11

11 Arrays

Suppose that, in the example on children's heights (example in 8.5.1), we had wanted to get details of all children whose heights were more than two standard deviations from the mean. (These would be the tallest 5% and the smallest 5% of the children).

We would have had problems. We couldn't have identified these children as we read the original data. All the data have to be read before we can calculate the standard deviation and the mean. We could use a clumsy mechanism where we read the data from file once and do the first calculations, then we "rewind the file" and read the data values a second time, this time checking each entry against the mean. (This scheme sort of works for files, but "rewinding" a human user and requesting the rekeying of all interactive input is a less practical proposition.)

Really, we have to store the data as they are read so we can make a second pass through the stored data picking out those elements of interest.

But how can these data be stored?

A scheme like the following is obviously impractical:

```
int main()
{
    double average, stand_dev;
    double height1, height2, height3, height4, …,
           height197, height198, …
           height499, height500;
    cout << "Enter heights for children" << endl;
    cin >> height1 >> height2 >> height3 >> … >> height500;
```

Quit apart from the inconvenience, it is illogical. The program would have had to be written specifically for the data set used because you would need the same number of variables defined, and input statements executed, as you have values in your data set.

The beginnings in assembly language programs

Collections of data elements of the same type, like this collection of different heights, are very commonly needed. They were needed back in the earliest days of programming when all coding was done in assembly language. The programmers of

the early 1950s got their assemblers to reserve blocks of memory locations for such data collections; an "assembler directive" such as

```
HEIGHTS     ZBLOCK 500
```

would have reserved 500 memory locations for the collection of heights data. The programmers would then have used individual elements in this collection by writing code like:

```
load address register with the address where heights data start
add on "index" identifying the element required
// now have address of data element required
load data register from address given in address register
```

By changing the value of the 'index', this code accesses different elements in the collection. So, such code inside loops can access each element in turn.

High level languages and "arrays"

Such collections were so common that when FORTRAN was introduced in 1956 it made special provision for "arrays". Arrays were FORTRAN's higher level abstraction of the assembler programmers' block of memory. A FORTRAN programmer could have defined an array using a statement like:

```
DIMENSION HEIGHTS(500)
```

and then used individual array elements in statements like:

```
HEIGHTS(N) = H;
…
IF(HEIGHTS(I) .LT. HMAX) GOTO 50
…
```

The definition specifies that there are 500 elements in this collection of data (numbered 1 to 500). The expressions like `HEIGHTS(N)` and `HEIGHTS(I)` access chosen individual elements from the collection. The integers `N` and `I` are said to "index" into the array. The code generated by the compiler would have been similar to the fragment of assembler code shown above. (The character set, as available on the card punches and line printers of the 1950s and 1960s, was limited, so FORTRAN used left (and right) parentheses for many purposes including array indexing; similarly a text string such as `.LT.` would have had to be used because the `<` character would not have been available.)

Arrays with bounds checks and chosen subscript ranges

As languages evolved, improved versions of the array abstraction were generally adopted. Thus, in Pascal and similar 1970s languages, arrays could be defined to have chosen subscript ranges:

```
heights : array [1..500] of real;
…
```

```
ch_table : array['a'..'z'] of char;
```

Individual array elements could be accessed as before by subscripting:

```
heights[n] := h;
…
if(heights[i] > hmax) hmax := heights[i];
```

The compilers for these languages generally produced more complex code that included checks based on the details of the subscript range as given in the definition. The assembly language generated when accessing array elements would typically include extra instructions to check that the index was in the appropriate range (so making certain that the requested data element was one that really existed). Where arrays were to be used as arguments of functions, the function declaration and array definition had to agree on the size of the array. Such checks improve reliability of code but introduce run-time overheads and restrict the flexibility of routines.

C's arrays

When C was invented, its array abstraction reverted more to the original assembly language programmer's concept of an array as a block of memory. This was partly a consequence of the first version of C being implemented on a computer with a somewhat primitive architecture that had very limited hardware support for "indexing". But the choice was also motivated by the applications intended for the language. C was intended to be used instead of assembly language by programmers writing code for components of an operating system, for compiler, for editors, and so forth. The final machine code had to be as efficient as that obtained from hand-coded assembly language, so overheads like "array bounds checking" could not be countenanced.

Many C programmers hardly use the array abstraction. Rather than array subscripting, they use "pointers" and "pointer arithmetic" (see Part IV) to access elements. Such pointer style code is often closely related to the optimal set of machine instructions for working through a block of memory. However, it does tend to lose the abstraction of an array as a collection of similar data elements that can be accessed individual by specifying an index, and does result in more impenetrable, less readable code.

C++'s arrays

The C++ language was intended to be an improvement on C, but it had to maintain substantial "backwards compatibility". While it would have been possible to introduce an improved array abstraction into C++, this would have made the new language incompatible with the hundreds of thousands of C programs that had already been written. So a C++ array is just the same as a C array. An array is really just a block of memory locations that the programmer can access using subscripting or using pointers; either way, access is unchecked. There are no compile time, and no run time checks added to verify correct usage of arrays; it is left to the programmer to get array code correct. Inevitably, array usage tends to be an area where many bugs occur.

11.1 DEFINING ONE DIMENSIONAL ARRAYS

Simple C++ variables are defined by giving the type and the name of the variable. Arrays get defined by qualifying the variable name with square brackets `[` and `]`, e.g.:

```
double      heights[500];
```

"Vector "or "table"

This definition makes `heights` an array with 500 elements. The terms *vector* and *table* are often used as an alternative names for a singly subscripted array like `heights`.

Array index starts at zero!

The low-level "block of memory" model behind C's arrays is reflected in the convention that array elements start with 0. The definition for the `heights[]` array means that we can access `heights[0] ... heights[499]`. This is natural if you are thinking in terms of the memory representation. The first data element is where the `heights` array begins so to access it you should not add anything to the array's address (or, if you prefer, you add 0).

This "zero-base" for the index is another cause of problems. Firstly, it leads to errors when adapting code from examples written in other languages. In FORTRAN arrays start at 1. In Pascal, the programmer has the choice but zero-based Pascal arrays are rare; most Pascal programs will use 1-based arrays like FORTRAN. All the loops in FORTRAN and Pascal programs tend to run from 1..N and the array elements accessed go from 1 .. N. It is quite common to find that you have to adapt some FORTRAN or Pascal code. After all, there are very large collections of working FORTRAN routines in various engineering and scientific libraries, and there a numerous text books that present Pascal implementations of standard algorithms. When adapting code you have to be careful to adjust for the different styles of array usage. (Some people chicken out; they declare their C arrays to have N+1 elements, e.g. `double X[N+1]`, and leave the zeroth element, `X[0]`, unused. This is OK if you have one dimensional arrays of simple built in types like doubles; but you shouldn't take this way out for arrays of structures or for multidimensional arrays.)

A second common problem seems just to be a consequence of a natural, but incorrect way of thinking. You have just defined an array, e.g. `xvals[NELEMENTS]`:

```
const int   NELEMENTS     = 100;
double      xmax;
double      xvals[NELEMENTS];
double      ymax;
…
```

So, it seems natural to write something like:

```
xvals[NELEMENTS] = v;     // WRONG, WRONG, WRONG
```

Of course it is wrong, there is no data element `xvals[NELEMENTS]`. However, this code is acceptable to the compiler; the instructions generated for this assignment will be

executed at run-time without any complaint. The value actually changed by this assignment will be (probably) either `xmax`, or `ymax` (it depends on how your compiler and, possibly, link-loader choose to organize memory). A program with such an error will appear to run but the results are going to be incorrect.

In general, *be careful with array subscripts*!

Constants in array definitions

You should use named constants, or #defined values, when defining the sizes of arrays rather than writing in the number of array elements. So, the definitions:

```
const int   MAXCHILDREN = 500;
…
double      heights[MAXCHILDREN];
```

are better than the definition given at the start of this section. They are better because they make the code clearer (`MAXCHILDREN` has some meaning in the context of the program, 500 is just a "magic number"). Further, use of a constant defined at the start of the code makes it easier to change the program to deal with larger numbers of data values (if you have magic numbers, you have to search for their use in array definitions and loops).

The [] operator

Technically, the `[` and `]` brackets together form a `[]` "postfix operator" that modifies a variable in a definition or in an expression. (It is a "postfix operator" because it comes after, "post", the thing it fixes!) In a definition (or declaration), the `[]` operator changes the preceding variable from being a simple variable into an array. In an expression, the `[]` operator changes the variable reference from meaning the array as a whole to meaning a chosen element of the array. (Usually, you can only access a single array element at a time, the element being chosen using indexing. But, there are a *few* places where we can use an array as a whole thing rather than just using a selected element; these will be illustrated shortly.)

Definitions of variables of the same basic type can include both array definitions and simple variable definitions:

```
const int   MAXPOINTS = 50;
…
double      xcoord[MAXPOINTS], xmax, ycoord[MAXPOINTS], ymax,
                   zcoord[MAXPOINTS], zmax;
```

This defines three simple data variables (`xmax`, `ymax`, and `zmax`), and three arrays (`xcoord`, `ycoord`, and `zcoord`)each with 50 elements.

Local and global arrays

Arrays can be defined as local, automatic variables that belong to functions:

```
int main()
{
    double heights[MAXCHILDREN];
    …
```

Local (automatic)

But often you will have a group of functions that need to share access to the same array data. In such situations, it may be more convenient to define these shared arrays as "global" data:

Global

```
double     xcoord[MAXPOINTS], ycoord[MAXPOINTS],
                  zcoord[MAXPOINTS];

void ScalePoints(double scaling)
{
    // change all points entries by multiplying by scaling
    for(int i=0;i<MAXPOINTS;i++) {
           xcoord[i] *= scaling;
           ycoord[i] *= scaling;
           …
}

void RotatePointsOnX(double angle)
{
    // change all points to accommodate rotation about X axis
    // by angle
    …
```

Variables declared outside of any function are "globals". They can be used in all the other functions in that file. In fact, in some circumstances, such variables may be used by functions in other files.

static qualifier

You can arrange things so that such arrays are accessible by the functions in one file while keeping them from being accessed by functions in other files. This "hiding" of arrays (and simple variables) is achieved using the `static` qualifier:

"file scope"

```
static double xccord[MAXPOINTS];
static double ycoord[MAXPOINTS];
static double zccord[MAXPOINTS];

void ScalePoints(double scaling)
{
    …
```

Here, `static` really means "file scope" – restrict access to the functions defined in this file. This is an unfortunate use of the word `static`, which C and C++ also use to mean other things in other contexts. It would have been nicer if there were a separate specialized "filescope" keyword, but we must make do with the way things have been defined.

