7 Templates

Aggregation, inheritance, and templates are the three techniques for building bigger classes out of smaller data types. Any data type can be thought of as a chunk of functionality. Templates give us a syntax for inserting these chunks into, or withholding them from, the bigger classes that we create.

We can do the insertion whenever we find ourselves plugging different data types into the same code:

```
class wrapper_int {
    int x;
    int f();
};

class wrapper_date {
    date x;
    date f();
};

class wrapper_pointer_to_wabbit {
    wabbit *x;
    wabbit *f();
};

class wrapper_wrapper_int {
    wrapper_int w;
    wrapper_int f();
};
```

A ‘‘template class’’ will let us write the code once and for all with something like a blank.

```
class wrapper {
    ____ x;
    ____ f();
};
```

The choice of data type will be plugged into the blank at a later time, perhaps several choices at several later times.

7.1 Template Functions

The above example plugged a data type into a ‘‘template class’’. We can also plug a data type into a ‘‘template function’’. A summary of the differences between template functions and template classes is on p. 757. A function is simpler than a class, but it will turn out that a template function is more complicated than a template class. One problem is that the data type plugged into a template class is always specified explicitly by the user, while the type plugged into a template function usually has to be ‘‘deduced’’ by the computer. In addition, template functions have to compensate for their lack of ‘‘partial specialization’’ (p. 702). Finally, function templates interact with function name overloading, but there is no class name overloading for class templates to interact with.

It will be a long road to the final apotheosis of the template paradigm in Chapters 8 and 9. We will bite the bullet and start with template functions.

7.1.1 Simple Examples: `min`, `print`, and `swap`

Operator overloading: its purpose revealed!

Let’s build our own version of the `min` function in the C++ Standard Library. The following lines 29, 34, and 39 define three functions with this name. Each function mentions a different data type, but they are otherwise identical. To keep them identical, the objects in line 39 are passed by value. They should have beeen passed by reference, and on p. 640 they will be.

The three functions can be overloads of the same name since their arguments are different. The functions belong to no namespace, while the `min` in the standard library belongs to namespace `std` (p. 641).

We acknowledge that the comparison in line 31 seems to be backwards. Wouldn’t it be more natural to write the code in the comment alongside? After all, the only apparent difference is that when `a` and `b` are equal, the code returns `a` and the comment returns `b`. How could this be significant when the variables are returned by value?

But the arguments and return value will soon be passed by reference (p. 640). And closer inspection reveals that the code and the comment do not test for equality at all. What actually happens is that when neither `a` nor `b` is less than the other, the code returns `a` and the comment returns `b`.

When `a` and `b` are integers, they must be equal when neither is less than the other. But for certain exotic data types, `a` and `b` could be unequal even though neither is less than the other (p. 778). When this happens, the `min` function in the library returns a reference to `a` (p. 641). We want our `min` to behave the same way.

Lines 19−21 call our `min` functions with arguments of different types: `int`, `double`, `date`. We assume that class `date` has the default constructor in line 16, the copy constructor in line 17, and three overloaded operators:

(1) the binary `operator+` in line 17;

(2) the `operator<<` in line 21;

(3) the `operator<` in line 41.

We also assume that the `min` we hard-coded for class date on p. 211 has been removed.

Thanks to the `operator<` friend of class `date`, the code at line 41 can be identical to that at 31 and 36. Operator overloading gives us a *convenient* notation for variables of all data types, including objects. But this is only a fortunate accident. The real purpose of operator overloading is to give us the *same* notation for variables of all data types. Now that the functions are identical, we will be able to replace them with a single ‘‘template’’. (For another exmple in which templates influence our coding style, see p. 648.)

Of course, a few exceptional data types will require different code. Line 47 needs the `strcmp` function to compare strings of `char`’s for alphabetical order. The rejected code in line 46 would merely tell us which string begins earlier in memory.

Our `min` functions belong to no namespace. Another `min`, belonging to namespace `std`, is declared in the header file `<algorithm>`. We did not include this header directly, but it might have been included by a header that we did include. The double colon in line 19 ensures that we will call the `min` that belongs to no namespace, even if `<algorithm>` was included. With `<algorithm>`, and without the `using namespace std;`, `std::min` would have been the one that belongs to namespace `std`, and an unadorned `min` would not have compiled.*

* Without the `using` directive in line 6, an unadorned `min` in line 19 would be the `min` that belongs to no namespace. We would then have to prepend `std::` to `cout`, `endl`, `wcout`, `strcmp`, and `wcscmp`. Alternatively, we could insert

The `endl` in line 22 flushes the characters sent to `cout` before line 24 begins to send wide characters to `wcout`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/min/min1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cstring>   //for strcmp
 4 #include <cwchar>    //for wcscmp
 5 #include "date.h"
 6 using namespace std;
 7
 8 int min(int a, int b);   //function declaration
 9 double min(double a, double b);
10 date min(date a, date b);
11 const char *min(const char *a, const char *b);
12 const wchar_t *min(const wchar_t *a, const wchar_t *b);
13
14 int main()
15 {
16     date today;
17     date tomorrow = today + 1;   //= operator+(today, 1);
18
19     cout << ::min(10, 20) << "\n"                  //calls line 29
20         << ::min(3.14, 2.71) << "\n"               //calls line 34
21         << ::min(today, tomorrow) << "\n"          //calls line 39
22         << ::min("hello", "goodbye") << endl;      //calls line 44
23
24     wcout << ::min(L"hello", L"goodbye") << L"\n"; //calls line 50
25
26     return EXIT_SUCCESS;
27 }
28
29 int min(int a, int b)      //function definition
30 {
31     return b < a ? b : a;      //why not return a < b ? a : b;
32 }
33
34 double min(double a, double b)
35 {
36     return b < a ? b : a;
37 }
38
39 date min(date a, date b)   //should be passed and returned by reference
40 {
41     return b < a ? b : a;      //return operator<(b, a) ? b : a;
42 }
43
44 const char *min(const char *a, const char *b)
45 {
46     //return b < a ? b : a; would be wrong for this data type
```

the declaration `using ::min;` into the `main` function before line 19. This would let line 19 have an unadorned `min` without the need to say `std::cout`.

```
47     return strcmp(b, a) < 0 ? b : a;
48 }
49
50 const wchar_t *min(const wchar_t *a, const wchar_t *b)
51 {
52     return wcscmp(b, a) < 0 ? b : a;   //wide character string compare
53 }
```

```
10              Line 19 passes int arguments to line 29.
2.71            Line 20 passes double arguments to line 34.
4/8/2014        Line 21 passes date arguments to line 39.
goodbye         Line 22 passes const char * arguments to line 44.
goodbye         Line 24 passes const wchar_t * arguments to line 50.
```

Consolidate the repetition with a function template

Instead of writing the same function over and over, plugging in a different data type each time, we will write a single *function template* named `min`. Lines 29−34 on p. 637 are the definition of `min`; lines 8−9 are the declaration.

Line 19 passes `int` arguments to `min`. This causes the computer to behave as if we had pasted into the program a copy of the function definition in lines 30−34, and the function declaration in line 9, with each `T` changed to `int`. The pasted-in copy is called an *instantiation* of the template. An instantiation is also called an *implicit specialization,* as opposed to the ‘‘explicit specialization’’ on pp. 664−669.

The computer *deduces* that `T` should be changed to `int` because the `10` in line 19 is of type `int`. Lines 20 and 21 create and call other instantiations of the same template, this time with each `T` changed to `double` and `date`. The `T` (for ‘‘type’’) is our conventional placeholder for the name of the data type.* The dummy name could be longer than one character, but please keep it uppercase for visibility.

The template made the source code smaller and less repetitious, but it had no effect on the size and speed of the executable file. So for the time being, a template is merely a shorthand for the source code. This will begin to change on pp. 734−735.

The preamble and the arguments

The following line 29 is the *template preamble,* which always starts with the keyword `template`. The same preamble appears on the function declaration in line 9. The keyword also has an obscure secondary usage; see pp. 725−726. The <angle brackets> in a preamble and those in an `#include` directive have nothing to do with each other.

Lines 29 and 30 could be written on the same line. No newline or other whitespace is required between the non-alphanumeric token `>` and the following alphanumeric token `T` (p. 101). But please keep them on separate lines for legibility.

The preamble declares that `T` is a *template argument* standing for the name of a data type. Despite the keyword `class`, this type does not necessarily have to be a class. In line 19, for example, `T` stands for `int`; in 20, `T` stands for `double`. Newer versions of C++ sensibly let us use the keyword `typename` instead of `class`, but we stick with the latter out of habit.

In contradistinction to the `T` in lines 29 and 30, the `a` and `b` in line 30 are called the *function arguments.* The `10` and `20` in line 19 are the *actual function arguments;* the data type `int` is the *actual template argument.*

Not all template arguments stand for data types. Some will represent constant values (pp. 690−696) or ‘‘template classes’’ (pp. 696−702).

* We assume that the data type has a name. See p. 660 for one that doesn’t.

Template functions are second-class citizens

The functions in lines 36 and 42 are *non-template functions.* They are merely overloads, i.e., other functions that happen to have the same name.

The `"hello"` and `"goodbye"` in line 22 have a choice between an exact match with the non-template function in line 36, and an equally exact match with an instantiation of the template function in line 30 with the `T` changed to `const char *`. But template functions are second-class citizens, so line 22 picks the non-template function. The non-template function would also win out against the template function `std::min`, so the double colon in lines 22 and 24 is not needed.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/min/min2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cstring>
 4 #include <cwchar>
 5 #include "date.h"
 6 using namespace std;
 7
 8 template <class T>
 9 T min(T a, T b);   //function declaration
10
11 const char *min(const char *a, const char *b);
12 const wchar_t *min(const wchar_t *a, const wchar_t *b);
13
14 int main()
15 {
16     date today;
17     date tomorrow = today + 1;
18
19     cout << ::min(10, 20) << "\n"                //calls line 30
20         << ::min(3.14, 2.71) << "\n"             //calls line 30
21         << ::min(today, tomorrow) << "\n"        //calls line 30
22         << ::min("hello", "goodbye") << endl;    //calls line 36
23
24     wcout << ::min(L"hello", L"goodbye") << L"\n"; //calls line 42
25
26     return EXIT_SUCCESS;
27 }
28
29 template <class T>
30 T min(T a, T b)           //function definition
31 {
32     cout << "::min<T>\n";   //to see which function is called
33     return b < a ? b : a;
34 }
35
36 const char *min(const char *a, const char *b)
37 {
38     cout << "::min(const char *)\n";
39     return strcmp(b, a) < 0 ? b : a;
40 }
41
42 const wchar_t *min(const wchar_t *a, const wchar_t *b)
43 {
```

```
44     cout << "::min(const wchar_t *)\n";
45     return wcscmp(b, a) < 0 ? b : a;
46 }
```

```
::min<T>                  Line 19 passes int arguments to line 30, changing T to int.
10
::min<T>                  Line 20 passes double arguments to line 30.
2.71
::min<T>                  Line 21 passes date arguments to line 30.
4/8/2014
::min(const char *)       Line 22 passes const char * arguments to line 36.
goodbye
::min(const wchar_t *)    Line 24 passes const wchar_t * arguments to line 42.
goodbye
```

Combine the declaration and definition

The declaration and definition of a template function can be combined, in the following lines 8−12. If the function is small enough, it can also be inline.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/min/min3.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cstring>
 4 #include <cwchar>
 5 #include "date.h"
 6 using namespace std;
 7
 8 template <class T>     //function declaration and definition
 9 inline T min(T a, T b)
10 {
11     return b < a ? b : a;
12 }
13
14 inline const char *min(const char *a, const char *b)
15 {
16     return strcmp(b, a) < 0 ? b : a;
17 }
18
19 inline const wchar_t *min(const wchar_t *a, const wchar_t *b)
20 {
21     return wcscmp(b, a) < 0 ? b : a;
22 }
23
24 int main()
25 {
26     date today;
27     date tomorrow = today + 1;
28
29     cout << ::min(10, 20) << "\n"                //calls line 9
30         << ::min(3.14, 2.71) << "\n"             //calls line 9
31         << ::min(today, tomorrow) << "\n"        //calls line 9
32         << ::min("hello", "goodbye") << endl;    //calls line 14
33
34     wcout << ::min(L"hello", L"goodbye") << L"\n"; //calls line 19
35
36     return EXIT_SUCCESS;
37 }
```

```
10            Line 29 passes int arguments to line 9.
2.71          Line 30 passes double arguments to line 9.
4/8/2014      Line 31 passes date arguments to line 9.
goodbye       Line 32 passes const char * arguments to line 14.
goodbye       Line 34 passes const wchar_t * arguments to line 19.
```

To mention the `min` functions in more than one `.C` file of a program, the above lines 8−22 could be written in a header file. The header would also have to include lines 3, 4, and 6. Until now, only a static function or variable (p. 99) could be defined in a header included by more than one `.C` file of a program; a non-static would incur the ‘‘multiply defined’’ error message. But the definition of a template—as opposed to an instantiation—occupies no memory. Any template function, static or not, can be defined in a header.*

In fact, defining the template function in a header file is the only portable way to mention it in more than one `.C` file of a program. For example, the template function `sort`, the flagship function of the C++ Standard Library, is defined in the header file `<algorithm>`. For nonportable attempts attempt to declare a template function in a header and define it in a `.C` file, see `export` in pp. 677−678 and ‘‘explicit instantiation’’ in pp. 720−721.

A ‘‘copy constructible’’ data type

We have assumed that the `min` in the above line 9 will accept function arguments of any data type `T`, with the exception of `char *` and `wchar_t *` and their `const` equivalents. But a careful reading of the template reveals two restrictions. The data type `T` must be *copy constructible* and *less-than comparable.*

The arguments and return value of `min` are passed by value, so a call to it will compile only if the function arguments are of a data type that can be copied. `T` must be a built-in, a pointer, an enumeration, or a class for which no private or protected copy constructor has been declared. Our class `date`, for example, has always been copy constructible. Class `rabbit` lost its copy constructibility on p. 200, regained it on pp. 234−236, and lost it again on p. 468. The current `rabbit` cannot be passed to `min`, although a `rabbit *` can.

A ‘‘less-than comparable’’ data type

The `min` in the above line 9 also applies the `<` operator to its function arguments, so a call to it will compile only if the function arguments can be operands of an operator `<` that yields a `bool` or a data type convertible thereto (p. 62). `T` could be a built-in, a pointer, an enumeration, or a class with an `operator<` that is not a private or protected member function. Class `date` became less-than comparable when we equipped it with the `operator<` friend on p. 281. Class `wabbit` has never been less-than comparable. Surprisingly, class `obj` (pp. 179−180) is less-than comparable even though it has no `operator<`. The `operator int` in the following line 6 implicitly converts the two `obj`’s to `int`’s, which are then compared.

```
 1 #include "obj.h"
 2
 3     obj ob1 = 10;
 4     obj ob2 = 20;
 5
 6     if (ob1 < ob2) {    //if (ob1.operator int() < ob2.operator int()) {
```

* One exception: a template that is an ‘‘explicit specialization’’ will occupy memory. Written in a header file, it will have to be declared `static` or `inline` just like a non-template function. See pp. 664−669.

To qualify as less-than comparable, a data type will have to satisfy two additional requirements. See pp. 776−777.

Concepts and comments

A *concept* is a set of requirements that a template argument `T` must satisfy; examples are copy constructibility and less-than comparability. We say that classes `date` and `obj` are *models* of these concepts, but `rabbit` is not.

To avoid nasty surprises, each template should have a comment stating the concepts of which its `T` must be a model.

```
//Version 1 (of 3) of the comment on min.

//Return the minimum of a and b.
//Return a if neither is less than the other.
//T must be copy constructible and less-than comparable.
```

Our `min` template is under no obligation to work correctly, or even compile, when `T` is not a model of these two concepts. Note that the template does not require *equality comparability* (the ability to say `a == b`); the comment makes no claim about what happens when `a` and `b` are equal.

A function argument whose type is more than an unadorned T

A template has no way of telling what the `T` will stand for. A function argument of an unknown type `T` might be expensive—or impossible—to copy. To avoid any attempted copy, the arguments and return value of `min` should have been passed by reference. The previous version of this template was in line 9 of `min.C` on p. 638; the improved version is in the following line 8. The requirements in the comment have been relaxed.

```
 1 //Version 2 of the comment on min.
 2
 3 //Return the minimum of a and b.
 4 //Return a if neither is less than the other.
 5 //T must be less-than comparable.
 6
 7 template <class T>
 8 inline const T& min(const T& a, const T& b)
 9 {
10     return b < a ? b : a;
11 }
```

In some cases, a reasonable assumption can be made about the data type that `T` stands for. The standard library assumes that pass-by-value is possible and affordable for four kinds of arguments.

(1) An iterator is always passed by value (p. 759).

(2) A function object (an object that has an `operator()` member function) is always passed by value (p. 766).

(3) A ‘‘`difference_type`’’ (an integer that counts the elements in a container) is always passed by value (p. 809).

(4) A number is always passed by value to a ‘‘numeric algorithm’’ (p. 962).

All other function arguments of type `T` are passed by reference. In particular, the values of the elements in a container are always passed by reference.

The min algorithm in the C++ Standard Library

We did not have to write our own `min` template function; one has already been defined for us in the `<algorithm>` header file. An *algorithm* is a template function whose arguments are iterators, usually a pair of iterators; our first official example will appear on p. 759. `min` is therefore not an algorithm, but it had to go in *some* header file.

```
//Excerpt from <algorithm> (or from a file included thereby).

template <class T>
inline const T& min(const T& a, const T& b)
{
    return b < a ? b : a;
}
```

Everything in the C++ Standard Library has the last name `std`.* This includes objects such as `cout` (p. 20), classes such as `exception` (p. 628), and functions such as `min`. We must therefore call `min` by its full name `std::min`, or say `using namespace std;` before mentioning `min`. In some versions of Microsoft Visual C++, `min` is named `_cpp_min`.

Change T to the simplest data type.

Ou first `min` program had a template function and two non-template functions sharing the same name (p. 637). Our next program has three template functions sharing the same name, in lines 8, 11, and 20. As usual, the name can be overloaded because the arguments are different. But despite their common name, these are still three separate functions. We will not have a single template function consisting of multiple templates until we get to ‘‘explicit specialization’’ on pp. 664−669.

The `i` in line 34 has no choice of which `print` to call. Of the three functions, only the one in line 8 will accept a non-pointer.

The `p` in line 38 has two alternatives. It can call

(1) line 8 with `T` changed to `const int *`

(2) line 11 with `T` changed to `int`

C++ picks the function that will turn `T` into the simpler data type, so line 38 picks the `print` in line 11. This is fortunate because line 11, knowing that its function argument is a pointer, can do more with it than line 8 can do with its featureless `t`.

The `pp` in line 42 has three alternatives, of which it will pick the third. It can call

(1) line 8 with `T` changed to `const int *const *`

(2) line 11 with `T` changed to `const int *`

(3) line 20 with `T` changed to `int`

The `ppp` in line 46 has three alternatives. none of which is adequate for its three levels of indirection. It will pick the third. It can call

(1) line 8 with `T` changed to `const int *const *const *`

(2) line 11 with `T` changed to `const *const int *`

(3) line 20 with `T` changed to `const int *`

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/min/overload1.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
4
5 //T must be puttable (printable with <<) in the following templates.
```

* Except the macros: `EXIT_SUCCESS` in `<cstdlib>`, `INT_MIN` in `<climits>`, etc. A macro cannot have a last name.

```
 6
 7 template <class T>
 8 inline void print(const T& t) {cout << t;}   //accepts any type
 9
10 template <class T>
11 void print(const T *p)                    //accepts pointer to any non-void type
12 {
13     cout << p;
14     if (p != 0) {
15         cout << " -> " << *p;
16     }
17 }
18
19 template <class T>
20 void print(const T *const *pp) //accepts pointer to pointer to any non-void type
21 {
22     cout << pp;
23     if (pp != 0) {
24         cout << " -> " << *pp;
25         if (*pp != 0) {
26             cout << " -> " << **pp;
27         }
28     }
29 }
30
31 int main()
32 {
33     int i = 10;
34     print(i);
35     cout << "\n";
36
37     const int *p = &i;
38     print(p);
39     cout << "\n";
40
41     const int *const *pp = &p;
42     print(pp);
43     cout << "\n";
44
45     const int *const *const *ppp = &pp;
46     print(ppp);
47     cout << "\n";
48
49     return EXIT_SUCCESS;
50 }
```

The machine addresses will be different on each platform. On my platform, they are formatted in hex.

```
10                                       Line 34 calls 8 with T → int
0xffbff1b0 -> 10                         Line 38 calls 11 with T → int
0xffbff1ac -> 0xffbff1b0 -> 10           Line 42 calls 20 with T → int
0xffbff1a8 -> 0xffbff1ac -> 0xffbff1b0   Line 46 calls 20 with T → const int *
```

If we erase the `const T *` function in the above lines 10−17, the `p` in line 38 will settle for the `const T&` in line 8. The output of line 38 loses one of its levels.

```
10                                        Line 34 calls 8 with T → int.
0xffbff0d0                                Line 38 calls 8 with T → const int *.
0xffbff0cc -> 0xffbff0d0 -> 10            Line 42 calls 20 with T → int.
0xffbff0c8 -> 0xffbff0cc -> 0xffbff0d0    Line 46 calls 20 with T → const int *.
```

If we restore lines 10−17 and erase the `const T *const *` function in 19−29, the `pp` in line 42 and the `ppp` in line 46 will settle for the `const T *` in line 11. Their output loses one of its levels.

```
10                           Line 34 calls 8 with T → int.
0xffbff1f0 -> 10             Line 38 calls 11 with T → int.
0xffbff1ec -> 0xffbff1f0     Line 42 calls 11 with T → const int *.
0xffbff1e8 -> 0xffbff1ec     Line 46 calls 11 with T → const int *const *.
```

If we erase lines 10−17 and 19−29, all four calls will settle for line 8.

```
10                           Line 34 calls 8 with T → int.
0xffbff104                   Line 38 calls 8 with T → const int *.
0xffbff100                   Line 42 calls 8 with T → const int *const *.
0xffbff0fc                   Line 46 calls 8 with T → const int *const *const *.
```

A `const void *` passed to `print` will call the above line 11 with `T` changed to `void`. Line 15 will then fail to compile because a `void *` (be it `const` or non-`const`) cannot be dereferenced.

One instantiation can call another instantiation of the same template.

The intent of the above program was to illustrate how the computer decides which template function to call. With that out of the way, I am disappointed that it handled only two levels of indirection. Here is a program that can handle any number of levels with only the two templates in lines 8 and 11.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/min/overload2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 //T must be puttable (printable with <<) in the following templates.
 6
 7 template <class T>
 8 inline void print(const T& t) {cout << t;}
 9
10 template <class T>
11 void print(const T *p)
12 {
13     cout << p;
14     if (p != 0) {
15         cout << " -> ";
16         print(*p);   //call line 11 if *p is a pointer, line 8 otherwise
17     }
18 }
19
20 int main()
21 {
22     int i = 10;
23     print(i);
24     cout << "\n";
25
26     const int *p = &i;
27     print(p);
28     cout << "\n";
29
30     const int *const *pp = &p;
31     print(pp);
32     cout << "\n";
33
34     const int *const *const *ppp = &pp;
35     print(ppp);
36     cout << "\n";
37
38     return EXIT_SUCCESS;
39 }
```

Line 23 calls line 8 with `T` changed to `int`.

Line 27 calls line 11 with `T` changed to `int`. Then line 16 calls 8 with the `T` in 8 changed to `int`.

Line 31 calls line 11 with `T` changed to `const int *`. Then line 16 calls another instantiation of 11 with `T` changed to `int` (one fewer `const` and one fewer `*`). Then the line 16 in the second instantiation calls 8 with the `T` in 8 changed to `int`.

Line 35 calls line 11 with `T` changed to `const int *const *`. Then line 16 calls another instantiation of 11 with `T` changed to `const int *`. Then the line 16 in the second instantiation calls a third instantiation of 11 with `T` changed to `int`. Then the line 16 in the third instantiation calls 8 with the `T` in 8 changed to `int`.

```
10
0xffbff190 -> 10
0xffbff18c -> 0xffbff190 -> 10
0xffbff188 -> 0xffbff18c -> 0xffbff190 -> 10    Line 35: three levels!
```

The top-level const

To say more precisely what data type `T` changes into, we introduce the notion of the *top-level* `const`. It indicates that the value of the variable being declared never changes. Examples are underlined:

```
1     const int i = 10;
2     const int *const p = &i;
3     const int *const *const pp = &p;
```

When a function argument is not a reference, a top-level `const` in the data type of the actual function argument does not become part of the `T`. For example, the `const int i` in line 17 changes the `T` in line 6 to `int` without the top-level `const`. This allows the function to change the value of its local copy of `i`. Of course, the function cannot change the value of `i`: it was passed by value. See pp. 658−659 for a way to put the deleted `const` back in.

When a function argument is a reference, the entire data type of the actual function argument, including the top-level `const`, is incorporated into the `T`. For example, the `const int i` in line 21 changes the `T` in line 9 to `const int`, preventing the increment from compiling.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/min/toplevel.C`

```
1 #include <iostream>
2 #include <cstdlib>
```

```
using namespace std;

template <class T>
inline void value(T t) {cout << ++t << "\n";}

template <class T>
inline void reference(T& t) {cout << ++t << "\n";}

int main()
{
    const int i = 10;
    const int *const p = &i;
    const int *const *const pp = &p;

    value(i);        //change T to int
    value(p);        //change T to const int *
    value(pp);       //change T to const int *const *

    //reference(i);  //change T to const int                 won’t compile
    //reference(p);  //change T to const int *const          won’t compile
    //reference(pp); //change T to const int *const *const   won’t compile
    return EXIT_SUCCESS;
}
```

```
11
0xffbff0c4
0xffbff0c0
```

More than one template argument

We might want to plug more than one data type into a function:

```
 1 inline void print(int a, date b)
 2 {
 3     cout << a << " " << b << "\n";
 4 }
 5
 6 inline void print(char a, const char *b)
 7 {
 8     cout << a << " " << b << "\n";
 9 }
10
11 inline void print(bool a, bool b)
12 {
13     cout << a << " " << b << "\n";
14 }
```

To accommodate these, we can provide more than one template argument. Remember to write the keyword `class` (or `typename`) twice in line 17.

```
15 //T1 and T2 must be copy constructible and puttable.
16
17 template <class T1, class T2>
18 inline void print(T1 a, T2 b)
19 {
```

```
20     cout << a << " " << b << "\n";
21 }
```

The template function will accept the following arguments.

```
22     print(10, date(date::july, 4, 1776));
23     print('A', "hello");
24     print(true, false);
```

Better yet, pass the arguments by reference.

```
25 //T1 and T2 must be puttable.
26
27 template <class T1, class T2>
28 inline void print(const T1& a, const T2& b)
29 {
30     cout << a << " " << b << "\n";
31 }
```

A local variable of type T

The trio of functions in lines 6, 13, and 20 is another candidate for templatization. Each function contains a local variable `temp` whose type depends on that of the arguments.

The header file `<algorithm>` declares a `std::swap`. If one of the headers in lines 1−2 included this header, the double colon in line 31 would be necessary to get the program to compile and call our `swap`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/swap/swap1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 inline void swap(int *a, int *b)
 7 {
 8     const int temp = *a;   //initialization
 9     *a = *b;               //assignment
10     *b = temp;
11 }
12
13 inline void swap(double *a, double *b)
14 {
15     const double temp = *a;
16     *a = *b;
17     *b = temp;
18 }
19
20 inline void swap(date *a, date *b)
21 {
22     const date temp = *a;
23     *a = *b;
24     *b = temp;
25 }
26
27 int main()
```

```
28 {
29     int i = 10;
30     int j = 20;
31     ::swap(&i, &j);
32     cout << "i == " << i << ", j == " << j << "\n";
33
34     double d = 3.14;
35     double e = 2.17;
36     ::swap(&d, &e);
37     cout << "d == " << d << ", e == " << e << "\n";
38
39     date today;
40     date tomorrow = today + 1;
41     ::swap(&today, &tomorrow);
42     cout << "today == " << today << ", tomorrow == " << tomorrow << "\n";
43
44     return EXIT_SUCCESS;
45 }
```

```
i == 20, j == 10                                  lines 29−32
d == 2.17, e == 3.14                              lines 34−37
today == 4/9/2014, tomorrow == 4/8/2014           lines 39−42
```

We consolidate the three functions into the template function in the following line 10. To call it, line 21 needs ampersands.

The template argument `T` must be *assignable:* capable of being the left operand of the assignment operator `=`. It must be a built-in, pointer, enumeration, or an object of a class that has no private or protected `operator=` member function. `T` must also be non-`const`. For example, `int` is assignable but `const int` is not. Class `date` has always been assignable. Class `wabbit` has never been assignable: it always had the private `operator=` member function in line 25 of `wabbit.h` on p. 536.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/swap/swap2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 //Swap the values of a and b.
 7 //T must be copy constructable (line 12) and assignable (lines 13-14).
 8
 9 template <class T>
10 inline void swap(T *a, T *b)
11 {
12     const T temp = *a;
13     *a = *b;
14     *b = temp;
15 }
16
17 int main()
18 {
19     int i = 10;
20     int j = 20;
21     ::swap(&i, &j);              //change T to int
22     cout << "i == " << i << ", j == " << j << "\n";
23
24     double d = 3.14;
25     double e = 2.71;
26     ::swap(&d, &e);
27     cout << "d == " << d << ", e == " << e << "\n";
28
29     date today;
30     date tomorrow = today + 1;
31     ::swap(&today, &tomorrow);   //change T to date
32     cout << "today == " << today << ", tomorrow == " << tomorrow << "\n";
33
34     const char *p = "hello";
35     const char *q = "goodbye";
36     ::swap(&p, &q);              //change T to const char *
37     cout << "p == \"" << p << "\", q == \"" << q << "\"\n";
38
39     return EXIT_SUCCESS;
40 }
```

```
i == 20, j == 10                               lines 19−22
d == 2.71, e == 3.14                           lines 24−27
today == 4/9/2014, tomorrow == 4/8/2014        lines 29−32
p == "goodbye", q == "hello"                   lines 34−37
```

Program in the same style with all data types.

Since `temp` is a constant, the above line 12 has to initialize it in the declaration. Even if it were not a constant, we could have done the same thing.

```
1     T temp = *a;   //If T is a class, call the copy constructor.
```

We could have split the above line into two statements:

```
2     T temp;        //If T is a class, call the default constructor.
3     temp = *a;     //If T is a class, call operator=.
```

But don't split it. For most classes, the copy constructor is faster than the default constructor. In class `date`, for example, the copy constructor merely copies an integer or two, while the default constructor gets the current date from the operating system (pp. 142−143). To add insult to injury, the hard-won current date is then overwritten by the `operator=` in the above line 3. Please keep the original code in line 1.

Now suppose that `temp` and `*a` were integers. In this case there are no constructors, so we could split

```
4     int temp = *a;
```

into

```
5     int temp;
6     temp = *a;
```

with no loss of speed. But don't split it. If you program with the built-in types as if they were objects, there will be less to change when you templatize the code. See also p. 634.

The C++ Standard Library swap

Don't write your own `swap`: call the one in the C++ Standard Library. Since its arguments are references, the following line 11 needs no ampersands.

```
//Excerpt from <algorithm>.

template <class T>
inline void swap(T& a, T& b)        //non-const references
{
    const T temp = a;
    a = b;
    b = temp;
}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/swap/swap3.C`

```
#include <iostream>
#include <cstdlib>
#include <algorithm>   //for swap
using namespace std;

int main()
{
    int i = 10;
    int j = 20;

    swap(i, j);   //Change T to int.
    cout << "i == " << i << ", j == " << j << "\n";

    return EXIT_SUCCESS;
}
```

```
i == 20, j == 10
```

7.1.2 Propaganda: Templates vs. Macros

Before template functions were invented, people faked them with macros. Let's try it with our `min` and `swap` functions and see what goes wrong.

min as a macro

The `MIN` in the following line 5 is the textbook example of a macro gone bad. It appears that line 13 increments `i` and `j`, and then passes the incremented values to the macro. But that order is clearly impossible: the increments are performed at runtime, while the macro is "called" at compile time. In reality, what is passed to the macro are the tokens `++` and `i`, `++` and `j`, not the incremented values. One of the increments will be performed twice. We could predict which one, if we knew which variable was bigger.

Similarly, it appears that line 16 calls `f` and `g`, and then passes the return values to the macro. In reality, what is passed to the macro is tokens such as `f`, `(`, and `)`, not the return value of `f`. One of the functions will be called twice. We could predict which one, if we knew which one returned a bigger value the first time it was called.

Our `min` template is a function, immune to these macro vagaries. It will increment each variable once and call each function once.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/propaganda/min.C`

```
1 #include <iostream>
2 #include <cstdlib>
```

```
using namespace std;

#define MIN(a, b)   ((b) < (a) ? (b) : (a))
int f();
inline int g() {cout << "g returns 40\n"; return 40;}

int main()
{
    int i = 10;
    int j = 20;
    int m = MIN(++i, ++j);
    cout << "i == " << i << ", j == " << j << ", m == " << m << "\n";

    m = MIN(f(), g());
    cout << "The minimum return value was " << m << ".\n";
    return EXIT_SUCCESS;
}

int f()
{
    static int n = 10;
    n += 20;
    cout << "f returns " << n << "\n";
    return n;
}
```

```
i == 12, j == 21, m == 12      i incremented twice, j once.
g returns 40     This line and the next might appear in the opposite order on some platforms.
f returns 30
f returns 50
The minimum return value was 50.
```

swap as a macro

`swap` is much worse. As a macro it requires an extra argument, the `int` in line 11.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/propaganda/swap.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 #define SWAP(T, a, b)   {const T temp = (a); (a) = (b); (b) = temp;}
 6
 7 int main()
 8 {
 9     int i = 10;
10     int j = 20;
11     SWAP(int, i, j);           //will compile
12     cout << "i == " << i << ", j == " << j << "\n";
13
14     int temp = 30;
15     //SWAP(int, i, temp);      //won't compile
16
17      size_t a[] = {10, 20, 30};
18      size_t k = 2;
19      SWAP(size_t, k, a[k]);     //undefined behavior
20      cout << "k == " << k << ", a[2] == " << a[2] << "\n";
21
22      const bool b = true;
23      int x = 10, y = 20, z = 30;
24      SWAP(int, b ? x : y, z);   //swap x and z
25      cout << "x == " << x << ", y == " << y << ", z == " << z << "\n";
26
27      return EXIT_SUCCESS;
28  }
```

The above line 15 will be rewritten as follows. Since it attempts to assign to the `const temp`, it will not compile.

```
29      {const int temp = (i); (i) = (temp); (temp) = temp;}
```

The above line 19 will be rewritten as follows, attempting to store the value `2` into the non-existent array element `a[30]`. If we are lucky, the program will crash.

