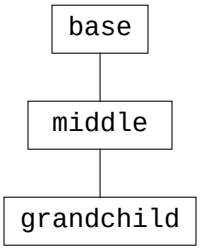
10

Miscellaneous

The topics in this chapter—RTTI, namespaces, and internationalization—are united only in having little interaction with the rest of the book.

10.1 Runtime Type Identification (RTTI)

Inheritance and virtual functions allow us to interact with an object without knowing what class it belongs to. Sometimes, however, there is a legitimate need to interrogate an object and discover its class. Runtime Type Identification lets a program do this, while the program is running. (In Java, this is called *reflection.*) Our example will be a family of three classes.



Each derived class will need a bigger and better `operator<<`. We'd like to make it a virtual function; but only a member, not a friend, can be virtual. The solution, as on pp. 496−497, is to have the friend delegate its work to a member function (line 15) and have the member function be virtual (line 9).

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

```
 1 #ifndef BASEH
 2 #define BASEH
 3 #include <iostream>
 4 using namespace std;
 5
 6 class base {
 7     int i;
 8 protected:
 9     virtual void print(ostream& os) const {os << i;}
10 public:
11     base(int initial_i): i(initial_i) {}
12     virtual ~base() {}
13
14     friend ostream& operator<<(ostream& ost, const base& b) {
15         b.print(ost);
```

```
16          return ost;
17      }
18  };
19  #endif
```

Class `middle` introduces a new member function in line 13 that wasn't in the `base` class.

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

```
 1  #ifndef MIDDLEH
 2  #define MIDDLEH
 3  #include <iostream>
 4  #include "base.h"
 5  using namespace std;
 6
 7  class middle: public base {
 8      int j;
 9  protected:
10      void print(ostream& os) const {base::print(os); os << ", " << j;}
11  public:
12      middle(int initial_i, int initial_j): base(initial_i), j(initial_j) {}
13      void newfunc() const {cout << "newfunc";}
14  };
15  #endif
```

Class `grandchild` inherits the `newfunc`.

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

```
 1  #ifndef GRANDCHILDH
 2  #define GRANDCHILDH
 3  #include <iostream>
 4  #include "middle.h"
 5  using namespace std;
 6
 7  class grandchild: public middle {
 8      int k;
 9      void print(ostream& os) const {middle::print(os); os << ", " << k;}
10  public:
11      grandchild(int initial_i, int initial_j, int initial_k)
12          : middle(initial_i, initial_j), k(initial_k) {}
13  };
14  #endif
```

10.1.1 `dynamic_cast`

RTTI is needed only when we have an object whose class cannot be predicted at compile time. As on pp. 487−489, the simplest way for this to happen is to loop through a container of pointers to objects of different classes. The container is defined in line 12. The loop in line 15 will call the `newfunc` of each object that has this member function.

At compile time, there is no way to tell the type of the object to which `a[i]` will be pointing when line 23 is executed. In fact, `a[i]` will be pointing to a different object each time it is executed. Furthermore, the most derived class (p. 479) of each object will be different each time.

This is where RTTI comes in. The `dynamic_cast` in line 23 is a keyword that is an operator, like `sizeof`, `static_cast`, and `new`. The <angle brackets> and (parentheses) are part of ihe operator. It interrogates an object at runtime, asking it what class it belongs to. If the `a[i]` points to an object of class `middle`, or to an object of a class publicly derived from `middle`, the `dynamic_cast` gives the argument back to us, cast to the data type "pointer to `middle`". Otherwise, it gives us zero.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/rtti/main1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "grandchild.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     base       b(10);
 9     middle     m(10, 20);
10     grandchild g(10, 20, 30);
11
12     const base *const a[] = {&b, &m, &g, 0};
13     const size_t n = sizeof a / sizeof a[0];
14
15     for (size_t i = 0; i < n; ++i) {
16         cout << "a[" << i << "]: ";
17         if (a[i] == 0) {
18             cout << "zero pointer";
19         } else {
20             cout << *a[i] << " ";
21         }
22
23         const middle *const p = dynamic_cast<const middle *>(a[i]);
24         if (p) {
25             p->newfunc();
26         }
27
28         cout << "\n";
29     }
30
31     return EXIT_SUCCESS;
32 }
```

As on pp. 38–39, we can combine the above lines 23–24 to

```
33         if (const middle *const p = dynamic_cast<const middle *>(a[i])) {
```

For Microsoft RTTI, go to `Project` → `Settings` → `C++` `Language` and select the boxes `Enable Exceptions` and `Enable Run-Time Type Information (RTTI)`.

```
a[0]: 10
a[1]: 10, 20 newfunc
a[2]: 10, 20, 30 newfunc
a[3]: zero pointer
```

When we dynamically cast to a pointer type, the argument in the parentheses must be a pointer that was declared to point to a class with at least one virtual member function. If we try to make `base::~base` and `base::print` non-virtual, the `dynamic_cast` will no longer compile.

```
main1.C: In function 'int main()':
main1.C:23:60: error: cannot dynamic_cast 'a[i]' (of type 'const class base*
const') to type 'const class middle*' (source type is not polymorphic)
```

A misuse of RTTI

Actually, the above example didn't need RTTI at all. A cleaner solution would have been to give class `base` a public member function that does nothing.

```
34      virtual void newfunc() const {}
```

Then the above lines 23−26 could simply have been

```
35      a[i]->newfunc();
```

We could even change `middle::newfunc` from public to private.

So this example was a quick and dirty way to do something that could have been done with a virtual function and a bit of foresight. But if extralinguistic reasons prevent us from retrofitting the virtual `newfunc` into the base class, we could fall back on RTTI.

Dynamic cast to a reference

We can also try to dynamically cast an object to a reference to an object. In line 25, `a[i]` is a pointer and `*a[i]` is the object.

A cast to a reference cannot indicate failure by giving us a zero reference: there is a zero pointer, but no such thing as a zero reference. It must indicate failure by throwing an exception. The `bad_cast` in line 28 is derived from the class `exception` on p. 628, from which it inherits the `what` member function.

Note that the operand `a[i]` in the above line 23 is a pointer while the operand `*a[i]` in the following 25 is an object. If `a[i]` is zero, a

```
dynamic_cast<const middle *>(a[i])
```

would execute correctly and give us a zero result, but the `*` in

```
dynamic_cast<const middle&>(*a[i])
```

might crash the program before the `dynamic_cast` has a chance to execute.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/rtti/main2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <typeinfo>   //for the bad_cast exception in line 28
 4 #include "grandchild.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     base       b(10);
10     middle     m(10, 20);
11     grandchild g(10, 20, 30);
12
13     const base *const a[] = {&b, &m, &g, 0};
14     const size_t n = sizeof a / sizeof a[0];
15
16     for (size_t i = 0; i < n; ++i) {
17         cout << "a[" << i << "]: ";
18         if (a[i] == 0) {
```

```
19              cout << "zero pointer";
20          } else {
21              cout << *a[i] << " ";
22
23              try {
24                  const middle& r =
25                      dynamic_cast<const middle&>(*a[i]);
26                  r.newfunc();
27              }
28              catch (const bad_cast& b) {
29                  cout << "caught bad_cast " << b.what();
30              }
31          }
32
33          cout << "\n";
34      }
35
36      return EXIT_SUCCESS;
37  }
```

```
a[0]: 10 caught bad_cast std::bad_cast
a[1]: 10, 20 newfunc
a[2]: 10, 20, 30 newfunc
a[3]: zero pointer
```

Once again, class `base` must have at least one virtual member function. If not, the `dynamic_cast` will not compile.

```
main2.C: In function 'int main()':
main2.C:25:39: error: cannot dynamic_cast '(const base&)((const base*)a[i])'
(of type 'const class base&') to type 'const class middle&' (source type is not
polymorphic)
```

10.1.2 `typeid`

The `dynamic_cast` operator told us if an object is of class `middle`, or of a class publicly derived from `middle`. The `typeid` operator answers a sharper question. It tells us if an object is of class `middle`, and of no further class. In other words, it tells us if the most derived class of the object is `middle`.

The `typeid` in lines 18 and 20 is another keyword that is an operator. (The parentheses are part of the operator.) Its operand can be an object (the `*a[i]` in lines 18 and 20) or the name of a data type (the `middle` in line 20). The operator constructs and returns an anonymous object of class `type_info`, just as the `make_pair` function constructed and returned an anonymous object of type `pair` (pp. 786–787). Note that class `type_info` has an underscore, while the header file `<typeinfo>` does not.

`type_info` is good for two things. In line 18, it has a public member function `name` that returns a string representing the name of the most derived type of an object. (We saw where this string came from on pp. 657–658.) In line 20, `type_info` objects can be compared for equality.

If the `a[i]` in the above line 25 was zero, the `*` in front of it might crash the program. But if the `a[i]` in the following lines 18 and 20 was zero, the `typeid` would be smart enough to intercept the `*` and throw a `bad_typeid` exception instead of crashing.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/rtti/main3.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <typeinfo>   //for type_info and bad_typeid
 4 #include "grandchild.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     base       b(10);
10     middle     m(10, 20);
11     grandchild g(10, 20, 30);
12
13     const base *const a[] = {&b, &m, &g, 0};
14     const size_t n = sizeof a / sizeof a[0];
15
16     for (size_t i = 0; i < n; ++i) {
17         try {
18             cout << "a[" << i << "]: " << typeid(*a[i]).name();
19
20             if (typeid(*a[i]) == typeid(middle)) {
21                 cout << " (middle is the most derived type)";
22             }
23         }
24
25         catch (const bad_typeid& b) {
26             cout << "caught bad_typeid " << b.what();
27         }
28
29         cout << "\n";
30     }
31
32     return EXIT_SUCCESS;
33 }
```

On my platform, the string representation begins with the number of characters in the name.

```
a[0]: 4base
a[1]: 6middle (middle is the most derived type)
a[2]: 10grandchild
a[3]: caught bad_typeid std::bad_typeid
```

We can change the `*a[i]` to `a[i]` in the above lines 18 and 20. If we do this, however, the `typeid` would merely tell us that `a[i]` is a pointer to a `base`. (On my platform, `PK` is a read-only pointer.)

```
a[0]: PK4base
a[1]: PK4base
a[2]: PK4base
a[3]: PK4base
```

But even at compile time we already knew that each `a[i]` is a pointer to a `base`. For this reason, `typeid` is most useful when its operand is a pointer dereferenced with `*` or `[]`.

As with `dynamic_cast`, the pointer in the operand must be declared to point to a class with at least one virtual function. If we change the `a[i]` back to `*a[i]` in the above lines 18 and 20, and make `base::~base` and `base::print` non-virtual, the `typeid` will still compile but its output collapses to

the following.

