16

Classes

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

- Introduction
- Class Basics
  Member Functions; Default Copying; Access Control; **class** and **struct**; Constructors; **explicit** Constructors; In-Class Initializers; In-Class Function Definitions; Mutability; Self-Reference; Member Access; **static** Members; Member Types
- Concrete Classes
  Member Functions; Helper Functions; Overloaded Operators; The Significance of Concrete Classes
- Advice

16.1 Introduction

C++ classes are a tool for creating new types that can be used as conveniently as the built-in types. In addition, derived classes (§3.2.4, Chapter 20) and templates (§3.4, Chapter 23) allow the programmer to express (hierachical and parametric) relationships among classes and to take advantage of such relationships.

A type is a concrete representation of a concept (an idea, a notion, etc.). 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 video game, 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, easier to reason about, and easier to modify than a program that does not. A well-chosen set of user-defined types

also 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 be found only through exhaustive testing.

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 its 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:

§16.2 Class Basics introduces the basic facilities for defining a class and its members.

§16.3 Concrete Classes discusses the design of elegant and efficient concrete classes.

The following chapters go into greater detail and presents abstract classes and class hierarchies:

Chapter 17 *Construction, Cleanup, Copy, and Move* presents the variety of ways to control initialization of objects of a class, how to copy and move objects, and how to provide "cleanup actions" to be performed when an object is destroyed (e.g., goes out of scope).

Chapter 18 *Operator Overloading* explains how to define unary and binary operators (such as **+**, *, and **!**) for user-defined types and how to use them.

Chapter 19 *Special Operators* considers how to define and use operators (such as **[]**, **()**, **->**, **new**) that are "special" in that they are commonly used in ways that differ from arithmetic and logical operators. In particular, this chapter shows how to define a string class.

Chapter 20 *Derived Classes* introduces the basic language features supporting object-oriented programming. Base and derived classes, virtual functions, and access control are covered.

Chapter 21 *Class Hierarchies* focuses on the use of base and derived classes to effectively organize code around the notion of class hierarchies. Most of this chapter is devoted to discussion of programming techniques, but technical aspects of multiple inheritance (classes with more than one base class) are also covered.

Chapter 22 *Run-Time Type Information* describes the techniques for explicitly navigating class hierarchies. In particular, the type conversion operations **dynamic_cast** and **static_cast** are presented, as is the operation for determining the type of an object given one of its base classes (**typeid**).

16.2 Class Basics

Here is a very brief summary of classes:

- A class is a user-defined type.
- A class consists of a set of members. The most common kinds of members are data members and member functions.
- Member functions can define the meaning of initialization (creation), copy, move, and cleanup (destruction).

- Members are accessed using `.` (dot) for objects and `->` (arrow) for pointers.
- Operators, such as `+`, `!`, and `[]`, can be defined for a class.
- A class is a namespace containing its members.
- The `public` members provide the class's interface and the `private` members provide implementation details.
- A `struct` is a `class` where members are by default `public`.

For example:

```
class X {
private:                    // the representation (implementation) is private
        int m;
public:                     // the user interface is public
        X(int i =0) :m{i} { }  // a constructor (initialize the data member m)

        int mf(int i)       // a member function
        {
                int old = m;
                m = i;      // set a new value
                return old; // return the old value
        }
};

X var {7}; // a variable of type X, initialized to 7

int user(X var, X* ptr)
{
        int x = var.mf(7);      // access using . (dot)
        int y = ptr->mf(9);     // access using -> (arrow)
        int z = var.m;          // error: cannot access private member
}
```

The following sections expand on this and give rationale. The style is tutorial: a gradual development of ideas, with details postponed until later.

16.2.1 Member Functions

Consider implementing the concept of a date using a `struct` (§2.3.1, §8.2) 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, `Date`, 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; §16.2.4) 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 (§8.2). 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 "knows" for which object it was invoked. But see §16.2.12 for the notion of a **static** member.