Caution on using "globals" and "file scope" variables

Some of you will have instructors who are strongly opposed to your use of global data, and even of file scope data. They have reasons. Global data are often a source of problems in large programs. Because such variables can be accessed from almost anywhere in the program there is no control on their use. Since these data values can be changed by any function, it is fairly common for errors to occur when programmers

writing different functions make different assumptions about the usage of these variables.

However for small programs, file scope data are safe, and even global data are acceptable. (Small programs are anything that a single programmer, scientist, or engineer can create in a few weeks.) Many of the simple programs illustrated in the next few chapters will make some use of file scope and/or global data. Different ways of structuring large programs, introduced in Parts IV and V, can largely remove the need for global variables.

11.2 INITIALIZING ONE DIMENSIONAL ARRAYS

Arrays are used mainly for data that are entered at run time or are computed. But sometimes, you need arrays that are initialized with specific data. Individual data elements could be initialized using the = operator and a given value:

```
long        count = 0;
long        min = LONG_MAX;
long        max = LONG_MIN;
```

Of course, if we are to initialize an array, we will normally need an initial value for each element of the array and we will need some way of showing that these initializing values form a group. The grouping is done using the same { begin and } end brackets as are used to group statements into compound statements. So, we can have initialized arrays like:

```
short grade_points[6] = { 44, 49, 64, 74, 84, 100 };
char grade_letters[6] = { 'F', 'E', 'D', 'C', 'B', 'A' };
```

As shown in these examples, an array definition-initialization has the form:

```
type array_name [ number_elements ] = {
            initial_val , initial_val , ...,
            initial_val
} ;
```

The various initializing values are separated by commas. The } end bracket is followed by a semicolon. This is different from the usage with compound statements and function definitions; there you don't have a semicolon. Inevitably, the difference is a common source of minor compilation errors because people forget the semicolon after the } in an array initialization or wrongly put one in after a function definition.

Generally, if an array has initial values assigned to its elements then these are going to be constants. Certainly, the marks and letter grades aren't likely to be changed when the program is run. Usually, initialized arrays are `const`:

```
const short grade_points[6] = { 44, 49, 64, 74, 84, 100 };
const char grade_letters[6] = { 'F', 'E', 'D', 'C', 'B', 'A' };
```

You don't have to provide initial values for every element of an array. You could have a definition and initialization like the following:

```
double     things[10] = { 17.5, 19.2, 27.1 };
```

Here, `things[0]`, `things[1]`, and `things[2]` have been given explicit initial values; the remaining seven `things` would be initialized to zero or left uninitialized. (Generally, you can only rely on zero initialization for global and file scope variables.) There aren't any good reasons why you should initialize only part of an array, but it is legal. Of course, it is an error to define an array with some specified number of elements and then try to use more initial values than will fit!

Naturally, there are short cuts. If you don't feel like counting the number of array elements, you don't have to. An array can be defined with an initialization list but no size in the `[]`; for example:

```
int mystery[] = { -3, 101, 27, 11 };
```

The compiler can count four initialization values and so concludes that the array is `mystery[4]`.

Seems crazy? It is quite useful when defining arrays containing entries that are character strings, for example a table of error messages. Examples are given in 11.7.

11.3 SIMPLE EXAMPLES USING ONE-DIMENSIONAL ARRAYS

11.3.1 Histogram

Lets get back to statistics on childrens' heights. This time we aren't interested in gender specific data but we want to produce a simple histogram showing the distribution of heights and we want to have listed details of the tallest and smallest 5% of children in the sample.

Specification:

1. The program is to read height and gender values from the file heights.dat. The height values should be in the range 100 … 200, the gender tags should be m or f. The gender tag information should be discarded. The data set will include records for at least ten children; averages and standard deviations will be defined (i.e. no need to check for zero counts).

2. The input is to be terminated by a sentinel data record with a height 0 and an arbitrary m or f gender value.

3. When the terminating sentinel data record is read, the program should print the following details: total number of children, average height, standard deviation in height.

4. Next the program is to produce a histogram showing the numbers of children in each of the height ranges 100…109 cm, 110…119 cm, …, 190…199 cm. The form of the histogram display should be as follows:

```
100   |
110   |*
120   |***
130   |******************
…
```

5. Finally, the program is to list all heights more than two standard deviations from the mean.

Program design

Preliminary design

The program breaks down into the following phases:

```
get the data
calculate mean and standard deviation
print mean and standard deviation
produce histogram
list extreme heights
```

Each of these is a candidate for being a separate function (or group of functions).

Decision in favour of "shared" data

The main data for the program will consist of an array used to store the heights of the children, and a count that will define the number of children with records in the data file. The array with the heights, and the count, will be regarded as data that are shared by all the functions. This will minimize the amount of data that need to be passed between functions.

First iteration through design

The suggested functions must now be elaborated:

```
get_data
    initialization – zeroing of some variables, opening file
    loop reading data, heights saved, gender-tags discarded

mean
    use shared count and heights array,
    loop summing values
    calculate average
```

```
std_dev
    needs mean, uses shared count and heights array
    calculate sum of squares etc and calculate
           standard deviation

histogram
    use shared count and heights array, also need
           own array for the histogram
    zero out own array
    loop through entries in main array identifying
           appropriate histogram entry to be incremented
    loop printing histogram entries

extreme_vals
    needs to be given average and standard deviation
    uses shared count and heights array

    loop checking each height for extreme value

main
    get_data
    m = mean(), print m (+ labels and formatting)
    sd = std_dev(), print sd
    histogram
    extreme_vals(m,sd)
```

Second iteration, further decomposition into simpler functions

The "get_data" and "histogram" functions both appear to be moderately elaborate and so they become candidates for further analysis. Can these functions be simplified by breaking them down into subfunctions?

Actually, we will leave the histogram function (though it probably should be further decomposed). Function getdata really involves two separate activities – organizing file input, and then the loop actually reading the data. So we can further decompose into:

```
open_file
    try to open named file,
           if fail terminate program

read_data
    read first data (height and gender tag)
    while not sentinel value
           save height data
           read next data

get_data
    initialize count to zero
    call open_file
    call read_data
```

The `ifstream` used for input from file will have to be another shared data element because it is used in several functions.

Third iteration, finalize shared data, derive function prototypes

The final stage before coding involves definition of i) constants defining sizes for arrays, ii) shared variables, and iii) the function prototypes.

```
const int MAXCHILDREN = 500;

static double s_heights[MAXCHILDREN];
static short s_count;

static ifstream s_input;

void open_file(void);

void read_data(void);

void get_data(void);

double mean(void);

double std_dev(double mean);

void histogram(void);

void extreme_vals(double mean, double sdev);

int main();
```

The shared data have been defined as `static`. As this program only consists of one source file, this "file scoping" of these variables is somewhat redundant. It has been done on principle. It is a way of re-emphasising that "these variables are shared by the functions defined in this file". The names have also been selected to reflect their role; it is wise to have a naming scheme with conventions like 's_' at the start of the name of a static, 'g_' at the start of the name for a global.

Implementation

The file organization will be:

#included headers

Definitions of constants and static (shared) variables

```
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include <assert.h>

const int MAXCHILDREN = 500;

static double s_heights[MAXCHILDREN];
static short s_count;
```

Function definitions

```
static ifstream s_input;

void open_file(void)
{
    …
}

void read_data(void)
{
    …
}

…

int main()
{
    …
    return 0;
}
```

main()

open_file()

The `open_file()` routine would specify the options `ios::in | ios::nocreate`, the standard options for a data input file, in the `open()` call. The ifstream should then be checked to verify that it did open; use `exit()` to terminate if things went wrong.

```
void open_file(void)
{
    s_input.open("heights.dat",ios::in | ios::nocreate);
    if(!s_input.good()) {
           cout << "Couldn't open the file.  Giving up."
                  << endl;
           exit(1);
    }
    return;
}
```

read_data()

The `read_data()` routine uses a standard while loop to read and process input data marked by a terminal sentinel record, (i.e. *read first record; while not sentinel record { process record, read next record }*). The data get copied into the array in the statement `s_heights[s_count] = h;`.

```
void read_data(void)
{
    double h;
    char   gender_tag;

    s_input >> h >> gender_tag;
    while(h > 0.0) {
           if(s_count < MAXCHILDREN)
                  s_heights[s_count] = h;
           else
```

Store data element in array

```
            if(s_count == MAXCHILDREN)
                   cout << "Ignoring some "
                          "data records" << endl;
            s_count++;
            s_input >> h >> gender_tag;
    }
    if(s_count>MAXCHILDREN) {                                    Report if data discarded
            cout << "The input file contained " << s_count
                          << " records." << endl;
            cout << "Only the first " << MAXCHILDREN <<
                          " will be processed." << endl;
            s_count = MAXCHILDREN;
    }
    s_input.close();
}
```

Note the defensive programming needed in `read_data()`. The routine must deal with the case where there are more entries in the file than can be stored in the array. It can't simply assume that the array is large enough and just store the next input in the next memory location because that would result in other data variables being overwritten. The approach used here is generally suitable. When the array has just become full, a warning gets printed. When the loop terminates a more detailed explanation is given about data elements that had to be discarded.

Because it can rely on auxiliary functions to do most of its work, function get_data() itself is nicely simple: ***get_data()***

```
void get_data(void)
{
    s_count = 0;
    open_file();
    read_data();
}
```

The `mean()` and `std_dev()` functions both involve simple for loops that accumulate the sum of the data elements from the array (or the sum of the squares of these elements). Note the for loop controls. The index is initialized to zero. The loop terminates when the index equals the number of array elements. This is required because of C's (and therefore C++'s) use of zero based arrays. If we have 5 elements to process, they will be in array elements [0] ... [4]. The loop must terminate before we try to access a sixth element taken from [5]. ***mean() and std_dev()***

```
double mean(void)
{
    double sum = 0.0;
    for(int i=0; i<s_count; i++)
            sum += s_heights[i];
    return sum / s_count;
}
```

```
double std_dev(double mean)
{
    double sumSq = 0.0;
    for(int i=0; i<s_count; i++)
           sumSq += s_heights[i]*s_heights[i];
    return sqrt((sumSq - s_count*mean*mean)/(s_count-1));
}
```

histogram()

Function `histogram()` is a little more elaborate. First, it has to have a local array. This array is used to accumulate counts of the number of children in each 10cm height range.

