Chapter 5

Arrays

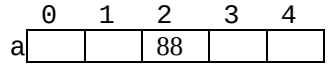
5.l DEFINING AND TRAVERSING ARRAYS

An *array* is a sequence of elements of the same type which are stored contiguously in memory and which can be accessed by means of an integer *subscript.* An array is declared by specifying its element type followed by its name followed by the number of elements enclosed in brackets [ ]. The elements are numbered consecutively, starting with 0. The elements of the array are accessed by specifying the array name followed by the element number enclosed in brackets.

EXAMPLE 5.l An Array of ints

```
int a[5]; // defines a to be an array of 5 elements of type int
a[2] = 88; // assigns 88 to element number 2
```

This array can be visualized as shown at right. Since array indexes always begin at `0`, the array size will always be one more than the index number of the last element. For example, the last element in a 5-element array is element number `4.`

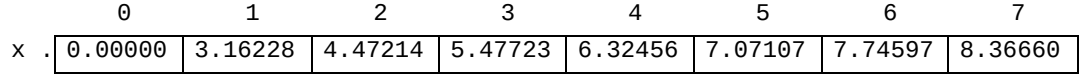


Arrays are usually processed with `for` loops, where the loop control variable is used as a variable index to traverse the array.

EXAMPLE 5.2 Using a for Loop to Traverse an Array

```
const int SIZE = 8;
float x[SIZE];
for (int i=0; i<SIZE; i++)
    x[i] = sqrt(10.0*i);
for (int i=SIZE-l; i>=0; i--)
    cout << i << ": " << x[i] << endl;
```

This array can be visualized like this:



Note how naturally the loop control variable matches the array index. If increasing, the control variable is initialized at 0 `(i = 0)` and the exit control should be `i<SIZE.` If decreasing, the control variable starts at `SIZE-1` and is controlled by the condition `i>=0.`

The size of an array must be constant. It may be an actual constant, as in Example 5.1, or it may be a symbolic constant like **`SIZE`** in Example 5.2. But it cannot be a variable:

```
int size;
cin >> size;
float x[size]; // ILLEGAL: size must be constant
```

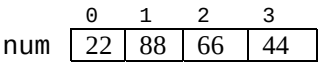
The reason for this restriction is that the compiler must be told how much space to allocate for the array when it compiles the code.

5.2 INITIALIZING AN ARRAY

An array can be initialized simply by listing its initial values.

EXAMPLE 5.3 Initializing an Array

```
int num[4] = {22, 88, 66, 44};
```

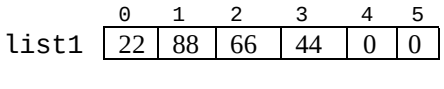


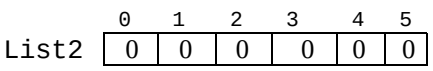
The initialized array can be visualized like this:

The number of constants in the initializer list must be less than or equal to the size of the array. If it has fewer, then the remaining elements will be initialized to 0 automatically.

EXAMPLE 5.4 Using the Default Value 0

```
int list1[6] = {22, 88,
66, 44};
int list2[6] = {0};
```





Unlike the fundamental (atomic) types (`char, int, float, double`, *etc.*), arrays are composites, consisting of several values. Consequently, many of the operations used on fundamental types do not work as expected with arrays:

```
int list3[6] = listl;  // ILLEGAL initialization!
list2 = listl;         // DOES NOT WORK as expected
if (listl == list2) .. // DOES NOT WORK as expected
cin >> listl;          // ILLEGAL extraction
listl += 2;            // ILLEGAL arithmetic/assignment
```

5.3 DUPLICATING AN ARRAY

Arrays cannot be assigned:

```
list2 = listl;           // ILLEGAL!
list2 = {22, 88, 66, 44);     // ILLEGAL!
```

The reason that these statements are illegal is that an array name itself is actually a constant, and it is illegal for a constant to be on the left side of an assignment statement.

The best way to copy one array into another is with a traversing `for` loop:

EXAMPLE 5.5 Copying an Array

This uses the array `x` defined in Example 5.2:

```
float y[SIZE];
for(int i=0; i<SIZE; i++)
    y[i] = x[i];
```

The `for` loop makes `y` a duplicate of `x.`

5.4 CONSTANT ARRAYS

Like fundamental types, arrays may be declared to be constant. And like any other constant type, a constant array must be initialized.

EXAMPLE 5.6 Computing the Month for a Given Day of the Year

This code uses a constant array that keeps track of the number of days in each month (in non-leap years). It also uses a constant integer and twelve constants defined indirectly in the enumeration type:

```
Const int DAYS_IN_MONTH[13] =
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
enum Month
    {JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
const int DAYS_IN_YEAR = 365;
Month month(int day_of_year)
{   assert(day_of_year >= 1 && day_of_year <= DAYS_IN_YEAR);
    int days = 0;
    for (Month m = JAN;  m <= DEC; m = Month(m+1))
    {   if(days >= day_of_year) return Month(m-l);
        days += DAYS_IN_MONTH[m];
    }
    return DEC;
}

int main()
{   for(int d=l; d<=DAYS_IN_YEAR; d += 14) // check each fortnight
        cout << "month(" << d << ") = " << month(d) << endl;
}
```

The function works by accumulating days one month at a time until the number exceeds the given number of days. Then the last `DAYS_IN_MONTH[m]` added should be the month after the current month, so it returns the preceding month.

Here is the output from this program:

```
month(1) = 1
month(15) = 1
month(29) = 1
month(43) = 2
   :
month(323) = 11
month(337) = 12
month(351) = 12
month(365) = 12
```

There were 18 more lines (replaced here by the colon). This shows, for example, that the 43rd day of the year is in February and that the 323rd day of the year is in November.

