12. Structures

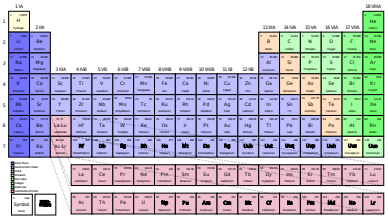
12.1. Introduction

Before the widespread use of fingerprinting, investigators in Europe and the U.S. used a system called the *portrait parlé* to identify criminals. Introduced in the 1880s by Alphonse Bertillon, this was a detailed written description of a person's physical characteristics. You'll find many references to the *portrait parlé* in Victorian detective fiction by writers like Gaston Leroux.

A "Bertillon card" describing the physical characteristics of Alphonse Bertillon himself.

Source: Wikimedia Commons

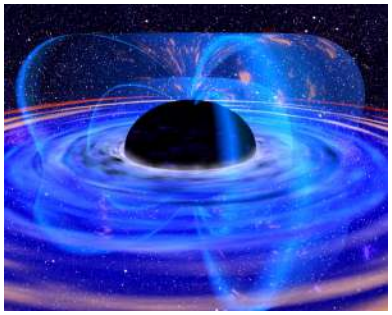
Anything studied by researchers will probably have more than one interesting property. In Chapter 6 we saw that researchers often make several measurements at the same time. For example, a census-taker visiting a house might record the number of children, the household income, the size of the house, and so forth. We could store each of these quantities in a separate variable, but we know that they're all actually properties of a single household. It might be convenient if that reality could be reflected in our programs.



Each chemical element has many properties: Atomic number, atomic mass, ionization energies, and so forth.

Source: Wikimedia Commons

Also, in Chapter 9 we learned that C functions can only return a single value. It would sometimes be useful to return several related values at once. Imagine, for example, that we were working with 3-dimensional vectors, which have x, y, and z components. Wouldn't it be great if we had a function that could add two vectors and return all three components of their sum?



Even black holes, perhaps the most featureless bodies in the universe, have at least three properties: mass, angular momentum, and charge.

Source: Wikimedia Commons

Fortunately, C allows us to do both of these things through a mechanism called "structures". Structures let us store several related measurements in one convenient place. Let's look at how to create and use structures by working through some examples.

12.2. The "struct" Statement

If we were comparing Virginia with other states, we might write a program that had variables like this:

```
int va_population;
double va_area;
double va_income;
```

It's tedious to create all of these variables, and could rapidly become confusing as we added more properties of the state, or more states. C provides us with a better way to do it, by allowing us to create custom-made variables that pack several properties together in one place. This is done with the "struct" statement:

```
struct {
  int population;
  double area;
  double income;
} va;
```

The statement above defines a new variable, `va`, that has, packed within it, several values of different types. The variable `va` isn't an `int` or a `double` or any of the other types we've used so far. It's a "structure". This structure is a completely new, custom-made type that contains whatever we want it to contain.

Once we've defined our `va` variable, we can set its properties like this:

```
va.population = 8326289;
va.area = 42774.2;
va.income = 61044;
```

The dot operator ("`.`") singles out one of the properties of the structure. Similarly, we can use the properties of the structure in just the same way we've used variables in the past:

```
if ( va.population < 8491079 ) {
  printf ("Virginia has fewer people than New York City\n");
}
```

We could even define an array of such structures, just as we've defined arrays of `int` or `double` variables:

```
struct {
  char name[20];
  int population;
```

```
  double area;
  double income;
  double birthrate;
  double deathrate;
} state[50];
```

Here we've defined an array of 50 structures, one for each of the 50 states. We've also added some more properties, such as the state's name. We can set the properties of an individual state by referring to it by its element number:

```
snprintf( state[0].name, 20, "Virginia" );
state[0].population = 8326289;
state[0].area = 42774.2;
state[0].income = 61044;
...etc.
```

This would make it easy to loop through all of the states:

```
for ( i=0; i<50; i++ ) {
  printf ( "Pop. of %s is %d\n", state[i].name, state[i].population );
}
```

What if we wanted to use the same structure for other variables? Say, for example, we wanted to store census data for a group of 100 countries. We could just re-type the structure definition:

```
struct {
  char name[20];
  int population;
  double area;
  double income;
  double birthrate;
  double deathrate;
} state[50];

struct {
  char name[20];
  int population;
  double area;
  double income;
  double birthrate;
  double deathrate;
} country[100];
```

There's a better way to do it, though. Instead of re-typing the "`struct`" statement, we can use "`typedef`" to define a name for this kind of structure.

12.3. Using "typedef"

If we want to re-use our structure, we can do something like this:

```
typedef struct {
  char name[20];
  int population;
  double area;
  double income;
  double birthrate;
  double deathrate;
} census;

census state[50];
census country[100];
```

The statements above define a new variable type named "`census`". We can use this new type to define variables, just like the `int` and `double` types we've used before. In the example above, we define two `census` arrays, one of 50 states and one of 100 countries.

`typedef` and `struct` are so often used together that many textbooks lump them into a single statement, "`typedef struct`", but they can be used separately too.

For example, `typedef` can be used to define an alternative name for any variable type. For example:

```
//Define aliases for some types:
typedef int funds;
typedef double weight;
typedef int days;

//Use these aliases to define some variables:
funds bank_balance;
weight fish_per_month[12];
days til_christmas;
```

This may make it easier for you to re-define your variables later on. Say, for example, that you've made so much money that you now need to

use a "`long int`"[1] to count your fortune! If your program uses the "funds" type for all of your accounting variables, then you'll only need to change one line: the `typedef` statement that defines "funds".

[1] In C, "`long int`" is a variable type that can hold larger number than a plain,old `int` is capable of holding.

12.4. Using Vectors in Programs

A "vector" is something that has both a magnitude and a direction. Physical properties like velocity and acceleration are vectors. Even though an eastbound car and a westbound car may have the same speed, their velocities are different, since they're going in different directions.

In many fields of science and mathematics, it's very useful to be able to define and use vectors in computer programs. Vectors are often represented by their x, y, and z components in a Cartesian coordinate system.

Portrait of Rene Descartes, French philosopher, mathematician, and scientist, for whom the "cartesian" coordinate system is named. As a philosopher, he's famous for his statement "*cogito ergo sum*" ("I think, therefore I am").

Source: Wikimedia Commons

We could use our new-found knowledge of the `struct` and `typedef` commands to make it easy to write programs that deal with vectors. All we need to do is define a new variable type that's designed to hold a vector's three cartesian components.

```
typedef struct {
  double x;
  double y;
  double z;
} Vector;
```

We can then define variables that have the new type `Vector`. For example, we might define a vector named "`velocity`", and give it a magnitude of 60 mph along the x axis:

```
Vector velocity;

velocity.x = 60;
velocity.y = 0;
velocity.z = 0;
```

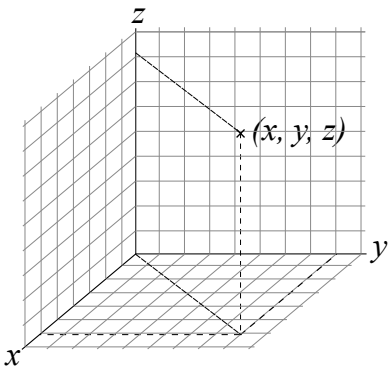


Figure 12.1: The cartesian coordinate system.

Source: Wikimedia Commons

When initializing a "structure" variable, we can also take advantage of the following shortcut:

```
Vector velocity = {60,0,0};
```

The items listed in the curly brackets are the initial values for each of

the structure's properties, in the same order they appear in the `struct` statement.

Now that we can define our own variable types, we can write functions that return more information. In C, functions can only return one "value", but that value can be a structure. Program 12.1 shows a few examples of this, with our new variable type highlighted. Notice that we can use our new type for arguments to our functions or the values the functions return.

The `add_vector` function defined in Program 12.1 adds two vectors together and returns their sum as a third vector. The `scale_vector` function "scales" a vector by multiplying its magnitude by some amount but leaving its direction unchanged. The `print_vector` function just prints out a vector's components.

We could generalize our vector functions to any number of dimensions, and reduce the amount of repetitive typing in them, by replacing the x, y, and z components with a three-element array of components:

```
const int dimension = 3;

typedef struct {
  double x[dimension];
} Vector;
```

Instead of x, y, and z, we would then use `x[0]`, `x[1]`, and `x[2]`. A function like `scale_vector` would then look like this:

```
Vector scale_vector ( double r, Vector v ) {
  Vector vscale;
  int i;
  for ( i=0; i<dimension; i++ ) {
          vscale.x[i] = r * v.x[i];
  }
  return ( vscale );
}
```

This would also require that we change the way we initialize our vectors:

```
  Vector v1 = {{0,0,1}};
  Vector v2 = {{1,0,0}};
```

(Notice the double curly brackets.) This says that the first (and only, in this case) property of v1 is an array containing the values 0, 0, and 1. Compare this to our earlier case with x, y, and z properties.

Program 12.1: vector.cpp

```
#include <stdio.h>
#include <math.h>

typedef struct {
  double x;
  double y;
  double z;
} Vector;

Vector scale_vector ( double r, Vector v ) {
  Vector vscale;
  vscale.x = r * v.x;
  vscale.y = r * v.y;
  vscale.z = r * v.z;
  return ( vscale );
}

Vector add_vectors ( Vector v1, Vector v2 ) {
  Vector sum;
  sum.x = v1.x + v2.x;
  sum.y = v1.y + v2.y;
  sum.z = v1.z + v2.z;
  return ( sum );
}

void print_vector ( Vector v ) {
  printf ( "x = %lf\n", v.x );
  printf ( "y = %lf\n", v.y );
  printf ( "z = %lf\n", v.z );
}

int main () {

  Vector v1 = {0,0,1};
  Vector v2 = {1,0,0};

  printf ( "Vector 1:\n" );
  print_vector( v1 );

  printf ( "Vector 2:\n" );
  print_vector( v2 );

  printf( "Sum of vectors:\n" );
  print_vector( add_vectors( v1, v2 ) );

  printf ( "Pi times Vector 1:\n" );
  print_vector( scale_vector( M_PI, v1 ) );
}
```

Exercise 58: What's Your Vector Victor?

Starting with `vector.cpp`, modify the program so that it prompts the user for the x, y, z components of two vectors, then prints out the components of the sum of the two vectors.

But what about...?

When we studied arrays in Chapter 6 and character strings in Chapter 8 we found that we couldn't just make two arrays equal by saying "array1 = array2", or compare arrays the way we'd compare single variables. Are there similar restrictions on structures?

First, regarding setting structures equal, there's good news. The following will work fine:

```
typedef struct {
    double width;
    double height;
} Rectangle;

Rectangle r1, r2;

r1.width = 8.5;
r1.height = 11.0;

// Make r2 equal r1:
r2 = r1;
```

Another easy shortcut for setting the value of a structure is this:

```
// Set the width and height of rectangle r1:
r1 = (Rectangle){8.5,11.0};
```

Regarding the comparison of two structures, the answer is the same as for arrays. The usual comparison operators (==, <, >, *et cetera*) won't work. You'll need to compare the properties of your structures yourself. For example, with our rectangles above we might write:

```
if ( r1.width == r2.width && r1.height == r2.height ) {
   printf ( "They're equal!\n" );
}
```

12.5. Gravitation

Okay, now we have vectors so let's do something with them. Imagine you have two masses, say the Earth and the Sun. Newton's Law of Gravitation tells us that each of these bodies will attract the other with a force whose magnitude is given by:

$$F = G\frac{m_{earth}m_{sun}}{r^2} \tag{12.1}$$

where m_{earth} and m_{sun} are the masses of the bodies, and r is the distance between them. G is Newton's gravitational constant, equal to about $6.67 \times 10^{-11}\frac{m^3}{kgs^2}$. Earth and Sun pull on each other with equal force, but in opposite directions.

Isaac Newton's grave in Westminster Abbey.

Source: Wikimedia Commons

In principle, if we know a body's initial position and velocity, its mass, and the forces acting on it, we can predict its future motion. Could we simulate the motion of the Earth and the Sun? Let's try! In the following, we won't spend much time discussing the physics of the problem. We'll focus on the programming challenges it presents.

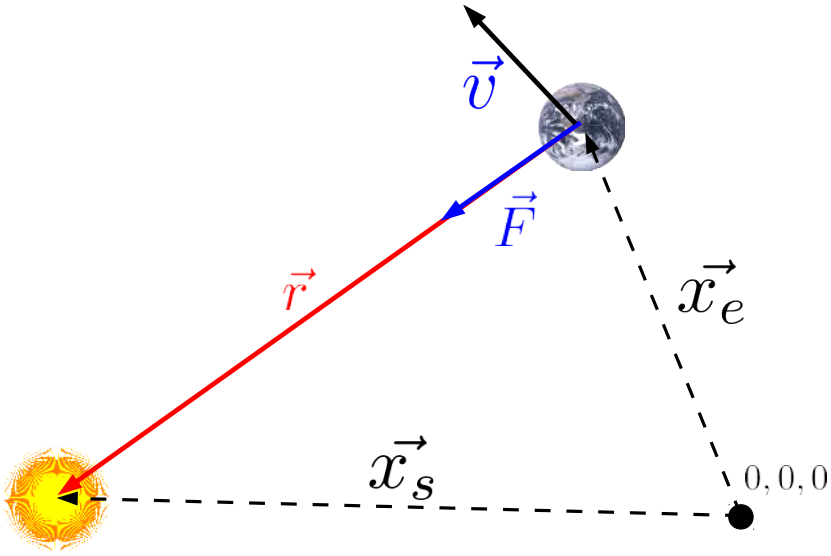


Figure 12.2: The earth-sun system, showing some of the vector quantities we might want to use in our program. $\vec{F}$ represents the force of the sun on the earth. The origin of our cartesian coordinate system is shown in the lower right corner. The vectors x_e and x_s represent the position of the earth and sun in this coordinate system.

First, we'll need a little more information from Newton. He also tells us that force and acceleration are related in this way:

$$\vec{F} = m\vec{a} \tag{12.2}$$

where $\vec{F}$ is a vector representing the magnitude and direction of the

total force on an object, m is the object's mass, and $\vec{a}$ is the object's acceleration. In order to predict the motion of an object, we'll need to know its acceleration. We can get this by rearranging Equation 12.2:

$$\vec{a} = \frac{\vec{F}}{m} \tag{12.3}$$

Next, we'll need to know how acceleration affects an object's motion. Acceleration is the rate of change of velocity, so we might guess that after a small amount of time, Δt, the object's velocity will change by $\vec{a}\Delta t$. We also know that velocity is the rate of change of position, so we might approximate the change of position during a short time as $\vec{v}\Delta t$.

Given this chain of relationships, we can start with an object's initial position and velocity, then move forward in time by small steps and follow the changes in the object's position.

Both the Earth and the Sun move in response to their mutual gravitational attraction. We'll be tracking several properties of each of these objects: mass, velocity, position, and the force acting on the object. This sounds like a good place to use C's structures.

Here are the structures we'll be using, with convenient "`typedef`" names given to them:

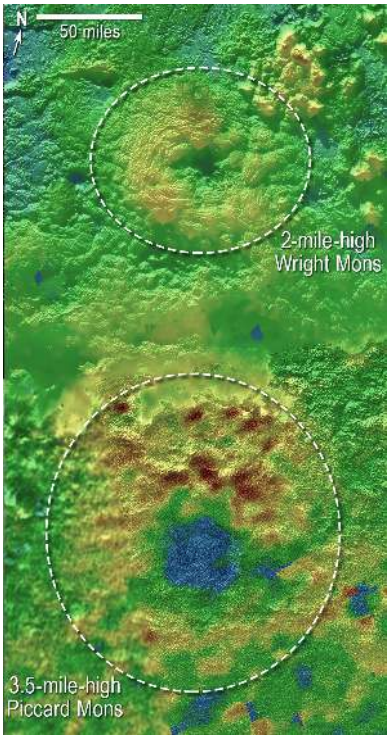
```
const int dimension = 3;

typedef struct {
  double x[dimension];
} Vector;

typedef struct {
  double mass;
  Vector x; // Position
  Vector v; // Velocity
  Vector f; // Force on the body
} Body;
```

As before, we've defined a new type of variable called `Vector` to hold vectors. Now we add a variable called `Body` that holds information about one of the bodies we'll be tracking. Notice that the `Body` type has several properties that are of the `Vector` type.

In addition to the vector functions we've already written, we'll need a



The photo above shows a false-color image of two icy "cryovolcanos" on Pluto. After nearly ten years in space, NASA's *New Horizons* space probe whizzed past Pluto during a few hours in 2015, snapping thousands of photos and taking measurements of many kinds. This kind of precise navigation demands highly sophisticated computer simulations.

Source: NASA/JHUAPL

few others:

```
Vector invert_vector ( Vector v ) {
  Vector inverse;
  int i;
  for ( i=0; i<dimension; i++ ) {
    inverse.x[i] = -v.x[i];
  }
  return ( inverse );
}

Vector subtract_vectors ( Vector v1, Vector v2 ) {
  return ( add_vectors( v1, invert_vector(v2) ) );
}

double vector_magnitude ( Vector v ) {
  double size2=0;
  int i;
  for ( i=0; i<dimension; i++ ) {
    size2 += v.x[i]*v.x[i];
  }
  return ( sqrt( size2 ) );
};
```

The function `subtract_vectors` is the companion to the `add_vectors` function we saw earlier. Subtraction is just equivalent to adding an inverse, so we implement the subtraction function by defining a new `invert_vector` function and then using it along with `add_vectors` to do the subtraction. Finally, we'll need to know the size of vector quantities, so we define a new function named `vector_magnitude` to do this.

Armed with all of these new tools, we're now ready to tackle the weighty problem of swinging the Earth around the Sun. The result is Program 12.2. Here we've swept all of the functions we've discussed so far into a header file named `gravity.h`.

The program steps through time in 10,000-second jumps (about 3 hours). In each "time slice" the program updates the position and velocity of Earth and Sun, based on the force of their mutual gravitational attraction, then writes out the current time and the position of each body.

Program 12.2: gravity.cpp

```
#include <stdio.h>
#include <math.h>

#include "gravity.h"

int main () {
  // All units in kilograms, meters, kilograms, and seconds.
  //            mass     position         velocity        force
  Body sun   = {2.0e+30, {{0,0,0}},       {{0,0,0}},      {{0,0,0}}};
  Body earth = {1.5e+24, {{1.5e+11,0,0}}, {{0,0,3.2e+4}}, {{0,0,0}}};

  double distance, force, deltat=1e+4; // About 3 hours.
  int i, nsteps = 10000;
  double G = 6.67e-11;
  Vector r, deltax, deltav;

  for ( i=0; i<nsteps; i++ ) {
    // Find forces from law of gravitation:
    r = subtract_vectors( earth.x, sun.x );
    distance = vector_magnitude( r );
    force = G*sun.mass*earth.mass/(distance*distance);
    sun.f = scale_vector ( force/distance, r );
    earth.f = invert_vector( sun.f );

    // Update positions and velocities for next step:

    // Sun
    deltax = scale_vector ( deltat, sun.v );
    sun.x = add_vectors( sun.x, deltax );
    deltav = scale_vector ( deltat/sun.mass, sun.f );
    sun.v = add_vectors( sun.v, deltav );

    // Earth
    deltax = scale_vector ( deltat, earth.v );
    earth.x = add_vectors( earth.x, deltax );
    deltav = scale_vector ( deltat/earth.mass, earth.f );
    earth.v = add_vectors( earth.v, deltav );

    // Write out current time and positions
    printf ("%lf ", deltat*(double)i );
    print_vector ( sun.x );
    print_vector ( earth.x );
    printf ("\n");
  }
}
```

If we save the program's output into a file we can plot the results with *gnuplot*. Some of the results are shown in Figures 12.3 and 12.4. As you can see, our simulation gives earth a rather rough ride. The orbit is approximately regular, but doesn't come back quite to the place it started. If we asked the program to simulate more time steps the orbit would eventually become a spiral.

By looking at the period of Earth's movement along the X-axis of our coordinate system, we can estimate the length of Earth's year. An actual year is about 3×10^7 seconds long, but our simulated Earth has a rather longer year of about 4.5×10^7 seconds.

Don't use this program to navigate your space probe to Pluto! We could probably improve the program in several ways. We could use more precise values for the mass, distance, and initial velocity of the Sun and Earth, for example. Still, for a relatively simple program the results aren't too bad.

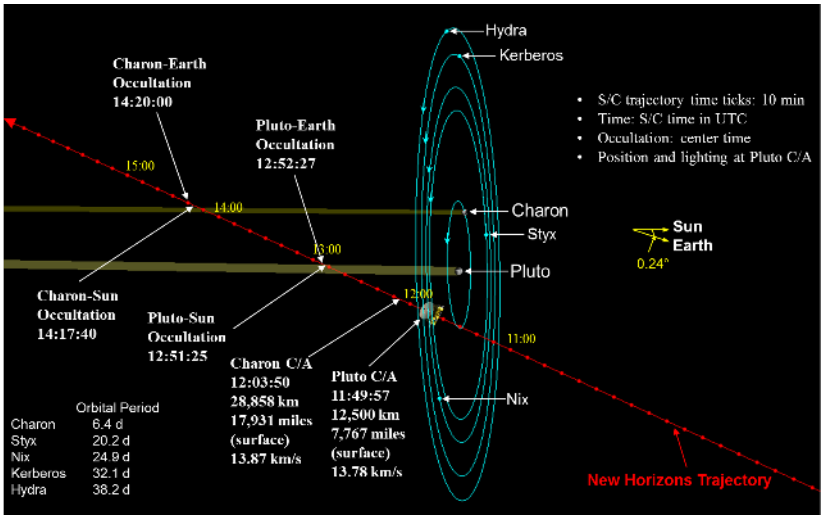


Figure 12.5: New Horizons' trajectory through the Pluto system is a stunning example of precise navigation.

Source: NASA/JHUAPL

But what about...?

Yow! that's all very impressive, but those vector function things are really hard to read:

```
sun.x = add_vectors( sun.x, deltax );
```

Wouldn't it be so much nicer if you could just write "`sun.x = sun.x + deltax`"? If you use the extra features of C++, you can!

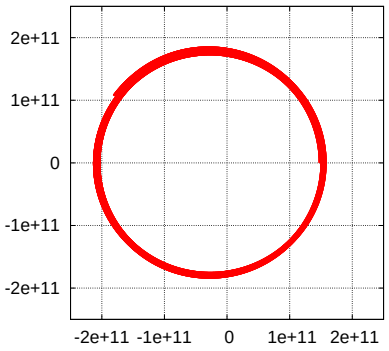


Figure 12.3: The Earth's orbit as approximated by Program 12.2.

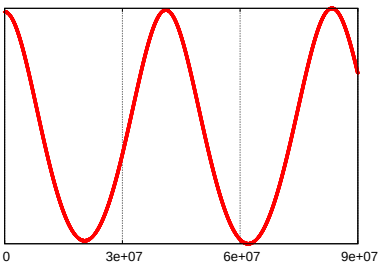


Figure 12.4: The Earth's position on the X-axis as a function of time, as approximated by Program 12.2.

C++ has extra capabilities beyond those of plain C. One of them is called "operator overloading". This allows you to define how operators like + and − act when used with structures.

All we need to do is add the following to the `gravity.h` file used by Program 12.2:

```
Vector operator+( Vector v1, Vector v2 ) {
  return ( add_vectors ( v1, v2 ) );
}

Vector operator-( Vector v1, Vector v2 ) {
  return ( subtract_vectors ( v1, v2 ) );
}
```

This tells the compiler that, when it sees you adding two vectors in an expression like "`vsum = v1 + v2`", it should pass `v1` and `v2` to your `add_vectors` function and use that to add the vectors. The "`operator-`" statement does the analogous thing for subtraction.

12.6. Complex Numbers

Like vectors, another natural use for structures is the representation of complex numbers. These are numbers of the form $a + ib$, where a and b are real numbers, and i is $\sqrt{-1}$. The value of a is called the "real part" of the complex number, and b is its "imaginary part".

Using `struct` and `typedef` we can define a new variable type for holding complex numbers:

```
typedef struct {
  double re;  // real part
  double im;  // imaginary part
} Complex;
```

The "magnitude" of a complex number represents its size, taking both of its components (real and imaginary) into account. The magnitude is just the same as though real and imaginary components were the x and y components of a cartesian vector: magnitude $= \sqrt{\mathrm{Re}^2 + \mathrm{Im}^2}$. Similarly, complex numbers add in the same way that 2-dimensional vectors add.

Strangeness enters the picture when we begin multiplying complex

numbers. Since they involve multiples of i, and $i \times i = -1$, minus signs begin to appear in surprising places.

We could use our new variable type to define a few functions for operating with complex numbers:

```
double magnitude_complex( Complex z ) {
  return sqrt( z.re*z.re + z.im*z.im );
}

Complex multiply_complex ( Complex a, Complex b ) {
  Complex result;
  result.re = a.re*b.re - a.im*b.im;
  result.im = a.im*b.re + a.re*b.im;
  return ( result );
}

Complex add_complex ( Complex a, Complex b ) {
  Complex result;
  result.re = a.re + b.re;
  result.im = a.im + b.im;
  return ( result );
}
```

Mathematician Robert W. Brooks, who along with Peter Matelski discovered the Mandelbrot set in 1978. The set was later named in honor of Benoit Mandelbrot, who studied it extensively.

Source: Wikimedia Commons

12.7. The Mandelbrot Set

Mathematicians love to play games with numbers. Let's try one here. Here are the rules:

Take two numbers, c and z_0. Pick any number you want for c, but set z_0 equal to zero. Now write down the value of $z_0^2 + c$. Call this new number z_1. Now write down the value of $z_1^2 + c$. Call this z_2. Keep doing this for more and more z_n values. We might expect that each z would be bigger than the last.

We coult write a little program to test this. Let's pick $c = 1$ as the c value and see what happens:

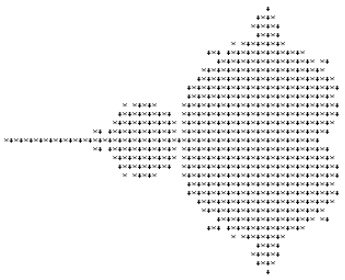


Figure 12.6: Brooks and Matelski's first published picture of the Mandelbrot set. We can do better than that!

Source: Wikimedia Commons

Program 12.3: mandelseries.cpp

```
#include <stdio.h>
#include <math.h>

int main () {
  double c=1;
  double z=0;
  int i;

  printf ("For c = %lf:\n", c );
  for ( i=0; i<10; i++ ) {
    printf ( "z %d = %lf\n", i, z );
    z = pow(z,2) + c;
  }
}
```

The output of this program would be:

```
For c = 1.000000:
z 0 = 0.000000
z 1 = 1.000000
z 2 = 2.000000
z 3 = 5.000000
z 4 = 26.000000
z 5 = 677.000000
z 6 = 458330.000000
z 7 = 210066388901.000000
z 8 = 44127887745906175377408.000000
z 9 = 1947270476915296285689291011464375055838871552.000000
```

Wow! We were right. The numbers get big pretty quickly. But is this true for all values of c? Let's try a negative number and see what happens. How about $c = -1$?:

```
For c = -1.000000:
z 0 = 0.000000
z 1 = -1.000000
z 2 = 0.000000
z 3 = -1.000000
z 4 = 0.000000
z 5 = -1.000000
z 6 = 0.000000
```

```
z 7 = -1.000000
z 8 = 0.000000
z 9 = -1.000000
```

Hmm. It just oscillates back and forth between zero and one, and never gets any bigger. Is this true for all negative numbers? No, if we try $c = -2$ we'll find that, after one flip, all of the rest of the values are equal to 2! If we use $c = -3$, though, we'll see that the numbers once again blow up, and become very large very quickly. This is intriguing![2]

[2] If you're a mathematician.

What if we extended this to complex numbers? Would they be even weirder? Yes, indeed they would!

In 1978 two mathematicians, Robert W. Brooks and Peter Matelski, tried this and discovered what's known today as the "Mandelbrot Set". It has manyfascinating qualities, including the fact that its boundary is infinitely rough. If you zoom in on most common-or-garden-variety shapes, you'll find that sooner or later you just see smooth surfaces or curves. Not so with the Mandelbrot set. This shape is equally squiggly at every scale. The Mandelbrot Set's boundary is so squiggly that it behaves as something more than a 1-dimensional curve. We call such shapes "fractals", because they appear to have "fractional" dimensionality.

Approximate fractal shapes often appear in nature. This piece of broccoli is a good example.
Source: Wikimedia Commons

Frost patterns on a window also exhibit fractal behavior.
Source: Wikimedia Commons

Figure 12.6 shows Brooks and Matelski's first illustration of the Mandelbrot set. It shows the "complex plane", where complex numbers are plotted with their real part on the x axis and their imaginary part on the y axis. This graph uses ASCII characters to indicate the c values on the complex plane which *don't* cause the series to blow up. (We call these c values "stable".) This graph was produced in 1978, when computer technology was much less capable than it is now. We should be able to make a much better illustration using the computers available to us in the 21^{st} century.

Program 12.4 is the result. It uses the `Complex` variables we defined in the preceding section. It uses a header file named `complex.h` which contains the `typedef` statement and function definitions we wrote earlier for dealing with complex numbers.

The program divides the complex plane into a 250 × 250 grid. Each point on the grid represents a complex number, c, that we'll test to see if it makes the "Mandelbrot series[3]" blow up. The program takes advantage of something mathematicians have proven about the Mandelbrot series: if a complex number has a magnitude greater than 2, we

[3] See Program 12.3.

know for sure that it will cause the series to blow up. This means we only need to look at a region within a distance of 2 from the origin. As soon as our series wanders outside of this region, we we know that it will blow up.

The heart of the program is the function `mandel_test` which we'll use to test each value of c. It calculates up to 100 terms of the Mandelbrot series for this value. If one of the terms wanders more than a distance of 2 away from the origin, we know this c value isn't stable, and we can stop calculating terms. If we make it all the way to 100 terms without becoming unstable, we assume that c is stable. The function just returns the number of terms before instability was detected, or 100 if we made it through all 100.

Each time we test a value of c, we write out its real and imaginary parts, and the number of terms returned by the `mandel_test` function, into a file named `mandel.dat`.

The resulting file will have three columns of numbers: Real part, imaginary part, and number of terms. We can ask *gnuplot* to read this file and interpet it as an image, with the first two columns giving the coordinates of each pixel, and the third column giving its color. We do this by giving *gnuplot* the command "`plot "mandel.dat" with image`". You can see the result in Figure 12.8. Beautiful!

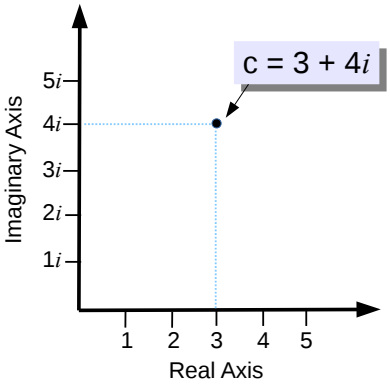


Figure 12.7: The complex number $c = 3 + 4i$ located on the complex plane.

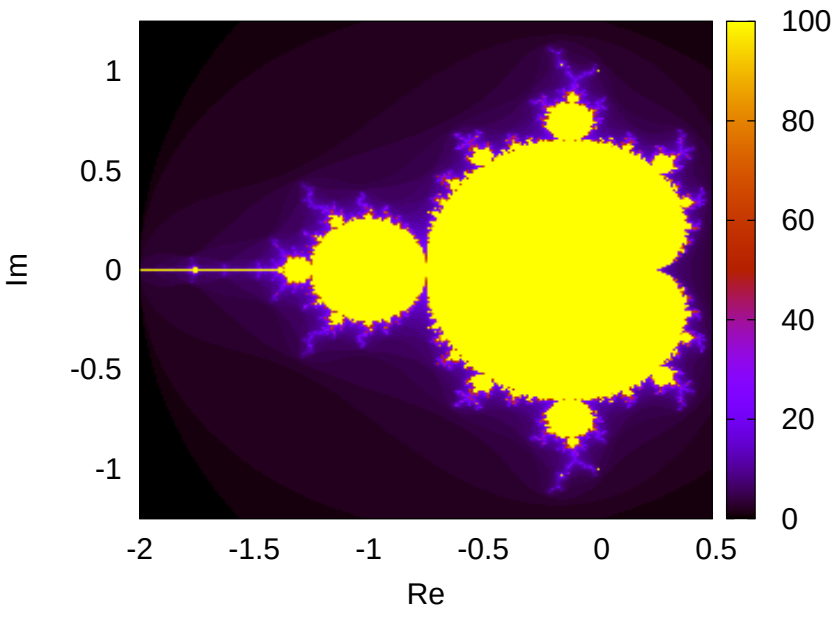


Figure 12.8: The Mandelbrot set, generated by Program 12.4 and visualized by *gnuplot*. See Wikipedia for much more information about this fascinating and beautiful structure.

Exercise 59: Fun with Fractals

Create `mandel.cpp` and `complex.h`. Compile and run `mandel.cpp`, then plot your results with *gnuplot*.

Now try modifying the program by changing the x and y limits in `main`. Make x go from -0.76 to -0.75, and y go from 0.04 to 0.06. This will zoom in on the edge of the circle on the left-hand side of the graph. The result should look like Figure 12.9.

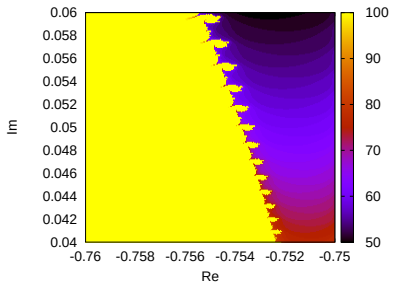


Figure 12.9: Zooming in on the edge of the large circle on the left-hand side of the graph.

12.8. Growing Domains

Let's briefly go away from structures and look at a problem that just uses good ol' `double` values.

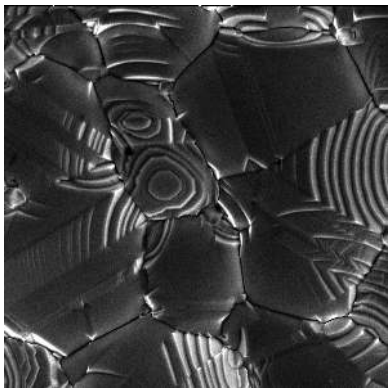
Imagine that we have a just-plowed 100 × 100 field of barren earth. Over time, a few seeds of two different species are blown onto the field. The seeds germinate, grow, and begin to reproduce, spreading each species outward from the sites where the initial seeds fell. The two species are incompatible, so new seeds won't grow in land already occupied by the other species.

This kind of problem is very common in science. It doesn't have to be the seeds of plants we're talking about. It could be two different crystal structures crystallizing out of a solution, it could be magnetic domains growing in a magnet, or it could be the expansion of human settlements in a formerly unoccupied territory.

Program 12.5 simulates the situation we've described. It defines the 2-dimensional array `color[100][100]` that will hold a color for each sqare area of the field. The color tells us which species is living in that area. The colors will just be three numbers: 0 means the square is empty, 1 means that species number 1 has colonized this area, and 2 means the same for species number 2.

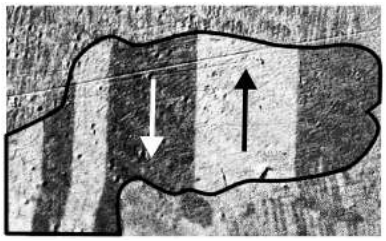
The program uses the `rand01` function we wrote in Chapter 9. At the beginning, the elements of `color` are initialized to one of the three colors, based on random "dice rolls" made using `rand01`.

In the middle of the program, we start looping through a large number (1 million) of "turns". During each turn, the program uses `rand01`



Silver crystals growing on a ceramic substrate. Note the different crystal domains that grow and bump into each other.

Source: Wikimedia Commons



Magnetic domains in a piece of steel.

Source: Wikimedia Commons

Program 12.4: mandel.cpp

```
#include <stdio.h>
#include <math.h>

const int NTRIALS=100;

#include "complex.h"

int mandel_test( Complex c ) {
  Complex z = c;
  int counts = 0;
  while ( magnitude_complex( z ) <= 2.0
&& counts<NTRIALS ) {
    counts++;
    // z -> z^2 + c
    z = add_complex( multiply_complex(z,z), c );
  }
  return counts;
}

int main(){
  double xmin = -2.0;
  double xmax = 0.5;
  double ymin = -1.25;
  double ymax = 1.25;
  Complex c;
  int nim,nre, counts;
  const int NSTEPS = 250;

  FILE *outp = fopen("mandel.dat","w");

  for (nre=0; nre<NSTEPS ; nre++) { // loop over real axis
    c.re = xmin + (xmax-xmin) * nre/NSTEPS;
    for (nim=0; nim<NSTEPS; nim++) { // loop over imaginary axis
      c.im = ymin + (ymax-ymin) * nim/NSTEPS;
      counts =  mandel_test(c);
      fprintf(outp,"%lf  %lf  %d\n",c.re,c.im,counts);
    }
  }
  fclose(outp);
  return 0;
}
```

to pick a random element of `color`. It then randomly picks another element one space left, right, up, or down from that element. If this second element is empty (that is, its color is 0), the second element's color is set equal to the first element. The first element has "colonized" the second.

To pick a random direction, we make use of a new operator that we haven't seen before. This is C's "ternary" operator. Most of the operators in C, like `+`, `-`, `*`, `/`, *et cetera*, work on one or two values. The ternary operator is the only operator in C that uses three values. It acts like an abbreviated "`if else`" statement, and is indeed exactly equivalent to this. It's just shorter to write.

The syntax of the ternary operator is:

```
condition ? do this if true : do this if false
```

The statement above is exactly equivalent to:

```
if ( condition ) {
   do this if true
} else {
   do this if false
}
```

At the end of Program 12.5 the elements of `color` are printed out in a particular way. We can think of a 2-dimensional array like `color` as being a matrix of some number of rows and some number of columns. In Program 12.5 we print out the array elements in just this way. Each line of the output corresponds to a row of the matrix, and the number of lines is equal to the number of rows.

If we run the program, directing it output into a file like this "`./domain > domain.dat`", we can plot the results with *gnuplot*. We've written the program's output as a matrix of values, which is different from the kinds of files we've asked *gnuplot* to read before. That's OK, though. We just need to let *gnuplot* know that the file is in this format. Here's how to do that:

```
plot "domain.dat" matrix with image
```

The word `matrix` tells *gnuplot* that the file is in the form of an $n \times m$ matrix with a newline at the end of each row.

If we modified the program so that it just showed us the initial dis-

Program 12.5: domain.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}

int main () {
  int color[100][100];
  int i,j,n,direction,inew,jnew,t;
  double roll;

  // Initialize:
  for ( i=0; i<100; i++ ) {
    for ( j=0; j<100; j++ ) {
      roll = rand01();
      if ( roll < 0.10 ) {
        color[i][j] = 1;
      } else if ( roll < 0.20 ) {
        color[i][j] = 2;
      } else {
        color[i][j] = 0;
      }
    }
  }

  // Take turns:
  for ( t=0; t<1000000; t++ ) {
    i = 1.0 + 98.0*rand01();
    j = 1.0 + 98.0*rand01();

    rand01() < 0.5 ? direction=0 : direction=1;
    rand01() < 0.5 ? n=0 : n=1;
    if ( direction == 0 ) {
      inew = i-1+2*n;
      jnew = j;
    } else {
      inew = i;
      jnew = j-1+2*n;
    }

    if ( color[inew][jnew] == 0 ) {
      color[inew][jnew] = color[i][j];
    }
  }

  // Write results:
  for ( i=0; i<100; i++ ) {
    for ( j=0; j<100; j++ ) {
      printf ("%d ", color[i][j] );
    }
    printf ("\n");
  }
}
```

tribution of seeds, the output would look like the left-hand graph in Figure 12.10. The unaltered program would show us the distribution of species after 1 million turns. That would look like the right-hand graph Figure 12.10.

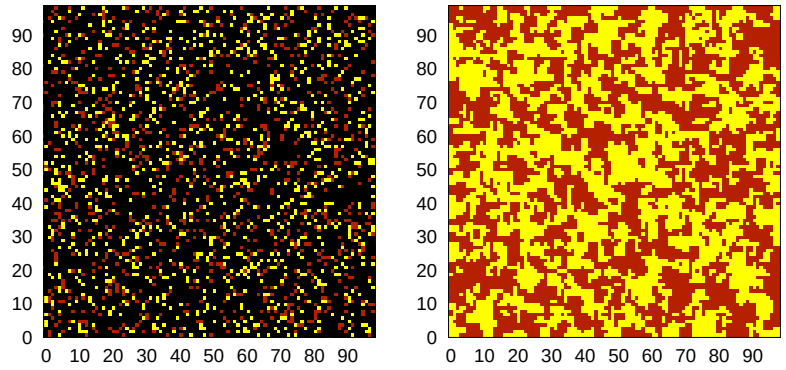


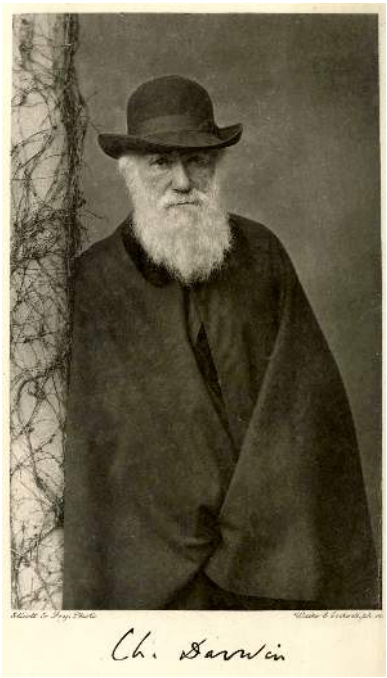
Figure 12.10: Initial distribution of two species in the beginning (left) and after some time has passed (right). Black represents uncolonized spaces.

12.9. Simulating Evolution

In the preceding example, each element of our array has only one property (color). What if the elements had more properties? We could store all of the properties by using an array of structures instead of an array of `int`s.

To illustrate this, let's assume that our two species are small animals that can compete with each other for resources. Each element of our simulation array is a small habitat that can contain a family of these animals, and let's follow the members of each species over a long time and watch them spread and respond to natural selection.

In 1859 Charles Darwin published his *Origin of Species*. Both he and Alfred Russel Wallace had hit, more or less simultaneously, on the idea of "evolution by natural selection". This theory says that evolution occurs because of three factors:



Charles Darwin, who developed the theory of evolution by natural selection, as described in his 1859 book *The Origin of Species*.

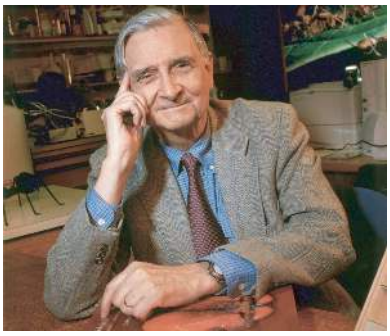
Source: Wikimedia Commons

- Inheritance of characteristics. (Individuals tend to pass along some of their characteristics to their offspring.)
- Variability. (Offspring aren't identical to their parents, due to random variations.)
- Natural Selection. (Some characteristics make individuals who possess them more likely to have offspring, either because they make the individual more long-lived, more competitive for resources, more fertile, or through other mechanisms.)

It's not uncommon for larger animals to have an advantage over smaller ones, so let's keep track of the average size of the individuals in each of our small habitats. If the family in one habitat tries to take over an occupied neighboring habitat, we'll assume that the side with bigger individuals will win.

There are also disadvantages to being larger, though. Larger animals tend to reproduce more slowly. This means that it should take more "turns" in our program for larger animals to colonize a new habitat.

A second disadvantage comes from the environment itself. A given area has limited food, water, and other resources. Larger individuals take more resources. If our individuals got too big, they'd be like elephants in a small back yard. There just wouldn't be enough food to support them and they'd eventually die. Mice, on the other hand, could thrive in the same environment.



If you're interested in this kind of thing, you should read a little book called *The Theory of Island Biogeography* by Robert H. MacArthur and Edward O. Wilson (pictured above). If you're even more interested, you should read Wilson's massive tome titled *Sociobiology: The New Synthesis*.

Source: Wikimedia Commons

With these considerations in mind, let's create a structure that could represent each of our array elements. It might look like this:

```
typedef struct {
  int species;  // Which species occupies?
  double size;  // Avg. size of individuals.
  int lastturn; // Last time this family tried expanding.
  double capacity; // Capacity of this habitat.
} Habitat;
```

The structure above records the identity of the species occupying this habitat. This will just be a number: 0, 1, or 2, as in our preceding example. Then it records the average size of the individuals who live in this habitat. As we'll see later, we'll assume that `size` is some measure of the individuals' height. The property `lastturn` records the last time these individuals tried to colonize a neighboring habitat. We'll use this to allow for the fact that it takes larger individuals longer to reproduce. Finally, `capacity` tells us the maximum size of indidviduals that can thrive in this habitat.

We might define a 100 × 100 array of such structures like this:

```
Habitat h[100][100];
```

It'll be convenient to have some functions for dealing with these structures, so let's create a header file that contains these. It might look like "Program" 12.6. You'll see our old friends `rand01` and `normal` from Chapter 9, as well as some new things specific to this program.

The function `init_habitats` sets up the initial conditions by choosing a random species (or no species) for each element of the array. It also sets the size of the inhabitants of this element. It assumes that, intially, the size of all individuals of any species is about "1" (in some arbitrary units), but that there's about a 10% variability between individuals. The function uses our `normal` function to generate the random variations. `init_habitats` also sets the "capacity" of each element to 50. If the size of the individuals in this habitat exceeds this value, bad things will begin to happen for them.

The function `dumpsnapshot` will be used to dump out a "snapshot" of the conditions in our field every once in a while, so we can see how things are progressing. It writes out a file with a name like "`habitat-nn.dat`", where "nn" is number we give `dumpsnapshot`. The file is in the same "matrix" format we used in Program 12.5. The function `meansize` calculates the mean size of the individuals in a given species. As we'll see, this will change as time progresses. This function will let us track those changes.

Program 12.7 uses these structures and functions to actually do our simulation. After calling `init_habitats`, it launches into a loop of 50,000,000 turns. In each turn, the program behaves similarly to Program 12.5. One difference appears in the next-to-last "`if`" statement, which no longer just checks to see if the neighboring element is unoccupied. Now, even if the neighbor is already occupied, it will still be taken over if the its occupants are smaller.

In the final "`if`" statement, we enforce a wait period after we've taken over an element. We're not allowed to take over another element until the wait period has passed. The wait period is calculated from our size. If the size is larger, the wait period is longer (simulating longer gestation periods for larger animals). We assume that the wait is proportional to the mass of individuals. Since we said that `size` was a measure of their height, we assume that their mass is proportional to `size` cubed.

Earlier in the program, after we've picked a random element, we check to see if the size of the individuals in that element has exceeded the element's capacity. If so, we assume they die, and set `species` equal to 0 for that element, making it empty and available for colonization.

When we run the program, it will make two kinds of output. First, it will create ten snapshot files, showing the state of our array at ten different times during its evolution. Second, it will periodically print to the screen two numbers, representing the mean size of species 1 and

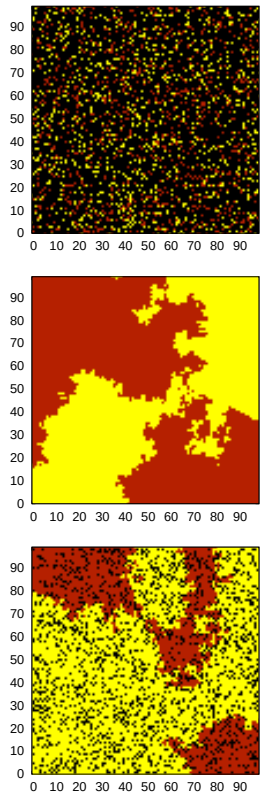


Figure 12.11: Three snapshots in the evolution of our field occupied by two competing species. At the top, we see the initial random distribution. Over time, the families in each of the initial habitats colonize neighboring habitats, perhaps driving out occupants of the other species. The middle snapshot shows an intermediate time, where the species have acheived some kind of equilibrium. Because size is advantageous, natural selection drives members of each species toward larger sizes. In the last snapshot, we see the result when the size of individuals exceeds the capacity of the habitat. Black squares show habitats where colonists have died out due to lack of resources.

When the program calculates `wait` it uses the function `ceil` from C's math library. This function rounds a number up to the nearest integer. There's also a `floor` function, which rounds down to the nearest integer.

Program 12.6: evolve.h