```
30      {const size_t temp = (k); (k) = (a[k]); (a[k]) = temp;}
```

On my platform, line 19 left `a[2]` unchanged.

```
i == 20, j == 10
k == 30, a[2] == 30          All behavior after line 19 is undefined.
x == 30, y == 20, z == 10
```

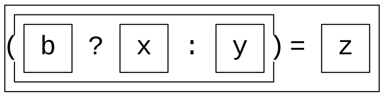
The preprocessor speaks a different language.

The following are some of the ways in which a macro conflicts with the rest of the language.

(1) Whitespace is permitted in C and C++ between any alphanumeric and nonalphanumeric token (p. 101)—except in exactly one place. There can be no whitespace in the above line 5 between the name `SWAP` and the left parenthesis (p. 97). With whitespace, the macro would have no arguments.

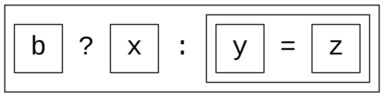
(2) In normal code, the parentheses around the `a`'s and `b`'s in the above line 5 would be unnecessary. But in a macro they are vital. For example, line 24 will be rewritten as the following line 31. Thanks to the parentheses, the middle `=` is always executed. When `b` is true, the middle `=` assigns a value to `x`.

```
31      {const int temp = (b ? x : y); (b ? x : y) = (z); (z) = temp;}
```



Without the parentheses in line 5, the middle `=` would be executed only when `b` is false. When it is executed, it would assign to `y`.

```
32      {const int temp = b ? x : y; b ? x : y = z; z = temp;}
```



With no parentheses in line 5, the output of lines 22−25 would change to the following.

```
x == 10, y == 20, z == 10
```

(3) A macro ignores the C++ scoping rules. A variable declared in a block can be mentioned only in that block (p. 32), but a macro `#define`’d in a block can be mentioned outside of it.

(4) We can take the address of a function, but not of a macro.

Macros should be used only for conditional compilation, such as the `#ifndef` that surrounds a header file, and as arguments and return values of C functions, such as the `EXIT_SUCCESS` argument of `exit` or the `EOF` returned by `getchar`. Every other macro should be replaced by a variable, an enumeration, or an inline function. For the macros `INT_MIN`, `INT_MAX`, and their cousins, see pp. 745−747.

7.1.3 Explicit Template Arguments

Our original `min` template function on p. 637 took function arguments of data type `T` and `T`, upgraded on p. 640 to `const T&` and `const T&`. In either case, the two actual arguments have to be of the same type, like the `i` and `j` in the following line 5. If the types differ, as in line 7, the computer will be unable to deduce what `T` should change into and the call will not compile. We encountered this problem, without explaining it, on pp. 43−44.

One workaround is to cast `i` to `double` in line 9. Now that both arguments are `double`, the computer can deduce that `T` is `double`. A more elegant solution is the *explicit template argument* in line 11. The `<double>` relieves the computer of the responsibility of deducing `T`. It specifies that `T` should be changed to `double`, regardless of the types of the actual arguments. When the instantiation is called, the `i` will be implicitly cast to `double`.

```
 1     int i = 10;
 2     int j = 20;
 3     double d = 3.14;
 4
 5     cout << ::min(i, j) << "\n";                       //will compile
 6
 7     cout << ::min(i, d) << "\n";                       //won’t compile
 8
 9     cout << ::min(static_cast<double>(i), d) << "\n"; //brute force
10
11     cout << ::min<double>(i, d) << "\n";               //elegant
```

The above line 7 has a surprising implication. The following class `wrapper` has an `operator int` in line 16. We would therefore expect that it could be used in any context that would accept an `int`, for example the place where the `w` is in line 31. But line 31 rejects the `w`. We can force it to compile with the explicit template arguments in lines 32 and 33, or by uncommenting exactly one of the functions in lines 23−27. For another example, see pp. 751−752.

```
12 class wrapper {
13     int i;
14 public:
15     wrapper(int initial_i): i(initial_i) {}
16     operator int() const {return i;}
17 };
18
19 template <class T>
20 inline const T& min(const T& a, const T& b) {return b < a ? b : a;}
21
22 /*
23 inline const int& min(const int& a, const int& b) {return b < a ? b : a;}
24
25 inline const wrapper& min(const wrapper& a, const wrapper& b) {
```

```
26     return b < a ? b : a;
27 }
28 */
29
30     wrapper w(10);
31     cout << ::min(20, w);            //won’t compile
32     cout << ::min<int>(20, w);       //will compile: convert w to int
33     cout << ::min<wrapper>(20, w);   //will compile: convert 20 to wrapper
```

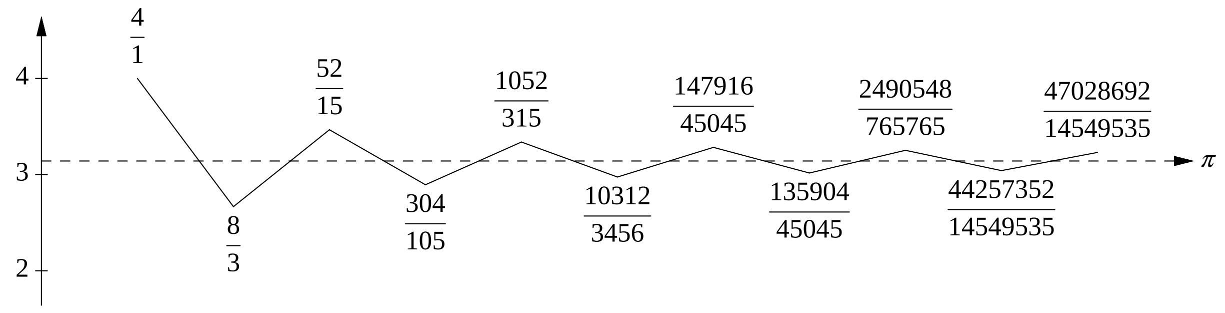
No function arguments

An explicit template argument is necessary whenever the computer cannot deduce `T` from the actual arguments. In the above lines 7 and 31, the actual arguments were contradictory. In the following line 13, they are nonexistent.

We will compute π with the Taylor series

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

As more and more terms are added, the sum zigzags in towards π. Line 44 divides the last term in half.



`pi_double` does its arithmetic with `double`’s. `pi_float` does its arithmetic with `float`’s. The functions need different names because they have no arguments. For the i/o manipulator `setprecision` in line 11, see pp. 355−356.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/explicit_argument/pi1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <iomanip>   //for setprecision
 4 using namespace std;
 5
 6 float pi_float();
 7 double pi_double();
 8
 9 int main()
10 {
11     cout << setprecision(19)
12         << "float       " << pi_float() << "\n"
13         << "double      " << pi_double() << "\n";
14
15     return EXIT_SUCCESS;
16 }
17
18 float pi_float()
```

```
{
    float pi = 0;
    float sign = 1;
    const long n = 1000000;

    for (long i = 1; i < n; i += 2) {
        pi += sign / i;
        sign = -sign;
    }

    pi += sign / (2 * n);
    return 4 * pi;
}

double pi_double()
{
    double pi = 0;
    double sign = 1;
    const long n = 1000000;

    for (long i = 1; i < n; i += 2) {
        pi += sign / i;
        sign = -sign;
    }

    pi += sign / (2 * n);
    return 4 * pi;
}
```

The digits that came out correctly on my machine are underlined. The `double` answer has seven more correct digits than the `float`. That's seven orders of magnitude, 10 million times more accurate.

`float`	`3.1415958404541015 62`	*correct to 6 significant digits: π = 3.141592653589793238...*
`double`	`3.141592653589691864`	*correct to 13 significant digits*

We can consolidate the functions into the template function in line 7. Since it has no function arguments, it must be called with the explicit template arguments in lines 25−27.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/explicit_argument/pi2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <iomanip>   //for setprecision
 4 using namespace std;
 5
 6 template <class T>
 7 T pi()
 8 {
 9     T pi_val = 0;
10     T sign = 1;
11     const long n = 1000000;
12
13     for (long i = 1; i < n; i += 2) {
14         pi_val += sign / i;
15         sign = -sign;
```

```
    }

    pi_val += sign / (2 * n);
    return 4 * pi_val;
}

int main()
{
    cout << setprecision(19)
        << "float       " << pi<float>() << "\n"
        << "double      " << pi<double>() << "\n"
        << "long double " << pi<long double>() << "\n";

    return EXIT_SUCCESS;
}
```

```
float       3.141595840454101562
double      3.141592653589691864
long double 3.141592653589793924     correct to 17 significant digits
```

Another template in which T cannot be deduced

My favorite function for peeking around in memory is in the following line 13. It displays the chunk of memory pointed to by `p` in the format specified by `T`.

To display a series of consecutive chunks, possibly in different formats, we provide the interface in line 23. The unusual `rpv` is a "reference to pointer to `void`", which will refer to the `p` in line 11 of `step.C` below. The *reference* is read/write, because no `const` follows the asterisk in line 23. This will allow line 26 to use `rpv` to change the value of the `p` in `step.C`. But the *pointer* is read-only, because of the `const` before the `void` in line 23. This will disallow `step` from using `rpv` to change the value of the variable `d` to which `p` is pointing.

The `rpt` in line 25 is a "reference to pointer to `T`". It refers to the `p` in `step.C`, even though that pointer is not a pointer to `T`. This *type punning* requires a `reinterpret_cast`; an earlier example was on p. 81.

`stand` and `step` will become member functions on p. 727.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/explicit_argument/step.h`

```
 1 #ifndef STEPH
 2 #define STEPH
 3 #include <iostream>
 4 using namespace std;
 5
 6 template <class T>
 7 inline void print(const T& t) {cout << t;}
 8
 9 inline void print(unsigned char c) {cout << static_cast<unsigned>(c);}
10 inline void print(const char *p) {cout << "\"" << p << "\"";}
11
12 template <class T>
13 const T& stand(const void *p)
14 {
15     cout << p << ": ";
16     const T& t = *static_cast<const T *>(p);
```

```
    print(t);
    cout << "\n";
    return t;
}

template <class T>
const T& step(const void *& rpv)   //non-const reference
{
    const T *& rpt = reinterpret_cast<const T *&>(rpv);
    return stand<T>(rpt++);
}
#endif
```

An application of step

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/explicit_argument/step.C`

```
#include <iostream>
#include <cstdlib>
#include "step.h"
using namespace std;

int main()
{
    double d = 3.14;
    const void *p = &d;

    step<unsigned char>(p);
    step<unsigned char>(p);
    step<unsigned short>(p);
    step<unsigned long>(p);

    return EXIT_SUCCESS;
}
```

On every machine, a `char` is by definition one byte. On my machine, `short` is two bytes, `long` is four, and `double` is eight. On my machine, a byte is eight bits.

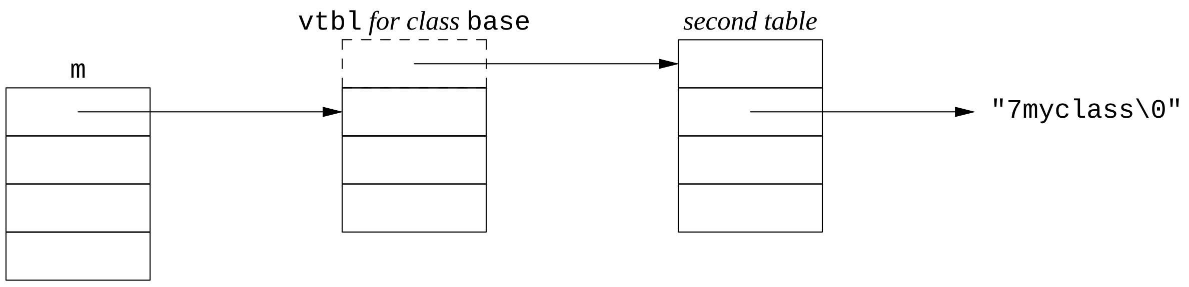
My machine represents a `double` as a sign bit (1 for negative, 0 for non-negative), an 11-bit exponent, and a 53-bit mantissa, in that order. 1022 is added to the exponent, so our exponent of 2 is stored as 1024 (binary `10000000000`). The mantissa of a non-zero number is always greater than or equal to ½ and less than 1, causing its first bit to always be 1. Since the first bit is always the same, it does not need to be stored in memory. In binary, our mantissa .785 is a fraction with 20 repeating bits.

$$\begin{aligned} 3.14 &= \frac{3.14}{4} \times 4 \\ &= .785 \times 2^2 \\ &= \mathtt{.110}\overline{\mathtt{01000111101011100001}} \times 2^{1024-1022} \end{aligned}$$

`0xffbff020:`	`64`	*sign bit (*`0`*), followed by first 7 bits of exponent (*`1000000`*)*
`0xffbff021:`	`9`	*last 4 bits of exponent (*`0000`*), followed by 1st 4 bits of mantissa (*`1001`*)*
`0xffbff022:`	`7864`	*next 16 bits of mantissa (*`0001111010111000`*)*
`0xffbff024:`	`1374389535`	*next 32 bits of mantissa (*`01010001111010111000010100011111`*)*

An application of stand

On my platform, each object with a virtual function begins with a pointer to element 0 of the virtual function table (vtbl) for that class. Element −1 of the vtbl is a pointer to a second table, whose element 1 is a pointer to the name of the class.



The following line 16 makes the rough-and-ready assumption that every element of the vtbl is a pointer to `void`. The pointer at the start of the object `m` is therefore a pointer to pointer to `void`, seen in the <angle brackets> in line 16.

(1) The call to `stand` in line 16 returns the address of element 0 of the vtbl. The subtraction computes the address of element −1 of the vtbl.

(2) The call in 17 returns the value of element −1 of the vtbl, which is the address of element 0 of the second table. The addition computes the address of element 1 of the second table.

(3) The call in 18 prints the value of element 1 of the second table, which is address of the first character of the class name.

(4) The call in 19 prints the characters being pointed to.

The name will be displayed by class `typeid` on p. 1015.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/explicit_argument/stand.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "step.h"
 4 using namespace std;
 5
 6 class myclass {
 7 public:
 8     virtual ˜myclass() {}
 9 };
10
11 int main()
12 {
13     myclass m;
14     const void *p = &m;
15
16     p = stand<const void *const *>(p) - 1;
17     p = stand<const void *const *>(p) + 1;
18     stand<const void *>(p);
19     stand<const char *>(p);
20
21     return EXIT_SUCCESS;
22 }
```

The name `myclass` has seven characters:

```
0xffbff070: 0x11480        ffbff070 is addr of m, 11480 is addr of vtbl[0].
0x1147c: 0x11494           1147c is addr of vtbl[-1], 11494 is addr of tab2[0].
0x11498: 0x11488           11498 is addr of tab2[1], 11488 is addr of '7'.
0x11498: "7myclass"
```

Override the deduction

An explicit template argument is also necessary when `T` could be deduced, but we want to override it ourselves. For example,

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/explicit_argument/override.C`

```
#include <iostream>
#include <cstdlib>
using namespace std;

template <class T>
inline void f(const T& t) {cout << t + 1 << "\n";}

int main()
{
    f(0);           //change T to int because 0 is int
    f<int *>(0);    //change T to int *
    f<double *>(0); //change T to double *
    return EXIT_SUCCESS;
}
```

```
1
0x4          On my machine, sizeof (int) == 4
0x8          On my machine, sizeof (double) == 8
```

An explicit template argument can prevent the loss of the top-level `const` in p. 644. The `T().f()` in the following line 12 constructs an anonymous object of type `T` and calls its member function `f`. The class `flavor` in line 5 has two `f`'s, showing us whether or not the object they belong to is `const`. For another class with `const` and non-`const` versions of the same member function, see p. 314.

Since the function argument `t` in line 12 is not a reference, the `const flavor s` in line 24 changes `T` into `flavor` without the top-level `const` To change `T` into `const flavor`, we can use need the explicit template argument in line 28.

Since the function argument `t` in line 15 is a reference, the `const flavor s` in line 25 changes `T` into `const flavor` with the top-level `const` intact (p. 644).

The explicit template argument in line 30 changes the `T` in 18 to `const flavor`. This results in a `t` that is formally of data type `const const flavor&`, but the redundant `const` is ignored.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/explicit_argument/explicit.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class flavor {
 6 public:
 7     void f() {cout << "non-const";}
 8     void f() const {cout << "const";}
 9 };
10
11 template <class T>
12 void value(T t) {t.f(); cout << "\t"; T().f(); cout << "\n";}
13
14 template <class T>
15 void reference(T& t) {t.f(); cout << "\t"; T().f(); cout << "\n";}
16
17 template <class T>
18 void const_reference(const T& t) {t.f(); cout << "\t"; T().f(); cout << "\n";}
19
20 int main()
21 {
22     const flavor s = flavor();
23
24     value(s);
25     reference(s);
26     const_reference(s);
27
28     value<const flavor>(s);
29     reference<const flavor>(s);
30     const_reference<const flavor>(s);
31
32     return EXIT_SUCCESS;
33 }
```

```
non-const   non-const   Line 24 calls 12 with T → flavor
const       const       Line 25 calls 15 with T → const flavor
const       non-const   Line 26 calls 18 with T → flavor
const       const       Line 28 calls 12 with T → const flavor
const       const       Line 29 calls 15 with T → const flavor
const       const       Line 30 calls 18 with T → const flavor
```

Apply an explicit template argument to an operator function

An `operator` function such as `operator==` can be a template function. An explicit template argument can be applied with the syntax in line 2.

```
1     if (a ==<int> b) {              //won't compile
2     if (operator==<int>(a, b) {     //will compile, needs no whitespace
3
4     if (operator< <int>(a, b) {     //needs whitespace; see p. 101
```

The C++ cast syntax

C++ has four operators for type conversion:

```
     static_cast
      const_cast
reinterpret_cast
    dynamic_cast
```

Their `<>()` syntax is borrowed from that of an explicit template argument.

```
1     double d = 3.14;
```

```

    cout << step<unsigned char>(&d) << "\n"
        << static_cast<int>(d) << "\n";
```

7.1.4 The Built-ins have Constructors too

Classes are not the only data types that have constructors and destructors. The built-in types, pointers, and enumerations have them too. But the special syntax that calls their constructors and destructors should be used only in a template.

Default constructor

Our `swap` function contained a local variable of type `T`; in line 12 of `swap2.C` on p. 647 we gave it an explicit initial value. But even with no explicit initialization, we usually want a local variable in a template to be born with a sane value.

This brings us face to face with an inconsistency in the syntax of C++. For an object, a definition with no explicit initial value will call the default constructor. See pp. 134−135.

```
 1     date d;                          //d is initialized.
 2     date *const pd = new date;       //The anonymous object is initialized.
```

But for a built-in, pointer, or enumeration, the same syntax will leave the variable full of garbage. See pp. 396−397.

```
 3     int i;                           //i is uninitialized.
 4     int *const pi = new int;         //The anonymous int is uninitialized.
```

Fortunately, the latter types have a default constructor that puts a zero into the newborn variable. This constructor can be called with the following syntax.

```
 5     int i = int();                   //i is initialized to 0.
 6     int *const pi = new int();       //The int is initialized to 0.
```

The same syntax will call the default constructor for an object. Lines 7−8 behave the same as the above lines 1−2, assuming that the compiler is smart enough to elide the temporary in line 7 (p. 137).

```
 7     date d = date();                 //d is initialized.
 8     date *const pd = new date();     //The anonymous object is initialized.
```

We have used this syntax in lines 7, 10, and 11 below. The same constructor, at the end of 17, will create an anonymous temporary.

But never use this syntax outside of a template. In place of the above lines 5−8, it is clearer, and just as fast, to say

```
 9     int i = 0;
10     int *const pi = new int(0);
11     date d;
12     date *const pd = new date;
```

Incidentally, line 33 would not compile without the name `stooge` in line 24. There would be no word for `T` to change into.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/default_constructor/default.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 template <class T>
 7 void f(T t1 = T())   //call default constructor if actual argument missing
 8 {
 9     //t1, t2, and *p will be initialized for every data type T
10     T t2 = T();                   //call default constructor
11     const T *const p = new T();   //call default constructor
12
13     //t3 and *q will be uninitialized if T is built-in, pointer, enumeration
14     T t3;                         //might not call any constructor at all
15     const T *const q = new T;     //might not call any constructor at all
16
17     cout << t1 << "\t" << t2 << "\t" << *p << "\t" << T() << "\t"
18         << t3 << "\t" << *q << "\n";
19
20     delete q;
21     delete p;
22 }
23
24 enum stooge {moe, larry, curly};
25
26 int main()
27 {
28     int i = 10;
29
30     f(date(date::december, 31, 2014));
31     f(i);
32     f(&i);
33     f(curly);
34
35     return EXIT_SUCCESS;
36 }
```

Line 30 outputs six sane values. The other lines have garbage in their last two values.

```
12/31/2014 4/8/2014   4/8/2014   4/8/2014   4/8/2014   4/8/2014   l. 30: T→date
10         0          0          0          4          157904     31: T→int
0xffbff0e8 0          0          0          0x4        0x268d0    32: T→int *
2          0          0          0          4          157904     33: T→stooge
```

Copy constructor

The built-in types, pointers, and enumerations also have copy constructors. Once again, the intent is to make them syntactically compatible with objects.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/default_constructor/copy.C`

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

template <class T>
void f(T t1)                        //call copy constructor
{
 9     T t2(t1);                         //call copy constructor
10     const T *const p = new T(t2);   //call copy constructor
11     cout << t1 << "\t" << t2 << "\t" << *p << "\t" << T(t2) << "\n";
12     delete p;
13 }
14
15 enum stooge {moe, larry, curly};
16
17 int main()
18 {
19     int i = 10;
20
21     f(date(date::december, 31, 2014));
22     f(i);
23     f(&i);
24     f(curly);
25
26     return EXIT_SUCCESS;
27 }
```

```
12/31/2014   12/31/2014   12/31/2014   12/31/2014   Line 21: T → date
10           10           10           10           Line 22: T → int
0xffbff168   0xffbff168   0xffbff168   0xffbff168   Line 23: T → int *
2            2            2            2            Line 24: T → stooge
```

The copy constructor can be used as a shorthand for a conversion (line 4). But don't do this: it's hard to search for.

```
1     double d = 3.14;
2
3     cout << static_cast<int>(d) << "\n"      //easy to find
4         << int(d) << "\n"                    //copy constructor: hard to find
5         << (int)d << "\n";                   //C-style cast: hard to find
```

Destructor

Finally, the built-in types, pointers, and enumerations have destructors which do nothing. On the rare occasions when a destructor is called explicitly (line 12) we can therefore safely assume that any type `T` has a destructor. For the placement syntax in line 10 and the explicit call to the destructor in line 12, see p. 406.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/default_constructor/destructor.C`

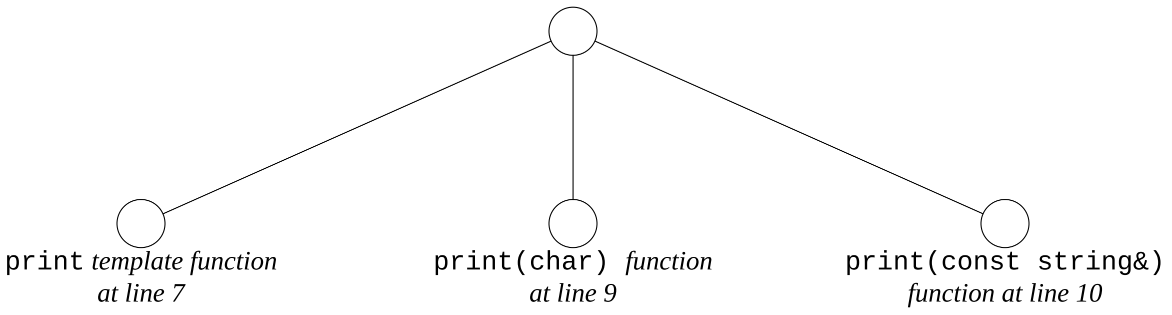
```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 template <class T>
 7 void f(const T& t)
 8 {
 9     T *const p = reinterpret_cast<T *>(new char[sizeof (T)]);
10     new(p) T(t);   //call the copy constructor
11     cout << *p << "\n";
12     p->~T();       //call the destructor
13     delete[] reinterpret_cast<char *>(p);
14 }
15
16 enum stooge {moe, larry, curly};
17
18 int main()
19 {
20     int i = 10;
21
22     f(date(date::december, 31, 2014));
23     f(i);
24     f(&i);
25     f(curly);
26
27     return EXIT_SUCCESS;
28 }
```

```
12/31/2014   Line 22: T → date
10           Line 23: T → int
0xffbff090   Line 24: T → int *
2            Line 25: T → stooge
```

7.1.5 Explicit Specialization of a Template Function

A template function and two non-template functions

Line 7 will print a value of almost any data type. Lines 9 and 10 are alternative code for types that require special handling. Since their arguments are different, the functions can be overloads of the same name. Any call to a function named `print` will have to make a three-way choice.


print template function at line 7
print(char) function at line 9
print(const string&) function at line 10

The `char c` in line 9 is fast enough to pass by value; the object `s` in 10 is not. The `t` in line 7 is unknown, so we pass it by reference just in case.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/explicit_specialization/overload.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <string>   //for class string
4 using namespace std;
5
6 template <class T>
7 inline void print(const T& t) {cout << t << "\n";}
8
9 inline void print(char c) {cout << "’" << c << "’\n";}
```

```
10 inline void print(const string& s) {cout << "\"" << s << "\"\n";}
11
12 int main()
13 {
14     int i = 10;
15     char c = 'A';
16     string s = "hello";
17
18     print(i);
19     print(c);
20     print(s);
21
22     return EXIT_SUCCESS;
23 }
```

`10`	*Line 18 calls line 7 with* `T` → `int`
`'A'`	*Line 19 calls line 9.*
`"hello"`	*Line 20 calls line 10.*

A template function consisting of three templates

Another way to invoke alternative code for an exceptional data type is with an ‘‘explicit specialization’’. Will demonstrate its quirks and then recommend that it should be used only when necessary.

The following line 7 is a *general-purpose* template. Lines 10 and 13 are *explicit specializations* of line 7. The `<char>` in line 10 says that whenever we decide to instantiate line 7 with `T` changed to `char`, we should instantiate 10 instead.

An explicit specialization must always follow the definition, or at least a declaration, of the general-purpose template. An explicit specialization has no `T`; its preamble (line 9) is always empty. Thus if any template argument is explicitly specialized, they all must be. (This will come back to haunt us on p. 709.)

In this example, the `<char>` in line 10 is redundant and can be removed. If the `const char&` in line 10 is indeed a special case of the `const T&` line 7, the computer can figure out that the `char` in 10 corresponds to the `T` in 7. We can remove the entire `<char>` or just the word `char`. Similarly, the `<string>` in line 13 can be removed. For an example where the `<char>` or `<string>` are needed, see p. 668.

I regret that the `char` has to be passed by reference in line 10. For a small built-in data type, pass by value would be faster because it avoids the extra fetch from memory. But line 10 will compile only if it is a special case of line 7, and line 7 is a pass-by-reference. If we change the function argument in 10 to an unadorned `char c`, we would have to match it by changing 7 to `T t` and 13 to `string s`.

What we really want is a general template for `const T&`, as in line 7, with an explicit specialization for `char` passed by value. We will be able to get this combination on pp. 779−781 when we have template classes as well as template functions.

Template function vs. function template

A note on nomenclature. A *function template* is a template that manufactures instantiations of a function. Later, a ‘‘class template’’ will manufacture instantiations of a class (p. 683).

The set of all possible instantiations of the following three `print` function templates is a *template function.* Included in this set are the `print` that takes an `int`, instantiated from line 7; the `print` that takes a `double`, also instantiated from line 7; the `print` that takes a `char`, instantiated from line 10; and the `print` that takes a `string`, instantiated from line 13. The set is indefinitely large because `T` could be any one of indefinitely many data types. Recall that a virtual function was also defined as a set of functions; see p. 488.

Lines 7, 10, and 13 represent a single template function that is written with three function templates. A template function must have exactly one general-purpose template, but it may have any number of explicit specializations.

Our `min` template function was a set of functions that differed only in the data type plugged into them. We were therefore able to instantiate that template function from one function template. But the following `print` template function is a set of functions that differ in other ways. This template function had to be instantiated from more than one function template.

Three kinds of specialization

An implicit specialization is the imaginary source code pasted into the program when a template is instantiated (p. 636). An explicit specialization is actual source code, as the following line 10. We will also see ‘‘partial’’ specializations (p. 702), but only for template classes, not functions.

The code in line 7 occupies no memory: it is not an instantiation. Even without the keywords `static` or `inline`, it could be written in a header file included by more than one `.C` file in the same program. But the code in lines 10 and 13 does occupy memory: these lines are instantiations. In a header file included by more than one `.C` file in the same program, they must be `static` or `inline`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/explicit_specialization/explicit.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <string>
 4 using namespace std;
 5
 6 template <class T>
 7 inline void print(const T& t) {cout << t << "\n";}
 8
 9 template <>
10 inline void print<char>(const char& c) {cout << "’" << c << "’\n";}
11
12 template <>
13 inline void print<string>(const string& s) {cout << "\"" << s << "\"\n";}
14
15 int main()
16 {
17     int i = 10;
18     char c = ’A’;
19     string s = "hello";
20
21     print(i);
22     print(c);
23     print(s);
24
25     return EXIT_SUCCESS;
26 }
```

```
10          Line 21 calls line 7 with T → int
’A’         Line 22 calls line 10.
"hello"     Line 23 calls line 13.
```

Binding is performed in two steps.

When writing a[n explicit] specialization, be careful about its location; or to make it compile will be such a trial as to kindle . . . self-immolation.

—The normally staid *C++ Standard*, §14.7.3, ¶ 7

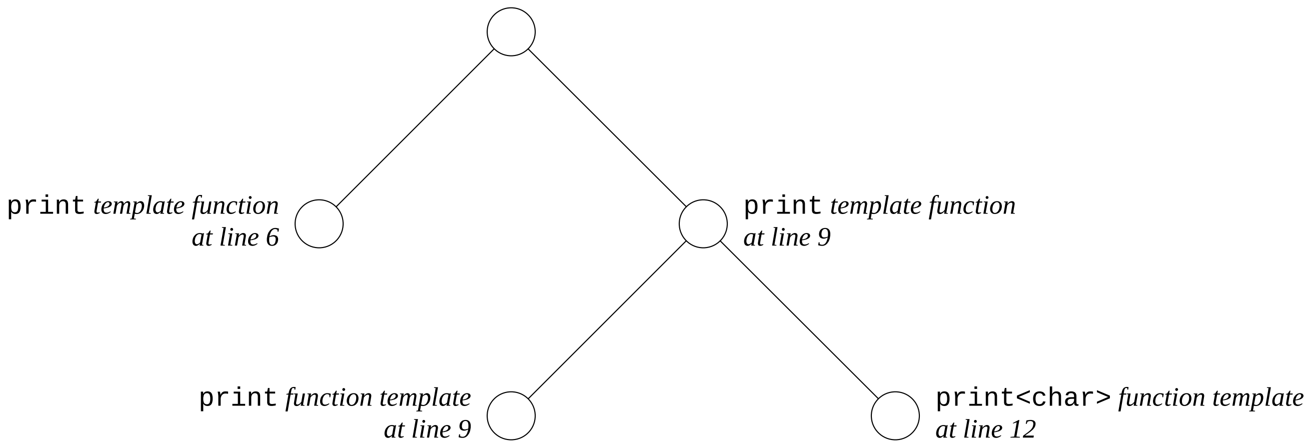
Line 6 is a template function that will print a value of almost any data type. Lines 9 and 12 constitute another template function that will print pointers. Since their arguments are different, the functions can be overloads of the same name.

The `<char>` in 12 is redundant and can be removed. The cast in line 9 ensures that only strings of `char`, not of `unsigned char` or `signed char`, are printed as strings. If the argument in line 9 is a pointer to a function, the cast will not compile; see line 24 of `reinterpret_cast.C` on p. 67.

Line 16 must decide which function to call. We say that it must *bind* the name `print` in line 16 to the function that is actually called. This decision is performed in two steps.

Step 1 chooses the template function. In this step, the explicit specializations are ignored. When line 16 calls `print`, only lines 6 and 9 are considered. Line 6 would change `T` into `const char *`; line 9 would change `T` into an unadorned `char`. Line 9 wins because it changes `T` into the simpler data type.

Step 2 chooses the function template within the template function. Now that we have decided to instantiate the template function whose general-purpose template is in line 9, lines 9 and 12 are compared. The latter is chosen.


print template function at line 6
print template function at line 9
print function template at line 9
print<char> function template at line 12

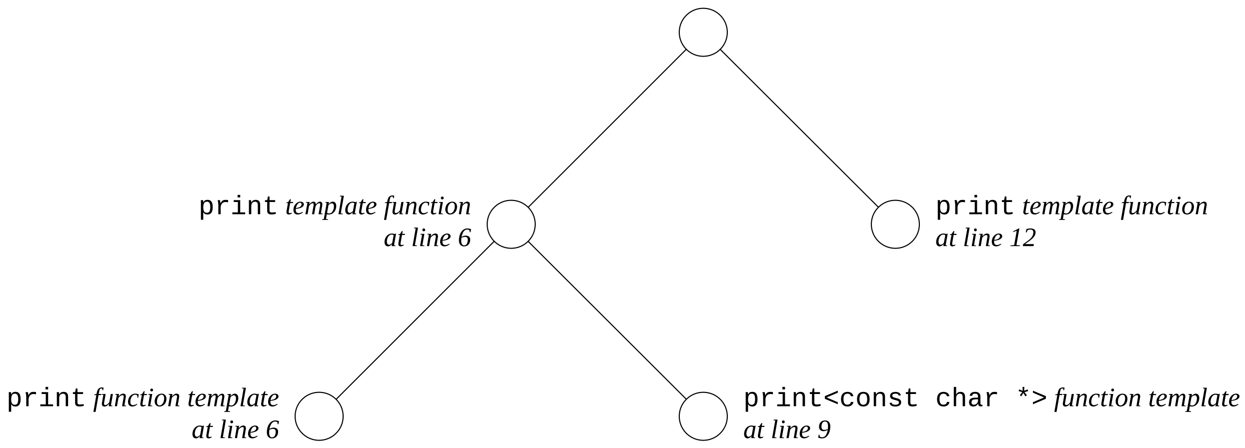
—On the Web at
`http://i5.nyu.edu/~mm64/book/src/explicit_specialization/immolate1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 template <class T>
 6 inline void print(const T& t) {cout << t;}
 7
 8 template <class T>
 9 inline void print(const T *p) {cout << static_cast<const void *>(p);}
10
11 template <>
12 inline void print<char>(const char *p) {cout << "\"" << p << "\"";}
13
14 int main()
15 {
16     print("hello");
17     cout << "\n";
18     return EXIT_SUCCESS;
19 }
```

`"hello"` *Line 16 calls the* `print<char>(const char *)` *in line 12 (good).*

The following program is almost the same. It seems that the `"hello"` in line 16 will once again call the explicit specialization in line 9. After all, the actual argument in 16 and the function argument in 9 are both `const char *`. But by placing the explicit specialization at line 9, we have made it an explicit specialization of line 6. An explicit specialization always belongs to the *previous* template of which it is a special case. Lines 6 and 9 now constitute one template function; line 12 consititutes another.