5.5 ARRAY INDEX OUT OF RANGE

One of the problems with arrays in C++ is that the compiler will not check the values of an array index variable to ensure that it remains within range.

EXAMPLE 5.7 Array Index Out of Range

```
double list[6] = {101.01, 202.02, 303.03, 404.04, 505.05, 606.06};
for(int i=0; i<1000; i++)
    cout << list[i] << endl;
```

This code will compile without error, even though `list[6], list[7],..., list[999]` are not defined! Of course, when it runs, it is likely to crash. But the compiler will be of no help in revealing this error. Here is what happened when this code was run on a UNIX system:

```
101.01
202.02
303.03
404.04
505.05
606.06
7.82007e-320
2.15515e-314
   :
1,03302e-47
Segmentation fault
```

The system printed 110 lines of numbers, the first 8 and the last of which are shown here. Obviously 104 of these numbers are "garbage;" *i.e.*, the result of the system trying to interpret random bit strings as floating-point numbers. The output Segmentation fault is a message from the system that it reached the end of the process's memory segment. That is the block of computer memory reserved for that particular program.

In C++, it is the responsibility of the programmer to ensure that the index of an array stays within its bounds. Other programming languages *(e.g.,* Pascal and Java) force the compiler to do that checking for the programmer. That slows the compilation process. Bjarne Stroustrup, the inventor of C++, intentionally left that responsibility to the programmer to facilitate rapid software development.

Note that standard C++ does provide an equivalent vector class, described in Chapter 7, which gives the programmer the option of having the compiler do index bounds checking.

5.6 THE sizeof OPERATOR

C++ provides a special operator, named `sizeof`, that can be used to find the number of bytes that an object occupies. It is used like a function. For example,

```
cout << sizeof(int) << endl;
```

will print the number of bytes that any object of type **`int`** occupies in memory.

EXAMPLE 5.8 Getting the Sizes of the Fundamental types

Here is a complete C++ program:

```
#include <iostream>
using namespace std;
int main()
{   cout << "sizeof(char)     = " << sizeof(char) << endl;
    cout << "sizeof(short)    = " << sizeof(short) << endl;
    cout << "sizeof(int)      = " << sizeof(int) << endl;
    cout << "sizeof(long)     = " << sizeof(long) << endl;
    cout << "sizeof(float)    = " << sizeof(float) << endl;
    cout << "sizeof(double)   = " << sizeof(double) << endl;
}
```

Here is its output when run on a UNIX workstation:

```
sizeof(char)     = 1
sizeof(short)    = 2
sizeof(int)      = 4
sizeof(long)     = 4
sizeof(float)    = 4
sizeof(double)   = 8
```

This shows, for example, that on this system an `int` uses `4` bytes.

The `sizeof` operator can also be used to find the total number of bytes that an array uses. Of course, that number is simply the product of the number of elements in the array with size of its element type:

`sizeof` (*array*) = (*number of elements*) `x sizeof` (*element type*)

EXAMPLE 5.9 Getting the Sizes of Arrays

```
int main()
{   char chars[10] = {0};
    short shorts[10] = {0};
    float floats[10] = {0.0};
    double doubles[10] = {0.0};
    cout << "sizeof(chars) = " << sizeof(chars) << endl;
    cout << "sizeof(shorts) = " << sizeof(shorts) << endl;
    cout << "sizeof(floats) = " << sizeof(floats) << endl;
    cout << "sizeof(doubles) = " << sizeof(doubles) << endl;
}
```

Here is its output when run on the same UNIX workstation:

```
sizeof(chars) = 10
sizeof(shorts) = 20
sizeof(floats) = 40
sizeof(doubles) = 80
```

This shows, for example, that an array of 10 elements of type `double` occupies 80 bytes.

5.7 PASSING AN ARRAY TO A FUNCTION

An array is passed to a function in the same way any other variable is passed, except that the array name must be followed by a pair of empty brackets to indicate that it is the name of an array.

EXAMPLE 5.10 A Function to Add the Elements of an Array

```
int sum(int a[], int n)
// Returns the sum of the first n elements in the array:
// a[0] + a[1] + a[2] + ... + a[n-l];
// PRECONDITIONS: n >= 0, n <= number of elements in the array
{   int s=0;
    for(int i=0; i<n; i++) s += a[i];
    return s;
}
```

When an array is passed to a function, it is actually only the array name that is passed. This name is actually a constant which contains the memory address of the first element (`a[0]`) of the array. Since array names are constants, they are always passed by value. That does not prevent the function from changing the elements in the array. They remain accessible and changeable by the function.

Note that there is no way to determine from within the function what the size of the array is. The `sizeof` operator won't help because the function parameter `a` is not an array; it is actually only the address of the first element of the array (which is all the information about the array that the function needs in order to carry out its task). So the expression `sizeof(a)` will evaluate to whatever number of bytes the system uses to store a memory address, probably 4.

5.8 APPLICATIONS OF ARRAYS

A *counter* is an integer variable that is used to count objects.

EXAMPLE 5.11 Counting Primes

Here is a complete C++ program that uses the `is_prime` () from Example 4.10:

```
#include <assert>
#include <iostream>
using namespace std;
bool is_prime(int n);
// Returns true iff n has no divisors except 1 and itself
int main()
{   int primes = 0;
    for(int n=l; n<1000; n++)
    if(is_prime(n)) ++primes;
    cout << "There are " << primes << " primes between 1 and 1000.\n";
}
```

The output is

```
There are 168 primes between 1 and 1000.
```

Here, the variable `primes` is a counter. It is initialized to be 0 and then it is incremented every time the `is_prime()` function returns true. That happened 168 times.

An array of counters is called a *frequency tally*. It is used to count elements in different categories so that their frequencies can be compared.

