We can now try to use our newly defined variables:

```
last.add_day(1);
add_day(2);                 // error: what date?
```

Note that the member function **add_day()** is called for a particular **Date** using the dot member-access notation. We'll show how to define member functions in §8.4.4.

In C++98, people used parentheses to delimit the initializer list, so you will see a lot of code like this:

```
Date last(2000,12,31);      // OK (old style)
```

We prefer **{ }** for initializer lists because that clearly indicates when initialization (construction) is done, and also because that notation is more widely useful. Logically, even built-in types have constructors so we can write:

```
int x {7};                  // OK (initializer list style)
x = int{9};
```

When we get to write functions that can be used for a mixture of built-in types and user-defined types (Chapter 18), the ability to use a uniform notation becomes essential.

8.4.3 Keep details private

We still have a problem: What if someone forgets to use the member function **add_day()**? What if someone decides to change the month directly? After all, we "forgot" to provide a facility for that:

```
Date birthday {1960,12,31};   // December 31, 1960
++birthday.d;                 // ouch! Invalid date (birthday.d==32 makes birthday invalid)

Date today {1924,2,3};
today.m = 14;                 // ouch! Invalid date (today.m==14 makes today invalid)
```

XX As long as we leave the representation of **Date** accessible to everybody, somebody will – by accident or design – mess it up; that is, someone will do something that produces an invalid value. In these examples, we gave **Date**s values that don't correspond to days on the calendar. Such invalid objects are time bombs; it is just a matter of time before someone innocently uses the invalid value and gets a run-time error or – usually worse – produces a bad result.

Such concerns lead us to conclude that the representation of **Date** should be inaccessible to users except through the public member functions that we supply. Here is a first cut:

```
// simple Date (control access)
class Date {
      int y, m, d;                    // year, month, day
public:
      Date(int y, int m, int d);      // check for valid date and initialize
      void add_day(int n);            // increase the Date by n days
      int month() { return m; }
      int day() { return d; }
      int year() { return y; }
};
```


We can use it like this:

```
Date birthday {1970, 12, 30};         // OK
birthday.m = 14;                      // error: Date::m is private
cout << birthday.month() << '\n';     // we provided a way to read m
```

The notion of a "valid **Date**" is an important special case of the idea of a valid value. We try to design our types so that values are guaranteed to be valid; that is, we hide the representation, provide a constructor that creates only valid objects, and design all member functions to expect valid values and leave only valid values behind when they return. The value of an object is often called its *state*, so the idea of a valid value is often referred to as a *valid state* of an object. AA

The alternative is for us to check for validity every time we use an object, or just hope that nobody left an invalid value lying around. Experience shows that "hoping" can lead to "pretty good" programs. However, producing "pretty good" programs that occasionally produce erroneous results and occasionally crash is no way to win friends and respect as a professional. We prefer to write code that can be demonstrated to be correct.

A rule for what constitutes a valid value is called an *invariant*. The invariant for **Date** ("A **Date** must represent a day in the past, present, or future") is unusually hard to state precisely: remember leap years, the Gregorian calendar, time zones, etc. However, for simple realistic uses of **Date**s we can do it. For example, if we are analyzing internet logs, we need not be bothered with the Gregorian, Julian, or Mayan calendars. If we can't think of a good invariant, we are probably dealing with plain data. If so, use a **struct**. CC

8.4.4 Defining member functions

So far, we have looked at **Date** from the point of view of an interface designer and a user. But sooner or later, we have to implement those member functions. First, here is a subset of **Date** reorganized to suit the common style of providing the public interface first:

```
// simple Date (many people prefer implementation details last)
class Date {
public:
      Date(int y, int m, int d);   // constructor: check for valid date and initialize
      void add_day(int n);         // increase the Date by n days
      int month();
      // ...
private:
      int y, m, d;                 // year, month, day
};
```

Many put the public interface first because the interface is what most people are interested in. In principle, a user need not look at the implementation details. In reality, we are typically curious and have a quick look to see if the implementation looks reasonable and if the implementer used some technique that we could learn from. However, unless we are the implementers, we do tend to spend much more time with the public interface. The compiler doesn't care about the order of class function and data members; it takes the declarations in any order you care to present them.

When we define a member outside its class, we need to say which class it is a member of. We do that using the *class_name* **::** *member_name* notation:

```
Date::Date(int yy, int mm, int dd)     // constructor
        :y{yy}, m{mm}, d{dd}           // note: member initializers
{
}

void Date::add_day(int n)
{
        // ...
}

int month()             // oops: we forgot Date::
{
        return m; // not the member function, can't access m
}
```

The :y{yy}, m{mm}, d{dd} notation is called a (member) initializer list. We use such lists to explicitly initialize members. We could have written

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

However, we would then in principle first have default initialized the members and then assigned values to them. We would then also open the possibility of accidentally using a member before it was initialized. The :y{yy}, m{mm}, d{dd} notation more directly expresses our intent. The distinction is exactly the same as the one between

```
int x;      // first define the variable x
// ...
x = 2;      // later assign to x
```

and

```
int x = 2;  // define and immediately initialize with 2
```

We can also define member functions right in the class definition:

```
class Date {
public:
        Date(int yy, int mm, int dd)
                :y{yy}, m{mm}, d{dd}
        {
        }

        void add_day(int n)
        {
                // ...
        }
```



```
        Date tomorrow;
        tomorrow.y = today.y;
        tomorrow.m = today.m;
        tomorrow.d = today.d+1;        // add 1 to today

        cout << tomorrow << '\n';      // use tomorrow
}
```

Here, we "forgot" to immediately initialize today and "someone" used it before we got around to calling init_day(). "Someone else" decided that it was a waste of time to call add_day() – or maybe hadn't heard of it – and constructed tomorrow by hand. As it happens, this is bad code – very bad code. Sometimes, probably most of the time, it works, but small changes lead to serious errors. For example, writing out an uninitialized Date will produce garbage output, and incrementing a day by simply adding 1 to its member d is a time bomb: when today is the last day of the month, the increment yields an invalid date. The worst aspect of this "very bad code" is that it doesn't look bad.