Assumptions about the particular problem pervade the routine. The array has ten elements. The scaling of heights into array indices utilizes the fact that the height values should be in the range 100 < height < 200. The final output section is set to generate the labels 100, 110, 120, etc. Although quite adequate for immediate purposes, the overall design of the routine is poor; usually, you would try to create something a little more general purpose.

Note that the local array should be zeroed out. The main array of heights was not initialized to zero because the program stored values taken from the input file and any previous data were just overwritten. But here we want to keep incrementing the counts in the various elements of the `histo[]` array, so these counts had better start at zero.

The formula that converts a height into an index number is:

```
int index = floor((s_heights[i] - 100.0)/10.0);
```

Say the height was 172.5, this should go in `histo[7]` (range 170 ... 179); (172.5-100)/10 gives 7.25, the `floor()` function converts this to 7.

The code uses `assert()` to trap any data values that are out of range. The specification said that the heights would be in the range 100 … 200 but didn't promise that they were all correct. Values like 86 or 200 would result in illegal references to `histo[-2]` or to `histo[10]`. If the index is valid, the appropriate counter can be incremented (`histo[index] ++ ;` – this applies the `++,` increment operator, to the contents of `histo[index]`).

Nested for loops

Note the "nested" for loops used to display the contents of the `histo[]` array. The outer loop, with index `i`, runs through the array elements; printing a label and terminating each line. The inner loop, with the `j` index, prints the appropriate number of stars on the line, one star for each child in this height range. (If you had larger data sets, you might have to change the code to print one star for every five children.)

```
void histogram(void)
{
    int    histo[10];
    for(int i = 0; i < 10; i++)
           histo[i] = 0;
```

Initialize elements of local array to zero

```
    for(i = 0; i < s_count; i++) {
           int index = floor((s_heights[i] - 100.0)/10.0);
           assert((index >= 0) && (index < 10));
           histo[index]++;
    }
    cout << "Histogram of heights : " << endl;
    for(i=0;i<10; i++) {
           cout << (10*(i+10)) << "   |";
           for(int j = 0; j<histo[i]; j++)
             cout << '*';
           cout << endl;

    }

}
```

Convert value to index and verify

Nested for loops

The function that identifies examples with extreme values of height simply involves a loop that works through successive entries in the `heights[]` array. The absolute value of the difference between a height and the mean is calculated and if this value exceeds twice the standard deviation details are printed.

```
void extreme_vals(double mean, double sdev)
{
    double two_std_devs = sdev*2.0;
    cout << "Heights more than two standard deviations from"
                  " mean: " << endl;
    int n = 0;
    for(int i=0;i<s_count;i++) {
           double diff = s_heights[i] - mean;
           diff = fabs(diff);
           if(diff > two_std_devs) {
                  cout << i << " : " << s_heights[i] << endl;
                  n++;
           }
    }
    if(n==0)
           cout << "There weren't any!" << endl;
}
```

Note the introduction of a local variable set to twice the standard deviation. This avoids the need to calculate `sdev*2.0` inside the loop. Such a definition is often not needed. If `two_std_devs` was not defined, the test inside the loop would have been `if(diff > sdev*2.0)` …. Most good compilers would have noticed that `sdev*2.0` could be evaluated just once before the loop; the compiler would have, in effect, invented its own variable equivalent to the `two_std_devs` explicitly defined here.

Because the real work has been broken down amongst many functions, the `main()` function is relatively simple. It consists largely of a sequence of function calls with just a little code to produce some of the required outputs.

```
int main()
{
    get_data();
    double avg = mean();
    double sd = std_dev(avg);
    cout << "Number of records processed : " << s_count <<
endl;
    cout << "Mean (average) : " << avg << endl;
    cout << "Standard deviation :" << sd << endl;
    histogram();
    extreme_vals(avg, sd);
    return 0;
}
```

11.3.2 Plotting once more

The plotting program, presented in section 10.11.2, can be simplified through the use of an array.

We can use an array to hold the characters that must be printed on a line. Most of the characters will be spaces, but the mid-point character will be the '|' axis marker and one other character will normally be a '*' representing the function value.

The changes would be in the function PlotFunctionValue(). The implementation given in section 10.11.2 had to consider a variety of special cases – the '*' on the axis, the '*' left of the axis, the '*' to the right of the axis. All of these special cases can be eliminated. Instead the code will fill an array with spaces, place the '|' in the central element of the array, then place the '*' in the correct element of the array; finally, the contents of the array are printed. (Note the -1s when setting array elements to compensate for the zero-based array indexing.)

```
void PlotFunctionValue(double value, double range)
{
    int where = floor(value/range*HALF) + HALF;

    char   line_image[WIDTH];
    for(int i=0;i<WIDTH;i++)
           line_image[i] = ' ';

    line_image[HALF-1] = '|';

    if((where>0) && (where < WIDTH))
           line_image[where-1] = '*';

    for(i=0;i<WIDTH;i++)
           cout << line_image[i];

    cout << endl;
```

Determine element to be set

Define array and fill with blanks

Place centre marker

Place value mark provided it is in range

Print contents of character array

```
    return;
}
```

This version of `PlotFunctionValue()` is significantly simpler than the original The simplicity is a consequence of the use of more sophisticated programming constructs, in this case an array data structure. Each new language feature may seem complex when first encountered, but each will have the same effect: more powerful programming constructs allow simplification of coding problems.

11.4 ARRAYS AS ARGUMENTS TO FUNCTIONS

The functions illustrated so far have only had simple variables as arguments and results. What about arrays, can arrays be used with functions?

Limitations of shared global and filescope data

Of course, you often need array data that are manipulated in different functions. The histogram example, 11.3.1, had a shared filescope (static) array for the heights and this was used by all the functions. But use of filescope, or global, data is only feasible in certain circumstances. The example in 11.3.1 is typical of where such shared data is feasible. But look at the restrictions. There is only one array of heights. All the functions are specifically designed to work on this one array.

But, generally, you don't have such restrictions. Suppose we had wanted histograms for all children, and also for boys and girls separately? The approach used in 11.3.1 would have necessitated three different variations on `void histogram(void);` there would be `c_histogram()` that worked with an array that contained details of the heights of all children, `b_histogram()` – a version that used a different filescope array containing the heights for boys, and `g_histogram()` – the corresponding function for processing the heights of girls. Obviously, this would be unacceptably clumsy.

Usually, you will have many arrays. You will need functions that must be generalized so that they can handle different sets of data, different arrays. There has to be a general purpose `histogram()` function that, as one of its arguments, is told which array contains the data that are to be displayed as a histogram. So, there needs to be some way of "passing arrays to functions".

Reasons for restrictions on arrays as arguments to functions

But arrays do present problems. A C (C++) array really is just a block of memory. When you define an array you specify its size (explicitly as in `x_coord[100]` or implicitly as in `grades[] = { 'f', 'e', 'd', 'c', 'b', 'a' };`), but this size information is not "packaged" with the data. Arrays don't come with any associated data specifying how many elements they have. This makes it more difficult for a compiler to generate instructions for functions that need argument arrays passed by value (i.e. having a copy of the array put onto the stack) or that want to return an array. A compiler might not be able to determine the size of the array and so could not work out the amount of space to be reserved on the stack, nor the number of bytes that were to be copied.

No arrays as value arguments or as results

The authors of the first C compilers decided on a simple solution. Arrays can't be passed by value. Functions can not return arrays as results. The language is slightly restricted, but the problems of the compiler writer have gone. What is more, a prohibition on the passing of arrays as value arguments or results preemptively eliminated some potentially inefficient coding styles. The copying operations involved in passing array arguments by value would have been expensive. The doubling of the storage requirement, original plus copy on the stack, could also have been a problem.

This decision restricting the use of arrays with functions still stands. Which makes C, and therefore C++, slightly inconsistent. Simple variables, struct types (Chapter 16), and class instances (Chapter 19) can all be passed by value, and results of these type can be returned by functions. Arrays are an exception.

Arrays are always passed by reference

When an array is passed as an argument to a function, the calling routine simply passes the address of the array; i.e. the caller tells the called function where to find the array in memory. The called function then uses the original array. This mechanism, where the caller tells the called function the address of the data, is called *pass by reference*.

As well as passing the array, the calling function (almost always) has to tell the called function how many elements are in the array. So most functions that take arrays are going to have prototypes like:

```
type function_name(array argument, integer argument)
```

For example, we could take the `mean()` function from 11.3.1 and make it more useful. Instead of

```
double mean(void); // finds mean of data in s_heights
```

we can have

```
double mean(double data_array[], int numelements);
```

This version could be used any time we want to find the mean of the values in an array of doubles. Note, the array argument is specified as an "open array" i.e. the `[]` contains no number.

```
double mean(double data_array[], int numelements)
{
    double sum = 0.0;
    for(int i=0; i<numelements; i++)
           sum += data_array[i];
    return sum / numelements;
}
```

This improved version of mean() is used in the following code fragment:

```
int main()
{
    double heights[100];
    double h;
    int n = 0;
    cout << "Enter heights:" << endl;
    cin >> h; // Not checking anything here
    while(h > 0.0) {
           heights[n] = h;
           n++;
           cin >> h;
           }
    double m = mean(heights, n);

    cout << "The mean is " << m << endl;

    return 0;
}
```

Function call with array as argument

Figure 11.1 illustrates the stack during a call to `mean()`. The hexadecimal numbers, e.g. , shown are actual addresses when the program was run on a particular machine. By convention, data addresses are always shown in hexadecimal ("hex").

