6. *Using Arrays*

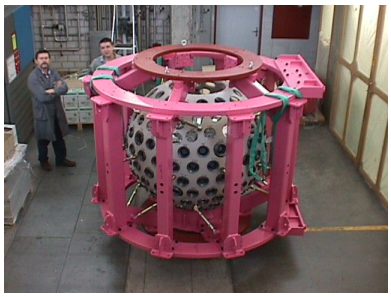
Galileo used his pulse to measure how long it took a ball to reach several marked locations while rolling down a ramp. This experiment established that the distance traveled is proportional to the square of the elapsed time, no matter how much the ball weighs.

Source: Wikimedia Commons

6.1. Introduction

Scientists often make groups of similar measurements under different conditions. We might measure the temperature of a metal bar at several different points along its length for example, or measure the velocity of a dropped ball at several times during its fall. A modern high-energy physics experiment might record the amount of energy deposited in each of hundreds of detectors every time an interesting event is seen.

Programs that analyze data need to store such measurements in variables. We could define one variable for each measurement, giving them names like `t1`, `t2`, `t3` and so forth, but that would be awkward if there were hundreds of measurements. For example, imagine adding them all up: we'd need to write an expression like `t1 + t2 + t3 + ...`, and we'd need to remember to change it if we added or removed any measurements the next time we used the program.



This detector assembly consists of 240 cesium iodide crystals. Each of them measures the energy of particles that pass through the crystal.

Source: PiBeta Collaboration

C provides us with an easier way of storing a group of related values. An "array" is a numbered list of boxes in the computer's memory. The array as a whole has a single name, and individual boxes can be referred to by number. In this chapter we'll see how to create and use arrays.

6.2. A Coal Train

Imagine that you're in charge of a rail system carrying coal. Each train has some number of coal cars, and each car can carry some amount of coal up to a maximum capacity. You'd like to keep track of how much coal is in each car, but you're also interested in the total amount of coal that the train is hauling. How might you store all of those numbers in a program?

A coal train in eastern Wyoming.

Source: Wikimedia Commons

Program 6.1 uses an *array* to store the weight of coal in each car. The array is defined by the statement:

```
double carweight[100];
```

This statement defines an array of one hundred elements[1], each capable of storing a floating-point number. The elements are numbered from zero to 99.

[1] Each "element" of an array is just a storage box for holding something.

We can refer to a particular element of the array by giving its number. For example, if we wanted to print out the value in element number 27 of the array, we could write `printf("%lf",carweight[27]);`. It's very important to remember that the last element in the array is `carweight[99]`, *not* `carweight[100]`. When we define the array, we say how many elements are in it, but the elements are numbered starting with zero, so the last element will always have a number that's *one less than* the total number of elements.[2]

[2] Programmers often refer to an element's number as its "index". Array indices are like the subscripts we use in mathematics when we write an expression like X_i. The index must be an integer, since it just counts the number of elements.

Figure 6.1: The first element of an array is number *zero*.
Source: Openclipart.org

The first loop in Program 6.1 puts a random weight of coal into each of the cars. The weights vary between 50 and 100 tons. In a real-world program, these weights probably wouldn't be random. They might be read out of a file, or they might be read from some kind of device that measures each car's weight as it goes by. This is just an example, though, so we'll use random numbers.[3] Notice that we can set the value of one of the array's elements by referring to it by number.

[3] Since we don't use the *srand* function to change the random number generator's seed, the program will always give us the same set of "random" numbers. (See Chapter 2.)

The program's second loop just prints out the weight of each car in a nice, readable format. Notice that the value of *i* in both loops runs from zero to 99, since the loop starts at zero and continues for as long as *i* is less than 100 (`i<100`).

Program 6.1 also tells us the total amount of coal the train is carrying. The variable `sum` starts with a value of zero, then has the weight of

each car added to it. At the end of the program, the total weight of all cars is printed.

Program 6.1: coal.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main () {
  double carweight[100];
  double w;
  double sum = 0.0;
  int i;

  for ( i=0; i<100; i++ ) {
    w = 50.0 + 50.0 * rand()/(1.0 + RAND_MAX);
    carweight[i] = w;
  }

  for ( i=0; i<100; i++ ) {
    printf ( "Car %d carries %lf tons\n", i, carweight[i] );
    sum += carweight[i];
  }

  printf( "The total weight of coal is %lf tons.\n", sum );
}
```

Exercise 32: "I think I can..."

Create, compile and run Program 6.1. Notice that the car numbers (the array indices) start at zero and end at 99.

Think about how you'd need to change the program to accommodate 200 cars instead of 100. What would be the index of the last car then?

6.3. How Arrays Are Stored

In our programs, a variable is just a temporary storage location in the computer's memory that has a name attached to it. The size of this storage location depends on the type of data we want to put into it. Just as a violin case is different from a trombone case, the box of memory reserved for an `int` variable will be different from the box reserved for a `double` variable.

Figure 6.2 shows how a group of variables might be placed in the computer's memory. Note that `int` and `double` variables require different-sized storage boxes. The data inside these boxes is also organized differently. Because of this, even though `int` data would fit into the space reserved for a `double`, the data would appear garbled when your program tried to read it, because the program would try to interpret these bits as a floating-point number. You might be able to squeeze a violin into a trombone case, but imagine trying to play the violin by blowing into it like a trombone!

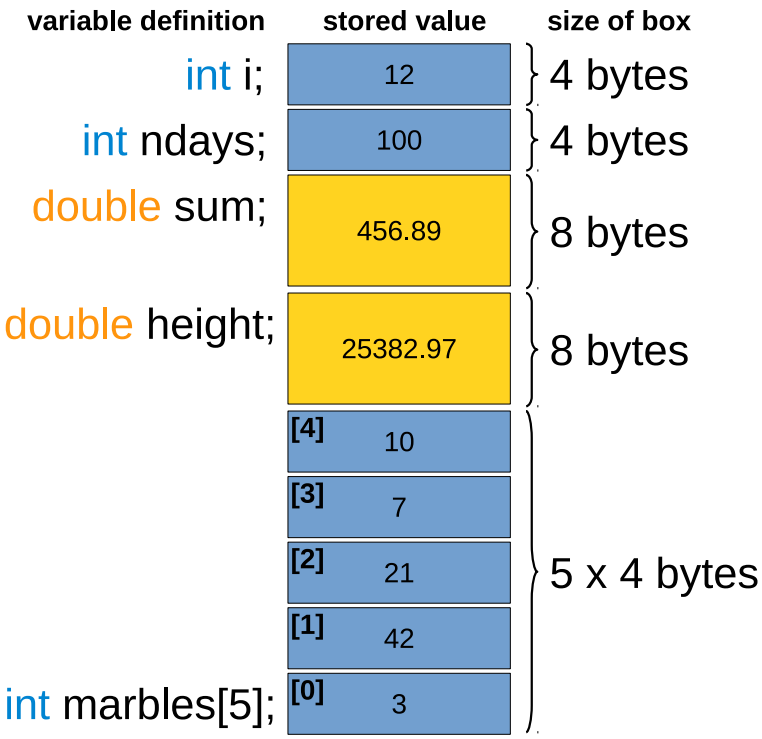


Figure 6.2 also shows how an array is stored. In the figure, a five-element `int` array named `marbles` is defined. Imagine that it records the number of marbles in each of five bags. As you can see, this array takes up the same amount of storage space as five regular `int` variables.

It's important to remember that each element of an array takes up just as much memory as a separate variable of that type. So, if we define a large array with thousands of elements, we may run into the limits of the computer's memory.

The great jazz violinist Stephane Grappelli.

Source: Wikimedia Commons

"Trombone Shorty" (aka Troy Andrews) began playing the trombone before the age of six, when he was so small he had to use his feet to reach the low notes.

Source: Wikimedia Commons

Figure 6.2: How a group of variables might be arranged in the computer's memory. The actual size of `double` or `int` variables may differ depending on the type of computer, operating system, or C compiler. The values shown here are typical, though.

The elements of an array are always stored one after another in the computer's memory. You could think of them as a stack of shoe boxes. In fact, when you ask the computer to find, say, `marbles[3]` it finds the memory address of the first element of `marbles` and then just skips forward by a distance equal to three times the size of a single `int` variable.

All of the elements of an array must have the same type, but this can be `int`, `double`, or any other type that C provides. In our train example, we defined an array of `double` elements called `carweight`. In Figure 6.2 we define an array of `int` elements called `marbles`.

If you have a small array, you might find it useful to set the initial values of the elements when you define the array. For the `marbles` array, for example, we could define and initialize the array by saying `int marbles[5] = {3,42,21,7,10};` The list of numbers in curly brackets will be put into elements zero through five of the array.

But what about...?

Is there a way to find out how much storage space is needed for a type of variable? Yes! You can use the `sizeof` function to find the size of a type, or of a particular variable.

Take a look at this example:

```
#include <stdio.h>
int main () {
  int i;
  double x;

  printf ("Size of int is %d bytes.\n", (int)sizeof( int ) );
  printf ("Size of double is %d bytes.\n", (int)sizeof( double ) );

  printf ("Size of i is %d bytes.\n", (int)sizeof( i ) );
  printf ("Size of x is %d bytes.\n", (int)sizeof( x ) );
}
```

If you ran this program, the output would look something like this:

```
Size of int is 4 bytes.
Size of double is 8 bytes.
Size of i is 4 bytes.
Size of x is 8 bytes.
```

The sizes may be different on your computer, but you can always use `sizeof` to find them if you need them. (Note that we force the value of `sizeof` to be an `int` by putting "`(int)`" in front of it. This is necessary on some computers because the value returned by `sizeof` isn't strictly an `int`.)

6.4. Selecting Array Elements

Let's get back to work on our coal-hauling business. As our train is travelling across the country, we might want to look up the weight of a particular car. Maybe we have a customer in Schenectady who wants at least 85 tons of coal. Will the last car in the train be full enough, or do we need to pick another one?

Program 6.2 adds another section to our earlier program. Now, after the program has listed the weights of all the cars and told us the total weight, it begins waiting for us to enter a car number, and will tell us how much coal is in that particular car.

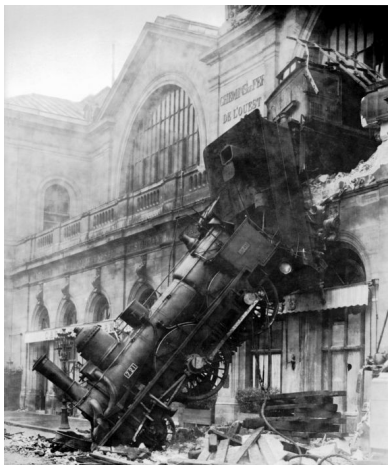
Program 6.2: coal.cpp, Version 2

```
#include <stdio.h>
#include <stdlib.h>
int main () {
  double carweight[100];
  double w;
  double sum = 0.0;
  int i;
  int carno;

  for ( i=0; i<100; i++ ) {
    w = 50.0 + 50.0 * rand()/(1.0 + RAND_MAX);
    carweight[i] = w;
  }

  for ( i=0; i<100; i++ ) {
    printf ( "Car %d carries %lf tons\n", i, carweight[i] );
    sum += carweight[i];
  }

  printf ("The total weight of coal is %lf tons.\n", sum );

  while (1) {
    printf ( "Enter car number (-1 to quit): " );
    scanf ( "%d", &carno );
    if ( carno < 0 ) {
      break;
    }
    printf ( "Car number %d carries %lf tons.\n", carno, carweight[carno] );
  }

}
```

Exercise 33: Runaway Train!

Create, compile and run Program 6.2. Try entering some numbers between zero and ninety-nine. Enter −1 to stop. Do the results look reasonable?

Now try entering `1000` and `1000000`. These values are clearly beyond the end of the train. What does the program do?



Look out! Arrays give us a lot of new abilities, but they also introduce a whole trainload of potential pitfalls to beware of.

Source: Wikimedia Commons

6.5. Checking Array Index Values

Many programs can run simultaneously on a modern computer. To keep programs from interfering with each other, the computer assigns a separate chunk of memory to each program. A program is only allowed to use the memory that belongs to it.

When your coal train program starts running, the computer reserves enough memory space to hold all of the variables you've defined, including the 100 elements of the `carweight` array.[4] However, as demonstrated in the exercise above, the computer *doesn't check your array indices* to make sure they stay within the bounds of the array. This can cause problems if you're not careful when writing your program.

[4] The memory reserved in this way is called "the stack", because it's like a stack of storage boxes, as illustrated in Figure 6.2.

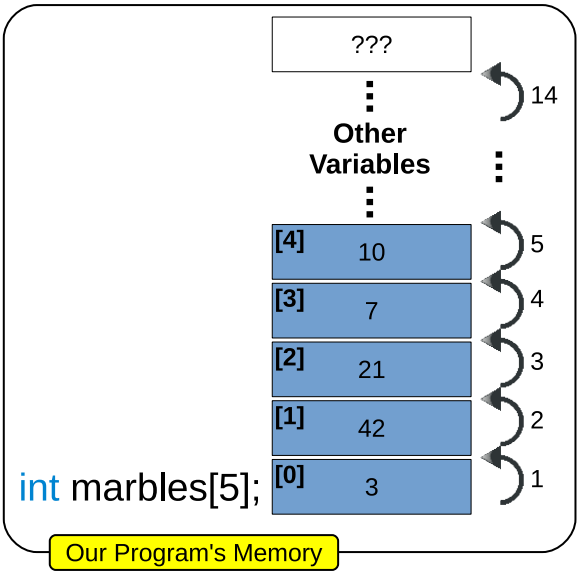
Take a look again at Figure 6.2. If we asked the program to print out the value of `marbles[14]` the computer would happily skip forward 14×4 bytes from the beggining of the `marbles` array, and try to read whatever was at that memory location.

If that part of memory is in the chunk belonging to our program, then the program will be able to successfully read whatever unpredictable value happens to be stored there (see Figure 6.3*a*). If this part of the computer's memory doesn't belong to our program, then the program will crash (see Figure 6.3*b*). Usually, a crash like this generates an error message that says "Segmentation fault". This means that the program has tried to do something in a segment of the computer's memory that doesn't belong to it.

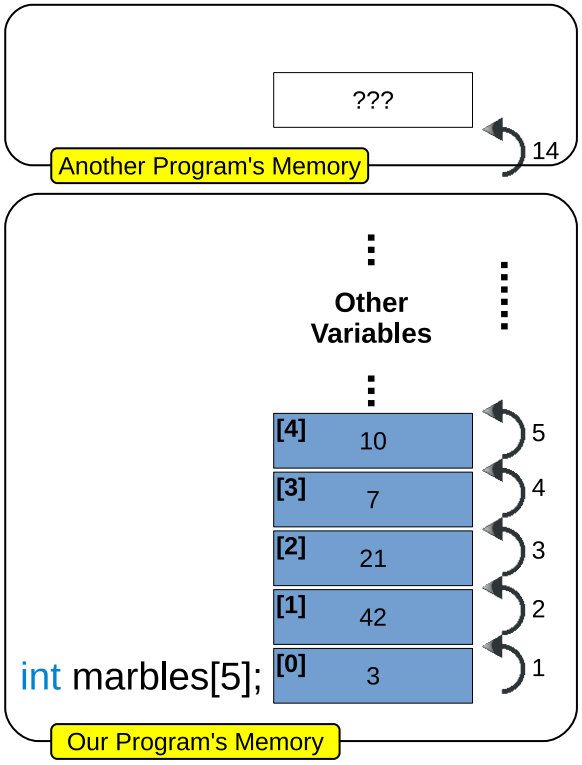
This might be an even worse problem if we tried to *change* the value of `marbles[14]`. In that case, if the program didn't crash, we'd be unexpectedly *modifying* the value of some completely different variable in our program.

If Jesse James were alive today he might have robbed computers instead of trains. Don't give Bad Guys a break! Check to make sure your array indices don't stray outside your arrays.

Source: Wikimedia Commons



(a) Reading beyond the end of an array, but still staying within the memory allocated to this program. This will succeed, but the number you get will likely be nonsense.



(b) Attempting to read outside the memory allocated for this program. This will fail and cause the program to crash.

Figure 6.3: Reading past the end of an array will give unexpected results.

It's up to the programmer to prevent these problems. In Program 6.2 for example, we could add an "`if`" statement to check to see if the number entered is between zero and ninety-nine, and tell the user to pick another number if it's not.

Reading or writing past the end of an array is one of the most common programming mistakes. It has led to many bugs in many programs, including some serious security bugs. Imagine what could happen if a banking program accidentally allowed users to change the value of any variable by entering, say, a very large account number! Bad Guys routinely look for bugs like this, and try to exploit them.

Let's move away from the hot, dirty coal industry for a little while now, and visit the cool, clean world of mathematics.

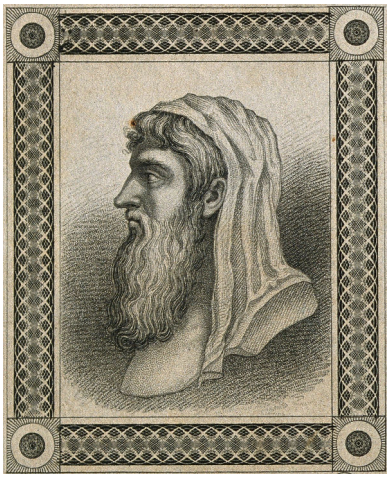
6.6. The Sieve of Eratosthenes

Prime numbers have fascinated mathematicians since ancient times. You'll recall that a prime number is a whole number that can only be divided evenly by itself and one. The first five prime numbers are 2, 3, 5, 7 and 11. (the number 1 isn't considered to be a prime.) Numbers that aren't prime are called *composite* numbers.

Early on, the Greek mathematician Euclid proved that there are infinitely many prime numbers. There doesn't, however, seem to be any simple rule for predicting them all. You just have to find them by searching.

Another Greek mathematician, Eratosthenes, described a straightforward procedure for searching for prime numbers. Today we call his technique "the Sieve of Eratosthenes". It finds primes by a process of elimination. First, write down all numbers in a range, and then mark out the ones that aren't prime. Anything left over (the "holes" in the sieve) is prime. But how to you know which numbers to eliminate?

Here's how it works: Write down all of the numbers from one to N, where N is the highest number you want to test. Then mark out all the multiples of 2 (4, 6, 8, ...). We know that none of these numbers can possibly be prime, since they can be divided evenly by 2. After that, mark out all of the multiples of 3 for the same reason, and so on. When you've gone through all of the numbers, anything that hasn't been marked out isn't a multiple of anything but 1 and itself, so it's a prime number. Figure 6.4 shows what it might look like after you'd



Euclid, who lived around 300 BCE, is best known as the father of geometry.

Source: Wikimedia Commons

Eratosthenes, born around 276 BCE, is perhaps best remembered for his remarkably accurate determination of the radius of the earth. (No, the ancient Greeks didn't think the earth was flat!)

Source: Wikimedia Commons

done this for the numbers 1 to 100.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Figure 6.4: White squares show the prime numbers between 1 and 100. Gray squares are numbers that have been marked out by the sieve process.

Program 6.3 uses Eratosthenes' technique to find all of the prime numbers smaller than 100,000. In terms of the description above, the program sets N equal to 100,000. It begins by defining an $N+1$ element array named `isprime` that will hold the "prime status" of each number. If the number i is prime, then `isprime[i]` will be equal to 1. Otherwise, this value will be zero.

Why does the array need $N+1$ elements? Remember that the last element of a 100-element array is number 99, not 100, since the first element is number zero. If we want the last element of `isprime` to be number N, then the array needs to have $N+1$ elements.

Notice that, instead of writing "`int isprime[100001];`" we've defined a variable, `N`, that says how many elements are in our array. The size of an array can't be changed once it's defined, though, so it's a good idea to mark a variable used this way as a "constant". By putting the word `const` in front of a variable definition, you tell the compiler that the value of this variable will never change.[5] If you try to change the variable's value somewhere later in the program, the compiler will give you an error message and refuse to compile the program.

[5] "constant variable?" Isn't that an oxymoron?

Program 6.3 assumes that all of the numbers are prime unless proven otherwise. The first "`for`" loop initializes all of the elements of `isprime` to a value of 1.

The next "`for`" loop begins with 2, and goes through all of the multiples of 2 that are smaller than N. For each of these multiples, the program

sets the corresponding element of `isprime` to a value of zero, thus flagging this number as a non-prime. The program then works its way through multiples of other numbers, up to N.

When it's done, anything that still has an `isprime` value of 1 is really a prime. The program prints out these numbers, and a count of how many primes were found.

You can probably think of some shortcuts we could have taken to make our program run faster. For one thing, if you've worked partway through the list and come to, say, 31, you know without going any farther that 31 is prime, since only smaller numbers could possibly be its factors. For another thing, it turns out that you only need to look for multiples of prime numbers. All the multiples of 4, for example, will already have been marked out, since they're also multiples of 2, and all multiples of 6 are also multiples of 2 and 3, which have already been marked out. Finally, we only need to test multiples of numbers smaller than $\sqrt{N}$. Any larger, non-prime numbers smaller than N must be a multiple of one of these.

To keep the program simple, Program 6.3 doesn't use these shortcuts. It trades speed for simplicity. This is a choice you'll often have to make as a programmer. Is a simple program fast enough? If I make the program more complicated in order to gain some speed, will I be more likely to do something wrong?

Exercise 34: Prime Time

Create, compile and run Program 6.3. How many primes does it find? Think about what problems you might run into if you tried to use this program to find even larger prime numbers.

The study of integers is an important part of the branch of mathematics called "number theory". Mathematician Leopold Kronecker famously said "God made the integers, all else is the work of man."

Source: Wikimedia Commons

N	Number of Primes
10	4
100	25
1,000	168
10,000	1,229
100,000	9,592
1,000,000	78,498
10,000,000	664,579
100,000,000	5,761,455
1,000,000,000	50,847,534
10,000,000,000	455,052,511
100,000,000,000	4,118,054,813
1,000,000,000,000	37,607,912,018
10,000,000,000,000	346,065,536,839

Figure 6.5: The number of primes less than N, for various values of N.

Source: https://primes.utm.edu/howmany.html

Program 6.3: sieve.cpp

```
#include <stdio.h>
int main () {
  const int N = 1e+5;
  int isprime[N+1];  // Why N+1? Number of elements, INCLUDING ZERO!
  int i;
  int multiple;
  int nprimes = 0;

  // Start by assuming everything is prime:
  for ( i=0; i<=N; i++ ) {
    isprime[i] = 1;
  }

  // Mark the non-primes:
  for ( i=2; i<=N; i++ ) { // Don't want to include multiples of 1!
    multiple = i+i; // First multiple of i
    while ( multiple <= N ) {
      isprime[multiple] = 0;
      multiple += i;
    }
  }

  // Print out what's left:
  for ( i=2; i<=N; i++ ) { // Why 2? Zero and 1 aren't prime by definition.
    if ( isprime[i] == 1 ) {
      printf ( "%d\n", i );
      nprimes++;
    }
  }

  printf ( "Total number of primes below %d is %d\n", N, nprimes );
}
```

6.7. Reading Array Elements

Because C doesn't prevent us from going past the end of an array (see Section 6.5 above) we need to be careful when we read data from a user or from a file and put it into an array. Take a look at Program 6.4, for example. This program defines a 5-element array named `marbles`, and asks the user to enter numbers into it, one element at a time. The numbers are then printed out in reverse order.

Notice that the program uses "`for`" loops that systematically go through the array's indices, from zero to 4. (Remember that the last element of a 5-element array is numbered 4, since the first element's number is zero.) Also notice that we put the array element into the `scanf` statement in just the same way that we'd put a non-array variable. In particular, we still need to put an ampersand in front of it.

After reading the numbers, the program prints them out in reverse order. It does this by starting with the last array element and working backwards through the array. We could have done this by saying "`for ( i=4; i>=0; i-- )`", but we've chosen to do it a different way. The program uses the same kind of "`for`" loop that it used when reading the numbers, but instead of printing `marbles[i]` it prints `marbles[4-i]`. Since `i` starts at zero and goes to 4, the value of `4-i` starts at 4 and goes to zero.

Program 6.4: reverse.cpp

```
#include <stdio.h>
int main () {
  int marbles[5];
  int i;

  for ( i=0; i<5; i++ ) {
    printf ( "Enter a number: " );
    scanf ( "%d", &marbles[i] );
  }

  printf ( "Numbers in reverse order:\n" );
  for ( i=0; i<5; i++ ) {
    printf ( "%d\n", marbles[4-i] );
  }
}
```

If you run Program 6.4 it might look like this:

```
./reverse
Enter a number: 2
Enter a number: 7
Enter a number: 5
Enter a number: 1
Enter a number: 9
Numbers in reverse order:
9
1
5
7
2
```

Exercise 35: Doing Flips

Create, compile and run Program 6.4. Does it work as expected?

Figure 6.6: The airplane image on this 1918 "Inverted Jenny" stamp was accidentally printed upside-down. Only 100 such stamps are known to have been printed, making them very valuable to collectors. In 2007 one of these stamps was sold for almost $1,000,000.

Source: Wikimedia Commons

Array indices give us a way to uniquely identify each element of an array, but they can also provide information about relationships between elements. For example, they tell us the order of the cars in our coal train, or the order of the numbers we entered in Program 6.4.

6.8. Sorting the Elements of an Array

We sometimes want to sort the elements of an array based on the values they contain. In our coal train example, for instance, we might want to put the heaviest cars at the back of the train, and the lightest at the front.

One of the simplest (but, unfortunately, slowest) ways to sort things is called a "bubble sort". Let's write a program that uses a bubble sort to arrange the cars of our train from lightest to heaviest.

A bubble sort works by comparing the values in two neighboring elements of an array. If the two values are in the proper order already (light car in front of heavy car, in our train example), they're left alone. Otherwise the two values are swapped to put them into the right order. We go through each pair of elements in the array, from first to last, swapping values when necessary. Then we do this again and again, until no more values need to be swapped. At that point, the array has been completely sorted. Figure 6.7 shows what the first pass might do to the values in our `marbles` array.

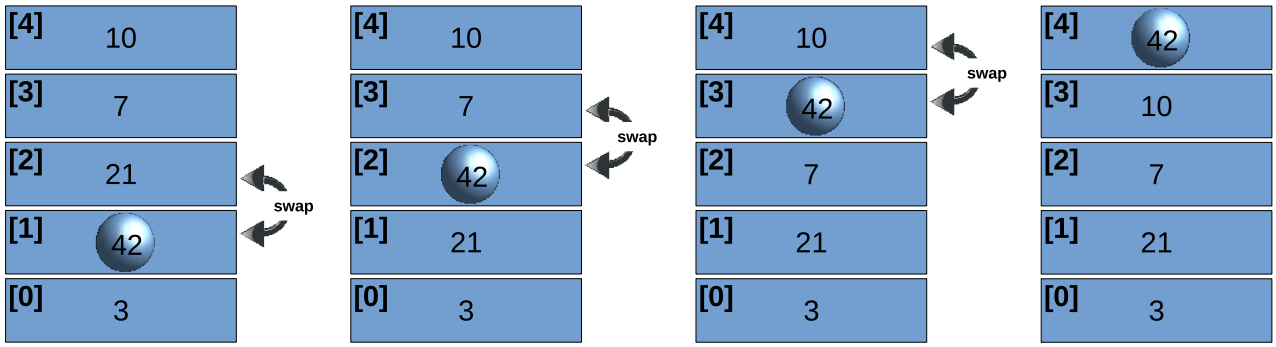


Figure 6.7: A bubble sort works its way through this array from bottom to top, comparing neighboring numbers and swapping them where necessary. When we get to the top of the array, we see that the largest number has "bubbled up". We could then start back at the bottom and repeat this procedure until all of the numbers had been sorted.

To write a program that does this, we'll first need to think about how to swap the values of two elements of an array. We can't just copy, say, `marbles[1]` into `marbles[2]`. If we did, we'd have two copies of the value in `marbles[1]`, and would have lost the value of `marbles[2]` completely! To swap values in a program, we'll generally need to have a temporary storage place to put one of the values while we're moving things around. This is illustrated in Figure 6.8.

Once we know how to swap the values in two elements, we're ready to write our bubble sort program. Program 6.5 is the result. The middle of the program is two nested loops.

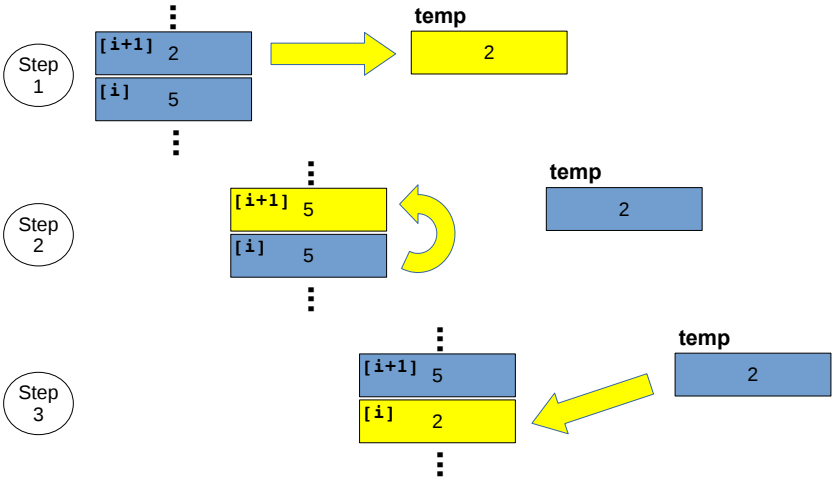


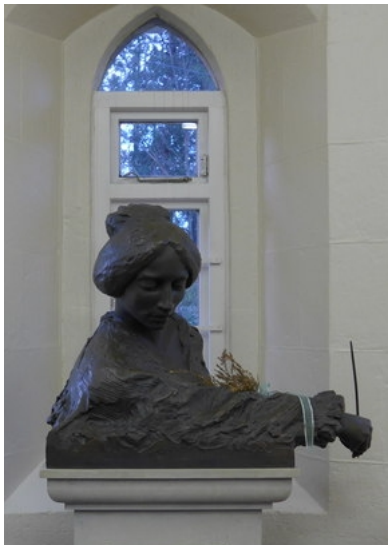
Figure 6.8: Swapping two values usually requires a temporary storage location. This illustration shows to swap the values in two adjacent elements of the `marbles` array. We use a variable called `temp` as a place to park one of the values while we're moving things around.

The inner loop is a "`for`" loop. This loop goes through each pair of array elements, starting with elements zero and one, then going to one and two, two and three, and so forth. The last pair will be 98 and 99, since the last element is number 99. The loop's counter variable, `i`, identifies the first member of each pair. The second member is `i+1`. The loop stops when `i` is equal to 98 and `i+1` is equal to 99 (the last element of the array).

The variable `temp` is a temporary storage location for use while swapping values, as shown in Figure 6.8. The variable `nswapped` keeps track of how many pairs needed to be swapped. before we begin each pass through the elements, `nswapped` is reset to zero.

The outer "`do-while`" loop repeats the inner loop until there are no more pairs that need swapping, indicated by a value of zero for `nswapped`.

A bubble sort is a simple sorting algorithm. An "algorithm" is just a recipe for doing something. Bubble sorts are easy to write, but there are much faster sorting algorithms. We'll look at one of these called "qsort" in a later chapter.



"A little later, remembering man's earthly origin, 'dust thou art and to dust thou shalt return,' they liked to fancy themselves bubbles of earth. When alone in the fields, with no one to see them, they would hop, skip and jump, touching the ground as lightly as possible and crying 'We are bubbles of earth! Bubbles of earth! Bubbles of earth!"' —Flora Thompson, in *Lark Rise* (1939)

Program 6.5: bubble.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main () {
  double carweight[100];
  double w;
  int i;
  double temp;
  int nswapped;

  for ( i=0; i<100; i++ ) {
    w = 50.0 + 50.0 * rand()/(1.0 + RAND_MAX);
    carweight[i] = w;
  }

  do {
    nswapped = 0;
    for ( i=0; i<99; i++ ) {  // Note: omit last element!
      if ( carweight[i] > carweight[i+1] ) {
        temp = carweight[i];
        carweight[i] = carweight[i+1];
        carweight[i+1] = temp;
        nswapped++;
      }
    }
  } while (nswapped > 0);

  for ( i=0; i<100; i++ ) {
    printf ("Car %d carries %lf tons.\n", i, carweight[i] );
  }
}
```

But what about...?

Where does the word "algorithm" come from anyway? Surprisingly, it has nothing to do with Al Gore. Instead, it's a variation on the name of Muhammed ibn Musa al-Kwarizmi. al-Kwarizmi was an 8^{th}-Century Persian mathematician who adopted a revolutionary new Indian method for writing numbers: the decimal number system we still use today. Before decimal numbers, arithmetic was a tedious process only known to specialists. Decimal numbers suddenly made arithmetic accessible to the masses.

al-Kwarizmi's writings, translated into Latin, brought the new number system to Europe, along with other insights into mathematics. The word "algebra" comes from the Arabic word *al-jabr*, meaning "make whole", used in the title of one of al-Kwarizmi's books: *al-Kitab al-mukhtasar fi hisab al-jabr wal-muqabala* (*The Compendious Book on Calculation by Completion and Balancing*).

al-Kwarizmi's mathematical writings were so influential that his name, transmogrified into "algorithm", became a shorthand for calculation in general.

A page from one of al-Kwarizmi's books.

Source: Wikimedia Commons

6.9. Fun with Metronomes

A metronome is a device that clicks in a regular rhythm. Music students sometimes use them while practicing. These devices have a straight, weighted arm that swings back and forth. Imagine that you have several metronomes sitting at various widely-separated places in a room. The metronomes are all ticking at the same rate, but the arm of each metronome has been set in motion at a different time. It might look like the top half of Figure 6.9. At any given time, the arms of the metronomes are in different places. We say that the metronomes are "out of phase", and they would stay that way for as long as we could tolerate their maddening ticking!

Now imagine we take the metronomes and put them side-by-side on a wobbly table[6]. Again, we start their arms moving at different times, so they're out of phase. But now we'd find that, over time, the metronomes begin to synchronize, until they are eventually "in phase" with each other, with all the arms at the same position at any given time, like the bottom half of Figure 6.9.

[6] This example was inspired by Matt Parker's video on this topic: https://www.youtube.com/watch?v=J4PO7NbdKXg

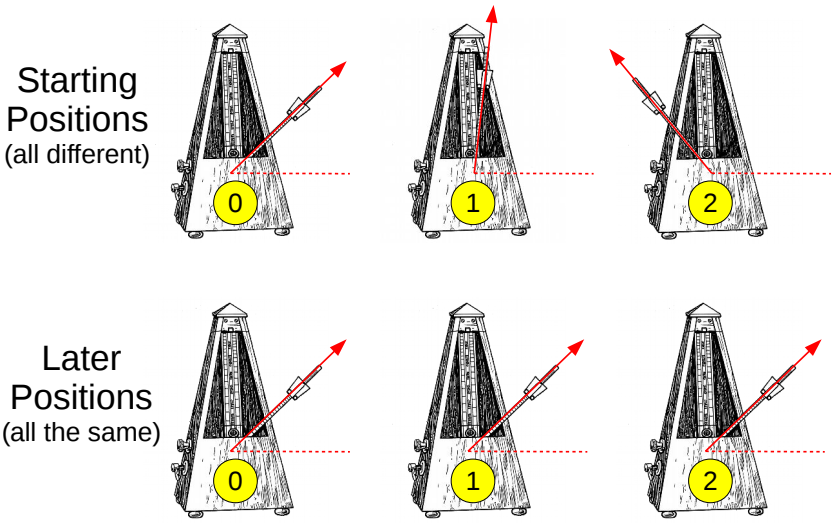


Figure 6.9: Even though the metronomes' arms might start out in different places, they can influence each other over time.

What's happening here is that the wobbly table lets the metronomes jiggle each other a little bit. We say that they're now "coupled", whereas they were "uncoupled" when they were spread out around the room. Over time, the coupling between the metronomes tends to bring them into phase with one another.

In the 1970s Yoshiki Kuramoto developed a simple mathematical model[7] that describes how the metronomes' motion evolves from out-of-phase to in-phase. Let's write a program that uses Kuramoto's model to simulate the behavior of a set of metronomes on a wobbly table.

We're going to need to keep track of each metronome's arm as it oscillates back and forth. Figure 6.10 shows the motion of a single metronome. The vertical axis shows the position of its arm, where 1 means all the way to the right, and -1 means all the way to the left. As time passes, the arm oscillates between these two extremes in a sine wave.

Figure 6.11 shows the motion of four uncoupled metronomes. They move in sine waves with the same frequency, but they're shifted relative to each other because the arms were started at different times.

When dealing with oscillating things, it's natural to measure time in terms of multiples of the oscillating period. We could say that the mentronome has gone through one cycle, two cycles, three cycles... The vertical axis (arm position) on our graph is the sine of an angle, and the horizontal axis (time) is an angle telling us how far "around" the

[7] http://go.owu.edu/ physics/StudentResearch/2005/BryanDaniels/kuramoto_paper.pdf

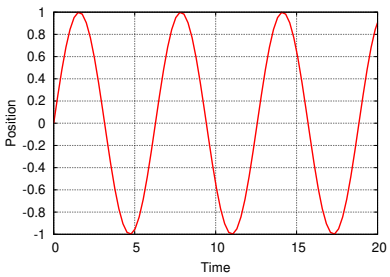


Figure 6.10: The motion of a single metronome arm.

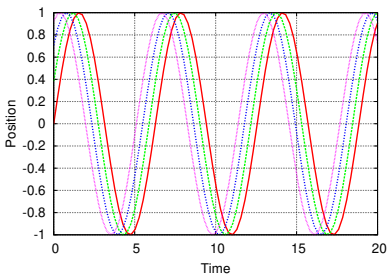


Figure 6.11: The motion of four "uncoupled" metronome arms.

cycle we've gone so far. (Note that it's perfecly OK to go around twice, or three times, or as many times as we want.) One complete cycle is equivalent to an angle of 2π radians.

The time to go through one complete cycle is the metronome's period. After some amount of time, t, the "angle" the metronome has traveled through in its cycle is $\theta = 2\pi t / period$. Note that this is different from the physical angle the metronome's arm makes. θ here is an abstract thing that just tells us what stage we're at in the metronome's cycle. If different metronomes are started at different times, that's just equivalent to shifting θ by some amount that we'll call each metronome's "phase angle". When the metronomes jiggle each other, they gradually change each other's phase angles until all they're all the same.

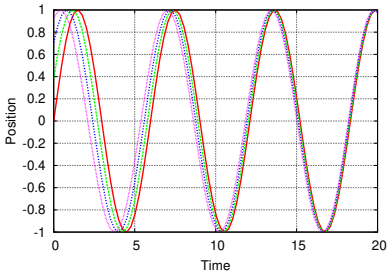


Figure 6.12: The motion of four metronomes that are "coupled" because they're sitting together on a wobbly table.

For coupled metronomes, the Kuromoto model tells us that each metronome is jiggled by each other metronome by an anount that's proportional to the difference in their phase angles. Mathematically, we could say that the change in phase angle of metronome i is:

$$correction_i = \frac{constant}{N} \times \sum_{j=0}^{N-1} (phase_j - phase_i)$$

where N is the number of metronomes and j is a label for each metronome, starting with zero. Over time, after many such small corrections, this would cause the phases of the metronomes to converge, as in Figure 6.12.

Program 6.6 tracks the motion of four metronomes that can jiggle each other. It initially gives the metronomes different phase angles spread evenly between zero and $\pi/2$ radians (1/4 of the way through a cycle). Then the program starts a loop that goes through four complete metronome cycles in 100 steps. During each step, the program loops through all the metronomes, and for each metronome it calculates the correction due to all the other metronomes. It then does a second loop and applies those corrections by modifying each metronome's phase angle. During each step, the program prints out the current value of θ and the position of each metronome arm (given by `sin(theta + phase[i])`.

At the top of the program we define two arrays, `phase` and `correction`, that will hold the current phase angle of each metronome and the correction to be applied to that phase angle before beginning the next step. We can't just change the numbers in `phase` because we still need those values until we're finished calculating the correction to each of the metronomes. That's why we store the corrections in a separate

array until we're ready to apply them.

When you run the program it will print five columns of numbers: θ, which represents time, and the position of each of the four metronome arms. If you want to simulate more metronomes, just change the value of `nmetronomes` in the program.

Program 6.6: metronome.cpp

```
#include <stdio.h>
#include <math.h>
int main () {
  const int nmetronome = 4;
  double nsteps = 100;
  double phase[nmetronome];
  double correction[nmetronome];
  double coupling_strength = 0.03;
  double thetamax = 8.0*M_PI;
  double diff,diffsum;
  double theta,thetastep;
  int istep;
  int i,j;

  diff = 0.5*M_PI/nmetronome;
  for ( i=0; i<nmetronome; i++ ) {
    phase[i] = diff*i;
  }

  thetastep = thetamax/nsteps;
  theta = 0;
  for ( istep=0; istep<nsteps; istep++ ) {

   printf( "%lf ", theta );

   for ( i=0; i<nmetronome; i++ ) {
     printf( "%lf ", sin(theta + phase[i]) );

     diffsum = 0;
     for ( j=0; j<nmetronome; j++ ) {
       diffsum += phase[j] - phase[i];
     }
     correction[i] = coupling_strength*diffsum/nmetronome;
   }

   printf ("\n");

   for ( i=0; i<nmetronome; i++ ) {
     phase[i] = phase[i] + correction[i];
   }

   theta += thetastep;
  }

}
```

Set initial values.

Print θ.

Loop through all metronomes.

Print arm position.

Add up the phase differences.

Loop through time

Apply corrections.

6.10. Multi-Dimensional Arrays

Each array we've seen so far can be visualized as a long, one-dimensional chain of elements, one after another. Arrays don't have to be one-dimensional, though. For example, the program below shows an array called `matrix` with two indices. We could think of this as representing a two-dimensional (20 × 30, in this case) matrix of values.

```
int main(){
  int matrix[20][30];
  int i,j;

  for (i=0; i<20; i++) {
    for (j=0; j<30; j++) {
      matrix[i][j] = i * j;
    }
  }
}
```

A two-dimensional array might store the barometric pressure at locations on a map grid, or the number of grains of wheat on each square of a chess board.

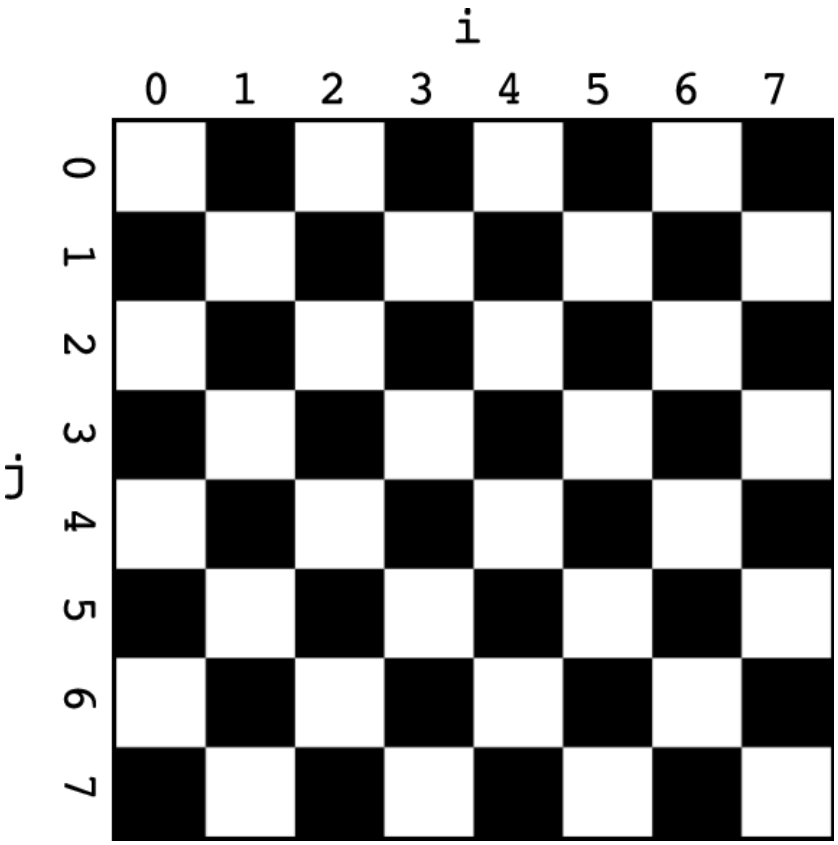


Figure 6.13: An 8 × 8 two-dimensional array, with the indices `i` and `j`.

The Chess Game (1555), by Sofonisba Anguissola

Source: Wikimedia Commons

There's a legend, of uncertain origin, that goes something like this: The inventor of chess presented the new game to his ruler, who was so pleased that he offered the inventor any prize he wanted. The apparently modest inventor asked only for some grains of wheat (or rice, in some versions). One grain was to be placed on the first square of the chessboard, two on the second, four on the third, and so forth, doubling the number of grains each time, until the last square was reached. "Certainly!" said the ruler, but he found that he couldn't honor his offer. To reach the last square would require over 2^{63} grains of wheat, more than all of the wheat in the world!

The Karl G. Jansky Very Large Array (VLA) is an array of radio telescopes near Socorro, New Mexico. The antennas can turn to follow celestial targets as the Earth rotates. Their motion is usually so slow as to be almost imperceptible, but they periodically need to "unwind" to avoid tangling cables. Astronomers describe the eerie scene when, in the middle of the night, a plain full of antennas suddenly begins twisting in unison, as though they've come to life.

Source: Wikimedia Commons

Arrays in C can have as many indices as you like. A three-dimensional array might hold data about a grid of points in space, or a four-dimensional array might be useful for problems in General Relativity, where space and time are combined into a four-dimensional continuum.

Each index of a multi-dimensional array starts with zero, just like arrays with a single index. In the example above, the first index of `matrix` goes from zero to 19, and the second index goes from zero to 29.

6.11. Working with 2-dimensional Arrays

You're an artillery sergeant in the Union Army during the American Civil War. The rebel forces are trying to float a barge full of coal down the Missisippi river to supply fuel for their new ironclad warship. Your job is to make sure that barge doesn't reach its destination. You set up camp on the side of the river and wait for the barge to come through. But wait! Suddenly a thick fog descends, blocking your view of the river! You'll have to fire blind, and listen for the sound of crackling wood to tell you whether you've hit the barge.

Quickly you sketch out a diagram of the river to help you keep track of hits and misses (see Figure 6.14).

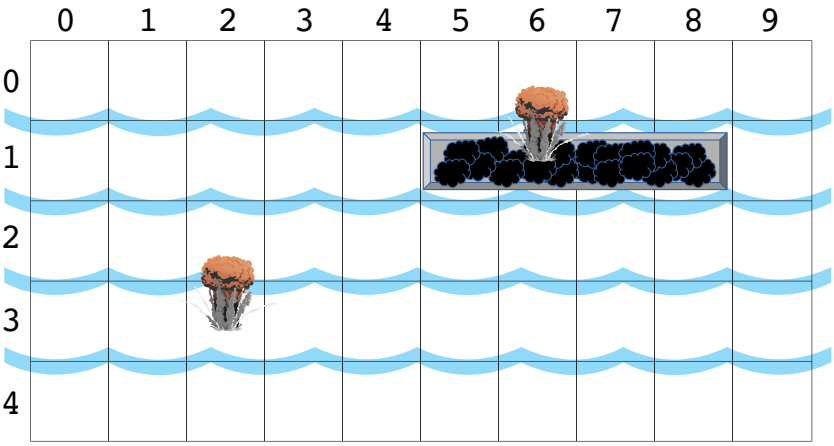


Figure 6.14: A coal barge floats down the Mississippi. The front of the barge is at `[5][1]`, and the vessel occupies the four elements to the right of that position. An artillery shell has hit the barge at position `[6][1]`, but another shell at `[2][3]` has missed.

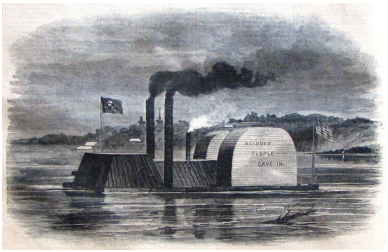
Hmmm. This sounds like it would make an exciting game! Fortunately, you learned C programming in Boot Camp, so after completing your mission you can return home and write Program 6.7.

These are the rules of the game: A coal barge occupies a line of four consecutive elements in a 2-dimensional array (the map). The barge is

oriented horizontally, along the flow of the river, and placed at some random location on the map. The barge must be completely on the map, it can't hang off the edge.

In order to win the game, the player must hit each of the four array elements that contain the barge. The player fires an artillery shell by giving the two indices, `[i][j]`, of an array element. The program tells the player whether the shell hits the barge.

The program uses a 2-dimensional array named `grid` to store a map of the river and the barge's position. Most of this array contains zeros, but the four elements occupied by the barge are initially marked with ones. When a player hits one of the barge elements its value is changed to -1. The variable `nhits` keeps track of the total number of hits. The program keeps running as long as `nhits` is less than four.

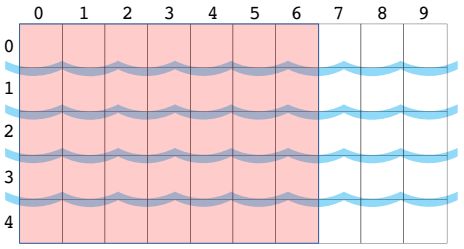


In 1863 Union forces built this dummy ironclad out of an old coal barge, and used it to frighten Confederates. The smokestacks were made of pork barrels and contained smudge pots to make smoke.

Source: Wikimedia Commons

Program 6.7: coalbarge.cpp

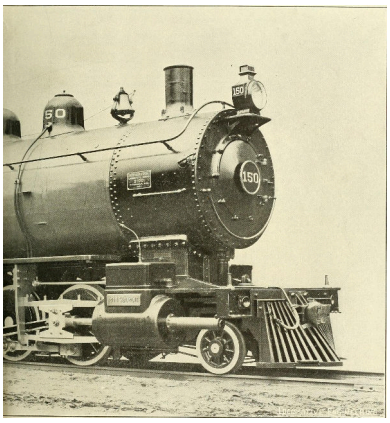
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
  int grid[10][5];
  int nhits = 0;
  int i, j, iprow, jprow;

  for ( i=0; i<10; i++ ) {
    for ( j=0; j<5; j++ ) {
      grid[i][j] = 0;
    }
  }

  srand( time(NULL) );
  iprow = (int)( 7.0*rand()/(1.0 + RAND_MAX) );
  jprow = (int)( 5.0*rand()/(1.0 + RAND_MAX) );

  for ( i=0; i<4; i++ ) {
      grid[iprow+i][jprow] = 1;
  }

  do {
    printf ("Enter x coordinate: ");
    scanf( "%d", &i );
    printf ("Enter y coordinate: ");
    scanf( "%d", &j );
    if ( i >= 10 || i < 0 || j >= 5 || j < 0 ) {
      printf ("Bad coordinates. Try again.\n");
      continue;
    }

    if ( grid[i][j] == 1 ) {
      printf ("Hit!\n");
      grid[i][j] = -1;
      nhits++;
    } else if ( grid[i][j] == -1 ) {
      printf ("Already hit!  Try again.\n");
    } else {
      printf ("Miss!  Try again.\n");
    }
  } while ( nhits < 4 );

  printf ("You sunk my coal barge!\n");
}
```



The front ("prow") of the barge must be in the shaded section to prevent the back end from hanging off the map.

6.12. Solving a Heat Problem

Let's use a two-dimensional matrix to help us solve a problem. Imagine that you have a square metal plate. One edge of the plate is connected to something very hot, like the engine of a locomotive. The other three edges are connected to a cooling system that keeps them cold. But what are the temperatures of the other parts of the plate?



A steam locomotive.

Source: Wikimedia Commons

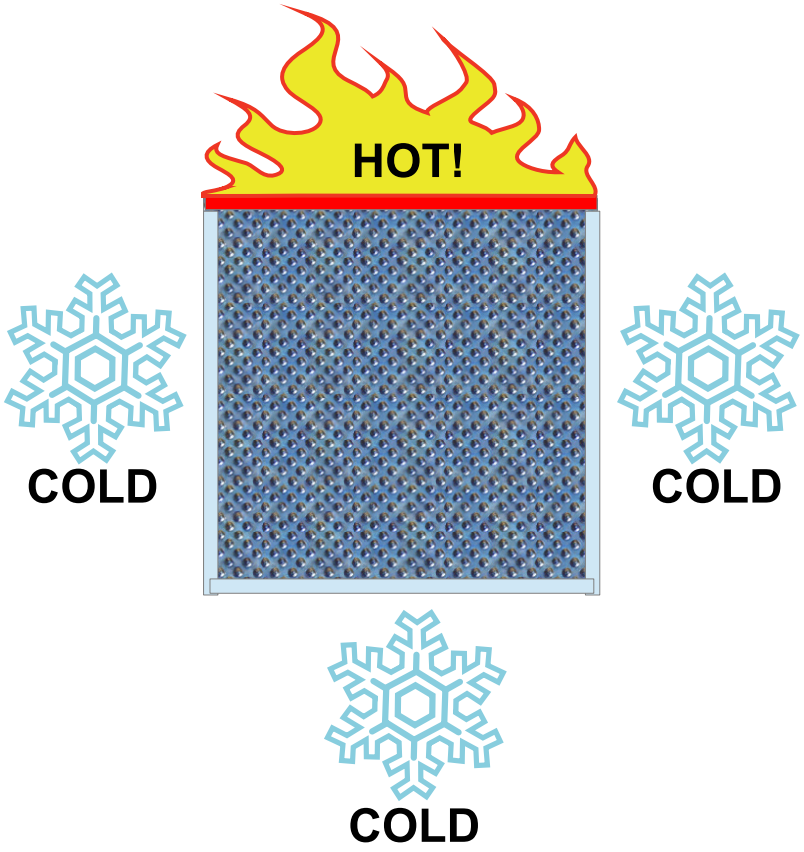


Figure 6.15: A metal plate, hot on one edge and cold on the others.

We might assume that points near each other on the plate would have similar temperatures. Points near the hot edge would tend to be hotter than points near the cold edges. In fact, it wouldn't be surprising if the temperature at any given point were close to the average of the temperatures at the points around it.

Let's write a program that tries to estimate the temperature at various points on the plate. Assume that the very hot edge of the plate has a temperature of 500 celsius, and that the cold sides are kept at a chilly zero celsius by our highly efficient cooling system.

We'll start by dividing the plate into a 100 $\times$ 100 grid of points.

Hmmm. . . Sounds like a 2-dimensional array might be useful here. We could define the array like this:

```
double temp[100][100];
```

Pierre-Simon, marquis de Laplace made important contributions to many areas of mathematics.

Source: Wikimedia Commons

The array named `temp` will hold the temperature values of the points on our grid. We already know the temperatures of some of these points. The points along the hot edge of our plate have a temperature of 500 celsius, and those along the cold edges are at 0. These are the "boundary conditions" of our problem. We need to determine the temperatures of the other, interior, points though.

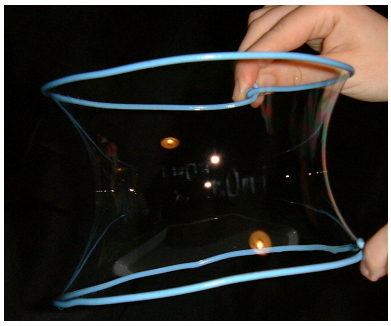
We'll start by just setting these interior temperatures arbitrarily to zero. This probably isn't a good guess for their temperature, especially for those points near the hot edge, but we can improve our estimate by using the approximation we mentioned above: We're going to assume that the temperature at any point is approximately the average of the temperatures of the neighboring points.

It turns out that this type of problem is a common one in physics and engineering. To arrive at the solution mathematically, we'd need to solve what's called "Laplace's Equation" for this system.[8] Fortunately, there's an easy way to find an approximate solution to Laplace's equation with a computer program. The technique is called "relaxation". You'll see why soon.

[8] In the language of math, Laplace's equation is expressed as $\nabla^2\phi = 0$.

After setting the temperatures, let's go through all of the interior points, changing each point's temperature to the average of its neighbors' temperatures. After doing this, we might expect that the temperatures are a now a little more realistic. How far off was our original estimate? We might look at how much difference there is between our original guess and the new estimate. What's the biggest difference?

If we did this averaging process again, we'd get an even better approximation for the temperatures, and we'd see that the maximum temperature change is smaller than it was the first time. If we keep averaging, again and again, the temperature values will eventually settle into stable values that don't change much each time we average. At some point, we decide that this approximation is good enough, and stop averaging.



A soap film stretching between two hoops.

Source: Wikimedia Commons

Our approximation started out very far from the true temperatures.

You could think of this original approximation as being a rubber sheet that's stretched out into some unnatural shape. Each time we do the averaging process, the sheet "relaxes" a little, until it falls into whatever natural shape fits the boundary conditions we've imposed. That's why this technique is called "relaxation". It can be used for temperature problems like this, but also for a real rubber sheet, or for a soap film on a wire frame. All of these are instances of a system controlled by Laplace's equation.

Program 6.8 follows the strategy we've described above and uses it to find approximate temperatures for interior points on our metal plate. First, it sets temperatures to some initial values, then repeatedly loops through all of the interior points, averaging temperatures. Every time it changes a temperature, it looks to at the size of the change and keeps track of the biggest change. When the changes get small enough (smaller than `smalldiff`), the program prints the final temperatures.

Notice that Program 6.8 uses some magic to make sure each element of the `temp` and `told` arrays contains a value of zero when the program starts. That's what the special value `{{0}}` means when defining a 2-dimensional array.

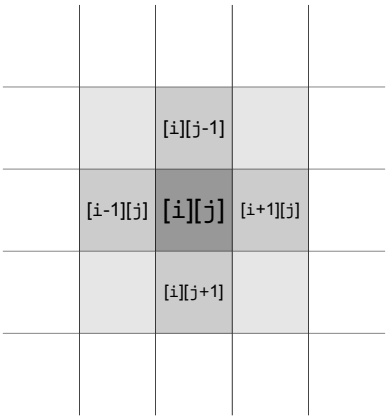


Figure 6.16: Program 6.8 sets the new value of `temp[i][j]` equal to the average temperature of the four array elements surrounding it. As we saw in Section 6.7, the array indices can be used to tell us something about the relationships between array elements. In this case, the indices give us information about which elements are near each other.

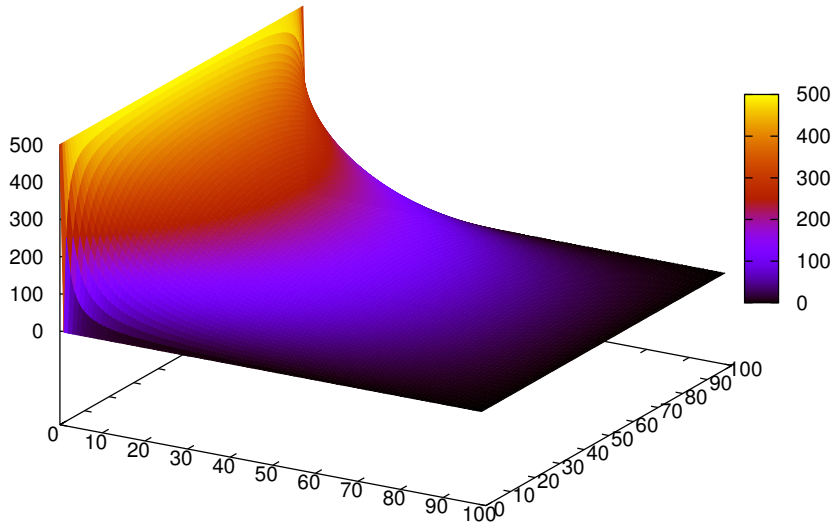


Figure 6.17: Temperatures at various points on the metal plate, as estimated by Program 6.8.

We can put the program's output into a file by writing `./relax > relax.dat`, then we can graph the results with *gnuplot*. Figure 6.17 shows the result of the following *gnuplot* command:

```
splot "relax.dat" with pm3d
```

The command `splot`[9]does a "surface plot". The qualifier "`with pm3d`" tells *gnuplot* to use a style of plotting called "palette-mapped 3-d". This color-codes values based on their height. The color scale shows which color corresponds to which value.

[9] Note that it's **s***plot*, not *plot*.

To enable *gnuplot* to read the data file, Program 6.8 wrote it in a particular format. If you look inside the data file (`relax.dat`) you'll see that it contains a single column of numbers. If you scroll down in the file a little, you'll see that there's an empty line after every 100 numbers. The numbers represent the temperature values along a row of our grid points on the metal plate. Each extra blank line indicates the beginning of the next row.

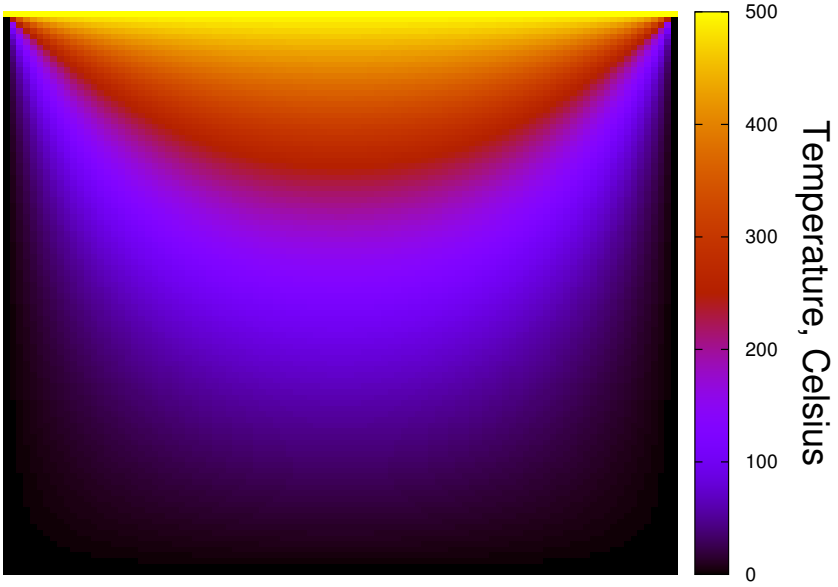


Figure 6.18: Another view of the temperature distribution, as seen from above the plate.

By choosing different "boundary conditions" (the unchanging temperatures at the plate's edges) we can simulate other interesting situations. For example, Figure 6.19 shows the temperature distribution on the plate when there are two hot spots at the top and one hot spot at the bottom.

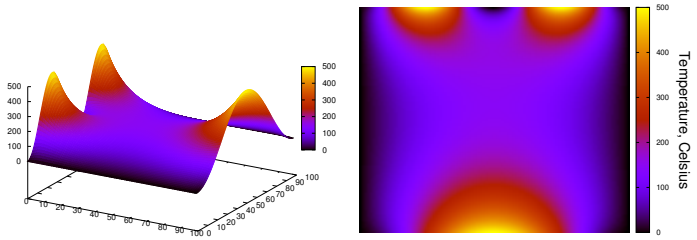
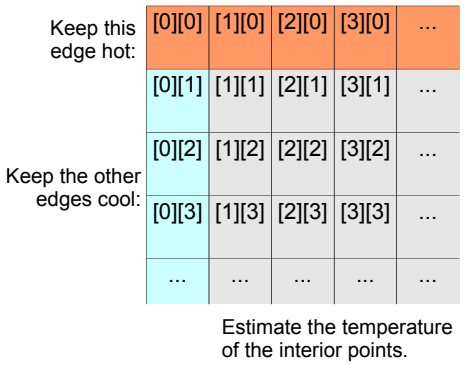


Figure 6.19: These graphs show the temperature distribution when there are two hot spots at the top of the plate and one hot spot at the bottom.

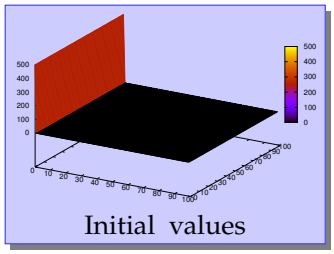


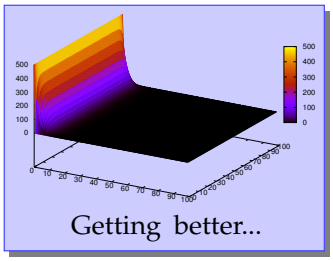
Program 6.8: relax.cpp

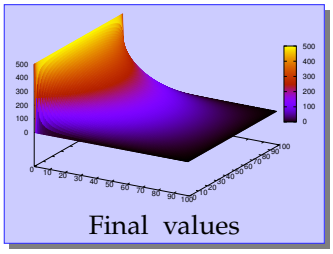
```
#include <stdio.h>
#include <math.h>
int main () {
  int i,j;
  double diff, maxdiff, smalldiff=1.0e-3;
  double temp[100][100] = {{0}}; // Current temps.
  double told[100][100] = {{0}}; // Previous temps.

  // Set elements along hot edge to 500 celsius:
  for (i=0;i<100;i++){
    temp[i][0] = 500.0;
  }

  // Keep averaging until maxdiff is small:
  do {
    for (i=0;i<100;i++){
      for (j=0;j<100;j++){
        told[i][j] = temp[i][j];
      }
    }

    maxdiff = 0;
    // These two nested loops go through all of the
    // interior points:
    for (i=1;i<99;i++){
      for (j=1;j<99;j++){
        temp[i][j] = 0.25 * (told[i-1][j] + told[i+1][j] +
                             told[i][j-1] + told[i][j+1]);
        diff = fabs(temp[i][j]-told[i][j]);
        if ( diff > maxdiff) {
          maxdiff = diff;
        }
      }
    }
  } while ( maxdiff > smalldiff );

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

Relax...

See Figure 6.16 on Page 189







But what about...?

Let's look more closely at the trick we used in Program 6.8 when defining the `temp` and `told` arrays. If a statement like:

```
double temp[100][100] = {{0}};
```

gives each of the array's elements a value of zero, could we do this:

```
double temp[100][100] = {{100}};
```

to set all of the elements to 100?

Unfortunately, no, but the real result might suprise you. If you printed the values stored in an array defined like this, you'd find that element `[0][0]` had the value 100, as expected, but all of the other elements would be set to zero.

Let's back up a little and see how these curly brackets work when we use them in an array definition. As we noted back on Page 167, we can initialize the elements of an array by explicitly giving a list of values in curly brackets, like this:

```
int marbles[5] = {7,9,3,15,8};
```

But what if the list contains fewer values than the number of array elements, like this?:

```
int marbles[5] = {7,9};
```

In that case, the first two elements would be set to 7 and 9, and the rest would be set to zero. Whenever there are too few values, the computer assumes that we want to set the rest of the values to zero.

As we've noted before (see Chapter 4), variables are just named sections of the computer's memory, and we can't assume that they contain any particular value before we explicitly give the variable a value. If we want all of an array's elements to be zero, we need to set them to zero. We could do this by saying:

```
int marbles[5] = {0,0,0,0,0};
```

or, as we've just seen, we could say:

```
int marbles[5] = {0};
```

causing the computer to set the first element to zero, and setting all of the other elements to zero by default, since we didn't say what we wanted them to be.

Some compilers will even let us say "`int marbles[5] = {}`", but that doesn't work with all of them, so it's best to always give at least one value.

So what about the double brackets we used in Program 6.8? That's because these are 2-dimensional arrays. With a 2-d array, we can initialize values like this:

```
double x[20][20] = {{7,9},{4,3}};
```

setting the first two elements of the first row to 7 and 9, and the first two elements of the second row to 4 and 3. All of the other elements would be set to zero. And, if we said:

```
double x[20][20] = {{0}};
```

all of the array's elements would be set to zero. This is the trick we used in Program 6.8.

Figure 6.20: A collection of marbles within the permanent collection of The Children's Museum of Indianapolis.
Source: Wikimedia Commons

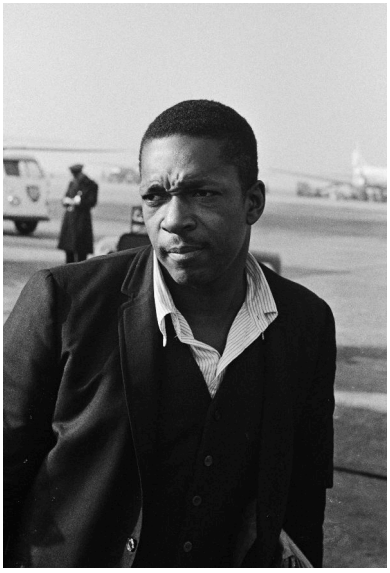
6.13. Conclusion

Arrays are useful whenever your program needs to store several related values. Array indices uniquely identify each array element, and they may also say something about relationships between array elements. (They can indicate the spatial or time ordering of measurements, for example.)

Some important things to remember about arrays are:

- The elements of an array can be of any type (but all elements of a given array must be of the same type).
- When defining an array, the number in square brackets says how many elements are in the array.
- It's important to remember that an N-element array's indices start with zero, and end at N-1.
- Arrays take up memory. It's easy to write "double x[1000]", but remember that this takes as much memory as a thousand single variables. Keep this in mind when defining large arrays.
- Array elements can be referred to by their indices.
- The index must be an integer.
- The index uniquely identifies a single array element.
- Arrays can optionally be initialized when they're defined.

Practice Problems



John Coltrane. Because "Coal Train".
Source: Wikimedia Commons

1. On page 171 it was suggested that adding an "`if`" statement to Program 6.2 could make it safer. Add an "`if/else`" statement to Program 6.2 (without changing anything else!) to check whether the number is too big or too small. If it is, ignore the number and give the user a helpful message. Call your program `coal.cpp`.

2. Create, compile and run Program 6.8. Use the "`ls`" command to make sure that the program created the file `relax.dat`.

   Use the *gnuplot* command described above to plot the data using *gnuplot*'s "pm3d" plotting style. If your version of *gnuplot* allows it, grab the figure with your mouse and rotate it around. Does it look like what you'd expect?

3. Imagine you have a very short coal train, containing only five cars. Each of the cars is to be sent to one of your customers. Each customer is identified by an integer "Valued Customer ID Number" (VCID) like "37654".

   (a) Using *nano*, create a data file named `orders.dat` that contains five rows of numbers, one row for each car in your train. Each row of the file will have two numbers: the weight of coal in that car (which might be a number with decimal places), and the ID of the customer it belongs to (which will always be an integer). The file might look like this:

   ```
   63.4 5487
   52.1 30978
   77.8 8765
   89.0 435
   95.3 789
   ```

   (b) Now write a program named `orders.cpp` that will read `orders.dat`. All of the weights should go into a 5-element array of `doubles` named `carweight` and all of the customer IDs should go into a 5-element array of `ints` named `vcid`, so that `carweight[0]` and `vcid[0]` are the weight and customer ID for the first car, and so forth. After the program reads the data, have it ask the user for a car number (a number between zero and four) and print out the weight and customer ID for that car. Make sure you **check the car number** to see if it's in the range zero to four, and tell the user if it's not. Also make sure the program tells the user which number is which.

   Hints: See Program 6.4 for something similar, and look at Chapter 5 for advice about reading things from files.

Problem 3 uses two "parallel arrays", `carweight` and `vcid`, to store two pieces of information about each car. If we wanted to add more information (maybe the car's age, so we know when to replace it?) we could add more arrays. We'll see a different way to do this sort of thing later, in Chapter 12.

4. In mathematics, a *matrix* is an array of numbers. Matrices are important in many fields of science, engineering and mathematics.

   Using *nano*, create a file named `matrix.dat` that contains a 3×3 matrix like this:

   ```
   8 4 1
   5 7 5
   1 0 3
   ```

   Write a program named `matrix.cpp` that reads data from `matrix.dat` and puts the numbers into 2-dimensional array, `double m[3][3]`. To do this, use nested pair of `for` loops that repeatedly uses `fscanf` to read the array's elements, one at a time. The `fscanf` statement might look like this:

   ```
   fscanf( input, "%lf", &m[col][row] );
   ```

   where `col` and `row` are the column and row numbers.

   Make the program do the following:

   (a) First, print out the elements of the matrix, so you can make sure they match the data in `matrix.dat`.

   (b) Second, compute and print out the *trace* of the matrix. The trace is defined as the sum of the matrix's diagonal elements. (See Figure 6.21.) (**Hint:** The diagonal elements are the ones where the row and column numbers are the same, like `m[0][0]` and `m[1][1]`.)

   (c) Third, compute and print out the *determinant* of the matrix. The determinant for a 3×3 matrix is:

   ```
   det =
     m[0][0]*( m[1][1]*m[2][2] - m[1][2]*m[2][1] ) +
     m[0][1]*( m[1][2]*m[2][0] - m[1][0]*m[2][2] ) +
     m[0][2]*( m[1][0]*m[2][1] - m[1][1]*m[2][0] );
   ```

   You'll obviously need to be careful when typing this into your program, but looking at the way the numbers in the statement line up vertically can help you avoid mistakes.

   If you make your `matrix.dat` file look like the example above, you should find that the matrix has a trace of 18 and a determinant of 121. Use these values to check your work.

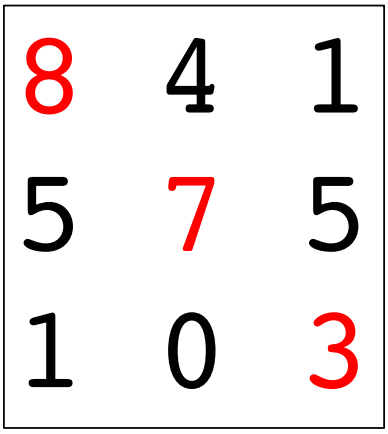
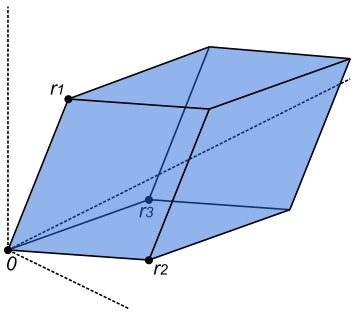


Figure 6.21: The *trace* of a matrix is defined as the sum of its diagonal elements. In the example above, the trace is equal to $8+7+3$.



We can think of each row of the matrix as the coordinates of a point in three-dimensional space. In the picture above, we call these points $r1$, $r2$, and $r3$. The *determinant* of the matrix is just the volume of the parallelepiped defined by these three points (the shaded volume above).

Source: Wikimedia Commons

5. Write a program named `fibarray.cpp` that fills an array with the first 20 terms of the Fibonacci sequence. The first two numbers in the Fibonacci sequence are 1, 1, and each subsequent number is the sum of the preceding two numbers. Your program should have a 20-element `int` array named `term`. The program should start out by setting `term[0]=1` and `term[1]=1`. Then the program should have a "`for`" loop that finds the value of each of the remaining 18 terms. Inside the loop you'll want to have a statement like `term[i+2] = term[i]+term[i+1]`. After this loop is done, add another loop that prints out all the elements of `term`. The program's output should look like this:

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

6. In the Fibonacci sequence each term is the sum of the two preceding terms. There's also a "Tribonacci sequence", in which each term is the sum of the *three* preceding terms. It starts out with the numbers 0, 0, 1. Referring to the instructions in Problem 5, write a program named `tribarray.cpp`, but with the Tribonacci numbers instead of the Fibonacci numbers. The program's output should look like:

```
0 0 1 1 2 4 7 13 24 44 81 149 274 504 927 1705 3136 5768 10609 19513
```