EXAMPLE 5.12 A Frequency Tally

Here is a list of test scores:

```
88 71 90 75 77 88 85 73 94 80
89 66 41 98 90 82 84 70 63 87
80 60 79 75 94 56 78 70 81 77
94 80 75 62 77 95 80 56 88 92
```

The following program reads all the scores and counts how many are in each of the following grade ranges: 90 – 100 for an A, 80 – 89 for a B, 70 – 79 for a C, 60 – 69 for a D, and 0 – 59 for an F. It keeps all five of these counts together in the `freq[ ]` array. Then it prints both the absolute and the relative frequencies of each grade,

```
enum {A, B, C, D, F};
int main()
{   int freq[5] = (0}, grade;
    cin >> grade;
    while(!cin.eof())
    {   assert(grade >= 0 && grade <= 100);
        if(grade >= 90) ++freq[A];
        else if(grade >= 80) ++freq[B];
        else if(grade >= 70) ++freq[C];
        else if(grade >= 60) ++freq[D];
        else ++freq[F];
        cin >> grade;
    }
    float total = freq[A] + freq[B] + freq[C] + freq[D] + freq[F];
    cout << "total = " << total << endl;
    cout << "A: " << frequ[A] << " = " << 100.0*freq[A]/total << "%\n";
    cout << "B: " << frequ[B] << " = " << 100.0*freq[B]/total << "%\n";
    cout << "C: " << frequ[C] << " = " << 100.0*freq[C]/total << "%\n";
```

```
    cout << "D: " << frequ[D] << " = " << 100.0*freq[D]/total << "%\n";
    cout << "E: " << frequ[E] << " = " << 100.0*freq[E]/total << "%\n";
}
```

The while loop continues as long as there is more input to be read through the cin input stream object. If the input is being input interactively on a UNIX system, then pressing `<Ctrl-D>` will send an end-of-file signal into the stream object which will then terminate the loop.

Notice the use of the anonymous enumeration type: **`enum`** `{A, B, C, D, F}`. This simply defines the five constants A, B, C, D, and E and gives them the default values 0, 1, 2, 3, and 4.

The next application is an implementation of an algorithm that is attributed to the ancient Greek astronomer Eratosthenes of Cyrene (c. 276-194 B.C.). He was the first person to have calculated an accurate estimate of the circumference of the Earth.

Algorithm 5.1 The Sieve of Eratosthenes

To obtain a list of all the prime numbers $(2, 3, 5, 7, \cdots)$ that are less than a given bound `max`:

1. Initialize an array `prime[max]` of `bools` to be all `true` except the first two.
2. Set `prime [2*j] = false` for all `j > 1` for which `2*j < max.`
3. Set `prime [3*j] = false` for all `j > 1` for which `3*j < max.`
4. Repeat setting `prime [p*j] = false` for all `j > 1` for which `p*j < max` for each prime number `p` (= 5, 7, 11, *etc.*). The next prime will be the next `i` for which `prime[i]` is `true.`
5. When Step 4 is finished, the values of `i` for which `prime [i]` is `true` are the primes.

EXAMPLE 5.13 The Sieve of Eratosthenes

```
int main()
{   const int MAX = 1000;
    bool prime[MAX];
    prime[0] = prime[1] = false;
    for(int i=2; i<MAX; i++) prime[i] = true;
    for(int j=2; 2*j < MAX; j++)     // even numbers > 2 are not prime
        prime[2*j] = false;
    int p = 3;
    while(p <= MAX/2)
    {   for (int j=2; p*j < MAX; j++) // multiples of p are not prime
            prime[p*j] = false;
        do ++p
        while(!prime[p]);     // set p = next prime
    }
    for(i=2; i<MAX; i++)
    {   if(prime[i]) cout << i << " "; // print the primes
        if(i%80 == 0) cout << endl;  // avoid line wrap-around
    }
    cout << endl;
}
```

Note the effect of the `do` loop: it repeatedly increments `p` until `prime[p]` is true; *i.e.*, until `p` is a prime number again. Since `p` is already a prime when the loop begins, it must increment `p` before it evaluates `prime[p]`; so it must use the preincrement operator `++p.` The loop could also be written as

```
do
while (!prime[++p]);
```

or even more simply as

```
while (!prime[++p]);
```

Both of those versions use the *empty statement* inside the loop.

Here is the output from the program:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79
83 89 97 101 103 107 109 113 127 131 137 139 149 151 157
163 167 173 179 181 191 193 197 199 211 223 227 229 233 239
241 251 257 263 269 271 277 281 283 293 307 311 313 317
331 337 347 349 353 359 367 373 379 383 389 397
401 409 419 421 431 433 439 443 449 457 461 463 467 479
487 491 499 503 509 521 523 541 547 557
```

```
563 569 571 577 587 593 599 601 607 613 617 619 631
641 643 647 653 659 661 673 677 6/83 691 701 709 719
727 733 739 743 751 757 761 769 773 787 797
809 811 821 823 827 829 839 853 857 859 864 877
881 883 887 907 911 919 929 937 941 947 953
967 971 977 983 991 997
```

Algorithm 5.2 The Bubble Sort

To sort a list of numbers into nondecreasing order:

1. Traverse the list, swapping adjacent pairs whenever they are out of order. The result will be that the largest element in the list is moved to the last position.
2. Repeat Step 1 except only on the sublist that omits the last element. The result will be that the largest element in the sublist is moved to the second-from-last position.
3. Repeat Step 2 *n*–3 more times, each time on the sublist that omits the last element of the previous sublist and those after it. The result each time will be that the largest element in the sublist is moved to the end of that sublist.

EXAMPLE 5.14 The Bubble Sort for Arrays of `int` type