This kind of thinking leads to a demand for an initialization function that can't be forgotten and for operations that are less likely to be overlooked. The basic tool for that is *member functions*, that is, functions declared as members of the class within the class body. For example:

```
// simple Date
// guarantee initialization with constructor and provide some notational convenience
struct Date {
        int y, m, d;                    // year, month, day
        Date(int y, int m, int d);      // check for valid date and initialize
        void add_day(int n);            // increase the Date by n days
};
```

A member function with the same name as its class is special. It is called a *constructor* and will be used for initialization ("construction") of objects of the class. It is an error – caught by the compiler – to forget to initialize an object of a class that has a constructor that requires an argument, and there is a special convenient syntax for doing such initialization:

```
Date birthday;                          // error: birthday not initialized
Date today {12,24,2027};                // oops! run-time error
Date last {2005,12,31};                 // OK (colloquial style)
Date next = {2014,2,14};                // also OK (slightly verbose)
Date Beethoven = Date{1770,12,16};      // also OK (verbose style)
```

The attempt to declare birthday fails because we didn't specify the required initial value. The attempt to declare today will pass the compiler, but the checking code in the constructor will catch the illegal date at run time ({12,24,2027} – there is no day 2027 of the 24th month of year 12).

The definition of last provides the initial value – the arguments required by Date's constructor – as a { } list immediately after the name of the variable. That's the most common style of initialization of variables of a class that has a constructor requiring arguments. We can also use the more verbose style where we explicitly create an object (here, Date{1976,12,24}) and then use that to initialize the variable using the = initializer syntax. Unless you actually like typing, you'll soon tire of that.

Was year 2000 a leap year? Are you sure?

What we do then is to provide some *helper functions* to do the most common operations for us. That way, we don't have to repeat the same code over and over again and we won't make, find, and fix the same mistakes over and over again. For just about every type, initialization and assignment are among the most common operations. For Date, increasing the value of the Date is another common operation, so we add those as helpers:

```
void init_day(Date& dd, int y, int m, int d)
{
        // ... check that (y,m,d) is a valid date. If it is, use it to initialize dd ...
}

void add_day(Date& dd, int n)
{
        // ... increase dd by n days ...
}
```

We can now try to use Date:

```
void f()
{
        Date today;
        init_day(today, 12, 24, 2025);    // oops! (no day 2025 in year 12)
        add_day(today,1);
}
```

AA First we note the usefulness of such "operations" – here implemented as helper functions. Checking that a date is valid is sufficiently difficult and tedious that if we didn't write a checking function once and for all, we'd skip the check occasionally and get buggy programs. Whenever we define a type, we want some operations for it. Exactly how many operations we want and of which kind will vary. Exactly how we provide them (as functions, member functions, or operators) will also vary, but whenever we decide to provide a type, we ask ourselves, "Which operations would we like for this type?"

8.4.2 Member functions and constructors

We provided an initialization function for Dates, one that provided an important check on the validity of Dates. However, checking functions are of little use if we fail to use them. For example, assume that we have defined the output operator << for a Date:

```
void f()
{
        Date today;
        // ...
        cout << today << '\n';          // use today
        // ...
        init_day(today,2008,3,30);
        // ...
```



```
        int month() { return m; }

        // ...
private:
        int y, m, d;     // year, month, day
};
```

The first thing we notice is that the class declaration became larger and "messier." In this example, the code for the constructor and add_day() could be a dozen or more lines each. This makes the class declaration several times larger and makes it harder to find the interface among the implementation details. Consequently, we don't define large functions within a class declaration.

However, look at the definition of month(). That's straightforward and shorter than the version that places Date::month() out of the class declaration. For such short, simple functions, we might consider writing the definition right in the class declaration.

Note that month() can refer to m even though m is defined after (below) month(). A member can refer to a function or data member of its class independently of where in the class that other member is declared. The rule that a name must be declared before it is used is relaxed within the limited scope of a class.

Writing the definition of a member function within the class definition has three effects: CC

- The function will be *inline*; that is, the compiler will try to generate code for the function at each point of call rather than using function-call instructions to use common code. This can be a significant performance advantage for functions, such as month(), that hardly do anything but are used a lot.
- All uses of the class will have to be recompiled whenever we make a change to the body of an inlined function. If the function body is out of the class declaration, recompilation of users is needed only when the class declaration is itself changed. Not recompiling when the body is changed can be a huge advantage in large programs.
- The class definition gets larger. Consequently, it can be harder to find the members among the member function definitions.

The obvious rule of thumb is: Don't put member function bodies in the class declaration unless you know that you need the performance boost from inlining tiny functions. Large functions, say five or more lines of code, don't benefit from inlining and make a class declaration harder to read. We rarely inline a function that consists of more than one or two expressions. AA

TRY THIS

Get some example uses of a version of Date so far to run. For that, we need an output operator for Date. There is one in PPP_support, but for now use

```
ostream& operator<<(ostream& os, Date d)
{
        return os << d.year() << '/' << d.month() << '/' << d.day();
}
```

Chapter 9 explains why and how that works.

