C++ Part III X52.9266 Starting Point

0.0.1 A Portable Interface to the Terminal

We will write a video game in C++ to explore objects, inheritance, templates, exceptions, and the C++ Standard Library. To concentrate on these features, our game will ignore color, pixels, and sound files. It will treat the screen as a monochrome display of rows and columns of characters, emitting a monotone beep. It will use the keyboard but not the mouse.

To make the game platform independent, all its graphics and special effects will be done by calling the following ten functions. They work the same way in Unix C++, Microsoft C++, and other platforms. They are written in C and their header file `term.h` can be included in a C file or a C++ file.

Call the function `term_put` in line 15 to display a character at any (`x`, `y`) position on the screen. Call `term_puts` in line 16 to display a string of characters starting at the given position. If we forget what character has been written at a position, we can read it back with `term_get` in line 19.

On all platforms, the column numbers (the `x`'s) go from left to right, starting at zero. The rows numbers (the `y`'s) go from top to bottom, starting at zero. The origin (0, 0) is therefore at the upper left corner of the screen. How high do the coördinates go? The answer will be different on each platform. The functions `term_xmax` and `term_ymax` in lines 11–12 will return the number of columns and rows on your platform. If the number of columns is 80, the column numbers will range from 0 to 79 inclusive.

We will always use unsigned numbers to represent a position in a space whose coördinates start at zero. This will prevent the coördinates from ever being negative. Our examples are the arguments of `term_put`, `term_puts`, and `term_get`, and the return values of `term_xmax` and `term_ymax`.

The function `term_key` in line 23 returns the most recently pressed keystroke from the keyboard. It differs from the C function `getchar` in that the keystroke is returned immediately, without waiting for the user to press `RETURN`. In other words, it makes the keyboard live. (Each call to `term_key` actually returns the most recent keystroke that was not returned by a previous call. If every keystroke has already been returned, or if there were no keystrokes yet, `term_key` will return the character `'\0'`.)

The function `term_wait` in line 25 pauses for the specified number of milliseconds. `term_beep` emits a beep.

I saved the two most important `term_` functions for last. There is always some setup to be done before we can do any special effects. On some platforms we have to put the screen into graphics mode; on others, we have to pop up a graphics window to draw in. Similarly, there is always some cleanup: we have to put the screen back into text mode or make the graphics window disappear.

On every platform, the function `term_construct` in line 5 will do whatever setup is necessary. It must be called before the other `term_` functions will work. The function `term_destruct` in line 6 will do the cleanup. It must be the last `term_` function called. Failure to call `term_destruct` may leave your terminal in an unusable state.

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9264/src/term/term.h`

```
 1 #ifndef TERMH
 2 #define TERMH
 3
 4 /* Must be called before and after the other term_ functions. */
 5 void term_construct(void);
 6 void term_destruct(void);
 7
 8 /* Legal x values go from 0 to term_xmax() - 1 inclusive.
 9    Legal y values go from 0 to term_ymax() - 1 inclusive. */
10
11 unsigned term_xmax(void);    /* number of columns of characters */
12 unsigned term_ymax(void);    /* number of rows of characters */
```

```

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

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

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

void term_wait(int milliseconds);    /* 1000 milliseconds == 1 second */
void term_beep(void);
#endif
```

▼ Homework 0.1: test the terminal interface

Compile and run the following C++ program on your machine. You'll also need the above `.h` file, and the corresponding `term.c` file of function definitions in the same directory on the web.

A C program would have had to declare all its variables immediately after the `{` in line 9. Our C++ program doesn't declare its variables until we have values to put into them in lines 11−15 and 18.

For the time being, we need two loops with two counters (`x` and `y` in lines 23–24) because the terminal is two-dimensional. But when we do containers and iterators in the C++ Standard Library, we'll be able to loop through the terminal with only one loop and one counter.

Do not write a newline onto the screen at the end of line 21. To move down the screen, simply give a larger `y` argument to the next call to `term_put` and `term_puts`.

