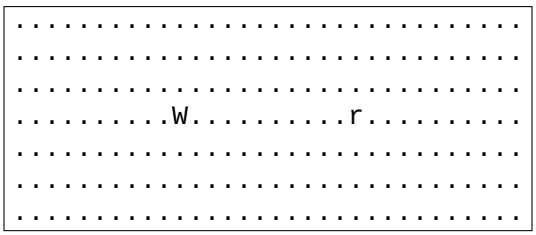
0.1 The Rabbit Game

▼ Homework 0.1a:
Version 1.0 of the Rabbit Game: initial version of the game



A large program will be required to demonstrate the powerful features of C++ and their interactions: inheritance (single and multiple), virtual functions, templates, and the Standard Template Library (STL). Instead of burdening the student with a separate program for each feature, we will deploy them all in one evolving program, a video game with moving animals.

Carnivores will be uppercase, herbivores lowercase. The `r`, for example, is a rabbit. It hops randomly around the terminal, one step at a time. It knows that it can't move off the screen or occupy the same place at the same time as another animal.

The `W` is the wolf. It is under manual control: you have to press keys to move it. To avoid the complexity of making the arrow keys work on all platforms, we use four letters:

`h`	*left*
`j`	*down*
`k`	*up*
`l`	*right (lowercase L)*

These four letters are in a row on a QWERTY keyboard. (They are also the motion keys in the Unix editor `vi`.) I readily concede that it is counterintuitive for L to mean ''right''.

You win the game by making the wolf stomp on the rabbit. You can also win merely by launching the game and going out to lunch. The rabbit, moving randomly around the screen, will eventually blunder into the wolf and be eaten.

The game has three objects: the `terminal`, `wolf`, and `rabbit`. The calls to their constructors will be visible: the `terminal` will fill the screen with its background character, and the two animals will draw themselves. Eventually, the calls to their destructors will also be visible: the rabbit and wolf will erase themselves, and the `terminal` will blank itself out.

Each animal will have `x` and `y` data members giving its current location. We will see the data members changing: whenever this happens, the animal will move.

The main function

The `srand` function in line 12 of `main.C` ''seeds'' the random number generator, ensuring that the subsequent calls to `rand` in lines 42–43 of `rabbit.C` on p. 4 will return a different series of random numbers each time the game is run. `srand` must be supplied with an initial random number, the seed on p. 163. For this we use the current time. The zero in line 12 would have been `NULL` in C; see p. 62.

Lines 18–19 construct the wolf and rabbit one-third of the screen apart, at middle height. The main loop in line 21 will then call the `move` member function of each animal four times per second. These functions return `true` if the rabbit is still alive, `false` if it has been eaten. When that happens, we break out of the main loop and the game is over.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9264/src/game1/main.C

```
 1 #include <cstdlib>   //for the srand function and EXIT_SUCCESS
 2 #include <ctime>     //for the time function
 3
 4 #include "terminal.h"
 5 #include "wolf.h"
 6 #include "rabbit.h"
 7
 8 using namespace std;
 9
10 int main()
11 {
12     srand(time(0));
13     const terminal term('.');
14
15     const unsigned xmax = term.xmax();
16     const unsigned ymax = term.ymax();
17
18     wolf   w(term,     xmax / 3, ymax / 2);
19     rabbit r(term, 2 * xmax / 3, ymax / 2);
20
21     for (;; term.wait(250)) {   //250 milliseconds equals .25 seconds
22         if (!w.move()) {
23             break;
24         }
25         if (!r.move()) {
26             break;
27         }
28     }
29
30     term.put(0, 0, "You killed the rabbit!");
31     term.wait(3000);     //Give user three seconds to read the message.
32     return EXIT_SUCCESS; //Destruct rabbit, wolf, & terminal, in that order.
33 }
```

The above lines 21−28 may be combined to

```
34     for (; w.move() && r.move(); term.wait(250)) {
35     }
```

But don't do it. We would just have to change it back in a later version of the game.

Class rabbit

The game is played on a `terminal` object shared by the two the animals. The animals call the member functions of the terminal.