```
void sort(int a[], int n)
{   for (int i=l; i < n; i++)
    for (int j=l; j <= n-i; j++)
        if(a[j-1] > a[j]) swap(a[j-1], a[j]);
    // INVARIANT: a[n-i], ..., a[n-l] are in their correct positions
}
```

i	j	a[0]	a[l]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
		44	88	77	22	55	99	66	33
1	2		77	88					
	3			22	88				
	4				55	88			
	6						66	99	
	7							33	99
2	2		22	77					
	3			55	77				
	5					66	88		
	6						33	88	
3	1	22	44						
	4				66	77			
	5					33	77		
4	4				33	66			
5	3			33	55				
6	2		33	44					

Here is the `swap()` function that this algorithm uses:

```
void swap(int& x, int& y)
{   float t = x;
    x = y;
    y = t;
}
```

Note that the parameters `x` and `y` are passed by reference since the function must change them.

A trace of the call `sort(a, 8)` on the array `a[ 8 ] = {44, 88, 77, 22, 55, 99, 66, 33}` is shown above. Note the correctness of the loop invariant: After the third iteration of the main loop `(i == 3)`, the order of the elements is `{22, 44, 55, 66, 33, 77, 88, 99}` so elements `a[5]`, `a[6]`, and `a[7]` (`77`, `88`, and `99`) are in their correct positions.

EXAMPLE 5.15 Reversing an Array

This function reverses the order of the first `n` elements in the array:

```
void reverse(float a[], int n)
{   for(int i=0; i < n/2; i++)
    swap(a[i], a[n-i-l]);
}
```

Here is the `swap()` function that this algorithm uses:

```
void swap(float& x, float& y)
{   float t = x;
    x = y;
    y = t;
}
```

Note that the only difference between this and the `swap()` function used in Example `5.14` is the type of its parameters: this swaps floats, the other swaps ints. This inefficiency of having two functions that do the same is remedied by means of a function template, described in Chapter 10.

5.9 TWO-DIMENSIONAL ARRAYS

A *two-dimensional array* is an array that has two independent subscripts. The syntax for its declaration is <*type*> <*name*>[<*num-rows*>][<*num-columns*>]; Individual elements are accessed the same way as with one-dimensional arrays.

EXAMPLE 5.16 A Two-Dimensional Array

```
int m[3][6]; // m has 18 elements, arranged in 3 rows and 6 columns
m[0][4] = 88; // assigns 88 to the element in row 0 and column 4
m[2] [1] = 44; // assigns 44 to the element in row 2 and column 1
```

The object `m` could be visualized as shown at right. Note that its rows are numbered `0-2` and its columns are numbered `0-5`.

m	0	1	2	3	4	5
0					88	
1						
2		44				

A two-dimensional array can be imagined as an array of (one-dimensional) arrays. That point of view helps to understand the way that a two-dimensional array is initialized. Its initialization list has to be a list of lists.

EXAMPLE 5.17 Initializing a Two-Dimensional Array

The code below initializes 9 of the 18 elements to non-zero values, as shown at right. Note that the initialization can be regarded as a list of rows.

m	0	1	2	3	4	5
0	77	55	11	33	88	0
1	99	0	0	0	0	0
2	66	44	22	0	0	0

```
int m[3][6] = { {77, 55, 11, 33, 88}, {99}, {66, 44, 22} };
```

EXAMPLE 5.18 Processing Test Scores

Here is a complete C++ program that processes student test scores:

```
#include <iostream>
using namespace std;

const STUDENTS = 4;
const TESTS = 5;

typedef int Table[STUDENTS][TESTS] ;

void get(Table);
void print(const Table);
void print_test_averages(const Table);
void print_class_averages(const Table);

int main()
{   Table scores;
    get(scores);
    print(scores);
    print_test_averages(scores);
    print_class_averages(scores);
}

void get(Table x)
{    for(int s = 0; s < STUDENTS; s++)
        for(int t = 0; t < TESTS; t++) cin >> x[s][t];
```

```
}
void print(const Table scores)
{   cout << "Test scores:\n";
    for(int s = 0; s < STUDENTS; s++)
    {   for(int t = 0; t < TESTS; t++)
            cout << scores[s][t] << "\t";
        cout << endl;
    }
}

void print_test_averages(const Table scores)
{   cout << "Test averages:\n";
    for(int s = 0; s < STUDENTS; s++)
    {   float sum = 0.0;
        for(int t = 0; t < TESTS; t++)
        sum += scores[s][t];
        cout << "\tStudent " << s << ": " << sum/TESTS << endl;
    }
}

void print_class_averages(const Table scores)
{   cout << "Class averages:\n";
    for(int t = 0; t < TESTS; t++)
    {   float sum = 0.0;
        for(int s = 0; s < STUDENTS; s++)
            sum += scores[s][t];
        cout << "\tTest " << t << ": " << sum/STUDENTS << endl;
    }
}
```

With this input

```
89 70 92 95 83
75 88 80 52 68
83 90 90 80 93
81 75 77 84 85
```

the output is