—On the Web at
`http://i5.nyu.edu/˜mm64/x52.9264/src/term/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 extern "C" {
 4 #include "term.h"   //beware if your machine has another file named term.h
 5 }
 6 using namespace std;
 7
 8 int main()
 9 {
10     term_construct();
11     const unsigned xmax = term_xmax();
12     const unsigned ymax = term_ymax();
13
14     unsigned x = xmax / 2;   //center of screen
15     unsigned y = ymax / 2;
16
17     term_put(x, y, 'X');
18     char c = term_get(x, y);
19     term_put(x + 1, y, c);
20
21     term_puts(0, 0, "Please type some characters ending with a q.");
22
23     for (y = 1; y < ymax; ++y) {
24         for (x = 0; x < xmax; ++x) {
```



```
25                 while ((c = term_key()) == '\0') {
26                 }
27
28                 if (c == 'q') {    //quit
29                     goto done;
30                 }
31
32                 term_put(x, y, c);
33             }
34         }
35
36         done:;
37         term_wait(1000);    //one full second
38         term_beep();
39         term_destruct();
40         return EXIT_SUCCESS;
41 }
```

The expression in the above line 25

```
((c = term_key()) == '\0')
```

executes the same operators (substituting == for `!=`), in the same order, as the following classic idiom in C.

```
((c = getchar()) != EOF)
```

▲

List of the three source files that constitute the test program

(1) `term.h` and `term.c` (both online in the same directory in pp. 1–2). They are the only two written in C; the other is in C++.

(2) `main.C` (pp. 2–3)

Select the platform.

The `term.c` file is written in C. Be sure that you don't accidentally tell your computer that it's written in C++ by giving it the wrong filename extension. Before compiling, uncomment (i.e., remove the comment delimiters from) exactly one of the following three macro definitions at the top of this file:

```
1 /* #define UNIX */
2 /* #define MICROSOFT */
3 /* #define BORLAND */
```

For example, if you were compiling under Microsoft, you would change line 2 to

```
4 #define MICROSOFT
```

Make no other change to `term.c`. Unix pros can use the `-D` option of `gcc` and `g++` instead of uncommenting.

Compile under Unix

See curses(3curses) at http://i5.nyu.edu/~mm64/man/, or *Programming with curses* by John Strang; O'Reilly & Associates, 1986; ISBN 0–937175–02–1.

http://www.oreilly.com/catalog/curses/

The "minus uppercase I dot" option tells gcc to #include the term.h file in the current directory instead of the one in the /usr/include/ directory.

The -DUNIX= option defines the macro UNIX to be the null string, eliminating the need for the above uncommenting. Bloomberg people should also give the compiler the option -D_WIDEC_H= to prevent the compiler from including the file /usr/include/widec.h. Remember to use this option whenever compiling term.c.

The minus lowercase c option tells gcc to create a .o file instead of an executable file. The minus lowercase L option tells g++ to link in the library /usr/ccs/lib/libcurses.a.

```
1$ gcc -I. -DUNIX= -c term.c                    Create the file term.o.
2$ ls -l term.o                                 minus lowercase L