The instruction sequence generated for the call `m = mean(heights, n)` would be something like:

```
load register with the address of array heights (i.e. address
of heights[0])
store in stack
load register with value of n
store in stack
subroutine call
load register with value in return slot of stack
store in m
clean up stack
```

The instructions in function `mean()` would be something like:

```
    load an address register a0 with contents of
                 first argument slot of stack frame
    load integer register d0 with contents of
                 second argument slot of stack frame
    zero contents of double register f0
    zero second integer register d1
loop
    compare integer registers d0 and d1
    if contents equal jump to end
```

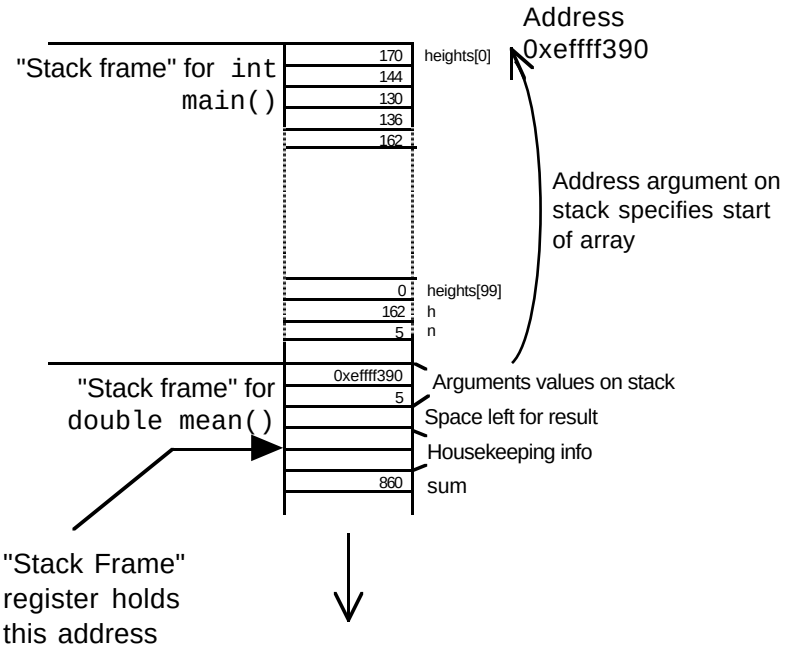


Figure 11.1 Stack with when calling double mean(double data_array[], int numitems).

```
    copy content d1 to register d2
    multiply d2 by 8
    copy content of a0 to address register a1
    add content of d2 to a1
    add double from memory at address a1 to register f0
    add 1 to d1
    jump to loop
end
    convert integer in d0 to double in f1
    divide f0 by f1
    store f0 in return slot of stack frame
```

This code uses the array identified by its address as provided by the calling routine. Typically, it would hold this "base" address in one of the CPU's address registers. Then when array elements have to be accessed, the code works out the offset relative to this base address to get the actual address of a needed element. The data are then fetched from the memory location whose address has just been determined. (The bit about "multiplying an integer index value by 8" is just part of the mechanics of converting the index into a byte offset as needed by the hardware addressing mechanisms of a machine. Programmers may think in terms of indexed arrays, machines have to use

byte addresses so these have to be calculated. The value of the 'multiplier', the 8 in this code, depends on the type of data stored in the array; for an array of longs, the multiplier would probably be 4. The operation wouldn't actually involve a multiplication instruction, instead the required effect would be achieved using alternative quicker circuits.)

const and non const arguments

The function `mean()` doesn't change the data array it is given; it simply reads values from that array. However, functions can change the arrays that they work on. So you could have the function:

```
void change_cm_to_inches(double h[], int n)
{
    // I'm too old fashioned to be happy with cm,
    // change those height values to inches
    for(int i = 0; i < n ; i++)
           h[i] /= 2.54;
}
```

If called with the `heights[]` array, this function would change all the values (or, at least, the first `n` values) in that array. (Coding style: no `return` – that is OK in a `void` function, though I prefer a function to end with `return`; the smart `h[i] /= 2.54;` – it is correct, I still find it easier to read `h[i] = h[i] / 2.54`.)

Function `mean()` treats its array argument as "read only" data; function `change_cm_to_inches()` scribbles all over the array it was given. That is a substantial difference in behaviour. If you are going to be calling a function and giving it an array to work with, you would usually like to know what is going to happen to the contents of that array.

C++ lets you make this distinction explicit in the function prototype. Since `mean()` doesn't change its array, it regards the contents as – well constant. So, it should state this in its prototype:

```
double mean(const double data_array[], int numelements);
```

As well as appearing in function declarations, this `const` also appears where the function is defined:

```
double mean(const double data_array[], int numelements)
{
    double sum = 0.0;
    …
}
```

The compiler will check the code of the function to make certain that the contents of the array aren't changed. Of course, if you have just declared an argument as `const`, it is unlikely that you will then write code to change the values.

The main advantage of such `const` (and default non `const`) declarations is for documenting functions during the design stage. In larger projects, with many programmers, each programmer works on individual parts and must make use of functions provided by their colleagues. Usually, all they see of a function that you have written is the prototype that was specified during design. So this prototype must tell them whether an array they pass to your function is going to be changed.

11.5 STRINGS: ARRAYS OF CHARACTERS

Sometimes you want arrays of characters, like the grade symbols used in one of the examples on array initialization:

```
const char grade_letters[6] = { 'F', 'E', 'D', 'C', 'B', 'A' };
```

The example in 11.8.2 uses a character array in a simple letter substitution approach to the encryption of messages. If you were writing a simple word processor, you might well use a large array to store the characters in the text that you were handling.

Most arrays of characters are slightly specialized. Most arrays of character are used to hold "strings" – these may be data strings (like names of customers), or prompt messages from the program, or error messages and so forth. All these strings are just character arrays where the last element (or last used element) contains a "null" character.

"null character"

The null character has the integer value 0. It is usually represented in programs as '\0'. All the standard output routines, and the string library discussed below, expect to work with null terminated arrays of characters. For example, the routine inside the iostream library that prints text messages uses code equivalent to the following:

```
void PrintString(const char msg[])
{
    int i = 0;
    char ch;
    while((ch = msg[i]) != '\0') { PrintCharacter(ch); i++; }
}
```

(The real code in the iostream library uses pointers rather than arrays.) Obviously, if the message `msg` passed to this function doesn't end in a null character, this loop just keeps printing out the contents of successive bytes from memory. Sooner or later, you are going to make a mistake and you will try to print a "string" that is not properly terminated; the output just fills up with random characters copied from memory.

You can define initialized strings:

```
const char eMsg1[] = "Disk full, file not saved.";
const char eMsg2[] = "Write error on disk.";
```

The compiler checks the string and determines the length, e.g. "Write … disk." is 20 characters. The compiler adds one for the extra null character and, in effect, turns the definition into the following:

```
const char eMsg2[21] = {
    'W', 'r', 'i', 't', 'e',
    ' ', 'e', 'r', 'r', 'o',
    'r', ' ', 'o', 'n', ' ',
    'd', 'i', 's', 'k', '.',
    '\0'
};
```

If you want string data, rather than string constants, you will have to define a character array that is going to be large enough to hold the longest string that you use (you have to remember to allow for that null character). For example, suppose you needed to process the names of customers and expected the maximum name to be thirty characters long, then you would define an array:

```
char customername[31];
```

(Most compilers don't like data structures that use an odd number of bytes; so usually, a size like 31 gets rounded up to 32.)

You can initialize an array with a string that is shorter than the specified array size:

```
char filename[40] = "Untitled";
```

The compiler arranges for the first nine elements to be filled with the character constants 'U', 'n', …, 'e', 'd', and '\0'; the contents of the remaining 31 elements are not defined.

Input and output of strings

The contents of the character array containing a string, e.g. `filename`, can be printed with a single output statement:

```
cout << filename;
```

(The compiler translates "*cout takes from character array*" into a call to a `PrintString()` function like the one illustrated above. Although `filename` has 40 characters, only those preceding the null are printed.)

cin* >> *character array

The input routines from the iostream library also support strings. The form "*cin gives to character array*", e.g.:

```
cin >> filename;
```

is translated into a call to a `ReadString()` function which will behave something like the following code:

```
void ReadString(char target[])
{
    char ch;
    // Skip white space
    ch = ReadCharacter();
    while(ch == ' ')
           ch = ReadCharacter();

    // read sequence of characters until next space
    int i = 0;
    while(ch != ' ') {
           target[i] = ch;
           i++;
           ch = ReadCharacter();
           }
    // Terminate with a null
    target[i] = '\0';
}
```

The input routine skips any leading "whitespace" characters (not just the ' ' space character, whitespace actually includes things like tab characters and newlines). After all leading whitespace has been consumed, the next sequence of characters is copied into the destination target array. Copying continues until a whitespace character is read.

Caution, area where bugs lurk

Of course there are no checks on the number of characters read! There can't be. The compiler can't determine the size of the target array. So, it is up to the programmer to get the code correct. An array used in an input statement must be large enough to hold the word entered. If the target array is too small, the extra characters read just overwrite other variables. While not a particularly common error, this does happen; so be cautious. (You can protect yourself by using the `setw()` "manipulator". This was illustrated in section 9.7 where it was shown that you could set the width of an output field. You can also use `setw()` to control input; for example, `cin >> setw(5) >> data` will read 4 characters into array data, and as the fifth character place a '\0'.)

The iostream library code is written with the assumption that any character array filled in this way is going to be used as a string. So, the input routine always places a null character after the characters that were read.

getline()

With "cin >> *array*", input stops at the first white space character. Sometimes, it is necessary to read a complete line containing several words separated by spaces. There is an alternative input routine that can be used to read whole lines. This routine,

`getline()`, is part of the repertoire of things that an `istream` object can do. Its prototype is (approximately):

```
getline(char target[], int maximum_characters,
                  char delimiter = '\n');
```

Function `getline()` is given the name of the character array that should hold the input data, a limit on the number of characters to read, and a delimiter character. A call to `getline()` has a form like:

```
cin.getline(target, 39, '\n');
```

(read this as "cin object: use your getline function to fill the array target with up to 39 characters of input data").

The delimiter character can be changed. Normally, `getline()` is used to read complete lines and the delimiter should be the '\n' character at the end of the line. But the function can be used to read everything up to the next tab character (use '\t' as the delimiter), or any other desired character. (Caution. The editors used with some development environments use '\r' instead of '\n'; if you try reading a line from a text data file you may not find the expected '\n'.)

Characters read from input are used to fill the target array. Input stops when the delimiter character is read (the delimiter character is thrown away, it doesn't get stored in the array). A null character is placed in the array after the characters that were read.

The integer limit specifies the maximum number of characters that should be read. This should be one less than the size of the target array so as to leave room for the null character. Input will stop after this number of characters have been read even though the specified delimiter has not been encountered. If input stops because this limit is reached, all unread characters on the input line remain in the input buffer. They will probably confuse the *next* input operation! (You can clear left over characters by using `cin.ignore()` whose use was illustrated earlier in section 8.2.)