8.4.5 Referring to the current object

Consider a simple use of the **Date** class so far:

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

void f(Date d1, Date d2)
{
        cout << d1.month() << ' ' << d2.month() << '\n';
}
```

How does **Date::month()** know to return the value of **d1.m** in the first call and **d2.m** in the second? Look again at **Date::month()**; its declaration specifies no function argument! How does **Date::month()** know for which object it was called? A class member function, such as **Date::month()**, has an implicit argument which it uses to identify the object for which it is called. So in the first call, **m** correctly refers to **d1.m** and in the second call it refers to **d2.m**. See §15.8 for more uses of this implicit argument.

8.4.6 Reporting errors

CC What do we do when we find an invalid date? Where in the code do we look for invalid dates? From §4.6, we know that the answer to the first question is "Throw an exception," and the obvious place to look is where we first construct a **Date**. If we don't create invalid **Date**s and also write our member functions correctly, we will never have a **Date** with an invalid value. So, we'll prevent users from ever creating a **Date** with an invalid state:

```
// simple Date (prevent invalid dates)
class Date {
public:
        class Invalid { };                  // to be used as exception
        Date(int y, int m, int d);          // check for valid date and initialize
        // ...
        bool is_valid();                    // return true if date is valid
private:
        int y, m, d;                        // year, month, day
};
```

We put the testing of validity into a separate **is_valid()** function because checking for validity is logically distinct from initialization and because we might want to have several constructors. As you can see, we can have private functions as well as private data:

8.4 Evolving a class: Date

Let's illustrate the language facilities supporting classes and the basic techniques for using them by showing how – and why – we might evolve a simple data structure into a class with private implementation details and supporting operations. We use the apparently trivial problem of how to represent a date (such as August 14, 1954) in a program. The need for dates in many programs is obvious (commercial transactions, weather data, calendar programs, work records, inventory management, etc.). The only question is how we might represent them.

8.4.1 struct and functions

How would we represent a date? When asked, most people answer, "Well, how about the year, the month, and the day of the month?" That's not the only answer and not always the best answer, but it's good enough for our use here, so that's what we'll do. Our first attempt is a simple **struct**:

```
// simple Date (too simple?)
struct Date {
        int y;      // year
        int m;      // month in year
        int d;      // day of month
};

Date today;         // a Date variable (a named object)
```

A **Date** object, such as **today**, will simply be three **int**s:

	Date:
y:	2025
m:	12
d:	24

There is no "magic" relying on hidden data structures anywhere related to a **Date** – and that will be the case for every version of **Date** in this chapter.

So, we now have **Date**s; what can we do with them? We can do everything in the sense that we can access the members of **today** (and any other **Date**) and read and write them as we like. The snag is that nothing is really convenient. Just about anything that we want to do with a **Date** has to be written in terms of reads and writes of those members. For example:

```
today.y = 2025;     // set today to December 24, 2025
today.m = 24;
today.d = 12;
```

This is tedious and error prone. Did you spot the error? Everything that's tedious is error-prone! AA
Some errors are harder to spot. How about

```
Date y;             // set today to December 24, 2025:
y.y = 2000;
y.m = 2;
y.d = 29;
```

Class members are private by default; that is,

```
class X {
	int mf(int);
	// ...
};
```

means

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

so that

```
X x;                // variable x of type X
int y = x.mf();     // error: mf is private (i.e., inaccessible)
```

A user cannot directly refer to a private member. Instead, we have to go through a public function that can use it. For example:

```
class X {
	int m;
	int mf(int);
public:
	int f(int i) { m=i; return mf(i); }
};

X x;
int y = x.f(2);
```

We use **private** and **public** to represent the important distinction between an interface (the user's view of the class) and implementation details (the implementer's view of the class). We explain that and give lots of examples as we go along. Here we'll just mention that for something that's just data, this distinction doesn't make sense. So, there is a useful simplified notation for a class that has no private implementation details. A **struct** is a **class** where members are public by default:

```
struct X {
	int m;
	// ...
};
```

means

```
class X {
public:
	int m;
	// ...
};
```

structs are primarily used for data structures where the members can take any value; that is, we can't define any meaningful invariant (§8.4.3).


```
Date::Date(int yy, int mm, int dd)
	: y{yy}, m{mm}, d{dd}           // initialize data members
{
	if (!is_valid())                // check for validity
		throw Invalid{};
}

bool Date::is_valid()                   // return true if date is valid
{
	return 0<m && m<13;             // very incomplete check
}
```

Given that definition of **Date**, we can write

```
void f(int x, int y)
try {
	Date dxy {2024,x,y};
	cout << dxy << '\n';
	dxy.add_day(2);
}
catch(Date::Invalid) {
	error("f(): invalid date");     // error() defined in §4.6.3
}
```

We now know that **<<** and **add_day()** will have a valid **Date** on which to operate.

Before completing the evolution of our **Date** class (§8.7), we take a detour to describe a couple of general language facilities that we need to do that well: enumerations and operator overloading.

8.5 Enumerations

CC

An **enum** (an *enumeration*) is a very simple user-defined type, specifying its set of values (its *enumerators*) as symbolic constants. For example:

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

The "body" of an enumeration is simply a list of its enumerators. The **class** in **enum class** means that the enumerators are in the scope of the enumeration. That is, to refer to **jan**, we have to say **Month::jan**.

You can give a specific representation value for an enumerator, as we did for **jan** here, or leave it to the compiler to pick a suitable value. If you leave it to the compiler to pick, it'll give each enumerator the value of the previous enumerator plus one. Thus, our definition of **Month** gave the months consecutive values starting with **1**. We could equivalently have written