One way to make the terminal accessible to the animals would be to make it a global variable.

```
1 const terminal term('.');
2
3 int main()
4 {
```

But if we did this, we would be locking ourselves into having exactly one terminal and exactly one game. It would be impossible to turn our program into a game server that runs many games simultaneously.

To keep our options open, we made the terminal accessible to the animals by giving each animal a pointer to the terminal it inhabits. This pointer `t` in line 6 is read-only to make it impossible for an animal to change the size or background character of its terminal. To ensure that `rabbit.h` can mention the name of class `terminal`, it must include `terminal.h`.

The data members in lines 6–8 of `rabbit.h` are of the built-in data types: integers, characters, pointers. They are not objects, they have no constructors, and nothing happens when they are constructed. As long as they are constructed before being used in lines 13 and 28 of `rabbit.C`, it doesn't matter what order they are constructed in.

This being the case, there is no reason at present to declare `t` before the other three data members. But perhaps there will be a reason in the future. The four data members might become objects, each initialized by its own constructor. When that happens, the error checking now performed in lines 13 and 28 of `rabbit.C` will be done in the constructors for `x`, `y`, and `c`. The data member `t` is used by this error checking code, so `t` will have to be constructed first. To prepare for this eventuality, `t` is constructed first by being declared first in lines 6–8 of `rabbit.h`, although we do not need this right now. It will be one less thing to change should the data members ever become objects.

We consistently use an unsigned number to represent a position in a space whose coördinates start at zero. Earlier examples were the unsigned data type `size_t` for an array subscript (pp. 59–60); the unsigned arguments and return value of the C functions `term_put` and `term_xmax` on p. 79; the unsigned arguments and return value of the `put` and `xmax` member functions of class `terminal` on pp. 148–149. In keeping with this practice, the coördinates of an animal are unsigned to keep them from becoming negative. These include the data members `x` and `y` in line 7 of `rabbit.h` and the local variables `newx` and `newy` in lines 49–50 of `rabbit.C`.

On the other hand, we use a signed number to represent a direction and distance of motion. Earlier examples were the signed argument of the function `date::next` and the local variables `dx` and `dy` in `life::next`. In keeping with this practice, horizontal or vertical motions are signed to let them be positive or negative. These include the offsets `dx` and `dy` in lines 43–44 of `rabbit.C`. The unsigned/signed distinction appeared in the C Standard Library as `size_t` vs. `ptrdiff_t`, and will reappear in the containers in the C++ Standard Library as `size_type` vs. `difference_type`.

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9264/src/rabbit1/rabbit.h`

```
 1 #ifndef RABBITH
 2 #define RABBITH
 3 #include "terminal.h"
 4
 5 class rabbit {
 6     const terminal *t;
 7     unsigned x, y;
 8     char c;
 9 public:
10     rabbit(const terminal& initial_t, unsigned initial_x, unsigned initial_y);
11     bool move();
12 };
13 #endif
```

We saw back on pp. 169–171 what will go wrong when calling `exit` in line 17: the objects that are not statically allocated will never be destructed. We will fix this bug when we cover ''exceptions''. For now, let's hope it never happens. Line 23 disallows two animals in the same location at the same time. Line 28 disallows an invisible wabbit: one whose ''color'' (character) is the same as the terminal's background.

The value of the expression `rand() % 3` in line 42 is either 0, 1, or 2. The value of the larger expression `rand() % 3 - 1` is therefore −1, 0, or 1, to indicate left, no motion, or right.

Line 34 ‘‘registers’’ the newborn `rabbit` with its `terminal`: it informs the `terminal` that the `rabbit` exists. This is a clear demonstration that an object’s constructor must sometimes do more than just put values into the object’s data members. It must also notify other objects about the birth of the new one.

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9264/src/rabbit1/rabbit.C`

```
 1 #include <iostream>
 2 #include <cstdlib>   //for rand and exit functions
 3 #include "rabbit.h"
 4 using namespace std;
 5
 6 rabbit::rabbit(const terminal& initial_t, unsigned initial_x, unsigned initial_y)
 7 {
 8     t = &initial_t;
 9     x = initial_x;
10     y = initial_y;
11     c = 'r';
12
13     if (!t->in_range(x, y)) {
14         cerr << "Initial rabbit position (" << x << ", " << y
15             << ") off " << t->xmax() << " by " << t->ymax()
16             << " terminal.\n";
17         exit(EXIT_FAILURE);
18     }
19
20     const char other = t->get(x, y);
21     const char background = t->background();
22
23     if (other != background) {
24         cerr << "Initial rabbit position (" << x << ", " << y
25             << ") already occupied by '" << other << "'.\n";
26         exit(EXIT_FAILURE);
27     }
28
29     if (c == background) {
30         cerr << "Rabbit character '" << c << "' can't be the same as "
31             "the terminal's background character.\n";
32         exit(EXIT_FAILURE);
33     }
34
35     t->put(x, y, c);
36 }
37
38 //Return false if this rabbit was eaten, true otherwise.
39
40 bool rabbit::move()
41 {
42     //The values of dx and dy are either -1, 0, or 1.
43     const int dx = rand() % 3 - 1;
44     const int dy = rand() % 3 - 1;
45
46     if (dx == 0 && dy == 0) {
47         return true;   //This rabbit had no desire to move.
48     }
```

```
49
50      const unsigned newx = x + dx;
51      const unsigned newy = y + dy;
52
53      if (!t->in_range(newx, newy)) {
54          return true;   //Can't move off the screen.
55      }
56
57      const char other = t->get(newx, newy);
58
59      if (other != t->background()) {
60          if (other == c) {
61              //This rabbit collided with another rabbit.
62              return true;
63          } else {
64              //This rabbit blundered into the wolf and was eaten.
65              return false;
66          }
67      }
68
69      t->put(x, y);      //Erase this rabbit from its old location.
70      x = newx;
71      y = newy;
72      t->put(x, y, c);   //Redraw this rabbit at its new location.
73
74      return true;
75  }
```

The above lines 59−65 may be combined to the single statement

```
76          return other == c;
```

But don't do it. It's clearer the way it is now.

The above lines 69−71 may be combined to

```
77      t->put(x = newx, y = newy, c);
```

But don't do it. C++ does not share C's rage to cram as much code as possible into a single expression.

The above lines 68 and 71 seem to form a pair. Should they be rewritten as calls to a constructor and destructor for some new kind of object? Closer inspection reveals that the constructor would be called at line 71 and the destructor at 68. Then should 68 be paired with 34, and 71 with a line you will write in the destructor for class `rabbit` in Homework 0.1b?

I decided to leave lines 68 and 71 as they stand because the new object would be too trivial to be of any use. What would we call this kind of object: an `apparition`? A `quantum`? See pp. 166−167.

Unfortunately, every line of class `rabbit` betrays the fact that our terminal is Cartesian and two-dimensional, from the data members `x` and `y` to the double-barreled arithmetic in the above lines 42−43 and 49−50. When we have ''iterators'', we will be able to rewrite the game without any mention of `x` and `y`, `dx` and `dy`. We will then be in a position to port the game to a terminal with a different topology: polar coördinates, three dimensions, etc.

Class wolf

The data members, and the declarations for the member functions, are the same in classes `wolf` and `rabbit`.

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9264/src/wolf1/wolf.h`

```
 1 #ifndef WOLFH
 2 #define WOLFH
 3 #include "terminal.h"
 4
 5 class wolf {
 6     const terminal *t;
 7     unsigned x, y;
 8     char c;
 9 public:
10     wolf(const terminal& initial_t, unsigned initial_x, unsigned initial_y);
11     bool move();
12 };
13 #endif
```

An array of structures is the easiest way for a C or C++ program to store information in rows and columns (lines 27−32). In both languages, we use the data type `size_t` for the number of elements in an array (line 33).

The declaration for `p` is tucked in the left parentheses of the `for` loop in line 36; see pp. 30−31. Similarly, the declaration for `k` is tucked in the left parentheses of the `if` in line 35. The `if` will be true if the initial value of `k` is non-zero, which will happen if the user pressed a key. `k` will be destructed when we reach the end of the `if`, marked by the `}` in line 59.

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9264/src/wolf1/wolf.C`

```
 1 #include <iostream>
 2 #include <cstdlib>   //for exit function
 3 #include "wolf.h"
 4 using namespace std;
 5
 6 wolf::wolf(const terminal& initial_t, unsigned initial_x, unsigned initial_y)
 7 {
 8     t = &initial_t;
 9     x = initial_x;
10     y = initial_y;
11     c = 'W';
12
13     //Copy lines 13-35 of the above rabbit.C here,
14     //changing the word "rabbit" to "wolf".
15 }
16
17 //Return false if this wolf ate another animal, true otherwise.
18
19 bool wolf::move()
20 {
21     struct keystroke {
22         char c;
23         int dx;   //horizontal difference
24         int dy;   //vertical difference
25     };
26
27     static const keystroke a[] = {
28         {'h',   -1,    0},   //left
29         {'j',    0,    1},   //down
30         {'k',    0,   -1},   //up
31         {'l',    1,    0}    //right
```

```
32      };
33      static const size_t n = sizeof a / sizeof a[0];
34
35      if (const char k = t->key()) {
36          for (const keystroke *p = a; p < a + n; ++p) {
37              if (k == p->c) {
38                  const unsigned newx = x + p->dx;
39                  const unsigned newy = y + p->dy;
40
41                  if (!t->in_range(newx, newy)) {
42                      break;          //Go to line 57.
43                  }
44
45                  const bool I_ate_him =
46                      t->get(newx, newy) != t->background();
47
48                  t->put(x, y);       //Erase this wolf from its old location.
49                  x = newx;
50                  y = newy;
51                  t->put(x, y, c);    //Redraw this wolf at its new location.
52
53                  return !I_ate_him;
54              }
55          }
56
57          //Punish user who pressed an illegal key or tried to move off screen.
58          t->beep();
59      }
60
61      return true;
62  }
```

Up to one quarter of a second may elapse between a keystroke and the next call to the wolf's `move` function, causing the wolf to respond sluggishly. This could be fixed by making the input ''interrupt driven'', but we will not pursue it for now.

For the present, there is an asymmetry in the behavior of colliding animals. When a wolf stomps on a rabbit, the rabbit disappears. But when a rabbit blunders into a wolf, the rabbit merely freezes because its move is never carried out. We'll fix this on p. 422 when we introduce ''dynamic memory allocation'', which will give us greater control over the exact moments of an object's birth and death.

Classes `wolf` and `rabbit` are identical in their data members, almost identical in their constructors, and similar in their remaining member functions. We will eventually consolidate this duplication by means of *inheritance* from a common base class.

List of the nine source files that constitute the game

(1) `term.h` and `term.c` (pp. 78−82). These are the only two written in C; the rest are C++.

(2) `terminal.h` and `terminal.C` (pp. 146−152)

(3) `main.C` (pp. 1−2)

(4) `rabbit.h` and `rabbit.C` (pp. 2−5)

(5) `wolf.h` and `wolf.C` (pp. 5−7)

Compile the game on Unix

```
1$ gcc -I. -DUNIX= -c term.c
2$ ls -l term.o

3$ g++ -I. -o ~/bin/game main.C wolf.C rabbit.C terminal.C term.o -lcurses
4$ ls -l ~/bin/game

5$ game                                  Run the game.
6$ echo $?                               See the game's exit status.
```

▲

▼ Homework 0.1b:
Version 1.1 of the Rabbit Game: destructors for classes `wolf` and `rabbit`

Write a destructor for class `wolf`, even though there currently is no animal that could eat one, and a destructor for class `rabbit`. Each destructor should do three things in the following order.

(1) Beep the terminal on which the dying animal is displayed.

(2) Pause for one second, so the stricken animal stands ''frozen in the headlights''.

(3) Call the `get` member function of the animal's terminal to see if the animal's location on the screen is occupied by the animal's character. If so, wipe the animal off the screen by displaying the terminal's background character there, as in line 48 of the above `wolf.C`. Otherwise, display nothing because the location is already occupied by another animal. Remember, there is one occasion when two animals are momentarily at the same place at the same time: right after the wolf stomps on the rabbit. This anomalous situation will be removed when we have ''dynamic memory allocation'', but for now we have to handle it.