```
Test scores:
 89       70       92       95       83
 75       88       80       52       68
 83       90       90       80       93
 81       75       77       84       85
Test averages:
      Student 0: 85.8
      Student 1: 72.6
      Student 2: 87.2
      Student 3: 80.4
Class averages:
      Test 0: 82
      Test 1: 80.75
      Test 2: 84.75
      Test 3: 77.75
      Test 4: 82.25
```

5.10 MACHINE STORAGE OF ARRAYS

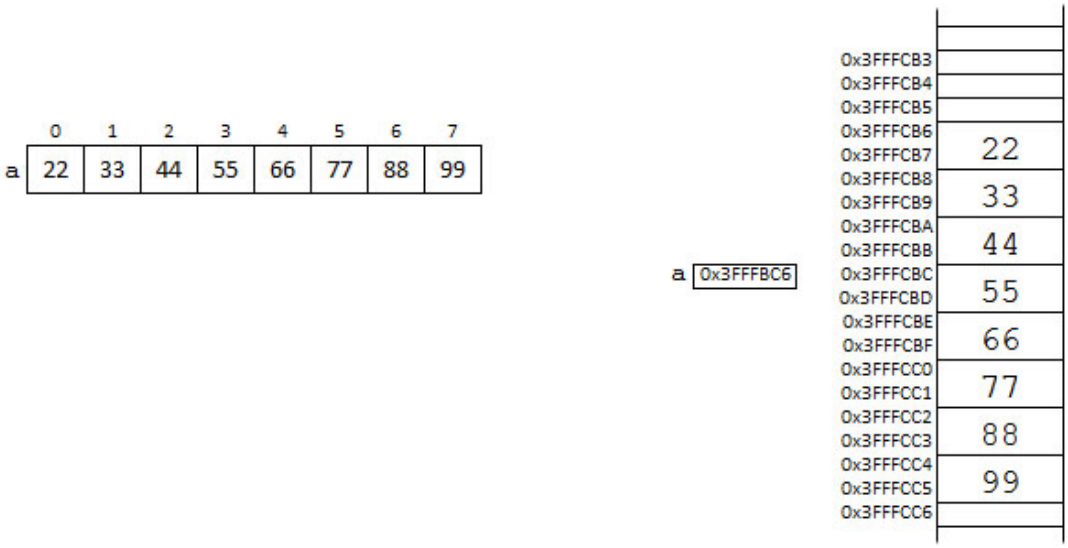
Machine storage is linear. Main memory can be regarded as a very long (one-dimensional) array of bytes. These bytes are accessed by their addresses, which are expressed in hexadecimal notation.

EXAMPLE 5.19 The Storage of an Array

Consider the array a defined as

```
short a[8] = {22, 33, 44, 55, 66, 77, 88, 99};
```

Below are two different ways to visualize the array:



The drawing on the right shows a small segment of memory: 20 bytes, with addresses from `0x3fffcb3` to `0xfffcc6`. The array contains 8 elements, each a 2-byte short integer, so the array occupies a total of 16 bytes. The picture shows it occupying bytes `0x3fffcb6-0x3fffcc5`. The object `a` itself actually contains only a single memory address: the address `0x3fffcb6` where the array begins.

When an element of an array is accessed, the system computes the address of that element by adding an offset to the base address that is stored in the array name object. The *offset* of an element `a[p]` is simply the difference between the address of the element and the address of the first element `a[0]`. It is the number of bytes that the system adds to the starting byte to compute the address of the element `a[p]`. For a one-dimensional array, the formula is

offset = (p)x (*size of element*)

EXAMPLE 5.20 Computing an Array Offset

Suppose the following statement executes for the array defined in Example 5.19:

```
a[5] = 11;
```

The offset for this element access is computed as

offset = (*element subscript*) x (*size of element*) = (5) x (2 bytes) = 10 bytes

Then the address of the element is computed by adding the offset to the base address stored in `a`:

address of `a[5] = a` + *offset* = `0x3fffcb6 + 10 = 0x3fffcc0`

This is the internal computation made by the system to locate `a[5]` in memory.

For a two-dimensional array, the offset has to include the number of bytes occupied by the preceding rows. For an array with `n` columns, the offset for the element `a[p][q]` is

offset = `(n*p + q)` x (*size of element*)

This depends on the number of columns, but not on the number of rows in the array.

EXAMPLE 5.21 The Offset for a Two-Dimensional Array

Here is an array of 70 `doubles`, each element occupying 8 bytes:

```
double x[7][10] = {0}; // x has 7 rows and 10 columns
x[6][3] = 999.99;      // there are 6 full rows above x[6][3]
```

The offset for memory location of the element `x[6][3]` is computed by the system this way:

offset = (10*6 + 3) x (8 bytes) = (63) x (8 bytes) = 504 bytes

So the actual location of element `x[6][3]` in memory is 504 bytes past the address stored in `x`.

Review Questions

5.1 Why are arrays usually processed with `for` loops?

5.2 Why doesn't the `sizeof` operator give the correct size of an array passed to a function?

Problems

5.3 Suppose that x is the array declared as

```
double x[8] = {0};
```

a. Compute the offset for the reference `x[5]`.

b. If `x` contains the address `0x3fffcb6`, what is the memory address of `x[5]`?

5.4 Suppose that `a` is the array declared as

```
int a[8][5] = {0};
```

a. Compute the offset for the reference `a[7][3]`.

b. If `a` contains the address `0x3fff000`, what is the memory address of `a[7][3]`?

Programming Problems

5.5 Implement the following `copy()` function:

```
void copy(float y[], float x[], int n);
// Copies the first n elements of the array x into the array y;
// PRECONDITION: x and y both have at least n elements.
// POSTCONDITION: x[i] == y[i] for 0 <= i < n.
```

5.6 Implement the following `are_equal()` function:

```
bool are_equal(float x[], float y[], int n);
// Returns true iff x[i] == y[i] for 0 <= i < n.
// PRECONDITION: x and y both have at least n elements.
```

5.7 Modify the program in Example 5.11 so that you can input an integer `max` and then it will find the number of primes that are less than `max`. Then use it to find how many primes are less than 2000 and how many primes are less than 5000.

5.8 Implement the following `mean()` function:

```
float mean(float x[], int n);
// Returns mean average of the first n elements of the array x.
// PRECONDITION: x has at least n elements
// EXAMPLE: if x[] = (2.2, 8.8, 4.4, 6.6}, mean(x,4) returns 5.5
```

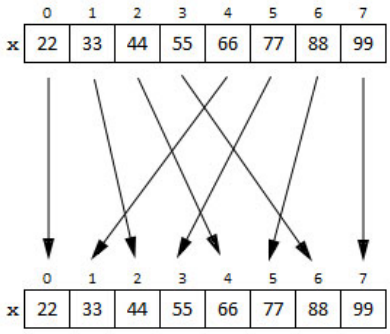
5.9 Implement the following `max()` function:

```
float max(float x[], int n);
// Returns the largest among the first n elements of array x.
// PRECONDITION: x has at least n elements
// EXAMPLE: if x[] = {2.2, 8.8, 4.4, 6.6}, max(x,4) returns 8.8
```

5.10 Implement the following `float()` function:

```
void insert(float x[], int n, float t);
// Inserts t into the sorted array, maintaining its order
// PRECONDITION: x[0] <= x[l] <= … <= x[n-1]
// PROSTCONDITION: x[0] <= x[l] <= … <= x[n-1] <= x[n]
// EXAMPLE: if x[] = {2.2, 4.4, 6.6, 8.8} then insert(x, 4, 5.5)
// changes x to {2.2, 4.4, 5.5, 6.6 8.8}
```

5.11 The *perfect shuffle* of an array interleaves its first half with its second half, like this:



Implement the following perfect shuffle() function:

```
void shuffle(Array a);
// Performs a perfect shuffle of the first SIZE
// PRECONDITION: a has at least SIZE elements
// POSTCONDITION: a[i] -> a[2*i] and a[SIZE/2+i] -> a[2*i+1]
// for 0 <= i < SIZE/2
// EXAMPLE: if a[] = {22, 33, 44, 55, 66, 77, 88, 99} then
// shuffle(a, 8) changes a to {22, 55, 33, 66, 44, 77, 55, 88}
```

Use the definitions:

```
const int SIZE = 8;
typedef int Array[SIZE];
```

5.12 Write a program that determines empirically the minimum number of perfect shuffles required to restore an array to its original order. Use your `copy()` function from Problem 5.5, your `shuffled()` function from Problem 5.11, and your `are_equal()` from Problem 5.6.

5.13 Implement the following `rotated()` function:

```
void rotate(Array a, int k);
// Shifts the left-most SIZE-k elements k places to the right
// and wraps the right-most k elements around to the left.
// PRECONDITION: 0 <= k < SIZE
// POSTCONDITION: a[i] -> a[i+k] for 0 <= i < SIZE-k,
// and a[i] -> a[i-SIZE+k] for SIZE-k <= i < SIZE
// EXAMPLE: if a[] = {22, 33, 44, 55, 66, 77, 88, 99} then
// shift(a, 3) changes a to {77, 55, 88, 22, 55, 33, 66, 44}
```

5.14 Write a program that reads characters and then prints the first most frequent character read. For example, if the input is

```
Master Pangloss taught the metaphysico-theologo-cosmolonigology. He could prove to
admiration that there is no effect without a cause; and, that in this best of all possible
worlds, the Baron's castle was the most magnificent of all castles, and My Lady the best
of all possible baronesses.
```

Then the output should be

```
The first most frequent letter was t It occurred 26 times.
```

5.15 Implement the following `std_dev()` function:

```
float std_dev(float x[], int n);
// Returns the standard deviation of the first n elements of x.
// PRECONDITION: x has at least n elements
```

The formula for the standard deviation of a sequence $\{x_0, x_1, \cdots, x_{n-1}\}$ of *n* numbers is

$$\frac{\sqrt{\sum_{i=0}^{n-1}(x_i - \mu)^2}}{n-1}$$

where μ is the mean average of the *n* numbers. For example, if *n* = 3, then the standard deviation of $\{x_0, x_1, x_2\}$ is

$$\sqrt{\frac{(x_0 - \mu)^2 + (x_1 - \mu)^2 + (x_2 - \mu)^2}{2}}$$

Use your `mean()` function from Problem 5.8 for μ, and use the `<cmath>` header file or your own implementation of the Babylonian algorithm for the `sqrt()` function.

5.16 Modify the Bubble Sort so that it is "smart" enough to stop when the array is sorted. Use a *flag* (*i.e.*, a `boo`lean variable) named `sorted` that is set `true` at the start of each iteration of the outer loop and then gets reset `false` inside the inner loop whenever the `swap ()` condition is `true`. Then use that flag to control the outer loop.

5.17 Implement the following `quintile()` function:

```
float quintile (float x[], int n, int q);
// Returns the qth quintile of the first n elements of x.
// PRECONDITION: x has at least n elements
// POSTCONDITION: (20*q)% of the a[i] are <= the value returned
```

First sort the array. (Use your "smart" Bubble Sort from Problem 5.16.) Then compute the `stop` value, which the boundary for `20q` percent of the numbers. Then use a loop which traverses the sorted array and `returns` when the values exceed your `stop` value.

5.18 The *trace* of a square matrix *(i.e.*, a two-dimensional array with the same number of rows as columns) is the sum of its diagonal elements. For example, the trace of the matrix

```
4.4  6.6  7.7  3.3  2.2
7.7  5.5  8.8  5.5  6.6
8.8  2.2  1.1  2.2  1.1
3.3  3.3  6.6  4.4  7.7
6.6  4.4  9.9  5.5  3.3
```

is `4.4 + 5.5 + 1.1 + 4.4 + 3.3 = 18.7`. Implement the following `trace()` function:

```
float trace(Matrix p);
// Returns the sum of the diagonal elements:
// p[0][0] + p[l][l] + ••• + p[SIZE-l][SIZE-1].
```

5.19 The *transpose* of a matrix (a two-dimensional array) is obtained by interchanging the elements that are are symmetrically opposite the diagonal. For example, the transpose of the matrix

```
4.4  6.6  7.7  3.3  2.2
7.7  5.5  8.8  5.5  6.6
8.8  2.2  1.1  2.2  1.1
3.3  3.3  6.6  4.4  7.7
```

is