When binding the name `print` in line 16, step 1 will choose the `print` in line 12 over the one in line 6. The explicit specialization in line 9 is ignored. Step 2 find that line 16 has no explicit specializations.


print template function at line 6
print template function at line 12
print function template at line 6
print<const char *> function template at line 9

Since line 12 always prevents 9 from being called, it seems anticlimactic to remark that the `<const char *>` in 9 is redundant and can be removed.

The reader will have noticed that the `const T& t` in line 6 of the above `immolate1.C` was changed to the unadorned `T t` in the following line 6. We would prefer to pass the unknown `T` by reference. But the following line 9 will compile only if it is a special case of line 6. Had we continued to pass `t` by reference in line 6, the `p` in line 9 would have to be a `const char *const& p`, a reference to a pointer. Because of the `const` after the `*`, the reference could not be used to change the value of the pointer; because of the other `const`, the pointer could not be used to change the value of the `char`.

Could we pass `t` by reference and `p` by value? Can the binding be made independent of the order in which the templates are written? We will accomplish both on pp. 779−781, when template classes interact with template functions.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/explicit_specialization/immolate2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 template <class T>
 6 inline void print(T t) {cout << t;}
 7
 8 template <>
 9 inline void print<const char *>(const char *p) {cout << "\"" << p << "\"";}
10
```

```
11 template <class T>
12 inline void print(const T *p) {cout << static_cast<const void *>(p);}
13
14 int main()
15 {
16     print("hello");
17     cout << "\n";
18     return EXIT_SUCCESS;
19 }
```

`0x10e68` *Line 16 calls the* `print(const T *)` *in line 12 (bad).*

Why make an explicit specialization?

Let’s sum up the difficulties with explicit specialization.

(1) We cannot have a general-purpose template taking a `const T&` followed by a specialization taking a pass by value (p. 664).

(2) We can specialize for `int *` and `const int *` but not for `T *` and `const T *`. An explicit specialization never has any `T`.

(3) An explicit specialization can accidentally belong to the wrong general-purpose template (pp. 667−668).

In the light of these infirmities, please use function name overloading when possible, explicit specialization only when necessary. The latter is necessary only when the general-purpose template has no function argument with a `T` in its data type. The simplest example is the template function in the following lines 6−12, which has no function arguments at all.

The `<char>` in line 9 is required because there is no `T` in the data types of the function arguments or return value. Without the `<char>`, the computer could not tell which data type this is an explicit specialization for.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/explicit_specialization/name.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <string>
 4 using namespace std;
 5
 6 template <class T>
 7 string name();   //general-purpose template deliberately undefined
 8
 9 template <> inline string name<char>()   {return "char";}
10 template <> inline string name<int>()    {return "int";}
11 template <> inline string name<double>() {return "double";}
12 //etc.
13
14 template <class T>
15 inline void print(const T& t) {
16     cout << static_cast<const void *>(&t) << ": "
17         << name<T>() << " ("  << sizeof (T) << " bytes) " << t << "\n";
18 }
19
20 int main()
21 {
22     char c = ’A’;
23     int i = 10;
24     double d = 3.14;
25
26     print(c);
27     print(i);
28     print(d);
29
30     return EXIT_SUCCESS;
31 }
```

```
0xffbff187: char (1 bytes) A
0xffbff180: int (4 bytes) 10
0xffbff178: double (8 bytes) 3.14
```

Number of bytes is platform-dependent.

7.1.6 Pass a read/write pointer to a template function

Template functions and non-template functions follow different rules when we pass them a read/write pointer. Our example will be the pointer `p` in line 31.

Lines 5 and 6 are two non-template functions named `f`. The `int *` argument in line 5 is an exact match for the `int *` argument in line 33.

Lines 9, 12, and 15 are three template functions named `g`. Lines 9 or 12 are exact matches for the `int *` argument in line 34. Line 12 is selected because it changes `T` to the simpler data type. Note that the `const T *` line 15 does not match the `int *` in line 34: an `int` is not a `const T`.

Lines 18, 21, and 24 are one template function with two explicit specializations. Lines 18 or 21 are exact matches for the `int *` argument in line 35. Line 21 is selected because it changes `T` to the simpler data type. The `<int *>` and `<const int *>` in lines 21 and 24 are unnecessary. Note that the `const T *` line 24 does not match the `int *` in line 35: an `int` is not a `const T`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/explicit_specialization/pointer.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 inline void f(int *p) {cout << "f(int *) " << *p << "\n";}
 6 inline void f(const int *p) {cout << "f(const int *) " << *p << "\n";}
 7
 8 template <class T>
 9 inline void g(const T& t) {cout << "g(const T&) " << t << "\n";}
10
11 template <class T>
12 inline void g(T *p) {cout << "g(T *) " << *p << "\n";}
13
14 template <class T>
15 inline void g(const T *p) {cout << "g(const T *) " << *p << "\n";}
16
17 template <class T>
18 inline void h(T t) {cout << "h(T t) " << t << "\n";}
19
20 template <>
21 inline void h<int *>(int *p) {cout << "h(int *) " << *p << "\n";}
22
```

```
23 template <>
24 inline void h<const int *>(const int *p) {
25     cout << "h(const int *) " << *p << "\n";
26 }
27
28 int main()
29 {
30     int i = 10;
31     int *p = &i;
32
33     f(p);
34     g(p);
35     h(p);
36     return EXIT_SUCCESS;
37 }
```

```
f(int *) 10     Line 33 calls the f(int *) in line 5.
g(T *) 10       Line 34 calls the g(T *) in line 12 with T → int.
h(int *) 10     Line 35 calls the h<int *>(int *) in line 21 with T → int.
```

How do the template functions differ from the non-template functions? If we comment out the `int *` function in the above line 5, line 33 will happily settle for the `const int *` function in line 6. It has no trouble casting the `int *` in line 33 to `const int *`.

But if we comment out the `T *` template function in line 12, something unexpected happens. Line 34 will not call the `g(const T *)` in line 15; as explained above, 15 is not a match for 34 at all. Line 34 will call the lowest common denominator `g` line 9.

Similarly, if we comment out the `int *` explicit specialization in line 21, line 35 will not call the `h<const int *>(const int *)` in line 24. Line 35 will call the general function template in line 18.

Here is the output with 5, 12, and 21 commented out.

```
f(const int *) 10          Line 33 now calls line 6.
g(const T&) 0xffbff0d4     Line 34 now calls line 9 with T → int *.
h(T t) 0xffbff0d4          Line 35 now calls line 18 with T → int *.
```

The `t` in the above line 18 must be passed by value to permit the `p`'s in lines 21 and 24 to be passed by value. If the `t` in 18 were passed by reference, the pointers in 21 and 24 would also have to be passed by reference. For example, the `p` in 21 would have to become the "read-only reference to a read/write pointer to an `int`" in the following line 42.

```
38 template <class T>
39 inline void h(const T& t) {cout << "h(const T&) " << t << "\n";}
40
41 template <>
42 inline void h<int *>(int *const& p) {
43     cout << "h(int *const&) " << *p << "\n";
44 }
45
46 template <>
47 inline void h<const int *>(const int *const& p) {
48         cout << "h(const int *const&) " << *p << "\n";
49 }
```

What we really want is a general template for `const T&` with explicit specializations for pointers passed by value. We will be able to get this combination on pp. 779−781 when we have template classes as well as template functions.

7.1.7 `typename`

A human never stands so tall as when stooping to help a small computer.

—Infocom poster

Although no one does it, we could write parentheses in the following declaration.

```
 1     int  i;  //declare and define i
 2     int (i); //does same thing; parens make no difference (see footnote*)
```

If we didn't recognize that the `int` in the above line 2 is the name of a data type, we might think that the `int(i)` was a call to a function named `int`, with the argument `i`.†

This problem can occur in a template, in the following line 24. Because of the preamble in line 17, we know that `T` is the name of a data type. And because of the `x` member of `T` in line 24, we know even more: `T` is a data type that is a class. But what kind of member is `T::x`? If `T::x` is the name of a data type, like `clinton::hillary_t` in line 17 of `clinton.h` on p. 420, we will do the comment in line 20. If `T::x` is not the name of a data type, we will do lines 21 or 22−23. As we saw in the above line 2, the identity of the name in front of the parentheses can spell the difference between a declaration and a function call.

A similar ambiguity occurs in the following lines 28 and 32. It would seem that line 28 should be a declaration, since it would serve no purpose as a multiplication: the product would be discarded. But if either `T::y` or `p` were of a user-defined type, line 28 would call an `operator*` function which might do work that is not discarded.

The ambiguity even prevents the computer from deciding if line 32 should compile. If `T::z` were a static data member, we would perform a "bitwise and" or call an `operator&` function, and everything will be fine. But if `T::z` were the name of a data type, we would try to declare a reference `r` to a variable of that type. The reference, having no initializer, would fail to compile.

```
13 int i = 10;
14 int p = 20;
15 int r = 30;
16
17 template <class T>
18 void f()
```

* The parentheses would make a difference in the name of a compound data type. The simplest examples are the pairs in lines 4−5 and 9−10.

```
 3     inline int func() {return 10;}
 4     int  *f ();                      //a function that returns a pointer to an int
 5     int (*p)() = &func;              //a pointer to a function that returns an int
 6
 7     int i = 10, j = 20;
 8     int arr[2] = {10, 20};
 9     int  *a [2] = {&i, &j};          //an array of two pointers to int
10     int (*q)[2] = &arr;              //a pointer to an array of two int's
```

† Even if we did recognize `int` as a data type, the expression `int(i)` in some contexts could still be a function call. For example, the `int(i)` in the following line 12 would call the copy constructor for type `int` (pp. 661−662), with the argument `i`.

```
11     int i = 10;
12     int j = int(i);   //unnecessarily complicated way to say int j = 10;
```

The `int(i)` in the above line 2, however, is a declaration. Whenever a statement could be a declaration or merely a function call (in this case, a constructor call that creates an anonymous temporary), C++ always treats it as a declaration. For a painful example, see pp. 854−855.

```
{
    //Declare a local variable named i of data type T::x,
    //or pass the argument i to a static member function named T::x,
    //or pass the argument i to an operator() member function of a static
    //data member named T::x?
    T::x (i);

    //Declare a local variable named p of data type "pointer to T::y",
    //or multiply a static data member named T::y times p?
    T::y * p;

    //Declare a local variable named r of data type "reference to T::z",
    //or "bitwise and" a static data member named T::z with r?
    T::z & r;
}
```

The ambiguity is a very real problem. Although most class members are not the name of a data type, some of them are. The ones we have seen so far are listed below. `hillary_t` and `bill` appeared only once, just to illustrate the syntax; the others occur quite frequently. In fact, every container class in the standard library has five data type members named `iterator`, `const_iterator`, `value_type`, `size_type`, and `difference_type`.

(1) the `hillary_t` member of class `clinton` in line 17 of `clinton.h` on p. 420;

(2) the `bill` member of class `clinton` in lines 21−26 of `clinton.h` on p. 420;

(3) the `bill` member of class `gates` in lines 6−13 of `gates.h` on p. 421;

(4) the `value_type` member of class `stack` in line 3 on p. 423;

(5) the `_matrix_t` and `matrix_t` members of class `life` in lines 4 and 7 on pp. 423−424;

(6) the `difference_type` member of class `vector<int>` in line 12 on p. 434;

(7) the `size_type` member of class `vector<int>` in line 12 of `iterator.C` on p. 434;

(8) the `iterator` member of class `vector<int>` in line 26 of `iterator.C` on p. 434;

(9) the `const_iterator` member of class `vector<int>` in line 14 of `const_iterator.C` on p. 436;

(10) the `master_t` member of class `game` on p. 465;

(11) the `const_iterator` member of class `wabbit` on p. 578.

One possible resolution: T::x is the name of a data type

To resolve the ambiguity, a member of `T` is assumed to be the name of a data type only when it is preceded by the keyword `typename`. The `T::x` in the following line 14, for example, is now the name of a data type, and the line declares a variable of this type. The parentheses are unnecessary and serve only to confuse the issue. But even without them, the `typename` would still be necessary.

Another keyword that helps the computer understand a template is the `template` on pp. 725−726. Class `obj` was on pp. 179−180.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/typename/resolve1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "obj.h"
 4 using namespace std;
 5
 6 //T must be a class with public members x, y, z that are names of data types.
 7 //T::x and T::z must have default constructors.
 8 //Warning: i will be uninitialized if T::x is built-in, pointer, or enumeration.
 9
10 template <class T>
11 void f()
12 {
13     //Declare a local variable named i of data type T::x.
14     typename T::x(i);
15
16     //Declare a local variable named p of data type "pointer to T::y".
17     typename T::y *p;
18
19     //Declare a local variable named z1 of data type T::z.
20     typename T::z z1 = typename T::z();
21
22     //Declare a local variable named r of data type "reference to T::z".
23     typename T::z& r = z1;
24
25     //Use the local variables i, p, r that we just defined.
26     cout << &i << " " << &p << " " << &r << "\n";
27 }
28
29 class myclass {
30 public:
31     typedef obj x;
32     typedef int y;
33     typedef int z;
34 };
35
36 int main()
37 {
38     f<myclass>();
39     return EXIT_SUCCESS;
40 }
```

```
default construct 0      Line 14 constructs i.
0xffbff140 0xffbff13c 0xffbff138
destruct 0               Line 27 destructs i.
```

The other resolution: T::x is not the name of a data type

Without the `typename`, a member of `T` is assumed *not* to be the name of a data type. It must therefore be a data member, member function, or enumeration value. The `T::x` in the following line 22, for example, is not the name of a data type. When `T` is the `myclass` in line 33, line 22 calls the function `myclass::x`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/typename/resolve2.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
4
5 class detectable {};
6 inline void operator*(int i, detectable d) {cout << "multiply\n";}
7 inline void operator&(int i, detectable d) {cout << "bitwise and\n";}
```

```
 8
 9 int i = 10;
10 detectable p;
11 detectable r;
12
13 //T must be a class with a static member function x, or a static data
14 //member x with an operator() member function that takes one int argument.
15 //T must also have static data members y and z that are convertible to int.
16
17 template <class T>
18 void f()
19 {
20     //Pass the argument i to a static member function named T::x, or to an
21     //operator() member function of a static data member named T::x.
22     T::x(i);
23
24     //Multiply a static data member named T::y times p.
25     T::y * p;
26
27     //"Bitwise and" a static data member named T::z with r.
28     T::z & r;
29
30     //Did not create any local variables named i, p, r.
31 }
32
33 class myclass {
34 public:
35     static void x(int i) {cout << "myclass::x(" << i << ")\n";}
36     static const int y = 10;
37     static const int z = 20;
38 };
39
40 int main()
41 {
42     f<myclass>();
43     return EXIT_SUCCESS;
44 }
```

```
myclass::x(10)      Line 22 calls myclass::x in line 35.
multiply            Line 25 calls operator* in line 6.
bitwise and         Line 28 calls operator in line 7.
```

The keyword `typename`, used in this sense, is needed only within a template. (We saw its other use back on p. 636.) Another member needing `typename` will be the `vector<T>::const_iterator` in line 15 of `set.h` on p. 697 and line 60 of `wrapper.h` on p. 704.

A realistic example of typename

The function `print` in line 13 takes a vector, list, or other container, and prints each element. To tell line 15 that `CONTAINER::const_iterator` is the name of a data type, we write `typename` in front of it. Other data type members are in lines 18, 22 and 25.

A member that is not the name of a data type is the `CONTAINER::size` in line 23; we write no `typename` in front of it. `size` is a public member function. For simplicity I would have preferred a data member, but the standard library containers have none that are public, and rightly so. Line 23 takes the address of this member function; line 22 stores the address into a pointer `p` exquisitely engineered for this purpose. It is a pointer to a `const` member function of class `CONTAINER`, taking no arguments and returning a `CONTAINER::size_type`. Line 25 calls the member function indicated by `p`, belonging to the object `c`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/typename/typename.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include <list>
 5 #include "date.h"
 6 using namespace std;
 7
 8 //Print the elements of CONTAINER c.
 9 //CONTAINER must have the members const_iterator, begin, and end.
10 //The elements must be puttable.
11
12 template <class CONTAINER>
13 void print(const CONTAINER& c)
14 {
15     for (typename CONTAINER::const_iterator it = c.begin();
16         it != c.end(); ++it) {
17
18         const typename CONTAINER::value_type x = *it;
19         cout << x << "\n";
20     }
21
22     typename CONTAINER::size_type (CONTAINER::*p)() const =
23         &CONTAINER::size;
24
25     typename CONTAINER::size_type s = (c.*p)();
26     cout << "The container has " << s << " elements.\n\n";
27 }
28
29 int main()
30 {
31     const int a[] = {10, 20, 30};
32     const size_t n = sizeof a / sizeof a[0];
33     list<int> li(a, a + n);
34     print(li);
35
36     const date d[] = {
37         date(date::july,       4, 1776),
38         date(date::october,   29, 1929),
39         date(date::december,   7, 1941),
40         date(date::july,      20, 1969),
41         date(date::september, 11, 2001)
42     };
43     const size_t dn = sizeof d / sizeof d[0];
44     vector<date> v(d, d + dn);
45     print(v);
46
47     return EXIT_SUCCESS;
48 }
```

The above lines 18−19 may be combined to

```
49          cout << *it << "\n";
```

The above lines 22−25 may be combined to

```
50          typename CONTAINER::size_type s = c.size();
```

```
10                  Line 34 prints li.
20
30
The container has 3 elements.

7/4/1776            Line 45 prints v.
10/29/1929
12/7/1941
7/20/1969
9/11/2001
The container has 5 elements.
```

With no `typename` in the above line 15, the program will not compile. The error messages eventually get to the point.

```
typename.C: In function ’void print(const CONTAINER&)’:
typename.C:15:7: error: need ’typename’ before ’CONTAINER::
const_iterator’ because ’CONTAINER’ is a dependent scope
typename.C:15:33: error: expected ’;’ before ’it’
typename.C:16:3: error: ’it’ was not declared in this scope
typename.C: In function ’void print(const CONTAINER&) [with CONTAINER =
std::list<int>]’:
typename.C:34:10:   instantiated from here
typename.C:15:47: error: dependent-name ’CONTAINER:: const_iterator’ is
parsed as a non-type, but instantiation yields a type
```

The `print` function in line 13 of `typename.C` on still has two limitations.

(1) It was hardwired to print every element of the container. We might want to print only some of them.

(2) The function argument of `print` had to be an object of a class satisfying the requirements in the above line 9. For example, the array in line 31 could not have been passed to `print`.

These problems will be solved on pp. 757−760 when the function arguments of `print` become a pair of iterators.

▼ Homework 7.1.7a: consolidate the repetition with a template function

We illustrated a “thunk” on pp. 547−548. The `main.C` file there contained three identical chunks of code differing only by a data type (lines 10−19, 22−31, 34−42). We had to write the same chunk over and over because until now we had no way of passing a data type to a function.

Consolidate the repetion with the following template function. This will also demonstrate why we gave each `layout` class a first name and last name (`father::layout`) rather than a compound-word name (`father_layout`).

There will be two complications:

(1) Lines 10 and 11 will need the keyword `typename` to tell the computer that `T::layout` is the name of a data type.

(2) For the time being, you will have to pass the name of the data type as the function argument `name` in line 8. We will eliminate this on p. 1017 when we have Runtime Type Identification.

```
 1 /*
 2 T must be father or a class derived therefrom.  T must also have a data
 3 type member layout containing a pointer v to a structure containing a
 4 pointer f to a function taking a pointer to T and returning void.
 5 */
 6
 7 template <class T>
 8 void print(const char *name, const T *p)
 9 {
10     const typename T::layout& flay =
11         reinterpret_cast<const typename T::layout &>(*p);
12
13     //etc.
14     p->f();
15     flay.ptr_to_vtbl->ptr_to_f(p);   //low-level way to do the same thing
16     //etc.
17 }
18
19 int main()
20 {
21     father fath(10);
22     print<father>("father", &fath);
23
24     derived d(20, 30, 40);
25     print<derived>("derived", &d);
26     print<father>("father", &d);
27
28     return EXIT_SUCCESS;
29 }
```

▲

7.1.8 Export a template definition

Can a template function (or template class) be declared in a header file and defined in a `.C` file? We attempt to do so with the keyword `export` in line 4.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/export/f.h`

```
1 #ifndef FH
2 #define FH
3
4 export template <class T>
5 void f(const T& t);           //declaration
6 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/export/f.C`

```
1 #include <iostream>
2 #include "f.h"
3 using namespace std;
4
```

```
template <class T>
void f(const T& t) {           //definition
    cout << "f<T>(" << t << ")\n";
}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/export/main.C`

```
#include <cstdlib>
#include "f.h"
using namespace std;

int main()
{
    f(10);
    return EXIT_SUCCESS;
}
```

The GNU `g++` compiler says

```
In file included from main.C:2:0:
In file included from f.C:2:0:
f.h:4:1: warning: keyword 'export' not implemented, and will be ignored
Undefined            first referenced
 symbol                  in file
void f<int>(int const&)             /var/tmp//ccGDaWEz.o
ld: fatal: symbol referencing errors. No output written to /dev/null
collect2: ld returned 1 exit status
```

The Sun `CC` compiler says

```
main.C:
f<T>(10)
```

7.1.9 Point of definition vs. point of instantiation

The following lines 7−12 define a template function `f`. These lines are the *point of definition* for the template.

Line 18 calls the template function, pasting an instantiation of it into the program at some *point of instantiation.* Where is the point of instantiation, and does it matter? Line 18 is inside the `main` function. According to the C++ Standard (§14.6.4.1, ¶1), the point of instantiation should therefore be at line 21, immediately after the definition of `main`.

Armed with this terminology, we can introduce more terminology. The name `print` in line 10 is *independent* because its binding—the choice of which function the name refers to—has nothing to do with which data type the `T` stands for. An independent name is bound at the point of definition for the template. At this point, lines 7−12, the computer has seen only the `print(double)` in line 5, not the other `print`'s in 14 and 22. The name `print` in line 10 is bound to the function `print(double)` in line 5, and the `'A'` is converted to a `double`. (For binding a name to a function, see p. 666.)

The name `print` in line 11 is *dependent* because its binding does depend on which data type the `T` stands for. A dependent name is bound at the point of instantiation for the template, which should be line 21. At this point, the computer has seen the `print(double)` in 5 and the `print(int)` in 14. The name `print` in line 11 is therefore bound to the function `print(int)`, since this is the best match for the `t`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/instantiation/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 inline void print(double d) {cout << fixed << d << "\n";}
 6
 7 template <class T>
 8 void f(const T& t)
 9 {
10     print('A');   //This print is bound at f's point of definition.
11     print(t);     //This print is bound at f's point of instantiation.
12 }
13
14 inline void print(int i) {cout << i << "\n";}
15
16 int main()
17 {
18     f('A');
19     return EXIT_SUCCESS;
20 }
21
22 inline void print(char c) {cout << "'" << c << "'\n";}
```

The correct output is

```
65.000000    Line 10 calls line 5 (assume ASCII).
65           Line 11 calls line 14.
```

The `GNU g++` compiler incorrectly placed the point of instantiation at the end of the program, after line 22.

```
65.000000    Line 10 calls line 5.
65.000000    Line 11 calls line 22.
```

The `Sun CC` compiler version 5.11 options and the Microsoft Optimizing Compiler version 16.00.21003.01 are even farther from the Standard.

```
'A'          Line 10 calls line 22.
'A'          Line 11 calls line 22.
```

The moral is: do not scatter the declarations of the `print` functions all around the program. Place them together.

7.2 Template Classes

We often find ourselves writing the same class several times, plugging in a different data type each time. Container classes are the classic examples: a `vector` of `int`'s will be almost identical to a `vector` of objects. We can define the class once and for all as a ‘‘template class’’.

7.2.1 A Simple Example: class `stack`

A template class

Our class `stack`, seen first on pp. 149−154 and most recently on 503−505, was hardwired to store and retrieve only `int`'s. Here it is again, renamed `stack_int`. We provide one of everything you would want to see: a data type member (line 7), data members both static and non-static (lines 9 and 10), member functions both inline and non-inline (lines 13 and 14), a friend function (line 19), and a function that is neither a member nor a friend (line 22). The `==` in line 23 calls the `operator==` in line 19.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stack_template/stack_int.h`

```
 1 #ifndef STACK_INTH
 2 #define STACK_INTH
 3 #include <cstddef>   //for size_t
 4
 5 class stack_int {
 6 public:
 7     typedef int value_type;
 8 private:
 9     static const size_t max_size = 100;
10     value_type a[max_size];
11     size_t n;   //stack pointer: subscript of next free element
12 public:
13     stack_int(): n(0) {}
14     ~stack_int();
15
16     void push(value_type i);
17     value_type pop();
18
19     friend bool operator==(const stack_int& s1, const stack_int& s2);
20 };
21
22 inline bool operator!=(const stack_int& s1, const stack_int& s2) {
23     return !(s1 == s2);   //return !operator==(s1, s2);
24 }
25 #endif
```

The `value_type` in the following line 15 does not need the last name `stack_int`, but the one in line 28 does. See pp. 422−423.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stack_template/stack_int.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "stack_int.h"
 4 using namespace std;
 5
 6 stack_int::~stack_int()
 7 {
 8     if (n != 0) {
 9         cerr << "Warning: stack still contains " << n << " value(s).\n";
10     }
11 }
12
13 //Push a value onto the stack.
```

```

void stack_int::push(value_type i)
{
    if (n == max_size) {   //overflow
        cerr << "Can’t push when size " << n << " == capacity "
            << max_size << ".\n";
        exit(EXIT_FAILURE);
    }

    a[n++] = i;
}

//Pop a value off the stack.

stack_int::value_type stack_int::pop()
{
    if (n == 0) {          //underflow
        cerr << "Can’t pop when size " << n << " == 0.\n";
        exit(EXIT_FAILURE);
    }

    return a[--n];
}

bool operator==(const stack_int& s1, const stack_int& s2)
{
    if (s1.n != s2.n) {
        return false;
    }

    for (size_t i = 0; i < s1.n; ++i) {
        if (s1.a[i] != s2.a[i]) {
            return false;
        }
    }

    return true;
}
```

Here is the class renamed and modified to store and retrieve `double`’s. Note that the data members `max_size` and `n` remain `size_t`’s.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stack_template/stack_double.h`

```
#ifndef STACK_DOUBLEH
#define STACK_DOUBLEH
#include <cstddef>   //for size_t

class stack_double {
public:
    typedef double value_type;
private:
    static const size_t max_size = 100;
    value_type a[max_size];
    size_t n;   //stack pointer: subscript of next free element
public:
    stack_double(): n(0) {}
    ˜stack_double();

    void push(value_type d);
    value_type pop();

    friend bool operator==(const stack_double& s1, const stack_double& s2);
};

inline bool operator!=(const stack_double& s1, const stack_double& s2) {
    return !(s1 == s2);
}
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stack_template/stack_double.C`

```
#include <iostream>
#include <cstdlib>
#include "stack_double.h"
using namespace std;

stack_double::˜stack_double()
{
    if (n != 0) {
        cerr << "Warning: stack still contains " << n << " value(s).\n";
    }
}

//Push a value onto the stack.

void stack_double::push(value_type d)
{
    if (n == max_size) {   //overflow
        cerr << "Can’t push when size " << n << " == capacity "
            << max_size << ".\n";
        exit(EXIT_FAILURE);
    }

    a[n++] = d;
}

//Pop a value off the stack.

stack_double::value_type stack_double::pop()
{
    if (n == 0) {          //underflow
        cerr << "Can’t pop when size " << n << " == 0.\n";
        exit(EXIT_FAILURE);
    }

    return a[--n];
}

```

```
bool operator==(const stack_double& s1, const stack_double& s2)
{
    if (s1.n != s2.n) {
        return false;
    }

    for (size_t i = 0; i < s1.n; ++i) {
        if (s1.a[i] != s2.a[i]) {
            return false;
        }
    }

    return true;
}
```

The stacks cannot be consolidated with inheritance.

Can we consolidate the above two classes by deriving them from a common base class. But we can do this only if the functions constituting a virtual function agree in the data types of their arguments and return values.

```
//will not compile

class stack {
public:
    typedef ??? value_type;         //What data type would go here?

    virtual ~stack();
    virtual void push(value_type v);
    virtual value_type pop();
};

class stack_int: public stack {
public:
    typedef int value_type;

    void push(value_type i);
    value_type pop();
};

class stack_double: public stack {
public:
    typedef double value_type;

    void push(value_type d);
    value_type pop();
};
```

Consolidate the repetition with a class template

To consolidate the above classes, we can define the *template class* in the following line 14. The standard library already has a template class `stack` (pp. 155−157), but we will write our own.

For a non-template class, the class itself can be defined in a header file but the static data members and non-inline member functions must be defined in the corresponding `.C` file. For a template class, the class definition and all of its member definitions can go in the header file. As on p. 639, this is the only portable way to mention the class in more than one `.C` file of a program. The template class `vector`, for example, the flagship class of the C++ Standard Library, is defined in the header file `<vector>`.

Once again, we provide one of everything you would want to see: a data type member (line 16), data members both static and non-static (lines 18 and 20), static data members initialized both inside and outside the class definition (lines 18 and 19), member functions both inline and non-inline (lines 23 and 24), a friend function (line 29), and a function that is neither a member nor a friend (line 87).

The class definition begins with the template preamble in line 13. A member definition written outside the class definition requires the same preamble. For example, the member function `push` in line 46 and the static data member `x` in line 33 have the preambles in 45 and 32 respectively. But a declaration or definition *inside* the class definition must not have a copy of the class preamble. For example, the constructor in 23 and the destructor in 24 have no preambles of their own.

We can say `stack` instead of `stack<T>` within the `{`curly braces`}` of the class definition in lines 14−30. The `<T>` after each `stack` in lines 23, 24, and 29 is therefore unnecessary. We can also say `stack` instead of `stack<T>` within the definition of a member, from the double colon (line 36) to the end of the definition (line 41). The last `<T>` in lines 36 and 33 are therefore unnecessary. But everywhere else, the `<T>` in `stack<T>` is required. For example, the first `<T>` in lines 36 and 33 will have to remain. So will the `<T>`’s in lines 71 and 87, because `operator==` and `operator!=` are not members of class `stack<T>`.

We could have written `value_type` in place of the last `T` in line 46, and `stack<T>::value_type` in place of the first `T` in line 60. But `T` is more concise.

A friend of a template class

The first `<T>` in line 29 shows that `operator==` is a template function. (The `T` is optional, but the `<angle` brackets`>` must be written.) For each data type `T`, `operator==<T>` will be a friend of the corresponding class `stack<T>`. Thus `operator==<int>` will be a friend of `stack<int>`; `operator==<double>` will be a friend of `stack<double>`. This correspondence is called a *one-to-one* friendship; for others, see pp. 729−734.

A `<T>` can be applied to the name of a function only if the function was previously declared to be a template function. We must therefore write the template declaration in line 11 before the friend declaration in line 29. Unfortunately, line 11 can be only a declaration, not the definition, for `operator==`. The definition for `operator==` mentions some of the members of class `stack` (e.g., the `n` in line 73), so it must come after the definition for class `stack`.

A `<T>` can be applied to the name of a class only if the class was previously declared to be a template class. We must therefore write the template declaration in line 8 before mentioning `stack` in line 11. For other examples of forward declarations, see pp. 465−466.

Template class vs. class template

As on pp. 664−665, a *class template* is a template that manufactures instantiations of a class. The following lines 13−30 are a class template; lines 35−41 are a function template. Lines 32−33 must be a static data member template.

The set of all possible instantiations of a class template is a *template class.* A template class is not a data type. It is an indefinitely large set of data types: `stack<int>`, `stack<double>`, etc.

Our template class `stack` is a set of classes that differ only in the data type plugged into them. We were therefore able to instantiate the template class from a single class template. On pp. 702−707 we will see a set of classes that differ in other ways. We will have to instantiate this template class from more than one class template. The extra templates will be called ‘‘partial’’ and ‘‘explicit’’ specializations.

The double colons in lines 71 and 87 ensure that we’re talking about the `stack` that belongs to no namespace, not `std::stack`. They are needed in case the headers in lines 3 and 4 include the standard library header `<stack>`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stack_template/stack.h`

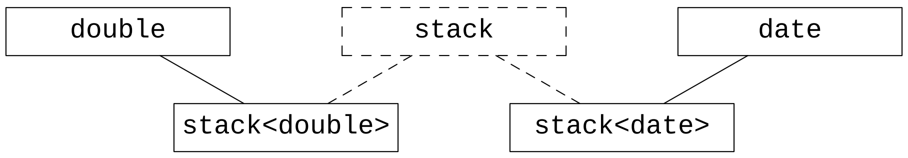
```
#ifndef STACKH
#define STACKH
#include <iostream> //iostream includes cstddef, which defines size_t
#include <cstdlib>
using namespace std;

template <class T>
class stack;        //forward declaration for a template class

template <class T>
bool operator==(const stack<T>& s1, const stack<T>& s2);

template <class T>
class stack {
public:
    typedef T value_type;
private:
    static const size_t max_size = 100;
    static size_t x;   //just to demonstrate the syntax
    T a[max_size];
    size_t n;          //stack pointer: subscript of next free element
public:
    stack<T>(): n(0) {}
    ˜stack<T>();

    void push(const T& t);
    T& pop();

    friend bool operator==<T>(const stack<T>& s1, const stack<T>& s2);
};

template <class T>
size_t stack<T>::x = sizeof (stack<T>);   //definition of static data member

template <class T>
stack<T>::˜stack<T>()
{
    if (n != 0) {
        cerr << "Warning: stack still contains " << n << " value(s).\n";
    }
}

//Push a value onto the stack.

template <class T>
void stack<T>::push(const T& t)
{
    if (n == max_size) {   //overflow
        cerr << "Can’t push when size " << n << " == capacity "
            << max_size << ".\n";
        exit(EXIT_FAILURE);
    }

    a[n++] = t;
}

//Pop a value off the stack.

template <class T>
T& stack<T>::pop()
{
    if (n == 0) {          //underflow
        cerr << "Can’t pop when size " << n << " == 0.\n";
        exit(EXIT_FAILURE);
    }

    return a[--n];
}

template <class T>
bool operator==(const ::stack<T>& s1, const ::stack<T>& s2)
{
    if (s1.n != s2.n) {
        return false;
    }

    for (size_t i = 0; i < s1.n; ++i) {
        if (s1.a[i] != s2.a[i]) {
            return false;
        }
    }

    return true;
}

template <class T>
inline bool operator!=(const ::stack<T>& s1, const ::stack<T>& s2) {
    return !(s1 == s2);
}
#endif
```

Create new data types

An instantiation of a template function usually requires no explicit template argument; the computer can deduce `T` from the function arguments. But an instantiation of a template class always requires an explicit template argument; examples are in the following lines 9, 16, and 23. Note that the data types `stack<double>` in 9 and `stack<date>` in 23 are not derived from a common base class and are not friends of each other.

We can now create many stack types with only one template class definition. The solid boxes in this diagram represent data types; the dashed box represents a template class. The dashed lines in this diagram represent instantiation. The solid lines mean “gets plugged into the <angle brackets> of”.

For the possibility of templates other than function templates and class templates, see pp. 706−707.



Lines 25−26 construct an object with a declaration and then insert it into a container. But an object mentioned only once should be an anonymous temporary, like the `date` in line 28.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stack_template/main.C`

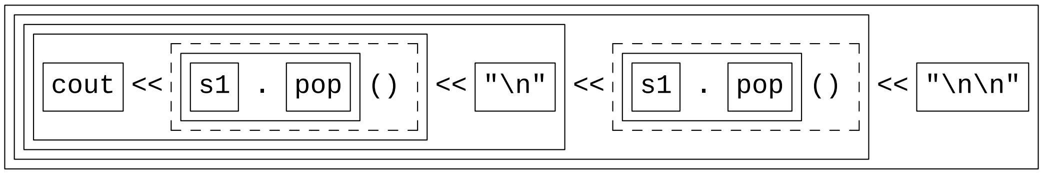
```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "stack.h"
 4 #include "date.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     ::stack<double> s1;
10     s1.push(2.71);   //e
11     s1.push(3.14);   //pi
12
13     cout << s1.pop() << "\n";
14     cout << s1.pop() << "\n\n";
15
16     ::stack<double> s2 = s1;          //copy constructor
17     if (s1 == s2) {                   //if (operator==(s1, s2)) {
18         cout << "They are equal.\n";
19     }
20
21     cout << "\n";
22
23     ::stack<date> s3;
24
25     date independence_day(date::july, 4, 1776);
26     s3.push(independence_day);
27
28     s3.push(date(date::october,   29, 1929));
29     s3.push(date(date::december,   7, 1941));
30     s3.push(date(date::july,      20, 1969));
31     s3.push(date(date::september, 11, 2001));
32
33     cout << s3.pop() << "\n";
34     cout << s3.pop() << "\n";
35     cout << s3.pop() << "\n";
36     cout << s3.pop() << "\n";
37     cout << s3.pop() << "\n";
38
39     return EXIT_SUCCESS;
40 }
```

```
3.14
2.71

They are equal.