The destructor should not change the value of any of the dying animal's data members. What would be the point? The animal is about to evaporate. Changing its data members would be like rearranging the deck chairs on the *Titanic*.

▲

▼ Homework 0.1c:
Version 1.2 of the Rabbit Game: make the animals impossible to copy

Wolves and rabbits will not be allowed to multiply—yet. Let's ensure that no animal can be copied.

A C++ object can be copied only by its copy constructor. We can therefore make an object impossible to copy simply by depriving it of a copy constructor. And in fact we didn't write a copy constructor for classes `wolf` and `rabbit`. But for that very reason the computer wrote them for us. See pp. 125−126.

To prevent the computer from supplying classes `wolf` and `rabbit` with a copy constructor, declare a private copy constructor for each class but do not define them. In other words, do not write a body for these functions. If a member function of one of these classes tries to call the copy constructor for that class, the program will not link because the copy constructor is undefined. And if any other function tries to call the copy constructor, the program will not even compile because the copy constructor is private. In either case, it will be impossible to copy the animal.

```
1 class rabbit {
2     const terminal *t;
3     unsigned x, y;
4     char c;
5     rabbit(const rabbit& another);   //deliberately undefined
6 public:
7     //etc.
```

▲

Improvements to the game

(1) Define destructors for classes `wolf` and `rabbit`. The test in line 6 lets us erase the dying animal only if no other animal is occupying the same location. The need for this `if` will disappear later, when we enforce the rule that two animals will never be at the same place at the same time.

```
1 wolf::~wolf()
2 {
3     t->beep();
4     t->wait(1000);
5
6     if (t->get(x, y) == c) {
7         t->put(x, y);  //Erase the animal by drawing the background char
8     }
9 }
```

(2) In classes `wolf` and `rabbit`, the `c` data member can, and therefore should, be a `const char`. The `t` data member can, and therefore should, be a `const terminal *const`. The constructors for `wolf` and `rabbit` will then have to initialize the data members, rather than assign to them:

```
1 wolf::wolf(const terminal& term, unsigned initial_x, unsigned initial_y)
2     : t(&term), x(initial_x), y(initial_y), c('W')
3 {
4     //etc.
```

(3) The `main` function should construct an array of `rabbit`'s. Since each element of the array will be initialized by the copy constructor for class `rabbit`, that class will need its copy constructor back. Remove the declaration for the undefined copy constructor.

```
 1 int main(int argc, char **argv)
 2 {
 3     call srand;
 4     const terminal term('.');
 5
 6     declare the wolf;
 7     rabbit a[] = {
 8         rabbit(term,  0,  0),
 9         rabbit(term, 10, 10),
10         rabbit(term, 20, 20)
11     };
12     const size_t n = sizeof a / sizeof a[0];
13
14     for (;; term.wait(250)) {
15         if (!w.move()) {
16             goto done;
17         }
18
19         for (rabbit *p = a; p < a + n; ++p) {
20             if (!p->move()) {
21                 goto done;
22             }
23         }
24     }
25
26     done:;
27     term.put(0, 0, "You killed a rabbit!");
28     term.wait(3000);
```

```
29      return EXIT_SUCCESS;
30  }
```

(4) In classes `wolf` and `rabbit`, let `c` be a static data member. Declare the static data members before the non-static ones, since the static data members are created first. If your comp[iler lets you, initialize it in the class declaration.

```
 1  //Excerpt from wolf.h.
 2
 3  class wolf {
 4      static const char c = 'W';
 5      //etc.
```

If your compiler won't let you initialize it there, initialize it in the `wolf.C` file instead.

```
 6  //Excerpt from wolf.h.
 7
 8  class wolf {
 9      static const char c;
10      //etc.

11  //Excerpt from wolf.C.
12
13  const char wolf::c = 'W';
14
15  //Definition of constructor for class wolf:
16
17  wolf::wolf(const terminal& term, unsigned initial_x, unsigned initial_y)
18      : t(&initial_t), x(initial_x), y(initial_y)
```