The string library and the ctype functions

string.h

There is a standard library of routines for manipulating strings. The header file string.h contains declarations for all the string functions. Some of the more commonly used functions are:

```
strcpy(char destination[], const char source[]);
strncpy(char destination[], const char source[], int numchars);

strcat(char target[], const char source[]);
strncat(char target[], const char source[], int numchars);

int strcmp(const char firststring[], const char secondstring);
```

```
int strncmp(const char firststring[], const char secondstring,
                  int num_chars_to_check);

int strlen(const char string[]);
```

(These aren't the exact function prototypes given in the string.h header file! They have been simplified to use only the language features covered so far. The return types for some functions aren't specified and character arrays are used as arguments where the actual prototypes use pointers. Despite these simplifications, these declarations are "correct".)

You can't assign arrays:

The following code does not compile:

```
char baskin_robertson[50]; // Famous for ice creams
cout << "Which flavour? (Enter 1..37) : ";
int choice;
cin >> choice;
switch(choice) {
case 1: baskin_robertson = "strawberry"; break;
case 2: baskin_robertson = "chocolate"; break;
…
case 37:  baskin_robertson = "rum and raisin"; break;
default: baskin_robertson = "vanilla"; break;
}
```

If you try to compile code like that you will get some error message (maybe "lvalue required"). In C/C++, you can not do array assignments of any kind, and those "cases" in the switch statement all involve assignments of character arrays.

You can't do array assignments for the same reason that you can't pass arrays by value. The decision made for the early version of C means that arrays are just blocks of memory and the compiler can not be certain how large these blocks are. Since the compiler does not know the size of the memory blocks, it can not work out how many bytes to copy.

strcpy()

However, you do often need to copy a string from some source (like the string constants above) to some destination (the character array `baskin_robertson[]`). The function `strcpy()` from the string library can be used to copy strings. The code given above should be:

```
switch(choice) {
case 1: strcpy(baskin_robertson, "strawberry"); break;
…
default: strcpy(baskin_robertson, "vanilla"); break;
}
```

Function `strcpy()` takes two arguments, the first is the destination array that gets changed, the second is the source array. Note that the function prototype uses the

const qualifier to distinguish between these roles; destination[] is modified so it is not const, source[] is treated as read only so it is const.

Function strcpy() copies all the characters from the source to the destination, and adds a '\0' null character so that the string left in the destination is properly terminated. (Of course, there are no checks! What did you expect? It is your responsibility to make certain that the destination array is sufficient to hold the string that is copied.)

Functions strncpy() copies a specified number of characters from the source to the destination. It fills in elements destination[0] to destination[numchars-1]. If the source string is greater than or equal to the specified length, strncpy() does not place a '\0' null character after the characters it copied. (If the source string is shorter than the specified length, it is copied along with its terminating null character.) You can use strncpy() as a "safe" version of strcpy(). If your destination array is of size N, you call strncpy() specifying that it should copy N-1 characters and then you place a null character in the last element:

```
const int cNAMESIZE = 30;
…
char flavour[cNAMESIZE];
…
strncpy(flavour, othername, cNAMESIZE-1);
flavour[cNAMESIZE-1] = '\0';
```

strcat()

The strcat() function appends ("catenates") the source string onto any existing string in the destination character array. Once again it is your responsibility to make sure that the destination array is large enough to hold the resulting string. An example use of strcat() is:

```
char order[40] = "pizza with ";
…
strcat(order,"ham and pineapple");
```

changes the content of character array order from:

```
pizza with \0.........................
```

to:

```
pizza with ham and pineapple\0..........
```

Function strncat() works similarly but like strncpy() it transfers at most the specified number of characters.

strlen()

Function strlen() counts the number of characters in the string given as an argument (the terminating '\0' null character is not included in this count).

strcmp() and strncmp()

These functions compare complete strings (strcmp()) or a specified number of characters within strings (strncmp()). These functions return 1 if the first string (or

first few characters) are "greater" than the second string, 0 if the strings are equal, or -1 if the second string is greater than the first.

Collating sequence of characters

The comparison uses the "collating" sequence of the characters, normally this is just their integer values according to the ASCII coding rules:

```
 32: ' '   33: '!'   34: '"'   35: '#'   36: '$'   37: '%'
 38: '&'   39: '''   40: '('   41: ')'   42: '*'   43: '+'
 44: ','   45: '-'   46: '.'   47: '/'   48: '0'   49: '1'
 50: '2'   51: '3'   52: '4'   53: '5'   54: '6'   55: '7'
 56: '8'   57: '9'   58: ':'   59: ';'   60: '<'   61: '='
 62: '>'   63: '?'   64: '@'   65: 'A'   66: 'B'   67: 'C'
…
 86: 'V'   87: 'W'   88: 'X'   89: 'Y'   90: 'Z'   91: '['
 92: '\'   93: ']'   94: '^'   95: '_'   96: '`'   97: 'a'
…
…
```

For example, the code:

```
char msg1[] = "Hello World";
char msg2[] = "Hi Mom";

cout << strcmp(msg1, msg2) << endl;
```

should print -1, i.e. `msg1` is less than `msg2`. The H's at position 0 are equal, but the strings differ at position 1 where `msg2` has i (value 105) while `msg1` has e (value 101).

ctype

Another group of functions, that are useful when working with characters and strings, have their prototypes defined in the header file ctype.h These functions include:

```
int isalnum(int);        // letter or digit?
int isalpha(int);        // letter?
int iscntrl(int);        // control character
int isdigit(int);        // digit?
int islower(int);        // One of 'a'…'z'?
int ispunct(int);        // Punctuation mark?
int isspace(int);        // Space
int isupper(int);        // One of 'A'…'Z"?
int tolower(int);        // Make 'A'…'Z' into 'a'…'z'
int toupper(int);        // Make 'a'…'z' into 'A'…'Z'
int isprint(int);        // One of printable ASCII set?
```

11.6 MULTI-DIMENSIONAL ARRAYS

You can have higher dimensional arrays. Two dimensional arrays are common. Arrays with more than two dimensions are rare.

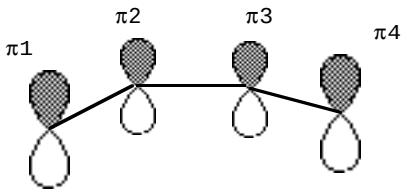
Two dimensional arrays are used:

for all kinds of tables, e.g. a spreadsheet:

Week \ Salesperson	1	2	3	4	5	
1	12	6	10	9	23	
2	9	11	12	4	18	
3	13	9	8	7	19	
4	8	4	6	6	13	
5	3	2	7	7	15	
6	9	11	13	10	19	
7	0	0	0	0	0	

to represent physical systems such as electronic interactions between atoms in a molecule:

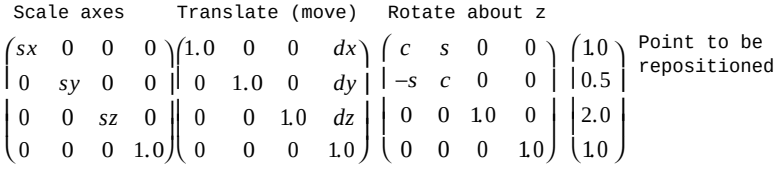
```
π-Orbital overlap integrals

  1.00  0.52  0.08  0.01
  0.52  1.00  0.44  0.08
  0.08  0.44  1.00  0.52
  0.01  0.08  0.52  1.00
```



or something like light intensities in a grey scale image (usually a very large 1000x1000 array):

```
200   200   180   168   100   …   …   160
200   190   180   170    99   …   …   155
200   194   170   …     …
…
…
 10    10   …
 10     8    4    …     …                 10
```

and in numerous mathematical, engineering, and scientific computations, e.g calculating coordinates of the positions of objects being moved:



Although two dimensional arrays are not particularly common in computing science applications, they are the data structures that are most frequently needed in scientific and engineering applications. In computing science applications, the most common two dimensional arrays are probably arrays of fixed length strings (since strings are arrays of characters, an array of strings can be regarded as a two dimensional array of characters).

11.6.1 Definition and initialization

In C (C++) two dimensional arrays are defined by using the `[]` operator to specify each dimension, e.g:

```
char        screen[25][80];
double      datatable[7][11];
double      t_matrix[4][4];
```

The first `[]` defines the number of rows; the second defines the number of columns. So in the example, `screen[][]` is a two dimensional array of characters that could be used to record the information displayed in each character position of a terminal screen that had 25 rows each with 80 columns. Similarly, `datatable[][]` is an array with 7 rows each of 11 columns and `t_matrix[][]` is a square 4x4 array.

As in the case of one dimensional arrays, the array indices start at zero. So, the top-left corner of the `screen[][]` array is `screen[0][0]` and its bottom right corner is `screen[24][79]`.

Two dimensional arrays can be initialized. You give all the data for row 0 (one value for each column), followed by all the data for row 1, and so on:

```
double mydata[4][5] = {
    { 1.0, 0.54, 0.21, 0.11, 0.03 },
    { 0.34, 1.04, 0.52, 0.16, 0.09 },
    { 0.41, 0.02, 0.30, 0.49, 0.19 },
    { 0.01, 0.68, 0.72, 0.66, 0.17}
};
```

You don't have to put in the internal { begin and } end brackets around the data for each individual row. The compiler will understand exactly what you mean if you write that initialization as:

```
double mydata[4][5] = {
    1.0, 0.54, 0.21, 0.11, 0.03, 0.34, 1.04, 0.52, 0.16, 0.09,
    0.41, 0.02, 0.30, 0.49, 0.19, 0.01, 0.68, 0.72, 0.66, 0.17
};
```

However, leaving out the brackets around the row data is likely to worry human readers – so put the { } row bracketing in!

With one dimensional arrays, you could define an initialized array and leave it to the compiler to work out the size (e.g. double `vec[] = { 2.0, 1.5, 0.5, 1.0 };`). You can't quite do that with two dimensional arrays. How is the compiler to guess what you want if it sees something like:

```
double guess[][] = {
    1.0, 0.54, 0.21, 0.11, 0.03, 0.34, 1.04, 0.52, 0.16, 0.09,
    0.41, 0.02, 0.30, 0.49, 0.19, 0.01, 0.68, 0.72, 0.66, 0.17
};
```

You could want a 4x5 array, or a 5x4 array, or a 2x10 array. The compiler can not guess, so this is illegal.

You must at least specify the number of columns you want in each row, but you can leave it to the compiler to work out how many rows are needed. So, the following definition of an initialized array is OK:

```
double mydata[][5] = {
    { 1.0, 0.54, 0.21, 0.11, 0.03 },
    { 0.34, 1.04, 0.52, 0.16, 0.09 },
    { 0.41, 0.02, 0.30, 0.49, 0.19 },
    { 0.01, 0.68, 0.72, 0.66, 0.17}
};
```

The compiler can determine that `mydata[][5]` must have four rows.