```
a[0]: 4base
a[1]: 4base
a[2]: 4base
a[3]: 4base
```

The copy constructor and `operator=` of class `type_info` are private, like those of classes `wabbit` (p. 200) and `ostream` (pp. 324−326). This means that the above line 18 could not be changed to

```
34          //Try to save a copy of the type_info for future use.
35          const type_info t = typeid(*a[i]);
36          cout << "a[" << i << "]: " << t.name();
```

But they could be changed to

```
37          //Save a reference to the type_info for future use.
38          const type_info& t = typeid(*a[i]);
39          cout << "a[" << i << "]: " << t.name();
```

▼ Homework 10.1.2a: print the name of the data type

Our very first template was the general template for the `min` function in `min2.C` on pp. 637−638. Have the template print out

```
1     cout << "min<" << typeid(T).name() << ">\n";
```

▲

▼ Homework 10.1.2b: print the name of the data type

Now that we have `typeid`, the `name` argument in the template function `print` on pp. 676−677 is unnecessary. Change the function to the following.

```
1  template <class T>
2  void print(const T *p)
3  {
4          const typename T::layout& flay =
5                  reinterpret_cast<const typename T::layout &>(*p);
6
7          cout << typeid(T).name() << " at address " << p
8          << " has an f whose address is "
9          //etc.
10
```

▲

▼ Homework 10.1.2c: write a rabbit game debugger

To record the data type of every object on the master list, insert the following code at the end of the constructor for class `game`. In line 10, `*it` is a pointer to a `wabbit` and `**it` is the `wabbit`.

```
1     ofstream ost("outfile");
2     if (!ost) {
3         couldn't open output file;
4     }
5
6     for (master_t::const_iterator it = master.begin(); it != master.end();
7         ++it) {
8
```

```
 9         try {
10             ost << typeid(**it).name();
11         }
12         catch (const bad_typeid& b) {
13             ost << b.what();
14         }
15
16         ost << "\n";
17     }
18 } //When we return from game::game, ost is destructed, closing the file.
```

Even better, call the `transform` algorithm instead of fooling around with `master_t::const_iterator` and writing your own loop. Loops are written for many purposes. Calling `transform` documents the purpose of this one. The loop, the output file, and the `typeid` are cleanly separated from each other. And you'll probably have many containers of pointers that could benefit from the function object `get_id`.

```
19 //POINTER must be dereferencable with a * because of line 25.
20
21 template <class POINTER>
22 struct get_id: public unary_function<POINTER, const char *> {
23     const char *operator()(POINTER p) const {
24         try {
25             return typeid(*p).name();
26         }
27         catch (const bad_typeid& b) {
28             return b.what();
29         }
30     }
31 };
32
33     //at the end of game::game
34     ofstream ost("outfile");
35     if (!ost) {
36         couldn't open output file;
37     }
38
39     transform(
40         master.begin(), master.end(),
41         ostream_iterator<const char *>(ost, "\n"),
42         get_id<master_t::value_type>()
43     );
44 }
```

Every pointer on the master list should point to an object of a class derived from class `wabbit`. Does it?
▲

▼ Homework 10.1.2d: create a multiplying rabbit

The `grandchild` classes are located in a two-dimensional space of classes. The dimensions represent the choice of movement and punishment, and the rank in the food chain. Imagine a third dimension, giving a strategy for reproduction. An animal derived from class `sterile`, for example, would never give birth to babies. An `asexual` animal would leave a trail of offspring behind it as it moves. A `vampire` would replace any animal that it kills with another `vampire`. Et cetera.

To build this third dimension, we would have to instate a new pure virtual function `reproduce` in class `wabbit`, create a new set of derived classes, and endow every grandchild class with a third parent.

But instead of doing the job right, let's use RTTI to build a quick and dirty multiplying rabbit. The following code is a kludge.

Create a multiplying rabbit data type.

```
1     typedef grandchild<brownian, victim, 'm'> multiplying_rabbit_t;
```

It will have to be visible to `game::game` and `wabbit::move`.

Insert lines 6–11 into `wabbit::move`.

```
2         if (!I_ate_him) {
3             //I bumped into a wabbit that I could neither eat nor
4             //be eaten by.
5
6             if (I and the other wabbit are both
7                 multiplying_rabbit's) {
8                 dynamically allocate a new multiplying_rabbit
9                 in the unoccupied location that is closest to
10                this wabbit.
11            }
12
13            punish();
14            return true;
15        }
```

To watch them multiply, pen two `multiplying_rabbit`'s inside a box of `boulder`'s.

```
bbbbbbb
b.....b
bm...mb
b.....b
bbbbbbb
```

▲

10.2 Namespaces

Lots of things can have names: variables, functions, classes, enumerations, typedefs, etc. Let's call them *named items.* Of course, not all variable have names; think of the ever-present anonymous temporaries. When discussing namespaces, source files and preprocessor macros are not considered named items.

A *namespace* is a family of named items that share a common last name. The items are called the *members* of the namespace.

Imagine that two teams of programmers, working in isolation from each other, have presented us with the following two header files. If a program tries to `#include` both of them, we will get name collisions. The two variables `x`, the two functions `f`, and the two classes `date` are totally different—they just happen to have the same names.

To include a header file in more than one `.C` file of the same program, the header file must not define anything that occupies memory and that would be visible in more than one `.C` file. That's why line 6 has the keyword `extern`: it makes the line a declaration, not a definition. Line 7 is also a declaration, even without the `extern`. The absence of a {function body} prevents it from being a definition.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/namespace/pre/lib1.h`

```
1 #ifndef LIB1H
2 #define LIB1H
3 #include <iostream>
4 using namespace std;
```

```
 5
 6 extern int x;
 7 void f();
 8
 9 class date {
10     int year;
11     int month;
12     int day;
13 public:
14     date() {cout << "lib1::date::date, x == " << x << "\n";}
15     void print() const;
16 };
17 #endif
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/namespace/pre/lib2.h

```
 1 #ifndef LIB2H
 2 #define LIB2H
 3 #include <iostream>
 4 using namespace std;
 5
 6 extern double x;
 7 void f();
 8
 9 class date {
10     int day;
11 public:
12     date() {cout << "lib2::date::date, x == " << x << "\n";}
13     void print() const;
14 };
15 #endif
```

10.2.1 Declare the members of a Namespace

Our program can `#include` both header files by giving a different last name to all the named items in each file.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/namespace/lib1.h

```
 1 #ifndef LIB1H
 2 #define LIB1H
 3 #include <iostream>
 4 using namespace std;
 5
 6 namespace lib1 {
 7     extern int x;
 8     void f();
 9
10     class date {
11         int year;
12         int month;
13         int day;
14     public:
15         date() {cout << "lib1::date::date, x == " << x << "\n";}
```

```
16          void print() const;
17      };                          //semicolon at end of class declaration
18  }                               //no semicolon at end of namespace
19  #endif
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/namespace/lib2.h

```
 1  #ifndef LIB2H
 2  #define LIB2H
 3  #include <iostream>
 4  using namespace std;
 5
 6  namespace lib2 {
 7      extern double x;
 8      void f();
 9
10      class date {
11          int day;
12      public:
13          date() {cout << "lib2::date::date, x == " << x << "\n";}
14          void print() const;
15      };
16  }
17  #endif
```

10.2.2 Define the members of a Namespace

We must do more than just declare the members of a namespace. We must also define them. One way to define them as members of `lib1` is with the *explicit qualification* `lib1::` in lines 5, 7, and 13.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/namespace/lib1.C

```
 1  #include <iostream>
 2  #include "lib1.h"
 3  using namespace std;
 4
 5  int lib1::x = 10;   //looks like the definition of a static data member
 6
 7  void lib1::f()      //looks like the definition of a member function
 8  {
 9      cout << "lib1::f: x == " << x
10          << ", sizeof (date) == " << sizeof (date) << "\n";
11  }
12
13  void lib1::date::print() const   //definition of non-inline member function
14  {
15      cout << "lib1::date::print, x == " << x << "\n";
16  }
```

Another way to define them as members of a namespace is to enclose them in the following lines 5 and 19. We did this when defining `iterator_traits<node::iterator>` as a member of namespace std; in lines 37 and 46 of `node.h` on p. 806.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/namespace/braces/lib1.C

```
 1 #include <iostream>
 2 #include "lib1.h"
 3 using namespace std;
 4
 5 namespace lib1 {
 6 int x = 10;
 7
 8 void f()
 9 {
10     cout << "lib1::f: x == " << x
11         << ", sizeof (date) == " << sizeof (date) << "\n";
12 }
13
14 void date::print() const   //definition of non-inline member function
15 {
16     cout << "lib1::date::print, x == " << x << "\n";
17 }
18
19 }   //no semicolon at end of namespace
```

We'll do the same for namespace `lib2`.

10.2.3 Use the Members of a Namespace

The members of a namespace can be used in three ways.

(1) We can write the explicit qualification `lib1::` and `lib2::` in lines 9−10, 13−14, 17−18.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/namespace/main1.C

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "lib1.h"
 4 #include "lib2.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     cout << lib1::x << "\n";
10     cout << lib2::x << "\n";
11     //cout << x << "\n"; //won't compile: name collision
12
13     lib1::f();
14     lib2::f();
15     //f();              //won't compile
16
17     lib1::date d1;
18     lib2::date d2;
19     //date d3;          //won't compile
20
21     return EXIT_SUCCESS;
22 }
```

```
10
20
lib1::f: x == 10, sizeof (date) == 12
lib2::f: x == 20, sizeof (date) == 4
lib1::date::date, x == 10
lib2::date::date, x == 20
```

(2) If we're going to use `lib1::x` more frequently than `lib2::x`, and `lib2::f` more frequently than `lib1::f`, we can write the *using declarations* in lines 7–8. A using declaration puts us on a first-name basis with our favorite item with a give name.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/namespace/main2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;         //using directive
 4
 5 #include "lib1.h"
 6 #include "lib2.h"
 7 using lib1::x;               //using declaration
 8 using lib2::f;               //using declaration
 9
10 int main()
11 {
12     cout << x << "\n";       //lib1::x
13     cout << lib2::x << "\n"; //override the using declaration
14
15     f();                     //lib2::f
16     lib1::f();               //override the using declaration
17
18     lib1::date d1;           //didn't write a using declaration for these
19     lib2::date d2;
20
21     return EXIT_SUCCESS;
22 }
```

```
10
20
lib2::f: x == 20, sizeof (date) == 4
lib1::f: x == 10, sizeof (date) == 12
lib1::date::date, x == 10
lib2::date::date, x == 20
```

(3) If we're going to use most of the members of namespace `lib1` more frequently than the members of namespace `lib2`, write the *using directive* in line 7.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/namespace/main3.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;         //using directive
4
5 #include "lib1.h"
6 #include "lib2.h"
7 using namespace lib1;        //using directive
```

```
 8
 9 int main()
10 {
11     cout << x << "\n";       //lib1::x and std::cout
12     cout << lib2::x << "\n";
13
14     f();                     //lib1::f
15     lib2::f();
16
17     date d1;                 //lib1::date
18     lib2::date d2;
19
20     return EXIT_SUCCESS;
21 }
```

```
10
20
lib1::f: x == 10, sizeof (date) == 12
lib2::f: x == 20, sizeof (date) == 4
lib1::date::date, x == 10
lib2::date::date, x == 20
```

How they gave a last name before namespaces were invented

A namespace is just a class whose members are all public and whose data members and member functions are all `static`. There is no need, however, for the data members and member functions of the nested classes to be static. Class `music`, `verb`, and `adjective` on pp. 227−228, and class `gates` on p. 421 should have been namespaces.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/namespace/static_members/lib1.h`

```
 1 #ifndef LIB1H
 2 #define LIB1H
 3 #include <iostream>
 4 using namespace std;
 5
 6 class lib1 {
 7 public:
 8     static int x;
 9     static void f();
10
11     class date {
12         int year;   //year doesn't have to be static member of date
13         int month;
14         int day;
15     public:
16         date() {cout << "lib1::date::date, x == " << x << "\n";}
17         void print() const;
18     };
19 };
20 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/namespace/static_members/lib1.C`

```
 1 #include <iostream>
 2 #include "lib1.h"
 3 using namespace std;
 4
 5 int lib1::x = 10;
 6
 7 void lib1::f()
 8 {
 9     cout << "lib1::f: x == " << x
10         << ", sizeof (date) == " << sizeof (date) << "\n";
11 }
12
13 void lib1::date::print() const
14 {
15     cout << "lib1::date::print, x == " << x << "\n";
16 }
```

Overload a base class member function

Here's a more elegant solution to a problem we solved back on p. 495. We want class `derived` to have two different member functions named `f`: one inherited from the base class, and one appearing for the first time in class We originally did this with a call-through, but it's easier to write the using declaration in line 12. `derived`.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class base {
 6 public:
 7     void f(int i) const {cout << i << "\n";}                //Print in decimal.
 8 };
 9
10 class derived: public base {
11 public:
12     using base::f;   //a using declaration instead of the call-through
13     void f(char c) const {cout << "'" << c << "'" << "\n";} //Print a character.
14 };
15
16 int main()
17 {
18     derived d;
19     d.f('A');   //Calls the derived::f in line 13.
20     d.f(66);    //Calls the derived::f in line 12,
21                 //which is just another name for base::f.
22     return EXIT_SUCCESS;
23 }
```

```
'A'
66
```

Without the above line 12, the output would have been

```
'A'
'B'
```

10.2.4 An Anonymous Namespace

As in C, a named item can be made local to one source file with the keyword `static`. The two variables `x` and the two functions `f` will not cause name collisions. Each can be mentioned only in its own source file.

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

```
1 #ifndef HEADERH
2 #define HEADERH
3 void f1();
4 void f2();
5 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/namespace/static/file1.C`

```
1 #include <iostream>
2 #include "header.h"
3 using namespace std;
4
5 static int x = 10;
6 static void f() {cout << "f in file1, x == " << x << "\n";}
7 void f1() {f();}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/namespace/static/file2.C`

```
1 #include <iostream>
2 #include "header.h"
3 using namespace std;
4
5 static double x = 20;
6 static void f() {cout << "f in file2, x == " << x << "\n";}
7 void f2() {f();}
```

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

```
1 #include <cstdlib>
2 #include "header.h"
3 using namespace std;
4
5 int main()
6 {
7     f1();
8     f2();
9     return EXIT_SUCCESS;
10 }
```

```
f in file1, x == 10
f in file2, x == 20
```

An easier way to do the same thing is to give each `.C` file its own anonymous (nameless) namespace. The members of an anonymous namespace can be mentioned only in that source file.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/namespace/anonymous/file1.C`

```
 1 #include <iostream>
 2 #include "header.h"
 3 using namespace std;
 4
 5 namespace {
 6     int x = 10;
 7     void f() {cout << "f in file1, x == " << x << "\n";}
 8 }
 9
10 void f1() {f();}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/namespace/anonymous/file2.C`

```
 1 #include <iostream>
 2 #include "header.h"
 3 using namespace std;
 4
 5 namespace {
 6     double x = 20;
 7     void f() {cout << "f in file2, x == " << x << "\n";}
 8 }
 9
10 void f2() {f();}
```

With the same `header.h` and `main.C`, we get the same output.

```
f in file1, x == 10
f in file2, x == 20
```

Class `_print` 7 can belong to an anonymous namespace in the example with a dispatching function in Chapter 7.

10.2.5 Namespace Alias

Here is the original version of a class `date`, followed by an upgrade.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/namespace/alias/version1.h`

```
 1 #ifndef VERSION1H
 2 #define VERSION1H
 3 #include <iostream>
 4 using namespace std;
 5
 6 namespace version1 {
 7     class date {
 8     public:
 9         date() {cout << "version1::date\n";}
```

```
10      };
11  }
12  #endif
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/namespace/alias/version2.h

```
 1  #ifndef VERSION2H
 2  #define VERSION2H
 3  #include <iostream>
 4  using namespace std;
 5
 6  namespace version2 {
 7      class date {
 8      public:
 9          date() {cout << "version2::date\n";}
10      };
11  }
12  #endif
```

Line 5 creates an *alias* for the namespace `version1`. To migrate to `version2`, only one word in the program has to be changed.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/namespace/alias/main1.C

```
 1  #include <cstdlib>
 2  #include "version1.h"
 3  using namespace std;            //using directive
 4
 5  namespace current = version1;   //namespace alias
 6
 7  int main()
 8  {
 9      current::date d;
10      return EXIT_SUCCESS;
11  }
```

```
version1::date
```

We can even combine the namespace alias in line 5 with the using directive in line 6.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/namespace/alias/main2.C

```
 1  #include <cstdlib>
 2  #include "version1.h"
 3  using namespace std;            //using directive
 4
 5  namespace current = version1;   //namespace alias
 6  using namespace current;        //using directive
 7
 8  int main()
 9  {
10      date d;
11      return EXIT_SUCCESS;
12  }
```

```
version1::date
```

10.2.6 Namespaces in the Header Files

iostream vs. iostream.h

The named items declared in the header file `iostream.h` belong to no namespace.

```
1 //Excerpt from iostream.h
2
3 istream  cin(argument(s), if any, for constructor);
4 ostream cout(argument(s), if any, for constructor);
5 ostream cerr(argument(s), if any, for constructor);
6 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/namespace/header/main1.C`

```
1 #include <iostream.h>  //for cout
2 #include <stdlib.h>    //for EXIT_SUCCESS
3
4 int main()
5 {
6     cout << "hello\n";     //cout belongs to no namespace
7     return EXIT_SUCCESS;
8 }
```

The named items declared in the header file `iostream` belong to the ‘‘standard’’ namespace `std`. Ditto for all the other C++ header files that have no C counterparts: `vector`, `list`, `algorithm`, etc.

```
9 //Excerpt from iostream
10
11 namespace std {
12     istream  cin(argument(s), if any, for constructor);
13     ostream cout(argument(s), if any, for constructor);
14     ostream cerr(argument(s), if any, for constructor);
15 }
```

When including `iostream`, we must therefore use either explicit qualification or a using directive.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/namespace/header/main2.C`

```
1 #include <iostream>   //for std::cout
2 #include <cstdlib>    //for EXIT_SUCCESS;
3
4 int main()
5 {
6     std::cout << "hello\n";   //explicit qualification
7     return EXIT_SUCCESS;
8 }
```

```
hello
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/namespace/header/main3.C

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;   //using directive
4
5 int main()
6 {
7     cout << "hello\n";
8     return EXIT_SUCCESS;
9 }
```

```
hello
```

The C header files come in pairs

The C++ Standard Library contains two versions of each header file in the C Standard Library. The ones with a trailing `.h` and no leading `c` declare items that belong to no namespace; the ones with a leading `c` and no trailing `.h` declare items that belong to namespace `std`. Macros, such as `EXIT_SUCCESS` and `EXIT_FAILURE`, whether defined in `stdlib.h` or `cstdlib.h`, belong to no namespace. A macro cannot belong to a namespace.

no namespace	`ctype.h`	`math.h`	`stddef.h`	`stdlib.h`	`string.h`	`time.h`
namespace `std`	`cctype`	`cmath`	`cstddef`	`cstdlib`	`cstring`	`ctime`

Another example is `<cfloat>` on p. 747.

The three string header files

There are three string header files. As in the above chart, `string.h` declares C Standard Library functions that belong to no namespace, while `cstring` declares the same functions belonging to namespace `std`.

```
1 //Excerpt from string.h
2 #include <stddef.h>   //for size_t
3
4 //C Standard Library functions:
5 size_t strlen(const char *s);
6 int strcmp(const char *s1, const char *s2);

7 //Excerpt from cstring
8 #include <cstddef>    //for size_t
9
10 namespace std {
11     //C Standard Library functions:
12     size_t strlen(const char *s);
13     int strcmp(const char *s1, const char *s2);
14 }
```

The third header file declares the C++ class `string`.

```
15 //Excerpt from string
16
17 namespace std {
18     class string {
19         //etc.
```

```
20      public:
21          string(const char *s);
22          //etc.
23      };
24  }
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/namespace/header/main4.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cstring>   //for strlen in line 9
 4 #include <string>    //for class string in line 11
 5 using namespace std;
 6
 7 int main(int argc, char **argv)
 8 {
 9     cout << strlen(argv[0]) << "\n";   //argv[0] is just a char *
10
11     string s = argv[0];
12     cout << s.size() << "\n";
13
14     return EXIT_SUCCESS;
15 }
```

```
9
9
```

10.3 Internationalization (I18n)

10.3.1 Locales in C and C++

Each language and/or country has its own set of formatting conventions. Examples are the decimal point character, the currency symbol, and the order in which the day, month, and year of a date are printed. A complete set of conventions is called a *locale.*

We can change a the locale of a program and create new locales. Best of all, we can ignore locales if internationalization is not an issue. The latter is often called ‘‘i18n’’ because it has eighteen letters between the I and the n.

Each locale has a name. Unfortunately, each platform gives you a different set of locales with a different set of names. To see the names on my platform (Sun Solaris Unix), we give the `-a` option (‘‘all’’) to the `locale` program. Canadian French, for example, is `fr_CA`: lowercase language and uppercase country. These names will be used as an argument of the function `setlocale` in C, and of a constructor for class `locale` in C++ .