9/11/2001
7/20/1969
12/7/1941
10/29/1929
7/4/1776
```

Warning: the above lines 13−14 cannot be combined to the following.

```
41      cout << s1.pop() << "\n"
42          << s1.pop() << "\n\n";
```

Had we done this, we could still predict that the `<<` operators will be executed from left to right. But we could not predict which call to `pop` would be executed first: neither of the dashed boxes contains the other. See pp. 14−16.



Hide the name of an instantiation of a template class

If you have a mother-in-law with only one eye and she has it in the center of her forehead, you don't keep her in the living room.

—Lyndon Baines Johnson, quoted in David Halberstam's *The Best and the Brightest*, Chapter 19

If you are uncomfortable with the <angle brackets> in the above line 9, you can hide them in a typedef.

```
 1 #include "stack.h"
 2 #include "date.h"
 3
 4     //From now on, stack_double_t means ::stack<double>.
 5     typedef ::stack<double> stack_double_t;
 6
 7     stack_double_t s1;   //means ::stack<double> s1;
 8
 9     //from now on, stack_date_t means stack<date>
10     typedef ::stack<date> stack_date_t;
11
12     stack_date_t s2;
```

Many common data types are actually typedefs for an instantiation of a template class. We have been using them without suspecting their true nature. Their full names will often appear in error messages.

```
13     //Excerpt from <string>.
14     typedef basic_string<char> string;

15     //Excerpts from <iostream>.
16     typedef basic_istream<char> istream;       //e.g., cin
17     typedef basic_ostream<char> ostream;       //e.g., cout, cerr, clog
```

```
18     //Excerpts from <fstream> for file i/o.
19     typedef basic_ifstream<char> ifstream;
20     typedef basic_ofstream<char> ofstream;
21     typedef basic_fstream<char> fstream;
```

The same header files contain a parallel series of typedefs for wide characters.

```
22     typedef basic_string<wchar_t> wstring;

23     typedef basic_istream<wchar_t> wistream;   //e.g., wcin
24     typedef basic_ostream<wchar_t> wostream;   //e.g., wcout, wcerr, wclog

25     typedef basic_ifstream<wchar_t> wifstream;
26     typedef basic_ofstream<wchar_t> wofstream;
27     typedef basic_fstream<wchar_t> wfstream;
```

Default value for a template argument

Let's make the `T` default to the data type `int` in the above template class `stack`. Change the preamble in line 13 of `stack.h` on p. 685 to the following. Do not change any of the other preambles.

```
 1 template <class T = int>
```

We can now create a stack of `int`'s as follows. Note that the `<`angle brackets`>` in line 4 are still required.

```
 2 #include "stack.h"
 3
 4     ::stack<> s1;                   //a stack of int's
 5     ::stack<int> s2;                //another stack of int's
 6     ::stack<double> s3;             //a stack of double's
 7     //::stack s4;                   //won't compile
```

Only a template class, not a template function, can take a default value for a template argument.

Nested instantiations need whitespace.

Instantiations can be nested. In other words, the name of an instantiation of a template class can be plugged into the `<`angle brackets`>` of another template. When we do this, we must always separate the closing `>`'s with whitespace. Whitespace is always needed between any consecutive tokens that would otherwise look like one big token. See p. 101.

The other template will usually be a template class:

```
 1 #include <vector>
 2 #include <list>
 3 #include <string>
 4 #include <complex>   //for class complex, p. 210
 5 using namespace std;
 6
 7     complex<double> c;                         //a complex number
 8     vector<complex<double> > v;                //a vector of complex numbers
 9     vector<list<string> > hs(100);             //a hash table of strings
10     vector<list<complex<double> > > hc(100); //a hash table of complex numbers
```

But the other template could also be a template function, such as the `step` on pp. 655−658. On my platform, a `vector` begins with a pointer to the first element. On every platform, the remaining elements are stored consecutively.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stack_template/nest.C`

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include <complex>
#include "step.h"
using namespace std;

int main()
{
    complex<double> a[] = {
        complex<double>(10, 20),
        complex<double>(30, 40),
        complex<double>(50, 60),
    };
    const size_t n = sizeof a / sizeof a[90];
    vector<complex<double> > v(a, a + n);
    const void *p = stand<const complex<double> *>(&v);

    step<complex<double> >(p);   //element 0
    step<complex<double> >(p);   //element 1
    step<complex<double> >(p);   //element 2
    return EXIT_SUCCESS;
}
```

```
0xffbff0f4: 0x22d68
0x22d68: (10,20)
0x22d78: (30,40)
0x22d88: (50,60)
```

7.2.2 Constant Template Arguments

We can plug any data type into the template class `stack`, but the maximum number of elements is still hardwired to 100. We will now parameterize this number with a *constant template argument.*

A constant template argument must be integral (p. 61), a pointer, or an enumeration, not a `double` or an object. For example, the constant template argument `MAX_SIZE` in line 13 is a `size_t`, which is a typedef for `unsigned` or `unsigned long`. We can give it a default value (100) because `stack` is a template class, not a template function.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/const_arg/stack.h`

```
 1 #ifndef STACKH
 2 #define STACKH
 3 #include <iostream>   //iostream includes cstddef, which defines size_t
 4 #include <cstdlib>
 5 using namespace std;
 6
 7 template <class T, size_t MAX_SIZE>
 8 class stack;
 9
10 template <class T, size_t MAX_SIZE>
11 bool operator==(const stack<T, MAX_SIZE>& s1, const stack<T, MAX_SIZE>& s2);
```

```

template <class T, size_t MAX_SIZE = 100> //Don’t write the = 100 anywhere else.
class stack {
public:
    typedef T value_type;
private:
    T a[MAX_SIZE];
    size_t n;   //stack pointer: subscript of next free element
public:
    stack(): n(0) {}
    ~stack();

    void push(const T& t);
    T& pop();

    friend bool operator==<T, MAX_SIZE>(const stack& s1, const stack& s2);
};

template <class T, size_t MAX_SIZE>
stack<T, MAX_SIZE>::~stack()
{
    if (n != 0) {
        cerr << "Warning: stack still contains " << n << " value(s).\n";
    }
}

//Push a value onto the stack.

template <class T, size_t MAX_SIZE>
void stack<T, MAX_SIZE>::push(const T& t)
{
    if (n == MAX_SIZE) {   //overflow
        cerr << "Can’t push when size " << n << " == capacity "
            << MAX_SIZE << ".\n";
        exit(EXIT_FAILURE);
    }

    a[n++] = t;
}

//Pop a value off the stack.

template <class T, size_t MAX_SIZE>
T& stack<T, MAX_SIZE>::pop()
{
    if (n == 0) {          //underflow
        cerr << "Can’t pop when size " << n << " <= 0.\n";
        exit(EXIT_FAILURE);
    }

    return a[--n];
}

template <class T, size_t MAX_SIZE>
```

```
bool operator==(const ::stack<T, MAX_SIZE>& s1, const ::stack<T, MAX_SIZE>& s2)
{
    if (s1.n != s2.n) {
        return false;
    }

    for (size_t i = 0; i < s1.n; ++i) {
        if (s1.a[i] != s2.a[i]) {
            return false;
        }
    }

    return true;
}

template <class T, size_t MAX_SIZE>
inline bool operator!=(const ::stack<T, MAX_SIZE>& s1,
                       const ::stack<T, MAX_SIZE>& s2) {
    return !(s1 == s2);
}
#endif
```

The value of a constant template argument must be a constant expression (p. 234), not a variable. An example is the `100` in line 9.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/const_arg/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "stack.h"
 4 #include "date.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     ::stack<double, 100> s1;          //could also say 50 + 50
10     ::stack<double> s2;               //same data type: 100 is the default
11
12     s1.push(2.71);   //e
13     s1.push(3.14);   //pi
14
15     cout << s1.pop() << "\n";
16     cout << s1.pop() << "\n\n";
17
18     ::stack<date, 4> s3;
19
20     s3.push(date(date::july,       4, 1776));
21     s3.push(date(date::october,   29, 1929));
22     s3.push(date(date::december,   7, 1941));
23     s3.push(date(date::july,      20, 1969));
24     s3.push(date(date::september, 11, 2001));   //will overflow the stack
25
26     cout << s3.pop() << "\n";
27     cout << s3.pop() << "\n";
28     cout << s3.pop() << "\n";
29     cout << s3.pop() << "\n";
30     cout << s3.pop() << "\n";
31 
32     return EXIT_SUCCESS;
33 }
```

```
3.14
2.71

Can’t push when size 4 == capacity 4.    caused by line 24 of main.C
```

Recognize the angle brackets

The default value of a constant template argument could be an expression.

```
template <class T, size_t MAX_SIZE = 10 + 20>
class stack {
```

But an expression containing the `>` operator must be enclosed in parentheses.

```
template <class T, bool B = (10 > 20)>
class stack {
```

For a real-world example, see p. 710. A similar use of parentheses is to enclose a comma operator in a function argument; see p. 264.

▼ Homework 7.2.2a: let the dimensions of the game of life be constant template arguments

We can give dimensions to an array. With constant template arguments, we can also give dimensions to a class. Change class `life` from

```
class life {
    static const size_t xmax = 10;
    static const size_t ymax = 10;
    typedef bool _matrix_t[ymax + 2][xmax + 2];   //array needs height first
    _matrix_t matrix;
public:
    typedef bool matrix_t[ymax][xmax];
```

to

```
template <size_t XMAX = 10, size_t YMAX = 10>     //users expect width first
class life {
    typedef bool _matrix_t[YMAX + 2][XMAX + 2];   //array needs height first
public:
    typedef bool matrix_t[YMAX][XMAX];
```

We can then construct games as follows.

```
    life<10, 20> g1 = argument for constructor;         //10 × 20 (width × height)
    life<30> g2 = argument for constructor;             //30 × 10
    life<> g3 = argument for constructor;               //10 × 10
```

▲

▼ Homework 7.2.2b:
Version 4.0 of the Rabbit Game: create the four rank classes with one template

The ranks in the food chain are represented by four classes, introduced on pp. 564−565 and last modified on p. 582.

```
inert
victim
predator
halogen
```

They are identical except for the levels of hunger and bitterness in lines 7−8.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/private/inert.h`

```
 1 #ifndef INERTH
 2 #define INERTH
 3 #include <climits>   //for INT_MIN and INT_MAX
 4 #include "wabbit.h"
 5
 6 class inert: private virtual wabbit {
 7     int hungry() const {return INT_MIN;}
 8     int bitter() const {return INT_MAX;}
 9 public:
10     inert(game *initial_g, unsigned initial_x, unsigned initial_y,
11         char initial_c)
12         : wabbit(initial_g, initial_x, initial_y, initial_c) {}
13 };
14 #endif
```

Remove the four classes `inert`, `victim`, `predator`, and `halogen`, and their header files. Replace them with the template class in the following line 7. Then reinstate classes `inert`, `victim`, `predator`, and `halogen` as the typedefs in 16−20. Class `boulder`, for example, will now be derived from `immobile` and `inert_t`, and the constructor for `boulder` will call the constructors for `immobile` and `inert_t`.

In place of the macro `INT_MIN` in line 18, I would rather call the function `numeric_limits<int>::min()` on pp. 745−747. But a template argument must be a constant expression (p. 234), not the return value of a function.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/rank.h`

```
 1 #ifndef RANKH
 2 #define RANKH
 3 #include <climits>
 4 #include "wabbit.h"
 5
 6 template <int HUNGRY, int BITTER>
 7 class rank: private virtual wabbit {
 8     int hungry() const {return HUNGRY;}
 9     int bitter() const {return BITTER;}
10 public:
11     rank(game *initial_g, unsigned initial_x, unsigned initial_y,
12         char initial_c)
13         : wabbit(initial_g, initial_x, initial_y, initial_c) {}
14 };
15
16 //Convenient names for the rank classes:
17
18 typedef rank<INT_MIN, INT_MAX> inert_t;
19 typedef rank<INT_MIN, INT_MIN> victim_t;
20 //etc.
```

```
21 #endif
```

▲

▼ Homework 7.2.2c:
Version 4.1 of the Rabbit Game: create the sixteen grandchild classes with one template

The various species of animals (`boulder`, `rabbit`, `wolf`, etc.) became grandchildren of class `wabbit` on pp. 565−566, and were last modified on p. 582. They are identical except for the names of the two base classes and the value of the `char` literal.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/private/boulder.h`

```
 1 #ifndef BOULDERH
 2 #define BOULDERH
 3 #include "immobile.h"
 4 #include "rank.h"
 5
 6 class boulder: private immobile, private inert_t {
 7 public:
 8     boulder(game *initial_g, unsigned initial_x, unsigned initial_y)
 9         : wabbit(initial_g, initial_x, initial_y, 'b'),
10         immobile(initial_g, initial_x, initial_y, 'b'),
11         inert_t (initial_g, initial_x, initial_y, 'b')
12         {}
13 };
14 #endif
```

Remove classes `boulder`, `rabbit`, `wolf`, etc., and their header files. Replace them with the template class in the following line 8.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/grandchild.h`

```
 1 #ifndef GRANDCHILDH
 2 #define GRANDCHILDH
 3
 4 //MOTION must have member functions decide and (optionally) punish;
 5 //RANK must have member functions hungry and bitter.
 6
 7 template <class MOTION, class RANK, char C>
 8 class grandchild: private MOTION, private RANK {
 9 public:
10     grandchild(game *initial_g, unsigned initial_x, unsigned initial_y)
11         : wabbit(initial_g, initial_x, initial_y, C),
12           MOTION(initial_g, initial_x, initial_y, C),
13             RANK(initial_g, initial_x, initial_y, C)
14         {}
15 };
16 #endif
```

Reinstate classes `boulder`, `rabbit`, `wolf`, etc., as the instantiations of class `grandchild` in the following lines 14, 18, etc.

```
 1 //Excerpt from game.C.
 2
 3 //The header files for the four styles of motion.
 4 #include "manual.h"
```

```
#include "brownian.h"
//etc.

#include "rank.h"
#include "grandchild.h"

//Excerpt from game::game

    case 'b':   //boulder
        new grandchild<immobile, inert_t, 'b'>(this, x, y);
        break;

    case 'r':   //rabbit
        new grandchild<brownian, victim_t, 'r'>(this, x, y);
        break;
```

Or make the code self-documenting by hiding each data type in a typedef:

```
    case 'b':
        typedef grandchild<immobile, inert_t, 'b'> boulder_t;
        new boulder_t(this, x, y);
        break;

    case 'r':
        typedef grandchild<brownian, victim_t, 'r'> rabbit_t;
        new rabbit_t(this, x, y);
        break;
```

▲

7.2.3 “Template” Template Arguments

A template argument can stand for a data type (`T`) or a constant value (`MAX_SIZE`). Both possibilities appeared in line 13 of `stack.h` on p. 691. A template argument can also stand for a template class such as `vector`, `list`, or `grandchild`.

Why can’t our existing `T` stand for `vector`? A `T` can indeed stand for any data type, but `vector` is not a data type. It is a template class, which is an indefinitely large set of data types. To stand for a template class, a new kind of template argument had to be invented.

Our examples will be rudimentary versions of the container classes `set` and `map` in the C++ Standard Library. The template arguments of the real `set` and `map` are different from the arguments used below. And the return values of the member functions of the real `set` and `map` are much more useful than the simple data types below.

A set

A `set` object contains values of type `T`, but only at most one copy of any value. We will present three implementations of the class; the third will have a ‘‘template’’ template argument.

We first implement the class on top of a `vector`, the data member `v` in line 12. Since we wrote no default constructor for `set`, it behaves as if we had written one that calls the default constructor for `v` and does nothing else.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/set/set1/set.h`

```
1 #ifndef SETH
2 #define SETH
```

```
 3 #include <iostream>
 4 #include <cstdlib>
 5 #include <vector>
 6 using namespace std;
 7
 8 //T must be copy constructable (line 31) and equality comparable (line 17).
 9
10 template <class T>
11 class set {
12     vector<T> v;
13 public:
14     bool find(const T& t) const {
15         for (typename vector<T>::const_iterator it = v.begin();
16             it != v.end(); ++it) {
17             if (*it == t) {
18                 return true;
19             }
20         }
21
22         return false;
23     }
24
25     void insert(const T& t) {
26         if (find(t)) {
27             cerr << "Sorry, the value is already in the set.\n";
28             exit(EXIT_FAILURE);
29         }
30
31         v.push_back(t);
32     }
33 };
34 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/set/set1/main.C`

```
#include <iostream>
#include <cstdlib>
#include <string>
#include "set.h"
using namespace std;
int main()
{
    ::set<string> s;   //born empty
    s.insert("Mercury");
    s.insert("Venus");
    s.insert("Earth");

    cout << boolalpha
        << s.find("Mercury") << "\n"
        << s.find("Mongo")   << "\n";

    return EXIT_SUCCESS;
}
```

`true`	`Mercury` *was inserted in line 8.*
`false`	`Mongo` *was never inserted.*

Our choice of `vector` as the underlying container was hardwired into lines 12 and 15 of the above `set.h`. We can parameterize it with the template argument `CONTAINER` in the following line 11.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/set/set2/set.h`

```
 1 #ifndef SETH
 2 #define SETH
 3 #include <iostream>
 4 #include <cstdlib>
 5 #include <vector>
 6 using namespace std;
 7
 8 //T must be copy constructable and equality comparable.
 9 //CONTAINER must have const_iterator, begin, end, push_back.
10
11 template <class T, class CONTAINER = vector<T> >   //needs whitespace
12 class set {
13     CONTAINER c;
14 public:
15     bool find(const T& t) const {
16         for (typename CONTAINER::const_iterator it = c.begin();
17             it != c.end(); ++it) {
18             if (*it == t) {
19                 return true;
20             }
21         }
22
23         return false;
24     }
25
26     void insert(const T& t) {
27         if (find(t)) {
28             cerr << "Sorry, the value is already in the set.\n";
29             exit(EXIT_FAILURE);
30         }
31
32         c.push_back(t);
33     }
34 };
35 #endif
```

Line 10 uses the default container; line 19 overrides it.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/set/set2/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <list>
 4 #include <string>
 5 #include "set.h"
 6 using namespace std;
 7
 8 int main()
 9 {
10     ::set<string> s;                  //implemented atop vector<string>
11     s.insert("Mercury");
12     s.insert("Venus");
13     s.insert("Earth");
14
15     cout << boolalpha
16         << s.find("Mercury") << "\n"
17         << s.find("Mongo")   << "\n\n";
18
19     ::set<string, list<string> > s2;   //implemented atop list<string>
20     s2.insert("Mercury");
21     s2.insert("Venus");
22     s2.insert("Earth");
23
24     cout << boolalpha
25         << s2.find("Mercury") << "\n"
26         << s2.find("Mongo")   << "\n";
27
28     return EXIT_SUCCESS;
29 }
```

```
true
false

true
false
```

I’m afraid that the two copies of `string` in the above line 19 might get out of sync. I wish we could say

```
30     ::set<string, list> s2;
```

But `CONTAINER` has to be a data type, and `list` is not a data type. If we try it, the program will not compile.

```
main.C: In function ’int main()’:
main.C:19:21: error: type/value mismatch at argument 2 in template
parameter list for ’template<class T, class CONTAINER> class set’
main.C:19:21: error:   expected a type, got ’list’
```

The solution is to let `CONTAINER` be a *template* template argument. The following line 15 declares that `CONTAINER` is a template class that will accept one template argument `U`. (We don’t have to write the `U`, but it makes the declaration of `CONTAINER` look more familiar.) For example, `CONTAINER` could be `myvector`, which is exactly the same as the standard library `vector` except that it takes only one template argument. (`std::vector` takes two.) Lines 17 and 20 apply one template argument to the `CONTAINER`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/set/set3/set.h`

```
1 #ifndef SETH
2 #define SETH
3 #include <iostream>
4 #include <cstdlib>
5 #include <vector>
```

```
using namespace std;

template <class T>
class myvector: public vector<T> {
};

//T must be copy constructable and equality comparable.
//CONTAINER must have const_iterator, begin, end, push_back.

template <class T, template <class U> class CONTAINER = myvector>
class set {
    CONTAINER<T> c;
public:
    bool find(const T& t) const {
        for (typename CONTAINER<T>::const_iterator it = c.begin();
            it != c.end(); ++it) {
            if (*it == t) {
                return true;
            }
        }

        return false;
    }

    void insert(const T& t) {
        if (find(t)) {
            cerr << "Sorry, the value is already in the set.\n";
            exit(EXIT_FAILURE);
        }

        c.push_back(t);
    }
};
#endif
```

We can now define the following template class at line 7 of `main.C` on p. 698.

```
template <class T>
class mylist: public list<T> {
};
```

and change line 19 of `main.C` to the following. The output remains the same.

```
    //string is a data type, mylist is a template class w/ 1 template arg
    ::set<string, mylist> s2;
```

In real life, we would never write the above class `set`. We would simply include the standard library header file `<set>`, declare a `set<string>`, and call its member functions `insert`, `find`, `erase`, etc.

A "template" template argument that takes two different T's

It scarcely seems worthwhile to introduce a new language feature just to avoid writing the name `string` twice in line 19 of `main.C` on p. 699. The real purpose of this feature is to allow a template to apply two different template arguments to the `CONTAINER`, as in the following lines 18 and 19.

A `map` object contains pairs of values of types `KEY` and `VALUE`. Think of it as an array whose subscripts are of type `KEY` and whose elements are of type `VALUE`. Lines 18 and 19 can apply one template argument to `CONTAINER` thanks to the declaration in line 16.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/mymap/map.h`

```
#ifndef MAP
#define MAP
#include <iostream>
#include <cstdlib>
using namespace std;

template <class T>
class myvector: public vector<T> {
};

//KEY must be copy constructable and equality comparable.
//VALUE must be copy constructable.
//CONTAINER must have const_iterator, begin, end, push_back.

template <class KEY, class VALUE,
    template <class U> class CONTAINER = myvector>
class map {
    CONTAINER<KEY> key;
    CONTAINER<VALUE> value;
public:
    const VALUE& find(const KEY& k) const {
        typename CONTAINER<VALUE>::const_iterator itv = value.begin();

        for (typename CONTAINER<KEY>::const_iterator itk = key.begin();
            itk != key.end(); ++itk, ++itv) {

            if (*itk == k) {
                return *itv;
            }
        }

        cerr << "key not found\n";
        exit(EXIT_FAILURE);
    }

    void insert(const KEY& k, const VALUE& v) {
        key  .push_back(k);
        value.push_back(v);
    }
};
#endif
```

Here is a `map` that contains each planet’s gravity as a fraction of the earth’s.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/mymap/main.C`

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include <string>
#include <list>
#include "map.h"
```

```
 7 using namespace std;
 8 
 9 template <class T>
10 class mylist: public list<T> {
11 };
12 
13 int main()
14 {
15     ::map<string, double> gravity;            //implemented atop 2 vectors
16 
17     gravity.insert("Mercury",  .27);
18     gravity.insert("Venus",    .85);
19     gravity.insert("Earth",   1.00);
20     cout << "Mercury" << ", " << gravity.find("Mercury") << "\n";
21 
22     ::map<string, double, mylist> gravity2;   //implemented atop 2 mylists
23     gravity2.insert("Mercury",  .27);
24     cout << "Mercury" << ", " << gravity2.find("Mercury") << "\n";
25 
26     return EXIT_SUCCESS;
27 }
```

```
Mercury, 0.27
Mercury, 0.27
```

A `map` should contain only at most one pair with a given subscript. We declined to check for this because a `find` function whose return type is `VALUE` or `VALUE&` has no graceful way of returning an indication of failure. Even worse, parallel containers such as those in the above lines 18 and 19 tend to fall out of sync. It would be better to have *one* data structure containing objects each of which has a pair of data members. These ‘‘pair’’ objects will appear on pp. 785−787.

In real life, we would never write the above class `map`. We would simply include the header file `<map>` and declare the `map<string, double>` on p. 787.

7.2.4 Partial and Explicit Specialization of a Template Class

We can overload the name of a function or template function, but not the name of a class or template class. To compensate, a template class can be *partially specialized* and *explicitly specialized.* Only template classes, not template functions, can be partially specialized.

The following class is merely a wrapper for the data member `t` in line 9. We provide member functions defined inside and outside the class definition, in lines 11 and 12.

Line 8 is the *primary template* for class `wrapper` because it has no <angle brackets> after the name `wrapper`. The primary template will instantiate a `wrapper` for any data type not covered by the specializations below it.

Line 19 is a *partial specialization* because it has angle brackets containing a `T`. This template will instantiate a `wrapper` for any type of pointer not covered by an even more specific specialization.

Line 37 is an *explicit specialization* because its angle brackets contain no `T`. This template will instantiate only one type of `wrapper`, for a pointer to a `const char`. Its preamble in line 36 has no `T`, just like the preamble for an explicit specialization of a template function. (See line 9 of `explicit.C` on p. 665.)

The primary template and its specializations may differ in the names and types of their members and friends; compare the `t` in line 9 with the `p`’s in 20 and 38.

Partial and explicit specializations can be written in any order as long as they come after the primary template. Thus an explicit specialization belongs to the template class as a whole, not to any particular partial specialization. For example, the explicit specialization for `const char *` could have been defined before the partial specialization for `const T *`, or without any partial specialization at all. Recall that the rules were different for template functions: an explicit specialization of a template function belonged to one particular template function, not to of all the template functions sharing the same name.

Oddly, the explicit specialization's member function in line 45 has no preamble of its own. Also note than lines 36−42 are redundant. We can comment them out because 18−24 will instantiate the same code. If we do this, however, we must comment line 44 back in.

Line 48 is a partial specialization for any type of `vector`. The `const_iterator` in line 60 is a member of an instantiation of `vector`. But the primary template for `vector` and its various specializations may differ in the names and types of their members. Theoretically, the `const_iterator` of `vector<int>` might be name of a data type, while the `const_iterator` of `vector<char>` might be a data member. To show that `const_iterator` is the name of a data type, we need the keyword `typename` as on pp. 671−676.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/partial/wrapper.h`

```
 1 #ifndef WRAPPERH
 2 #define WRAPPERH
 3 #include <iostream>
 4 #include <vector>
 5 using namespace std;
 6
 7 template <class T>
 8 class wrapper {                 //primary template
 9     const T t;
10 public:
11     wrapper(const T& initial_t): t(initial_t) {}
12     void print() const;
13 };
14
15 template <class T>
16 inline void wrapper<T>::print() const {cout << t;}
17
18 template <class T>
19 class wrapper<const T *> {      //partial specialization
20     const T *const p;
21 public:
22     wrapper(const T *initial_p): p(initial_p) {}
23     void print() const;
24 };
25
26 template <class T>
27 void wrapper<const T *>::print() const
28 {
29     cout << p;
30     if (p != 0) {
31         cout << " -> ";
32         wrapper<T>(*p).print();
33     }
34 }
35
36 template <>
```

```
37 class wrapper<const char *> {   //explicit specialization
38     const char *const p;
39 public:
40     wrapper(const char *initial_p): p(initial_p) {}
41     void print() const;
42 };
43
44 //template <>     //No preamble.
45 inline void wrapper<const char *>::print() const {cout << "\"" << p << "\"";}
46
47 template <class T>
48 class wrapper<vector<T> > {     //another partial specialization
49     const vector<T> v;
50 public:
51     wrapper(const vector<T>& initial_v): v(initial_v) {}
52     void print() const;
53 };
54
55 template <class T>
56 void wrapper<vector<T> >::print() const
57 {
58     cout << "(";
59
60     for (typename vector<T>::const_iterator it = v.begin(); it != v.end();
61         ++it) {
62         if (it != v.begin()) {
63             cout << ", ";
64         }
65         wrapper<T>(*it).print();
66     }
67
68     cout << ")";
69 }
70 #endif
```

The above line 32 constructs an anonymous object and calls its `print` function. If this makes you uncomfortable, give the object a name:

```
71     const wrapper<T> w(*p);
72     w.print();
```

The `for` loop in the above lines 60−61 can tamed with a typedef:

```
73     typedef typename vector<T>::const_iterator const_iterator;
74
75     for (const_iterator it = v.begin(); it != v.end(); ++it) {
```

The `vvi` in the following line 25 is born holding two empty `vector<int>`'s. We then push integers into them.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/partial/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include "wrapper.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     int i = 10;
10     wrapper<int>(i).print();   //construct anonymous temporary and print it
11     cout << "\n";
12
13     wrapper<const int *>(&i).print();
14     cout << "\n";
15
16     wrapper<const char *>("hello").print();
17     cout << "\n";
18
19     const char *a[] = {"moe", "larry", "curly"};
20     const size_t n = sizeof a / sizeof a[0];
21     vector<const char *> v(a, a + n);
22     wrapper<vector<const char *> >(v).print();
23     cout << "\n";
24
25     vector<vector<int> > vvi(2);   //2nd func arg defaults to vector<int>()
26     vvi[0].push_back(10);
27     vvi[0].push_back(20);
28     vvi[1].push_back(30);
29     vvi[1].push_back(40);
30     vvi[1].push_back(50);
31     wrapper<vector<vector<int> > >(vvi).print();
32     cout << "\n";
33
34     return EXIT_SUCCESS;
35 }
```

```
10                            line 10: int
0xffbff084 -> 10              line 13: const int *
"hello"                       line 16: const char *
("moe", "larry", "curly")     line 22: vector<const char *>
((10, 20), (30, 40, 50))      line 31: vector<vector<int> >
```

▼ Homework 7.2.4a: a partial specialization for T *

Without the `const`, line 13 of the above `main.C` would instantiate the primary template for class `wrapper`, not the template for `wrapper<const T *>`. After all, `int` is not a `const T`, so the `int *` in line 13 could not be a `const T *`.

```
0xffbff1fc
```

Remedy this by defining a partial specialization for `wrapper<T *>` without the `const`.

▲

▼ Homework 7.2.4b: create an operator<< friend

Create `operator<<` friends for class `wrapper` and each of its specializations. Then let each `print` member function do its work by calling the corresponding `operator<<`.

You will have to define four `operator<<` functions (five, with the previous homework). The `operator<<` that takes a `wrapper<const char *>` will not be a template function; the others will be. The ones that take `wrapper<T>` and `wrapper<const char *>` will be inline; the others will not be.

▲

▼ Homework 7.2.4c: try to make a partial specialization of a template function

A template function cannot have a partial specialization. Learn to recognize the error message on your platform when you accidentally try it.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/partial/function.C`

```
#include <iostream>
#include <cstdlib>
using namespace std;

template <class T>
inline void print(const T& t) {cout << t;}

template <class T>
void print<const T *>(const T *p)
{
    cout << p;
    if (p != 0) {
        cout << " -> " << *p;
    }
}

int main()
{
    int i = 10;
    print(&i);
    cout << "\n";
    return EXIT_SUCCESS;
}
```

The GNU `g++` compiler says

```
function.C:9:33: error: function template partial specialization
'print<const T*>' is not allowed
```

The Sun `CC` compiler says

```
sh[1]: CC: not found [No such file or directory]
```

▲

Simulating other kinds of templates

The only kinds of templates are function templates and class templates. But two others can be convincingly faked.

(1) Imagine a “variable template”. Let’s say that each built-in data type `T` needed its own `int` variable named `digits10`, giving the number of decimal digits that can be held in a `T`.

```
template <class T>      //T must be a numeric type (char to long double).
void f()
{
    //wishful thinking, won't compile

    cout << "This data type can hold " << digits10<T>
        << " decimal digits.\n";
}
```

We can implement this family of variables by making a template class with a public static data member named `digits10`, and then specializing the class for each type. In fact, this has already been done for us. See the `digits10` static data member of the template class `numeric_limits` on p. 746.

(2) Imagine a “typedef template”. Let’s say that each of the built-in character types, `char` and `wchar_t`, needed its own typedef giving the corresponding type of integer big enough to hold any value for that type of character, as well as the end-of-file value.