```
enum class Month {
	jan=1, feb=2, mar=3, apr=4, may=5, jun=6, jul=7, aug=8, sep=9, oct=10, nov=11, dec=12
};
```

However, that's tedious and opens the opportunity for errors. It is better to let the compiler do simple, repetitive "mechanical" things. The compiler is better at such tasks than we are, and it doesn't get bored.

If we don't initialize the first enumerator, the count starts with **0**. For example:

```
enum class Day {
    monday, tuesday, wednesday, thursday, friday, saturday, sunday
};
```

Here **monday** is represented as **0** and **sunday** is represented as **6**. Starting with **0** is often a good choice.

We can use our **Month** like this:

```
Month m1 = Month::feb;
Month m2 = feb;             // error: feb is not in scope
Month m3 = 7;               // error: can't assign an int to a Month
Month m4 = Month{7};        // OK: explicit conversion
Month m5 {7};               // OK: explicit initialization

int x1 = m1;                // error: can't assign a Month to an int
int x2 = int{m1};           // error: narrowing conversion
int x3 = to_int(m1);        // convert Month to int; see below
```

Month is a separate type from its "underlying type" **int**. Every **Month** has an equivalent integer value, but most **int**s do not have a **Month** equivalent. For example, we really do want this initialization to fail:

```
Month bad = 9999;       // error: can't convert an int to a Month
```

XX The explicit **Month{7}** conversion is unchecked so use it only when you are certain that the value to be converted really fits your idea of a **Month**. We cannot define a constructor for an enumeration to check initializer values, but it is trivial to write a simple checking function:

```
Month int_to_month(int x)
    // checked conversion
{
    if (x<to_int(Month::jan) || to_int(Month::dec)<x)
        error("bad month");
    return Month{x};
}
```

We use the **to_int(Month::jan)** notation to get the **int** representation of **Month::jan**. For example:

```
void f(int m)
{
    Month mm = int_to_month(m);
    // ...
}
```

The ways of converting a **Month** to its underlying type **int** are a bit messy, so in **PPP_support**, we define a function to do it:

In C++, a class is the key building block for large programs – and very useful for small ones as well, as we saw for our calculator (Chapter 5 and Chapter 6).

8.2 Classes and members

CC A class is a user-defined type. It is composed of built-in types, other user-defined types, and functions. The parts used to define the class are called *members*. A class has zero or more members. For example:

```
class X {
public:
    int m;                                          // data member
    int mf(int v) { int old = m; m=v; return old; } // function member
};
```

Members can be of various types. Most are either data members, which define the representation of an object of the class, or function members, which provide operations on such objects. We access members using the *object.member* notation. For example:

```
X var;                  // var is a variable of type X
var.m = 7;              // assign to var's data member m
int x = var.mf(9);      // call var's member function mf()
```

You can read **var.m** as **var**'s **m**. Most people pronounce it "**var** dot **m**" or "**var**'s **m**." The type of a member determines what operations we can do on it. For example, we can read and write an **int** member and call a member function.

A member function, such as **X**'s **mf()**, does not need to use the **var.m** notation. It can use the plain member name (**m** in this example). Within a member function, a member name refers to the member of that name in the object for which the member function was called. Thus, in the call **var.mf(9)**, the **m** in the definition of **mf()** refers to **var.m**.

8.3 Interface and implementation

CC Usually, we think of a class as having an interface plus an implementation. The interface is the part of the class's declaration that its users access directly. The implementation is that part of the class's declaration that its users access only indirectly through the interface. The public interface is identified by the label **public:** and the implementation by the label **private:**. You can think of a class declaration like this:

```
class X {       // this class's name is X
public:
    // the interface to users (accessible by all)
    // functions, types, and data (data is often best kept private)
private:
    // the implementation details (used by members of this class only)
    // functions, types, and data
};
```

8.1 User-defined types

CC The C++ language provides you with some built-in types, such as **char**, **int**, and **double**. A type is called built-in if the compiler knows how to represent objects of the type and which operations can be done on it (such as **+** and *) without being told by declarations supplied by a programmer in source code.

CC Types that are not built-in are called *user-defined types*. They can be standard-library types – available to all C++ programmers as part of every ISO standard C++ implementation – such as **string**, **vector**, and **ostream** (Chapter 9), or types that we build for ourselves, such as **Token** and **Token_stream** (§5.3.2 and §5.8). As soon as we get the necessary technicalities under our belt, we'll build graphics types such as **Shape**, **Line**, and **Text** (Chapter 11). The standard-library types are as much a part of the language as the built-in types, but we still consider them user-defined because they are built from the same primitives and with the same techniques as the types we built ourselves; the standard-library builders have no special privileges or facilities that you don't have. Like the built-in types, most user-defined types provide operations. For example, **vector** has **[ ]** and **size()** (§3.6.1), **ostream** has **<<** (§9), **Token_stream** has **get()** (§5.8), and **Shape** has **add(Point)** and **set_color()** (§12.2).

AA Why do we build types? The compiler does not know all the types we might like to use in our programs. It couldn't, because there are far too many useful types – no language designer or compiler implementer could know them all. We invent new ones every day. Why? What are types good for? Types are good for directly representing ideas in code. When we write code, the ideal is to represent our ideas directly in our code so that we, our colleagues, and the compiler can understand what we wrote. When we want to do integer arithmetic, **int** is a great help; when we want to manipulate text, **string** is a great help; when we want to manipulate calculator input, **Token** and **Token_stream** are a great help. The help comes in two forms:

- *Representation*: A type ''knows'' how to represent the data needed in an object.
- *Operations*: A type ''knows'' what operations can be applied to objects.

Many ideas follow this pattern: ''something'' has data to represent its current value – sometimes called the current *state* – and a set of operations that can be applied. Think of a computer file, a Web page, a toaster, a music player, a coffee cup, an electric motor, a cell phone, a telephone directory; all can be characterized by some data and all have a more or less fixed set of standard operations that you can perform. In each case, the result of the operation depends on the data – the current state – of an object.

So, we want to represent such an ''idea'' or ''concept'' in code as a data structure plus a set of functions. The question is: ''Exactly how?'' This chapter presents the technicalities of the basic ways of doing that in C++.

CC C++ provides two kinds of user-defined types: classes and enumerations. The class is by far the most general and important, so we first focus on classes. A class directly represents a concept in a program. A *class* is a (user-defined) type that specifies how objects of its type are represented, how those objects can be created, how they can be used, and how they can be destroyed (Chapter 17). If you think of something as a separate entity, it is likely that you should define a class to represent that ''thing'' in your program. Examples are vector, matrix, input stream, string, FFT (fast Fourier transform), valve controller, robot arm, device driver, picture on screen, dialog box, graph, window, temperature reading, and clock.