16.2.2 Default Copying

By default, objects can be copied. In particular, a class object can be initialized with a copy of an object of its class. For example:

```
Date d1 = my_birthday;   // initialization by copy
Date d2 {my_birthday};   // 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 (§3.3, §17.5).

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

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

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 (§3.3, §17.5).

16.2.3 Access Control

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 (§16.2.4); member functions can be defined and used exactly as before. For example:

```
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
}
```

The **init()** function is now essential because making the data private forces us to provide a way of initializing members. For example:

```
Date dx;
dx.m = 3;               // error: m is private
dx.init(25,3,2011);     // OK
```

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, 2016) 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 definitions of the member functions in order to learn to use a class. A more subtle, but most significant, advantage is that focusing on the design of a good interface simply leads to better code because thoughts and time otherwise devoted to debugging are expended on concerns related to proper use.

The protection of private data relies on restriction of the use of the class member names. It can therefore be circumvented by address manipulation (§7.4.1) and explicit type conversion (§11.5). But this, of course, is cheating. C++ protects against accident rather than deliberate circumvention (fraud). Only hardware can offer perfect protection against malicious use of a general-purpose language, and even that is hard to do in realistic systems.

16.2.4 **class** and **struct**

The construct

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

is called a *class definition*; it defines a type called **X**. 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 (§15.2.3).

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: /* ... */ };
```

These two definitions of **S** are interchangeable, though it is usually wise to stick to one style. Which style you use depends on circumstances and taste. I tend to use **struct** for classes that I think of as "just simple data structures." If I think of a class as "a proper type with an invariant," I use **class**. Constructors and access functions can be quite useful even for **struct**s, but as a shorthand rather than guarantors of invariants (§2.4.3.2, §13.4).

By default, members of a **class** are private:

```
class Date1 {
        int d, m, y;            // private by default
public:
        Date1(int dd, int mm, int yy);
        void add_year(int n);       // add n years
};
```

However, we can also use the access specifier **private:** to say that the members following are private, just as **public:** says that the members following are public:

```
struct Date2 {
private:
        int d, m, y;
public:
        Date2(int dd, int mm, int yy);
        void add_year(int n);       // add n years
};
```

Except for the different name, **Date1** and **Date2** are equivalent.

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, though, and might affect the object layout (§20.5). So does having more than one private section. However, allowing many access specifiers in a class is useful for machine-generated code.

16.2.5 Constructors

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 {
        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 form
Date my_birthday;               // 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 form
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.

By providing several constructors, we can provide a variety of ways of initializing objects of a type. 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 ordinary functions (§12.3). As long as the constructors differ sufficiently in their argument types, the compiler can select the correct one for a use:

```
Date today {4};                 // 4, today.m, today.y
Date july4 {"July 4, 1983"};
Date guy {5,11};                // 5, November, today.y
Date now;                       // default initialized as today
Date start {};                  // 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 (§12.2.5). For **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**).

Alternatively, we could use the default values directly as default arguments:

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

Date::Date(int dd, int mm, int yy)
{
        // check that the Date is valid
}
```

However, I chose to use **0** to avoid building actual values into **Date**'s interface. That way, we have the option to later improve the implementation of the default.

Note that by guaranteeing proper initialization of objects, the constructors greatly simplify the implementation of member functions. Given constructors, other member functions no longer have to deal with the possibility of uninitialized data (§16.3.1).

16.2.6 explicit Constructors

By default, a constructor invoked by a single argument acts as an implicit conversion from its argument type to its type. For example:

```
complex<double> d {1};        // d=={1,0} (§5.6.2)
```

Such implicit conversions can be extremely useful. Complex numbers are an example: if we leave out the imaginary part, we get a complex number on the real axis. That's exactly what mathematics requires. However, in many cases, such conversions can be a significant source of confusion and errors. Consider **Date**:

```
void my_fct(Date d);

void f()
{
    Date d {15};    // plausible: x becomes {15,today.m,today.y}
    // ...
    my_fct(15);     // obscure
    d = 15;         // obscure
    // ...
}
```

At best, this is obscure. There is no clear logical connection between the number **15** and a **Date** independently of the intricacies of our code.