```
template <class CHAR>   //CHAR could be char or wchar_t.
void f(CHAR c)          //can be passed by value
{
    //wishful thinking; won’t compile
    int_type<CHAR> i = c;

    cout << "The corresponding integer value is " << i << "\n";
}
```

We can implement this family of typedefs by making a template class with a public typedef member named `int_type`, and then specializing the class for each type. In fact, this has already been done for us. See the `int_type` typedef member of the template class `char_traits` in lines 8 and 11 of `char_traits.C` on p. 749.

7.2.4.1 Template Metaprogramming

Whenever possible, we want to compute at compile time rather than at runtime. Consider

```
    cout << 10 + 20 << "\n";
```

or even

```
    int i = 10;
    const int j = 20;
    cout << i + j << "\n";
```

All the operands of the `+` operator are *constants*, in the sense of “values that are known at compile time”. A smart, well-motivated compiler can perform *constant folding* and behave as if we had said

```
    cout << 30 << "\n";
```

Constant folding also includines making decisions at compile time. When we write

```
    if (true) {
        cout << "true\n";
    } else {
        cout << "false\n";
    }
```

a smart compiler can behave as if we had said

```
    cout << "true\n";
```

To guarantee that a constant will be folded at compile time, we use a technique called *template metaprogramming*. It exploits constant template arguments, explicit specialization, and enumerations.

Change function arguments to template arguments.

Our first example is the classic `for` loop, which has to perform a comparison and increment during each iteration.

```
 1      for (int i = 1; i <= 4; ++i) {
 2          cout << i << "\n";
 3      }
```

We could avoid this runtime arithmetic by *unrolling* the loop:

```
 4      cout << 1 << "\n";
 5      cout << 2 << "\n";
 6      cout << 3 << "\n";
 7      cout << 4 << "\n";
```

Template metaprogramming will let us unroll the loop without typing it over and over. To ease the transition to metaprogramming, we first rewrite the loop using recursion. For the time being, the comparisons and increments will still be done at runtime.

```
 8 //Output the integers from 1 to last inclusive.
 9 //If last <= 0, output nothing.
10
11 inline void count(int last)
12 {
13     if (last > 0) {
14         count(last - 1);
15         cout << last << "\n";
16     }
17 }
18
19 int main()
20 {
21     count(4);
22 }
```

Incidentally, the `inline` declaration in the above line 11 can be honored only if the value of `last` is known at compile time. A smart compiler might look ahead to the `4` in line 21, but there is no guarantee of this. If the value is unknown, an inlined function that calls itself will blow up to infinite size.

We now change the function argument `last` to the template argument `LAST` in the following line 9. This results in a series of different functions: `count<0>`, `count<1>`, `count<2>`, etc. The calls can now be inline because no function calls itself.

The value of a constant template argument is *always* computed at compile time, so each subtraction in line 12 will now be done at compile time. And the name of a function is *always* bound at compile time, unless the function is a virtual member function. For example, each `count<LAST-1>` in line 12 will be bound at compile time either to an instantiation of line 10 or to line 17. Each comparison of `LAST-1` to zero will therefore be done at compile time.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/meta/unroll1.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
4
5 //Output the integers from 1 to LAST inclusive.   LAST must be non-negative.
6 //count is a template function consisting of the general-purpose template
7 //in line 10 and the explicit specialization in line 17.
```

```
 8
 9 template <int LAST>
10 inline void count()
11 {
12     count<LAST - 1>();
13     cout << LAST << "\n";
14 }
15
16 template <>
17 inline void count<0>() {}
18
19 int main()
20 {
21     count<4>();      //4 lines of output
22     count<0>();      //no output
23     //count<-1>();   //won’t compile: "instantiation depth exceeds maximum"
24
25     return EXIT_SUCCESS;
26 }
```

```
1
2
3
4
```

▼ Homework 7.2.4.1a: allow LAST to be negative

The above program will not compile if `LAST` is negative (line 23). Remedy this in three easy steps.

(1) In the above lines 5−17, change the name of the template function from `count` to `_count`.

(2) After the definition of `_count`, define the following template function. Line 5 is the only place where the program will call `_count`.

```
1 //Output the integers from 1 to LAST inclusive.
2 //If LAST < 1, output nothing.
3
4 template <int LAST>
5 inline void count() {_count<LAST < 1 ? 0 : LAST>();}
```

(3) The `main` function will continue to call `count`, but now it will be the `count` we just introduced.

▲

Explicit specialization of more than one template argument

The starting point 1 was implicitly hardwired into the above loop, although it is hard to see where. It took the form of the explicit specialization for zero in line 17 of `unroll1.C`. The starting point can also be parameterized as the first template argument in the following line 20. When line 12 senses that the loop is done, it will pass two zeroes to the explicit specialization in line 16. As on p. 664, if any template argument is explicitly specialized, they all must be.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/meta/unroll2.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
```

```
 4
 5 //Output the integers from FIRST to LAST inclusive.
 6 //If FIRST > LAST, just print FIRST.
 7
 8 template <int FIRST, int LAST>
 9 void count()
10 {
11     cout << FIRST << "\n";
12     count<FIRST >= LAST ? 0 : FIRST + 1, FIRST >= LAST ? 0 : LAST>();
13 }
14
15 template <>
16 inline void count<0, 0>() {}
17
18 int main()
19 {
20     count<-1, 4>();    //-1 to 4 inclusive
21     return EXIT_SUCCESS;
22 }
```

To consolidate the repetition in the above line 12, rewrite it as follows. The value of an enumeration is always computed at compile time, allowing it to be part of a constant template argument.

```
23     enum {done = FIRST >= LAST};
24     count<done ? 0 : FIRST + 1, done ? 0 : LAST>();
```

```
-1
0
1
2
3
4
```

▼ Homework 7.2.4.1b: allow FIRST to be greater than LAST

The above function should produce no output if `FIRST > LAST`. Accomplish this in three easy steps.

(1) In the above lines 5−16, change the name of the template function from `count` to `_count`.

(2) After the definition of `_count`, define the following template function. Line 6 is the only place where the program will call `_count`. Note that a template argument containing the operator `>` must be enclosed in parentheses; see p. 693.

```
1 //Output the integers from FIRST to LAST inclusive.
2 //If FIRST > LAST, output nothing.
3
4 template <int FIRST, int LAST>
5 inline void count() {
6     _count<(FIRST > LAST ? 0 : FIRST), (FIRST > LAST ? 0 : LAST)>();
7 }
```

To consolidate the repetition in the above line 6, rewrite it as follows.

```
8     enum {empty = FIRST > LAST};
9     _count<empty ? 0 : FIRST, empty ? 0 : LAST>();
```

(3) The `main` function will continue to call `count`, but now it will be the `count` we just introduced.

▲

▼ Homework 7.2.4.1c: bubblesort

The following function bubblesorts an array of `n` integers into increasing order (pp. 47−48). In contrast to the usual C++ practice, we pass it a starting address and a count of elements. Warning: the `size_t n` is unsigned. If `n` were zero, the `n-1` in line 4 would be a huge positive number and the loop would iterate too many times. Line 3 prevents this from happening.

```
 1 void sort(int *p, size_t n)
 2 {
 3     for (; n > 1; --n) {
 4         for (size_t i = 0; i < n - 1; ++i) {
 5             if (p[i + 1] < p[i]) {       //if in wrong order,
 6                 const int temp = p[i];   //swap them
 7                 p[i] = p[i + 1];
 8                 p[i + 1] = temp;
 9             }
10         }
11     }
12 }
```

The comparision in the above line 5, and the initializations and assignments in lines 6−8, will have to be performed at runtime. Only then are the values of the array elements known. But if the number of elements was known at compile time, we could unroll the loops. The comparisons, decrement, and increment in lines 3 and 4, and the miscellaneous additions in lines 5−8, could all be done at compile time.

The first step is to write the loops recursively.

```
13 //This function does the work of the inner loop (lines 4-10 above).
14 //Examine and modify the elements whose subscripts are i to j inclusive.
15 //Since this function is called only from lines 27 and 37,
16 //it can assume that i < j.
17
18 void inner(int *p, size_t i, size_t j)
19 {
20     if (p[i + 1] < p[i]) {       //if in wrong order,
21         const int temp = p[i];   //swap them
22         p[i] = p[i + 1];
23         p[i + 1] = temp;
24     }
25
26     if (i + 1 < j) {
27         inner(p, i + 1, j);
28     }
29 }
30
31 //This function does the work of the outer loop (lines 3 and 11 above).
32 //Sort n elements (subscripts 0 to n-1 inclusive).
33
34 void sort(int *p, size_t n)
35 {
36     if (n > 1) {
37         inner(p, 0, n - 1);
38         sort(p, n - 1);
39     }
40 }
```

The next step is to change the function arguments to template arguments. Define the following template functions, including the explicit specializations shown below. The `if` in the above line 20 will still be done at runtime; the ones in lines 26 and 36 will disappear.

```
41 //Examine and modify the elements whose subscripts are I to J inclusive.
42
43 template <size_t I, size_t J>
44 void inner(int *p)
45 {
46     //fill this in;
47 }
48
49 template <>
50 inline void inner<0, 0>(int *p) {}
51
52 //Sort N elements (subscripts 0 to N-1 inclusive).
53
54 template <size_t N>
55 void sort(int *p)
56 {
57     //fill this in;
58 }
59
60 template <>
61 inline void sort<0>(int *p) {}
```

When you are done, try to change all the `int`’s in the above lines 41−64 to `T`’s. What goes wrong?

▲

Change a template function to a template class.

The factorial function is the product of all the positive integers up to a given integer. For example, the factorial of 4 is

$$4! = 1 \times 2 \times 3 \times 4 = 24$$

The factorial of zero is defined to be zero; the factorial of a negative integer is undefined.

A straightforward way to compute a factorial is with the following loop.

```
int factorial(int n)
{
    int product = 1;

    for (; n > 1; --n) {
        product *= n;
    }

    return product;
}
```

A more elegant function uses recursion.

```
inline int factorial(int n) {return n <= 1 ? 1 : n * factorial(n - 1);}
```

We can perform the comparisons and subtractions at compile time by changing the function arguments to template arguments.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/meta/factorial1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 template <int I>
 6 inline int factorial() {return I * factorial<I - 1>();}
 7
 8 template <>
 9 inline int factorial<0>() {return 1;}
10
11 int main()
12 {
13     cout << factorial<4>() << "\n";
14     return EXIT_SUCCESS;
15 }
```

```
24
```

What about the multiplications? For all we know, the above line 13 might still be compiled as

```
16     cout << 4 * 3 * 2 * 1 << "\n";
```

leaving the product to be computed at runtime. We would like a guarantee that the line will be compiled as follows.

```
17     cout << 24 << "\n";
```

A simple adjustment is all that is necessary. We change `factorial` from a function with a return value to a class with a public enumeration. The `*` in the following line 7 computes the value of the enumeration, and the value of an enumeration is *always* computed at compile time.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/meta/factorial2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 template <int I>
 6 struct factorial {
 7     enum {value = I * factorial<I - 1>::value};
 8 };
 9
10 template <>
11 struct factorial<0> {
12     enum {value = 1};
13 };
14
15 int main()
16 {
17     cout << factorial<4>::value << "\n";
18     return EXIT_SUCCESS;
19 }
```

```
24
```

Were the multiplications really moved up to compile time? There is no way to tell from the output; we will have to examine the translation of the program into assembly language. My compiler (GNU `g++`) lets me see this with the `-S` option.

```
1$ g++ -S factorial2.C                    minus uppercase S
2$ ls -l factorial2.s                     minus lowercase L
3$ more factorial2.s

main:
    !etc.
    mov 24, %o1
```

A compile-time array

Each class `j` in the following lines 52−55 takes a Julian date in the range 1 to 365 inclusive and offers public enumerations giving the corresponding month and day of the month. All arithmetic, testing, and looping are done at compile time.

The general-purpose template for class `length` must be declared in line 8 before the explicit specializations can be defined in lines 10−21. But the general-purpose template need not be defined: there is no such value as the length of a general-purpose month.

In line 31, `same_month` is true (or at least non-zero) if the Julian day `J` belongs to the same month as `J-1`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/meta/month.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 //length<MONTH>::value is the number of days in month MONTH (1 to 12 inclusive).
 6
 7 template <int MONTH>
 8 struct length;
 9
10 template <> struct length< 1> {enum {value = 31};};   //january
11 template <> struct length< 2> {enum {value = 28};};   //february
12 template <> struct length< 3> {enum {value = 31};};   //march
13 template <> struct length< 4> {enum {value = 30};};   //april
14 template <> struct length< 5> {enum {value = 31};};   //may
15 template <> struct length< 6> {enum {value = 30};};   //june
16 template <> struct length< 7> {enum {value = 31};};   //july
17 template <> struct length< 8> {enum {value = 31};};   //august
18 template <> struct length< 9> {enum {value = 30};};   //september
19 template <> struct length<10> {enum {value = 31};};   //october
20 template <> struct length<11> {enum {value = 30};};   //november
21 template <> struct length<12> {enum {value = 31};};   //december
22
23 //j<J>::month is the month (1 to 12 inclusive) and j<J>::day is the day
24 //of the month (1 to 31 inclusive) of Julian date J (1 to 365 inclusive).
```

```

template <int J>
class j {
    enum {
        d = j<J - 1>::day,
        m = j<J - 1>::month,
        same_month = d < length<m>::value
    };
public:
    enum {
        month = same_month ? m : m + 1,
        day = same_month ? d + 1 : 1
    };
};

template <>
class j<1> {   //Julian date 1 is january 1.
public:
    enum {
        month = 1,
        day = 1
    };
};

int main()
{
    cout
        << j<  1>::month << " " << j<  1>::day << "\n"    //january 1
        << j< 31>::month << " " << j< 31>::day << "\n"    //january 31
        << j< 32>::month << " " << j< 32>::day << "\n"    //february 1
        << j<365>::month << " " << j<365>::day << "\n";   //december 31

    return EXIT_SUCCESS;
}
```

```
1 1
1 31
2 1
12 31
```

Plug the derived class into the base class.

Our metaprogramming examples have used constant template arguments such as `I` and `J`. We can also use a data type template argument `T`. First, though, we will make a side trip to examine a surprising but curiously recurrent template pattern.

Consider a base class that keeps count of how many of its objects currently exist.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/meta/curious1.C`

```
#include <iostream>
#include <cstdlib>
using namespace std;

class base {
 6     static int n;
 7 public:
 8     base() {++n;}
 9     base(const base& another) {++n;}
10     ˜base() {--n;}
11     static int count() {return n;}
12 };
13
14 int base::n = 0;
15
16 class derived1: public base {
17     //etc.
18 };
19
20 class derived2: public base {
21     //etc.
22 };
23
24 int main()
25 {
26     derived1 a, b, c;
27     derived2 d;
28
29     cout << "derived1::count " << derived1::count() << "\n"
30         << "derived2::count " << derived2::count() << "\n";
31     return EXIT_SUCCESS;
32 }
```

```
derived1::count 4
derived2::count 4
```

We now have the total number of objects of class `base` and its descendants. For this reason, it would make more sense for the above lines 29−30 to call the function as `base::count`. But what if we wanted a separate count for each derived class? Each derived class would need its own static data member `n` and its own static member function `count`. In fact, we would have to replicate the entire base class for each derived class.

Not surprisingly, we effect this replication by letting the base class be a template. Surprisingly, the argument passed to the template in line 18 will be the derived class. Can we use `derived1` in the angle brackets in line 18 before we have seen the end of its class definition in line 20? Yes, as long as the template class `base` has no code that constructs a `T` or needs to know the size of a `T`. A simpler example was the class `node` on p. 214, whose name was mentioned before the end of its class definition.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/meta/curious2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 template <class T>
 6 class base {
 7     static int n;
 8 public:
 9     base() {++n;}
10     base(const base& another) {++n;}
```

```
    ˜base() {--n;}
    static int count() {return n;}
};

template <class T>
int base<T>::n = 0;

class derived1: public base<derived1> {
    //etc.
};

class derived2: public base<derived2> {
    //etc.
};

int main()
{
    derived1 a, b, c;
    derived2 d;

    cout << "derived1::count " << derived1::count() << "\n"
        << "derived2::count " << derived2::count() << "\n";
    return EXIT_SUCCESS;
}
```

```
derived1::count 3
derived2::count 1
```

Polymorphism at compile time

We are now ready for our final example of metaprogramming. The following line 9 calls the `implementation` member function of a derived class. The function is selected at runtime because of the keyword `virtual` in line 8. To make the runtime selection possible, each object contains a pointer to a vtbl (p. 498).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/meta/polymorphism1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class base {
 6 public:
 7     virtual ˜base() {}
 8     virtual void implementation() const = 0;
 9     void interface() const {implementation();}
10 };
11
12 class derived1: public base {
13 public:
14     void implementation() const {cout << "derived1\n";}
15 };
16
17 class derived2: public base {
```

```
18 public:
19     void implementation() const {cout << "derived2\n";}
20 };
21
22 int main()
23 {
24     derived1 d1;
25     d1.interface();
26
27     derived2 d2;
28     d2.interface();
29
30     cout << "sizeof (base) == " << sizeof (base) << "\n"
31         << "sizeof (derived1) == " << sizeof (derived1) << "\n"
32         << "sizeof (derived2) == " << sizeof (derived2) << "\n";
33
34     return EXIT_SUCCESS;
35 }
```

```
derived1
derived2
sizeof (base) == 4
sizeof (derived1) == 4
sizeof (derived2) == 4
```

When the above line 9 is called from line 25, it always selects `derived1::implementation`. A smart compiler might recognize this. If it does, it can let the member function be called without use of the vtbl. To guarantee that *any* compiler will recognize this, the following program will use the curiously recurrent template pattern. The code is faster and the objects smaller. The bad news is that `d1` and `d2` are no longer derived from a commmon base class; no pointer (beyond a rock-bottom `void *`) can point to both of them.

A *downcast* is a conversion from ‘‘pointer to base’’ to ‘‘pointer to derived’’. Since the cast in the following line 16 is merely a downcast, it can be performed with a `static_cast`, not a `reinterpret_cast`. But downcasting is not always this simple. If the `DERIVED` class was derived from two copies of class `base` class (say, from a `base` mother and a `base` paternal grandparent), the downcast would have no way to tell which of the two `base` objects it is receiving the address of. For an upcast, see p. 544.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/meta/polymorphism2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 /*
 6 DERIVED must be publicly derived from class base
 7 and must have a member function named implementation.
 8 Class DERIVED can have only at most one class base among its ancestors.
 9 interface can be called only when the base object is part of a DERIVED object.
10 */
11
12 template <class DERIVED>
13 class base {
14 public:
    void interface() const {
        static_cast<const DERIVED *>(this)->implementation();
    }
};

class derived1: public base<derived1> {
public:
    void implementation() const {cout << "derived1\n";}
};

class derived2: public base<derived2> {
public:
    void implementation() const {cout << "derived2\n";}
};

int main()
{
    derived1 d1;
    d1.interface();

    derived2 d2;
    d2.interface();

    cout << "sizeof (base<derived1>) == " << sizeof (base<derived1>)
        << "\n"
        << "sizeof (base<derived2>) == " << sizeof (base<derived2>)
        << "\n"
        << "sizeof (derived1) == " << sizeof (derived1) << "\n"
        << "sizeof (derived2) == " << sizeof (derived2) << "\n";

    return EXIT_SUCCESS;
}
```

```
derived1
derived2
sizeof (base<derived1>) == 1
sizeof (base<derived2>) == 1
sizeof (derived1) == 1
sizeof (derived2) == 1
```

7.2.5 Explicit Instantiation

A template is *instantiated* when we make the computer behave as if we had pasted a copy of the template into the program, changing each template argument to what it stands for. The most common way to instantiate a template class is by constructing an object of that class. A template class can also be instantiated without actually constructing an object.

How can we confirm the instantiation if no object of that class is constructed? In fact, why would we want to do this at all? Well, we might want to instantiate the following template class `absent` in order to construct its static data members. In fact, we'll instantiate it twice in order to construct the static data members `absent<int>::s` and `absent<double>::s`.

The output of lines 6 and 7 of `main.C` confirm the two instantiations. The lack of output from line 12 of `absent.h` confirms that no object of class `absent<int>` or `absent<double>` has been constructed.

Class `obj` was on pp. 179−180. For another example where a class must be explicitly instantiated, see line 27 of `main.C` on p. 733.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/instantiate_class/absent.h`

```
#ifndef ABSENTH
#define ABSENTH
#include <iostream>
#include "obj.h"
using namespace std;

template <class T>
class absent {
    static const obj s;
    T t;
public:
    absent(const T& initial_t): t(initial_t) {cout << "constructed\n";}
};

template <class T>
const obj absent<T>::s = static_cast<int>(sizeof (T));
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/instantiate_class/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "absent.h"
 4 using namespace std;
 5
 6 template class absent<int>;
 7 template class absent<double>;
 8
 9 int main()
10 {
11     return EXIT_SUCCESS;
12 }
```

```
construct 4      line 6 of main.C instantiates class absent<int>
construct 8      line 7 instantiates class absent<double>
destruct 8       line 11 destructs the statically allocated objects
destruct 4
```

Derive a template class from a base class

As the above output shows, each instantiation of a template class has its own copy of a static data member. To make all the instantiations share the same copy, the member can be placed in a non-template base class. The template classes can then be derived from the base class. For an example, see `curious1.C` on pp. 715−716.

Explicit instantiation of a template function

Lines 3 and 4 of the following `f.C` instantiate a template function without calling it, possibly to place the instantiations in a library. The details are platform dependent, of course. For another example of instantiating a function without calling it, see line 12 of `function.C` on p. 781.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/instantiate_function/f.h`

```
#ifndef FH
#define FH
#include <iostream>
using namespace std;

template <class T>
inline void f() {cout << sizeof (T) << "\n";}
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/instantiate_function/f.C`

```
#include "f.h"

template void f<int>();
template void f<double>();
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/instantiate_function/main.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 #include "f.h"
4 using namespace std;
5
6 int main()
7 {
8     return EXIT_SUCCESS;
9 }
```

The symbol table for the ‘‘object’’ file `f.o` shows that `f` has been instantiated twice. The `-C` option unmangles the name of the function.

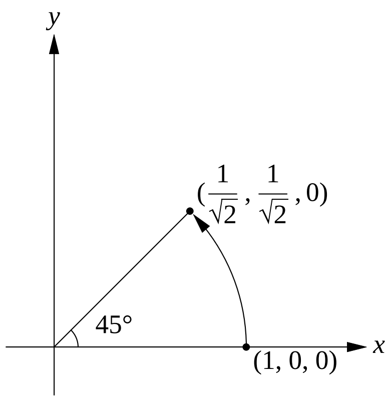
```
1$ g++ -c f.C
2$ ls -l f.o
3$ nm -C f.o | egrep '^\[Index\]|f<.*>'
[Index]   Value      Size      Type  Bind  Other Shndx   Name
[23]|         0|        56|FUNC |WEAK |0    |6      |void f<double>()
[18]|         0|        56|FUNC |WEAK |0    |5      |void f<int>()
```

7.2.6 Template Member Functions

A member function of a template class is *ipso facto* a template function. A member function can also be a template function in its own right, regardless of whether its class is a template class.

A template member function of a non-template class

The following class represents a point in a three-dimensional space, with member functions for rotating the point around the X, Y, or Z axis. The point $(1, 0, 0)$ in the diagram is on line 11 of `main.C` on p. 723; its 45° rotation around the Z axis is in line 14. Since the point lies in the X-Y plane, and the Z axis rises vertically out of the plane, the point simply rotates around the origin.



—On the Web at
`http://i5.nyu.edu/~mm64/book/src/rot1/point.h`

```
#ifndef POINTH
#define POINTH
#include <iostream>
using namespace std;

class point {
    double x;   //Cartesian coordinates
    double y;
    double z;
public:
    point(double initial_x = 0, double initial_y = 0, double initial_z = 0)
        : x(initial_x), y(initial_y), z(initial_z) {}

    point& xrot(double theta);   //theta in radians
    point& yrot(double theta);
    point& zrot(double theta);

    friend ostream& operator<<(ostream& ost, const point& p) {
        return ost << "(" << p.x << ", " << p.y << ", " << p.z << ")";
    }
};
#endif
```

The three rotation functions are identical except for their choice of data members. They convert the point's coördinates from Cartesian to polar *r*, θ in the plane of rotation, perform the rotation, and convert them back. As on p. 364, we avoid calling `atan2` with two zero arguments.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/rot1/point.C`

```
#include <cmath>   //for atan2
#include "point.h"
using namespace std;

//Rotate this point around the X axis.

point& point::xrot(double theta)
{
    if (y != 0.0 || z != 0.0) {  //if this point is not on the X axis,
        const double r = sqrt(y * y + z * z);
        theta += atan2(z, y);
        y = r * cos(theta);
        z = r * sin(theta);
    }
    return *this;
}

//Rotate this point around the Y axis.

point& point::yrot(double theta)
{
    if (z != 0.0 || x != 0.0) {  //if this point is not on the Y axis,
        const double r = sqrt(z * z + x * x);
        theta += atan2(x, z);
        z = r * cos(theta);
        x = r * sin(theta);
    }
    return *this;
}

//Rotate this point around the Z axis.

point& point::zrot(double theta)
{
    if (x != 0.0 || y != 0.0) {  //if this point is not on the Z axis,
        const double r = sqrt(x * x + y * y);
        theta += atan2(y, x);
        x = r * cos(theta);
        y = r * sin(theta);
    }
    return *this;
}
```

The following line 14 multiplies the 45 degrees by $\frac{\pi}{180}$ to convert it to radians. Each function returns `*this`, allowing lines 20−21 to chain the calls together and print the new value of the `point`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/rot1/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cmath>
 4 #include "point.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     const double pi = 4 * atan2(1, 1);
10
11     point p(1, 0, 0);
12     cout << p << "\n";
13
14     p.zrot(45 * pi / 180);
15     cout << p << "\n";
16
17      p.xrot(45 * pi / 180);
18      cout << p << "\n";
19
20      cout << p.xrot(-45 * pi / 180)        //Undo the previous rotations.
21               .zrot(-45 * pi / 180) << "\n";
22
23      return EXIT_SUCCESS;
24  }
```

The `0.707107` represents $\frac{1}{\sqrt{2}}$. The `-7.85046e-17` should have been a perfect zero, but the point didn't quite come back to its original position.

```
(1, 0, 0)                       lines 11−12
(0.707107, 0.707107, 0)         lines 14−15: rotate around the Z axis
(0.707107, 0.5, 0.5)            lines 17−18: rotate around the X axis
(1, 0, -7.85046e-17)            lines 20−21: back to original position
```

Instead of writing the same member function three times, we can define it once and for all as a *template member function.* Since it is a template function, its declaration in the following line 16 and its definition in 30 have a preamble. The `A` and `B` in the preamble are constant template arguments of type "pointer to `double` data member of class `point`". As on p. 254, this type of pointer can be dereferenced only with `.*` and `->*`. Since these operators are binary, an operand must be written in front of them, the unfortunate `this` in lines 33−38. Of course, the template arguments could also be of a less exotic type.

A copy constructor cannot be a template member function, because a class can have only one copy constructor. Also, a virtual member function cannot be a template function. If it were, the vtbl would be infinitely large.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/rot2/point.h`

```
 1  #ifndef POINTH
 2  #define POINTH
 3  #include <iostream>
 4  #include <cmath>
 5  using namespace std;
 6
 7  class point {
 8      double x;   //cartesian coordinates
 9      double y;
10      double z;
11  public:
12      point(double initial_x = 0, double initial_y = 0, double initial_z = 0)
13          : x(initial_x), y(initial_y), z(initial_z) {}
14
15      template <double point::*A, double point::*B>
16      point& rot(double theta);   //theta in radians
17
18      friend ostream& operator<<(ostream& ost, const point& p) {
19          return ost << "(" << p.x << ", " << p.y << ", " << p.z << ")";
20      }
21  };
22
23  /*
24  Rotate this point around the axis that is neither A nor B.
25  A positive theta rotates in the direction from the positive half of the A axis
```

```
26 towards the positive half of the B axis.
27 */
28
29 template <double point::*A, double point::*B>
30 point& point::rot(double theta)
31 {
32     //if this point is not on the axis,
33     if (this->*A != 0.0 || this->*B != 0.0) {
34         const double r =
35             sqrt(this->*A * this->*A + this->*B * this->*B);
36         theta += atan2(this->*B, this->*A);
37         this->*A = r * cos(theta);
38         this->*B = r * sin(theta);
39     }
40     return *this;
41 }
42 #endif
```

To call the template member function, change lines 14−21 of `main.C` on pp. 723−724 to the following. To mention the data members `x`, `y`, and `z` in `main`, they must become public.

```
43     p.rot<&point::x, &point::y>(45 * pi / 180);
44     cout << p << "\n";
45
46     p.rot<&point::y, &point::z>(45 * pi / 180);
47     cout << p << "\n";
48
49     cout << p.rot<&point::y, &point::z>(-45 * pi / 180)
50              .rot<&point::x, &point::y>(-45 * pi / 180) << "\n";
```

The output should remain the same. The source code looks grim, but the next group of homeworks will clean it up.

▼ Homework 7.2.6a: call the template member function from a template

There might be many classes of objects that we want to rotate around an axis: a point, a circle, a rectangle. The following `spin` function is therefore a template function. In line 10 it should accept the address of the `point p` in line 19. But the explicit template arguments `<&T::x, &T::y>` in line 13 do not compile.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/rot2/spin.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cmath>
 4 #include "point.h"
 5 using namespace std;
 6
 7 //T must have data members x and y, and template member function rot.
 8
 9 template <class T>
10 void spin(T *p)
11 {
12     static const double pi = 4 * atan2(1, 1);
13     p->rot<&T::x, &T::y>(45 * pi / 180);
14 }
```

```
15
16 int main()
17 {
18     point p(1, 0, 0);
19     spin(&p);
20     cout << p << "\n";
21     return EXIT_SUCCESS;
22 }
```

What happens when the computer first sees the above line 13? It knows that `rot` is a member of `T`, thanks to the `->` in front of it. It also knows that `rot` is not the name of a data type, thanks to the absence of `typename` in front of it. The computer might even figure out that `rot` is a member function of `T`, thanks to the argument list in parentheses. But the computer will not believe that `rot` is a *template* member function of `T`. It thinks that the angle brackets are the "less than" and "greater than" operators, resulting in cryptic error messages.

```
spin.C: In function 'void spin(T*) [with T = point]':
spin.C:19:9:   instantiated from here
spin.C:13:2: error: invalid operands of types '<unresolved overloaded
function type>' and 'double point::*' to binary 'operator<'
spin.C:13:2: error: invalid operands of types 'double point::*' and
'double' to binary 'operator>'
```

To tell the computer that the `rot` in the above line 13 is a template member function, insert the keyword `template` immediately before the `rot`.

▲

▼ Homework 7.2.6b: simplify the function calls

To spare the user the pain of calling the `rot` template function directly, give class `point` the following three inline public member functions.

```
    //Rotate this point around the X, Y, or Z axis.
    point& xrot(double theta) {return rot<&point::y, &point::z>(theta);}
    point& yrot(double theta) {return rot<&point::z, &point::x>(theta);}
    point& zrot(double theta) {return rot<&point::x, &point::y>(theta);}
```

You can now change `main` back to its original wording. The `rot` template function can be private, and the data members can be private again.

▲

▼ Homework 7.2.6c: simplify the function definition

Introduce two references, `a` and `b`, to make the `rot` template function more legible.

```
template <double point::*A, double point::*B>
point& point::rot(double theta)
{
    double& a = this->*A;
    double& b = this->*B;

    if (a != 0.0 || b != 0.0) {  //if this point is not on the axis,
        const double r = sqrt(a * a + b * b);
        //etc.
```

▲

▼ Homework 7.2.6d: a template member function

A series of function calls that use the same variable should be packaged as an object. The functions should be member functions; the variable should be a data member. See p. 177.

We saw a series of calls in lines 11−14 of `step.C` on p. 656. Package them as an object, with the data member `p` in the following line 2 and the template member functions in lines 16 and 19. The `print` functions can be member functions too.

```
 1 class stepper {
 2     const void *p;
 3 public:
 4     stepper(const void *initial_p): p(initial_p) {}
 5     stepper operator=(const void *new_p) {p = new_p; return *this;}
 6
 7     template <class T>
 8     static void print(const T& t) {cout << t;}
 9
10     static void print(unsigned char c) { /* etc. */ }
11
12     //etc.: static print member functions for other data type(s)
13     //that require special handling
14
15     template <class T>
16     const T& stand() const { /* etc. */ }
17
18     template <class T>
19     const T& step() { /* etc. */ }
20 };
```

Since the `step` member function is called the most frequently, I would like to name it `operator()` as on p. 299. But an explicit template argument can be applied to an `operator` function only if we write the name of the function in full; see p. 659.

▲

A template member function of a template class

Ever wonder how class `vector` got so many two-argument constructors? The following line 14 calls a constructor that takes two pointers, 15 the one that takes two `list<obj>` iterators (pp. 179−180), and 16 the one that takes two `vector<int>` iterators. Where did all these constructors come from? How many are there?