```
int to_int(Month m)
{
      return static_cast<int>(m);
}
```

What do we use enumerations for? Basically, an enumeration is useful whenever we need a set of related named integer constants. That happens all the time when we try to represent sets of alternatives (**up**, **down**; **yes**, **no**, **maybe**; **on**, **off**; **n**, **ne**, **e**, **se**, **s**, **sw**, **w**, **nw**) or distinctive values (**red**, **blue**, **green**, **yellow**, **maroon**, **crimson**, **black**).

8.5.1 ''Plain'' enumerations

In addition to the **enum class**es, also known as *scoped enumerations*, there are ''plain'' enumerations that differ from scoped enumerations by implicitly ''exporting'' their enumerators to the scope of the enumeration and allowing implicit conversions to **int**. For example:

```
enum Month {            // note: no "class"
      jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

Month m1 = feb;               // OK: feb in scope
Month m2 = Month::feb;        // also OK
Month m3 = 7;                 // error: can't assign an int to a Month
Month m4 = Month{7};          // OK: explicit conversion

int x1 = m1;                  // OK: we can assign a Month to an int
```

Obviously, ''plain'' **enum**s are less strict than **enum class**es. Their enumerators can ''pollute'' the scope in which their enumerator is defined. That can be a convenience, but it occasionally leads to surprises. For example, if you try to use this **Month** together with the **iostream** formatting mechanisms (§9.10.1), you will find that **dec** for December clashes with **dec** for decimal.

Similarly, having an enumeration value convert to **int** can be a convenience by saving us from being explicit when we want a conversion to **int**. However, when we don't want such implicit conversion, it can lead to surprises and errors. For example:

```
void bad_code(Month m)
{
      if (m==17)                  // huh: 17th month?
            do_something();
      if (m==monday)              // huh: compare month to Monday?
            do_something_else();
}
```

If **Month** is an **enum class**, neither condition will compile. If **Month** is a plain **class** and **monday** is an enumerator of a ''plain'' **enum**, rather than an **enum class**, both comparisons will succeed, most likely with undesirable results.

Prefer the simpler and safer **enum class**es to ''plain'' **enum**s, but expect to find ''plain'' **enum**s in older code: **enum class**es were new in C++11.

8.6 Operator overloading

You can define almost all C++ operators for class or enumeration operands. That's often called *operator overloading*. We use it when we want to provide conventional notation for a type we design. For example, we can provide an increment operator for our **Month** type:

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

Month operator++(Month& m)        // prefix increment operator
{
        m = (m==Month::dec) ? Month::jan : Month{to_int(m)+1};      // "wrap around"
        return m;
}
```

The **? :** construct is an "arithmetic if": **m** becomes **Jan** if **(m==Dec)** and **Month(to_int(m)+1)** otherwise. It is a reasonably elegant way of expressing the fact that months "wrap around" after December. The **Month** type can now be used like this:

```
Month m = Month::oct;
++m;        // m becomes nov
++m;        // m becomes dec
++m;        // m becomes jan ("wrap around")
```

You might not think that incrementing a **Month** is common enough to warrant a special operator. That may be so, but how about an output operator? We can define one like this:

```
vector<string> month_tbl = {"Not a month", "January", "February", "March", /* ... */ };

ostream& operator<<(ostream& os, Month m)
{
        return os << month_tbl[to_int(m)];
}
```

We gave **Month::jan** the conventional integer value **1**, so **month_tbl[0]** does not represent a month.

We can now control the appearance of a month on output by changing **month_tbl**. For example, we could set **month_tbl[to_int(Month::mar)]** to **"marzo"** or some other suitable name for that month; see §9.9.3.

CC We can define just about any operator provided by C++ for our own types, but only existing operators, such as **+**, **–** *, /, **%**, **[ ]**, **( )**, ^, **!**, **&**, **<**, **<=**, **>**, and **>=**. We cannot define our own operators; we might like to have ** or **@=** as operators in our program, but C++ won't let us. We can define operators only with their conventional number of operands. For example, we can define unary **–**, but not unary **<=** (less than or equal), and binary **+**, but not binary **!** (not). Basically, the language allows us to use the existing syntax for the types you define, but not to extend that syntax.

