types. For most types, however, 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, thus making explicit the fact that initialization is being done. Often, there also is an efficiency advantage to using the initializer syntax. 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.

10.4.6.2 Member Constants [class.memconst]

It is also possible to initialize a static integral constant member by adding a *constant-expression* initializer to its member declaration. For example:

```
class Curious {
public:
    static const int c1 = 7;          // ok, but remember definition
    static int c2 = 11;               // error: not const
    const int c3 = 13;                // error: not static
    static const int c4 = f(17);      // error: in-class initializer not constant
    static const float c5 = 7.0;      // error: in-class not integral
    // ...
};
```

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;            // necessary, but don't repeat initializer here

const int* p = &Curious::c1;      // ok: Curious::c1 has been defined
```

Alternatively, you can use an enumerator (§4.8, §14.4.6, §15.3) as a symbolic constant within a class declaration. For example:

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

In that way, you are not tempted to initialize variables, floating-point numbers, etc. within a class.

10.4.6.3 Copying Members [class.mem.copy]

A default copy constructor or default copy assignment (§10.4.4.1) copies all elements of a class. If this copy cannot be done, it is an error to try to copy an object of such a class. For example:

```
class Unique_handle {
private:        // copy operations are private to prevent copying (§11.2.2)
    Unique_handle(const Unique_handle&);
    Unique_handle& operator=(const Unique_handle&);
public:
    // ...
};

struct Y {
    // ...
    Unique_handle a;    // requires explicit initialization
};
```

```
Y y1;
Y y2 = y1;      // error: cannot copy Y::a
```

In addition, a default assignment cannot be generated if a nonstatic member is a reference, a ***const***, or a user-defined type without a copy assignment.

Note that the default copy constructor leaves a reference member referring to the same object in both the original and the copied object. This can be a problem if the object referred to is supposed to be deleted.

When writing a copy constructor, we must take care to copy every element that needs to be copied. By default, elements are default-initialized, but that is often not what is desired in a copy constructor. For example:

```
Person::Person(const Person& a) : name(a.name) { }    // beware!
```

Here, I forgot to copy the ***address***, so ***address*** is initialized to the empty string by default. When adding a new member to a class, always check if there are user-defined constructors that need to be updated in order to initialize and copy the new member.

10.4.7 Arrays [class.array]

If an object of a class can be constructed without supplying an explicit initializer, then arrays of that class can be defined. For example:

```
Table tbl[10];
```

This will create an array of ***10 Table***s and initialize each ***Table*** by a call of ***Table::Table()*** with the default argument ***15***.

There is no way to specify explicit arguments for a constructor in an array declaration. If you absolutely must initialize members of an array with different values, you can write a default constructor that directly or indirectly reads and writes nonlocal data. For example:

```
class Ibuffer {
      string buf;
public:
      Ibuffer() { cin>>buf; }
      // ...
};

void f()
{
      Ibuffer words[100]; // each word initialized from cin
      // ...
}
```

It is usually best to avoid such subtleties.

The destructor for each constructed element of an array is invoked when that array is destroyed. This is done implicitly for arrays that are not allocated using ***new***. Like C, C++ doesn't distinguish between a pointer to an individual object and a pointer to the initial element of an array (§5.3). Consequently, the programmer must state whether an array or an individual object is being deleted. For example:

```
void f(int sz)
{
    Table* t1 = new Table;
    Table* t2 = new Table[sz];
    Table* t3 = new Table;
    Table* t4 = new Table[sz];

    delete t1;      // right
    delete[] t2;    // right
    delete[] t3;    // wrong: trouble
    delete t4;      // wrong: trouble
}
```

Exactly how arrays and individual objects are allocated is implementation-dependent. Therefore, different implementations will react differently to incorrect uses of the ***delete*** and ***delete[]*** operators. In simple and uninteresting cases like the previous one, a compiler can detect the problem, but generally something nasty will happen at run time.

The special destruction operator for arrays, ***delete[]***, isn't logically necessary. However, suppose the implementation of the free store had been required to hold sufficient information for every object to tell if it was an individual or an array. The user could have been relieved of a burden, but that obligation would have imposed significant time and space overheads on some C++ implementations.

As always, if you find C-style arrays too cumbersome, use a class such as ***vector*** (§3.7.1, §16.3) instead. For example:

```
void g()
{
    vector<Table>* p1 = new vector<Table>(10);
    Table* p2 = new Table;

    delete p1;
    delete p2;
}
```

10.4.8 Local Static Store [class.obj.static]

The constructor for a local static object (§7.1.2) is called the first time the thread of control passes through the object's definition. Consider this:

```
void f(int i)
{
    static Table tbl;
    // ...
    if (i) {
        static Table tbl2;
        // ...
    }
}
```

```
int main()
{
    f(0);
    f(1);
    f(2);
    // ...
}
```

Here, the constructor is called for *tbl* once the first time *f*() is called. Because *tbl* is declared *static*, it does not get destroyed on return from *f*() and it does not get constructed a second time when *f*() is called again. Because the block containing the declaration of *tbl2* doesn't get executed for the call *f*(0), *tbl2* doesn't get constructed until the call *f*(1). It does not get constructed again when its block is entered a second time.

The destructors for local static objects are invoked in the reverse order of their construction when the program terminates (§9.4.1.1). Exactly when is unspecified.

10.4.9 Nonlocal Store [class.global]

A variable defined outside any function (that is, global, namespace, and class *static* variables) is initialized (constructed) before *main*() is invoked, and any such variable that has been constructed will have its destructor invoked after exit from *main*(). Dynamic linking complicates this picture slightly by delaying the initialization until the code is linked into the running program.

Constructors for nonlocal objects in a translation unit are executed in the order their definitions occur. Consider:

```
class X {
    // ...
    static Table memtbl;
};

Table tbl;

Table X::memtbl;

namespace Z {
    Table tbl2;
}
```

The order of construction is *tbl*, then *X*::*memtbl*, and then *Z*::*tbl2*. Note that a declaration (as opposed to a definition), such as the declaration of *memtbl* in *X*, doesn't affect the order of construction. The destructors are called in the reverse order of construction: *Z*::*tbl2*, then *X*::*memtbl*, and then *tbl*.

No implementation-independent guarantees are made about the order of construction of nonlocal objects in different compilation units. For example:

```
// file1.c:
    Table tbl1;

// file2.c:
    Table tbl2;
```

Whether *tbl1* is constructed before *tbl2* or vice versa is implementation-dependent. The order isn't even guaranteed to be fixed in every particular implementation. Dynamic linking, or even a small change in the compilation process, can alter the sequence. The order of destruction is similarly implementation-dependent.

Sometimes when you design a library, it is necessary, or simply convenient, to invent a type with a constructor and a destructor with the sole purpose of initialization and cleanup. Such a type would be used once only: to allocate a static object so that the constructor and the destructor are called. For example:

```
class Zlib_init {
    Zlib_init();      // get Zlib ready for use
    ~Zlib_init();     // clean up after Zlib
};

class Zlib {
    static Zlib_init x;
    // ...
};
```

Unfortunately, it is not guaranteed that such an object is initialized before its first use and destroyed after its last use in a program consisting of separately compiled units. A particular C++ implementation may provide such a guarantee, but most don't. A programmer may ensure proper initialization by implementing the strategy that the implementations usually employ for local static objects: a first-time switch. For example:

```
class Zlib {
    static bool initialized;
    static void initialize() { /* initialize */ initialized = true; }
public:
    // no constructor

    void f()
    {
        if (initialized == false) initialize();
        // ...
    }
    // ...
};
```

If there are many functions that need to test the first-time switch, this can be tedious, but it is often manageable. This technique relies on the fact that statically allocated objects without constructors are initialized to *0*. The really difficult case is the one in which the first operation may be time-critical so that the overhead of testing and possible initialization can be serious. In that case, further trickery is required (§21.5.2).

An alternative approach for a simple object is to present it as a function (§9.4.1):