At one time, you could only initialize aggregates like arrays if they were global or filescope. But that restriction has been removed and you can initialize automatic arrays that are defined within functions (or even within blocks within functions).

11.6.2 Accessing individual elements

Individual data elements in a two dimensional array are accessed by using two [] operators; the first [] operator picks the row, the second [] operator picks the column. Mostly, two dimensional arrays are processed using nested for loops:

Nested for loops for processing two dimensional arrays

```
const int NROWS = 25;
const int NCOLS = 80;

char screen[NROWS][NCOLS];

// initialize the "screen" array to all spaces
for(int row = 0; row < NROWS; row++)
    for(int col = 0; col < NCOLS; col++)
           screen[row][col] = ' ';
```

Quite often you want to do something like initialize a square array so that all off-diagonal elements are zero, while the elements on the diagonal are set to some specific value (e.g. 1.0):

```
double t_matrix[4][4];
…
for(int i = 0; i < 4; i++) {
    for(int j = 0; j < 4; j++)
           t_matrix[i][j] = 0.0;
    t_matrix[i][i] = 1.0;  // set element on diagonal
    }
```

The "cost" of calculations

These "cost" of these double loops is obviously proportional to the product of the number of rows and the number of columns. As you get to work with more complex algorithms, you need to start estimating how much each part of your program will "cost".

The idea behind estimating costs is to get some estimate of how long a program will run. Estimates can not be very precise. After all, the time a program takes run depends on the computer (120MHz Pentium based machine runs a fair sight faster than 15MHz 386). The time that a program takes also depends on the compiler that was used to compile the code. Some compilers are designed to perform the compilation task quickly; others take much longer to prepare a program but, by doing a more detailed compile time analysis, can produce better code that results in a compiled program that may run significantly faster. In some cases, the run time of a program depends on the actual data values entered.

Since none of these factors can be easily quantified, estimates of how long a program will take focus solely on the number of data elements that must be processed. For some algorithms, you need a reasonable degree of mathematical sophistication in order to work out the likely running time of a program implementing the algorithm. These complexities are not covered in this book. But, the estimated run times will be described for a few of the example programs.

These estimates are given in a notation known as "big-O". The algorithm for initializing an MxN matrix has a run time:

```
initialize matrix    =      O(MxN)
```

or for a square matrix

```
initialize matrix    =      O(NxN)       or     O(N²)
```

Such run time estimates are independent of the computer, the compiler, and the specific data that are to be processed. The estimates simply let you know that a 50x50 array is going to take 25 times as long to process as a 10x10 array.

One of the most frequent operations in scientific and engineering programs has a cost $O(N^3)$. This operations is "matrix multiplication". Matrix multiplication is a basic step in more complex mathematical transforms like "matrix inversion" and "finding eigenvalues". These mathematical transforms are needed to solve many different problems. For example, you have a mechanical assembly of weights and springs you can represent this using a matrix whose diagonal elements have values related to the weights of the components and whose off diagonal elements have values that are determined by the strengths of the springs joining pairs of weights. If you want to know how such an assembly might vibrate, you work out the "eigenvalues" of the matrix; these eigenvalues relate to the various frequencies with which parts of such an assembly would vibrate.

Restricting consideration to square matrices, the "formula" that defines the product of two matrices is:

$$\begin{pmatrix} \sum_{i=0}^{M} a_{0i} * b_{i0} & \sum_{i=0}^{M} a_{0i} * b_{i1} & \dots & \dots & \dots & \sum_{i=0}^{M} a_{0i} * b_{iM} \\ \sum_{i=0}^{M} a_{1i} * b_{i0} & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \sum_{i=0}^{M} a_{Mi} * b_{i0} & \dots & \dots & \dots & \dots & \sum_{i=0}^{M} a_{Mi} * b_{iM} \end{pmatrix} =$$

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & \dots & \dots & a_{0M} \\ a_{10} & a_{11} & \dots & \dots & \dots & a_{1M} \\ a_{20} & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{M0} & a_{M1} & \dots & \dots & \dots & a_{MM} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} & b_{02} & \dots & \dots & b_{0M} \\ b_{10} & b_{11} & \dots & \dots & \dots & b_{1M} \\ b_{20} & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ b_{M0} & b_{M1} & \dots & \dots & \dots & b_{MM} \end{pmatrix}$$

(These are "NxN" arrays with subscripts running from 0 … M, where M = N - 1.). Each element in the product matrix is the sum of N product terms; these product terms combine an element from a row in the first array and an element from a column in the second array. For example, if N = 6, then entry product[2][3] is given by:

```
product[2][3] = a[2][0]*b[0][3] + a[2][1]*b[1][3] +
          a[2][2]*b[2][3] + a[2][3]*b[3][3] +
          a[2][4]*b[4][3] + a[2][5]*b[5][3];
```

Calculating such a product involves a triple loop:

O(N^3) algorithm: matrix multiplication

```
for(int row = 0; row < N; row++)
    for(int col = 0; col < N; col ++) {
          double prdct = 0.0;
          for(int i = 0; i <N; i++)
                prdct += a[row][i] * b[i][col];
          product[row][col] = prdct;
          }
```

(The variable `prdct` is introduced as a minor optimization. The code would be slower if you had `product[row][col] += a[row][i] * b[i][col]` because in this form it would be necessary to reevaluate the subscripts each time it needed to update `product[row][col]`.)

The O(N^3) cost of this triple loop code means that if you had a program that involved mainly matrix multiplication that ran in one minute for a data set of size 10, then you would need more than two hours to process a data set of size 50.

Another trap for Pascal programmers

Just a warning, you must use two separate [] operators when accessing an element in a two dimensional array. Now in languages like Pascal, elements of two dimensional arrays are accessed using square brackets surrounding a list of array indices separate by commas:

```
data[i,j]          Pascal style array subscripting
```

Now, in C/C++ you can write code like the following:

```
double d[10][10];
…
…
…   d[ i, j] …
```

That form `d[ i, j]` will get past the compiler. But, like several earlier examples of legal but freaky constructs, its meaning is odd.

As explained in section 7.7, a comma separated list can be used to define a set of expressions that must be evaluated in sequence. So `i, j` means "evaluate i, e.g. getting the value 4, and throw this value away, then evaluate j, eg. getting the value 9, and use this value as the final value for the complete expression".

Consequently,

```
d[ 4, 9]
```

in C++ actually means

```
d[9]
```

What is `d[9]`?

The expression `d[9]` means the tenth row of the `d[][]` array (tenth because row 0 is the first row so row 9 is the tenth row).

In C++ it is legal to refer to a complete row of an array. There aren't many places where you can do anything with a complete row of an array, but there are a few. Consequently, d[9] is a perfectly legal data element and the compiler would be happy if it saw such an expression.

Now, if you do accidentally type d[i, j] instead of d[i][j] you are going to end up getting an error message from the compiler. But it won't be anything like "You've messed up the array subscripts again.". Instead, the error message will be something like "Illegal operand type." Your code would be something like: `double t = 2.0* d[i,j];`. The compiler would see `2.0*`… and expect to find a number (an int, or a long, or a double). It wouldn't be able to deal with "`d[j]` row of array" at that point and so would thing that the data type was wrong.

Where might you use an entire row of an array?

Using a row of a two dimensional array

There aren't many places where it is possible (and even fewer where it is sensible). But you could do something like the following. Suppose you have an array representing the spreadsheet style data shown at the start of this section. Each row represents the number of sales made by each of a set of salespersonnel in a particular week. You might want the average sales, and have a function that finds the average of the elements of an array:

```
double average(const double d[], int n);

double salesfigures[20][10];
…
// Get average sales in week j
cout << "Average for this week " <<
    average(salesfigures[j], 10);
…
```

The `average()` function can be given a complete row of the array as shown in this code fragment.

While this is technically possible, it really isn't the kind of code you want to write. It is "clever" code using the language to its full potential. The trouble with such code is that almost everyone reading it gets confused.

11.6.3 As arguments to functions

The example just shown at the end of the preceding section illustrated (but discouraged) the passing of rows of a two-dimensional array to a function that expected a one dimensional array. What about passing the entire array to a function that wants to use a two dimensional array?

This is feasible, subject to some minor restrictions. You can't have something like the following:

```
void PrintMatrix(double d[][], int nrows, int ncols);
```

The idea would be to have a `PrintMatrix()` function that could produce a neatly formatted printout of any two dimensional array that it was given – hence the "doubly open" array `d[][]`.

The trouble is that the compiler wouldn't be able to work out the layout of the array and so wouldn't know where one row ended and the next began. Consequently, it wouldn't be able to generate code to access individual array elements.

You *can* pass arrays to functions, but the function prototype has to specify the number of columns in the arrays that it can deal with:

```
void PrintMatrix(double d[][5], int nrows);
```

The compiler will check calls made to `PrintMatrix()` to make certain that if you do call this function then any array you pass as an argument has five columns in each row.

11.7 ARRAYS OF FIXED LENGTH STRINGS

We may have a program that needs to use lots of "Messages", which for this example can be character arrays with 30 characters. We may need many of the Message things to hold different strings, and we may even need arrays of "Messages" to hold collections of strings. Before illustrating examples using arrays of strings, it is worth introducing an additional feature of C++ that simplifies the definition of such variables

Typedefs

C and C++ programs can have "typedef" definitions. Typedefs don't introduce a new type; they simply allow the programmer to introduce a *synonym* for an existing type. Thus, in types.h header used in association with several of the libraries on Unix you will find typedefs like the following:

```
typedef long      daddr_t;
typedef long      id_t;
typedef long      clock_t;
typedef long      pid_t;
typedef long      time_t;
```

The type `daddr_t` is meant to be a "disk address"; a `pid_t` is a "process type". They are introduced so that programmers can specify functions where the prototype conveys a little bit more about the types of arguments expected, e.g.:

```
void WriteArray(double d[], long n_elems, daddr_t where);
```

The fact that `where` is specified as being a `daddr_t` makes it a little clearer that this function needs to be given a disk location.

Variables of these "types" are actually all long integers. The compiler doesn't mind you mixing them:

```
time_t a = 3;
pid_t b = 2;
id_t r = a*b;
cout << r << endl;
```

So these typedef "types" don't add anything to the compiler's ability to discriminate different usage of variables or its ability to check program correctness.