An overloaded operator must have at least one user-defined type as operand:

```
int operator+(int,int);                              // error: you can't overload built-in +
Vector operator+(const Vector&, const Vector &);     // OK
Vector operator+=(const Vector&, int);               // OK
```

8

Technicalities: Classes, etc.

Remember, things take time.
– Piet Hein

In this chapter, we keep our focus on our main tool for programming: the C++ programming language. We present language technicalities, mostly related to user-defined types, that is, to classes and enumerations. Much of the presentation of language features takes the form of the gradual improvement of a **Date** type. That way, we also get a chance to demonstrate some useful class design techniques.

[10] Write a function `maxv()` that returns the largest element of a `vector` argument.

[11] Write a function that finds the smallest and the largest element of a `vector` argument and also computes the mean and the median. Do not use global variables. Either return a `struct` containing the results or pass them back through reference arguments. Which of the two ways of returning several result values do you prefer and why?

[12] Improve `print_until_s()` from §7.4.2. Test it. What makes a good set of test cases? Give reasons. Then, write a `print_until_ss()` that prints until it sees a second occurrence of its `quit` argument.

[13] Write a function that takes a `vector<string>` argument and returns a `vector<int>` containing the number of characters in each `string`. Also find the longest and the shortest `string` and the lexicographically first and last `string`. How many separate functions would you use for these tasks? Why?

[14] Can we declare a non-reference function argument `const` (e.g., `void f(const int);`)? What might that mean? Why might we want to do that? Why don't people do that often? Try it; write a couple of small programs to see what works.

Postscript

We could have put much of this chapter (and much of the next) into an appendix. However, you'll need most of the facilities described here in the rest of this book. You'll also encounter most of the problems that these facilities were invented to help solve very soon. Most simple programming projects that you might undertake will require you to solve such problems. So, to save time and minimize confusion, a somewhat systematic approach is called for, rather than a series of ''random'' visits to manuals and appendices.

It is generally a good idea *not* to define operators for a type unless you are really certain that it makes a big positive change to your code. Also, define operators only with their conventional meaning: `+` should be addition, binary `*` multiplication, `[ ]` access, `( )` call, etc. This is just advice, not a language rule, but it is good advice: conventional use of operators, such as `+` for addition, can significantly help us understand a program. After all, such use is the result of hundreds of years of experience with mathematical notation. Conversely, obscure operators and unconventional use of operators can be a significant distraction and a source of errors. We will not elaborate on this point. Instead, in the following chapters, we will simply use operator overloading in a few places where we consider it appropriate. **AA**

Note that the most interesting operators to overload aren't `+`, `–`, `*`, and `/` as people often assume, but `=` (assignment), `==` (equality), `<` (less than), `–>` (dereference), `[ ]` (subscript), and `( )` (call).

TRY THIS
Write, compile, and run a small example using `++` and `<<` for `Month`.

8.7 Class interfaces

We have argued that the public interface and the implementation parts of a class should be separated. As long as we leave open the possibility of using `struct`s for types that are just collections of data, few professionals would disagree. However, how do we design a good interface? What distinguishes a good public interface from a mess? Part of that answer can be given only by example, but there are a few general principles that we can list and that are given some support in C++: **AA**

- Keep interfaces complete.
- Keep interfaces minimal.
- Provide constructors.
- Support copying (or prohibit it) (see §12.4.1).
- Use types to provide good argument checking.
- Identify nonmodifying member functions (see §8.7.4).
- Free all resources in the destructor (see §15.5).

See also §4.5 (how to detect and report run-time errors).

The first two principles can be combined to

- Keep the interface as small as possible, but no smaller.

We want our interface to be small because a small interface is easy to learn and easy to remember, and the implementer doesn't waste a lot of time implementing unnecessary and rarely used facilities. A small interface also means that when something is wrong, there are only a few functions to check to find the problem. On average, the more public member functions a class has, the harder it is to find bugs. But of course, we want a complete interface; otherwise, it would be useless. We couldn't use an interface that didn't allow us to do all we really needed. For operations beyond the minimal set, use ''helper functions'' (§8.7.5).

Let's look at the other – less abstract and more directly supported – ideals.

8.7.1 Argument types

When we defined the constructor for **Date** in §8.4.3, we used three **int**s as the arguments. That caused some problems:

```
Date d1 {4,5,2005};    // oops: year 4, day 2005
Date d2 {2005,4,5};    // April 5 or May 4?
```

The first problem (an illegal day of the month) is easily dealt with by a test in the constructor. The second problem is simply that the conventions for writing month and day-in-month differ; for example, 4/5 is April 5 in the United States and May 4 in England. We can't calculate our way out of this, so we must do something else. The obvious solution is to use a **Month** type:

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

// simple Date (use Month type)
class Date {
public:
        Date(int y, Month m, int d);          // check for valid date and initialize
        // . . .
private:
        int y;          // year
        Month m;
        int d;          // day
};
```

When we use a **Month** type, the compiler will catch us if we swap month and day, and using an enumeration as the **Month** type also gives us symbolic names to use. It is usually easier to read and write symbolic names than to play around with numbers, and therefore, less error-prone:

```
Date dx1 {1998, 4, 3};                  // error: 2nd argument not a Month
Date dx2 {1998, 4, Month::mar};         // error: 2nd argument not a Month
Date dx3 {4, Month::mar, 1998};         // oops: run-time error: day 1998
Date dx4 {Month::mar, 4, 1998};         // error: 2nd argument not a Month
Date dx5 {1998, Month::mar, 30};        // OK
```

This takes care of most "accidents." Note the use of the qualification of the enumerator **mar** with the enumeration name: **Month::mar**. We don't say **Month.mar** because **Month** isn't an object (it's a type) and **mar** isn't a data member (it's an enumerator – a symbolic constant). Use **::** after the name of a class, enumeration, or namespace (§7.6.1) and . (dot) after an object name.

AA When we have a choice, we catch errors at compile time rather than at run time. We prefer for the compiler to find the error rather than for us to try to figure out exactly where in the code a problem occurred. Importantly, errors caught at compile time don't require us to write tests and error-handling code. Catching errors at compile time makes code simpler and faster.

Thinking like that, could we catch the swap of the day of the month and the year also? We could, but the solution is not as simple or as elegant as for **Month**; after all, there was a year 4 and you might want to represent it. Even if we restricted ourselves to modern times there would probably be too many relevant years for us to list them all in an enumeration.

Terms

activation record	function	pass-by-reference	argument
function definition	pass-by-value	argument passing	global scope
recursion	call stack	header file	**return**
class scope	initializer	return value	**const**
local scope	scope	**constexpr**	**namespace**
statement scope	declaration	namespace scope	technicalities
definition	nested block	undeclared identifier	**extern**
parameter	**using** declaration	forward declaration	pass-by-**const**-reference
using directive	**auto**	**–>**	suffix return type

Exercises

[1] Modify the calculator program from Chapter 6 to make the input stream an explicit parameter (as shown in §7.4.8), rather than simply using **cin**. Also give the **Token_stream** constructor (§6.8.2) an **istream&** parameter so that when we figure out how to make our own **istream**s (e.g., attached to files), we can use the calculator for those. Hint: Don't try to copy an **istream**.

[2] Write a function **print()** that prints a **vector** of **int**s to **cout**. Give it two arguments: a **string** for "labeling" the output and a **vector**.

[3] Create a **vector** of Fibonacci numbers and print them using the function from exercise 2. To create the **vector**, write a function, **fibonacci(x,y,v,n)**, where integers **x** and **y** are **int**s, **v** is an empty **vector<int>**, and **n** is the number of elements to put into **v**; **v[0]** will be **x** and **v[1]** will be **y**. A Fibonacci number is one that is part of a sequence where each element is the sum of the two previous ones. For example, starting with 1 and 2, we get 1, 2, 3, 5, 8, 13, 21, ... Your **fibonacci()** function should make such a sequence starting with its **x** and **y** arguments.

[4] An **int** can hold integers only up to a maximum number. Find an approximation of that maximum number by using **fibonacci()**.

[5] Write two functions that reverse the order of elements in a **vector<int>**. For example, 1, 3, 5, 7, 9 becomes 9, 7, 5, 3, 1. The first reverse function should produce a new **vector** with the reversed sequence, leaving its original **vector** unchanged. The other reverse function should reverse the elements of its **vector** without using any other **vector**s (hint: **swap**).

[6] Write versions of the functions from exercise 5, but with a **vector<string>**.

[7] Read five names into a **vector<string> name**, then prompt the user for the ages of the people named and store the ages in a **vector<double> age**. Then print out the five (**name[i],age[i]**) pairs. Sort the names (**sort(name.begin(),name.end())**) and print out the (**name[i],age[i]**) pairs. The tricky part here is to get the **age vector** in the correct order to match the sorted **name vector**. Hint: Before sorting **name**, take a copy and use that to make a copy of **age** in the right order after sorting **name**.

[8] Do the previous exercise but allow an arbitrary number of names.

[9] Write a function that given two **vector<double>**s **price** and **weight** computes a value (an "index") that is the sum of all **price[i]*weight[i]**. Make sure to have **weight.size()==price.size()**.

```
extern int foo;
void print_foo();
void print(int);
```

Write a file foo.cpp that implements the functions declared in foo.h. Write file use.cpp that #includes foo.h and tests it. Get the resulting program to compile and run.

Review

[1] What is the difference between a declaration and a definition?
[2] How do we syntactically distinguish between a function declaration and a function definition?
[3] How do we syntactically distinguish between a variable declaration and a variable definition?
[4] Why can't you use the functions in the calculator program from Chapter 5 without declaring one or more of them first?
[5] Is int a; a definition or just a declaration?
[6] Why is it a good idea to initialize variables as they are declared?
[7] What can a function declaration consist of?
[8] What is the *suffix return type* notation, and why might you use it?
[9] What good does indentation do?
[10] What is the scope of a declaration?
[11] What kinds of scope are there? Give an example of each.
[12] What is the difference between a class scope and local scope?
[13] Why should a programmer minimize the number of global variables?
[14] What is the difference between pass-by-value and pass-by-reference?
[15] What is the difference between pass-by-reference and pass-by-const-reference?
[16] What is a swap()?
[17] Would you ever define a function with a vector<double> as a by-value parameter?
[18] Give an example of undefined order of evaluation. Why can undefined order of evaluation be a problem?
[19] What do x&&y and x||y, respectively, mean?
[20] Which of the following is standard-conforming C++: functions within functions, functions within classes, classes within classes, classes within functions?
[21] What goes into an activation record?
[22] What is a call stack and why do we need one?
[23] What is the purpose of a namespace?
[24] How does a namespace differ from a class?
[25] What is a using declaration?
[26] Why should you avoid using directives in a header?
[27] What is namespace std?

Probably the best we could do (without knowing quite a lot about the intended use of Date) would be a minimal Year type:

```
struct Year {
      int y;
};

class Date {
public:
      Date(Year y, Month m, int d);          // check for valid date and initialize
      // ...
private:
      Year y;
      Month m;
      int d;             // day
};
```

Now we get