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/template_constructor/main1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include <list>
 5 #include "obj.h"
 6 using namespace std;
 7
 8 int main()
 9 {
10     const int a[] = {10, 20, 30};
11     const size_t n = sizeof a / sizeof a[0];
12     list<obj> li(a, a + n);
13
```

```
    vector<int> v1(a, a + n);                //born containing 10, 20, 30
    vector<int> v2(li.begin(), li.end());    //born containing 10, 20, 30
    vector<int> v3(v2.begin() + 1, v2.end()); //born containing 20, 30

    return EXIT_SUCCESS;
}
```

The following class `vector` shows how they were defined, without bothering to actually hold any values. It is a template class, with the familiar `<class T>` preamble in lines 6 and 16. Its constructor is a template member function, with its own `<class ITERATOR>` preamble in lines 9 and 17. The function definition at line 18 has both preambles. Do not attempt to combine them.

Line 18 is a two-argument constructor taking any pair of *iterators*—variables to which lines 21−22 can apply the operators `!=`, `*`, and `++`. Specifically, the arguments are what pp. 834−837 will call ‘‘input iterators’’, which can be used to read a series of values. The standard library assumes that an iterator can be passed by value; we follow this convention here.

For a template member function defined inside the template class definition, see line 13.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/template_constructor/vector.h`

```
 1 #ifndef VECTORH
 2 #define VECTORH
 3 #include <iostream>
 4 using namespace std;
 5
 6 template <class T>
 7 class vector {
 8 public:
 9     template <class ITERATOR>
10     vector(ITERATOR first, ITERATOR last);           //declaration
11
12     template <class ITERATOR>
13     void f(ITERATOR it) {}              //declaration and definition
14 };
15
16 template <class T>
17 template <class ITERATOR>
18 ::vector<T>::vector(ITERATOR first, ITERATOR last)   //definition
19 {
20     cout << "Constructing a vector that contains";
21     for (; first != last; ++first) {
22         cout << " " << *first;
23     }
24     cout << ".\n";
25 }
26 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/template_constructor/main2.C`

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include <list>
#include "obj.h"
#include "vector.h"
```

```
 7 using namespace std;
 8
 9 int main()
10 {
11     const int a[] = {10, 20, 30};
12     const size_t n = sizeof a / sizeof a[0];
13     list<obj> li(a, a + n);
14     std::vector<int> v(a, a + n);
15
16     ::vector<int> v1(a, a + n);
17     ::vector<int> v2(li.begin(), li.end());
18     ::vector<int> v3(v.begin() + 1, v.end());
19
20     return EXIT_SUCCESS;
21 }
```

```
construct 10                                        Line 13 constructs a list.
copy construct 10
destruct 10
construct 20
copy construct 20
destruct 20
construct 30
copy construct 30
destruct 30
Constructing a vector that contains 10 20 30.       line 16
Constructing a vector that contains 10 20 30.       line 17
Constructing a vector that contains 20 30.          line 18
destruct 10                                         Line 20 destructs the list.
destruct 20
destruct 30
```

7.2.7 “One-to-Many” and “Many-to-Many” Friendships

Our template class `stack` and its `operator==` friend enjoy the kind of friendship that you probably want for your template classes and their friends. Each instantiation of `operator==` is a friend of, and takes arguments of, the corresponding instantiation of `stack`. Thus `operator==<int>` is a friend of `stack<int>`; `operator==<double>` is a friend of `stack<double>`. See line 29 of `stack.h` on p. 685.

There are actually three possible correspondences between a friend function and a template class:

(1) One-to-many. A non-template function can be a friend of every instantiation of a template class.

(2) One-to-one. Each instantiation of a template function can be a friend of the corresponding instantiation of a template class. The function and class must agree in the number and type of their template argument(s). For example, `operator==` and `stack` both have the template argument list `class T`.

(3) Many-to-many. Every instantiation of a template function can be a friend of every instantiation of a template class. The function and class do not have to agree in the number and type of their template argument(s).

Of course, a non-template class can also have a friend function. This friend can be a non-template function (one-to-one) or any instantiation of a template function (many-to-one). But these combinations are completely straightforward, so only template classes are discussed here.

One-to-many

The template class `wrapper` has the data member `t` in the following line 7, the constructor in 9, and precious little else. The non-template function `outside` in line 10 is a friend of every instantiation of `wrapper`. To demonstrate this universal friendship, it mention the private member `t` of three different instantiations. (For each instantiation, an anonymous object is constructed.)

Surprisingly, `outside` was able to achieve these friendships without being a template function. But `outside` had to be defined outside the body of the class definition, and it had to be defined after the class definition since it mentions `t`. Had `outside` been defined within the class definition, it would have been instantiated—copied and pasted into the program—every time the class template was instantiated. This could cause `outside` to be multiply defined, or not defined at all.

But don't probe the limits of what will compile. Please define this kind of friend outside the class definition, at line 13, allowing the tem,plate to be instantiated any number of times.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/many/one_to_many.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 template <class T>
 6 class wrapper {
 7     T t;
 8 public:
 9     wrapper(const T& initial_t = T()): t(initial_t) {}
10     friend void outside();
11 };
12
13 void outside()
14 {
15     cout << wrapper<bool>().t << "\n";
16     cout << wrapper<int>().t << "\n";
17     cout << wrapper<double>().t << "\n";
18 }
19
20 int main()
21 {
22     cout << boolalpha << fixed;
23     outside();
24     return EXIT_SUCCESS;
25 }
```

```
false                  Line 23 calls outside.
0
0.000000
```

A non-template function with a T

Like the above functions, the following `outside` and `inside` in lines 14 and 15 are friends of every instantiation of the class. Oddly, their declarations can mention `T` even though they are not template functions. Each instantiation of the class defines another `inside` and declares another `outside`. A separate definition has to be written for each `outside` function (lines 18−20); the GNU `g++` compiler questions our judgement in undertaking this obligation. We recommend that you define this kind of friend inside the class definition.

If the functions had a `T` in their body or return value, but not in their arguments, function name overloading would be impossible. In this case, we would be unable to instantiate the class more than once. The GNU `g++` compiler sometimes lets us get away with this, but it shouldn’t.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/many/t.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 bool inside(const bool& t1);
 6 int inside(const int& t1);
 7 double inside(const double& t1);
 8
 9 template <class T>
10 class wrapper {
11         T t;
12 public:
13         wrapper(const T& initial_t = T()): t(initial_t) {}
14         friend T outside(const T& t1);
15         friend T inside(const T& t1) {return wrapper<T>(t1).t;}
16 };
17
18 inline bool   outside(const bool& b)   {return wrapper<bool>(b).t;}
19 inline int    outside(const int& i)    {return wrapper<int>(i).t;}
20 inline double outside(const double& d) {return wrapper<double>(d).t;}
21
22 int main()
23 {
24     cout << boolalpha
25         << outside(true) << "\n"
26         << outside(10) << "\n"
27         << outside(3.14) << "\n"
28
29         << inside(true) << "\n"
30         << inside(10) << "\n"
31         << inside(3.14) << "\n";
32
33         return EXIT_SUCCESS;
34 }
```

```
t.C:14:37: warning: friend declaration ’T outside(const T&)’ declares a
non-template function
t.C:14:37: note: (if this is not what you intended, make sure the
function template has already been declared and add <> after the
function name here)
true        Lines 25−27 call outside.
10
3.14
true        Lines 29−31 call inside.
10
3.14
```

One-to-one with a template function

We have already endorsed the `operator==` template function for its one-to-one friendship with a template class (line 29 of `stack.h` on p. 685). Here its is again, with some notes on portability.

Each instantiation of `operator==` is a friend of the corresponding instantiation of `wrapper`. `operator==` and `wrapper` must agree in the number and type of their template arguments (lines 8 and 11). `<class T>` and `<class T>`. Since they do, we can omit the leftmost `T` in lines 18 and 20, although the <angle brackets> must remain. But before these lines can apply `<T>` or `<>` to the name `operator==`, there must be a prior declaration that `operator==` is a template function (line 9). And before line 9 can apply `<T>` to the name `wrapper`, there must be a prior declaration that `wrapper` is a template class (line 6).

The GNU `g++` forces us to define `operator==` outside the body of the class definition, at line 28. Line 20 is rejected, possibly because it looks like a partial specialization of a template function (p. 702). The program can be conditionally compiled with the macro `__GNUC__` (four underscores), predefined for the `GNU` compiler.

We recommend that you define this kind of friend outside the class definition, in line 28, thus satisfying every compiler. That's what we did in line 71 of `stack.h` on p. 686.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/many/one_to_one1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 template <class T>
 6 class wrapper;
 7
 8 template <class T>
 9 bool operator==(const wrapper<T>& w1, const wrapper<T>& w2);
10
11 template <class T>
12 class wrapper {
13     T t;
14 public:
15     wrapper(const T& initial_t = T()): t(initial_t) {}
16
17 #ifdef __GNUC__
18     friend bool operator==<T>(const wrapper<T>& w1, const wrapper<T>& w2);
19 #else
20     friend bool operator==<T>(const wrapper<T>& w1, const wrapper<T>& w2) {
21         return w1.t == w2.t;
22     }
23 #endif
24 };
25
26 #ifdef __GNUC__
27 template <class T>
28 inline bool operator==(const wrapper<T>& w1, const wrapper<T>& w2) {
29     return w1.t == w2.t;
30 }
31 #endif
32
33 int main()
34 {
```

```
35     cout << boolalpha << (wrapper<int>() == wrapper<int>()) << "\n";
36     return EXIT_SUCCESS;
37 }
```

```
true
```

Many-to-many

Every instantiation of the template functions in the following lines 13 and 16 is a friend of every instantiation of the class. The functions and the class need not agree in the number and type of their template arguments. To emphasize this, the preambles in line 6 and 15 are totally different.

I’m sorry that `outside` needs the additional template argument `U` in line 12. I hoped I could eliminate the `class U`, and change the `const U& u` in line 13 to `const T& t1`. But when I tried it, I had to remove the `class U` from line 22. I then had to write a copy of lines 22−25 with each `U` changed to `int` (because of the `10` in line 31), and another copy of lines 22−25 with each `U` changed to `unsigned` (because of the `10u` in line 33).

I also tried to change the `U` to `T` in lines 22 and 23. For consistency, I then wanted to make the same change in lines 12 and 13. But a `T` in line 12 would conflict with the `T` in line 6. We recommend that you define this kind of friend inside the class definition.

Line 27 instantiates the class so that `inside` will be declared before we get to line 31.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/many/many_to_many.C`

```
 1 #include <iostream>
 2 #include <iomanip>
 3 #include <cstdlib>
 4 using namespace std;
 5
 6 template <class T>
 7 class wrapper {
 8     T t;
 9 public:
10     wrapper(const T& initial_t = T()): t(initial_t) {}
11
12     template <int BASE, int WIDTH, class U>
13     friend void outside(const U& u);
14
15     template <int BASE, int WIDTH>
16     friend void inside(const T& t1) {
17         cout << setbase(BASE) << setw(WIDTH)
18             << wrapper<T>(t1).t << "\n";
19     }
20 };
21
22 template <int BASE, int WIDTH, class U>
23 inline void outside(const U& u) {
24     cout << setbase(BASE) << setw(WIDTH) << wrapper<U>(u).t << "\n";
25 }
26
27 template class wrapper<double>;  //explicit instantiation
28
29 int main()
30 {
    outside<16, 2>(10);    //The 10 changes U to int.
    inside<10, 2>(10);
    outside<16, 2>(10u);   //The 10u changes U to unsigned.
    inside<10, 2>(10u);
    return EXIT_SUCCESS;
}
```

```
 a
10
 a
10
```

7.2.8 Uncouple the Data Types

A breadboard for plugging data types together

We introduced templates as glorified carbon paper, letting us avoid writing the same source code over and over. But we can also think of templates as a way of building bigger data types out of smaller ones, fully coequal with the aggregation and inheritance on p. 257.

Consider how we joined classes `brownian` and `victim_t` with private multiple inheritance to create the following class. See lines 18 and 26 on p. 696.

```
grandchild<brownian, victim_t, 'r'>
```

The user who plugs the template arguments into the above <angle brackets> does not need to worry about private vs. public, single vs. multiple. The glue that joins the arguments is hidden by the template. Perhaps a `grandchild` contains a `brownian`, or contains a pointer or reference to a `brownian`, or is derived from class `brownian`, either publicly or privately. Or perhaps there is exactly one `brownian` object, shared by all the `grandchild` objects. Or perhaps no `brownian` object exists at all, and a `grandchild` object merely calls the static member functions of class `brownian`.

The arguments and angle brackets can even be hidden in a typedef. The person writing line 27 in the code on p. 696 does not even need to know that a template is involved.

The standard library has many classes in which a template hides how the data types were plugged together. Consider the familiar class `vector`.

```
#include <vector>
#include "date.h"
using namespace std;

    vector<date> v;
```

Class `vector` actually accepts a second template argument. We never had to write it because it has the following default:

```
    vector<date, allocator<date> > v;
```

The member functions of class `allocator<date>` perform the dynamic memory allocation and deallocation for a growing and shrinking `vector<date>`. Once again, the template hides the exact relationship between the data types. Does a `vector` contain an `allocator` or a pointer thereto? Or does a `vector` call the static member functions of class `allocator`? The user does not need to know.

An interface for keeping the data types separate

Pages 163−179 presented four reasons to package a chunk of code or functionality as a class. A fifth reason is that a class is a unit of syntax that can be plugged into—or withheld from—a template. We will see three examples.

(1) The error checking in lines 39−43 of `printable.h` on p. 736 has been packaged as a class `printable` so that it can be plugged into—or withheld from—the `terminal` template on pp. 740−745.

(2) The `>` operator in line 14 of `main3.C` on p. 768 has been packaged as a class `greater_int` so that it can be plugged into the `sorter` template in line 17 of `sorter2.h` on p. 767.

(3) A `wabbit`'s style of motion has been packaged as classes such as `brownian` and `manual` so that it can be plugged into the `grandchild` template on pp. 695−696.

If the template is a function template, we can pass it more than just a template argument `T`. We can also pass it a function argument whose data type is `T`. For example, `T` could be a class and the function argument could be an object of that class, carrying data members. An example is in line 59 of `main4.C` on p. 771. Or `T` could be a the data type of a pointer to a function, and the function argument could be a pointer of that type. See line 55 of `main4.C` on p. 771.

The template class `grandchild` was a breadboard for plugging other data types together (pp. 695−696). In the following example, a template will keep the data types `printable` and `terminal` cleanly separated from each other. These two ways of using templates are opposite sides of the same coin.

Until now, our C++ code has been a straightforward extension of C. An object is just a glorified structure; a virtual member function is a call through a pointer; a template is an overgrown macro. But starting here, and culminating in Chapters 8 and 9, these features will come together in a synthesis that has no counterpart in C. Functions, classes, objects, and templates will interpenetrate in new ways. The very appearance of the code on the page will become remote from anything seen in a C program.

Class printable

Here is the class `printable` we wrote on pp. 343−344, upgraded to throw the exceptions on pp. 628−629. Its heart is the `char` data member in the following line 11.

A `printable` object has the look and feel of a `char`. Whenever we try to read the object's value, the `operator char` in line 33 is transparently called. For example, line 14 of `main.C` on p. 738 behaves as if we had written the comment alongside.

But a `printable` object will accept only printable values. Whenever we try to write a value into an object, the `operator=` member function in line 37 of `printable.h` is ultimately called. This function will reject non-printing characters such as `'\a'` (alarm), `'\b'` (backspace), and `'\f'` (formfeed). It will store only a printable value into the data member of a `printable` object.

For example, line 12 of `main.C` calls the constructor in line 31 of `printable.h`, which calls the `operator=` in line 37. And the `operator +=` in line 49 of `printable.h` calls `operator char` to read and `operator=` to write.

The `=` operator in line 37 is binary: it takes two operands, as in line 10 of `printable.C` on p. 738. An `operator=` must do nothing if its operands are the same variable (`x = x`). Most `operator=`'s therefore begin with an `if` to verify that their arguments are in fact two different variables. But the `if` is unnecessary here. We already know that the operands are different variables because of their different data types: the left operand (`*this`) is a `printable`, while the right operand (`t`) is a `T` that is not a `printable`. (If the right operand was a `printable`, we would have called the `operator=` function provided for us implicitly.) For an earlier `operator=` that needed no `if`, see p. 309. For the tests in lines 14 and 18 of `printable.h`, and the conversion in line 21, see pp. 343−344.

The `isprint` member function in line 13 calls the `isprint` function in the standard library. Without the `std::` in line 14, we would go into an infinite loop.

We wrote no copy constructor for class `printable` because we were satisfied with the one provided implicitly. It can assume that its argument is a legal `printable`. We also get an implicit `operator=` whose argument is a `printable`.

Class `printable` delivers no functionality other than the error checking in the following lines 39−41. We have packaged the error checking as a class so that we can plug it cleanly into—and unplug it cleanly from—the class `terminal` on pp. 740−745.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/printable2/printable.h`

```
#ifndef PRINTABLEH
#define PRINTABLEH
#include <iostream>
#include <sstream>  //for ostringstream
#include <cctype>   //for std::isprint
#include <climits>  //for UCHAR_MAX
#include "except.h"
using namespace std;

class printable {
    char c;

    static bool isprint(char c) throw () {
        return std::isprint(static_cast<unsigned char>(c)) != 0;
    }

    static bool isprint(int i) throw () {
        return i >= 0 && i <= UCHAR_MAX && std::isprint(i) != 0;
    }

    static unsigned code(char c) throw () {
        return static_cast<unsigned char>(c);
    }

    static int code(int i) throw () {return i;}
public:
    template <class T>   //T must be char or int.
    printable& operator=(T t) throw (except);

    template <class T>   //T must be char or int.
    printable(T t) throw (except) {*this = t;}

    operator char() const throw () {return c;}
};

template <class T>
printable& printable::operator=(T t) throw (except)
{
    if (!isprint(t)) {
        ostringstream ost;
        ost << "character code " << code(t) << " is not printable";
        throw except(ost);
    }

    c = t;
    return *this;
}

inline printable& operator+=(printable& p, int i) throw (except) {
    return p = p + i; //return p.operator=(p.operator char() + i);
}

```

```
53 inline printable& operator-=(printable& p, int i) throw (except) {
54     return p = p - i;
55 }
56
57 inline printable& operator++(printable& p) throw (except) {return p += 1;}
58 inline printable& operator--(printable& p) throw (except) {return p -= 1;}
59
60 inline const printable operator++(printable& p, int) throw (except) {
61     const printable old = p;
62     ++p;
63     return old;
64 }
65
66 inline const printable operator--(printable& p, int) throw (except) {
67     const printable old = p;
68     --p;
69     return old;
70 }
71
72 inline const printable operator+(printable p, int i) throw (except) {
73     return p += i;
74 }
75
76 inline const printable operator+(int i, printable p) throw (except) {
77     return p += i;
78 }
79
80 inline const printable operator-(printable p, int i) throw (except) {
81     return p -= i;
82 }
83
84 istream& operator>>(istream& ist, printable& p) throw (except);
85 #endif
```

The constructor for class `printable`, and many other functions that take and return `printable`, call each other. But this is no sin. All are inline, so no time is wasted.

(1) The `=`'s in lines 31, 50, and 54 of the above `printable.h`, and line 10 of `printable.C`, call the `operator=` in line 37 of `printable.h`. But the `=` in line 45 of `printable.h` does not call any `operator=` function. It is the built-in `=` that performs assignment to `char` or `int`.

(2) The `+=`'s in lines 57, 73, and 77 of `printable.h` call the `operator+=` in line 49. The `-=`'s in lines 58 and 81 of `printable.h` call the `operator-=` in line 53.

(3) The prefix `++` in line 62 of `printable.h` calls the prefix `operator++` in line 57. The prefix `--` in line 68 of `printable.h` calls the prefix `operator--` in line 58.

(4) The `+` and `-` in lines 50 and 54 of `printable.h` do not call the `operator+` and `operator-` in lines 72 and 80, since these functions have not yet been seen. (The rules will change on p. 751.)

(5) The rightmost `p` (the one used as an rvalue) in lines 50 and 54 of `printable.h` calls the `operator char` in line 34.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/printable2/printable.C`

```
1 #include <iostream>
2 #include "printable.h"
```

```
using namespace std;

istream& operator>>(istream& ist, printable& p) throw (except)
{
    char c;           //uninitialized variable

    if (ist >> c) {   //if (operator>>(ist, c).operator void *()) {
        p = c;        //p.operator=(c);
    }

    return ist;
}
```

We can easily combine the following lines 14−17 into a single statement, but this would complicate the comments.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/printable2/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "except.h"
 4 #include "printable.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     int status = EXIT_FAILURE;
10
11     try {
12         printable p = 'A';
13
14         cout << p;   //operator<<(cout, p.operator char());
15         cout << "\n";
16         cout << ++p; //operator<<(cout, p.operator++().operator char());
17         cout << "\n";
18
19         p = '\0';
20         status = EXIT_SUCCESS;
21     }
22
23     catch (const except& e) {
24         cerr << e.what() << "\n";
25     }
26
27     return status;
28 }
```

```
A
B              my machine is ASCII
character code 0 is not printable
```

Class terminal has the same error checking as class printable.

A well-written function begins by checking that its arguments are valid. For example, the following member function, originally appearing on p. 161, checks that its `char` argument is printable.

```
//Excerpt from terminal.C.

void terminal::put(unsigned x, unsigned y, char c) const
{
    if (isprint(static_cast<unsigned char>(c)) == 0) {
        ostringstream ost;
        ost << "unprintable character "
            << static_cast<unsigned>(static_cast<unsigned char>(c))
            << ".\n";
        throw except(ost);
    }

    check(x, y);
    term_put(x, y, c);
}
```

The error checking becomes more intrusive if we make it optional via conditional compilation, and also provide an exception specification.

```
void terminal::put(unsigned x, unsigned y, char c) const
#ifdef CHECK
    throw (except)
#endif
{
#ifdef CHECK
    if (isprint(static_cast<unsigned char>(c)) == 0) {
        ostringstream ost;
        ost << "unprintable character "
            << static_cast<unsigned>(static_cast<unsigned char>(c))
            << ".\n";
        throw except(ost);
    }
#endif

    check(x, y);
    term_put(x, y, c);
}
```

Now that we are checking the function arguments, we should also check the return values. For example, here is the `terminal::get` from p. 160.

```
//Excerpt from terminal.C, showing the definition of a public member function
//that was inline when it originally appeared.

char terminal::get(unsigned x, unsigned y) const
{
    check(x, y);
    return term_get(x, y);
}
```

With conditionally compiled error checking for printability, it needs an extra variable.

```
char terminal::get(unsigned x, unsigned y) const
```

```
#ifdef CHECK
    throw (except)
#endif
{
    check(x, y);

    const char c = term_get(x, y);
#ifdef CHECK
    if (isprint(static_cast<unsigned char>(c)) == 0) {
        ostringstream ost;
        ost << "unprintable character "
            << static_cast<unsigned>(static_cast<unsigned char>(c))
            << ".\n";
        throw except(ost);
    }
#endif
    return c;
}
```

Is there a less intrusive way to check that every character is printable? And if we did want to live fast and dangerously, is there a cleaner way to turn the checking on and off at compile time? We are about to see that the printability checking should never have been bundled together with class `terminal`. Templates will let us put asunder what should never have been joined.*

▼ Homework 7.2.8a:
Version 4.2 of the Rabbit Game: class `terminal` becomes a template class

Independent concepts should be independently represented . . .

—Bjarne Stroustrup, *The C++ Programming Language*, p. 327

The member functions of class `terminal` will check that their character arguments and return values are printable. To turn the checking on, the arguments and return values will be `printable` objects that are passed and returned by value. (We assume that anything playing the rôle of a character is fast enough to pass by value.) The checking will then be performed by the constructors for these objects. To turn the checking off, we will change the arguments and return values back to plain old `char`'s.

Class `terminal` will be a template class, just like `vector`, `list`, and all the other containers. It should have been a template class all along. Its template argument will be the data type of the characters passed to and from the member functions. This will let us turn the error checking on and off cleanly.

We will now be able to construct the terminals in lines 4−6. We will also have taken our first step towards the terminal in line 8.

```
 1 #include "printable.h"
 2 #include "terminal.h"
 3
 4 terminal<char> term1('.');
 5 terminal<> term2('.');           //the same data type
 6 terminal<printable> term3('.');
 7
 8 terminal<wchar_t> term4(L'.');   //w is "wide", L is "long"
```

```
 9 //This file is terminal.h, showing some of the members of class terminal.
10 #ifndef TERMINALH
11 #define TERMINALH
```

* For another example of counterproductive bundling, see p. 563.

```
12
13 #extern "C" {
14 #include "term.h"
15 }
16
17 template <class CHAR = char>
18 class terminal {
19     const CHAR _background;
20     const unsigned _xmax;
21     const unsigned _ymax;
22
23 public:
24     terminal(const CHAR& initial_char);
```

The `put` member function will be simplified to the following template. We turn the character error checking on and off, without any explicit conditional compilation, by our choice of the argument in the <angle brackets> in the following lines 31 and 38. Do not write an exception specification at the end of line 26.

```
25 template <class CHAR>
26 void terminal<CHAR>::put(unsigned x, unsigned y, CHAR c) const
27 {
28     check(x, y);
29     term_put(x, y, c);
30 }
```

If we say

```
31     terminal<char> term('.');
32     term.put(0, 0, 'A');
```

the above line 32 will instantiate line 26 as the following function. No error checking will be performed on the third argument:

```
33 void terminal::put(unsigned x, unsigned y, char c) const
34 {
35     check(x, y);
36     term_put(x, y, c);
37 }
```

But if we say

```
38     terminal<printable> term('.');
39     term.put(0, 0, 'A');
```

the above line 39 will instantiate line 26 as the following function.

```
40 void terminal::put(unsigned x, unsigned y, printable c) const
41 {
42     check(x, y);
43     term_put(x, y, c);
44 }
```

The third argument in the above line 40 is passed by value, calling a constructor for class `printable`. The third argument in the above line 39 was a `char`, so the constructor will be the one that takes a `char` in line 31 of `printable.h` on p. 736. It will throw an exception if its argument is not printable.

(1) Move the definitions of the non-inline member functions of class `terminal` from `terminal.C` to `terminal.h`. Then remove `terminal.C` entirely.

(2) Give the following preamble to the template class definition, as in the above line 17.

```
45 template <class CHAR = char>
```

But do not write the default value in the preamble for the definition of any non-inline member function, as in the above line 25. The default `= char` is written only once.

(3) Change the data member `_background` from `const char` to `const CHAR`. Change the following from `char` to `CHAR`: the return type of the member function `background`; the argument of the constructor for class `terminal`; the return value of the member function `get`; and the `char` argument of the member function `put`.

(4) The character `'\0'` is not printable, so there can be no `'\0'`-terminated arrays of `printable`'s. This means that the `terminal::put` whose third argument is a `CHAR *` is now useless and should be removed. Temporarily change the messages at the end of `game::play` to single character such as `'!'` or `'?'`. Don't worry: we will regain the ability to print a string of `printable`'s on p. 982.

(5) The return value of the member function `key` will remain `char`. `CHAR` is only for screen characters, not keystrokes.

(6) The arguments and return values of the C functions in `term.h` will remain `char`. C does not have templates.

(7) As in the above lines 11−23, `terminal::put` will no longer explicitly call `isprint`. We will now rely on the constructor for class `CHAR` to perform any checking that needs to be done.
`terminal.h` no longer needs to include `<cctype>`.

(8) Every container in the C++ Standard Library has a public member named `value_type`, which is a typedef for the data type of each element held in the container. For example,
`vector<int>::value_type` is a typedef for `int`:

```
46 #include <vector>
47 using namespace std;
48
49     vector<int> v(argument(s) for constructor);
50     if (!v.empty()) {
51         vector<int>::value_type x = v[0];   //x is int
52     }
```

Other examples of `value_type` were in lines 6, 10, 16−17 of `stack2.h` on pp. 153−154; lines 6 and 9 of `node.h` on p. 214; line 18 of `typename.C` on p. 675; line 16 of `stack.h` on p. 685.

What is `value_type` good for? After all, isn't it obvious that the above `vector<int>` would contain `int`'s and that `x` should therefore be an `int`? We use `value_type` when we *don't* know what type of container we're dealing with:

```
53 #define CONTAINER vector<int> //suppose this #define was off in another file.
54
55 void f(const CONTAINER& c)
56 {
57     if (!c.empty()) {
58         CONTAINER::const_iterator it = c.begin();
59         CONTAINER::value_type x = *it;
60     }
61 }
```

In contemporary C++, the opaque name `CONTAINER` is more likely to be the template argument in lines 62−63 than the macro in the above 53:

```
62 template <class CONTAINER>
63 void f(const CONTAINER& c)
64 {
65     if (!c.empty()) {
66         typename CONTAINER::const_iterator it = c.begin();
67          typename CONTAINER::value_type x = *it;
68      }
69  }
```

So add the following public member to the template class `terminal`:

```
70      typedef CHAR value_type;
```

(9) The following line compares two `char`'s in the original constructor for class `terminal` in line 9 of `terminal.C` on p. 160:

```
71      if (_background != ' ') {
```

The `_background` is now a `CHAR`, but the `' '` remains a `char`. Since we have not written an `operator!=` whose left and right operands are `CHAR` and `char`, the above line 71 will be torn between two equally good alternatives:

(a) it can convert the `' '` from `char` to `CHAR` and then perform a `CHAR` comparison; or

(b) it can convert the `_background` from `CHAR` to `char` and then perform a `char` comparison.

We will have to decide for it. Go with alternative (a) by writing

```
72          if (_background != static_cast<CHAR>(' ')) {
```

We have now finished modifying class `terminal`.

(10) If classes `game` and `wabbit` were changed into template classes, they would look as follows. (Just look—do not make this change.)

```
73  template <class CHAR>   //The forward declaration is now a template declaration.
74  class wabbit;
75
76  template <class CHAR>
77  class game {
78      typedef terminal<CHAR> terminal_t;
79      const terminal_t term;
80
81      typedef list<wabbit<CHAR> *> master_t;
82      master_t master;
83
84  public:
85      game(CHAR initial_c = '.'): term(initial_c) {}
86      //etc.
87  };
88
89  template <class CHAR>
90  class wabbit {
91      game<CHAR> *const g;
92      //etc.
93
94  public:
95      wabbit(game<CHAR> *initial_g, //etc.
96  };
```

We could now have several flavors of game in the same program:

```
97      game<char> g1;
98      game<printable> g2;
```

But let's not go this far. We don't need the multiple flavors and the resulting code has too many `<CHAR>`'s.

We will not change classes `game` and `wabbit` into template classes.

(11) Instead, simply write the typedef in line 106 and the protected typedef in line 124. Change every `terminal` to `terminal_t` in classes `game`, `wabbit`, and the classes derived from `wabbit`. The first example is in line 107.

(12) Every `char` that represents a character on the screen (as opposed to a keystroke from the keyboard) should be changed to `terminal_t::value_type` in classes `game`, `wabbit`, and the classes derived from `wabbit`. Examples are in lines 113, 116, 126 (which must come after 124), and 130.

The `char`'s in `manual::decide` will remain `char`'s. They represent keystrokes, not screen characters. The `char`'s in the array `a` in `game::game` will remain `char`'s. I just don't want to fool around with arrays of `printable` objects yet.

The typedef `terminal_t` in line 106 is a member of class `game`. It can be mentioned only in `game.h` and the files that include this header. The typedef `terminal_t` in line 124 is a member of class `wabbit`. It can be mentioned only in `wabbit.h` and the files that include this header. Therefore do not change `terminal` to `terminal_t`, and `char` to `terminal_t::value_type`, anywhere in the files `printable.h`, `printable.C`, `terminal.h`, `terminal.C`, `term.h`, `term.c`.

```
 99 //Excerpt from game.h.
100 #include "printable.h"
101 #include "terminal.h"
102
103 class wabbit;           //forward declaration
104
105 class game {
106     typedef terminal<printable> terminal_t;
107     const terminal_t term;
108
109     typedef list<wabbit *> master_t;
110     master_t master;
111
112 public:
113     game(terminal_t::value_type initial_c = '.')
114         : term(initial_c)           //etc.
115
116     master_t::size_type count(terminal_t::value_type c) const;
```

```
117 //Excerpt from wabbit.h.
118 #include "game.h"
119
120 class wabbit {
121     game *const g;
122     unsigned x, y;
123 protected:
124     typedef game::terminal_t terminal_t;
125 private:
126     const terminal_t::value_type c;
127
128 public:
129     wabbit(game *initial_g, unsigned initial_x, unsigned initial_y,
130         terminal_t::value_type initial_c);
```

(13) In `grandchild.h`, do not change

```
131 template <class MOTION, class RANK, char C>     //lowercase char
```

to

```
132 template <class MOTION, class RANK, CHAR C>     //uppercase char
```

A constant template argument cannot be an object.

▲

7.2.9 Altruistic Template Classes: `numeric_limits`, `iterator_traits`, etc.

An *altruistic* template class is one whose only purpose is to give us information about other data types. The class `numeric_limits<double>`, for example, will give us information about the data type `double`. There is never any reason to construct any object of this class: all of its data members and member functions are static. Furthermore, all the data members and all the return values of the member functions are constant values.

On pp. 754−755 we will use an altruistic class in a template to get information about a template argument `T`. Our first example, however, will use an altruistic class in `main`.

Numeric limits

A C program gets information about the numeric data types from macros in the C Standard Library. For example, the `INT_MAX` and `DBL_MAX` in the following lines 11 and 13 are the maximum value that each data type can hold.

Of greater practical importance, unless you are an astronomer, is the number of significant digits that a `double` can hold. Line 15 prints this number. In the output of line 16, the digits that came out correctly are underlined; there happens to be one more than expected.

Why can a `double` hold 15 decimal significant digits? A 15-significant-digit number can hold any whole number in the range 0 to $10^{15} - 1$ inclusive. (Of course, it could also hold fractions.) A `double` on my machine has a mantissa of 53 bits, so it can hold any whole number in the range 0 to $2^{53} - 1$ inclusive. (Of course, it could also hold fractions.) A `double` can hold any 15-significant-digit number because

$$10^{15} - 1 = 999,999,999,999,999 \le 9,007,199,254,740,991 = 2^{53} - 1$$

But not every 16-significant-digit number will fit into a `double`, because

$$2^{53} - 1 = 9,007,199,254,740,991 < 9,999,999,999,999,999 = 10^{16} - 1$$

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/numeric/limits.c`

```
 1 #include <stdio.h>   /* C example */
 2 #include <stdlib.h>
 3 #include <limits.h>  /* for INT_MIN, INT_MAX */
 4 #include <float.h>   /* for DBL_MAX, DBL_DIG */
 5
 6 int main()
 7 {
 8     double d = 123456789012345678.0;
 9
10     printf("Minimum int is %d.\n", INT_MIN);
11     printf("Maximum int is %d.\n\n", INT_MAX);
12
13     printf("Maximum double is %g.\n", DBL_MAX);
14     printf("A double has a %d-bit mantissa.\n", DBL_MANT_DIG);
15     printf("A double can hold %d decimal significant digits.\n", DBL_DIG);
16     printf("%.*g\n", DBL_DIG + 3, d);
17
18     return EXIT_SUCCESS;
19 }
```

Without the `.0` in the above line 8, the computer would think that the literal is of an integral data type. We would then get an error message if no integral type is big enough to hold this value.

```
Minimum int is -2147483648.
Maximum int is 2147483647.

Maximum double is 1.79769e+308.
A double has a 53-bit mantissa.
A double can hold 15 decimal significant digits.
12345678901234568 0
```

A C++ program would get information about data types from the template class `numeric_limits`. Only built-in numeric data types—`int`, `char`, `double`, but not pointers, enumerations or objects—can be plugged into the <angle brackets> of `numeric_limits`.

The static member functions `min` and `max` return the minimum and maximum possible values for a data type. Class `numeric_limits<int>` has the `min` function in the following line 9 that returns the smallest `int`; class `numeric_limits<double>` has the `max` function in 11 that returns the biggest `double`. This class also has the public static data member `digits10` in line 15 corresponding to the C macro `DBL_DIG`. Line 18 outputs three more than this number of digits; the sixteen that came out correctly are underlined.

The members of `numeric_limits` that give us a value of type `T` are member functions (`min`, `max`, etc). The members that always give us an integral or enumeration value are data members (`digits10`). This is because only integral or enumeration data members can be initialized in a class declaration; see p. 238.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/numeric/limits.C`

```
 1 #include <iostream>   //C++ example
 2 #include <iomanip>    //for setprecision
 3 #include <cstdlib>
 4 #include <limits>     //for numeric_limits
 5 using namespace std;
 6
 7 int main()
 8 {
 9     cout << "Minimum int is " << numeric_limits<int>::min() << ".\n"
10         << "Maximum int is " << numeric_limits<int>::max() << ".\n\n"
11         << "Maximum double is " << numeric_limits<double>::max() << ".\n"
12         << "A double has a " << numeric_limits<double>::digits
13         << "-bit mantissa.\n";
14
15     int prec = numeric_limits<double>::digits10;
16     cout << "A double can hold " << prec
17         << " decimal significant digits.\n";
18
19     double d = 123456789012345678.0;
20     cout << setprecision(prec + 3) << d << "\n";
21
22     return EXIT_SUCCESS;
23 }
```

The output may be different on your machine.