Fortunately, we can specify that a constructor is not used as an *implicit* conversion. A constructor declared with the keyword **explicit** can only be used for initialization and explicit conversions. For example:

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

Date d1 {15};           // OK: considered explicit
Date d2 = Date{15};     // OK: explicit
Date d3 = {15};         // error: = initialization does not do implicit conversions
Date d4 = 15;           // error: = initialization does not do implicit conversions

void f()
{
    my_fct(15);         // error: argument passing does not do implicit conversions
    my_fct({15});       // error: argument passing does not do implicit conversions
    my_fct(Date{15});   // OK: explicit
    // ...
}
```

An initialization with an **=** is considered a *copy initialization*. In principle, a copy of the initializer is placed into the initialized object. However, such a copy may be optimized away (elided), and a move operation (§3.3.2, §17.5.2) may be used if the initializer is an rvalue (§6.4.1). Leaving out the **=** makes the initialization explicit. Explicit initialization is known as *direct initialization*.

By default, declare a constructor that can be called with a single argument **explicit**. You need a good reason not to do so (as for **complex**). If you define an implicit constructor, it is best to document your reason or a maintainer may suspect that you were forgetful (or ignorant).

If a constructor is declared **explicit** and defined outside the class, that **explicit** cannot be repeated:

```
class Date {
        int d, m, y;
public:
        explicit Date(int dd);
        // ...
};

Date::Date(int dd) { /* ... */ }            // OK
explicit Date::Date(int dd) { /* ... */ }   // error
```

Most examples where **explicit** is important involve a single constructor argument. However, **explicit** can also be useful for constructors with zero or more than one argument. For example:

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

X x1 = {};          // error: implicit
X x2 = {1,2};       // error: implicit

X x3 {};            // OK: explicit
X x4 {1,2};         // OK: explicit

int f(X);

int i1 = f({});     // error: implicit
int i2 = f({1,2});  // error: implicit

int i3 = f(X{});    // OK: explicit
int i4 = f(X{1,2}); // OK: explicit
```

The distinction between direct and copy initialization is maintained for list initialization (§17.3.4.3).

16.2.7 In-Class Initializers

When we use several constructors, member initialization can become repetitive. 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
        // ...
};
```

We can deal with that by introducing default arguments to reduce the number of constructors (§16.2.5). Alternatively, we can add initializers to data members:

```
class Date {
        int d {today.d};
        int m {today.m};
        int y {today.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
        // ...
```

Now, each constructor has the **d**, **m**, and **y** initialized unless it does it itself. For example:

```
Date::Date(int dd)
        :d{dd}
{
        // check that the Date is valid
}
```

This is equivalent to:

```
Date::Date(int dd)
        :d{dd}, m{today.m}, y{today.y}
{
        // check that the Date is valid
}
```

16.2.8 In-Class Function Definitions

A member function defined within the class definition – rather than simply declared there – is taken to be an inline (§12.1.5) member function. That is, in-class definition of member functions is for small, rarely modified, 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, the member function's meaning must be the same wherever it is **#include**d (§15.2.3).

A member can refer to another member of its class independently of where that member is defined (§6.3.4). Consider:

```
class Date {
public:
        void add_month(int n) { m+=n; }     // increment the Date's m
        // ...
private:
        int d, m, y;
};
```

That is, function and data member declarations are order independent. I could equivalently have written:

```
class Date {
public:
        void add_month(int n) { m+=n; }      // increment the Date's m
        // ...
private:
        int d, m, y;
};

inline void Date::add_month(int n)  // add n months
{
        m+=n;      // increment the Date's m
}
```

This latter style is often used to keep class definitions simple and easy to read. It also provides a textual separation of a class's interface and implementation.

Obviously, I simplified the definition of `Date::add_month`; just adding `n` and hoping to hit a good date is too naive (§16.3.1).

16.2.9 Mutability

We can define a named object as a constant or as a variable. In other words, a name can refer to an object that holds an *immutable* or a *mutable* value. Since the precise terminology can be a bit clumsy, we end up referring to some variables as being constant or briefer still to `const` variables. However odd that may sound to a native English speaker, the concept is useful and deeply embedded in the C++ type system. Systematic use of immutable objects leads to more comprehensible code, to more errors being found early, and sometimes to improved performance. In particular, immutability is a most useful property in a multi-threaded program (§5.3, Chapter 41).

To be useful beyond the definition of simple constants of built-in types, we must be able to define functions that operate on `const` objects of user-defined types. For freestanding functions that means functions that take `const T&` arguments. For classes it means that we must be able to define member functions that work on `const` objects.

16.2.9.1 Constant Member Functions

The `Date` as defined so far provides member functions for giving a `Date` a value. 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;

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

The **const** after the (empty) argument list in the function declarations 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:

```
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:

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

In other words, **const** is part of the type of **Date::day()**, **Date::month()**, 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 a const Date
}
```

16.2.9.2 Physical and Logical Constness

Occasionally, a member function is logically **const**, but it still needs to change the value of a member. That is, to a user, the function appears not to change the state of its object, but 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. 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 {
public:
    // ...
    string string_rep() const;      // string representation
private:
    bool cache_valid;
    string cache;
    void compute_cache_value(); // fill cache
    // ...
};
```

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** and **cache_valid** members must change occasionally for the design to make sense.

Such problems could be solved through brute force using a cast, for example, a **const_cast** (§11.5.2). However, there are also reasonably elegant solutions that do not involve messing with type rules.

16.2.9.3 mutable

We can define a member of a class to be **mutable**, meaning that it can be modified even in a **const** object:

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

Now we can define **string_rep()** in the obvious way:

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

We can now use **string_rep()** for both **const** and non-**const** objects. For example:

```
void f(Date d, const Date cd)
{
        string s1 = d.string_rep();
        string s2 = cd.string_rep();            // OK!
        // ...
}
```

16.2.9.4 Mutability through Indirection

Declaring a member **mutable** is most appropriate when only a small part of a representation of a small object is allowed to change. More complicated cases are often better handled by placing the changing data in a separate object and accessing it indirectly. If that technique is used, the string-with-cache example becomes:

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

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

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.

Note that **const** does not apply (transitively) to objects accessed through pointers or references. The human reader may consider such an object as "a kind of subobject," but the compiler does not know such pointers or references to be any different from any others. That is, a member pointer does not have any special semantics that distinguish it from other pointers.

16.2.10 Self-Reference

The state update functions **add_year()**, **add_month()**, and **add_day()** (§16.2.3) 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 (non-**static**) member function knows for which object it was invoked and can explicitly 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.

In a non-**static** 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***. However, **this** is considered an rvalue, so it is not possible to take the address of **this** or to assign to **this**. In a **const** member function of class **X**, the type of **this** is **const X*** to prevent modification of the object itself (see also §7.5).

Most uses of **this** are implicit. In particular, every reference to a non-**static** 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. For example:

```
struct Link {
        Link* pre;
        Link* suc;
        int data;

        Link* insert(int x)    // insert x before this
        {
                return pre = new Link{pre,this,x};
        }
```

```
        void remove()  // remove and destroy this
        {
                if (pre) pre->suc = suc;
                if (suc) suc->pre = pre;
                delete this;
        }

        // ...
};
```

Explicit use of **this** is required for access to members of base classes from a derived class that is a template (§26.3.7).

16.2.11 Member Access

A member of a class **X** can be accessed by applying the **.** (dot) operator to an object of class **X** or by applying the **->** (arrow) operator to a pointer to an object of class **X**. For example:

```
struct X {
        void f();
        int m;
};

void user(X x, X* px)
{
        m = 1;          // error: there is no m in scope
        x.m = 1;        // OK
        x->m = 1;       // error: x is not a pointer
        px->m = 1;      // OK
        px.m = 1;       // error: px is a pointer
}
```

Obviously, there is a bit of redundancy here: the compiler knows whether a name refers to an **X** or to an **X***, so a single operator would have been sufficient. However, a programmer might be confused, so from the first days of C the rule has been to use separate operators.