However, typedefs aren't limited to defining simple synonyms, like alternative words for "long". You can use typedefs to introduce synonyms for derived types, like arrays of characters:

```
typedef char Message[30];
```

This typedef makes "`Message`" a synonym for the type `char … [30]` (i.e. the type that specifies an array of 30 characters). After this typedef has been read, the compiler will accept the definition of variables of type `Message`:

```
Message m1;
```

and arrays of `Message` variables:;

```
Message error_msgs[10]:
```

and declarations of functions that process `Message` arguments:

```
void PrintMessage(Message to_print);
```

Individual `Messages`, and arrays of `Messages`, can be initialized:

```
Message improbable = "Elvis lives in Wagga Wagga";

Message errors[] = {
    "Disk full",
    "Disk write locked",
```

```
    "Write error",
    "Can't overwrite file",
    "Address error",
    "System error"
};
```

(Each `Message` in the array `errors[]` has 30 characters allocated; for most, the last ten or more characters are either uninitialized or also to '\0' null characters because the initializing strings are shorter than 30 characters. If you try to initialize a Message with a string longer than 30 characters the compiler will report something like "Too many initializers".)

As far as the compiler is concerned, these `Messages` are just character arrays and so you can use them anywhere you could have used a character array:

```
#include <iostream.h>
#include <string.h>

typedef char Message[30];

Message m1;
Message m2 = "Elvis lives in Wagga Wagga";

Message errors[] = {
    "Disk full",
    "Disk write locked",
    "Write error",
    "Can't overwrite file",
    "Address error",
    "System error"
};

void PrintMessage(const Message toprint)
{
    cout << toprint << endl;
}

int main()
{
    cout << strlen(m2) << endl;

    PrintMessage(errors[2]);
    strcpy(m1, m2);

    PrintMessage(m1);
    return 0;
}
```

cout* << *Message

strlen() and Message

strcpy() and Message

This little test program should compile correctly, run, and produce the output:

```
26
Write error
Elvis lives in Wagga Wagga
```

A `Message` isn't anything more than a character array. You can't return any kind of an array as the result of a function so you can't return a `Message` as the result of a function. Just to restate, typedefs simply introduce synonyms, they don't really introduce new data types with different properties.

As in some previous examples, the initialized array `errors[]` shown above relies on the compiler being able to count the number of messages and convert the open array declaration `[]` into the appropriate `[6]`.

You might need to know the number of messages; after all, you might have an option in your program that lists all possible messages. Of course you can count them:

```
const int NumMsgs = 6;

…

Message errors[NumMsgs] = {
    "Disk full",
    "…
    "System error"
};
```

but that loses the convenience of getting the compiler to count them. (Further, it increases the chances of errors during subsequent revisions. Someone adding a couple of extra error messages later may well just change `errors[NumMsgs]` to `errors[8]` and fail to make any other changes so introducing inconsistencies.)

sizeof()

You can get the compiler work out the number of messages and record this as a constant, or you can get the compiler to provide data so you can work this out at run time. You achieve this by using the "`sizeof()`" operator.

Although `sizeof()` looks like a call to a function from some library, technically `sizeof()` is a built in operator just like + or %. It can be applied either to a type name or to a variable. It returns the number of bytes of memory that that type or variable requires:

```
int n = sizeof(errors);  // n set to number of bytes needed for
                         // the variable (array) errors
int m = sizeof(Message); // m set to number of bytes needed to
                         // store a variable of type Message
```

Using such calls in your code would let you work out the number of `Messages` in the `errors` array:

```
cout << "Array 'errors' :" << sizeof(errors) << endl;
```

```
cout << "Data type Message :" << sizeof(Message) << endl;
int numMsgs = sizeof(errors)/sizeof(Message);
cout << "Number of messages is " << numMsgs << endl;
```

But you can get this done at compile time:

```
Message errors[] = {
    "Disk full",
    "…
    "System error"
};

const int NumMsgs = sizeof(errors) / sizeof(Message);
```

This might be convenient if you needed a second array that had to have as many entries as the errors array:

```
Message errors[] = {
    "Disk full",
    …
};

const int NumMsgs = sizeof(errors) / sizeof(Message);

int err_counts[NumMsgs];
```

11.8 EXAMPLES USING ARRAYS

11.8.1 Letter Counts

Problem

The program is to read a text file and produce a table showing the total number of occurrences for each of the letters a … z.

This isn't quite as futile an exercise as you think. There are some uses for such counts.

For example, before World War 1, the encryption methods used by diplomats (and spies) were relatively naive. They used schemes that are even simpler than the substitution cipher that is given in the next example. In the very simplest encryption schemes, each letter is mapped onto a cipher letter e.g. a -> p, b ->n, c->v, d->e, e->r, …. These naive substitution ciphers are still occasionally used. But they are easy to crack as soon as you have acquired a reasonable number of coded messages.

The approach to cracking substitution ciphers is based on the use of letter frequencies. Letter 'e' is the most common in English text; so if a fixed substitution like e->r is used, then letter 'r' should be the most common letter in the coded text. If you

tabulate the letter frequencies in the encoded messages, you can identify the high frequency letters and try matching them with the frequently used letters. Once you have guessed a few letters in the code, you look for patterns appearing in words and start to get ideas for other letters.

More seriously, counts of the frequency of occurrence of different letters (or, more generally, bit pattern symbols with 8 or 16 bits) are sometimes needed in modern schemes for compressing data. The original data are analyzed to get counts for each symbol. These count data are sorted to get the symbols listed in order of decreasing frequency of use. Then a compression code is made up which uses fewer bits for high frequency symbols then it uses for low frequency symbols.

Specification:

1. The program is to produce a table showing the number of occurrences of the different letters in text read from a file specified by the user.

2. Control characters, punctuation characters, and digits are to be discarded. Upper case characters are to be converted to lower case characters so that the final counts do not take case into consideration.

3. When all data have been read from the file, a report should be printed in the following format:

```
a: 2570,    b:  436,    c: 1133,    d: 1203,    e: 3889,
f:  765,    g:  640,    h: 1384,    i: 2301,    j:   30,
k:  110,    l: 1296,    m:  782,    n: 2207,    o: 2108,
p:  594,    q:   51,    r: 1959,    s: 2046,    t: 2984,
u:  989,    v:  335,    w:  350,    x:  125,    y:  507,
z:   71,
```

Design:

First iteration

The program will be something like:

```
get user's choice of file, open the file, if problems give up
zero the counters
loop until end of file
    get character
    if letter
            update count
print out counters in required format
```

Again, some "functions" are immediately obvious. Something like "zero the counters" is a function specification; so make it a function.

Second iteration The rough outline of the program now becomes:

```
open file
    prompt user for filename
    read name
    try opening file with that name
    if fail give up

initialize
    zero the counters

processfile
    while not end of file
           read character
           if is_alphabetic(character)
                  convert to lower
                  increment its counter

printcounters
    line count = 0
    for each letter a to z
           print letter and associated count in appropriate fields
           increment line count
           if line count is 4 print newline and zero line count

main
    open file
    initialize
    processfile
    printcounters
```

Third iteration At this stage it is necessary to decide how to organize the array of letter counts. The array is going to be:

```
long       lettercounts[26]
```

We seem to need an array that you can index using a letter, e.g. `lettercounts['a']`; but that isn't going to be correct. The index is supposed to be a number. The letter 'a' interpreted as a number would be 97. If we have an 'a', we don't want to try incrementing the non-existent `lettercounts[97]`, we want to increment `lettercounts[0]`. But really there is no problem here. Letters are just integers, so you can do arithmetic operations on them. If you have a character ch in the range 'a' to 'z' then subtracting the constant 'a' from it will give you a number in the range 0 to 25.

We still have to decide where to define the array. It could be global, filescope, or it could belong to `main()`. If it is global or filescope, all routines can access the array. If it belongs to `main()`, it will have to be passed as a (reference) argument to each of the other routines. In this case, make it belong to `main()`.

The only other point that needs further consideration is the way to handle the filename in the "open file" routine. Users of Macs and PCs are used to "File dialog" windows popping up to let them explore around a disk and find the file that they want to open. But you can't call those run-time support routines from the kinds of programs that you are currently writing. All you can do is ask the user to type in a filename. There are no problems provided that the file is in the same directory ("folder") as the program. In other cases the user is supposed to type in a qualified name that includes the directory path. Most Windows users will have seen such names (e.g. c:\bc4\examples\owlexamples\proj1\main.cpp) but most Mac users will never have seen anything like "HD:C++ Examples:Autumn term:Assignment 3:main.cp". Just let the user type in what they will; use this input in the file open call. If the name is invalid, the open operation will fail.

The array used to store the filename (a local to the openfile function) should allow for one hundred characters; this should suffice even if the user tried to enter a path name. The file name entered by the user should be read using `getline()` (this is actually only necessary on the Mac, Mac file names can include spaces and so might not be read properly with a "cin >> character_array" style of input).

The `ifstream` that is to look after the file had better be a filescope variable.

Fourth Iteration

This involves nothing more than finalising the function prototypes before we start coding:

```
void Openfile(void);

void Initialize(int counts[]);

void Processfile(int counts[]);

void PrintCounters(const int counts[]);

int main()
```

and checking the libraries needed so that we remember to include the correct header files:

```
checking letters and case conversion          ctype.h
i/o                                           iostream.h
files                                         fstream.h
formatting                                    iomanip.h
termination on error                          stdlib.h
```

Note the difference between the prototype for `PrintCounters()` and the others. `PrintCounters()` treats the counts array as read only; it should state this in its prototype, hence the `const`. None of the functions using the array takes a count argument; in this program the array size is fixed.

Implementation:

We might as well define a constant ALPHABETSIZE to fix the size of arrays. The "includes", filescopes, and const declaration parts of the program are then:

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <ctype.h>
#include <stdlib.h>

const int ALPHABETSIZE = 26;

static ifstream infile;
```

Reusable OpenFile() routine

The `OpenFile()` routine should be useful in other programs (the prompt string would need to be replaced). Standard "cut and paste" editing will suffice to move this code to other programs; in Chapter 13 we start to look at building up our own libraries of useful functions.