```
4.4  7.7  8.8  3.3
6.6  5.5  2.2  3.3
7.7  8.8  1.1  6.6
3.3  5.5  2.2  4.4
2.2  6.6  1.1  7.7
```

Implement the following `transpose()` function:

```
void transpose(Matrix m);
// Transposes the matrix by swapping the elements that are
// symmetrically opposite the diagonal: m[i][] <-> m[j][i].
```

5.20 In the theory of games and economic behavior, founded by John von Neumann, certain two-person games can be represented by a single two-dimensional array, called the *payoff matrix.* Players can obtain optimal strategies when the payoff matrix has a saddle point. A *saddle point* is an entry in the matrix that is both the minimax and the maximin. The *minimax* of a matrix is minimum of the column maxima, and the *maximin* is the maximum of the row minima. The optimal strategies are possible when these two values are equal. Write a program that prints the minimax and the maximin of a given matrix.

5.21 Pascal's Triangle looks like this:

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5  10  10   5   1
1   6  15  20  15   6   1
```

Each interior number is the sum of the one above it and the one above and to its left. For example, 20 = 10 + 10. Write a complete C++ program that computes Pascal's triangle down to row number 14, stores it in a `15x15` matrix, and then prints the non-zero part of it.

Solutions

5.1 A `for` loop easily traverses an array using its control variable for the array subscript. For example:

```
for(int i=0; i<n; i++)
    cout << a[i] << "\t";
```

This prints the first `n` elements of the array `a: a[0], a[l],..., a[n-l].`

5.2 When an array is passed to a function, the only information about the array that the function receives is the array's element type and the memory address of its first element. So the function does not know how many elements the array has, and thus is unable to determine its size.

5.3 The array's element type is `double,` which occupies 8 bytes.

a. `The offset for the reference x[5] is 5*8 = 40.`

b. If `x` contains the address `0x3fffcb6`, then the memory address of `x[5]` is `0x3fffcb6 + 40 = 0x3fffcb6 + 0x28 = 0x3fffcde`

5.4 The array's element type is `int,` which we assume occupies 4 bytes.

a. `The offset for the reference a[7][3] is (7*5 + 3)*4 = 38*4 = 152`

b. If the array `a` contains the address `0x3fff000,` then the memory address of `a[7][3]` is `0x3fff000 + 152 = 0x3fff000 + 0x98 = 0x3fff098`

5.5
```
void copy(float b[], float a[], int n)
{   for(int i=0; i<n; i++)
        b[i] = a[i];
}
```

5.6
```
bool are_equal(float a[], float b[], int n)
{   for(int i=0; i<n; i++)
        if(a[i] != b[i]) return false;
    return true;
}
```

5.7
```
bool is_prime(int n)
// Returns true iff n has no divisors except 1 and itself
{   assert(n > 0);
    if(n == 1) return false;
    if(n == 2) return true;        // 2 is the first prime
    if(n%2 == 0) return false;           // 2 is the only even prime
    for(int d=3; d<n; d += 2)            // look for an odd divisor
         if (n%d == 0) return false;
    return true;                         // no odd divisors were found
}

int main()
{   int max;
    cout << "Enter upper bound: ";
    cin >> max;
    int primes = 0;
    for(int n=l; n<max; n++)
        if(is_prime(n)) ++primes;
    cout << "There are " << primes << max << endl;
}
```

There are 303 primes less than 2000, and there are 669 primes less than 5000.

5.8
```
float mean (float a[], int n)
{   float sum = 0.0;
    for(int i=0; i<n; i++)
        sum += a[i];
    return sum/n;
}
```

5.9
```
float max(float a[], int n)
{   int m = 0;
    for(int i=l; i<n; i++)
        if(a[i] > a[m]) m = i;
    return a[m];
}
```

5.10
```
void insert(float x[], int n, float t)
{   for(int i=n; i>0 && x[i-l] > t; i--)
        x[i] = x[i-l]; // shift larger elements up
    x[i] = t;
}
```

5.11
```
void shuffle(Array a)
{   int temp[SIZE];
    const half = SIZE/2;
    for(int i=0; i<half; i++)
    {   temp[2*i] = a[i];
        temp[2*i+l] = a[half+i];
    }
    for(int i=0; i<SIZE; i++)
        a[i] = temp[i];
}
```

5.12
```
const int SIZE = 6;
typedef int Array[SlZE];

void copy(Array y, Array x);
// Copies the first SIZE elements the array x into array y;
// PRECONDITION: x and y both have at least SIZE elements.
// POSTCONDITION: x[i] == y[i] for 0 <= i < SIZE.

void shuffle(Array a);
// Implements the perfect shuffle of first SIZE elements of a.
// EXAMPLE: if a[] = {22, 33, 44, 55, 66, 77, 88, 99}, then
// shuffle(a, 8) changes a to {22, 55, 33, 66, 44, 77, 55, 88}.
// PRECONDITION: a has at least SIZE elements.
// POSTCONDITION: a[i] -> a[2*i] and a[SIZE/2+i] -> a[2*i+l]
// for 0 <= i < SIZE/2.

bool are_equal(Array a, Array b);
// Returns true iff x[i] == y[i] for 0 <= i < SIZE.
// PRECONDITION: x and y both have at least SIZE elements.

void print(Array a);
// Prints the first SIZE elements of the array a.
// PRECONDITION: a has at least SIZE element.

int main()
{   Array a = {22, 33, 44, 55, 66, 77};
    Array b;
    cout << "a =\t";
    print(a);
    copy(b, a);
    cout << "b =\t";
    print(b);
    int count = 0;
    do
    {   shuffle(b);
        ++count;
        cout << count << " : \t";
        print(b);
     } while(!are_equal(a, b));
     cout << "It took " << count << " shuffles to restore the "
          << SIZE << "-element array to its original state.\n";
}

void copy(Array b, Array a)
{   for(int i=0; i<SIZE; i++)
```

```
    b[i] = a[i];
}
void shuffle(Array a)
{   int temp[SIZE];
    const half = SIZE/2;
    for(int i=0; i<half; i++)
    {   temp[2*i] = a[i];
        temp[2*i + 1] = a[half+i];
}
for(int i = 0; i<SIZE; i++)
    a[i] = temp[i];
}
bool are_equal(Array a, Array b)
{   for(int i=0; i<SIZE; i++)
        if(a[i] != b[i]) return false;
    return true;
}
void print(Array a)
{   for(int i=0; i<SIZE; i++)
        cout << a[i] << " \t";
    cout << endl;
}
```