3$ g++ -I. -o ~/bin/tester main.C term.o -lcurses
4$ ls -l ~/bin/tester
5$ tester                Run it; but first make sure your terminal is set to vt100, bit ansi.
```

Compile under Microsoft

Create a "Win32 Console Application".

0.1 An Interface Class for the Terminal

Another way to write on the screen

The major classes we have seen so far are date, life, and stack. We will introduce one more, class terminal, before talking about classes in general.

Instead of doing our special effects by calling the ten C functions term_ in pp. 1–4, we will now do them by constructing an object of class terminal and calling its ten member functions. Compare the following test program with the main.C back in pp. 2–3. It does exactly the same demo but with a different notation. For each C function, there is now a member function that does the same job.

For convenience, we also introduce two member functions which have no C counterparts. The function next in lines 18 and 23 takes a pair of coördinates, x and y, and advances them to the next location. It would change (0, 0) to (1, 0), one location to the right. And on a screen with 80 columns, it would change (79, 0) to (0, 1), the first location on the next line. Finally, with 24 rows it would change (79, 23) to (0, 24), one step below the bottom of the screen, but would refuse to advance it any farther.

Pages 69–71 asked you to make every reference argument read-only. The call to next in line 18 shows the danger of violating this rule. Although there's no way to see it by inspecting that line, next changes the values of x and y. We will clean this up when we do "iterators".

The function in_range in line 23 returns true if the pair of coördinates is on the screen. It was named after the out_of_range "exception" which we will see later. See Lippman p. 291, Stroustrup pp. 53–54, 384–386, 445–446, 586–587.

We need two variables, x and y, to loop across the screen in line 23. When we have iterators we will be able to do the loop with only one, even though the screen is two-dimensional. To prepare for that day, I changed the pair of nested loops in the original test program lines 23–24 of main.Cin p. 2) to the single loop in line 23. This change is premature: if there are two variables, there should be two loops. But try to think of the x and y as a single structure with two fields. In time they will be.

—On the Web at
`http://i5.nyu.edu/˜mm64/x52.9264/src/terminal/main.C`

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

int main()
{
    const terminal term('.'); //The constructor for term calls term_construct.

    const unsigned xmax = term.xmax();
    const unsigned ymax = term.ymax();

    unsigned x = xmax / 2;       //center of screen
    unsigned y = ymax / 2;

    term.put(x, y, 'X');
    const char c = term.get(x, y);
    term.next(x, y);
    term.put(x, y, c);

    term.put(0, 0, "Please type some characters ending with a q.");

    for (x = 0, y = 1; term.in_range(x, y); term.next(x, y)) {
        char c;                  //uninitialized variable
        while ((c = term.key()) == '\0') {
        }

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

        term.put(x, y, c);
    }

    term.wait(1000);
    term.beep();
    return EXIT_SUCCESS;      //The destructor for term calls term_destruct.
}
```

An interface class

Let's read the definition for class `terminal`, starting with the simplest member function. The `beep` function in line 29 simply calls the corresponding C function `term_beep`. A function that does all its work by calling another function is called a *call-through.* See Stroustrup pp. 719, 778–781.

Most of the member functions of class `terminal` are ultimately call-throughs; the three simplest examples are in lines 27–29. Since this class does almost no work on its own, we call it an *interface class.* It merely delivers the results of another piece of software.

Now let's look at the data members. The constructor takes a `char` argument and stores it in the data member `_background` in line 9 of `terminal.C`. It has an underscore because a class can't have a data member and a member function with the same name. I burdened the private member with the underscore to keep the name of the public member short and simple. The constructor also initializes the screen in line 11 of `terminal.C`, and then stores the dimensions of the screen into the other two data members `_xmax` and `_ymax` in lines 13 and 14. The member functions `background`, `xmax`, and `ymax` in lines 18–20 of `terminal.h` simply return these data members.

The two-argument `put` in line 23 of `terminal.h` passes the `_background` data member to the three-argument `put` in line 22. This has the effect of letting `_background` be the default value for the third argument. I wish we could combine the two functions into one with a default value for its third argument:

```
    void put(unsigned x, unsigned y, char c = _background) const;
```

But the language just does not let us do this. A data member of an object can be mentioned inside the *body* of a member function of the same object, but it cannot be mentioned inside the *argument list* of a member function of the same object.

The `in_range` in line 31 has no need to check if `x` and `y` are negative. They never can be, because they are unsigned.