```
int& obj() { static int x = 0; return x; }  // initialized upon first use
```

First-time switches do not handle every conceivable situation. For example, it is possible to create objects that refer to each other during construction. Such examples are best avoided. If such

objects are necessary, they must be constructed carefully in stages. Also, there is no similarly simple last-time switch construct. Instead, see §9.4.1.1 and §21.5.2.

10.4.10 Temporary Objects [class.temp]

Temporary objects most often are the result of arithmetic expressions. For example, at some point in the evaluation of *x*y+z* the partial result *x*y* must exist somewhere. Except when performance is the issue (§11.6), temporary objects rarely become the concern of the programmer. However, it happens (§11.6, §22.4.7).

Unless bound to a reference or used to initialize a named object, a temporary object is destroyed at the end of the full expression in which it was created. A *full expression* is an expression that is not a subexpression of some other expression.

The standard ***string*** class has a member function ***c_str()*** that returns a C-style, zero-terminated array of characters (§3.5.1, §20.4.1). Also, the operator + is defined to mean string concatenation. These are very useful facilities for ***strings***. However, in combination they can cause obscure problems. For example:

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

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

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

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

Please note that in this case, as in many others, the problems with temporaries arose from using a high-level data type in a low-level way. A cleaner programming style would have not only yielded a more understandable program fragment, but also avoided the problems with temporaries completely. For example:

```
void f(string& s1, string& s2, string& s3)
{
    cout << s1+s2;
    string s = s2+s3;

    if (s.length()<8 && s[0]=='a') {
        // use s here
    }
}
```

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

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

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

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

This is fine. The temporary is destroyed when ''its'' reference or named object go out of scope. Remember that returning a reference to a local variable is an error (§7.3) and that a temporary object cannot be bound to a non-***const*** reference (§5.5).

A temporary object can also be created by explicitly invoking a constructor. For example:

```
void f(Shape& s, int x, int y)
{
    s.move(Point(x,y));      // construct Point to pass to Shape::move()
    // ...
}
```

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

10.4.11 Placement of Objects [class.placement]

Operator ***new*** creates its object on the free store by default. What if we wanted the object allocated elsewhere? Consider a simple class:

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

We can place objects anywhere by providing an allocator function with extra arguments and then supplying such extra arguments when using ***new***:

```
void* operator new(size_t, void* p) { return p; }    // explicit placement operator

void* buf = reinterpret_cast<void*>(0xF00F);    // significant address
X* p2 = new(buf)X;  // construct an X at 'buf;' invokes: operator new(sizeof(X),buf)
```

Because of this usage, the ***new(buf)X*** syntax for supplying extra arguments to ***operator new()*** is known as the *placement syntax*. Note that every ***operator new()*** takes a size as its first argument and that the size of the object allocated is implicitly supplied (§15.6). The ***operator new()*** used by the ***new*** operator is chosen by the usual argument matching rules (§7.4); every ***operator new()*** has a ***size_t*** as its first argument.

The ''placement'' ***operator new()*** is the simplest such allocator. It is defined in the standard header ***<new>***.

The ***reinterpret_cast*** is the crudest and potentially nastiest of the type conversion operators (§6.2.7). In most cases, it simply yields a value with the same bit pattern as its argument with the type required. Thus, it can be used for the inherently implementation-dependent, dangerous, and occasionally absolutely necessary activity of converting integer values to pointers and vice versa.

The placement ***new*** construct can also be used to allocate memory from a specific arena:

```
class Arena {
public:
        virtual void* alloc(size_t) =0;
        virtual void free(void*) =0;
        // ...
};

void* operator new(size_t sz, Arena* a)
{
        return a->alloc(sz);
}
```

Now objects of arbitrary types can be allocated from different ***Arena***s as needed. For example:

```
extern Arena* Persistent;
extern Arena* Shared;

void g(int i)
{
        X* p = new(Persistent) X(i);    // X in persistent storage
        X* q = new(Shared) X(i);        // X in shared memory
        // ...
}
```

Placing an object in an area that is not (directly) controlled by the standard free-store manager implies that some care is required when destroying the object. The basic mechanism for that is an explicit call of a destructor:

```
void destroy(X* p, Arena* a)
{
        p->~X();        // call destructor
        a->free(p);     // free memory
}
```

Note that explicit calls of destructors, like the use of special-purpose *global* allocators, should be avoided wherever possible. Occasionally, they are essential. For example, it would be hard to implement an efficient general container along the lines of the standard library ***vector*** (§3.7.1, §16.3.8) without using explicit destructor class. However, a novice should think thrice before

calling a destructor explicitly and also should ask a more experienced colleague before doing so.

See §14.4.7 for an explanation of how placement new interacts with exception handling.

There is no special syntax for placement of arrays. Nor need there be, since arbitrary types can be allocated by placement new. However, a special ***operator delete*( )** can be defined for arrays (§19.4.5).

10.4.12 Unions [class.union]

A named union is defined as a ***struct***, where every member has the same address (see §C.8.2). A union can have member functions but not static members.

In general, a compiler cannot know what member of a union is used; that is, the type of the object stored in a union is unknown. Consequently, a union may not have members with constructors or destructors. It wouldn't be possible to protect that object against corruption or to guarantee that the right destructor is called when the union goes out of scope.

Unions are best used in low-level code, or as part of the implementation of classes that keep track of what is stored in the union (see §10.6[20]).

10.5 Advice [class.advice]

[1] Represent concepts as classes; §10.1.

[2] Use public data (***struct***s) only when it really is just data and no invariant is meaningful for the data members; §10.2.8.

[3] A concrete type is the simplest kind of class. Where applicable, prefer a concrete type over more complicated classes and over plain data structures; §10.3.

[4] Make a function a member only if it needs direct access to the representation of a class; §10.3.2.

[5] Use a namespace to make the association between a class and its helper functions explicit; §10.3.2.

[6] Make a member function that doesn't modify the value of its object a ***const*** member function; §10.2.6.

[7] Make a function that needs access to the representation of a class but needn't be called for a specific object a ***static*** member function; §10.2.4.

[8] Use a constructor to establish an invariant for a class; §10.3.1.

[9] If a constructor acquires a resource, its class needs a destructor to release the resource; §10.4.1.

[10] If a class has a pointer member, it needs copy operations (copy constructor and copy assignment); §10.4.4.1.

[11] If a class has a reference member, it probably needs copy operations (copy constructor and copy assignment); §10.4.6.3.

[12] If a class needs a copy operation or a destructor, it probably needs a constructor, a destructor, a copy assignment, and a copy constructor; §10.4.4.1.

[13] Check for self-assignment in copy assignments; §10.4.4.1.

[14] When writing a copy constructor, be careful to copy every element that needs to be copied (beware of default initializers); §10.4.4.1.

[15] When adding a new member to a class, always check to see if there are user-defined constructors that need to be updated to initialize the member; §10.4.6.3.
[16] Use enumerators when you need to define integer constants in class declarations; §10.4.6.1.
[17] Avoid order dependencies when constructing global and namespace objects; §10.4.9.
[18] Use first-time switches to minimize order dependencies; §10.4.9.
[19] Remember that temporary objects are destroyed at the end of the full expression in which they are created; §10.4.10.

10.6 Exercises [class.exercises]

1. (∗1) Find the error in ***Date*::*add_year*()** in §10.2.2. Then find two additional errors in the version in §10.2.7.
2. (∗2.5) Complete and test ***Date***. Reimplement it with ''number of days after 1/1/1970'' representation.
3. (∗2) Find a ***Date*** class that is in commercial use. Critique the facilities it offers. If possible, then discuss that ***Date*** with a real user.
4. (∗1) How do you access ***set_default*** from class ***Date*** from namespace ***Chrono*** (§10.3.2)? Give at least three different ways.
5. (∗2) Define a class ***Histogram*** that keeps count of numbers in some intervals specified as arguments to ***Histogram***'s constructor. Provide functions to print out the histogram. Handle out-of-range values.
6. (∗2) Define some classes for providing random numbers of certain distributions (for example, uniform and exponential). Each class has a constructor specifying parameters for the distribution and a function ***draw*** that returns the next value.
7. (∗2.5) Complete class ***Table*** to hold (name,value) pairs. Then modify the desk calculator program from §6.1 to use class ***Table*** instead of ***map***. Compare and contrast the two versions.
8. (∗2) Rewrite ***Tnode*** from §7.10[7] as a class with constructors, destructors, etc. Define a tree of ***Tnode***s as a class with constructors, destructors, etc.
9. (∗3) Define, implement, and test a set of integers, class ***Intset***. Provide union, intersection, and symmetric difference operations.
10. (∗1.5) Modify class ***Intset*** into a set of nodes, where ***Node*** is a structure you define.
11. (∗3) Define a class for analyzing, storing, evaluating, and printing simple arithmetic expressions consisting of integer constants and the operators `+`, `-`, `*`, and `/`. The public interface should look like this:

```
class Expr {
        // ...
public:
        Expr(char*);
        int eval();
        void print();
};
```

The string argument for the constructor ***Expr*::*Expr*()** is the expression. The function ***Expr*::*eval*()** returns the value of the expression, and ***Expr*::*print*()** prints a representation

of the expression on ***cout***. A program might look like this:

```
Expr x("123/4+123*4-3");
cout << "x = " << x.eval() << "\n";
x.print();
```

Define class ***Expr*** twice: once using a linked list of nodes as the representation and once using a character string as the representation. Experiment with different ways of printing the expression: fully parenthesized, postfix notation, assembly code, etc.

12. (∗2) Define a class ***Char_queue*** so that the public interface does not depend on the representation. Implement ***Char_queue*** (a) as a linked list and (b) as a vector. Do not worry about concurrency.
13. (∗3) Design a symbol table class and a symbol table entry class for some language. Have a look at a compiler for that language to see what the symbol table really looks like.
14. (∗2) Modify the expression class from §10.6[11] to handle variables and the assignment operator =. Use the symbol table class from §10.6[13].
15. (∗1) Given this program:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

modify it to produce this output:

```
Initialize
Hello, world!
Clean up
```

Do not change ***main()*** in any way.

16. (∗2) Define a ***Calculator*** class for which the calculator functions from §6.1 provide most of the implementation. Create ***Calculator***s and invoke them for input from ***cin***, from command-line arguments, and for strings in the program. Allow output to be delivered to a variety of targets similar to the way input can be obtained from a variety of sources.
17. (∗2) Define two classes, each with a ***static*** member, so that the construction of each ***static*** member involves a reference to the other. Where might such constructs appear in real code? How can these classes be modified to eliminate the order dependence in the constructors?
18. (∗2.5) Compare class ***Date*** (§10.3) with your solution to §5.9[13] and §7.10[19]. Discuss errors found and likely differences in maintenance of the two solutions.
19. (∗3) Write a function that, given an ***istream*** and a ***vector<string>***, produces a ***map<string,vector<int>>*** holding each string and the numbers of the lines on which the string appears. Run the program on a text-file with no fewer than 1,000 lines looking for no fewer than 10 words.
20. (∗2) Take class ***Entry*** from §C.8.2 and modify it so that each union member is always used according to its type.

.