```
Date dx1 {Year{1998}, 4, 3};                    // error: 2nd argument not a Month
Date dx2 {Year{1998}, 4, Month::mar};           // error: 2nd argument not a Month
Date dx3 {4, Month::mar, Year{1998}};           // error: 1st argument not a Year
Date dx4 {Month::mar, 4, Year{1998}};           // error: 2nd argument not a Month
Date dx5 {Year{1998}, Month::mar, 30};          // OK
Date dx6 {Year{4}, Month::mar, 1998};           // run-time error: Year::Invalid
```

We could modify Year to check for unlikely years, but would the extra work be worthwhile? Naturally, that depends on the constraints on the kind of problem you are solving using Date.

AA

When we program, we always have to ask ourselves what is good enough for a given application. We usually don't have the luxury of being able to search "forever" for the perfect solution after we have already found one that is good enough. Search further, and we might even come up with something that's so elaborate that it is worse than the simple early solution. This is one meaning of the saying "The best is the enemy of the good" (Voltaire).

8.7.2 Copying

We always have to create objects; that is, we must always consider initialization and constructors. Arguably they are the most important members of a class: to write them, you have to decide what it takes to initialize an object and what it means for a value to be valid (what is the invariant?). Just thinking about initialization will help you avoid errors.

The next thing to consider is often: Can we copy our objects? And if so, how do we copy them?

For Date or Month, the answer is that we obviously want to copy objects of that type and that the meaning of *copy* is trivial: just copy all of the members. Actually, this is the default case. So as long as you don't say anything else, the compiler will do exactly that. For example, if you copy a Date as an initializer or right-hand side of an assignment, all its members are copied:

```
Date holiday {Year{1978}, Month::jul, 4};          // initialization
Date d2 = holiday;
Date d3 = Date{Year{1978}, Month::jul, 4};
```

```
holiday = Date{Year{1978}, Month::dec, 24};          // assignment
d3 = holiday;
```

This will all work as expected. The **Date{Year{1978}, Month::dec, 24}** notation makes the appropriate unnamed **Date** object, which you can then use appropriately. For example:

```
cout << Date{Year{1978}, Month::dec, 24};
```

This is a use of a constructor that acts much as a literal for a class type. It often comes in as a handy alternative to first defining a variable or **const** and then using it once.

What if we don't want the default meaning of copying? We can either define our own (§17.4) or **delete** the copy constructor and copy assignment (§12.4.1).

8.7.3 Default constructors

CC

Uninitialized variables can be a serious source of errors. To counter that problem, we have the notion of a constructor to guarantee that every object of a class is initialized. For example, we declared the constructor **Date::Date(int,Month,int)** to ensure that every **Date** is properly initialized. In the case of **Date**, that means that the programmer must supply three arguments of the right types. For example:

```
Date d0;                          // error: no initializer
Date d1 {};                       // error: empty initializer
Date d2 {Year{1998}};             // error: too few arguments
Date d3 {Year{1},2,3,4};          // error: too many arguments
Date d4 {Year{1},"jan",2};        // error: wrong argument type
Date d5 {Year{1},Month::jan,2};   // OK: use the three-argument constructor
Date d6 {d5};                     // OK: use the copy constructor
```

Note that even though we defined a constructor for **Date**, we can still copy **Date**s.

Many classes have a good notion of a default value; that is, there is an obvious answer to the question "What value should it have if I didn't give it an initializer?" For example:

```
string s1;              // default value: the empty string ""
vector<string> v1;      // default value: the empty vector; no elements
```

This looks reasonable. It even works the way the comments indicate. That is achieved by giving **vector** and **string** each a *default constructor* that implicitly provide the desired initialization. A constructor that can be called with no arguments is called a default constructor.

Using a default constructor is not just a matter of looks. Just imagine the errors we could get if we could have an uninitialized **string** or **vector**:

```
string s;                           // imagine that s could be uninitialized
for (int i = 0; i<s.size(); ++i)    // oops: loop an undefined number of times
        s[i] = toupper(s[i]);       // oops: read and write a random memory location

vector<string> v;                   //imagine that v could be uninitialized
v.push_back("bad");                 // oops: write to random address
```



```
int x = 7;
int y = 9;
swap_?(x,y);          // replace ? by v, r, or cr
swap_?(7,9);
const int cx = 7;
const int cy = 9;
swap_?(cx,cy);
swap_?(7.7,9.9);
double dx = 7.7;
double dy = 9.9;
swap_?(dx,dy);
swap_?(7.7,9.9);
```

Which functions and calls compiled, and why? After each swap that compiled, print the value of the arguments after the call to see if they were actually swapped. If you are surprised by a result, consult §7.5.

[2] Write a program using a single file containing three namespaces **X**, **Y**, and **Z** so that the following **main()** works correctly:

```
int main()
{
        X::var = 7;
        X::print();             // print X's var
        using namespace Y;
        var = 9;
        print();                // print Y's var
        {
                using Z::var;
                using Z::print;
                var = 11;
                print();        // print Z's var
        }
        print();                // print Y's var
        X::print(); // print X's var
}
```

Each namespace needs to define a variable called **var** and a function called **print()** that outputs the appropriate **var** using **cout**.

[3] Create a module **foo** with the suffix appropriate to your system:

```
int foo = 0;
export void print_foo() { ... };
export void set_foo(int x) { foo = x; }
export int get_foo() { return x; }
```

Add what it takes to get the **...** part to print **foo**. Write file **use.cpp** that **import**s **foo** and tests it. Get the resulting program to compile and run.

[4] Create a header file: **foo.h**: