9

Banish the Complexity to a Container: a Case Study

9.1 Read and Write the Elements of an STL-Compliant Container

▼ Homework 9.1a:

Almost every line of the rabbit game is wrong. We will rewrite it the way it should have been all along. To show how far we have come, and where we will go, here is code to fill a terminal with uppercase `'A'`s.

(1) In the beginning we called our C functions directly. This exposed the number of dimensions of the terminal: there are two `for` loops and two loop counters, `x` and `y`. It also exposed our choice of coördinate system: Cartesian vs. polar. Compare the loop in lines 23–34 of `main.C` in p. 87.

```
 1 extern "C" {
 2 #include "term.h"
 3 }
 4
 5     term_construct();
 6
 7     for (unsigned y = 0, y < term_ymax(); ++y) {
 8         for (unsigned x = 0; x < term_xmax(); ++x) {
 9             term_put(x, y, 'A');
10         }
11     }
12
13     term_destruct();
```

(2) Then we called the member functions of a `terminal` object. The number of dimensions and the coördinate system are still exposed, although there is now only one `for` loop. Perhaps it was premature: a loop with two counters is an accident waiting to happen. Compare the loop in lines 23–32 of `main.C` on p. 158.

The `put` and `next` member functions in lines 21 and 20 are highly idiosyncratic: no STL container has a `put` or `next`. And although it's dangerously unobvious, the call to `next` changes the values of its arguments `x` and `y`.

```
14 #include "printable.h"
15 #include "terminal.h"
16
17     typedef terminal<printable_t> terminal_t;   //introduced on p. 744
18     terminal_t term('.');
19
20     for (unsigned x = 0, y = 0; y < term.ymax(); term.next(x, y)) {
```

```
21         term.put(x, y, 'A');
22     }
```

(3) We should have made an iterator class whose last name is `terminal_t`. We would then need only one `for` loop and only one loop counter, named `it`. We would loop through the characters in a `terminal_t` in exactly the same way we would loop through the elements in any STL container. Compare the `iterator` loop in lines 26–28 of `iterator.C` on p. 434.

Like the `next` function in the above line 20, the ++ in line 29 will wrap around from the end of one row to the beginning of the next.

```
23 #include "printable.h"
24 #include "terminal.h"
25
26     typedef terminal<printable_t> terminal_t;
27     terminal_t term('.');
28
29     for (terminal_t::iterator it = term.begin(); it != term.end(); ++it) {
30         *it = 'A';                   //it.operator*() = 'A';
31     }
```

Let's imagine that a `terminal_t::iterator` contains two `unsigned` data members, `x` and `y`. The `x` and `y` are therefore still present in the above lines 29–31, just as they were in lines 20–22, but we no longer see them. They are now private data members of `it`. (Or so we imagine for the present.)

Since a `terminal_t` contains `printable_t`'s, the expression `*it` in the above line 30 should be of data type `printable_t`. This should be the return type of the `operator*` member function of class `terminal_t::iterator`. (Or so we imagine for the present.)

(4) Now that we have iterators, we won't have to write any loop at all. The loop has already been written for us in the `fill` algorithm in the STL.

```
32 #include <algorithm>    //for fill
33 #include "printable.h"
34 #include "terminal.h"
35 using namespace std;
36
37     typedef terminal<printable_t> terminal_t;
38     terminal_t term('.');
39
40     fill(term.begin(), term.end(), 'A');
```

Here is a simple definition for the `fill` algorithm. The iterators must be forward because there is no guarantee that mere output iterators can be compared with `!=`.

```
41 //Excerpt from <algorithm>
42
43 template <class FORWARD, class T>
44 void fill(FORWARD first, FORWARD last, const T& t)
45 {
46     for (; first != last; ++first) {
47         *first = t;
48     }
49 }
```

9.2 `difference_type`

Two coördinates or one coördinate?

A `difference_type` is the number we add to an `iterator` to make it move. See the addition in line 22 of `size_type.C` on p. 451.

Here are two possible designs for the data members inside `terminal_t::iterator` and `terminal_t::difference_type`.

(1) `iterator` could be an object with two `unsigned` data members, `x` and `y`. `difference_type` could be an object with two `int` data members, `dx` and `dy`.

(2) `iterator` could be an object with one `unsigned` data member `i`, ranging in value from 0 to `xmax` $\times$ `ymax` $= 80 \times 24 = 1920$ inclusive. For example, the `begin` member function of class `terminal` would construct and return an iterator whose data member had the value 0; the `end` member function would construct and return an iterator whose data member had the value `xmax` $\times$ `ymax`. `difference_type` could be an object with one `int` data member `d`. In fact, `difference_type` could simply be a typedef for `int`, like the `hillary_t` in line 17 of `clinton.h` on p. 420.

But actually we have no choice. The definitions in the STL say that `difference_type` must be an integral type (p. 61), i.e., an `int` or `long` rather than a `class` or `struct`. We must therefore adopt the second design. A `difference_type` will be a typedef for a single number, not an object with two data members.

Adding a difference_type to an iterator

If a `terminal_t::difference_type` is a single number, how can it move an iterator horizontally and vertically?

Adding a `difference_type` of 0 to an `iterator` does not move the `iterator`. Now suppose that the width of the terminal is 80. Then adding a `difference_type` of 80 moves the `iterator` down one row. Adding a `difference_type` of −80 moves the `iterator` up one row. Adding a `difference_type` of 81 moves the `iterator` one space to the lower right. Et cetera:

−162	−161	−160	−159	−158
−82	−81	−80	−79	−78
−2	−1	0	1	2
78	79	80	81	82
158	159	160	161	162

9.3 An Iterator that Yields an Lvalue and an Rvalue

terminal_t::iterator::operator* must do opposite things to the left or right of an equal sign

Line 8 writes into the terminal with an iterator; line 9 reads from the terminal with the iterator. The comments alongside show that both lines call the `operator*` member function of class `terminal_t::iterator`.

```
1 #include "printable.h"
```

```
 2 #include "terminal.h"
 3
 4     typedef terminal<printable_t> terminal_t;
 5     terminal_t term('.');
 6     terminal_t::iterator it = term.begin();
 7
 8     *it = 'A';        //write to terminal:          it.operator*() = 'A';
 9     char c = *it;     //read from terminal: char c = it.operator*();
```

When `operator*` is called in the above line 8, it should ultimately call

```
10     term_put(it.x, it.y, 'A');
```

And when `operator*` is called in the above line 9, it should ultimately call

```
11     term_get(it.x, it.y);
```

But `operator*` takes no arguments, so how can we tell it to do these two opposite things?

Other operators that must do opposite things to the left or right of an =

Before we divulge the answer, observe that there are several operators that must do opposite things to the left or right of an =. In each case, the expression to the right of the = must read information from an object; the expression to the left of the = must write information into an object.

```
1        *p = *q;       //p.operator*()       = q.operator*();
2     p->f1 = q->f2;    //p.operator->()->f1 = q.operator->()->f2;
3     a[10] = b[20];    //a.operator[](10)    = b.operator[](20);
```

Let us digress further. Which of these three operators should we use?

(1) If the object contains only one item of data, or if the object can make only one item available at a time, use the `*` operator to access the data. This makes the object look like a pointer.

(2) If the object contains several items of data, use the `->` operator to access them. This makes the object look like a pointer to a structure.

(3) If the object contains many items of data, use the `[]` operator to access them. This makes the object look like an array.

How to get the two opposite behaviors

Suppose that the value of the expression `*it` in lines 8–9 was an object, not a character. In other words, suppose that the `operator*` member function of the iterator returned an (anonymous) object, not a character. The comments show that line 8 would then call the `operator=` member function of the anonymous object, and line 9 would call the `operator char` member function of the anonymous object:

```
1 #include "printable.h"
2 #include "terminal.h"
3
4     typedef terminal<char> terminal_t;
5     terminal_t term('.');
6     terminal_t::iterator it = term.begin();
7
8     *it = 'A';        //write:          it.operator*().operator=('A');
9     char c = *it;     //read:  char c = it.operator*().operator char();
```

Now that we're calling two different member functions, we can do two different things. The `operator=` member function of the anonymous object will call `term_put`, and the `operator char` member function of the anonymous object will call `term_get`.

The "anonymous object" will be of class `terminal::proxy`. For another proxy object see pp. 828–829.

Class terminal, rewritten as an STL container

We create two classes with the last name `terminal`, `terminal::proxy` and `terminal::iterator`. A `proxy` contains an `iterator` (line 247), which is why class `iterator` (line 34) had to be defined before class `proxy` (line 245). We can't create an object until we have defined its class.

Class `terminal` is a friend of class `iterator` (line 35). Class `terminal` should therefore be able to access the private members of `terminal::iterator`. If it can't in Microsoft, make the members public.

Since we gave only two template arguments to class `std::iterator` (line 34), its `difference_type` defaults to `ptrdiff_t`. If your `ptrdiff_t` is a typedef for `long`, you'll have to change `div` and `div_t` to `ldiv` and `ldiv_t` in line 129.

The left `*` in line 241 calls the `operator*` in line 236.

The `static_cast<CHAR>` in lines 280ff calls the `operator CHAR` above them in line 275.

Without the `static_cast<CHAR>`, line 318 would be torn between two equally good alternatives and would not compile. The line could convert the return value of `background` from `CHAR` to `char` and then perform a `char` comparison with the blank; or it could convert the blank from `char` to `CHAR` and then perform a `CHAR` comparison with the return value of `background`.

The `std::`'s in lines 341–342 cause the `rand` function in line 340 to call the `rand` function in the standard library. Without them, we'd go into an infinite loop. Microsoft people might have to remove the `std`'s (but keep the double colons).

A `CHAR` is the type of character displayed on the screen. But the `char` in line 350 is the type of character typed at the keyboard, which might be a different data type. That's why it's not a `CHAR`.

Constructing a proxy

The constructor for class `proxy` in line 250 is called only when an `iterator` is dereferenced. To enforce this, the constructor is private and is called only from the `operator*` member function of class `iterator` (line 237). The `operator*` is a friend of class `proxy` (line 310), so it should be able to call the constructor. If it can't in Borland, make the constructor public. Borland people should also define the following macro at line 2½.

```
#define _MSC_VER
```

The copy constructor for class `proxy` in line 263 is called in the postfix `operator++` and `operator--` member functions of this class. It will also be called in the following delicate situation.

```
 1 //Excerpt from <algorithm>
 2
 3 template <class T>
 4 void swap(T& a, T& b)
 5 {
 6     const T temp = a;                 //Call swap's copy constructor.
 7     a = b;
 8     b = temp;
 9 }

10     typedef terminal<char> terminal_t;
11     terminal_t term(',');
12     terminal_t::iterator it = term.begin();
13     swap(it[0], it[1]);               //Pass two proxy's to swap.
```

If the `temp` and `a` in the above line 6 contained the same iterator, they would be Siamese twins. The assignment to `a` in line 7 would overwrite the character to which `temp` refers. To prevent this, a `proxy` constructed by a copy constructor contains a character rather than an iterator. The data member `b` in line

246 is false if the `proxy` was constructed by a copy constructor, true otherwise.

I'm afraid that I have turned a `proxy` into the moral equivalent of a union. I concede that it would be more natural to have two different types of `proxy`: the one constructed by `iterator::operator*` referring to a character on the screen, and the one constructed by a copy constructor referring to a character that has been copied from the screen. But I want all `proxy`'s to be the same data type, to satisfy the templates such as `swap` that demand two arguments of the same type.

Two difference_type 's

Class `iterator` receives two typedefs named `difference_type`. Fortunately, both of them stand for the same thing (namely, `ptrdiff_t`).

(1) the `terminal::difference_type` defined in line 22;

(2) the `terminal::iterator::difference_type` that class `terminal::iterator` inherits from its base class `std::iterator` in line 34.

The first `difference_type` eclipses the second one, so the unadorned `difference_type` in line 53 is the `terminal::difference_type` in line 22.

We can make this explicit by changing the `difference_type` to `typename terminal::difference_type` in lines 53, 58, 65, etc. There are two reasons why we might want to do this:

(1) The Sun `CC` compiler thinks that the unadorned `difference_type` in line 53 is the `terminal::iterator::difference_type`.

(2) Some versions of the GNU `g++` recognize that the unadorned `difference_type` in line 53 is the `terminal::difference_type`, but complain about it.

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

```
 1 #ifndef TERMINALH
 2 #define TERMINALH
 3 #include <iostream>    //for ostream and <<
 4 #include <sstream>     //for ostringstream
 5 #include <cstdlib>     //for div and rand
 6 #include <cmath>       //for sqrt
 7 #include <map>
 8 #include <iterator>    //for class std::iterator
 9 #include <algorithm>   //for fill, iterator_category, random_access_iterator_tag
10 using namespace std;
11
12 extern "C" {
13 #include "term.h"
14 }
15 #include "except.h"
16
17 template <class CHAR = char>
18 class terminal {
19 public:
20     typedef CHAR value_type;
21     typedef size_t size_type;
22     typedef ptrdiff_t difference_type;
23
24 private:
25     const CHAR _background;
26     const size_type _xmax;
27     const size_type _ymax;
```

```

public:
    //Need forward declaration of class proxy here,
    //because the name proxy is mentioned several times in class iterator.
    class proxy;

    class iterator: public std::iterator<random_access_iterator_tag, CHAR> {
        friend class terminal;
        friend class proxy;

        const terminal *const t;
        size_type i;   //distance from begin to this iterator
        size_type x() const {return i % t->xmax();}
        size_type y() const {return i / t->xmax();}

    public:
        /*
        An iterator can be off the screen, as long as we do not attempt
        to dereference it.  (For example, the iterator returned by
        terminal::end is off the screen.)  Therefore the constructor
        for class iterator does not check that x and y are legal.
        */
        iterator(const terminal& initial_t, size_type x, size_type y)
            : t(&initial_t), i(y * t->xmax() + x) {}

        iterator& operator+=(const difference_type& d) {
            i += d;
            return *this;
        }

        iterator& operator-=(const difference_type& d) {
            i -= d;
            return *this;
        }

        //Microsoft won't let operator+ be a friend.

        const iterator operator+(difference_type d) const {
            iterator it = *this;   //Construct a copy of *this.
            return it += d;        //return it.operator+=(d);
        }

        const iterator operator-(difference_type d) const {
            iterator it = *this;
            return it -= d;
        }

        iterator& operator++() {return *this += 1;}
        iterator& operator--() {return *this -= 1;}

        const iterator operator++(int) {
            const iterator old = *this;
            ++*this;   //(*this).operator++();
            return old;
```

```
        }

        const iterator operator--(int) {
            const iterator old = *this;
            --*this;
            return old;
        }

        iterator& operator=(const iterator& other) {
            if (t != other.t) {
                ostringstream ost;
                ost << "= with 2 different terminals";
                throw except(ost);
            }
            i = other.i;
            return *this;
        }

        /*
        Return the horizontal component of a difference_type.
        Assuming an 80-character line,
        the horizontal component of -81 would be -1
        the horizontal component of -79 would be  1
        the horizontal component of  81 would be  1
        the horizontal component of  79 would be -1
        */

        difference_type dx(difference_type d) const {
            const difference_type xm = t->xmax();
            difference_type diff = (d + xm / 2) % xm;
            if (diff < 0) {
                diff += xm;
            }
            return diff - xm / 2;
        }

        /*
        Return the vertical component of a difference_type.
        Assuming an 80-character line,
        the vertical component of -81 would be -1
        the vertical component of -79 would be -1
        the vertical component of  81 would be  1
        the vertical component of  79 would be  1
        */

        difference_type dy(difference_type d) const {
            const difference_type xm = t->xmax();
            div_t di = div(d + xm / 2, xm);
            if (di.rem < 0) {
                --di.quot;
            }
            return di.quot;
        }

```

```
        /*
        Return true if it + d would stay on the screen.
        Return false if adding d to it would cause it to wrap around
        from the left edge of the screen to right edge or vice versa.
        */

        bool in_range(difference_type d = 0) const {
            const size_type myx =
                static_cast<difference_type>(x()) + dx(d);
            const size_type myy =
                static_cast<difference_type>(y()) + dy(d);
            return myx < t->xmax() && myy < t->ymax();
        }


        friend difference_type difference(const iterator& it1,
                                          const iterator& it2) {
            if (it1.t != it2.t) {
                ostringstream ost;
                ost << "difference with 2 different terminals";
                throw except(ost);
            }

            return it2.x() - it1.x()
                + it1.t->xmax() * (it2.y() - it1.y());
        }

        friend double dist(const iterator& it1, const iterator& it2) {
            const difference_type d = difference(it1, it2);
            const difference_type ddx = it1.dx(d);
            const difference_type ddy = it1.dy(d);
            return sqrt(static_cast<double>(ddx * ddx + ddy * ddy));
        }

        //Return -1 if d is negative, 1 if d is positive, 0 if d is 0.
        //They're like the three return values of the C strcmp function.

        friend difference_type signum(difference_type d) {
            return d < 0 ? -1 : d > 0;
        }

        //Return the difference_type that would take one step
        //from it1 to it2.

        friend difference_type step(const iterator& it1,
                                    const iterator& it2) {
            const difference_type d = difference(it1, it2);
            const terminal *const t = it1.t;
            return signum(it1.dx(d))
                + t->xmax() * signum(it1.dy(d));
        }

        friend difference_type operator-(const iterator& it1,
                                         const iterator& it2) {
```

```
            if (it1.t != it2.t) {
                ostringstream ost;
                ost << "- with 2 different terminals";
                throw except(ost);
            }
            return it1.i - it2.i;
        }

        friend bool operator==(const iterator& it1,const iterator& it2){
            return it1.t == it2.t && it1.i == it2.i;
        }

        friend bool operator<(const iterator& it1, const iterator& it2){
            if (it1.t != it2.t) {
                ostringstream ost;
                ost << "< with 2 different terminals";
                throw except(ost);
            }
            return it1.i < it2.i;
        }

        friend bool operator<=(const iterator& it1,const iterator& it2){
            if (it1.t != it2.t) {
                ostringstream ost;
                ost << "<= with 2 different terminals";
                throw except(ost);
            }
            return it1.i <= it2.i;
        }

        friend bool operator!=(const iterator& it1,const iterator& it2){
            return !(it1 == it2);
        }

        friend bool operator>(const iterator& it1, const iterator& it2){
            return it2 < it1;
        }

        friend bool operator>=(const iterator& it1,const iterator& it2){
            return it2 <= it1;
        }

        friend ostream& operator<<(ostream& ost, const iterator& it) {
            return ost << "(" << it.x() << ", " << it.y() << ")";
        }

        const proxy operator*() const {
            return proxy(*this);
        }

        const proxy operator[](const difference_type& d) const {
            return *(*this + d);
        }
    };
```

```

    class proxy {
        bool b;   //true if this proxy refers to a CHAR in a terminal
        const iterator it;   //used only if b == true
        mutable CHAR c;      //used only if b == false

        proxy(const iterator& initial_it)
            : b(true), it(initial_it), c('A') {
            if (!it.in_range()) {
                ostringstream ost;
                ost << "location " << it
                    << " off screen whose size is ("
                    << it.t->xmax() << ", " << it.t->ymax()
                    << ")";
                throw except(ost);
            }
        }

    public:
        proxy(const proxy& another)
            : b(false), it(another.it), c(another) {}

        const proxy& operator=(CHAR c) const {
            if (b) {
                term_put(it.x(), it.y(), c);
            } else {
                this->c = c;
            }
            return *this;
        }

        operator CHAR() const {
            return b ? CHAR(term_get(it.x(), it.y())) : c;
        }

        const proxy& operator+=(int i) const {
            return *this = static_cast<CHAR>(*this) + i;
        }

        const proxy& operator-=(int i) const {
            return *this = static_cast<CHAR>(*this) - i;
        }

        const proxy& operator++() const {return *this += 1;}
        const proxy& operator--() const {return *this -= 1;}

        const proxy operator++(int) const {
            const proxy old = *this;
            ++*this;   //(*this).operator++();
            return old;
        }

        const proxy operator--(int) const {
            const proxy old = *this;
```

```
            --*this;   //(*this).operator--();
            return old;
        }

        bool operator==(const CHAR& c) const {
            return static_cast<CHAR>(*this) == c;
        }

        bool operator<(const CHAR& c) const {
            return static_cast<CHAR>(*this) < c;
        }

        friend const proxy iterator::operator*() const;
    };

    terminal(CHAR initial_background = ' ')
        : _background(initial_background),
        _xmax((term_construct(), term_xmax())),
        _ymax(term_ymax())
    {
        if (background() != static_cast<CHAR>(' ')) {
            fill(begin(), end(), background());
        }
    }

    ~terminal() {
        fill(begin(), end(), ' ');
        term_destruct();
    }

    CHAR background() const {return _background;}
    size_type  xmax() const {return _xmax;}
    size_type  ymax() const {return _ymax;}
    size_type  size() const {return xmax() * ymax();}

    iterator begin() const {return iterator(*this, 0, 0);}
    iterator   end() const {return iterator(*this, 0, ymax());}

    static char key() {return term_key();}
    static void wait(int milliseconds) {term_wait(milliseconds);}
    static void beep() {term_beep();}

    difference_type rand() const {
        return (std::rand() % 3 - 1) * xmax() +
                std::rand() % 3 - 1;
    }

    difference_type right() const {return 1;}
    difference_type  down() const {return xmax();}
    difference_type  left() const {return -right();}
    difference_type    up() const {return -down();}

    typedef map<char, difference_type> keypad_t;
    typedef keypad_t::value_type pair_t;
```

```

    keypad_t keypad() const {
        static const pair_t a[] = {
            pair_t('h',  left()),
            pair_t('j',  down()),
            pair_t('k',    up()),
            pair_t('l', right())
        };
        static const size_t n = sizeof a / sizeof a[0];
        return keypad_t(a, a + n);
    }
};
#endif
```

9.4 Template Argument Deduction

As usual, the `operator==` that compares two `iterator`'s is a friend function because it deals evenhandedly with two objects. See the above line friend. Normally the `operator!=` that compares two `iterator`'s would be neither a member function nor friend of any class. It would call `operator==` to do its work:

```
 1 template <class CHAR>
 2 inline bool operator!=(const typename terminal<CHAR>::iterator& it1,
 3                        const typename terminal<CHAR>::iterator& it2)
 4 {
 5     return !(it1 == it2);           //return !(operator==(it1, it2));
 6 }
```

But an obscure restriction prevents us from doing this.

Recall the explicit template arguments on pp. 652–660:

```
 7     cout << min<double>(i, d) << "\n";    //contradictory arguments
 8     cout << pi<float>() << "\n";          //no arguments
```

Most of the time the explicit template arguments would be redundant, so they're not written at all:

```
 9     cout << min<int>(10, 20) << "\n";     //Could write this, but nobody does.
10     cout << min(10, 20) << "\n";          //Please write this.
```

The explicit template argument `<int>` in the above line 9 is not needed because the data type of each function argument in line 12 is a simple `T`. When called from the above line 10, the computer can figure out that the `T` in 12 stands for `int`. This is called *template argument deduction.*

```
11 template <class T>                        //ll. 29–30 of min2.C on p. 637
12 T min(T a, T b)
13 {
14     //etc.
```

Even if the data type of each function argument is a bit more complicated than an unadorned `T`, the explicit template argument is still not required.

```
15 template <class T>
16 const T& min(const T& a, const T& b)      //p. 640
17 {
18     //etc.
```

Even if the `T` is somewhat buried in the data type of the function argument, the explicit template argument is still not required. (From now on we'll use `CHAR` instead of `T`, to agree with the above `terminal.h`.

The computer doesn't care what name we use.)

```
19 template <class CHAR>
20 void f(const terminal<CHAR>& t)
21 {
22 }
```

The function `f` in the above line 20 can still be called without an explicit template argument. When called from line 24, the computer can still deduce that the `CHAR` in the above line 20 stands for `printable_t`.

```
23     terminal<printable_t> term('.');
24     f(term);
```

But there is a limit to how deeply we can bury the `CHAR` in the data type of the argument in the parentheses, while still expecting the computer to deduce what the `CHAR` stands for. If we bury it any further, we'll have to help the computer along with an explicit template argument when we call the function. The computer needs the explicit template argument `<printable_t>` in line 32 to deduce that the `CHAR` in line 26 stands for `printable_t`.

```
25 template <class CHAR>
26 void g(const typename terminal<CHAR>::iterator& it)
27 {
28 }
29
30     terminal<printable_t> term('.');
31     terminal<printable_t>::iterator it = term.begin();
32     g<printable_t>(it);
```

This is the situation in which we would find ourselves in the `operator!=` in the above lines 2–3. We would be forced to call that `operator!=` with an explicit template argument:

```
33     terminal<printable_t>::iterator it1 = term.begin();
34     terminal<printable_t>::iterator it2 = term.begin();
35
36     if (it1 !=<printable_t> it2) {  //won't compile
```

Unfortunately, the syntax of the language doesn't let us slap an explicit template argument on an operator in the above line 36. We would therefore have to call `operator!=` explicitly:

```
37     //Will compile, but no one wants to write this.
38     if (operator!=<printable_it>(it1, it2)) {
```

To allow the user to write the familiar

```
39     if (it1 != it2) {
```

we throw in the towel and let `operator!=` be a friend of class `iterator` in line 198 of the above `terminal.h`. Note that `operator!=` uses no private members of class `iterator`, so in a normal situation it would be neither a member function or a friend. Ditto for the `operator+` in line

The C++ Standard (§14.8.2.4 subsection 9) lists the ways in which we can adorn the `T` or `CHAR` and still have the computer figure out what it stands for. The list includes

```
40 T                                    unadorned
41 const T&                             slightly adorned
42 const name_of_template_class<T>&     heavily adorned
```

but not

```
43 const typename name_of_template_class<T>::name_of_member&
```

The above lines 2–3 stepped over this limit. For an earlier example, see p. 858.

▼ Homework 9.4a: template argument deduction

Look at the definitions for the `operator==` and `operator!=` functions that compare `list<>::iterator`'s. Are they member functions of class `list<>::iterator`, friends, or neither? Do they mention any private member of their class?

Look for them in the header file `<list>` or in other header files included by this one. The functions may be member functions or friends of `list<>::iterator`, or they may be member functions of a base class of `list<>::iterator`.

▲

Test the new class terminal

```
qwertyuiopasdfghjklzxcvbnm................................NEBDL.................
abcdefghijklmnopqrstuvwxyz...............................RyujtxQ................
zyxwvutsrqponmlkjihgfedcba...............................FogeinGW...............
fzbagmkuidcrpnsteqyvxhlowj..............................UzlbackCT...............
........................................................XKsfdhpJZ...............
Please type some characters ending with a q: ............SvrmqwP................
....................m.....................................OIAHM.................
...........................................................YV...................
................................................................................
................................................................................
................................................................................
.......................................Y........................................
........................................XX......................................
.......................................R.Z......................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
```

Line 9 of the following `main.C` defines the `basic_printable` we wrote on pp. 749–753; the `printable_t` in line 15 stands for `basic_printable<char>`. Microsoft people will have to rename the `it1` in lines 106–109 to `it3`, because the `it1` declared in line 100 is still alive for them.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <ctime>
 4
 5 #include <vector>
 6 #include <string>
 7 #include <algorithm>   //for copy, fill, find, greater, random_shuffle, sort
 8
 9 #include "printable.h"
10 #include "terminal.h"
11
12 using namespace std;
```

```
void f();
typedef terminal<printable_t> terminal_t;

class closer_to {
    const terminal_t::iterator it;
public:
    closer_to(const terminal_t::iterator& initial_it) throw()
        : it(initial_it) {}

    bool operator()(terminal_t::difference_type d1,
                    terminal_t::difference_type d2) const throw() {

        return dist(it, it + d1)
             < dist(it, it + d2);
    }
};

int main(int argc, char **argv)
{
    int status = EXIT_FAILURE;
    srand(static_cast<unsigned>(time(0)));

    try {
        f();
        status = EXIT_SUCCESS;
    }

    catch (const exception& e) {
        cerr << argv[0] << ": " << e.what() << "\n";
    }

    catch (...) {
        cerr << argv[0] << ": main caught unexpected exception\n";
    }

    return status;
}

void f()
{
    const terminal_t term('.');
    const terminal_t::difference_type down = term.down();

    const terminal_t::iterator center(term, term.xmax() / 2, term.ymax() / 2);
    terminal_t::iterator it = center;   //copy constructor

    it[1] = it[0] = 'X';

    //Move one step from the center towards the begin, and write a 'Y'.
    const terminal_t::difference_type d = step(center, term.begin());
    it += d;
    *it = 'Y';
```

```
    //Move one step from the center away from the begin, and write a ’Z’.
    it = center;
    it -= d;
    *it = ’Z’;

    //Move a random step away from the center, and write an ’R’.
    it = center;
    it += term.rand();
    *it = ’R’;

    it = term.begin();
    string s = "qwertyuiopasdfghjklzxcvbnm";
    copy(s.begin(), s.end(), it);

    it += down;
    copy(s.begin(), s.end(), it);
    sort(it, it + s.size());

    it += down;
    copy(s.begin(), s.end(), it);
    sort(it, it + s.size(), greater<terminal_t::value_type>());

    it += down;
    copy(s.begin(), s.end(), it);
    random_shuffle(it, it + s.size());

    //Display the characters of the string in order of increasing distance
    //from it0.
    const terminal_t::iterator it0 = it + 3 * term.xmax() / 4;
    s = "abcdefghijklmnopqrstuvwxyz"
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    vector<terminal_t::difference_type> v;
    for (terminal_t::iterator it1 = term.begin(); it1 != term.end(); ++it1) {
        v.push_back(it1 - it0);
    }

    sort(v.begin(), v.end(), closer_to(it0));

    vector<terminal_t::difference_type>::const_iterator it1 = v.begin();
    for (string::iterator it2 = s.begin(); it2 != s.end(); ++it1, ++it2) {
        it0[*it1] = *it2;
    }

    //Midpoint between begin and center:
    term.begin()[(center - term.begin()) / 2] = ’m’;

    it += 2 * down;   //Move two lines down.
    s = "Please type printable characters ending with a q: ";
    it = copy(s.begin(), s.end(), it);

    for (; it != term.end(); ++it) {
        char c;   //uninitialized variable
        while ((c = term.key()) == ’\0’) {
```

```
121         }
122
123         if (c == 'q') {
124             break;
125         }
126
127         *it = c;
128     }
129
130     fill(term.begin(), term.end(), term.background());   //Clear the screen.
131     s = "Etch-a-sketch: please type hjklq for left, down, up, right, quit.";
132     copy(s.begin(), s.end(), term.begin());
133
134     it = find(term.begin(), term.end(), 's');
135     if (it != term.end()) {
136         *it = 'S';
137     }
138
139     const terminal_t::keypad_t k = term.keypad();
140
141     for (it = center;;) {
142         *it = 'X';
143
144         char c;   //uninitialized variable
145         while ((c = term.key()) == '\0') {
146         }
147
148         if (c == 'q') {
149             break;
150         }
151
152         const terminal_t::keypad_t::const_iterator i = k.find(c);
153         //i->first is the char, i->second is its difference_type
154         if (i == k.end() || !it.in_range(i->second)) {
155             term.beep();
156         } else {
157             it += i->second;
158         }
159     }
160
161     term.beep();
162     term.wait(1000);
163 }
```

A terminal iterator is a random access iterator, so lines 60, 108, and 112 can apply a subscript to it. Line 60 could have been split into

```
164     *it = 'X';
165     it[1] = *it;
```

but don't do it. We are not wimps.

The three `terminal::put` functions in lines 22–24 of `terminal.h` on p. 160 have been abolished. To put a character on the screen, we will now write the above lines 65 or 60 (or 108 or 112). To put a string on the screen, we will call `copy` in line 79. We have regained the ability to display strings that we lost on p. 742.

The terminal::get function in line 25 of terminal.h on p. 160 has also been abolished. To get a character from the screen, we will now write line 60 (or 166).

Line 87 calls the constructor for class greater<terminal_t::value_type>, passing no arguments to it. This class is part of the STL; we saw its source code on p. 770. The newly constructed, anonymous object has an operator() member function that takes two terminal_t::value_type's and returns true if the first is greater than the second. sort will call this member function many times.

We could have dispensed with the anonymous object by defining the following function at line 13:

```
166 inline bool greater_printable(const printable_t& a, const printable_t& b) {
167     return a > b;
168 }
```

We could then change line 87 to

```
169     sort(it, it + s.size(), greater_printable);
```

The find in the above line 135 performs the following comparison.

```
    if (*it == 's') {    //it is a terminal_t::iterator
```

The expression *it is of type terminal_t::proxy. I wish the computer would convert the terminal_t::proxy into a printable_t, and thence into a char, to permit the comparison to 's'. But the language will not apply more than one user-defined implicit conversion to an expression (pp. 320–322). To get the comparison to compile, define the following functions in terminal.h. They are not member functions or friends of any class. Also define the five other pairs of comparison functions.

```
inline bool operator==(const printable_t& a, const printable_t& b) {
    return static_cast<char>(a) == static_cast<char>(b);
}

inline bool operator==(const wprintable_t& a, const wprintable_t& b) {
    return static_cast<wchar_t>(a) == static_cast<wchar_t>(b);
}
```

List of the six source files that constitute the terminal test program

(1) term.h and term.c (pp. 86–87). These are the only two written in C; the rest are in C++.

(2) except.h (pp. 628–629), which is included by printable.h.

(3) printable.h (pp. 749–753)

(4) terminal.h (pp. 969–977). There is no more terminal.C as of the following Homework.

(5) main.C (pp. 979–983)

▼ Homework 9.4b:
Version 5.0 of the Rabbit Game: port the game to the new class terminal

Uncouple the Cartesian coördinate system from the game by removing all mention of the two-dimensional Cartesian coördinates x, y. They should never have been there to begin with. Once the game has been purified, we will port the game effortlessly to a terminal of a totally different topology: three dimensions, one dimension, or two dimensions with polar coördinates; a cylinder, Möbius strip, or torus; a honeycomb of hexagonal cells instead of rows and columns of squares. (This will happen in a future homework, not this one.)

Make the following three changes in class game, class wabbit, and all of the classes derived from wabbit. Remember, if you mention the terminal_t member of class wabbit at any point at which you are not on a first-name basis with the members of that class, you will have to call it by its full name: wabbit::terminal_t.

(1) Change every pair of variables `unsigned x`, `unsigned y` to a single `terminal_t::iterator` object named `it`.

(2) Change every pair of variables `int dx`, `int dy`, and every `pair<int, int>` and `game::step_t`, to a single `terminal_t::difference_type` variable named `d`.

(3) Remove every pair of `int *dx`, `int *dy` that are function arguments. Replace them by a single `terminal_t::difference_type` as the function return value.

But do not make the above three changes in the following four places.

(1) Do not make the three changes in the C `term_` functions. C knows nothing of iterators or `difference_type`'s.

(2) Do not make the three changes in the new class `terminal`: that's the one place where the `x`'s and `y`'s can remain, as long as the user never sees them.

(3) Do not make the three changes to the `a` array in the constructor for class `game`. As long as we're using a two-dimensional terminal, the array will have to remain a rectangular picture and we'll still have to loop through it with a pair of `x`, `y` variables.

Follow these steps to perform the uncoupling.

(1) Remove the old class `terminal` with the `put` and `get` member functions. Replace it with the new class `terminal` that has the `iterator` and `difference_type` members. Make no change to the new class `terminal`.

(2) Now that we can easily `copy` a string to the terminal (lines 77−79 of `main.C` on p. 981), change the victory and defeat messages back to complete sentences.

(3) Change the `x`, `y` data members of class `wabbit` to a single `terminal_t::iterator` named `it`.

```
1 //Excerpt from wabbit.h
2
3 class wabbit {
4     game *const g;
5     terminal_t::iterator it;              //used to be unsigned x, y
6     const terminal_t::value_type c;       //already was a terminal_t::value_type
```

(4) In the constructors for class `wabbit` and its derived classes, change the two `initial_x` and `initial_y` arguments to a single argument of type `terminal_t::iterator`. We'll follow the STL convention of passing an iterator by value.

```
 7 wabbit::wabbit(
 8     game *initial_g,
 9     terminal_t::iterator initial_it,
10     terminal_t::value_type initial_c
11     ): g(initial_g), it(initial_it), c(initial_c)
12 {
```

In the template class `grandchild`, the `terminal_t` inherited from class `wabbit` via class `MOTION` will have to be written `typename MOTION::terminal_t`.

(5) Here's an excerpt from `game::game`. Line 15 is entirely new. Other new code is on the left; the old code is in the comments on the right (lines 16, 23, etc).

```
13     for (size_t y = ...
14         for (size_t x = ...
15             const terminal_t::iterator it(term, x, y);
16             if (it.in_range() && ...          //if (term.in_range(x, y) && ...
17                 const map_t::const_iterator i = m.find(a[y][x]);
18                 if (we didn't find the character a[y][x]) {
19                     ...
```

```
20                  }
21
22                  //Call the make_grandchild function for this species.
23                  i->second(this, it);         //i->second(this, x, y);
```

(6) Change the `x`, `y` arguments of `game::get` to a single `terminal_t::iterator it`, passed by value. `game::get` already has a local variable named `it` (which is a different type of iterator), so rename the local variable to `i`.

(7) The `decide` functions will now return one value instead of assigning a pair of values through a pair of pointers. Remove the two arguments of the `decide` functions. Change their return value from `void` to `terminal_t::difference_type`. For example, the entire body of `immobile::decide` will now be

```
24     return 0;                                      //used to be *dx = *dy = 0;
```

(8) The new version of `wabbit::move` is on the left; the old code is in the comments on the right. The new line 27 has only one variable `d`, and it's a `const`. There are no more uninitialized variables `dx` and `dy`. In the new line 31, we have to write only one comparison; in lines 35–36, we have to write only one addition. There are fewer function arguments in lines 38 and 43, and those in line 27 have disappeared entirely. Nowhere does the new code betray the number of dimensions in a `terminal_t`.

Our only regret is that line 46 can no longer use the elegant default value of the third argument of `terminal_t::put`. But `terminal_t::put` no longer exists.

The price we pay for this brave new code is that the names of the data types have become more complicated. For example, the plain old `int`'s in lines 27–28 have become a

```
const terminal_t::difference_type
```

where the `terminal_t` is itself a typedef for `terminal<printable_t>` and the `printable_t` is a typedef for `printable<char>`.

```
25 bool wabbit::move()
26 {
27  const terminal_t::difference_type d=decide();//int dx; //uninitialized variables
28                                               //int dy;
29                                               //decide(&dx, &dy);
30
31     if (d == 0) {                              //if (dx == 0 && dy == 0) {
32         return true;                           //    return true;
33     }                                          //}
34
35     const terminal_t::iterator newit = it + d;//const unsigned newx = x + dx;
36                                               //const unsigned newy = y + dy;
37
38     if (!newit.in_range()) {                   //if (!g->term.in_range(newx, newy)) {
39         punish();                              //    punish();
40         return true;                           //    return true;
41     }                                          //}
42
43     if (wabbit *const other = g->get(newit)) {//if(wabbit*const other=g->get(newx,newy)
44     //etc.                                     //etc.
45
46         *it = g->term.background();            //    g->term.put(x, y);
47         it = newit;                            //    x = newx;
48                                                //    y = newy;
49         *it = c;                               //    g->term.put(x, y, c);
```

See if you can figure out how `wabbit::move` can use the optional argument of `in_range` in line 142 of the above `terminal.h` to prevent the `wabbit` from wrapping around the left and right edges of the screen.

(9) Give class `wabbit` a public inline member function named `rand`, declared as

```
terminal_t::difference_type rand() const {body of function}
```

Like the `beep`, `key`, and `wait` member functions of class `wabbit`, `wabbit::rand` should do all its work simply by calling the corresponding member function of the terminal in the game that `g` points to. This member function is in line 340–342 of the new `terminal.h`.

The entire body of `brownian::decide` will now be

```
return rand();                                  //*dx = rand() % 3 - 1;
                                                //*dy = rand() % 3 - 1;
```

Since `brownian.h` no longer mentions the `rand` in the standard library, it will no longer need to include `cstdlib` or use namespace `std`.

(10) The keystrokes `hjkl`, for the directions left, down, up, right, work only for a two-dimensional, Cartesian terminal. A terminal with polar coördinates or three dimensions would require different keystrokes and directions. The keystrokes and directions have therefore been moved to the new class `terminal`, in line 345 of `terminal.h`.

An expression such as `step_t(1, 0)` in `game::claim` can be changed to `t->right()`.

(11) Our `dist` function contains a Pythagorean distance formula that works for only a two-dimensional, Cartesian terminal. It has therefore been moved to the new class `terminal`, in line 163 of `terminal.h`. Remove the `dist` function in `visionary.C` that you wrote for the first `visionary` homework, and replace it with the following inline friend of class `wabbit`.

```
friend double dist(const wabbit *w1, const wabbit *w2) {
    return dist(w1->it, w2->it);
}
```

(12) Similarly, our `step` function applies only to a two-dimensional, Cartesian terminal. It has therefore been moved to the new class `terminal`, in line 180 `terminal.h`. `step` now returns a `terminal_t::difference_type`. Remove the `step` function in `visionary.C` that you wrote for the first `visionary` Homework, and replace it with the following inline friend of class `wabbit`.

```
friend terminal_t::difference_type step(const wabbit *w1,
                                        const wabbit *w2) {
    return step(w1->it, w2->it);
}
```

(13) Our `difference` function works only for a two-dimensional, Cartesian terminal. It has therefore been moved to the new class `terminal`, in line of `terminal.h`. Remove the `difference` friend of class `wabbit` that you wrote for the first `visionary` homework.

Our original `signum` function in `visionary.C` took a generic `int`, and was not a member function or friend of any class. The new `signum` function in line 173 of `terminal.h` takes a `terminal<CHAR>::difference_type`. Since the data type of its argument was specific to class `termianl<CHAR>`, I made it a friend of that class. Remove the original `signum` in `visionary.C`.

(14) The `radius` of vision is unsigned, so it should become a `terminal_t::size_type`.

▲

How we could have managed the transition to iterators and difference_type's

(1) The previous Homework could have begun by changing the name of class `terminal` to `oldterminal`:

```
template <class CHAR = char>
```

```
class oldterminal {
    declarations for private members;
public:
    declarations for the public members:
    oldterminal, ~oldterminal
    background
    xmax, ymax
    get, put, put, put
    key, wait, beep,
    in_range, next
    distance, step,
    keypad_t, keypad
};
```

(2) Then we could have publicly derived a `terminal` class containing the new public members, and call-throughs for the old member functions that we want to keep.

```
template <class CHAR = char>
class terminal: public oldterminal<CHAR> {
public:
    terminal(CHAR initial_background): oldterminal<CHAR>(initial_background) {}

    //Seven new members:

    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef CHAR value_type;
    typedef map<char, difference_type> keypad_t;

    class iterator {declarations for members and friends};
    class proxy    {declarations for members and friends};
    keypad_t keypad() const;

    //Six old members that we want to keep permanently:

    CHAR background() const {return oldterminal<CHAR>::background();}
    unsigned xmax() const {return oldterminal<CHAR>::xmax();}
    unsigned ymax() const {return oldterminal<CHAR>::ymax();}

    char key() const {return oldterminal<CHAR>::key();}
    void wait() const {oldterminal<CHAR>::wait();}
    void beep() const {oldterminal<CHAR>::beep();}

    //Old members that we want to keep only during the transition period:

    bool in_range(int x, int y, int dx = 0, int dy = 0) const {
        return oldterminal<CHAR>::in_range(x, y, dx, dy);
    }

    double distance(int x1, int y1, int x2, int y2) const {
        return oldterminal<CHAR>::distance(x1, y1, x2, y2);
    }

    void step(int x1, int y1, int x2, int y2, int *dx, int *dy) const {
        oldterminal<CHAR>::step(x1, y1, x2, y2, dx, dy);
```

```
53     }
54 };
```

Because the inheritance in the above line 16 was public, we can employ either style:

```
55     terminal<printable_t> term('.');
56 
57     //Old style.
58     //x, y, x1, y1, x2, y2 are unsigned's, s is a string.
59 
60     term.put(x, y, 'A');
61     term.put(x, y, s);
62     double dist = term.difference(x1, y1, x2, y2);
63 
64     int dx, dy;                      //uninitialized variables
65     term.step(x1, y1, x2, y2, &dx, &dy);
66 
67     //New style.
68     //it, it1, it2 are terminal<printable_t>::iterator's, s is a string.
69 
70     *it = 'A';
71     copy(s.begin(), s.end(), it);
72     double d = difference(it1, it2);
73     terminal<printable_t>::difference_type d = term.step(it1, it2);
```

(3) After the transition period, we could have disallowed the above lines 57−65 by changing the `public` to `private` in the above line 16 and removing lines 41−53. The new class `terminal` will be a *container adaptor* providing access to part of the functionality of the underlying `oldterminal`. See p. 935 for other examples.

9.5 Alternative Traversals

Traverse the same container in two different orders

In a container, the order of the elements is not a property of the elements or of the container. An order is imposed on the elements by the iterator that traverses them. A different iterator can impose a different order. We already saw that every container with bidirectional iterators can have a reverse iterator (pp. 856−858). Here is another example.

In Western music the interval between two consecutive notes is called a *half step.* An *octave* is composed of twelve half steps. In the following program the first iterator traverses the notes of an octave in order of increasing pitch. In this order, C and C# are right next to each other: they are consecutive keys on a piano keyboard.

But in the chord progressions of most pieces of music, C and C# are very remote from each other. A C chord is more likely to be followed by a G than a C#; a song that starts in the key of C would be more likely to switch to G than to C#. We say that C and G are consecutive keys in a different ordering, called the *circle of fifths.* The key signatures also follow the circle of fifths: the key of G has one sharp, D has two sharps, A has three sharps, etc. The iterator in line 28 of `main.C` traverses the notes along the circle of fifths.

The `operator*` in line 63 constructs and returns a `note` object. It must therefore return by value, not by reference. Ditto for the `operator[]` in line 64. `operator[]` calls `operator+` (line 132), so `operator[]` must be defined after this `operator+` is declared (lines 125−126). I didn't bother to define postfix operations for the iterator. And comparisons with < or > are meaningless: since the container is circular, any such comparison would return true.

Five of the notes have alternative names (lines 22–27), so we provide the i/o manipulators `sharp` and `flat` analogous to the `cartesian` and `polar` in pp. 362–366. Columns 2 and 3 of the output exercise them.

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

```
 1 #ifndef NOTEH
 2 #define NOTEH
 3 #include <iostream>
 4 #include <iterator>
 5 #include <cassert>
 6 using namespace std;
 7
 8 ostream& sharp(ostream& os);
 9 ostream& flat(ostream& os);
10
11 class note {
12     static const int subscript;  //subscript of new element in ios_base::iword
13
14     unsigned i;                  //number of half steps above C
15                                  //in range 0 to n - 1 inclusive
16 public:
17     static const unsigned n = 12; //number of half steps in an octave
18     /*
19     The value of each enum is its number of half steps above C.
20     The five sharps are the black keys on the piano: a pair and a trio.
21     */
22     enum {
23         C, C_SHARP, D, D_SHARP, E, F, F_SHARP, G, G_SHARP, A, A_SHARP, B,
24
25         //alternative names for the five black keys
26         D_FLAT = C_SHARP,
27         E_FLAT = D_SHARP,
28         G_FLAT = F_SHARP,
29         A_FLAT = G_SHARP,
30         B_FLAT = A_SHARP
31     };
32
33     note(unsigned initial_i): i(initial_i) {assert(i < n);}
34
35     friend ostream& operator<<(ostream& os, const note& no);
36
37     friend ostream& sharp(ostream& os) {
38         os.iword(note::subscript) = 0;   //the default
39         return os;
40     }
41
42     friend ostream& flat(ostream& os) {
43         os.iword(note::subscript) = 1;
44         return os;
45     }
46
47     class const_iterator:
48         public std::iterator<random_access_iterator_tag, note, unsigned,
49             const note *, const note&> {
```

```

        unsigned i;   //number of half steps above C
        const unsigned stride;

        static difference_type distance(const const_iterator& it1,
                                        const const_iterator& i2);
    public:
        const_iterator(unsigned initial_i, unsigned initial_stride = 1)
            : i(initial_i), stride(initial_stride) {
            assert(initial_i < n);
            assert(0 < stride && stride < n);
        }

        const note operator*() const {return i;}
        const note operator[](int i) const;

        const_iterator& operator=(int j) {
            assert(0 <= j && j < n);
            i = j;
            return *this;
        }

        const_iterator& operator+=(int j) {
            i += j * stride;
            i %= n;
            return *this;
        }

        const_iterator& operator-=(int j) {
            i -= j * stride;
            i %= n;
            return *this;
        }

        const_iterator& operator++() {return *this += 1;}
        const_iterator& operator--() {return *this -= 1;}

        friend difference_type operator-(const const_iterator& it1,
                                         const const_iterator& it2);

        friend bool operator==(const const_iterator& it1,
                               const const_iterator& it2) {
            return it1.i == it2.i;
        }

        friend bool operator>(const const_iterator& it1,
                              const const_iterator& it2) {
            return distance(it1, it2) < n;
        }

        friend bool operator>=(const const_iterator& it1,
                               const const_iterator& it2) {
            return it1 == it2 || it1 > it2;
        }
```

```
    };
};

inline bool
operator!=(const note::const_iterator& it1, const note::const_iterator& it2) {
    return !(it1 == it2);   //return !operator==(it1, it2);
}

inline bool
operator<=(const note::const_iterator& it1, const note::const_iterator& it2) {
    return it2 >= it1;
}

inline bool
operator<(const note::const_iterator& it1, const note::const_iterator& it2) {
    return it2 > it1;
}

inline const note::const_iterator
operator-(note::const_iterator it, int i) {return it -= i;}

inline const note::const_iterator
operator+(note::const_iterator it, int i) {return it += i;}

inline const note::const_iterator
operator+(int i, note::const_iterator it) {return it += i;}

inline const note
note::const_iterator::operator[](int i) const {return *(*this + i);}
#endif
```

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

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

const int note::subscript = ios_base::xalloc();

//Return the distance from it2 to it1 (the integer you'd have to add to it2 to
//get to it1, or note::n if there is no such integer.

note::const_iterator::difference_type
note::const_iterator::distance(const const_iterator& it1,
                               const const_iterator& it2)
{
    assert(it1.stride == it2.stride);
    difference_type d = 0;

    for (const_iterator it = it2; it != it1 && d < note::n; ++it, ++d) {
    }

    return d;
```

```
}

note::const_iterator::difference_type operator-(const note::const_iterator& it1,
                                                 const note::const_iterator& it2)
{
    const note::const_iterator::difference_type d =
        note::const_iterator::distance(it1, it2);

    if (d >= note::n) {
        cerr << *it1 << " is inaccessible from " << *it2
            << " via a stride of " << it1.stride << ".\n";
        exit(EXIT_FAILURE);
    }

    return d;
}

ostream& operator<<(ostream& os, const note& no)
{
    static const char *const a[][12] = {
        {"C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B"},
        {"C", "Db", "D", "Eb", "E", "F", "Gb", "G", "Ab", "A", "Bb", "B"}
    };

    assert(no.i < note::n);
    const int s = os.iword(note::subscript);
    assert(0 <= s && s < 2);
    return os << a[s][no.i];
}
```

To begin and end each scale on the same note, `note::C`, I would like to loop 13 times. I wish I could use the following call to `copy`, but it would loop only one time and then halt.

```
    copy(note::const_iterator(note::C),
         note::const_iterator(note::C) + 1, os);
```

The algorithms were not designed for circular containers. See pp. 999–1000 for another example.

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

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

template <class INPUT, class SIZE_TYPE, class OUTPUT>
void my_copy_n(INPUT in, SIZE_TYPE s, OUTPUT out)
{
    for (; s > 0; --s) {
        *out = *in;
        ++in;
        ++out;
    }
}

int main()
```

```
{
    ostream_iterator<note> os(cout, "\n");

    my_copy_n(note::const_iterator(note::C), note::n + 1, os);
    cout << "\n";

    cout << flat;
    my_copy_n(note::const_iterator(note::C), note::n + 1, os);
    cout << "\n";

    //Traverse the Circle of Fifths.
    my_copy_n(note::const_iterator(note::C, 7), note::n + 1, os);
    cout << sharp;

    return EXIT_SUCCESS;
}
```

Output printed in three columns to save space.

```
C                        C                        C
C#                       Db                       G
D                        D                        D
D#                       Eb                       A
E                        E                        E
F                        F                        B
F#                       Gb                       Gb
G                        G                        Db
G#                       Ab                       Ab
A                        A                        Eb
A#                       Bb                       Bb
B                        B                        F
C                        C                        C
```

▼ Homework 9.5a: spiral iterator

Write a `spiral_iterator` for class `terminal`.

```
typedef terminal<printable> terminal_t;
terminal_t term('.');

string s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
           "abcdefghijklmnopqrstuvwxyz";

const terminal_t::iterator center(term, term.xmax() / 2,
                                        term.ymax() / 2);

copy(s.begin(), s.end(), terminal_t::spiral_iterator(center));
```

	k	j	i	h	g	f	e	
	l	Q	P	O	N	M	d	
	m	R	E	D	C	L	c	
	n	S	F	A	B	K	b	
	o	T	G	H	I	J	a	z
	p	U	V	W	X	Y	Z	y
	q	r	s	t	u	v	w	x

Give it a two-argument constructor too.

```
terminal_t::spiral_iterator it(term.xmax() / 2, term.ymax() / 2);
```

The arguments of the two-argument constructor do not necessarily have to be on the screen. For example, the following iterator would sweep the screen the way we read a printed page, left to right and top to bottom.

```
terminal_t::spiral_iterator it(term.xmax() / 2, -term.xmax());
```

In the constructor for `wabbit`, we currently look only in one direction for an empty location.

```
while (*it != g->term.background()) {
    ++it;
}

```

Change this to

```
game::terminal_t::spiral_iterator s = it;
while (*s != g->term.background()) {
    ++s;
}
it = s;
```

Of course, we should also have a test for end-of-screen.

A spiral iterator would radically simplify `visionary::decide`. *The "closest animal" code in visionary::decide really belongs in spiral_iterator::operator++.*

▲

9.6 Port the Game to Terminal with a Different Topology

▼ **Homework 9.6a:**

Run the server for the ring terminal

The ring terminal is a Java applet. The Java compiler `javac` created the two `.class` files.

```
1$ cd $dir/term0
2$ /bin/javac Term0.java
3$ chmod 444 Term0.class ReadFromClient.class
```

When two programs have a conversation carried by TCP, the one that initiates the conversation is called the *client* and the other program is called the *server*. We say that the client "connects" to the server, and the server "accepts" the client. On the other hand, the server begins running before the client. In fact, most servers run 24 hours per day.

The Java applet is a TCP server; our C++ program will be its client. Since the applet is a TCP server, we'll have to give it permission to do things that it would not normally be allowed to do in its "sandbox". In Internet Explorer on Windows,

(1) Pull down the Tools menu and select `Internet Options...`. An `Internet Options` window will appear.

(2) Click on the `Security` tab.

(3) Select the `Internet` content zone. It has a picture of a globe.

(4) Press the `Custom Level...` button. A `Security Settings` window will appear.

(5) Scroll down to `Microsoft VM` and click on it if necessary to pop up the `Java Permissions` under it. Press the `Custom` radio button.

(6) Press the `Java Custom Settings...` button. An `Internet` window will appear.

(7) Click on the `Edit Permissions` tab.

(8) Double-click on the `Unsigned Content` padlock if necessary to make the `Run Unsigned Content` padlock appear.

(9) Double-click on the `Run Unsigned Content` padlock if necessary to make the three `Run in Sandbox`, `Disable`, `Enable` radio buttons appear.

(10) Press the `Enable` radio button.

(11) Press the `OK` button to dismiss the `Internet` window.

(12) Press the `OK` button to dismiss the `Security Settings` window.

(13) If it says "Are you sure you want to change the security settings for this zone?", click `Yes`.

(14) Click the `OK` button to dismiss the `Internet Options` window.

On my iMac,

```
Apple menu ->
System Preferences... ->
Network ->
Proxies
```

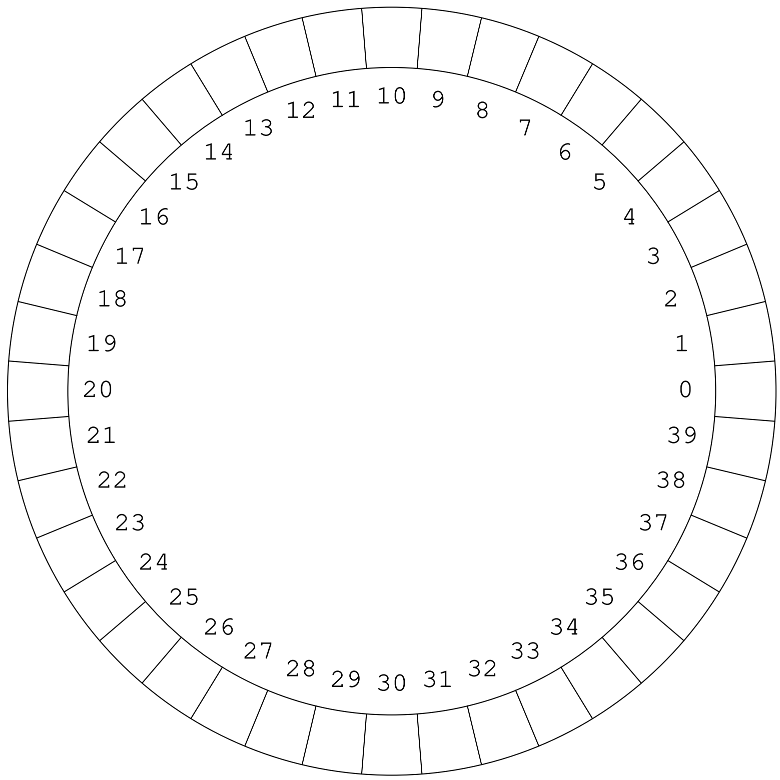
Then I checked the first three (`FTP`, `Web`, and `Secure Web`), and unchecked the last three (`Streaming`, `Gopher`, and `SOCKS Firewall`).

To launch the server, point your web browser at

```
http://i5.nyu.edu/~mm64/INFO1-CE9266/term0/
```

There is a tilde in front of the `mm64`. `term0` has a zero, not an uppercase letter `O`.

For debugging, the terminal lists the ordinate of each character and the IP address and port number of the host.




```
Listening on port 9266 of 192.168.2.10...
```

To verify that the server was in the `LISTEN` state, I launched the `Terminal` application on the Mac where I was running the browser. This gave me a Unix shell window in which I checked the network status.

```
netstat -a -f inet -p TCP | awk 'NR <= 2 || /\.9266/'
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  *.9266                 *.*                    LISTEN
```

You can select `View Source` from the browser's menu bar to examine the `PARAM` tags named `port` and `xmax`:

```
7$ cat -n /home1/m/mm64/public_html/INFO1-CE9266/term0/index.html |
awk '8 <= NR && NR <= 13'
 8  <APPLET CODE = "Term0.class"
 9  WIDTH = 500 HEIGHT = 500>
10  <PARAM NAME = "port" VALUE = 9266>
11  <PARAM NAME = "xmax" VALUE = 40>
12  This browser does not understand the APPLET tags or does not have Java enabled.
13  </APPLET>
```

The value of `port` must be in the range 0–65535, because a port number is two bytes in the TCP protocol. The value of `xmax` must be in the range 1–255, because it must fit into a single byte of X52.9266 protocol.

The web page also has a link to the Java source code of the server, consisting of two classes named `Term0` and `ReadFromClient`.

We must launch the server before the client. Unlike most servers, this one will accept only one client. We must therefore launch the server again every time we run our client. To re-launch the server, press the browser's "Refresh" button while holding down the "Ctrl" key. (Without the control key, we would be refreshing the HTML page but not the applet in it.)

To re-launch the server immediately, we would have to bind it to a different TCP port number each time. (There is a two-minute waiting period before a TCP port number ceases to be in use; look up the `TIME_WAIT` state in a TCP/IP book.) If you don't want to wait two minutes, change the port number by changing the `PARAM` tag for `port`, and line 11 of the following `main.C`.

Find your IP address

(1) To find the IP address of a Macintosh host running OSX,

```
Apple Menu → System Preferences... → Network → TCP/IP
```

(2) To find the IP address of a Windows host,

```
Start → Programs → Accessories → Command Prompt
ipconfig        for Windows 2000 or NT
winipcfg        for Windows 95 or 98
```

(3) To find the IP address(es) of a Unix host,

```
1$ ifconfig -a | more
```

The client

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/term0/term0.h`

```
#ifndef TERM0H   /* This file can be #include'd in either C or C++. */
#define TERM0H

/* These two functions must be called in pairs. */
void term0_construct(const char *ip, unsigned short port);
void term0_destruct(void);

/* Legal x values go from 0 to term0_xmax() - 1 inclusive. */
unsigned term0_xmax(void);

/* Display a character or string on the screen. */
void term0_put (unsigned x, char c);
void term0_puts(unsigned x, const char *s);

/* Return the character at the given position on the screen. */
char term0_get(unsigned x);

/* Return the key the user pressed.  If no key was pressed, return '\0'
immediately. */
char term0_key(void);

void term0_wait(int milliseconds);    /* 1000 milliseconds == 1 second */
void term0_beep(void);
#endif
```

Line 10 must have the IP address and TCP port number of the server. Line 20 can apply the `%=` operator to the expression `++x` because the prefix `++` returns an lvalue (pp. 12–13).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/term0/main.C`

```
#include <iostream>
#include <cstdlib>
extern "C" {
#include "term0.h"
}
using namespace std;

int main()
{
    term0_construct("192.168.66.96", 9266);
    const unsigned xmax = term0_xmax();
    unsigned x = 0;

    term0_put(x, 'X');
    char c = term0_get(x);
    term0_put(x + 1, c);

    term0_puts(2, "Type printable chars ending w/ q.");

    for (x = 0; x < xmax; ++x %= xmax) {   //x = x + 1, x = x % xmax
        while ((c = term0_key()) == '\0') {
        }

        if (c == 'q') {   //quit
            goto done;
        }

        term0_put(x, c);
    }

    done:;
    term0_wait(3000);    //three seconds
    term0_beep();
    term0_destruct();
    return EXIT_SUCCESS;
}
```

List of the three source files that constitute the client

(1) `term0.h` and `term0.c`. These are the only two written in C; the other is in C++.

(2) `main.C`

Compile the client under Unix

The minus lowercase L's stand for "library". `nsl` is the "Network Services Library".

```
1$ gcc -c term0.c
2$ g++ -o ~/bin/tester main.C term0.o -lcurses -lsocket -lnsl
3$ ls -l ~/bin/tester
```

Run the client

After the server has started listening, you can run the client. The client begins by outputting the server's `xmax` to confirm that it has established a connection to the server.

```
1$ tester
Trying 192.168.20.196...
Connected to 192.168.20.196.
term0 xmax == 40
```

Meanwhile, the server will display `Accepted client` and the client's IP address. The server should now be in the `ESTABLISHED` state.

```
netstat -a -f inet -p TCP | awk 'NR <= 2 || /\.9266/'
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  ftcg5faculty2.ed.9266  i5.nyu.edu.46053       ESTABLISHED
```

The client should also be in the `ESTABLISHED` state. On i5.nyu.edu (Solaris), the `-P` option of `netstat` is uppercase, the `tcp` argument is lowercase, and the header is the first four lines of output.

```
netstat -a -f inet -P tcp | awk 'NR <= 4 || /\.9266/'

TCP: IPv4
   Local Address        Remote Address    Swind Send-Q Rwind Recv-Q  State
-------------------- -------------------- ----- ------ ----- ------ -------
i5.46053 FTCG5FACULTY2.EDLAB.ITS.NYU.EDU.9266 65535  0 49640      0 ESTABLISHED
```

You may now have to "wake up" the server by clicking on its window or by hiding the other windows. You may even have to hide the server's window and pop it up again.

The characters you type will be displayed counterclockwise around the ring. When the client disconnects from the server by calling `term0_destruct`, this function will output

```
Connection closed.
2$ echo $?                              See the client's exit status.
0                                       The client's exit status should be 0.
```

Meanwhile, the server will display `Connection closed by client`. Wait two minutes before launching the server again.

The X52.9266 protocol

The server and client obey a set of rules called the X52.9266 protocol. This protocol is carried by TCP; the server is bound to port 9266. After accepting the client, the server sends one byte to the client giving the value of `xmax`. (This means that `xmax` must be less than 256.) In the above picture, `xmax` is 40.

After receiving the above byte, the client sends pairs of bytes to the server. The first byte in each pair is an `x` coördinate. The second byte is the character to be displayed at that coördinate. If the second byte is `00000111`, the server will emit a beep instead of displaying a character. In this case, the first byte will be ignored.

Meanwhile, the server will send each keystroke to the client as a separate byte.

The conversation can be terminated by either the client or the server. Calling the `destroy` method of the server will close the connection, but we have no way of predicting when the browser will do this.

▼ Homework 9.6b:

Class `terminal` saved us the trouble of calling the `term_` C functions directly. Create a class `terminal0` to save us the trouble of calling the `term0_` functions directly. I invite you to copy as much as possible from the class `terminal` on pp. 969–977.

The member functions `key` and `beep` of class `terminal0` will no longer be static. Perhaps we should throw in the towel and make `wait` non-static too.

Class `terminal0` will be a template class. Also create the four data types that the user will be aware of,

`terminal0::size_type`
`terminal0::difference_type`
`terminal0::value_type`
`terminal0::iterator`, *derived from* `std::iterator<random_access_iterator_tag, T>`

and the data type that the user will be unaware of.

`terminal0::proxy`

Write a test program for class `terminal0`. I invite you to copy as much as possible from the test program for class `terminal` on pp. 979–983.
▲

▼ Homework 9.6c:
Version 5.1 of the Rabbit Game: port the game to the ring terminal

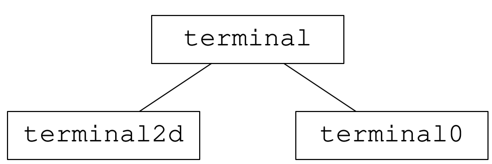
The holy grail

A `game` and its `wabbit`'s are currently hardwired to occupy a `terminal<CHAR>`. We can templatized classes `game` and `wabbit` to inhabit any container whose iterators are bidirectional:

```
1 game<terminal<printable_t> > game1('.');
2 game<terminal<char> > game2('.');
3 game<terminal<> > game3('.');       //same as line 2
4 game<> game4('.');                  //same as line 2
5
6 //w isi "wide", L is "long".
7 game<terminal<wchar_t> > game5(L'.');
8
9 game<terminal0<printable_t> > game6("192.168.20.196", 9266, '.');
```

Derive the two terminal classes from an abstract base class.

The following diagram shows only the inheritance relationships. No attempt was made to show the data types that are plugged in as template arguments (e.g., class `printable_t`). The two-dimensional class `terminal` has been renamed `terminal2d`.




```
template <class CHAR = char, class DIFF = ptrdiff_t>
class terminal {
public:
    virtual void beep() const = 0;
    virtual CHAR background() const = 0;
    virtual DIFF rand() const = 0;

    typedef map<char, DIFF> keypad_t;
    virtual keypad_t keypad() const = 0;
```

```
};

template <class CHAR>
class terminal2d: public terminal<CHAR> {
    //declarations for members
};

template <class CHAR>
class terminal0: public terminal<CHAR> {
    //declarations for members
};
```

9.7 Avoid a Fat Interface with Virtual Base Classes

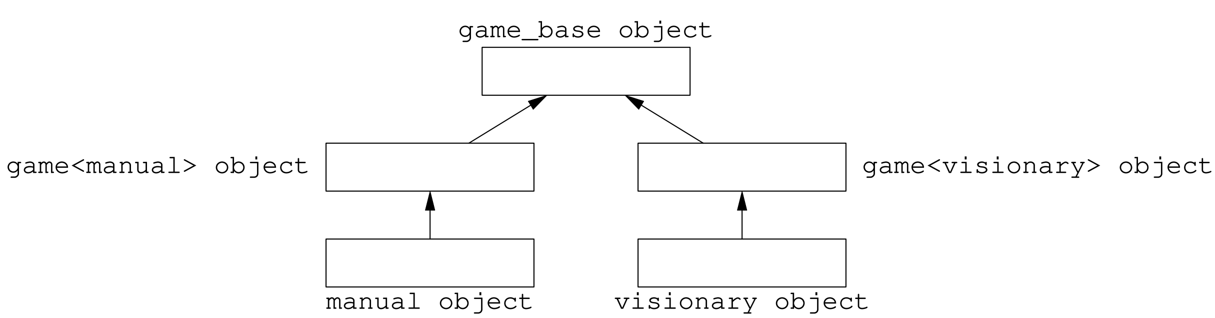
Class wabbit contains all the members and friends needed for classes manual and visionary. These members are present even if there are no animals of these classes.

```
class wabbit {
    //For original (pre-map) class manual, whose needs were modest.

    char key() const {return g->term.key();}
    void beep() const {g->term.beep();}

    //For original (pre-difference_type) class visionary

    typedef game::master_t::const_iterator const_iterator;
    const_iterator begin() const {return g->master.begin();}
    const_iterator   end() const {return g->master.end();}

    friend void difference(const wabbit *w1, const wabbit *w2,
        int *dx, int *dy);
};
```

How can we avoid loading the base class wabbit with special-purpose features for individual derived classes?

Why are all of these functions up in class wabbit anyway? Well, they have to be there because the g pointer is a private data member of class wabbit. This makes it impossible for the derived classes to get services from the game without going through class wabbit.

Perhaps g should be somewhere else. Is there a way the derived classes can get these services directly from the game? If so, can we avoid loading class game with special-purpose features? Let's move g. What design pattern is this?



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

```
class game_base {
protected:
    typedef terminal<printable_t> terminal_t;
    const terminal_t term;

    typedef list<wabbit *> master_t;
    master_t master;

public:
    game(const terminal_t::value_type& c = ’.’);
};

template <class T>
class game: public virtual game_base {
public:
    game(const terminal_t::value_type& c): game_base(c) {}
};
```

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

```
class wabbit {
private:
    typedef game::terminal_t terminal_t;
    terminal_t::iterator it;
    terminal_t::value_type c;
    //no longer has a game *

    virtual game *get_game() const = 0;
public:
    wabbit(arguments for constructor) {game_get()->push_back(this);}
    //etc.
};
```

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

```
class manual;   //forward declaration

template <>      //specialization of a template class
class game<manual>: public virtual game_base {
    game(const terminal_t::value_type& c = ’.’): game_base(c) {}

    friend class manual;
    char key() const {return term.key();}
    void beep() const {term.beep();}
};

class manual: public virtual wabbit {
    game<manual> *const g;
    game *get_game() const {return g;}

    void punish() const {g->beep();}
```

```
    terminal_t::difference_type decide() const {g->key();}
};
```

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

```
class visionary;   //forward declaration

template <>      //specialization of a template class
class game<visionary>: public virtual game_base {
    game(const terminal_t::value_type& c = ’.’): game_base(c) {}

    friend class visionary;
    typedef master_t::const_iterator const_iterator;
    const_iterator begin() const {return master.begin();}
    const_iterator   end() const {return master.end();}

    //a friend of class wabbit
    void difference(const wabbit *w1, const wabbit *w2, int *dx, int *dy);
};

class visionary: public virtual wabbit {
    game<visionary> *const g;
    game *get_game() const {return g;}

    terminal_t::difference_type decide() const {g->begin();}
};
```

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

```
class great_game: public game<manual>, public game<visionary> {
    great_game(const terminal_t::value_type& c = ’.’)
        : game_base(c), game<manual>(c), game<visionary>(c) {}
};

int main()
{
    great_game g;
    return EXIT_SUCCESS;
}
```

9.8 The Evolution of a Member: a Diachronic Flipbook

The evolution of a member function

(1) The original version of `terminal::put` lines 36–47 of `termial.C` on p. 161 was

```
void terminal::put(unsigned x, unsigned y, char c)
{
    if (isprint(static_cast<unsigned char>(c)) == 0) {
        cerr << "unprintable character "
            << static_cast<unsigned>(c) << ".\n";
        exit(EXIT_FAILURE);
    }
```

```
 8
 9     check(x, y); //error checking for x, y
10     term_put(x, y, c);
11 }
```

(2) After introducing exceptions, the `cerr` and `exit` in the above lines 4–6 went somewhere else, to the `print` member function of class `unprintable`.

```
12 void terminal::put(unsigned x, unsigned y, char c) throw (unprintable)
13 {
14     if (isprint(static_cast<unsigned char>(c)) == 0) {
15         ostringstream ost;
16         ost << "unprintable character "
17             << static_cast<unsigned>(static_cast<unsigned char<(c))
18             << " at location (" << x << ", " << y << ")";
19         throw except(ost);
20     }
21
22     check(x, y);
23     term_put(x, y, c);
24 }
```

(3) After introducing templates, the `if` statement in the above line 14 went somewhere else, to a constructor for class `CHAR`. The argument `c` was passed by value, and any error checking was performed by a constructor for class `CHAR`, not by `terminal::get`.

```
25 template <class CHAR>
26 void terminal<CHAR>::put(unsigned x, unsigned y, CHAR c)
27 {
28     check(x, y);
29     term_put(x, y, c);
30 }
```

(4) Finally, the member function `terminal::put` was abolished. Instead of calling this function, we now write an expression such as

```
31     *it = 'A';   //it.operator*().operator=('A');
```

which calls these three functions:

```
32 template <class CHAR>
33 const element iterator::operator*() const
34 {
35     return proxy(*this);   //call constructor for class proxy, shown below
36 }

37 template <class CHAR>
38 proxy::proxy(const iterator& initial_it): it(initial_it)
39 {
40     throw exception if it is off the screen;
41 }

42 //Constructor for c does error checking for c, if any.
43 template <class CHAR>
44 proxy& proxy::operator=(CHAR c) const
45 {
46     term_put(it.x(), it.y(), c);
47     return *this;
```

```
}
```

The evolution of the data members of class rabbit

(1) In the original class `rabbit`, every `rabbit` contained a pointer to a `terminal` object that was not a member of any class.

```
1 const terminal term('.');
2
3 class rabbit {
4     const terminal *t;        //read-only pointer to the terminal in line 1
5     unsigned x, y;
6     char c;
7 };
```

(2) For a while, the data member `c` became static. The data members `t` and `c` became `const`.

```
 8 const terminal term('.');
 9
10 class rabbit {
11     const terminal *const t;
12     unsigned x, y;
13     static const char c;
14 };
```

(3) When the `rabbit`'s shared a `list` as well as a `terminal`, we grouped the two shared variables into a `game` object. The `t` data member in the above line 11 became a read/write pointer `g` to a `game` in line 21. (Read/write, so a `rabbit` could put itself on, and take itself off, the master list.

```
15 class game {
16     const terminal term;
17     list<rabbit *> master;
18 };
19
20 class rabbit {
21     game *const g;
22     unsigned x, y;
23     static const char c;
24 };
```

(4) When we introduced single inheritance, the data members `g`, `x`, `y`, and `c` moved from class `rabbit` to class `wabbit`. ("Data members follow code.") The master list changed from a `list<rabbit *>` to a `list<wabbit *>`. `c` became non-static again, now that it the same `c` served different species.


```
class game {
    const terminal term;
    list<wabbit *> master;
};

class wabbit {
    game *const g;
    unsigned x, y;
    const char c;              //non-static again
};

class rabbit: public wabbit {
    //no longer has data members of its own
};
```

(5) With multiple inheritance, class `rabbit` changed from a child of `wabbit` to a grandchild. Then the inheritance changed from public to private or protected.

```
class game {
    const terminal term;
    list<wabbit *> master;
};

class wabbit {
    game *const g;
    unsigned x, y;
    const char c;
};

class brownian: protected virtual wabbit {
    //no data members of its own
};

class victim: private virtual wabbit {
    //no data members of its own
};

class rabbit: private brownian, private victim {
    //no data members of its own
};
```

(6) Classes `victim` and `rabbit` became instantiations of template classes.

```
class game {
    const terminal term;
    list<wabbit *> master;
};

class wabbit {
    game *const g;
    unsigned x, y;
    const char c;
};

class brownian: protected virtual wabbit {
```

```
 73     //no data members of its own
 74 };
 75
 76 template <int HUNGRY, int BITTER>
 77 class rank: private virtual wabbit {
 78     //no data members of its own
 79 };
 80
 81 typedef rank<INT_MIN, INT_MIN> victim_t;
 82
 83 template <class MOTION, class RANK, char C>
 84 class grandchild: private MOTION, private RANK {
 85     //no data members of its own
 86 };
 87
 88 typedef grandchild<brownian, victim_t, 'r'> rabbit_t;
```

(7) Class `terminal` became a template class, with a public member named `value_type`. The `const char c` in the above line 69 changed to `terminal_t::value_type` in line 105.

```
 89 template <class CHAR>
 90 class terminal {
 91 public:
 92     typedef CHAR value_type;
 93 };
 94
 95 typedef terminal<printable_t> terminal_t;
 96
 97 class game {
 98     const terminal_t term;
 99     list<wabbit *> master;
100 };
101
102 class wabbit {
103     game *const g;
104     unsigned x, y;
105     const terminal_t::value_type c;
106 };
107
108 class brownian: protected virtual wabbit {
109     //no data members of its own
110 };
111
112 template <int HUNGRY, int BITTER>
113 class rank: private virtual wabbit {
114     //no data members of its own
115 };
116
117 typedef rank<INT_MIN, INT_MIN> victim_t;
118
119 template <class MOTION, class RANK, char C>
120 class grandchild: private MOTION, private RANK {
121     //no data members of its own
122 };
123
```

```
124 typedef grandchild<brownian, victim_t, 'r'> rabbit_t;
```

(8) Class `terminal` became an STL-compliant container, with public members named `iterator`, `size_type`, and `difference_type`. The `unsigned` `x` and `y` in the above line 104 became the `terminal_t::iterator` in line 147.

```
125 template <class CHAR>
126 class terminal {
127 public:
128     typedef CHAR value_type;
129     typedef size_t size_type;
130     typedef ptrdiff_t difference_type;
131
132     class iterator {
133         size_type i;                //takes the place of x and y
134     };
135 };
136
137 typedef terminal<printable_t> terminal_t;
138
139 class game {
140     const terminal_t term;
141     list<wabbit *> master;
142     map<terminal_t::value_type, master_t::size_type> count;
143 };
144
145 class wabbit {
146     game *const g;
147     terminal_t::iterator it;
148     const terminal_t::value_type c;
149 };
150
151 class brownian: protected virtual wabbit {
152     //no data members of its own
153 };
154
155 template <int HUNGRY, int BITTER>
156 class rank: private virtual wabbit {
157     //no data members of its own
158 };
159
160 typedef rank<INT_MIN, INT_MIN> victim_t;
161
162 template <class MOTION, class RANK, char C>
163 class grandchild: private MOTION, private RANK {
164     //no data members of its own
165 };
166
167 typedef grandchild<brownian, victim_t, 'r'> rabbit_t;
```

9.9 Move the Complexity into the Data Types

Here are declarations for typical variables in C and C++ respectively. They are exaggerated, but only slightly:

```
 1     int i;                          /* C */
 2
 3     map<pair<unsigned, unsigned>, wabbit *>::const_iterator it = m.begin(); //C++
```

The data types in C are generic. For example, the above `i` could be used as a counter for any loop or as a subscript for any array. The data types in C++ are specialized. For example, the `it` can be used only for a `list<wabbit *>`.

Here are declarations for typical containers,

```
 4     int a[10];                                                  /* C */
 5
 6     map<terminal_t::value_type, master_t::size_type> count;    //C++
```

The name of the data type of the C++ container in the above line 6 is actually even more complicated: the `terminal_t` and `master_t` are typedefs for the template classes `terminal<printable_t>` and `list<wabbit *>`.

The data types of the variables have become more complicated in C++. What do we get in return? There are fewer variables and fewer lines of executable code.

```
 7 void wabbit::move()                 //C style
 8 {
 9     int dx;                         //uninitialized variables
10     int dy;
11     decide(&dx, &dy);
12
13     x += dx;
14     y += dy;
```

```
15 void wabbit::move()                 //C++ style
16 {
17     const terminal_t::difference_type d = decide();
18
19     it += d;
```

In other words, some of the complexity of the program has been moved from the executable code to the declarations. This is good because the compiler is more likely to catch an error in a declaration than an error in executable code. For example, if we accidentally write −= instead of += in the above line 13, or if we forget the line entirely, the compiler will not catch it. But if `d` is declared to be the wrong data type in line 17, the program will not compile.

The C++ data types also make the code more flexible. For example,

(1) We can turn features on and off at compile time by plugging in the names of different data types. This eliminates one need for conditional compilation.

```
20         terminal<printable_t> term1('.');     //This object does error checking.
21         terminal<char>      term2('.');       //This object doesn't.
```

(2) We can mix and match features just by plugging in the names of different data types. This eliminates one need for copying and pasting. Furthermore, the template hides the private multiple inheritance.

```
22         new grandchild<immobile, predator_t, 'B'>(this, it);
23         new grandchild<brownian,   victim_t, 'r'>(this, it);
```

(3) We can migrate to a different environment just by plugging in the names of different data types.

```
24     //typedef terminal<char> terminal_t;
25     //typedef terminal<printable_t> terminal_t;
26     typedef terminal0<printable_t> terminal_t;
```

```
class game {
    terminal_t it;
    list<wabbit *> master;
    map<terminal_t::value_type, master_t::size_type> count;
};
```