```
1$ locale -a | sort -df | pr -3 -l6                    minus lowercase L six
C                       fr_FR.UTF-8             POSIX
de_DE.UTF-8             it_IT.UTF-8             pt_BR.UTF-8
en_US.UTF-8             ja_JP.UTF-8             zh_CN.UTF-8
es_ES.UTF-8             ko_KR.UTF-8             zh_TW.UTF-8
```

To see the names on Windows,

`Start → Settings → Control Panel → Regional and Language Options`

http://msdn.microsoft.com/library/default.asp
?url=/library/en-us/vclib/html/_crt_Language_and_Country_Strings.asp

A locale in C

At any given time, a single locale governs all aspects of a C program. These include the standard input, standard output, and functions such as the `strftime` in line 60, the `isprint` in 68, and the `strcoll` in 89. To see the name of the current locale, give `setlocale` the argument `NULL` in lines 16 and 56. There is no telling how long a name will be. To save it, lines 17–23 must copy it into a block of dynamically allocated memory.

On my platform, the dynamic allocation is overkill because a C program always starts in a locale named `"C"`. To change the locale, pass a different name to `setlocale` in line 37. A C locale consists of six categories of formats: numeric, date and time, monetary, etc. We can set all of them with `LC_ALL`, or only one of them with `LC_NUMERIC`, `LC_TIME`, `LC_MONETARY`, etc.

In lines 57–58, the single quote lets the locale separate the digits of a number into thousands if it wants to. In line 68, `isprint` needs the cast for the reason on pp. 63–64. In line 79, the `%u` works only on machines where `size_t` is another name for `unsigned`.

The loop in lines 67–74 is careful not to increment a `char` that already contains `CHAR_MAX`. An overflow would probably wrap around safely, even on a machine where `char`'s are signed. But why risk it? Officially, overflow causes ‘‘undefined behavior’’.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/locale/locale.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>   /* for malloc, NULL, exit, EXIT_*, qsort */
 3 #include <string.h>   /* for strlen, strcpy, strcoll */
 4 #include <locale.h>   /* for setlocale, LC_ALL */
 5 #include <time.h>     /* for time, localtime, strftime, time_t, struct tm */
 6 #include <limits.h>   /* for CHAR_MIN, CHAR_MAX */
 7 #include <ctype.h>    /* for isprint */
 8
 9 void set(const char *name);
10 void print();
11 int compare(const void *p1, const void *p2);   /* function passed to qsort */
12
13 int main(int argc, char **argv)
14 {
15     /* Save the name of the current locale of the entire program. */
16     const char *const p = setlocale(LC_ALL, NULL);
17     char *const save = malloc(strlen(p) + 1);
18     if (save == NULL) {
19         fprintf(stderr, "%s: couldn't save original locale \"%s\"\n",
20             argv[0], p);
21         return EXIT_FAILURE;
22     }
23     strcpy(save, p);
24
25     print();
26     set("fr_CA"); /* Canadian French */
27     print();
28     set(save);    /* return to saved locale */
29     print();
30
```

```
31      free(save);
32      return EXIT_SUCCESS;
33  }
34
35  void set(const char *name)
36  {
37      if (setlocale(LC_ALL, name) == NULL) {
38          fprintf(stderr, "couldn't setlocale \"%s\".\n", name);
39          exit(EXIT_FAILURE);
40      }
41  }
42
43  void print()
44  {
45      const time_t t = time(NULL);
46      const struct tm *const ptm = localtime(&t);
47
48      char buffer[CHAR_MAX - CHAR_MIN + 2];    /* includes terminating '\0' */
49      char *p = buffer;
50      char c;
51
52      size_t len;           /* # of printable characters in this locale */
53      const size_t n = 70;  /* # of characters to print per line */
54      size_t i;
55
56      printf("Locale \"%s\":\n", setlocale(LC_ALL, NULL));
57      printf("integer: %'d\n", 123456789);
58      printf("double:  %'.3f\n", 123456789.123);
59
60      if (strftime(buffer, sizeof buffer, "%c (%x)", ptm) == 0) {
61          fprintf(stderr, "strftime overflowed\n");
62      } else {
63          printf("time formats: %s\n", buffer);
64      }
65
66      /* Make a buffer of all characters that are printable in this locale. */
67      for (c = CHAR_MIN;; ++c) {
68          if (isprint((unsigned char)c)) {
69              *p++ = c;
70          }
71          if (c == CHAR_MAX) {
72              break;
73          }
74      }
75      *p = '\0';
76
77      len = strlen(buffer);
78      qsort(buffer, len, sizeof (char), compare);
79      printf("collating order for the %u printable characters:\n", len);
80
81      /* Print the buffer n characters per line. */
82      for (i = 0; i < len; i += n) {
83          printf("%.*s\n", n, buffer + i);
84      }
```

```
85      printf("\n");
86  }
87
88  int compare(const void *p1, const void *p2)
89  {
90      const char a[] = {*(const char *)p1, '\0'};
91      const char b[] = {*(const char *)p2, '\0'};
92      return strcoll(a, b);
93  }
```

```
Locale "C":
integer: 123456789
double:  123456789.123
time formats: Tue Apr 08 09:14:38 2014 (04/08/14)
collating order for the 95 printable characters:
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_‘abcde
fghijklmnopqrstuvwxyz{|}~
```

A locale object in C++

In C++, a locale is represented by an object of class `locale`. Each stream can have a different locale, and a stream's `locale` is returned by the member function `getloc` in line 10. A locale can be copied but not modified: all of its non-static data members and member functions are `const`.

There are other ways to construct a locale. Line 13 constructs the ‘‘user's preferred locale'', set by the operating system. Line 14 constructs the *classic* locale, that the era before people gave any thought to i18n. Line 15 constructs the *global* local, with which all newborn streams are imbued. The global locale is initially the classic locale, so `cin` and `cout` were classically imbued.

Line 18 constructs the French Canadian locale. Bear in mind that the argument of the constructor may be a different string on each platform. It must also be a `char *`, not a `string` object. If the name is not recognized, the constructor will throw an exception of class `runtime_error`, derived from class `exception` on p. 628. Line 22 installs the new locale as the global locale. The existing streams will retain their current locales, but any new stream will be Québecois.

Let's construct a stream to make sure. We could open an output file with an `ofstream`, but then we'd have to clean up the disk when the program is over. Instead, we'll open a string with the `ostringstream` in line 30. Its locale has the same name as the global one (line 32) and is in fact equal to the global one (line 33).

C++ programs compiled with the GNU `g++` compiler on my platform do not recognize the names of the locales, so I had to compile this program with the Sun `CC` compiler.

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

```
 1  #include <iostream>
 2  #include <cstdlib>
 3  #include <locale>     //for locale
 4  #include <stdexcept>  //for runtime_error
 5  #include <sstream>    //for ostringstream
 6  using namespace std;
 7
 8  int main()
 9  {
10      const locale loc1 = cout.getloc();     //copy the locale of cout
11      cout << "cout's locale is \"" << loc1.name() << "\".\n";
```

```
12
13      const locale loc2("");                  //the user's preferred locale
14      const locale loc3 = locale::classic(); //static member function
15      const locale loc4;                      //global locale
16
17      try {
18          const locale loc5("fr_CA");
19          cout << "The French Canadian locale is \"" << loc5.name()
20              << "\".\n";
21
22          locale::global(loc5);          //static member function
23          cout << "The new global locale is \"" << locale().name()
24              << "\".\n";
25      }
26      catch (const runtime_error& e) {
27          cerr << e.what() << "\n";
28      }
29
30      ostringstream ost;
31      const locale loc6 = ost.getloc();
32      cout << "The locale of a new stream is \"" << loc6.name() << "\".\n";
33      if (loc6 == locale()) {
34          cout << "ost's locale is the same as the global one.\n";
35      }
36
37
38      return EXIT_SUCCESS;
39  }
```

```
cout's locale is "C".
The locale of a new stream is "C".
ost's locale is the same as the global one.
```

Imbue a stream with a different locale

The expression `locale("fr_CA")` in line 17 constructs an anonymous object of class `locale`. To imbue a stream with it, we pass the object to the `imbue` member function in lines 17 and 24. Each stream has its own locale, so we should also have imbued `cin`, `cerr`, and `clog`.

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

```
 1 #include <iostream>
 2 #include <iomanip>
 3 #include <cstdlib>
 4 #include <locale>
 5 #include <stdexcept>  //for runtime_error
 6 using namespace std;
 7
 8 void print();
 9
10 int main()
11 {
12      const locale save = cout.getloc();   //save cout's locale
13
```

```
14      print();
15
16      try {
17          cout.imbue(locale("fr_CA"));
18      }
19      catch (const runtime_error& e) {
20          cerr << e.what() << "\n";
21      }
22
23      print();
24      cout.imbue(save);                    //return to saved locale
25      print();
26      return EXIT_SUCCESS;
27  }
28
29  void print()
30  {
31      cout << "Locale \"" << cout.getloc().name() << "\":\n"
32          << "integer: " << 123456789 << "\n"
33          << "double:  " << fixed << setprecision(3) << 123456789.123
34          << "\n\n";
35  }
```

```
Locale "C":
integer: 123456789
double:  123456789.123

Locale "C":
integer: 123456789
double:  123456789.123

Locale "C":
integer: 123456789
double:  123456789.123
```

10.3.2 Facets

A C++ locale is made of *facets*. Each facet in a locale belongs to a different data type, all of them derived from class `locale::facet`.

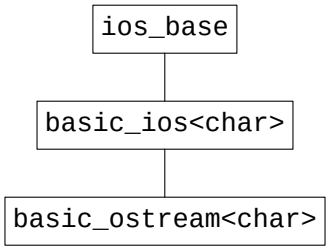
Not every type of facet is present in every locale. To see if a locale has given type of facet, we call the template function `has_facet` (lines 13, 29). The data type of the facet goes in the <angle brackets> and the locale object goes in the (parentheses).

Once we have established that a facet of the desired type is present, we call `use_facet` to access the facet (lines 14, 30). Like a `stream` object, a facet cannot be copied (pp. 324−326), and like a `locale` object, a facet cannot be modified. The return value of `use_facet` can be stored only in a read-only reference.

If we recklessly skip the call to `has_facet` and attempt to `use_` a facet that is not present, `use_facet` will throw an exception of type `bad_cast`, derived from class `exception` on p. 628. I don't expect this will happen here, because `numpunct<char>` and `num_put<char>` are *standard facets*. They should be present in every `locale` object. (Maddeningly, `num_put` has an underscore but `numpunct` doesn't.)

Each type of facet has different member functions. A `numpunct<char>` has the function `decimal_point` in line 17, and a `num_put<char>` has the function `put` in line 32. The functions of one facet often call those of another facet belonging to the same locale; for example, `put` calls `decimal_point`. To see how this happens, we first have to explain why `cout` is passed to `put` twice in line 32.

Recall that `cout` is of class `ostream`, a typedef for class `basic_ostream<char>`. This class was built in layers using inheritance; a more extensive diagram appeared on pp. 383−385.



The base class `ios_base` knows about formatting a stream—field width, left and right justification, even the locale—but its view of reality has one huge gap. It does not know whether the stream's characters are `char`'s or `wchar_t`'s. This information is added in the next layer, `basic_ios`, which deals with characters, buffers, and the `streambuf_iterator`'s that read and write them. For example, the `fill` function inherited by `cout` (p. 354) originates at this layer. But a `basic_ios` does not know whether the stream is input stream or output. This information is added in the last layer, class `basic_ostream`, where the `operator<<` functions are defined.

The `put` in line 32 does not require a complete `basic_ostream<char>` object, or even bits and pieces of the same `basic_ostream<char>` object. It would be happy to write the characters of `1234.56` to any buffer, with formatting specified by any `ios_base` object.

The first argument received by `put` is actually an output iterator that refers to a buffer. In line 32, the first `cout` is implicitly converted to an output iterator that refers to `cout`'s buffer. `cout` is a `basic_ostream<char>`; this type of stream has a buffer whose iterator is of class `ostreambuf_iterator<char>`; this type of iterator has a non-`explicit` constructor whose argument is a `basic_ostream<char>`. The constructor gives us an iterator referring to the `basic_ostream<char>`'s buffer.

The second argument received by `put` is an `ios_base` that holds formatting information and a locale. Any object can be implicitly converted to a public base class, and `ios_base` is a public base of `cout`.

To remember what `put` takes from the two `cout`'s, we could have written line 32 as follows. One loose end: an `ios_base` has no `fill` character (or any other character), so this must be passed separately.

```
1     num_put<char>::iter_type it = f.put(
2         ostreambuf_iterator<char>(cout),  //call a one-arg constructor
3         static_cast<ios_base&>(cout),     //cast derived to base
4         cout.fill(),
5         1234.56
6     );
```

Like the `copy` algorithm, `put` returns the an output iterator of the same type that it received as an argument, referring to a point just beyond the last value written. For an `ostream`, this iterator is of type `ostreambuf_iterator<char>`. This being quite a mouthful, the facet gives us the `iter_type` in line 31 as a shorter name for it. This type of iterator has a `failed` function that tells us if the `put` failed (line 34).

Now we can explain how a member function of one facet can call a member function of another. The number in line 32 has a decimal point, so `put` will call the `decimal_point` function of the `numpunct<char>` facet of the `locale` object of the `ios_base` object that was the second argument of `put`. See p. 1048 for why the `ios_base` must be passed as a non-`const` reference.

For the `convert` template class in line 24, see p. 877.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <locale>
 4 #include <iterator>    //for ostream_iterator
 5 #include <algorithm>   //for transform
 6 #include "convert.h"   //for convert
 7 using namespace std;
 8
 9 int main(int argc, char **argv)
10 {
11     const locale loc = cout.getloc();
12
13     if (has_facet<numpunct<char> >(loc)) {
14         const numpunct<char>& f = use_facet<numpunct<char> >(loc);
15         cout << "truename: \"" << f.truename() << "\"\n"
16             << "falsename: \"" << f.falsename() << "\"\n"
17             << "decimal point: '" << f.decimal_point() << "'\n"
18             << "thousands sep: '" << f.thousands_sep() << "'\n";
19
20         cout << "grouping: ";
21         const string s = f.grouping();
22         transform(s.begin(), s.end(),
23             ostream_iterator<unsigned>(cout, " "),
24             convert<char, unsigned char>()
25         );
26         cout << "\n";
27     }
28
29     if (has_facet<num_put<char> >(loc)) {
30         const num_put<char>& f = use_facet<num_put<char> >(loc);
31         const num_put<char>::iter_type it =
32             f.put(cout, cout, cout.fill(), 1234.56);
33
34         if (it.failed()) {
35             cerr << "put failed\n";
36             return EXIT_FAILURE;
37         }
38         cout << "\n";
39     }
40
41     return EXIT_SUCCESS;
42 }
```

My Sun CC compiler would accept a template function only if at least one of the function arguments had a `T` in its type. `has_facet` and `use_facet` therefore demanded an extra function argument, a pointer to the type I wanted to write as the explicit template argument. To add insult to injury, this change was necessary only if a certain ‘‘Rogue Wave’’ macro is defined. For example, the above line 13 had to

become

```
43 #ifdef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
44     if (has_facet(loc, static_cast<numpunct<char> *>(0))) {
45 #else
46     if (has_facet<numpunct<char> >(loc)) {
47 #endif
```

```
truename: "true"
falsename: "false"
decimal point: '.'
thousands sep: ','
grouping:              classic locale doesn't use the thousands separator at all
1234.56
```

▼ Homework 10.3.2a: other facets

(1) Imbue `cout` with a different locale in the above program. For example, insert the following at line 10½.

```
 1     try {
 2         cout.imbue(locale("fr_CA"));
 3     }
 4     catch (const runtime_error& e) {   //must #include <stdexcept>
 5         cerr << e.what() << "\n";
 6     }
```

On my platform, the new output is

```
```

(2) Does `cout`'s locale have a facet of type `time_put<char>`, similar to `num_put<char>`? If so, make a read-only reference `f` to it. The `put` member function of this facet takes five arguments, the last one being a `strftime` conversion character such as the `%c` in line 60 of `locale.c` on p. 1033.

```
 7     const time_t t = time(0);
 8     const tm *const p = localtime(&t);
 9     const time_put<char>::iter_type it =
10         f.put(cout, cout, cout.fill(), p, 'c');   //like strftime("%c")
11     cout << "\n";
```

```
Tue Apr 08 09:14:55 2014              classic C locale
```

```
```

Try other `strftime` conversion characters, such as `'x'`.

```
04/08/14                              classic C locale
```

```
```

This facet also has a six-argument `put`, taking the beginning and end of a `strftime` format string.

```
12     const char format[] = "%c (%x)";
13     const time_put<char>::iter_type it = f.put(cout, cout, cout.fill(), p,
14         format, format + sizeof format - 1);
```

```
Tue Apr 08 09:15:07 2014 (04/08/14)
```
classic C locale

(3) Does `cout`'s locale have facets of type `moneypunct<char, true>` and `moneypunct<char, false>`, similar to `numpunct<char>`? If so, make a read-only reference `f` to each one and print the return values of some of the member functions. The `bool` controls the currency symbol: `true` for international (`USD`), `false` for local (`$`). The international symbol, if it exists, is always four characters.

```
15      cout << "curr_symbol: \"" << f.curr_symbol() << "\"\n"
16           << "frac_digits: << f.frac_digits() << "\n";
```

```
curr_symbol: ""
frac_digits: 0
```
classic locale, international symbol

```
curr_symbol: ""
frac_digits: 0
```
classic locale, local symbol

(4) Does `cout`'s locale have a facet of type `money_put<char>`, similar to `num_put<char>` and `time_put<char>`? If so, make a read-only reference `f` to it. The `put` member functions of this facet takes five arguments. The second is a `bool` that controls the currency symbol (true for international, false for local); the last is a `long double` or `string` representing the amount of money.

```
17      cout << showbase;    //see the currency symbol, if any
18      const money_put<char>::iter_type it =
19          f.put(cout, true, cout, cout.fill(), 123.45);
```

```
123
```
classic locale, international symbol

```
123
```
classic locale, local symbol

The `pos_format` and `neg_format` member functions of `moneypunct` control whether the sign comes before or after the currency symbol.

```
20      const string name[] = {
21          "none",
22          "space",
23          "symbol",
24          "sign",
25          "value"
26      };
27
28      const money_base::pattern pat = f.pos_format();
29      const size_t n = sizeof pat.field / sizeof pat.field[0];
30      for (size_t i = 0; i < n; ++i) {
31          cout << name[pat.field[i]] << " ";
32      }
33      cout << "\n";
```

(5) Does `cout`'s locale have a facet of type `collate<char>`, similar to `numpunct<char>`? If so, make a read-only reference `f` to it. Print the return value of its `compare` member function. `compare` does not recognize `'\0'` as a string terminator. You have to pass it the address of the character beyond the end of the string.

```
34      const char a[] = "A";    //could have made strings of more than 1 char
35      const char b[] = "~";
36      const int i = f.compare(a, a + sizeof a - 1, b, b + sizeof b - 1);
37
38      if (i < 0) {
39          cout << "\"" << a << "\" comes before \"" << b << "\".\n";
40      } else if (i > 0) {
41          cout << "\"" << b << "\" comes before \"" << a << "\".\n";
42      } else {
43          cout << "\"" << a << "\" and \"" << b
44              << "\" are the same string.\n";
45      }
```

```
"A" comes before "~".                    classic locale
```

(6) Does `cout`'s locale have a facet of type `ctype<char>`, similar to `numpunct<char>`? If so, make a read-only reference `f` to it and print the return values of some of the member functions. The `ctype_base::upper` in line 47 is an enumeration; the others are `lower`, `alpha`, `digit`, etc.

```
46      const char c = '\xC7';          //uppercase C with cedilla
47      cout << boolalpha << f.is(ctype_base::upper, c) << "\n"
48          << f.tolower(c) << "\n";
```

```
false                    classic locale does not recognize Ç as uppercase
Ç                        remains unchanged
```

(7) What other types of facets are there? Look in the `<locale>` header file or the compiler documentation.

▲

A locale is a predicate.

Among its many faculties, a C++ `locale` is a predicate of two arguments. It takes two `string` objects and returns true if the first is less than the second in the locale's collating order (line 19). We can therefore pass it to algorithms such as the `sort` in line 49.

The C Standard Library had a one-argument `isprint` that used the C global locale set by `setlocale`. Its argument was an `int`, with a value that had to be within the range of an `unsigned char`. It usually required a cast (pp. 63–64).

The C++ Standard Library has another `isprint` that takes a `locale` object (line 41). It is a template function whose first argument can be a `char` or `wchar_t`, with any value at all. It needs no cast. The other `<cctype>` functions (`isupper`, `toupper`, etc.) are handled similarly.

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

```
#include <iostream>
#include <cstdlib>
#include <locale>     //for the two-argument isprint
#include <stdexcept>
#include <vector>
#include <string>
#include <iterator>   //for ostream_iterator
#include <algorithm>  //for sort, copy
using namespace std;

void f(const locale& loc);

int main()
{
    const locale loc = cout.getloc();
    const string a = "apple";
    const string b = "banana";

    if (loc(a, b)) {   //if (loc.operator()(a, b)) {
        cout << "\"" << a << "\" comes before \"" << b << "\".\n\n";
    }

    f(loc);

    try {
        f(locale("fr_CA"));
    }
    catch (const runtime_error& e) {
        cerr << e.what() << "\n";
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}

void f(const locale& loc)
{
    vector<string> v;  //a one-char string for each printable character

    for (char c = numeric_limits<char>::min();; ++c) {
        if (isprint(c, loc)) {            //accepts char, returns bool
            v.push_back(string(1, c));    //a string of one c
        }
        if (c == numeric_limits<char>::max()) {
            break;
        }
    }

    sort(v.begin(), v.end(), loc);

    //Print the vector n elements per line.
    ostream_iterator<string> it(cout);
    const vector<string>::size_type n = 70;

```

```
55      for (vector<string>::size_type i = 0; i < v.size(); i += n) {
56          copy(v.begin() + i, min(v.begin() + i + n, v.end()), it);
57          cout << "\n";
58      }
59      cout << "\n";
60  }
```

```
"apple" comes before "banana".