From inside a class no operator is needed. For example:

```
void X::f()
{
        m = 1;          // OK: "this->m = 1;" (§16.2.10)
}
```

That is, an unqualified member name acts as if it had been prefixed by **this->**. Note that a member function can refer to the name of a member before it has been declared:

```
struct X {
        int f() { return m; }   // fine: return this X's m
        int m;
};
```

If we want to refer to a member in general, rather than to a member of a particular object, we qualify by the class name followed by **::**. For example:

```
struct S {
        int m;
        int f();
        static int sm;
};

int X::f() { return m; }          // X's f
int X::sm {7};                    // X's static member sm (§16.2.12)
int (S::*) pmf() {&S::f};         // X's member f
```

That last construct (a pointer to member) is fairly rare and esoteric; see §20.6. I mention it here just to emphasize the generality of the rule for **::**.

16.2.12 [static] Members

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 (§6.4.2). 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 dd, int mm, int yy); // set default_date to Date(dd,mm,yy)
};
```

We can now define the **Date** constructor to use **default_date** 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 ...
}
```

Using **set_default()**, 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);    // call Date's static member set_default()
}
```

If used, a **static** member – a function or data member – must be defined somewhere. The keyword **static** is not repeated in the definition of a **static** member. For example:

```
Date Date::default_date {16,12,1770};          // definition of Date::default_date

void Date::set_default(int d, int m, int y)    // definition of Date::set_default
{
    default_date = {d,m,y};                    // assign new value to default_date
}
```

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{};

void f(Date);

void g()
{
    f(Date{});
}
```

Consequently, we don't need a separate function for reading the default date. Furthermore, where the target type is unambiguously a **Date**, plain **{}** is sufficient. For example:

```
void f1(Date);

void f2(Date);
void f2(int);

void g()
{
    f1({});          // OK: equivalent to f1(Date{})
    f2({}):          // error: ambiguous: f2(int) or f2(Date)?
    f2(Date{});      // OK
}
```

In multi-threaded code, **static** data members require some kind of locking or access discipline to avoid race conditions (§5.3.4, §41.2.4). Since multi-threading is now very common, it is unfortunate that use of **static** data members was quite popular in older code. Older code tends to use **static** members in ways that imply race conditions.

16.2.13 Member Types

Types and type aliases can be members of a class. For example:

```
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&);
        // ...
};
```

A *member class* (often called a *nested class*) can refer to types and **static** members of its enclosing class. It can only refer to non-**static** members when it is given an object of the enclosing class to refer to. To avoid getting into the intricacies of binary trees, I use purely technical "**f()** and **g()**"-style examples.

A nested class has access to members of its enclosing class, even to **private** members (just as a member function has), but has no notion of a current object of the enclosing class. For example:

```
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
}
```

A class does not have any special access rights to the members of its nested class. For example:

```
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::right is private
        p->f(this);                             // OK
}
```

Member classes are more a notational convenience than a feature of fundamental importance. On the other hand, member aliases are important as the basis of generic programming techniques relying on associated types (§28.2.4, §33.1.3). Member **enum**s are often an alternative to **enum class**es when it comes to avoiding polluting an enclosing scope with the names of enumerators (§8.4.1).

16.3 Concrete Classes

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, currency values, 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++ directly supports a few of these abstractions as built-in types. However, most are not, and cannot be, directly supported by the language 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 (§20.4) and classes in class hierarchies (§20.3, §21.2).