5.13

```
void rotate(Array a, int k)
{   Array temp;
    copy(temp, a);
    for(int i=0; i< SIZE-k; i++)
        a[i+k] = temp[i];
    for(int i=SIZE-k; i<SIZE; i++)
        a[i-SIZE + k] = temp[i];
}
```

5.14

```
int main()
{   const int SIZE = 128;
    int freq[SIZE] = {0};
    char c;
    cin >> c;
    while (!!cin)
    {   if(c >= 'a' && c <= 'z')
            ++freq[c];
        cin >> c;
    }
    char m = 'a';
    for(c='b'; c <= 'z'; c++)
    if(freq[c] > freq[m])  m = c;
    cout << "The most frequent character was " << m << endl;
    cout << "It occurred " << freq[m] << " times.\n";
}
```

5.15

```
float std_dev(float x[], int n)
{   assert(n > 1);
    float m = mean(x, n);
    float s = 0.0;
    for(int i=0; i<n; i++)
        s += (x[i] - m) * (x[i] - m);
    return sqrt(s/(n-1));
}
```

5.16

```
void sort(float x[], int n)
{   bool sorted;
    for(int i=l; i < n; i++)
    {   sorted = true;
        for(int j=l; j <= n-i; j++)
            if(x[j-l] > x[j])
            {   swap(x[j-1], x[j]);
                sorted = false;
            }
```

```
     }
}
```

5.17
```
float quintile(float x[], int n, int q)
{   sort(x, n);                // after sorting, x[n-l] == max num
    float stop = x[n-l]*q/5;   // q/5 is proportion to be <= max
    for(int i=l; i<n; i++)
         if (x[i] > stop) return x[i-l];
    return x[n-1];
}
```

5.18
```
float trace (Matrix p)
{   float sum = p[0][0];
    for(int i=l; i<SIZE; i++)
        sum += p[i][i];
    return sum;
}
```

5.19
```
void transpose(Matrix m)
{   for(int i=l; i<SIZE; i++)
        for(int j = 0; j<i; j++)
            swap(m[i][j], m[j][i]);
}
```

5.20
```
const int SIZE = 5;
typedef double Matrix[SIZE][SIZE];
typedef double Vector[SIZE];
void print(Matrix x);
// Prints the matrix x.
float minimax(Matrix x);
// Returns min(max(x[i][j]: 0 <= i < SIZE): 0 <= j < SIZE)
float maximin(Matrix x);
// Returns max(min[x[i][j]: 0 <= j < SIZE): 0 <= i < SIZE)
int main()
{   Matrix x = { {44, 66, 77, 33, 22},
                 {77, 55, 88, 55, 66},
                 {88, 22, 11, 22, 11},
                 {33, 33, 66, 44, 77},
                 {66, 44, 99, 55, 44} };
    print(x);
    cout << "The minimax is " << minimax(x) << endl;
    cout << "The maximin is " << maximin(x) << endl;
}

void print(Matrix x)
{   for(int i=0; i<SIZE; i++)
    {   for(int j=0; j<SIZE; j++)
        cout << x[i][j] << "\t";
        cout << endl;
    }
}

double col_max(Matrix x, int j)
{   double max = x[0][j];
    for(int i=l; i<SIZE; i++)
    if(x[i][j] > max)  max = x[i][j];
    return max;
}

double row_min(Matrix x, int i)
{   double min = x[i][0];
    for(int j=l; j<SIZE; j++)
    if(x[i][j] < min)  min = x[i][j];
    return min;
}

float minimax(Matrix x)
{   Vector max;
    for(int j =0; j<SIZE; j++)
        max[j] = col_max(x,j);
    double min = max[0];
```

```
    for(int j =1; j <SIZE; j++)
        if(max[j] < min)  min = max[j];
    return min;
}

float maximin(Matrix x)
{   Vector min;
    for(int i=0; i<SIZE; i++)
        min[i] = row_min(x,i);
    double max = min[0];
    for(int i=l; i<SIZE; i++)
        if(min[i] > max)  max = min[i];
    return max;
}
```

5.21

```
const int SIZE = 16;
typedef int Matrix[SIZE][SIZE];
void load_pascal(Matrix p);
// Loads the matrix so that p[i][j] = p[i-l][j-l] + p[i-l][j]

void print(Matrix);
// Prints the lower triangle of the matrix x.

int main()
{   Matrix p = {0};
    load_pascal(p);
    print(p);
}

void load_pascal(Matrix p)
{   for(int i=0; i<SIZE; i++)
        p[i][0] = p[i][i] = 1;
    for(int i=2; i<SIZE; i++)
        for(int j=l; j<i; j++)
            p[i][j] - p[i-l][j-1] + P[i-1][j];
}
void print(Matrix x)
{   setiosflags(ios::right); // right justify each number printed
    for(int i=0; i<SIZE; i++)
    {   for(int j = 0; j <= i; j++)
            cout << setw(5) << x[i][j];
        cout << endl;
    }
}
```