 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_‘abcde
fghijklmnopqrstuvwxyz{|}~
```

The `operator()` function in the above line 19 is a template member function, like the function `rot` in lines 15–16 of `point.h` on p. 724. It is implemented as the following call-through to the `compare` function we just saw on p. 1041.

Recall that our familiar data type `string` is just a typedef for class `basic_string<char>` (p. 688). If `CHAR` is `char`, the `s1` and `s2` in line 59 will be `string`'s. The `data` member function of class `string` is just like `c_str`, except that it doesn't terminate the characters with a `'\0'`.

```
61  template <class CHAR, plus more arguments you don't want to know about>
62  int locale::operator()(const basic_string<CHAR>& s1,
63                         const basic_string<CHAR>& s2) const
64  {
65      const CHAR *const p1 = s1.data();
66      const CHAR *const p2 = s2.data();
67
68      const collate<CHAR>& f = use_facet<collate<CHAR> >(*this);
69      return f.compare(p1, p1 + s1.size(), p2, p2 + s2.size()) < 0;
70  }
```

A locale-specific string?

We have achieved French Canadian order by passing a third argument to the `sort` algorithm in the above line 46. Can we eliminate this argument by inventing a `fr_CA_string` (or a `string<fr_CA>`) that is intrinsically Québecois?

```
1       vector<fr_CA_string> v(argument(s) for constructor);
2       sort(v.begin(), v.end());
```

We have said that a `string` is a `basic_string<char>`. But now let's reveal a bit more of the truth. A `string` is actually a

```
basic_string<char, char_traits<char> >
```

Given a container of these objects, the two-argument `sort` algorithm calls the `operator<` friend of this class, which calls the `compare` member function of class `char_traits<char>`. `compare` returns a negative, zero, or positive `int`, just like `strcmp`, to tell us which argument comes first in the alphabetical order for this particular type of character. Since `compare` does not recognize `'\0'` as a string terminator, it needs an additional argument of type `size_t` telling it how many characters to compare.

```
3   #include <string>  //for char_traits
4   using namespace std;
5
6       if (char_traits<char>::compare("apple", "banana", 5) < 0) {
7           cout << "For arrays of char, "
8                "
```

```
 9      }
```

To create a `fr_CA_string`, we could therefore write a new `compare` function in a class derived from `char_traits<char>`. Line 12 overrides the above `compare` function. The `char_type` in line 12 is another name for `CHAR`, inherited from `char_traits`. Line 15 calls the above `collate<CHAR>::compare` function.

```
10 template <class CHAR>
11 struct fr_CA_char_traits: public char_traits<CHAR> {
12     static int compare(const char_type *p1, const char_type *p2, size_t n) {
13         somehow let f be a reference to the collate<CHAR> facet
14             of the "fr_CA" locale;
15         return f.compare(p1, p1 + 1, p2, p2 + 1);
16     }
17 };
18
19 typedef basic_string<char, fr_CA_char_traits<char> > fr_CA_string;
20
21     vector<fr_CA_string> v(argument(s) for constructor);
22     sort(v.begin(), v.end());
```

But a `fr_CA_string` will not interact with the rest of the standard library. We can't even print one:

```
23     cout << v[0];                    //won't compile
```

The `operator<<` for class `string` is actually

```
24 //There are default values for TRAITS and ALLOCATOR,
25 //so we usually don't write them.
26
27 template <class CHAR, class TRAITS, class ALLOCATOR>
28 basic_ostream<CHAR, TRAITS>& operator<<(basic_ostream<CHAR, TRAITS>&,
29     basic_string<CHAR, TRAITS, ALLOCATOR>
```

`cout` is a

```
basic_ostream<char, char_traits<char> >
```

but a `fr_CA_string` is a

```
basic_string<char, fr_CA_char_traits<char> >
```

Since the traits disagree, the `operator<<` template will not accept them. We would have to write our own `operator<<` and `operator>>`.

In the same way, our new strings cannot be compared to, or assigned to, normal strings with the existing `operator<` or `operator=`. Let's not pursue class `fr_CA_string` any further.

But what if we want a `map` whose subscripts are sorted in `fr_CA` order? We have decided not to create class `fr_CA_string`, so we cannot say

```
30     map<fr_CA_string, int> m; //won't compile: there is no class fr_CA_string
```

We will have to write a third template argument within the <angle brackets>. We cannot write

```
31     map<fr_CA_string, int, locale("fr_CA")> m;   //won't compile
```

because a template argument cannot be an object. It will have to be a class such as the following.

```
32 template <class CHAR>
33 class fr_CA_order {
34 public:
```

```
35     static bool operator()(const basic_string<CHAR>& s1,
36                            const basic_string<CHAR>& s2) {
37
38         somehow let f be a reference to the collate<CHAR> facet
39             of the "fr_CA" locale;
40
41         const CHAR *const p1 = s1.data();
42         const CHAR *const p2 = s2.data();
43         return f.compare(p1, p1 + s1.size(), p2, p2 + size()) < 0;
44     }
45 };
```

Given the above class, we can declare our map:

```
46     map<string, int, fr_CA_order> m;
```

Now how do we get the `f` in the above lines 31−32? There is a class `collate_byname<CHAR>` that is exactly like class `collate<CHAR>`, except that it has a constructor that will take a locale name. (Every class of facet has this `_byname` variant.) We can almost write lines 31−32 as follows. The `f` is now an actual facet, not a reference. We also pass a numeric argument. If this argument was zero (the default), and if the facet was part of a `locale` object (which this isn't), the `locale` object would `delete` the facet when the `locale` object is destructed.

```
47     const collate_byname<CHAR> f("fr_CA", 1);
```

A facet declared at the above lines 31−32 will be destructed at line 37. But the destructor for class `facet` is protected, preventing line 37 from compiling. This was done to discourage us from constructing a facet for local use only. A facet is intended to be part of a locale that will govern an entire stream.

But we are professionals. We will have to derive from `collate_byname<CHAR>` a class whose destructor is public. The `r` in line 42 stands for ''reference count'' .

```
48 template <class CHAR>
49 struct destructable_collate_byname: public collate_byname<CHAR> {
50     destructable_collate_byname(const char *name, size_t r = 0)
51         : collate_byname<CHAR>(name, r) {}
52
53     ~destructable_collate_byname() {}
54 };
```

The above lines 31−32 will now be

```
55     const destructable_collate_byname<CHAR> f("fr_CA", 1);
```

Even better, let `f` be a private, static data member of class `fr_CA_order`.

10.3.3 A locale-sensitive `operator<<` for a built-in type

We can easily write a class `roman_numeral` that prints as a Roman numeral:

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

```
 1 #ifndef ROMAN_NUMERALH
 2 #define ROMAN_NUMERALH
 3 #include <iterator>   //for ostream_iterator
 4 #include <algorithm>  //for fill_n
 5 using namespace std;
 6
 7 class roman_numeral {
```

```
 8     int i;
 9 public:
10     roman_numeral(int initial_i): i(initial_i) {}
11
12     friend ostream& operator<<(ostream& ost, const roman_numeral& n) {
13         fill_n(ostream_iterator<char>(ost), n.i, 'I');   //simplified
14         return ost;
15     }
16 };
17 #endif
```

But what if we wanted to print an `int` as a Roman numeral?

```
1     int i = 10;
2     cout << i << "\n";
```

The `operator<<` that formats an `int` is the flagship member function of class `ostream`. It has already been written for us and is shown below. In fact, it has been engraved in granite in the C++ Standard Library. It can be overridden only by discarding `cout` and deriving a new class from `ostream`.

Bear in mind that the `ostream` in the following line 1 is actually a typedef for class `basic_ostream<char>`. In real life, line 1 would be

```
3 template <class CHAR, class TRAITS = char_traits<CHAR> >
4 basic_ostream<CHAR, TRAITS>& operator<<(int i)
```

and the `char`'s in lines 6–7 would be `CHAR`.

The punchline is line 8, which calls the `put` member function of the `num_put<char>` facet of the `locale` object of the `ostream`. There are actually several `put` functions. The one we saw in line 32 of `facet.C` on p. 1038 took a `double`; others take `long` or `unsigned long`. But no `put` an `int`, which is why line 8 needs the cast.

At the beginning and/or end of every `operator<<` function that makes a direct call to the `put` function of a facet, certain administrative tasks have to be performed, including flushing the buffer and throwing exceptions. A class `ostream::sentry` has been written whose constructor and destructor do this setup and cleanup. Line 3 constructs an anonymous sentry object and checks that the constructor was successful. If the anonymity makes you uncomfortable, give the object a name:

```
5     const sentry s(*this);
6     if (s) {              //if (s.operator bool()) {
```

Do we really need lines 10–12? Can't `put` in line8 call `setstate` for us? Well, `setstate` is a member function of class `basic_ios`, but the second argument received by `put` is merely an an `ios_base`.

Exceptions can be thrown at several places in the `operator<<`. If the stream has no `num_put<char>` facet, the call to `use_facet` in line 6 will throw a `bad_cast` exception (p. 1036). If we have requested it to do so, the `setstate` in line 11 will throw an `ios_base::failure` exception. (Line 7 of `failure.C` on p. 624 shows how to make this request; the following line 22 checks if the request has been made.) The `put` function in line 8 might also throw various unpredictable exceptions.

We set the `failbit` if the `put` function has failed (line 11); we set the `badbit` if any exception has been thrown (line 17). Line 17 itself might throw an `ios_base::failure`, which we contain in lines 16–20. We do this because the `operator<<` function would be more informative if the exception that escapes from it at line 23 could be the one that got us to line 15 in the first place, rather than a predictable `ios_base::failure` from line 17.

Note that an exception thrown by the sentry constructor in line 3 is not caught by line 15. It will escape from the `operator<<`.