```
double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}

double normal () {
  int nroll = 12;
  double sum = 0;
  int i;
  for ( i=0; i<nroll; i++ ) {
    sum += rand01();
  }
  return ( sum - 6.0 );
}

double meansize ( int species ) {
  int i,j;
  double sum=0;
  int n=0;
  for ( i=0; i<100; i++ ) {
    for ( j=0; j<100; j++ ) {
      if ( h[i][j].species == species ) {
        sum += h[i][j].size;
        n++;
      }
    }
  }
  return( sum/(double)n );
}

void dumpsnapshot (int isnap) {
  FILE *output;
  char filename[100];
  int i,j;

  sprintf (filename,"habitat-%02d.dat",isnap);
  output = fopen( filename,"w" );
  for ( i=0; i<100; i++ ) {
    for ( j=0; j<100; j++ ) {
      fprintf (output, "%d ", h[i][j].species );
    }
    fprintf (output,"\n");
  }
  fclose( output );
}

void init_habitats () {
  int i, j;
  double roll;
  for ( i=0; i<100; i++ ) {
    for ( j=0; j<100; j++ ) {
      roll = rand01();
      if ( roll < 0.10 ) {
        h[i][j].species = 1;
      } else if ( roll < 0.20 ) {
        h[i][j].species = 2;
      } else {
        h[i][j].species = 0;
      }
      h[i][j].size = 1.0 + variability*normal();
      h[i][j].lastturn = 0;
      h[i][j].capacity = 50.0;
    }
  }
}
```

the mean size of species 2.

We can plot the snapshots with *gnuplot* just as we plotted the output of Program 12.5:

```
plot "habitat-00.dat" matrix with image
```

The result will be plots like the ones shown in Figure 12.11.

If you run the program several times, you'll find that the results will vary widely. Sometimes the two species acheive an equilibrium, as in Figure 12.11, but often one species will completly take over the field.

By directing the program's output into a file ("`./evolve > evolve.dat`") we can look at how the mean size of each species varies over time. Figure 12.12 shows the trend in size for one species in one simulation.

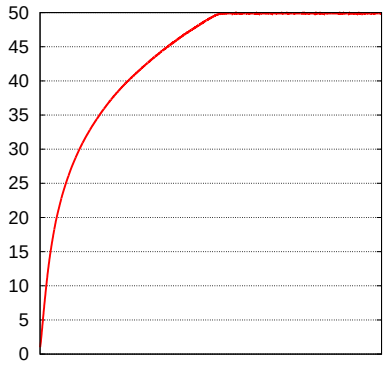


Figure 12.12: Because larger size is favored by natural selection in our model, the mean size of individuals in each species grows over time, until it reaches a plateau at the maximum capacity the habitats can accommodate.

The trend toward larger sizes over time is very common in nature, and is sometimes referred to by evolutionary biologists as "phyletic size increas" or "Cope's Rule" (after palaeontologist Edward Drinker Cope). We're all made familiar with this tendency in childhood, when we first see pictures of tiny early horses like Eohippus (see Figure 12.13).

Figure 12.13: The tiny horse-ancestor, Eohippus, as illustrated by palaeontological artist Charles R. Knight in 1905. Stephen Jay Gould has written an interesting essay about the long history of comparing the size of Eohippus to that of a "fox terrier". You can find it in his collection of essays *Bully for Brontosaurus*.

Source: Wikimedia Commons

Program 12.7: evolve.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

typedef struct {
  int species;
  double size;
  int lastturn;
  double capacity;
} Habitat;

Habitat h[100][100];
double variability=0.1;

#include "evolve.h"

int main () {

  int i,j,n,direction,inew,jnew,t,isnap=0;
  double wait;
  int turns=50000000;

  // Initialize:
  init_habitats();

  for ( t=0; t<turns; t++ ) {

    if ( t%(100*100) == 0 ) {
      printf ( "%lf %lf\n", meansize(1), meansize(2) );
    }
    if ( t%(turns/10) == 0 ) {
      dumpsnapshot( isnap );
      isnap++;
    }

    i = 1.0 + 98.0*rand01();
    j = 1.0 + 98.0*rand01();

    if ( h[i][j].size > h[i][j].capacity ) {
      h[i][j].species = 0;
      continue;
    }

    rand01() < 0.5 ? direction=0 : direction=1;
    rand01() < 0.5 ? n=0 : n=1;
    if ( direction == 0 ) {
      inew = i-1+2*n;
      jnew = j;
    } else {
      inew = i;
      jnew = j-1+2*n;
    }
    if ( h[inew][jnew].species == 0 ||
         h[inew][jnew].size < h[i][j].size ) {
      wait = ceil( pow(h[i][j].size,3) );
      if ( t - h[i][j].lastturn > wait ) {
        h[inew][jnew].species = h[i][j].species;
        h[inew][jnew].size = h[i][j].size + variability*normal();
        h[inew][jnew].lastturn = t;
        h[i][j].lastturn = t;
      }
    }
  }
}
```

12.10. Conclusion

Objects in the real world always have more than one interesting property. C's structures allow us to encapsulate an object's multiple properties in a single variable. As we've seen in this chapter, the `typedef` statement can allow us to simplify our code by defining new variable types using these structures.

If we moved forward into the extra features offered by C++, we's see that structures are the precursor of even more powerful things called "classes". A C++ class incorporates multiple properties as well as a set of functions (called "methods") that are particular to a given type of object.

We've also seen several new computing techniques in this chapter. The techniques we used for dealing with gravitational interactions can be refined and improved to make them suitable for really useful calculations of the orbits of celestial bodies. The techniques we've seen for dealing with interactions between neighboring objects (the domain example and the evolution example) have wide applicability in physics and biology.