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9264/src/terminal/terminal.h`

```
 1 #ifndef TERMINALH
 2 #define TERMINALH
 3
 4 extern "C" {
 5 #include "term.h"
 6 }
 7
 8 class terminal {
 9     char _background;   //default value for third argument of put
10     unsigned _xmax;     //number of columns of characters
11     unsigned _ymax;     //number of rows of characters
12
13     void check(unsigned x, unsigned y) const;
14 public:
15     terminal(char initial_background = ' ');
16     ~terminal();
17
18     char background() const {return _background;}
19     unsigned   xmax() const {return _xmax;}
20     unsigned   ymax() const {return _ymax;}
21
22     void put(unsigned x, unsigned y, char c) const;
23     void put(unsigned x, unsigned y) const {put(x, y, _background);}
24     void put(unsigned x, unsigned y, const char *s) const;
25     char get(unsigned x, unsigned y) const {check(x, y); return term_get(x, y);}
26
27     char key() const {return term_key();}
28     void wait(int milliseconds) const {term_wait(milliseconds);}
29     void beep() const {term_beep();}
30
31     bool in_range(unsigned x, unsigned y) const {return x < _xmax && y < _ymax;}
32     void next(unsigned& x, unsigned& y) const;
33 };
34 #endif
```

Every character is ultimately put on the screen by the three-argument `put` in line 36, which calls the C Standard Library function `isprint` to check that the character is printable. If it is not, we cast the character to `unsigned` to print its ASCII code. See line 12 of `cast.C` in p. 24.

The `initial_background` argument of the constructor is checked when line 19 fills the screen with the background character.

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9264/src/terminal/terminal.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cctype>   //for isprint (or ctype.h)
 4 #include "terminal.h"
 5 using namespace std;
 6
 7 terminal::terminal(char initial_background)
 8 {
 9     _background = initial_background;
10
11     term_construct();
12
13     _xmax = term_xmax();
14     _ymax = term_ymax();
15
16     if (_background != ' ') {
17         for (unsigned y = 0; y < _ymax; ++y) {
18             for (unsigned x = 0; x < _xmax; ++x) {
19                 put(x, y);
20             }
21         }
22     }
23 }
24
25 terminal::~terminal()
26 {
27     for (unsigned y = 0; y < _ymax; ++y) {
28         for (unsigned x = 0; x < _xmax; ++x) {
29             put(x, y, ' ');
30         }
31     }
32
33     term_destruct();
34 }
35
36 void terminal::put(unsigned x, unsigned y, char c) const
37 {
38     if (!isprint(c)) {
39         cerr << "unprintable character " << static_cast<unsigned>(c) << ".\n";
40         exit(EXIT_FAILURE);
41     }
42
43     check(x, y);
44     term_put(x, y, c);
45 }
46
47 void terminal::put(unsigned x, unsigned y, const char *s) const
48 {
49     for (; *s != '\0'; ++s) {
50         put(x, y, *s);
```

```
        next(x, y);
    }
}

//Move to the next (x, y) position: left to right, top to bottom.
//Warning: will change the values of the arguments.

void terminal::next(unsigned& x, unsigned& y) const
{
    check(x, y);

    if (++x >= _xmax) {
        x = 0;
        ++y;
    }
}

void terminal::check(unsigned x, unsigned y) const
{
    if (!in_range(x, y)) {
        cerr << "coordinates (" << x << ", " << y
            << ") must be >= (0, 0) and < ("
            << _xmax << ", " << _ymax << ")\n";
        exit(EXIT_FAILURE);
    }
}
```

List of the five source files that constitute the test program

(1) `term.h` and `term.c` (pp. 1–4.) These are the only two written in C; the rest are in C++.

(2) `terminal.h` and `terminal.C` (pp. 5–8)

(3) `main.C` (pp. 4–5)

Compile the test under Unix

```
1$ gcc -I. -DUNIX= -c term.c          minus uppercase I
2$ ls -l term.o                       minus lowercase L

3$ g++ -I. -o ~/bin/tester main.C terminal.C term.o -lcurses
4$ ls -l ~/bin/tester
5$ tester                             Run it.
```

It's just as fast to call the member functions of class terminal.

Instead of calling the C functions directly, we are now calling them through the member functions of a `terminal` object. In a moment we will see the benefits of this extra layer of software. But first we must consider if the extra layer has slowed the program down.

When we write a call to an inline function, the computer behaves as if we had written the body of the inline function in place of the call. When we write line 2, for example, the computer behaves as if we had written line 3. Calling the member function `beep` in line 2 is therefore just as fast as calling the C function `term_beep` in line 3.