A class is called *concrete* (or *a concrete class*) if its representation is part of its definition. This distinguishes it from abstract classes (§3.2.2, §20.4) which provide an interface to a variety of implementations. Having the representation available allows us:

- To place objects on the stack, in statically allocated memory, and in other objects
- To copy and move objects (§3.3, §17.5)
- To refer directly to named objects (as opposed to accessing through pointers and references)

This makes concrete classes simple to reason about and easy for the compiler to generate optimal code for. Thus, we prefer concrete classes for small, frequently used, and performance-critical types, such as complex numbers (§5.6.2), smart pointers (§5.2.1), and containers (§4.4).

It was an early explicit aim of C++ to support the definition and efficient use of such user-defined 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:

```
namespace Chrono {

    enum class Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

    class Date {
    public:          // public interface:
        class Bad_date { };  // exception class

        explicit Date(int dd ={}, Month mm ={}, int yy ={});          // {} means "pick a default"
```

```
        // nonmodifying 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[], in max) const;      // C-style string representation

        // (modifying) 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:
            bool is_valid();                            // check if this Date represents a date
            int d, m, y;                                // representation
        };

        bool is_date(int d, Month m, int y);            // true for valid date
        bool is_leapyear(int y);                        // true if y is a leap year

        bool operator==(const Date& a, const Date& b);
        bool operator!=(const Date& a, const Date& b);

        const Date& default_date();                     // the default date

        ostream& operator<<(ostream& os, const Date& d);    // print d to os
        istream& operator>>(istream& is, Date& d);          // read Date from is into d
    } // Chrono
```

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 (§16.2.5).

[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 modify **Date**s without actually having to know the details of the representation or fiddle with the intricacies of the semantics.

[4] Implicitly defined operations that allow **Date**s to be freely copied (§16.2.2).

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

[6] A set of useful helper functions. The helper functions are not members and have no direct access to the representation of a **Date**, but they are identified as related by the use of the namespace **Chrono**.

I defined a **Month** type to cope with the problem of remembering the month/day order, for example, to avoid confusion about whether the 7th of June is written **{6,7}** (American style) or **{7,6}** (European style).

I considered introducing separate types **Day** and **Year** to cope with possible confusion of **Date{1995,Month::jul,27}** and **Date{27,Month::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. Dealing 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.

To save the user from having to explicitly mention year and month even when they are implied by context, I added a mechanism for providing a default. Note that for **Month** the **{}** gives the (default) value **0** just as for integers even though it is not a valid **Month** (§8.4). However, in this case, that's exactly what we want: an otherwise illegal value to represent "pick the default." Providing a default (e.g., a default value for **Date** objects) is a tricky design problem. For some types, there is a conventional default (e.g., **0** for integers); for others, no default makes sense; and finally, there are some types (such as **Date**) where the question of whether to provide a default is nontrivial. In such cases, it is best – at least initially – not to provide a default value. I provide one for **Date** primarily to be able to discuss how to do so.

I omitted the cache technique from §16.2.9 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 {16,Month::dec,d.year()};

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

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

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

    Date dd;  // initialized to the default date
    cin>>dd;
    if (dd==d) cout << "Hurray!\n";
}
```

This assumes that the addition operator, **+**, has been declared for **Date**s. I do that in §16.3.3.

Note the use of explicit qualification of **dec** and **feb** by **Month**. I used an **enum class** (§8.4.1) specifically to be able to use short names for the months, yet also ensure that their use would not be obscure or ambiguous.

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

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

Each programmer could then 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. 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.

To simplify, I decided to eliminate the notion of changing the default date. Doing so eliminates some opportunities for confusion and the likelihood of race conditions in a multi-threaded program (§5.3.1). I seriously considered eliminating the notion of a default date altogether. That would have forced users to consistently explicitly initialize their **Date**s. However, that can be inconvenient and surprising, and more importantly common interfaces used for generic code require default construction (§17.3.3). That means that I, as the designer of **Date**, have to pick the default date. I chose January 1, 1970, because that is the starting point for the C and C++ standard-library time routines (§35.2, §43.6). Obviously, eliminating **set_default_date()** caused some loss of generality of **Date**. However, design – including class design – is about making decisions, rather than just deciding to postpone them or to leave all options open for users.

To preserve an opportunity for future refinement, I declared **default_date()** as a helper function:

```
const Date& Chrono::default_date();
```

That doesn't say anything about how the default date is actually set.

16.3.1 Member Functions

Naturally, an implementation for each member function must be provided somewhere. For example:

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

    if (!is_valid()) throw Bad_date();
}
```

The constructor checks that the data supplied denotes a valid **Date**. If not, say, for **{30,Month::feb,1994}**, it throws an exception (§2.4.3.1, Chapter 13), which indicates that something went wrong. 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 §2.4.3.2, §13.4).

I'm using the value **Month{}** – which doesn't represent a month and has the integer value **0** – 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 `Month{}`, meaning `0`, can be used because it is within the range guaranteed for the enumeration `Month` (§8.4).

I use the member initializer syntax (§17.4) to initialize the members. After that, I check for `0` and modify the values as needed. This clearly does not provide optimal performance in the (hopefully rare) case of an error, but the use of member initializers leaves the structure of the code obvious. This makes the style less error-prone and easier to maintain than alternatives. Had I aimed at optimal performance, I would have used three separate constructors rather than a single constructor with default arguments.

I considered making the validation function `is_valid()` public. However, I found the resulting user code more complicated and less robust than code relying on catching the exception:

```
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 §4.4.2
    }
}
```

However, checking that a `{d,m,y}` set of values is a valid date is not a computation that depends on the representation of a `Date`, so I implemented `is_valid()` in terms of a helper function:

```
bool Date::is_valid()
{
    return is_date(d,m,y);
}
```

Why have both `is_valid()` and `is_date()`? In this simple example, we could manage with just one, but I can imagine systems where `is_date()` (as here) checks that a `(d,m,y)`-tuple represents a valid date and where `is_valid()` does an additional check on whether that date can be reasonably represented. For example, `is_valid()` might reject dates from before the modern calendar became commonly used.

As is common for such simple concrete types, the definitions of `Date`'s 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;                     // number of whole years
        int mm = static_cast<int>(m)+n%12;      // number of months ahead
        if (12 < mm) {                          // note: dec is represented by 12
            ++delta_y;
            mm -= 12;
        }

        // ... handle the cases where the month mm doesn't have day d ...

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

    // ... handle negative n ...

    return *this;
}
```

I wouldn't call the code for `add_month()` pretty. In fact, if I added all the details, it might even approach the complexity of relatively simple real-world code. This points to a problem: adding a month is conceptually simple, so why is our code getting complicated? In this case, the reason is that the `d,m,y` representation isn't as convenient for the computer as it is for us. A better representation (for many purposes) would be simply a number of days since a defined "day zero" (e.g., January 1, 1970). That would make computation on `Date`s simple at the expense of complexity in providing output fit for humans.

Note that assignment and copy initialization are provided by default (§16.2.2). Also, `Date` doesn't need a destructor because a `Date` owns no resources and requires no cleanup when it goes out of scope (§3.2.1.2).

16.3.2 Helper Functions

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 is_leapyear(int y);
bool is_date(int d, Month m, int y);

const Date& default_date();
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**? In early C++, as in C, their declarations were simply placed in the same file as the declaration of class **Date**. Users who needed **Date**s would make them all available by including the file that defined the interface (§15.2.2). For example:

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

In addition (or alternatively), we can make the association explicit by enclosing the class and its helper functions in a namespace (§14.3.1):

```
namespace Chrono {          // facilities for dealing with time

    class Date { /* ... */};

    int diff(Date a, Date b);
    bool is_leapyear(int y);
    bool is_date(int d, Month m, int y);
    const Date& default_date();
    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 overelaboration that leads to inconvenience.