```
Minimum int is -2147483648.
Maximum int is 2147483647.

Maximum double is 1.79769e+308.
A double has a 53-bit mantissa.
A double can hold 15 decimal significant digits.
123456789012345680
```

Oddly, the value of the macro `CHAR_MAX` was of type `int` (`127` for a signed, 8-bit `char`). The value returned by `numeric_limits<char>::max` is of type `char`, which makes more sense.

There are two places where `numeric_limits` cannot be used, so don't discard your `_MIN` and `_MAX` macros yet. The C++ preprocessor does not know about `numeric_limits`, so `#if` directives will still have to be written in terms of the macros. And the value for a constant template argument must be a constant expression (p. 234), which does not allow function calls. The `INT_MIN` and `INT_MAX` will therefore have to remain in `rank<INT_MIN, INT_MAX>`. An example is in line 18 of `rank.h` on p. 694.

▼ Homework 7.2.9a: numeric limits

The `1.79769e+308` returned by `numeric_limits<double>::max()` on my machine is

$$(1 - 2^{-53}) \times 2^{1024}$$

My base is 2, my mantissa has 53 bits, and my maximum exponent is 1024. These values are available as three `int` data members of class `numeric_limits<double>`, or as three macros in the C++ header file `<cfloat>`.

	data members of `numeric_limits`	*macros in* `<cfloat>`
2	`radix`	`FLT_RADIX`
53	`digits`	`DBL_MANT_DIG`
1024	`max_exponent`	`DBL_MAX_EXP`

The $1 - 2^{-53}$ is the sum of the following series of 53 terms. It represents a mantissa of all 1's, visible as the numerators of the 53 fractions.

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots + \frac{1}{9,007,199,254,740,992} = 1 - \frac{1}{9,007,199,254,740,992}$$

A non-zero mantissa always starts with a 1 bit, not stored in memory, so the mantissa occupies only 52 bits. Together with the sign bit and the 11-bit exponent, this accounts for the 64 bits of a `double` on my machine.

Let's verify that $(1 - 2^{-53}) \times 2^{1024}$ is in fact equal to `1.79769e+308`. The standard library has several `pow` functions; to get the one we want, line 9 converts the `radix` from `int` to `double`. Unfortunately, the `pow` in line 12 cannot raise 2 to the power 1024, because 2^{1024} would be infinitesimally beyond the maximum `double` value of $(1 - 2^{-53}) \times 2^{1024}$. The workaround is to raise 2 to the power 1023 in line 12 and then double the result in line 14.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/numeric/max.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <limits>   //for numeric_limits
4 #include <cmath>    //for pow
5 using namespace std;
6
7 int main()
```

```
{
    const double radix = numeric_limits<double>::radix;

    double m = (1.0 - pow(radix, -numeric_limits<double>::digits))
        * pow(radix, numeric_limits<double>::max_exponent - 1);

    m *= radix;

    if (m == numeric_limits<double>::max()) {
        cout << fixed << m << "\n";   //fixed prevents rounding
    }

    return EXIT_SUCCESS;
}
```

```
179769313486231570814527423731704356798070567525844996598917476803157260
780028538760589558632766878171540458953514382464234321326889464182768467
546703537516986049910576551282076245490090389328944075868508455133942304
583236903222948165808559332123348274797826204144723168738177180919299881
250404026184124858368.000000
```

What is the next-to-largest value that a `double` can hold?

What is the smallest positive whole number that a `double` cannot hold? (It would be 9 if the mantissa consisted of 3 bits, no matter how big the exponent could be.) If you try to put this number into a `double`, which way will it round? See the `round_style` data member of class `numeric_limits<double>`, whose value is one of the `float_round_style` enumerations defined in `<limits>`.

▲

▼ **Homework 7.2.9b: ignore an input line**

To count the number of characters that we input or output, use the data type `streamsize` in line 10 of `double.C` on p. 355. This is also the data type of the first argument of the `ignore` function in `input.C` on p. 359. It ignores the specified number of characters, or everything up to and including a delimiter such as a newline, whichever comes first.

To place no ceiling on the number of characters ignored, let the first argument of `ignore` be the maximum value of the data type `streamsize`. With this special value, `ignore` will ignore everything up to and including the delimiter.

▲

An altruistic class with a data type member

The data types that hold characters are `char` and `wchar_t`. In C, the vital statistics for these types come from macros. For example, `EOF` and `WEOF` are the end-of-file value for each type, integers guaranteed to be different from any possible character code. We can store `EOF` or any `char` value into an `int`; we can store `WEOF` or any `wchar_t` value into a `wint_t`. The latter is a typedef for a machine-dependent integral type (probably `int` or `long`), so the `%d` in the following line 12 is not portable.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/numeric/char_traits.c`

```
 1 #include <stdio.h>  /* for EOF */
 2 #include <stdlib.h>
 3 #include <wchar.h>  /* for wint_t, WEOF */
 4
 5 int main()
```

```
{
    int c = EOF;
    printf("End-of-file integer for char is %d.\n", c);

    wint_t wc = WEOF;
    /* not portable: may be %ld on other machines */
    printf("End-of-file integer for wchar_t is %d.\n", wc);

    return EXIT_SUCCESS;
}
```

```
End-of-file integer for char is -1.
End-of-file integer for wchar_t is -1.
```

In C++, the vital statistics for `char` and `wchar_t` come from classes `char_traits<char>` and `char_traits<wchar_t>`. For example, the `eof` static member function in the following line 8 returns the end-of-file value for the type.

The `eof` function gives us a number. The `int_type` in the following line 8 gives us a data type. It is another name for the type of integer we could use to hold the end-of-file value: `int`, `long`, etc. The next altruistic class, `iterator_traits` on pp. 753−757, will have many members that are names of data types.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/numeric/char_traits.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <string>  //for char_traits
 4 using namespace std;
 5
 6 int main()
 7 {
 8     char_traits<char>::int_type c = char_traits<char>::eof();
 9     cout << "End-of-file integer for char is " << c << ".\n";
10
11     char_traits<wchar_t>::int_type wc = char_traits<wchar_t>::eof();
12     cout << "End-of-file integer for wchar_t is " << wc << ".\n";
13
14     return EXIT_SUCCESS;
15 }
```

```
End-of-file integer for char is -1.
End-of-file integer for wchar_t is -1.
```

▲

▼ Homework 7.2.9c: use char_traits to templatize class printable

The data type `char` was hardwired into class `printable` on pp. 343−344 and pp. 735−738. Parameterize the choice of character by turning `printable` into the template class in line 8, renamed `basic_printable` to agree with the convention on pp. 688−689. Define the class in a header file named `printable.h`; there will no longer be a `printable.C`. Every explicit specialization defined in `printable.h` must be `static`, or `inline` if it is short enough, so that the header file can be included in more than one `.C` file of the same program. Finally, reincarnate the data type `printable` as the typedef in line 35.

The template argument `CHAR` will be either `char` or `wchar_t`. These types, and the corresponding `int_type`'s, are fast enough to pass by value.

```
 1 #include <string>    //for char_traits
 2 #include "except.h"  //for except, pp. 628−629
 3 using namespace std;
 4
 5 //CHAR must be char or wchar_t.
 6
 7 template <class CHAR>
 8 class basic_printable {
 9     CHAR c;
10
11     static bool isprint(CHAR c) throw ();
12     static bool isprint(typename char_traits<CHAR>::int_type i) throw ();
13
14     static typename char_traits<CHAR>::int_type code(CHAR c) throw ();
15
16     static typename char_traits<CHAR>::int_type code(
17         typename char_traits<CHAR>::int_type i) throw () {
18         return i;
19     }
20 public:
21     basic_printable& operator=(CHAR c) throw (except);
22
23     basic_printable& operator=(typename char_traits<CHAR>::int_type i)
24         throw (except);
25
26     basic_printable(CHAR c) throw (except) {*this = c;}
27
28     basic_printable(typename char_traits<CHAR>::int_type i) throw (except) {
29         *this = i;
30     }
31
32     operator CHAR() const throw () {return c;}
33 };
34
35 typedef basic_printable<char> printable_t;
36 typedef basic_printable<wchar_t> wprintable_t;
37
38 //four explicit specializations of isprint
39 //two explicit specializations of code
40 //definitions for operator=, operator+=, operator++ (pre- & postfix), etc.
```

(1) For convenience, let the following `typedef` be a private member of `basic_printable` at the above line 10.

```
41     typedef typename char_traits<CHAR>::int_type int_type;
```

Then change `typename char_traits<CHAR>::int_type` to `int_type` in the rest of the class definition in the above lines 11−33.

(2) Define four explicit specializations for `basic_printable::isprint` at the above line 38. They must have no template preambles, as in line 44 of `wrapper.h` on p. 704.

(a) The `basic_printable<char>::isprint` whose function argument is a `char` will cast its function argument to `unsigned char` and pass it to the standard library `isprint`. You will have to refer to this function as `std::isprint`.

(b) The `basic_printable<char>::isprint` whose function argument is a `char_traits<char>::int_type` will verify that its function argument is greater than or equal to zero and less than or equal to `numeric_limits<unsigned char>::max()`. If so, it will pass its function argument to the standard library `isprint`.

(c) The `basic_printable<wchar_t>::isprint` whose function argument is a `wchar_t` will pass its function argument to the standard library `iswprint`; include the header `<cwctype>` for this function.

(d) The `basic_printable<wchar_t>::isprint` whose function argument is a `char_traits<wchar_t>::int_type` will verify that its function argument is ≥ `numeric_limits<wchar_t>::min()` and ≤ `numeric_limits<wchar_t>::max()`. If so, it will pass its function argument to the standard library `iswprint`.

(3) Define two explicit specializations for `basic_printable::code` at the above line 39. They must have no template preambles.

(a) The `basic_printable<char>::code` whose function argument is a `char` will convert the function argument to `unsigned char` and then to `char_traits<char>::int_type`. The first conversion must be explicit; the second can be implicit. Warning: the function argument of `basic_printable<char>::isprint` can be declared as an unadorned `int_type`, but the return type of `basic_printable<char>::code` must be declared as `basic_printable<char>::int_type`.

(b) The `basic_printable<wchar_t>::code` whose function argument is a `wchar_t` will convert the function argument directly to `char_traits<wchar_t>::int_type`.

(4) In the class `printable` on p. 736, an infinite loop would have resulted if `operator+=` (line 49) and `operator+` (line 72) called each other. We declared `operator+=` *before* we defined `operator+` to allow `operator+` to call `operator+=`. We declared `operator+` *after* we defined `operator+=` to prevent `operator+=` from calling `operator+`.

But now `operator+=` and `operator+` will be template functions. Each function can call the other, because the definition (instantiation) of each one can come after the declaration (template) for the other. To prevent an infinite loop, declare and define `operator+=` before `operator+`. In the body of `operator+=`, change

```
	//Would call operator+ before is declared!
	//return p.operator=(p.operator+(i));
	return p = p + i;
```

to

```
	//return p.operator=(p.operator CHAR() + i);
	return p = static_cast<CHAR>(p) + i;
```

For the only other place where something can be mentioned before it is declared, see p. 119.

(5) The standard libary contains a template similar to the following. Recall that the data type of `cout` is `ostream`, which is a typedef for `basic_ostream<char>`. Similarly, the data type of `wcout` is `wostream`, which is a typedef for `basic_ostream<wchar_t>`.

```
//CHAR must be char or wchar_t.

template <class CHAR>
basic_ostream<CHAR>& operator<<(basic_ostream<CHAR>& ost, CHAR c);
```

By itself, the template will let us send a `char` to an `ostream`. But it will not let us send a `printable<char>` there; see line 31 on p. 653. For this reason, the library also has a function similar to the following.

```
51 template <>
52 basic_ostream<char>& operator<<(basic_ostream<char>& ost, char c);
```

Similarly, the template in the above line 50 will let us send a `wchar_t` to a `wostream`, but it will not let us send a `printable<wchar_t>` there. The library has no template for `wchar_t` corresponding to the above line 52 so we must write our own:

```
53 inline wostream& operator<<(wostream& wost, basic_printable<wchar_t>& p) {
54         return wost << static_cast<wchar_t>(p);
55 }
```

Test `basic_printable` with the template class `stream` and the template function `f`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/numeric/main.C`

```
#include <iostream>
#include <cstdlib>
#include "printable.h"
using namespace std;

template <class CHAR>
struct stream {
    static basic_ostream<CHAR>& r;
};

template <>
basic_ostream<char>& stream<char>::r = cout;

template <>
basic_ostream<wchar_t>& stream<wchar_t>::r = wcout;

template <class CHAR>
void f()
{
    try {
        basic_printable<CHAR> p = static_cast<CHAR>('A');
        stream<CHAR>::r << ++p << static_cast<CHAR>('\n');

        p = static_cast<CHAR>('\0');
    }

    catch (const except& e) {
        cerr << e.what() << "\n";
    }

    stream<CHAR>::r << flush;
}

int main()
{
    f<char>();
    f<wchar_t>();
    return EXIT_SUCCESS;
}
```

We will have to define an operator== for basic_printable on p. 983.
▲

▼ **Homework 7.2.9d:**
Version 4.3 of the Rabbit Game: let the terminal contain printable_t objects

Run the game on a terminal<printable_t>, where printable_t is a typedef for the basic_printable<char> in the previous homework.
▲

Iterator traits

Our final example of an altruistic template class is iterator_traits. It gives us information about any data type that is an iterator, including any type of pointer except void * and const void *. iterator_traits is intended for use only within a template, to provide information about a template argument T. For simplicity, however, our first example will use it in main.

The data type plugged into the <angle brackets> of iterator_traits must be an iterator. Examples are the const int * in line 14 and the list<double>::iterator in line 19.

Each container has a value_type member giving the data type of the values stored in the container; our first example was in class stack on p. 423. The iterator_traits for each container's iterator has a member with the same name and almost the same purpose. It gives the data type of a variable that can hold a value stored and/or retrieved by the iterator.

For example, the *it1 in line 14 is a const int. To hold this value, the x1 in line 14 is an int. It is not necessary to make x1 a const int. We can remove the top-level const (p. 644).

We could simply have declared x1 to be an int. Another way to make x1 an int is to declare it to be an iterator_traits<int *>::value_type, which is another name (i.e., a typedef) for the data type that can hold the result of applying the * operator to a const int *.

Let's see one more example. The iterator it2 in line 19 is a vector<double>::iterator; we retrieve a double when we apply the operator * to it. To create a variable to hold this double, we could simply have declared x2 to be a double. The iterator_traits<vector<double>::iterator>::value_type in line 19 is a gloriously baroque name for double.

Lines 14 and 19 did not need iterator_traits at all, because we could see the data types of the iterators. But in line 22, the type of the iterator is written in a macro. Let's say that the definition of the macro, and the initial value of it3, were hidden in another file where we couldn't see them. What then should the data type of x3 be? The iterator_traits in line 24 answers this question very neatly.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/iterator_traits/in_main.C

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <iterator>  //for class iterator_traits
 4 #include <vector>    //includes <iterator>, so previous line not needed here
 5 using namespace std;
 6
 7 int main()
 8 {
 9     const int a[] = {10, 20, 30, 40, 50};
10     size_t n = sizeof a / sizeof a[0];
11     cout << fixed;
12
13     const int *it1 = a;
14     iterator_traits<const int *>::value_type x1 = *it1;
15     cout << x1 << "\n";   //x1 is an int (not a const int)
16
17     vector<double> v(a, a + n);
18     vector<double>::iterator it2 = v.begin();
19     iterator_traits<vector<double>::iterator>::value_type x2 = *it2;
20     cout << x2 << "\n";   //x2 is a double.
21
22 #define ITERATOR vector<double>::iterator
23     ITERATOR it3 = v.begin();
24     iterator_traits<ITERATOR>::value_type x3 = *it3;
25     cout << x3 << "\n";   //x3 is a double.
26
27     return EXIT_SUCCESS;
28 }
```

```
10          lines 13−15
10.000000   lines 17−20
10.000000   lines 22−25
```

iterator_traits in a template

In contemporary C++, an `ITERATOR` is more likely to be the template argument in the following line 7 than the macro in the above line 23. A realistic use of `iterator_trait<ITERATOR>::value_type` is in line 10. The template function `f` can now create a variable of the type retrieved by applying the `*` to whatever type of iterator was passed to it. For the `typename`, see p. 675.

Lines 13, 17, and 21 show three more data types provided by `iterator_traits`. Use the first two if you need a pointer or reference to the element to which the iterator refers. To ensure that line 15 always prints the address of the element, even if the element is a `char`, we cast the address to `const void *`.

Lines 21−27 measure the distance in elements between the elements to which the two iterators refer. The data type of `d` in line 21 is the appropriate one for holding this number, positive or negative. I wish we could compute the distance with a subtraction:

```
    cout << "The distance n elements is " << last - first << ".\n";
```

But some types of iterators permit only increment, not subtraction; an example is the `list` iterator, whose infirmities first appeared on pp. 449−450. We therefore resort to the pedestrian loop in lines 22−24.

We have now seen what `iterator_traits` gives to a template whose template argument is an iterator: four data types that will probably be needed to manipulate the iterator and the element to which it refers. The next example will have a fifth one.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/iterator_traits/in_template.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <iterator>
 4 #include <vector>
 5 using namespace std;
 6
 7 template <class ITERATOR>
 8 void f(ITERATOR first, ITERATOR last)
 9 {
10     typename iterator_traits<ITERATOR>::value_type x = *first;
11     cout << x << " is a copy of the value to which first refers.\n";
12
13     typename iterator_traits<ITERATOR>::pointer p = &*first;
14     cout << "first refers to the value " << *p << " at address "
15         << static_cast<const void *>(p) << ".\n";
16
17     typename iterator_traits<ITERATOR>::reference r = *first;
18     cout << "first refers to the value " << r << " at address "
19         << static_cast<const void *>(&r) << ".\n";
20
21     typename iterator_traits<ITERATOR>::difference_type d = 0;
22     for (; first != last; ++first) {
23         ++d;
24     }
25
26     cout << "The iterators refer to elements that are " << d
27         << " elements apart.\n\n";
28 }
29
30 int main()
31 {
32     int a[] = {10, 20, 30, 40, 50};
33     const size_t n = sizeof a / sizeof a[0];
34     cout << fixed;
35
36     int *it1 = a;
37     int *it2 = a + n - 1;
38     f(it1, it2);
39
40     vector<double> v(a, a + n);
41     vector<double>::iterator it3 = v.begin();
42     f(it3, it3 + 3);
43
44     return EXIT_SUCCESS;
45 }
```

If it is to your taste, you can create a local, one-word name, `value_type`, to stand for `typename iterator_traits<ITERATOR>::value_type` in the above line 10. That line would then become

```
46     typedef typename iterator_traits<ITERATOR>::value_type value_type;
47     value_type x = *it;
```

```
10 is a copy of the value to which first refers.
first refers to the value 10 at address 0xffbff04c.
first refers to the value 10 at address 0xffbff04c.
The iterators refer to elements that are 4 elements apart.

10.000000 is a copy of the value to which first refers.
first refers to the value 10.000000 at address 0x25878.
first refers to the value 10.000000 at address 0x25878.
The iterators refer to elements that are 3 elements apart.
```

A dispatching function

The fifth data type provided by `iterator_traits` is named `iterator_category`. The following lines 16−17 construct a variable named `category` of this type, and pass it to one of the `print` functions. The function `f` that contains these lines is merely a *call-through,* doing all its work by calling some other function. `f` is also a *dispatching function,* since the data type of the `it` in line 15 determines which `print` function is called in line 17.

`iterator_category` is always a typedef for one of five possible classes. These classes have no data members, their constructors take no arguments, and the resulting objects have no value. But the objects do have five possible data types, which lets us have the five functions with the same name in lines 7−11. A function argument whose value is unused—or nonexistent—should be declared without a name (pp. 289−290). Needless to say, these arguments are small enough to pass by value.

We will use this `iterator_category` when we talk about ‘‘iterator categories’’ on pp. 834−843.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/iterator_traits/overload.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <iterator>
 4 #include <vector>
 5 #include <list>
 6 using namespace std;
 7
 8 inline void print(input_iterator_tag)         {cout << "input\n";}
 9 inline void print(output_iterator_tag)        {cout << "output\n";}
10 inline void print(forward_iterator_tag)       {cout << "forward\n";}
11 inline void print(bidirectional_iterator_tag) {cout << "bidirectional\n";}
12 inline void print(random_access_iterator_tag) {cout << "random access\n";}
13
14 template <class ITERATOR>
15 void f(ITERATOR it)
16 {
17     typename iterator_traits<ITERATOR>::iterator_category category;
18     print(category);
19 }
20
21 int main()
22 {
23     int a[] = {10, 20, 30, 40, 50};
24     size_t n = sizeof a / sizeof a[0];
25
26     int *it1 = a;
27     f(it1);
28
29     vector<int> v(a, a + n);
30     vector<int>::iterator it2 = v.begin();
31     f(it2);
32
33     list<int> li(a, a + n);
34     list<int>::iterator it3 = li.begin();
35     f(it3);
36
37     return EXIT_SUCCESS;
38 }
```

If it is to your taste, the variable `category` in the above lines 17−18 can be an anonymous temporary. Replace these lines by the following.

```
39      print(typename iterator_traits<ITERATOR>::iterator_category());
```

```
random access      Line 27: a pointer is a random access iterator.
random access      Line 31: a vector iterator is a random access iterator.
bidirectional      Line 35: a list iterator is merely a bidirectional iterator.
```

7.3 Template Functions and Template Classes

Template functions and template classes can interact with each other in unexpected ways. Here is a quick summary of the differences between these kinds of templates. The most important ones are (2) and (4).

(1) Only template functions have template argument deduction (p. 636). The template argument of a template class must be written explicitly.

(2) Only template functions have name overloading (p. 641). Every template class must have a different name.

(3) Only template classes have default values for template arguments (p. 689).

(4) Only template classes have partial specialization (p. 702).

7.3.1 Pass a Pair of Iterators to an Algorithm

Our `print` function in line 13 of `typename.C` on p. 675 was capable of printing any type of standard library container: `vector`, `list`, etc. The elements of the container could be of any type that was printable with `<<`.

But the function would accept only standard library containers, and was hardwired to print every element. We will now define a more flexible one that can accept an array as well as a container, and that can print only some of the elements.

7.3.1.1 An Algorithm to Print Part of Container, including an Array

The `print` function in the following line 45 can print all or part of the array in line 14. The `print` in 52 can print all or part of the vector in 33. In the next example, we will combine them with a template.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/print/print1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include "date.h"
 5 using namespace std;
 6
 7 void print(const int *first, const int *last);
 8
 9 void print(vector<date>::const_iterator first,
10            vector<date>::const_iterator last);
11
12 int main()
13 {
14     int a[] = {1776, 1929, 1941, 1969, 2001};
15     const size_t n = sizeof a / sizeof a[0];
```

```

    print(a, a + n);                        //Print all the elements.
    cout << "\n";

    if (n > 2) {
        print(a + 1, a + n - 1);            //Print all but 1st and last.
        cout << "\n";
    }

    const date d[] = {
        date(date::july,       4, 1776),
        date(date::october,   29, 1929),
        date(date::december,   7, 1941),
        date(date::july,      20, 1969),
        date(date::september, 11, 2001)
    };
    const size_t dn = sizeof d / sizeof d[0];
    vector<date> v(d, d + dn);

    print(v.begin(), v.end());              //Print all the elements.
    cout << "\n";

    if (v.size() > 2) {
        print(v.begin() + 1, v.end() - 1); //Print all but 1st and last.
    }

    return EXIT_SUCCESS;
}

void print(const int *first, const int *last)
{
    for (; first != last; ++first) {
        cout << *first << "\n";
    }
}

void print(vector<date>::const_iterator first,
       vector<date>::const_iterator last)
{
    for (; first != last; ++first) {
        cout << *first << "\n";
    }
}
```

```
1776            Line 17 prints all the elements of a.
1929
1941
1969
2001

1929            Line 21 prints all but the first and last elements of a.
1941
1969

7/4/1776        Line 35 prints all the elements of v.
10/29/1929
12/7/1941
7/20/1969
9/11/2001

10/29/1929      Line 39 prints all but the first and last elements of v.
12/7/1941
7/20/1969
```

The `print` in the following line 12 prints the *range* of zero or more consecutive elements specified by the pair of iterators passed as function arguments. The C++ convention is to name these arguments `first` and `last`, and to pass them by value.

This `print` is our first official *algorithm,* a template function whose arguments are a pair of iterators that delimit a range.* The two iterators must be of the same data type, which is passed as a template argument.

If `first` and `last` have the same value, as in the following line 23, the range is empty and an algorithm processes no elements at all. Otherwise, the algorithm starts with the element to which `first` refers, and process all the elements up to *but not including* the one to which `last` refers. An algorithm never assumes that `last` refers to an element; line 15 never attempts to dereference an iterator whose value is equal to `last`. The `[)` notation in line 8 means ‘‘including `*first`, but not including `*last`’’.

The range delimited by `first` and `last` must have a finite number of elements, possibly zero. In other words, `last` must be *reachable* from `first`. This means that `first` must already be equal to `last`, or that `first` will become equal to `last` if the algorithm increments it a finite number of times. To be reachable from `first`, `last` must be equal to `first`, or must refer to a later element in the same container, or to the location after the last element in the container.

The `if` in line 39 calls the algorithm only when `last` is reachable from `first` and the range is non-empty. If the comparison were `>=`, the algorithm would still be called only when `last` is reachable from `first`, but it would now be called for an empty range.

The range passed to our `print` algorithm must be an *input range:* a series of elements whose values can be read. This means that the expression `*first` in line 15 must yield a value which can be used as an rvalue. `first` cannot be a ‘‘pointer to `void`’’ or an object whose `operator*` function returns `void`. We summarize these requirements by saying that the data type `ITERATOR` must be an ‘‘input iterator’’. For the formal definition, see pp. 834−837.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/print/print2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include <list>
```

* The function `f` in line 8 of `in_template.C` on p. 754 was an unofficial algorithm.

```
 5 #include "date.h"
 6 using namespace std;
 7
 8 //Print all the elements in the range [first, last).
 9 //ITERATOR must be an input iterator; its * must yield a puttable value.
10
11 template <class ITERATOR>
12 void print(ITERATOR first, ITERATOR last)
13 {
14     for (; first != last; ++first) {
15         cout << *first << "\n";
16     }
17 }
18
19 int main()
20 {
21     const int a[] = {10, 20, 30};
22     const size_t n = sizeof a / sizeof a[0];
23     print(a, a);                   //Print no elements (an empty range).
24
25     list<int> li(a, a + n);
26     print(li.begin(), li.end());   //Print all the elements.
27     cout << "\n";
28
29     const date da[] = {
30         date(date::july,       4, 1776),
31         date(date::october,   29, 1929),
32         date(date::december,   7, 1941),
33         date(date::july,      20, 1969),
34         date(date::september, 11, 2001)
35     };
36     const size_t dn = sizeof da / sizeof da[0];
37     vector<date> v(da, da + dn);
38
39     if (v.size() > 2) {
40         print(v.begin() + 1, v.end() - 1); //Print all but 1st and last.
41     }
42
43     return EXIT_SUCCESS;
44 }
```

```
10               Line 26 prints all the elements of li.
20
30

10/29/1929       Line 40 prints all but the first and last elements of v.
12/7/1941
7/20/1969
```

7.3.1.2 An Algorithm that uses the Traits of its Iterators

Following the success of our `print` algorithm on p. 760, we will now attempt a more ambitious one. The following function, containing code seen on pp. 47−48, sorts an array of `int`'s into ascending order.

The arguments `first` and `last` point to the first element and to the one immediately after the last to be sorted. The number of elements to be sorted is `last - first`. The smallest value will end up in `first[0]`, a.k.a. `*first`; the largest, in `last[-1]`, a.k.a. `first[last - first - 1]`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/sorter/sorter.C`

```
 1 void sorter(int *first, int *last)
 2 {
 3     while (first < --last) {
 4         for (int *p = first; p < last; ++p) {
 5             if (p[1] < p[0]) {   //if p[0] and p[1] in wrong order,
 6                 const int temp = p[0];   //swap them
 7                 p[0] = p[1];
 8                 p[1] = temp;
 9             }
10         }
11     }
12 }
```

For example,

```
13     int a[] = {1, 9, 6, 8, 5, 5, 2, 0, 0, 1}
14     const size_t n = sizeof a / sizeof a[0];
15
16     sorter(a, a + n);     //means sorter(&a[0], &a[n]); sort the entire array
17     sorter(a, a + n / 2); //means sorter(&a[0], &a[n/2]); sort only first half
```

The above function is too good to be used only for sorting `int`'s. It deserves to be turned into the following algorithm. We name it `sorter` because the C++ Standard Library already has an algorithm named `sort`, in the header file `<algorithm>`.

The `sorter` algorithm will still accept pointer arguments. But it will also accept any kind of iterator that can be copy constructed, and to which we can apply the four operators `++ -- < []` in lines 15−17. Such an iterator will be called a ‘‘random access iterator’’ on p. 841.

The data type `list<date>::iterator` in line 40 of the following `main2.C` is not a random access iterator: the operators `<` and `[]` cannot be applied to it. (It is merely a ‘‘bidirectional’’ iterator, pp. 840−841.) That’s why class `list` has been provided with the `sort` member function in line 41. See pp. 449−450.

The data type to which the iterators refer must be copy constructible and assignable. It must also be ‘‘strict weakly comparable’’, an elaboration of the less-than comparable on pp. 639−640. For the full story, see pp. 778−779.

The comparison in line 17 could have been written with ‘‘greater than’’.

```
1         if (it[0] > it[1]) {
```

But the C++ convention is to code a template so that the only inequality function the user has to define is `operator<`. Another example is on p. 778.

When sorting a container of `char *` or `const char *`, we would probably want alphabetical order. But the `<` operator applied to two `char *`’s gives us geographical order: it tells us which string is located first in memory. We’ll fix this problem with a third argument to `sorter` (pp. 764−770). In the meantime, do not pass a container of these types to the `sorter` algorithm.

Lines 18−19 construct a variable `temp` of the data type which can hold the value of the expression `it[0]`. If it is to your taste, you can create a local, one-word name, `value_type`, to stand for this type. Lines 18−19 would then become the following.

```
2                 typedef typename iterator_traits<ITERATOR>::value_type
3                     value_type;
```

```
                const value_type temp = it[0];
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/sorter/sorter.h`

```
#ifndef SORTERH
#define SORTERH
#include <iterator>   //for iterator_traits
using namespace std;

/*
ITERATOR must be a random access iterator.  The data type of the values to
which it refers, typename iterator_traits<ITERATOR>::value_type, must be copy
constructable, assignable, and strict weakly comparable.
*/

template <class ITERATOR>
void sorter(ITERATOR first, ITERATOR last)
{
    while (first < --last) {
        for (ITERATOR it = first; it < last; ++it) {
            if (it[1] < it[0]) {
                const typename iterator_traits<ITERATOR>::value_type
                    temp = it[0];
                it[0] = it[1];
                it[1] = temp;
            }
        }
    }
}
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/sorter/main2.C`

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include <list>
#include "date.h"
#include "sorter.h"
using namespace std;

template <class ITERATOR>
void print(ITERATOR first, ITERATOR last)
{
    for (; first != last; ++first) {
        cout << *first << "\n";
    }
}

int main()
{
    int a1[] = {10, 30, 20, 50, 40};
    const size_t n1 = sizeof a1 / sizeof a1[0];
    sorter(a1, a1 + n1);
22     print(a1, a1 + n1);
23     cout << "\n";
24
25     const date a2[] = {
26         date(date::october,   29, 1929),
27         date(date::july,      20, 1969),
28         date(date::july,       4, 1776),
29         date(date::september, 11, 2001),
30         date(date::december,   7, 1941)
31     };
32     const size_t n2 = sizeof a2 / sizeof a2[0];
33
34     vector<date> v(a2, a2 + n2);
35     sorter(v.begin(), v.end());
36     print(v.begin(), v.end());
37     cout << "\n";
38
39     list<date> li(a2, a2 + n2);
40     //sorter(li.begin(), li.end());   //won't compile
41     li.sort();
42     print(li.begin(), li.end());
43     return EXIT_SUCCESS;
44 }
```

```
10                  lines 19−23
20
30
40
50

7/4/1776            lines 25−37
10/29/1929
12/7/1941
7/20/1969
9/11/2001

7/4/1776            lines 39−42
10/29/1929
12/7/1941
7/20/1969
9/11/2001
```

If the above line 40 is uncommented, the `it` in line 17 pf `sorter.h` will be an object that has no `operator[]` member function.

```
main2.C:40:   instantiated from here
sorter.h:17: error: no match for 'operator[]' in 'it[1]'
```

▼ Homework 7.3.1.2a: let sorter call swap or iter_swap

Replace lines 18−21 of `sorter.h` on p. 762 with to a call to the `swap` algorithm (pp. 648−649).

```
45                  swap(it[0], it[1]);
```

`sorter.h` will now include `<algorithm>` for `swap`, not for `iterator_traits`.

Better yet, replace lines 18−21 with a call to the `iter_swap` algorithm in the standard library.

```
//Excerpt from <algorithm>

template <class ITERATOR1, class ITERATOR2>
inline void iter_swap(ITERATOR1 it1, ITERATOR2 it2)
{
    const typename iterator_traits<ITERATOR1>::value_type temp = *it1;
    *it1 = *it2;
    *it2 = temp;
}
```

Since `iter_swap` dereferences the iterators for us, the call will be

```
                iter_swap(it, it + 1);
```

▲

7.3.2 Pass a Predicate to an Algorithm

Sort into any order, not just increasing

The `sorter` algorithm on p. 762 sorts a range of values into ascending order. The `<` operator that does this is hardwired into line 17 of `sorter.h` on p. 762.

Could the order and direction of the sort somehow be passed to `sorter` as an argument?

```
    int a[] = {20, 30, 10};
    const size_t n = sizeof a / sizeof a[0];

    //Just a dream--won’t compile
    sorter(a, a + n, <);            //ascending order
    sorter(a, a + n, >);            //descending order
    sorter(a, a + n, in order of increasing absolute value);
    sorter(a, a + n, in order of how close the numbers are to 16);
    sorter(a, a + n, in order of how many distinct prime factors each number has);
```

This dream is realized in the language Perl, whose `sort` function takes a block of code in {curly braces} indicating which of two values, `$a` or `$b`, should come first. The names `$a` and `$b` are built into the language, so they need not be declared. The binary operator `<=>` yields −1, 0, or 1 depending on whether its left operand is numerically less than, equal to, or greater than the right operand.

`http://i5.nyu.edu/~mm64/book/src/sorter/sort.pl`

```
#!/bin/perl

@v = (20, 30, 10);   #Create a list named @v containing 3 integers.