```
1    const terminal term('.');
2    term.beep();              //When we write this,
3    term_beep();              //the computer behaves as if we had written this.
```


Sometimes the member functions of class `terminal` are even faster. When we write line 5, the computer behaves as if we had written line 6. But line 6 calls no function; it simply uses the value of a data member. Calling the member function `xmax` in line 5 is therefore faster than caling the C function `term_xmax` in line 7.

```
4     const terminal term('.');
5     cout << term.xmax() << "\n";    //No function is called.
6     cout << term._xmax << "\n";
7     cout << term_xmax() << "\n";    //A function is called.
```

Why bother with an interface class?

> The B words were in all cases compound words.
>
> —George Orwell, *1984*, Appendix: The Principals of Newspeak

Class `terminal` does not slow down the program, and in a few cases it makes it faster. But the real reason we introduced this extra layer is for æsthetics. Here is how calling the member functions of an object is more convenient than calling naked C functions.

(1) The C function names had to be compound words because we might have several devices to manipulate. If there are n devices and m functions for each device, the number of different function names will be $n \times m$.

```
term_beep();          /* C: number of names increases geometrically */
modem_beep();
pager_beep();
```

But the C++ member function names can be shorter because the member functions belong to an object. If there are n devices and m functions for each device, the number of different names will be only $n + m$.

```
term.beep();          //C++: number of names increases arithmetically
modem.beep();
pager.beep();
```

Our first example of shortening the names was in p. 86.

(2) The C++ member functions also have fewer and simpler names thanks to function name overloading.

```
/* C: every function must have a different name. */
term_put(x, y, c);    /* display a character */
term_puts(x, y, s);   /* display a string */
```

//C++: can use same name for similar functions term.put(x, y, c); //display a character term.put(x, y, s); //display a string

(3) The most frequently used value for an argument can be made the default in C++. For example, the most frequently displayed character is the background character.

```
term_put(x, y, '.');      /* C */
term.put(x, y);           //C++: display term's background character
```

(4) Instead of two widely separated function calls

```
term_construct();
//the whole game
term_destruct();
```

we now write only a single declaration:

```
terminal term('.');       //This declaration calls the constructor.
//the whole game
```



```
17      return from main;          //The return from main calls the destructor.
```

If the call to `term_construct` in the above line 16 was missing in C, a call to `term_put` at line 17 would still compile but would execute incorrectly. But if the declaration for `term` in the above line 19 was missing in C++, a call to `term.put` at line 20 would not even compile. Not compiling is better than executing incorrectly.

(5) Packaging the ten C functions as a class would also make it easier to have a program with more than one terminal. As we will see, this is one of the main reasons for making a class.

Two improvements to class terminal

(6) The three member functions `beep`, `wait`, and `key` can, and therefore should, be static. And now that they are static, they can no longer be `const`. Only a non-static member function can be `const`.

(7) The three data members `_background`, `_xmax`, and `_ymax` can, and therefore should, be `const`. And now that they are `const`, they can no longer be assigned to in lines 9, 13, and 14 of the above `term.C`. You will have to initialize them with a colon. Unfortunately, the function `term_construct` must be called before we initialize `_xmax` and `_ymax`. We can do this with tricky code with the comma operator.

```
//Definition of constructor for class terminal.

terminal::terminal(char initial_background)
    : _background(initial_background),
    _xmax(term_construct(), term_xmax()),
    _ymax(term_ymax())
{
    if (_background != ' ') {       //etc.
```