Naturally, the helper function must be defined somewhere:

```
bool Chrono::is_date(int d, Month m, int y)
{
    int ndays;

    switch (m) {
    case Month::feb:
        ndays = 28+is_leapyear(y);
        break;
    case Month::apr: case Month::jun: case Month::sep: case Month::nov:
        ndays = 30;
        break;
    case Month::jan: case Month::mar: case Month::may: case Month::jul:
    case Month::aug: case Month::oct: case Month::dec:
        ndays = 31;
        break;
    default:
        return false;
    }

    return 1<=d && d<=ndays;
}
```

I'm deliberately being a bit paranoid here. A **Month** shouldn't be outside the **jan** to **dec** range, but it is possible (someone might have been sloppy with a cast), so I check.

The troublesome **default_date** finally becomes:

```
const Date& Chrono::default_date()
{
        static Date d {1,Month::jan,1970};
        return d;
}
```

16.3.3 Overloaded Operators

It is often useful to add functions to enable conventional notation. For example, **operator==()** 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) { return d.add_day(1); }          // increase Date by one day
Date& operator--(Date& d) { return d.add_day(-1); }         // decrease Date by one day

Date& operator+=(Date& d, int n) { return d.add_day(n); }   // add n days
Date& operator-=(Date& d, int n) { return d.add_day(-n); }  // subtract n days

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

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

These operators are defined in **Chrono** together with **Date** to avoid overload problems and to benefit from argument-dependent lookup (§14.2.4).

For **Date**, these operators can be seen as mere conveniences. However, for many types – such as complex numbers (§18.3), vectors (§4.4.1), and function-like objects (§3.4.3, §19.2.2) – 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 18.

For **Date**, I was tempted to provide **+=** and **-=** as member functions instead of **add_day()**. Had I done so, I would have followed a common idiom (§3.2.1.1).

Note that assignment and copy initialization are provided by default (§16.3, §17.3.3).

16.3.4 The Significance of Concrete Classes

I call simple user-defined types, such as **Date**, *concrete types* to distinguish them from abstract classes (§3.2.2) and class hierarchies (§20.4), and also to emphasize their similarity to built-in types such as **int** and **char**. Concrete classes are used just like built-in types. Concrete types 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 called object-oriented programming (§3.2.4, Chapter 21).

The intent of a concrete type is to do a single, relatively simple 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 run-time polymorphic behavior (see §3.2.3, §20.3.2).

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);
        // ...
};
```

Alternatively, the derived class mechanism discussed in Chapter 20 can be used to define new types from a concrete class by describing the desired differences. The definition of **Vec** from **vector** (§4.4.1.2) is an example of this. However, derivation from a concrete class should be done with care and only rarely because of the lack of virtual functions and run-time type information (§17.5.1.4, Chapter 22).

With a reasonably good compiler, a concrete class such as **Date** incurs no hidden overhead in time or space. In particular, no indirection through pointers is necessary for access to objects of concrete classes, and no "housekeeping" data is stored in objects of concrete classes. 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 an 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. In particular, they can be used to make interfaces more specific and less error-prone. For example:

```
Month do_something(Date d);
```

This is far less likely to be misunderstood or misused than:

```
int do_something(int d);
```

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 represented as

simple aggregates of built-in types. Alternatively, 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.

16.4 Advice

[1] Represent concepts as classes; §16.1.
[2] Separate the interface of a class from its implementation; §16.1.
[3] Use public data (**struct**s) only when it really is just data and no invariant is meaningful for the data members; §16.2.4.
[4] Define a constructor to handle initialization of objects; §16.2.5.
[5] By default declare single-argument constructors **explicit**; §16.2.6.
[6] Declare a member function that does not modify the state of its object **const**; §16.2.9.
[7] A concrete type is the simplest kind of class. Where applicable, prefer a concrete type over more complicated classes and over plain data structures; §16.3.
[8] Make a function a member only if it needs direct access to the representation of a class; §16.3.2.
[9] Use a namespace to make the association between a class and its helper functions explicit; §16.3.2.
[10] Make a member function that doesn’t modify the value of its object a **const** member function; §16.2.9.1.
[11] 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; §16.2.12.