@v = sort {$a <=> $b} @v;   #ascending numeric order
print "@v\n";

@v = sort {$b <=> $a} @v;   #descending numeric order

@v = sort {abs($a - 16) <=> abs($b - 16)} @v;   #how close to 16

exit 0;
```

```
10 20 30       result of sort {$a <=> $b}
```

The language Ruby has a similar block of code. This time, the names `a` and `b` are not built into the language. They have to be declared by being surrounded by vertical bars.

`http://i5.nyu.edu/~mm64/book/src/sorter/sort.rb`

```
#!/opt/sfw/bin/ruby

v = [20, 30, 10]
v = v.sort {|a, b| a <=> b}   #ascending numeric order
puts v
exit 0
```

Now back to reality. The bare `<` operator in the above C++ line 5 will not compile. Now how could the `<` be passed to the `sorter` function? What would be the minimal argument that could carry the `<` and nothing else?

A binary predicate

To design this argument, let's pose a question about syntax. What could the `x` be in the following statement? What kind of expression `x` would accept an argument list of two `int`'s and give us back a `bool` or a value convertible thereto?

```
1       bool b = x(10, 20);
```

There are three possibilities. The `x` could be

(1) a function, like the `f` in the following line 17;

(2) a pointer to a function, like the `p` in line 22; or

(3) an object with an `operator()` member function, like the `g` in line 26.

The `p` declared in line 20 is a pointer to a function. The expression `(*p)(i, j)` in line 21 calls the function to which `p` points. The expression first applies the dereferencing operator `*` to `p`. Then it applies the function call operator `()` to the expression `*p`. The parentheses around the `*p` are not an operator. They merely force the dereferencing operator to be applied before the function call operator. See p. 248.

$$\boxed{(\boxed{*\ \boxed{p}})(\ \boxed{i}\ ,\ \boxed{j}\ )}$$

The `*` in line 21 is optional; line 22 does the same thing without it. And now that the `*` is gone, we can dispense with the surrounding parentheses.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/sorter/apply.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class function_object {
 6 public:
 7     bool operator()(int a, int b) const {return a < b;}
 8 };
 9
10 inline bool f(int a, int b) {return a < b;}
11
12 int main()
```

```
13 {
14     int i = 10;
15     int j = 20;
16
17     bool b = f(i, j);           //f is a function.
18     cout << boolalpha << b << "\n";
19
20     bool (*p)(int, int) = f;   //p is a pointer to a function.
21     b = (*p)(i, j);             //Lines 21-22 are two ways to call the ...
22     b = p(i, j);                //... function to which p points.
23     cout << b << "\n";
24
25     function_object g;
26     b = g(i, j);                //b = g.operator()(i, j);
27     cout << b << "\n";
28
29     return EXIT_SUCCESS;
30 }
```

```
true      lines 17−18
true      lines 19−23
true      lines 25−27
```

An expression that can take one or more arguments and give us back a value of type `bool` (or convertible thereto) is called a *predicate.* The examples `f`, `p`, and `g` are *binary predicates* because they take two arguments. There are also *unary predicates,* taking one argument.

Pass a predicate to an algorithm

The order and direction of a sort can be passed to an algorithm as a binary predicate. Our example is the `comp` in the following line 17.

Like an iterator, a predicate is always passed by value. `comp` must be a binary predicate because line 21 applies two arguments to it and gives the result to the keyword `if`. `comp` could be a pointer to a function, such as the `f` passed to `sorter` in line 36 of `main3.C` on p. 768.* Or the predicate could be a function object, such as the `gi` constructed in line 39 of `main3.C` and passed to `sorter` in line 40.

The class of `gi` (`greater_int`, lines 12−15 of `main3.C`) is merely stereotyped boilerplate that holds the > comparison. Similarly, class `printable` on p. 735 was merely a holder for the `isprint` error checking. In each case, the rest of the class is just connective tissue. The comparison and the error checking are packaged as classes because a class is the smallest chunk of syntax that can be passed as a template argument.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/sorter/sorter2.h`

```
 1 #ifndef SORTERH
 2 #define SORTERH
 3 #include <iterator>   //for iterator_traits
 4 using namespace std;
 5
 6 /*
 7 ITERATOR must be random access.
 8
```

* This `f` plays the same rôle as the pointer to a function that will be passed to the C Standard Library `qsort` in line 12 of `qsort.c` on p. 775.

```
 9 typename iterator_traits<ITERATOR>::value_type must be copy constructable
10 and assignable.
11
12 COMPARE must be a binary predicate accepting arguments of type
13 typename iterator_traits<ITERATOR>::value_type.
14 */
15
16 template <class ITERATOR, class COMPARE>
17 void sorter(ITERATOR first, ITERATOR last, COMPARE comp)
18 {
19     while (first < --last) {
20         for (ITERATOR it = first; it < last; ++it) {
21             if (comp(it[1], it[0])) {
22                 const typename iterator_traits<ITERATOR>::value_type
23                     temp = it[0];
24                 it[0] = it[1];
25                 it[1] = temp;
26             }
27         }
28     }
29 }
30 #endif
```

If the `comp` in the above line 21 is an object, then the line behaves as if we had said

```
31     if (comp.operator()(it[1], it[0])) {
```

If the `comp` is a pointer to a function, it behaves as if we had said

```
32     if ((*comp)(it[1], it[0])) {
```

As on p. 764, the above lines 22−25 may be replaced by a call to the `iter_swap` algorithm.

When the following line 40 passes `gi` to `sorter`, the above line 21 will call the `operator()` member function of `gi`. Since `gi` is used only once, in line 40 below, we could have made it an anonymous temporary like the one in 43. It has no constructor, but 43 can pretend that it has a constructor with no arguments.

The following line 56 constructs an anonymous temporary of class `greater_date`. The above line 21 will call its `operator()` member function, which compares two `date`'s. To allow this member function to apply the `>` operator to a pair of `date`'s (line 19), we must define an `operator>` function for class `date`.

Why are we fooling around with temporary objects when the non-member function `f` in line 36 of `main3.C` does the job more simply? We will see two reasons: an object can have data members (pp. 770−772), and a member function can be called faster than a non-member function (pp. 772−776).

The member function `operator()` in line 14 of `main3.C` uses no members of the object to which it belongs. We would therefore like to make it a static member function for extra speed. But the syntax of the language says that `operator()` must always be non-static; see p. 287. After all, what would the syntax look like for a call to a static `operator()`?

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/sorter/main3.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include <list>
 5
 6 #include "date.h"
```

```
#include "sorter2.h"
using namespace std;

inline bool f(int a, int b) {return a > b;}

class greater_int {
public:
        bool operator()(int a, int b) const {return a > b;}
};

class greater_date {
public:
        bool operator()(const date& a, const date& b) const {return a > b;}
};

template <class ITERATOR>
void print(ITERATOR first, ITERATOR last, char c)
{
    for (; first != last; ++first) {
        cout << *first << c;
    }
    cout << "\n";
}

int main()
{
    int a1[] = {10, 30, 20, 50, 40};
    const size_t n1 = sizeof a1 / sizeof a1[0];

    sorter(a1, a1 + n1, f);
    print(a1, a1 + n1, ' ');

    greater_int gi;
    sorter(a1, a1 + n1, gi);
    print(a1, a1 + n1, ' ');

    sorter(a1, a1 + n1, greater_int()); //construct an anonymous object
    print(a1, a1 + n1, ' ');

    const date a2[] = {
        date(date::october,    29, 1929),
        date(date::july,       20, 1969),
        date(date::july,        4, 1776),
        date(date::september,  11, 2001),
        date(date::december,    7, 1941)
    };
    const size_t n2 = sizeof a2 / sizeof a2[0];
    vector<date> v(a2, a2 + n2);

    sorter(v.begin(), v.end(), greater_date());
    print(v.begin(), v.end(), '\n');

    return EXIT_SUCCESS;
}
```

```
50 40 30 20 10       lines 36−37
50 40 30 20 10       lines 39−41
50 40 30 20 10       lines 43−44
9/11/2001            lines 56−57
7/20/1969
12/7/1941
10/29/1929
7/4/1776
```

Class greater in the C++ Standard Library

Classes `greater_int` and `greater_date`, in lines 12 and 17 of the above `main3.C`, are almost identical except for the data types of `a` and `b`. They have therefore been written once and for all as the following template class in the standard library. A C++ `struct` is the same as a class, except that its members are public by default. In particular, a C++ `struct` can have member functions.

```
 1 //Provisional excerpt from the header file <functional>
 2 //Version 1 of struct greater.  T must be greater-than comparable.
 3
 4 template <class T>
 5 struct greater {
 6     bool operator()(const T& a, const T& b) const {return a > b;}
 7 };
```

To use the standard library `greater` in the above `main3.C`, include the header file `<functional>`. Change `greater_int` to `greater<int>` in lines 39 and 43 and remove class `greater_int`; change `greater_date` to `greater<date>` in line 56 and remove class `greater_date`. The output should remain unchanged.

The standard library `greater` library actually has three additional members not shown above. These are the typedefs in the following lines 13−15, providing information about the member function in line 17. Although the first argument in line 17 is a `const T&`, the typedef in line 13 is an unadorned `T`. This is because the intent of the typedef is to show the data type of the value passed to or from the `operator()`, not the mechanism by which the value is passed. The reason for this will appear when the typedefs are used on p. 863.

```
 8 //Provisional excerpt from <functional>
 9 //Version 2 of struct greater.  T must be greater-than comparable.
10
11 template <class T>
12 struct greater {
13     typedef T first_argument_type;
14     typedef T second_argument_type;
15     typedef bool result_type;
16
17     bool operator()(const T& a, const T& b) const {return a > b;}
18 };
```

The same three typedefs are present in many similar classes, including the following six relational classes.

```
    equal                   less                    greater
    not_equal_to            greater_equal           less_equal
```

For convenience, the typedefs are defined once and for all in the base class `binary_function`.

```
19 //Excerpt from <functional>
```

```
20
21 template <class T1, class T2, class T3>
22 struct binary_function {
23     typedef T1 first_argument_type;
24     typedef T2 second_argument_type;
25     typedef T3 result_type;
26 };
```

For example, class `binary_function<int, int, bool>` has the public member

```
27     typedef int first_argument_type;
```

and class `binary_function<double, double, bool>` has the public member

```
28     typedef double first_argument_type;
```

Class `greater<int>` was then derived from class `binary_function<int, int, bool>`. As above, the first function argument in line 35 is a `const T&`, but the first template argument in line 34 is an unadorned `T`.

```
29 //Excerpt from <functional>
30 //Version 3 (the final one) of struct greater.
31 T must be greater-than comparable.
32
33 template <class T>
34 struct greater: public binary_function<T, T, bool> {
35     bool operator()(const T& a, const T& b) const {return a > b;}
36 };
```

For another base class that contains nothing but typedefs to be inherited by derived classes, see class `iterator` on pp. 813−815. For another class similar to `greater`, see class `multiplies` in line 57 on p. 810. This time, the `result_type` member and the `operator()` return type are both `T`. The product must be returned by value, no matter what `T` is, since the anonymous temporary that holds `a * b` is automatically allocated.

A predicate containing a data member

On p. 767 we asked why a predicate should be written as a function object rather than as a function. Consider the predicates in the following lines 11 and 13, will tell which argument is closer to the number 1955. The function in line 11 is simpler, but the number is hardwired in. The class in line 13 is more flexible: the number is passed to a constructor and stored in a data member. Although the class is more verbose than the function, the extra code is generic boilerplate.

Our predicate in this example has a `const` data member, but a non-`const` is also possible.

We can do even better. Instead of hardwiring the data type into lines 14 and 21, we can write it as the template argument `T` in line 32. Note that the predicate, including the `T` inside it, is still passed to the algorithm by value. It is unimportant if we make this one copy of the `T`. Overwhelmingly more important is that any `T` argument of the predicate's `operator()` be passed by reference, since the algorithm will probably call `operator()` many times.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/sorter/main4.C`

```
 1 #include <iostream>
 2 #include <cstdlib>  //for the abs that takes and returns an int
 3 #include <vector>
 4 #include <list>
 5 #include <functional>
 6
 7 #include "date.h"
```

```
#include "sorter2.h"
using namespace std;

inline bool f(int a, int b) {return abs(a - 1955) < abs(b - 1955);}

class closer_int {
    const int i;
public:
    closer_int(int initial_i): i(initial_i) {}
    bool operator()(int a, int b) const {return abs(a - i) < abs(b - i);}
};

class closer_date {
    const date d;
public:
    closer_date(const date& initial_d): d(initial_d) {}

    bool operator()(const date& a, const date& b) const {
        return abs(a - d) < abs(b - d);
    }
};

template <class T>   //T must be copy constructable and have binary -
class closer {
    const T t;
public:
    closer(const T& initial_t): t(initial_t) {}

    bool operator()(const T& a, const T& b) const {
        return abs(a - t) < abs(b - t);
    }
};

template <class ITERATOR>
void print(ITERATOR first, ITERATOR last, char c)
{
    for (; first != last; ++first) {
        cout << *first << c;
    }
    cout << "\n";
}

int main()
{
    int a1[] = {1929, 1969, 1776, 2001, 1941};
    const size_t n1 = sizeof a1 / sizeof a1[0];

    sorter(a1, a1 + n1, f);
    print(a1, a1 + n1, ’ ’);

    closer_int ci(1955);
    sorter(a1, a1 + n1, ci);
    print(a1, a1 + n1, ’ ’);

62     sorter(a1, a1 + n1, closer_int(1955)); //construct an anonymous object
63     print(a1, a1 + n1, ' ');
64
65     sorter(a1, a1 + n1, closer<int>(1955));
66     print(a1, a1 + n1, ' ');
67
68     const date a2[] = {
69         date(date::october,    29, 1929),
70         date(date::july,       20, 1969),
71         date(date::july,        4, 1776),
72         date(date::september,  11, 2001),
73         date(date::december,    7, 1941)
74     };
75     const size_t n2 = sizeof a2 / sizeof a2[0];
76     vector<date> v(a2, a2 + n2);
77
78     sorter(v.begin(), v.end(), closer_date(date(date::july, 12, 1955)));
79     print(v.begin(), v.end(), '\n');
80
81     sorter(v.begin(), v.end(), closer<date>(date(date::july, 12, 1955)));
82     print(v.begin(), v.end(), '\n');
83
84     return EXIT_SUCCESS;
85 }
```

```
1969 1941 1929 2001 1776   lines 55−56: sorter calls f in line 11
1969 1941 1929 2001 1776   lines 58−60: sorter calls operator() of closer_int object
1969 1941 1929 2001 1776   lines 62−63: sorter calls operator() of closer_int object
1969 1941 1929 2001 1776   lines 65−66: sorter calls operator() of closer<int> object
12/7/1941                  lines 78−79: sorter calls operator() of closer_date object
7/20/1969
10/29/1929
9/11/2001
7/4/1776

12/7/1941                  lines 81−82: sorter calls operator() of closer<date> object
7/20/1969
10/29/1929
9/11/2001
7/4/1776
```

Three ways of calling a function

Another reason to write a predicate as a function object rather than as a function is to make the algorithm run faster. Let's review the three ways of calling a function in C++.

(1) the normal way, in the following line 11;

(2) the faster way, making the call inline in line 12;

(3) the slower way, calling the function via the pointer `p` in lines 15 and 16.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/sort/three.C`

```
1 #include <iostream>
```

```
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 void normal();
 6
 7 inline void faster() {cout << "faster\n";}
 8
 9 int main()
10 {
11     normal();               //call line 21
12     faster();               //call line 7
13
14     void (*p)() = normal;   //p is a pointer to a function
15     (*p)();                 //call line 21, slower than normal
16     p();                    //simpler way to write line 15
17
18     return EXIT_SUCCESS;
19 }
20
21 void normal()
22 {
23     cout << "normal\n";
24 }
```

```
normal      line 11
faster      line 12
normal      line 15
normal      line 16
```

Mathematicians have proved that when sorting *n* items, we must perform $n \log_2 n$ comparisons in the worst case.* To sort a million items we might have to perform almost 20 million comparisons, and `sorter` will call its predicate almost 20 million times. It is to be hoped that each of these calls will be as fast as possible.

n	$n \log_2 n$
10	10×3.32
100	100×6.64
1,000	$1{,}000 \times 9.97$
10,000	$10{,}000 \times 13.3$
100,000	$100{,}000 \times 16.6$
1,000,000	$1{,}000{,}000 \times 19.9$

When `sorter` is called in line 55 of `main4.C` on p. 771, the `comp` in line 17 of `sorter2.h` on p. 767 is a pointer to the function `f`. `sorter` is forced to call it in the slowest possible way, unable to take advantage of the fact that `f` is inline. But when `sorter` is called in line 65 of `main4.C`, the `comp` is an object of class `closer<int>`. Although the object was passed to `sorter` as an argument, the call to its member function `operator()` can still be inline.

* $\log_2 n$ is the power to which 2 must be raised to produce the desired number *n*. For example, $\log_2 16 = 4$ and $\log_2 32 = 5$.

The C++ sort algorithm vs. the C qsort function

The C++ Standard Library has two `sort` algorithms, both of them much faster than our `sorter`. The following line 22 calls the two-argument version, which, like our two-argument `sorter` in line 13 of `sorter.h` on p. 762, is hardwired to apply the operator `<` to each pair of values that it compares. Line 36 calls the three-argument version, which lets us supply a predicate. The predicate that you want can often be found in the standard library.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/sort/sort.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include <algorithm>   //for 2- and 3-argument sort
 5 #include <functional>  //for greater
 6 #include "date.h"
 7 using namespace std;
 8
 9 template <class ITERATOR>
10 void print(ITERATOR first, ITERATOR last)
11 {
12     for (; first != last; ++first) {
13         cout << *first << "\n";
14     }
15 }
16
17 int main()
18 {
19     int a[] = {1969, 2001, 1941, 1929, 1776};
20     const size_t n = sizeof a / sizeof a[0];
21
22     sort(a, a + n);
23     print(a, a + n);
24     cout << "\n";
25
26     const date da[] = {
27         date(date::july,      20, 1969),
28         date(date::september, 11, 2001),
29         date(date::december,   7, 1941),
30         date(date::october,   29, 1929),
31         date(date::july,       4, 1776)
32     };
33     const size_t dn = sizeof da / sizeof da[0];
34     vector<date> v(da, da + dn);
35
36     sort(v.begin(), v.end(), greater<date>());
37     print(v.begin(), v.end());
38     return EXIT_SUCCESS;
39 }
```

```
1776          lines 22−24
1929
1941
1969
2001

9/11/2001     lines 36−37
7/20/1969
12/7/1941
10/29/1929
7/4/1776
```

qsort in the C Standard Library

Let’s glance back at the C Standard Library function `qsort`, called in the following line 12. It must always be passed the address of a comparison function (lines 21−33), causing `qsort` to call this function in the slowest possible way.

`qsort` is also more dangerous. The arguments in line 21 have to be `const void *` because `qsort` is declared as

```
void qsort(void *base, size_t n, size_t width,
    int (*p)(const void *, const void *));
```

The conversions in 23 and 24 have no way to check that `p1` and `p2` actually point to integers. They might point to anything.

In C++, the two-argument `sort` algorithm performs its comparisons with the `<` operator. For objects, the `<` calls an `operator<` function which we can make inline. For the built-in types and pointers, the `<` is built into the language and calls no comparison function at all. For enumerations, we could write an `operator<` (inline, of course) but probably don’t need to. The predicate passed to the three-argument `sort` algorithm can be a function object whose `operator()` is inline.

The out-of-control conversions are not needed by the `sort` algorithm. When sorting a range of elements of type `T`, we provide an `operator<` or a predicate whose arguments are of type `T`. There are no conversions at all.

Of course, this type safety comes at a price. We call a different instantiation of the two-argument `sort` for each type of `ITERATOR` that we pass to it. We call a different instantiation of the three-argument `sort` for each combination of `ITERATOR` and `COMPARE` that we pass to it. Furthermore, the predicate for the three-argument `sort` will probably be an object of a template class, or a pointer to a template function. A separate instantiation of the predicate will be created for each type of `T` we pass to it. All of these instantiations will make the executable program larger. But the extra speed and safety are worth it.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/sort/qsort.c`

```
 1 #include <stdio.h>
 2 #include <stdlib.h>   /* for qsort */
 3
 4 int comp(const void *p1, const void *p2);
 5
 6 int main()
 7 {
 8     int a[] = {1, 3, 0, 2, 5};
 9     const size_t n = sizeof a / sizeof a[0];
10     const int *p;
11
12     qsort(a, n, sizeof a[0], comp);   /* last arg is pointer to function */
13
14     for (p = a; p < a + n; ++p) {
15         printf("%d\n", *p);
16     }
17
18     return EXIT_SUCCESS;
19 }
20
21 int comp(const void *p1, const void *p2)  /* sort in increasing numeric order */
22 {
23     const int i = *(const int *)p1;   /* can’t dereference void * */
24     const int j = *(const int *)p2;
25
26     if (i < j) {
27         return -1;
28     }
29
30     if (i > j) {
31         return 1;
32     }
33
34     return 0;
35 }
```

The above lines 26−34 may be combined to the following, with some loss of clarity.

```
36     return i < j ? -1 : i > j;
```

7.3.3 Concepts and Models

Our `sorter` algorithm has a bug. The bug is not in the code, last seen in `sorter2.h` on pp. 766−767. The bug is something missing from the comment. But this is not merely a sermon on the importance of comments. The output of `sorter` could be wrong.

To get `sorter` to work, and indeed to get our original `min` template on pp. 637−638 to work, we will have to give a better definition of the concept of ‘‘less-than comparable’’. For `sorter`, we will also have to define the concept of ‘‘strict weakly comparable’’.

Less-than comparable

For a data type to qualify as less-than comparable, the `<` operator must yield a result of type `bool` or convertible thereto. If it does not, the following lines 3 and 7 will not compile.

```
 1     //Excerpt from the min template function.
 2     //The left operand of ?: must be bool or convertible thereto.
 3     return b < a ? b : a;
 4 }

 5     //Excerpt from sorter algorithm.
 6     //The expression in parentheses must be bool or convertible thereto.
 7     if (p[1] < p[0]) {
```

The `<` must also have the following two properties.

(1) If the left and right operands are references to the same value, either in the same variable

```
a < a
```

or in two equal expressions

```
six < half_dozen
```

the inequality must be false. This property is called *irreflexivity.*

(2) If both of the following are true,

```
a < b
b < c
```

then it must also be true that

```
a < c
```

This property is called *transitivity.* For an unfortunate `operator<` that did not have these properties, see p. 442.

Application of less-than comparability to min

What should `min` return if its two arguments `a` and `b` are equal?

```
a == b
```

`min` is under no obligation to check for equality. In fact, `min` is not even allowed to check for it. An expression such as `a == b` in `min` would be within its rights if it failed to compile. `min` is obliged to compile for data types that are less-than comparable. But it is under no obligation to compile for data types that are *equality comparable,* unless they happen also to be less-than comparable.

What should `min` return if both of the following are true?

```
a < b
b < a
```

`min` is under no obligation to work correctly in this case, or even to check for it. If both of the above inequalities were true, `T` would not be less-than comparable. The proof is simple. If both inequalities were true, and if `<` were transitive, then it would also be true that

```
a < a
```

But this would mean that `<` is not irreflexive, and hence that `T` is not less-than comparable. `min` is required to work only when `T` is less-than comparable. This simplifies the design of `min`.

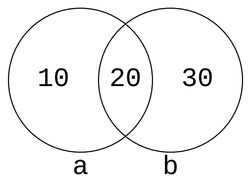
What should `min` return if both of the following are *false?*

```
a < b
b < a
```

This is something that `min` does have to handle. It certainly happens when `a` and `b` have the same value, for example when they are equal integers. Let's also look at an example where it happens when `a` and `b` have different values.

The standard library has an `operator<` that compares two `set<int>` objects, `a` and `b`, returning true if `a` comes before `b` in "lexicographic order"; see p. 952. Let's imagine another `operator<` that would return true if `a` is a *proper subset* of `b`. This means that every element of `a` is an element of `b`, but at least one element of `b` is not an element of `a`. (We will write this function on p. 861.)

The following `a` and `b` make both inequalities false, even though they are different values. `a < b` is false, because `a` contains 10 but `b` does not. `b < a` is false, because `b` contains 30 but `a` does not.



If both inequalities are false, the standard library `min` function returns a reference to its first argument. For compatibility, our `min` behaves the same way: it returns a reference to its first argument when neither argument is less than the other. That's why we had to write the body as

```
    return b < a ? b : a;  //Return a if neither one is less than the other.
```

rather than

```
    return a < b ? a : b;  //Return b if neither one is less than the other.
```

We could also have written the body as follows.

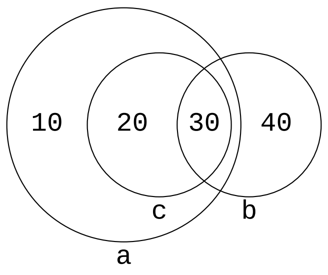
```
    return !(a > b) ? a : b;
```

It's maladroit but it does return the correct answer. The convention in C++, however, is to code a template so that `<` is the only inequality applied to `T`. A previous example was on p. 761.

Strict weakly comparable

The `T` passed to `sorter` must be copy constructible, assignable, and less-than comparable. Is there any other concept of which `T` must be a model?

Consider once again class `set<int>`, equipped with an `operator<` that checks for proper subset. We'll make three of these objects, `a`, `b`, and `c`, and store them in an array.



```
#include <set>      //for the template class set
#include "sorter.h" //for our sorter algorithm
using namespace std;

    set<int> arr[] = {a, b, c};
    const size_t n = sizeof arr / sizeof arr[0];
    sorter(arr, arr + n);
```

`b < a` is false, because `b` contains 40 but `a` does not. `sorter` will therefore leave `arr[0]` and `arr[1]` unmoved. `c < b` is also false, because `c` contains 20 but `b` does not. `sorter` will leave `arr[1]` and `arr[2]` unmoved. But it is wrong for all three elements to be left unmoved. `c` should have been moved in front of `a`, because `c < a`.

The `sorter` algorithm failed because there is one more concept of which `T` must be a model. `T` must be *strict weakly comparable,* a more demanding concept than mere less-than comparability. Strict weak comparability is defined in terms of *equivalence.* We say that two values are equivalent if neither one is less than the other.* In the above example, `a` and `b` were equivalent, and `b` and `c` were equivalent.

For a data type to be strict weakly comparable, equivalence must be transitive. In other words, if `a` and `b` are equivalent, and `b` and `c` are equivalent, then `a` and `c` must also be equivalent. Our `set<int>` data type, with the `operator<` that checks for proper subset-hood, was not a model of this concept. `a` and `b` were equivalent, and `b` and `c` were equivalent, but `a` and `c` were not equivalent.

* The author finds it helpful to paraphrase ‘‘neither `a` nor `b` is less than the other’’ as ‘‘`a` and `b` are about the same size’’.

The `sorter` algorithm will work only when `T` is copy constructible, assignable, and strict weakly comparable. Mere less-than comparability was adequate for `min`, but not for `sorter`.

7.3.4 Call a Function by Instantiating a Class

In §7.3.4, template classes will come to the aid of template functions. In §7.3.5, template functions will reciprocate by coming to the aid of template classes.

A template class can have partial and explicit specializations. Somewhat arbitrarily, a template function can have only explicit specializations. Furthermore, we saw two problems with the latter:

(1) We cannot have a template function that takes `T` by reference, with an explicit specialization that takes a specific type by value. See p. 664.

(2) An explicit specialization is a specialization of one specific template function, and must be written below it. See pp. 667−668.

Both of these limitations are avoided by the `print` function in the following line 55. It is merely a dispatching function (p. 756), doing its work by calling some other function selected by its template argument `T`. This other function will be the `print` static member function of some class `_print<T>`, selected from the partial and explicit specializations of the template class `_print`. With this simple technique, partial specialization can be extended to template functions.

The `print` member function takes `T` by reference in the primary template in line 12, and takes `char` by value in the explicit specialization in line 17. This is the combination we were unable to achieve on p. 664. The `char *` in line 22 no longer has to be below the `const T *` in line 27; this is a freedom we did not have on pp. 667−668.

To mention the non-member function `print` in lines 31 and 47, we must first declare it in line 8. To make the name `print` in 31 and 47 refer to this non-member function, we must adorn it with the unary scope operator `::`. Without this operator, line 31 would try to call another instantiation of line 27.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/specialize_class/print.h`

```
 1 #ifndef PRINTH
 2 #define PRINTH
 3 #include <iostream>
 4 #include <vector>
 5 using namespace std;
 6
 7 template <class T>
 8 inline void print(const T& t);   //declaration for non-member print in line 55
 9
10 template <class T>
11 struct _print {
12     static void print(const T& t) {cout << t;}
13 };
14
15 template <>
16 struct _print<char> {
17     static void print(char c) {cout << "’" << c << "’";}
18 };
19
20 template <>
21 struct _print<const char *> {
22     static void print(const char *p) {cout << "\"" << p << "\"";}
23 };
24
25 template <class T>
```

```
26 struct _print<const T *> {
27     static void print(const T *p) {
28         cout << p;
29         if (p != 0) {
30             cout << " -> ";
31             ::print(*p);   //call the non-member print (line 55)
32         }
33     }
34 };
35
36 template <class T>
37 struct _print<vector<T> > {
38     static void print(const vector<T>& v) {
39         cout << "(";
40
41         for (typename vector<T>::const_iterator it = v.begin();
42             it != v.end(); ++it) {
43
44             if (it != v.begin()) {
45                 cout << ", ";
46             }
47             ::print(*it);   //call the non-member print (line 55)
48         }
49
50         cout << ")";
51     }
52 };
53
54 template <class T>
55 inline void print(const T& t) {_print<T>::print(t);}   //dispatching function
56 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/specialize_class/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include "print.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     int i = 10;
10     print(i);
11     cout << "\n";
12
13     print('A');
14     cout << "\n";
15
16     const int *p = &i;
17     print(p);
18     cout << "\n";
19
20     const char *a[] = {"moe", "larry", "curly"};
21     const size_t n = sizeof a / sizeof a[0];
22     vector<const char *> v(a, a + n);
23     print(v);
24     cout << "\n";
25
26     vector<vector<int> > vvi(2);   //2nd func arg defaults to vector<int>()
27     vvi[0].push_back(10);
28     vvi[0].push_back(20);
29     vvi[1].push_back(30);
30     vvi[1].push_back(40);
31     vvi[1].push_back(50);
32     print(vvi);
33     cout << "\n";
34
35     return EXIT_SUCCESS;
36 }
```

```
10                           L. 10 of main.C calls l. 59 of print.h, which calls l. 12 of print.h.
0xffbff158 -> 10             L. 17 of main.C calls l. 59 of print.h, which calls l. 27 of print.h.
("moe", "larry", "curly")    L. 23 of main.C calls l. 59 of print.h, which calls l. 38 of print.h.
((10, 20), (30, 40, 50))     L. 32 of main.C calls l. 59 of print.h, which calls l. 38 of print.h.
```

7.3.5 Construct an Object by Calling a Helper Function

An explicit template argument is usually unnecessary when instantiating a template function. The following line 10 instantiates `print<double>` and calls it; line 12 instantiates `print<char>` without calling it.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/helper/function.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 template <class T>
 6 inline void print(const T& t) {cout << t << "\n";}
 7
 8 int main()
 9 {
10     print(3.14);
11
12     void (*p)(const char&) = print;  //p is a pointer to function
13     p('A');                          //(*p)('A'); would do the same thing
14
15     return EXIT_SUCCESS;
16 }
```

```
3.14
A
```

But an explicit template argument is always necessary when instantiating a template class. The following line 18 instantiates `wrapper<int>` without constructing any object thereof; line 20 instantiates `wrapper<double>` and constructs a named object thereof; line 23 instantiates `wrapper<char>` and constructs an anonymous object thereof.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/helper/class.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 template <class T>
 6 class wrapper {
 7     const T t;
 8 public:
 9     wrapper(const T& initial_t): t(initial_t) {}
10     void print() const {cout << t << "\n";}
11 };
12
13 template <class T>
14 inline void f(const wrapper<T>& w) {w.print();}
15
16 int main()
17 {
18     cout << sizeof (wrapper<int>) << "\n";
19
20     wrapper<double> w(3.14);
21     f(w);
22
23     f(wrapper<char>('A'));
24
25     return EXIT_SUCCESS;
26 }
```

```
4            size may be different on other machines
3.14
A
```

Even though the `3.14` in the above line 20 is obviously a `double`, we had to write the explicit template argument `<double>` anyway. I wish we didn't have to.

We can avoid the `<double>`, at least if the object is anonymous. The `make_wrapper` in the following line 15 is called a *helper function*. It constructs and returns an anonymous object of the class indicated by its function argument `t`. The call to `make_wrapper(3.14)` in line 12 of `main.C` constructs and returns a `wrapper<double>`; the `make_wrapper('A')` in line 13 constructs and returns a `wrapper<char>`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/helper/wrapper.h`

```
1 #ifndef WRAPPERH
2 #define WRAPPERH
3 #include <iostream>
4 using namespace std;
5
6 template <class T>
7 class wrapper {     //primary template
8     const T t;
9 public:
    wrapper(const T& initial_t): t(initial_t) {}
    void print() const {cout << t;}
};

template <class T>
inline wrapper<T> make_wrapper(const T& t) {return wrapper<T>(t);}
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/helper/main.C`

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include "wrapper.h"
using namespace std;

template <class T>
inline void f(const wrapper<T>& w) {w.print(); cout << "\n";}

int main()
{
    f(make_wrapper(3.14));
    f(make_wrapper('A'));

    return EXIT_SUCCESS;
}
```

```
3.14
A
```

There are no helper functions for constructing a `vector` or `list`. Helper functions are provided only for classes whose objects are constructed anonymously, passed to functions, and never seen again. Chapter 8 will present the six groups of helper functions in the C++ Standard Library. The first group is readily accessible; the last four are extremely abstract.

(1) The function `make_pair` constructs a `pair` object (pp. 786−787).

(2) The functions `inserter`, `front_inserter`, and `back_inserter` construct an `insert_iterator`, `front_insert_iterator`, and `back_insert_iterator` respectively (pp. 848−849).

(3) The functions `bind1st` and `bind2nd` construct a `binder1st` and `binder2nd` respectively (pp. 861−864).

(4) The functions `ptr_fun`, `mem_fun_ref`, and `mem_fun` construct a `pointer_to_unary_function` or `pointer_to_binary_function`, `mem_fun_ref_t` or `mem_fun1_ref_t`, and `mem_fun_t` or `mem_fun1_t` respectively (pp. 869−875).

(5) The functions `not1` and `not2` construct a `unary_negate` and `binary_negate` respectively (pp. 876 and 942−943).

(6) The functions `compose1` and `compose2` construct a `unary_compose` and `binary_compose` respectively (pp. 871 and 864−867). These are extensions to the library, not part of the standard.