```
1 ostream& ostream::operator<<(int i)
```

```
 2 {
 3     if (sentry(*this)) {   //if (sentry(*this).operator bool()) {
 4         try {
 5             const locale& loc = getloc();
 6             const num_put<char>& f = use_facet<num_put<char> >(loc);
 7             const num_put<char>::iter_type it =
 8                 f.put(*this, *this, fill(), static_cast<long>(i));
 9
10             if (it.failed()) {
11                 setstate(failbit);
12             }
13         }
14
15         catch (...) {
16             try {
17                 setstate(badbit);
18             }
19             catch (...) {
20             }
21
22             if (exceptions() & badbit) {
23                 throw;
24             }
25         }
26     }
27
28     return *this;
29 }
```

The `loc` in the above line 5 does not have to be a reference (a `locale` can be copied), but the `f` in line 6 does have to be a reference. To avoid these issues, combine the above lines 5−12 to

```
30             if (use_facet<num_put<char> >(getloc())
31                 .put(*this, *this, fill(), static_cast<long>(i)).failed()) {
32                 setstate(failbit);
33             }
```

A virtual member function of a facet

The `put` functions of class `num_put<char>` are public, non-virtual member functions. Each one does its work by calling a protected, virtual member function `do_put` of the same class. Here is the one for `long`.

```
1 //IT has a default value (ostreambuf_iterator<CHAR>),
2 //so we haven't been bothering to write it within the <angle brackets>.
3
4 template <class CHAR, class IT>
5 IT num_put<CHAR, IT>::put(IT it, ios_base& b, char fill, long lo)
6 {
7     return do_put(it, b, fill, lo);   //Just a one-line call-through.
8 }
```

The `do_put` functions of class `num_put<char>` are rather pedestrian. They call

```
b.width()
```

to see if padding is required, evaluate

```
b.flags() & ios_base::hex
```

to see if an integer should be in hex, and use the return value of

```
use_facet<numpunct<CHAR, IT> >(b.getloc()).decimal_point()
```

as the decimal point for a `double`.

Let's override a `num_put<char>::do_put` with a radically different function, with no hex or decimal point. We'll define it in a class derived from `num_put`.

Line 18 is the first constructor we have seen for a facet. A zero argument ensures that the facet will be `delete`'d by the `locale` to which it belongs. (The `r` stands for "reference count".)

Recall that our familiar data type `string` is just a typedef for class `basic_string<char>` (p. 688). If `CHAR` is `char`, the `s` in line 42 will be a `string`. Similarly, type `ostringstream` is just a typedef for class `basic_ostringstream<char>`. If `CHAR` is `char`, the `ost` in line 29 will be an `ostringstream`. Finally, `ostream_iterator` has always taken a template argument giving the data type of the values that are written (`int`, `double`; in line 32 it is `CHAR`). It can take a second template argument giving the type of character of the stream in which these values are rendered (`char` or `wchar_t`).

Lines 26–30 create a string containing the Roman numeral. (The actual `string` object is inside of `ost`.) Lines 32–36 create a possibly longer string that also contains the filling for left and right justification. See pp. 460–461 for an earlier example of this technique.

Now we can see why the `ios_base` in line 25 must be a read/write reference. It allows line 38 to zero the width.

The header file is named `roman.h`, not `roman_put.h`, because we will probably want to define a `basic_roman_get` in the same file. See p. 938

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

```
 1 #ifndef ROMANH
 2 #define ROMANH
 3 #include <iostream>
 4 #include <locale>     //for num_put
 5 #include <sstream>    //for basic_string, basic_ostringstream
 6 #include <iterator>   //for ostream_iterator
 7 #include <algorithm>  //for fill_n, copy
 8 using namespace std;
 9
10 template <class CHAR, class IT = ostreambuf_iterator<CHAR> >
11 class basic_roman_put: public num_put<CHAR, IT> {
12     typedef typename num_put<CHAR, IT>::iter_type iter_type;
13     typedef typename num_put<CHAR, IT>::char_type char_type;
14
15     iter_type do_put(iter_type it, ios_base& b, char_type fill, long val)
16         const;
17 public:
18     explicit basic_roman_put(size_t r = 0): num_put<CHAR>(r) {}
19 };
20
21 typedef basic_roman_put<char> roman_put;
22
23 template <class CHAR, class IT>
24 typename basic_roman_put<CHAR, IT>::iter_type
25 basic_roman_put<CHAR, IT>::do_put(iter_type it, ios_base& b, char_type fill,
26     long val) const
```

```
27 {
28     //Generate a simplified Roman numeral.
29     basic_ostringstream<CHAR> ost;
30     const ios_base::fmtflags flags = b.flags();
31     const char_type c = flags & ios_base::uppercase ? 'I' : 'i';
32     fill_n(ostream_iterator<CHAR, CHAR>(ost), val, c);
33
34     //Generate the filling.
35     basic_ostringstream<CHAR> ost2;
36     ost2.flags(b.flags());   //Copy b's flags into ost2 (left justify, etc).
37     ost2 << setfill(fill) << setw(b.width()) << ost.str();
38     b.width(0);   //Make the width evaporate after its one use.
39
40     //Copy the Roman numeral and its padding
41     //into the container to which the output iterator it refers.
42     const basic_string<CHAR>& s = ost2.str();
43     return copy(s.begin(), s.end(), it);
44 }
45 #endif
```

The two-argument constructor in line 15 gives us a locale identical to `save`, except that it has the `roman_put` facet instead of the `num_put<char>` from which `roman_put` is derived. The facet constructor argument has a default valuue of zero, ensuring that `loc` will `delete` the `roman_put` facet at line 30.

We did not override the `num_put<char>::do_put` that takes a `double`, so line 26 makes no attempt to print π as a Roman numeral.

Surprisingly, the `has_facet` function cannot distinguish between `basic_roman_put<char>` and `num_put<char>`. But that is not the purpose of `has_facet`. We will see on p. 1056 that these two facets share the same "id", and `has_facet` can distinguish only between types with different id's.

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

```
 1 #include <iostream>
 2 #include <iomanip>
 3 #include <cstdlib>
 4 #include <locale>    //for locale, has_facet
 5 #include "roman.h"   //for roman_put
 6 using namespace std;
 7
 8 int main()
 9 {
10     const locale& save = cout.getloc();
11     if (has_facet<roman_put>(cout.getloc())) {
12         cout << "cout's locale has roman_put.\n";
13     }
14
15     const locale loc(save, new roman_put);
16     cout << "Name of new locale is \"" << loc.name() << "\".\n";
17
18     cout.imbue(loc);
19     if (has_facet<roman_put>(cout.getloc())) {
20         cout << "cout's locale has roman_put.\n";
21     }
22
```

```
23      cout << setfill('*')
24           << setw(5) << 3 << "\n"
25           << left << uppercase << setw(5) << 4 << "\n"
26           << 3.14159 << "\n";
27
28      cout.imbue(save);
29      cout << 3 << "\n";
30      return EXIT_SUCCESS;
31  }
```

```
Name of new locale is "*".      Lines 11−13: it doesn't, but has_facet can't tell.
cout's locale has roman_put.    Line 16: not much of a name.
**iii                           Lines 18−21: now it does.
IIII*
3.14159
3                               Line 26: didn't override double do_put.
```

To test it with wide characters,

```
32      wcout.imbue(locale(wcout.getloc(), new basic_roman_put<wchar_t>));
33      wcout << 3 << "\n";
```

▼ Homework 10.3.3a: do as the Romans do

Write the logic to render Roman numerals greater than 3. No rendering is possible if the number is zero or negative (or greater than MMMMCMXCIX, if we're not using bars). In these cases, do_put should construct and return an iter_type whose failed member function will return true. Just pass zero to the constructor for iter_type.

```
1  //Excerpt from basic_roman_put<CHAR, IT>::do_put.
2
3      if (lo < 0) {
4          return iter_type(0);
5      }
```

You can now print the year as a Roman numeral in the French Revolutionary format (p. 366).

Create a locale in which an unsigned long is rendered in binary. Or create a locale in which a double is rendered as a sign, mantissa, and exponent. See p. 89.

```
1      count.imbue(locale(cout.getloc(), new mantissa<char>));
2      double d = -65;
3      cout << d << "\n";
```

```
-(0.507812 * 2 ** 7)
```

Nothing will go wrong when you print the exponent; the exponent is not a double. But when printing the mantissa, be careful that your do_put does not call itself and go into an infinite loop. In other words, print the mantissa in the traditional format, not in the sign/mantissa/exponent format. Derive a destructable_num_put from num_put with a public destructor .

```
4  //Excerpts from your basic_frexp_put<CHAR, IT>::do_put.
5
6      const destructable_num_put<CHAR, IT>f(1);
7      it = f.put(it, b, fill, mantissa);
8
9      const basic_string<CHAR> s(" * 2 ** ");   //#iclude <string>
```

```
10     it = copy(s.begin(), s.end(), it);        //#include <algorithm>
11     it = f.put(it, b, fill, static_cast<long>(exponent));
```

▲

10.3.4 A locale-sensitive operator<< for a user-defined type

Here is a class date whose operator<< is locale-sensitive. It uses the time_put<char> facet of the ostream's locale.

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

```
 1 #ifndef DATEH
 2 #define DATEH
 3 #include <iostream>
 4 using namespace std;
 5
 6 class date {
 7     int year;
 8     int month;   //1 to 12 inclusive
 9     int day;
10 public:
11     date(int initial_month, int initial_day, int initial_year)
12         : year(initial_year), month(initial_month), day(initial_day) {}
13
14     friend ostream& operator<<(ostream& ost, const date& d);
15 };
16 #endif
```

The operator<< on pp. 1046–1047 was a member of class ostream; it referred to the ostream object as *this. The following operator<< is a member of no class; it refers to the ostream object as ost. It also needs the class names in lines 6, 17, 23, and 28.

Until now, we have obtained a tm structure by calling the localtime function. In lines 8–9, we simply create the tm by hand.

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

```
 1 #include <locale>
 2 #include "date.h"
 3 using namespace std;
 4
 5 ostream& operator<<(ostream& ost, const date& d) {
 6     if (ostream::sentry(ost)) {
 7         try {
 8             const tm t =
 9                 {0, 0, 0, d.day, d.month - 1, d.year - 1900, 0};
10             const locale& loc = ost.getloc();
11             const time_put<char>& f =
12                 use_facet<time_put<char> >(loc);
13             const time_put<char>::iter_type it =
14                 f.put(ost, ost, ost.fill(), &t, 'x');
15
16             if (it.failed()) {
17                 ost.setstate(ios_base::failbit);
18             }
```

```
19          }
20
21          catch (...) {
22              try {
23                  ost.setstate(ios_base::badbit);
24              }
25              catch (...) {
26              }
27
28              if (ost.exceptions() & ios_base::badbit) {
29                  throw;
30              }
31          }
32      }
33      return ost;
34  }
```

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

```
 1  #include <iostream>
 2  #include <cstdlib>
 3  #include <locale>
 4  #include <stdexcept>
 5  #include "date.h"
 6  using namespace std;
 7
 8  int main()
 9  {
10      date d(12, 31, 2014);
11      cout << d << "\n";
12
13      try {
14          cout.imbue(locale("fr_CA"));
15      }
16
17      catch (const runtime_error& e) {
18          cerr << e.what() << "\n";
19      }
20
21      cout << d << "\n";
22      return EXIT_SUCCESS;
23  }
```

```
12/31/14
12/31/14
```

▼ Homework 10.3.4a: do as the Americans do

The `time_put<char>` facet was fine as long as we were satisfied with the formats available with `strftime`. For example, the `'x'` format in the classic locale gave us `12/31/14`. But what if we want `12/31/2014`?

It would be tempting to replace lines 13−14 of the above `operator<<` with the following.

```
24      const char format[] = "%m/%d/%Y";
```

```
25
26      const time_put<char>::iter_type it =
27          f.put(ost, ost, ost.fill(), &t, format, format + sizeof format - 1);
```

But we must not do this. The code in an `operator<<` must work for all locales.

Keep the existing `operator<<` and its call to `time_put<char>::put`. The latter does its work by calling a virtual member function `do_put`, and this is the function we should override. Derive a class from `time_put` with a `do_put` member function that will give us the format `12/31/2014`. As in class `num_put`, `do_put` will have the same arguments and return value as the `put`. However, your `do_put` will ignore the `strftime` format character that it receives from `put`.

▲

10.3.5 A locale-sensitive `operator>>`

The integer `operator>>` is similar to its `operator<<` counterpart on pp. 1046−1047.

The data type `iostate` in line 4 holds a bit pattern consisting of a stream's `eof`, `bad`, and `fail` bits; see p. 332. The value `goodbit` has all three of these bits turned off. (There is no ''good bit'' in the bit pattern.)

Class `num_get<char>` has several `get` functions, but none of them take a reference to an `int`. Line 10 calls the one whose last argument is a reference to a `long`.

The first two arguments of the `get` are a pair of input iterators; the second one represents end-of-input. The return value of `get` is the same type of iterator. It has no `failed` member function, so `get` turns on the bits in its `state` argument to indicate that something has gone wrong.

Lines 19−30 have the same exception handling we saw in `operator<<`.

If the input was successful, lines 32−34 install the new value into `i`. For example, an input stream consisting of the two characters `'3'` and `'\n'` would leave all three bits turned off in `state`. The input might be successful even if end-of-input was encountered while reading the number; an input stream consisting of the single character `'3'` would turn on the `eofbit` but leave the other two bits off. If the input failed, we skip line 33. For example, a stream consisting of the single character `'-'` would turn on the `failbit` bit and leave the other two bits off.

```
 1 istream& istream::operator>>(int& i)   //non-const reference
 2 {
 3     if (sentry(*this)) {             //This sentry is an istream::sentry.
 4         iostate state = goodbit;     //a bit pattern of all zeroes
 5         long lo;
 6
 7         try {
 8             const locale& loc = getloc();
 9             const num_get<char>& f = use_facet<num_get<char> >(loc);
10             f.get(
11                 *this,   //num_get<char>::iter_type(*this)
12                 num_get<char>::iter_type(),
13                 *this,   //static_cast<ios_base&>(*this),
14                 state,   //passed as a non-const reference
15                 lo       //passed as a non-const reference
16             );
17         }
18
19         catch (...) {
20             try {
21                 setstate(badbit);
22             }
```

```
23              catch (...) {
24              }
25
26              if (exceptions() & badbit) {
27                  throw;
28              }
29              return *this;
30          }
31
32          if (state == goodbit || state == eofbit) {
33              i = lo;
34          }
35          setstate(state);
36      }
37
38      return *this;
39  }
```

▼ **Homework 10.3.5a: input a Roman numeral**

Each `get` function of class `num_get` calls a corresponding protected, virtual member function `do_get` of the same class. Derive a template class named `basic_roman_get`, parallel to `basic_roman_put`, with a `do_get` that inputs a `long` in Roman numeral format.

▲

A locale-sensitive operator>> for a user-defined type

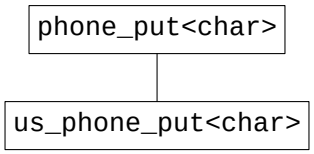
A locale-sensitive `operator>>` for class `date` would include the following code, plus more not shown. The `ist` in line 5 is the `istream` passed to the `operator>>`. As usual, `get_date` calls a protected, virtual member function named `do_get_date`.

Class `time_get<char>` has a `do_get_date` that attempts to input the year, month, day of the date in the order specified by the `'x'` format of `strftime` in your locale (p. 1033). You can write your own `do_get_date` if this does not satisfy you.

```
 1      ios_base::iostate state = ios_base::goodbit;
 2      struct tm t;
 3
 4      try {
 5          const locale loc = ist.getloc();
 6          const time_get<char>& f = use_facet<time_get<char> >(loc);
 7          f.get_date(
 8              ist,   //time_get<char>::iter_type(ist),
 9              time_get<char>::iter_type(),
10              ist,   //static_cast<ios_base&>(ist),
11              state,
12              &t
13          );
14      }
```

10.3.6 A family of facets with its own id

The standard library has input and output facets for numbers, dates, times, and currency. A new data type will require a new family of facets. For the phone number class in line 8, we'll make a base class and one derived class.



To create a family, we derive the base class directly from class `locale::facet` (line 16). The base class must have a public static data member named `id`, of class `locale::id`. This member is declared in line 18, defined in lines 28−29.

Our `operator<<`, in line 32, has three new features.

(1) This the first `operator<<` that we have defined as a template function. It can write to a `char` stream such as `cout`, or to a `wchar_t` stream such as `wcout`. You'll want to do this for all your `operator<<`'s and `operator>>`'s from now on.

(2) This is our first `operator<<` that is neither a member function nor a friend. For simplicity, I provided class `phone` with an `operator unsigned long` that exposes the value of its private data member. If this makes you uncomfortable, let `operator<<` and class `phone_put` be friends of class `phone`. `phone_put` could then have a protected member function taking a `phone` and returns the `unsigned long` value inside of it. Speaking of `unsigned long`, the name of this data type should be written once and for all in a typedef member of class `phone`.

(3) This is our first locale-sensitive `operator<<` that calls other `operator<<`'s to do its work. If we were concerned with speed, we could have given class `phone_put` a public, non-virtual `put` member function and a protected, virtual `do_put` that writes directly to the `ostreambuf`. (I even left the `IT` template argument in, just in case we use it in the future.) But it was simpler to make a `to_str` function whose return value is output with `<<` in line 36. This `<<` has its own sentry, so we don't need to make our own.

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

```
 1 #ifndef PHONEH
 2 #define PHONEH
 3 #include <iostream>
 4 #include <sstream>   //for basic_string and basic_ostringstream
 5 #include <locale>
 6 using namespace std;
 7
 8 class phone {                        //a telephone number
 9     unsigned long n;
10 public:
11     phone(unsigned long initial_n): n(initial_n) {}
12     operator unsigned long() const {return n;}
13 };
14
15 template <class CHAR, class IT = ostreambuf_iterator<CHAR> >
16 class phone_put: public locale::facet {
17 public:
18     static locale::id id;             //declaration
19     explicit phone_put(size_t r = 0): locale::facet(r) {}
20
21     virtual basic_string<CHAR> to_str(const phone& p) const {
22         basic_ostringstream<CHAR> ost;
23         ost << static_cast<unsigned long>(p);   //cast calls line 12
24         return ost.str();
25     }
```

```
26 };
27
28 template <class CHAR, class IT>
29 locale::id phone_put<CHAR, IT>::id;   //definition of static data member
30
31 template <class CHAR>
32 basic_ostream<CHAR>&
33 operator<<(basic_ostream<CHAR>& ost, const phone& p)
34 {
35     const locale& loc = ost.getloc();
36     if (has_facet<phone_put<CHAR> >(loc)) {
37         ost << use_facet<phone_put<CHAR> >(loc).to_str(p);
38     } else {
39         ost << static_cast<unsigned long>(p);   //cast calls line 12
40     }
41     return ost;
42 }
43 #endif
```

The derived class inherits the `id` from the base class. It must not have its own.

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

```
 1 #ifndef US_PHONE_PUTH
 2 #define US_PHONE_PUTH
 3 #include <iostream>
 4 #include <iomanip>   //for setfill, setw
 5 #include <sstream>   //for basic_ostringstream
 6 #include <cstdlib>   //for ldiv, ldiv_t
 7 #include "phone.h"
 8 using namespace std;
 9
10 template <class CHAR, class IT = ostreambuf_iterator<CHAR> >
11 class us_phone_put: public phone_put<CHAR, IT> {
12     basic_string<CHAR> to_str(const phone& p) const {
13         //dash before last four digits
14         const ldiv_t d = ldiv(p, 10000);   //need L for long
15         basic_ostringstream<CHAR> ost;
16         ost.imbue(locale::classic());
17         ost << setfill<CHAR>('0')
18             << setw(3) << d.quot << "-"
19             << setw(4) << d.rem;
20         return ost.str();
21     }
22
23 public:
24     explicit us_phone_put(size_t r = 0): phone_put<CHAR, IT>(r) {}
25 };
26 #endif
```

`has_facet` and `get_facet` pay attention only to the `id` numbers. If two classes of facet share the same `id` number, these functions cannot distinguish between them.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <locale>
 4 #include "phone.h"
 5 #include "us_phone_put.h"
 6 using namespace std;
 7
 8 int main()
 9 {
10     if (!has_facet<phone_put<char> >(cout.getloc())) {
11         cout << "The locale \"" << cout.getloc().name()
12             << "\" has no phone_facet.\n";
13     }
14     const phone p = 2345678;
15     cout << p << "\n\n";
16
17     const locale loc(cout.getloc(), new us_phone_put<char>);
18     if (has_facet<phone_put<char> >(loc)
19         && has_facet<us_phone_put<char> >(loc)) {
20         cout << "The locale \"" << loc.name()
21             << "\" has phone_put<char> and us_phone_put<char>.\n";
22     }
23
24     cout.imbue(loc);
25     cout << p << "\n";
26
27     wcout.imbue(locale(wcout.getloc(), new us_phone_put<wchar_t>));
28     wcout << p << "\n";
29
30     return EXIT_SUCCESS;
31 }
```

```
The locale "C" has no phone_facet.
2345678                                  but you can still print a phone number

The locale "*" has phone_put<char> and us_phone_put<char>.
234-5678
234-5678
```

To test it with wide characters,

```
32     const phone p = 2345678;
33     wcout.imbue(locale(wcout.getloc(), new phone_put<wchar_t>));
34     wcout << p << "\n";
```