```
void Openfile(void)
{
    const int kNAMESIZE = 100;
    char   filename[kNAMESIZE];

    cout << "Enter name of file with text for letter"
                 " frequency analysis" << endl;

    cin.getline(filename, kNAMESIZE-1, '\n');

    infile.open(filename, ios::in | ios::nocreate);

    if(!infile.good()) {
           cout << "Can't open that file. Quitting." << endl;
           exit(1);
           }
}
```

The check whether the file opening was unsuccessful could have been coded `if(!infile)` but such shortcuts really make code harder to read the more long winded `if(!infile.good())`.

The `Initialize()` function is pretty simple:

```
void Initialize(int counts[])
{
    // counts is supposed to be an array of
    //       ALPHABETSIZE elements
    for(int i = 0; i< ALPHABETSIZE; i++)
           counts[i] = 0;
```

```
}
```

In `Processfile()` we need all the alphabetic characters and don't want whitespace. It would have been sufficient to use "infile >> ch" (which would skip whitespace in the file). However the code illustrates the use of the `get()` function; many related programs will want to read all characters and so are likely to use `get()`.

```
void Processfile(int counts[])
{
    while(!infile.eof()) {
           char ch;
           infile.get(ch);
           if(!isalpha(ch))
                  continue;

           ch = tolower(ch);
           int ndx = ch - 'a';
           counts[ndx]++;
           }
    infile.close();
}
```

The `isalpha()` function from ctype.h can be used to check for a letter. Note the use of "`continue`" in the loop. Probably your instructor will prefer the alternative coding:

```
           …
           infile.get(ch);
           if(isalpha(ch)) {
                  ch = tolower(ch);
                  int ndx = ch - 'a';
                  counts[ndx]++;
                  }
           }
```

It is just a matter of style.

The statement:

```
ch = tolower(ch);
```

calls `tolower()` for all the alphabetic characters. You might have expected something like:

```
if(isupper(ch))
    ch = tolower(ch);
```

but that is actually more expensive. Calling `tolower()` for a lower case letter does not harm.

Function `PrintCounters()` has a loop processing all 26 characters. The `for(;;)` statement illustrates some common tricks. The main index runs from 'a' to 'z'. The subsidiary counter `j` (used for formatting lines) is defined along with the control index `i` in the start up part of the loop. Variable `i` is an integer so `cout << i` would print a number; its forced to print it as a character by the coding `cout << char(i)`.

```
void PrintCounters(const int counts[])
{
    for(int i='a', j=0; i<='z'; i++) {
           cout << char(i) << ":" << setw(5) <<
                  counts[i-'a'] << ",    ";
           j++;
           if(j == 5) { j = 0; cout << endl; }
           }
    cout << endl;
}
```

The `main()` function is nice and simple as all good mains should be:

```
int main()
{
    int lettercounts[ALPHABETSIZE];
    Openfile();
    Initialize(lettercounts);
    Processfile(lettercounts);
    PrintCounters(lettercounts);
    return 0;
}
```

Someone can read that code and immediately get an overall ("abstract") idea of how this program works.

11.8.2 Simple encryption

Problem

Two programs are required; the first is to encrypt a clear text message, the second is to decrypt an encrypted message to get back the clear text.

The encryption mechanism is a slightly enhanced version of the substitution cipher noted in the last example. The encryption system is for messages where the clear text is encoded using the ASCII character set. A message will normally have printable text characters and a few control characters. The encryption scheme leaves the control characters untouched but substitutes for the printable characters.

Characters whose integer ASCII codes are below 32 are control characters, as is the character with the code 127. We will ignore characters with codes 128 or above (these

will be things like π, ¤, fi, ‡, etc.). Thus, we need only process characters whose codes are in the range 32 (space) to 126 (~) inclusive. The character value will be used to determine an index into a substitution array.

The program will have an array containing the same characters in randomized order, e.g.:

```
static int substitutions[] = {
'U', 'L', 'd', '{', 'p', 'Q', '@', 'M',
'-', ']', 'O', 'r', 'c', '6', '^', '#',
'Y', 'w', 'W', 'v', '!', 'F', 'b', ')',
'+', 'h', 'J', 'f', 'C', '8', 'B', '7',
'.', 'k', '2', 'u', 'Z', '9', 'K', 'o',
'3', 'x', ' ', '\'', '=', '&', 'N', '*',
'1', 'z', '(', '`', 'R', 'P', ':', 'l',
'4', '0', 'e', '$', '_', '}', 'j', 't',
'?', 'S', 'q', '>', ';', 'T', 'y', 'i',
'\\', 'A', 'D', 'V', '5', '|', '<', '/',
'E', 'g', 'm', ',', '[', 'H', '%', 'a',
's', 'n', 'I', 'X', '~', '"', 'G'
};
```

(Note the way of handling problem characters like the single quote character and the backslash character.; these become \' and \\.)

Letter substitution for enciphering

You can use this to get a simple substitution cipher. Each letter of the message is converted to an index (by subtracting 32) and the letter in the array at that location is substituted for the original.

```
int ch = message[i];
int ndx = ch - 32;
int subs = substitutions[ndx];
cout << char(subs)
```

Using this simple substitution cipher a message:

```
Coordinate your attack with that of Colonel K.F.C. Sanders.  No
support artillery available.  Instead, you will be reinforced
by Rocket Battery 57.
```

comes out as

```
u//m;A<S[TUn/HmUS[[S>VUaA[\U[\S[U/yUu/5/<T5U'^K^u^U`S<;Tm,^UUN/
U,HEE/m[USm[A55TmnUS%SA5Sq5T^UUx<,[TS;cUn/HUaA55UqTUmTA<y/m>T;U
qnU(/>VT[U2S[[TmnUF)^
```

(Can you spot some of the give away patterns. The message ends with ^ and ^UU occurs a couple of times. You might guess ^ was '.' and U was space. That guess

would let you divide the text into words. You would then focus on two letter words N/, qn, etc.)

You might imagine that it was unnecessary to analyze letter frequencies and spend time trying to spot patterns. After all decrypting a message just involves looking up the encrypted letters in a "reverse substitution" table. The entry for an encrypted letter gives the original letter:

```
'J', '4', '}', '/', '[', 'v', 'M', 'K',
'R', '7', 'O', '8', 's', '(', '@', 'o',
'Y', 'P', 'B', 'H', 'X', 'l', '-', '?',
'=', 'E', 'V', 'd', 'n', 'L', 'c', '`',
'&', 'i', '>', '<', 'j', 'p', '5', '~',
'u', 'z', ':', 'F', '!', ''', 'N', '*',
'U', '%', 'T', 'a', 'e', ' ', 'k', '2',
'{', '0', 'D', 't', 'h', ')', '.', '\',
'S', 'w', '6', ',', '"', 'Z', ';', 'q',
'9', 'g', '^', 'A', 'W', 'r', 'y', 'G',
'$', 'b', '+', 'x', '_', 'C', '3', '1',
'I', 'f', 'Q', '#', 'm', ']', '|',
```

(The underlined entries correspond to indices for 'u' and '/', the first characters in the example encrypted message.)

Reversing the letter substitution process

If you have the original substitutions table, you can generate the reverse table:

```
void MakeRTable(void)
{
    for(int i = 0;i < len; i++) {
           int tem = substitutions[i];
           tem -= ' ';
           rtable[tem] = i+32;
    }
}
```

and then use it to decrypt a message:

```
void Decrypt(char m[])
{
    char ch;
    int i = 0;
    while(ch = m[i]) {
           int t = rtable[ch - 32];
           cout << char(t);
           i++;
    }
}
```

Of course, the enemy's decryption specialist doesn't know your original substitution table and so must guess the reverse table. They can fill in a guessed table, try it on the message text, and see if the "decrypted" words were real English words.

How could the enemy guess the correct table? Why not just try different possible tables? A program could easily be devised to generate the possible tables in some systematic way, and then try them on the message; the success of the decryption could also be automatically checked by looking up "words" in some computerized dictionary.

The difficulty there relates to the rather large number of possible tables that would have to be tried. Suppose all messages used only the letters a, b, c; the enemy decrypter would have to try the reverse substitution tables bac, bca, cba, cab, acb, (but presumably not abc itself). If there were four letters, there would be more possibilities: abcd, abdc, acbd, acdb, …, dbca; in fact, 24 possibilities. For five letters, it is 120 possibilities. The number of possibilities is the number of different permutations of the set of letters. There are N! (i.e. factorial(N)) permutations of N distinct symbols.

O(N!) !

Because there are N! permutations to be tried, a dumb "try everything" program would have a running time bounded by O(N!). Now, the value of N! grows rather quickly with increasing N:

N	N!	time scale
1	1	1 second
2	2	
3	6	
4	24	
5	120	two minutes ago
6	720	
7	5040	
8	40320	half a day ago
9	362880	
10	3628800	ten weeks ago
11	39916800	
12	479001600	
13	6227020800	more than a lifetime
14	87178291200	
15	1307674368000	back to the stone age!
16	20922789888000	
17	355687428096000	
18	6402373705728000	Seen any dinosaurs yet?

With the letter permutations for a substitution cipher using a full character set, N is equal to 95. Any intelligence information in a message will be "stale" long before the message is decrypted.

You can't just guess, but the cipher is still weak

Trying different permutations to find the correct one is impractical. But letter counting and pattern spotting techniques are very easy. As soon as you have a few hundred characters of encrypted messages, you can start finding the patterns. Simple character substitution cipher schemes are readily broken.

A safer encryption scheme

A somewhat better encryption can be obtained by changing the mapping after each letter. The index used to encode a letter is:

```
ndx = (ch - 32 + k) % 95;
```

the value `k` is incremented after each letter has been coded. (The modulo operator is used to keep the value of index in the range 0 to 94. The value for `k` can start at zero or at some user specified value; the amount it gets incremented can be fixed, or to make things a bit more secure, it can change in some systematic way.) Using this scheme the message becomes:

```
uEgH\<[\~<OQX{^EQ@,%UF^L6dJYQGv7Y-2y!Wbb^b %e,_mjz@FuJC
=ie$r&j`:(``leyx$jR40`SAE`APjq4eTt]X~o|,Hi;\w@LaGO{HnI#m%W@nGQ"
-c"LL8Lh.,WMYrfoDr7.W.3*T\?
```

This cipher text is a bit more secure because there aren't any obvious repeating patterns to give the enemy cryptanalyst a starting point. Apparently, such ciphers are still relatively easy to break, but they aren't trivial.

Specification:

1. Write a program "encrypt" and a program "decrypt" that encode and decode short messages. Messages are to be less than 1000 characters in length.

2. The encrypt program is to prompt the user for a filename, the encrypted message will be written to this file.

3. The encrypt program is then to loop prompting the user to enter the next line of the clear text message; the loop terminates when a blank line is entered.

4. If the line is not blank, the encrypt program checks the message line just entered. If the line contains control characters (apart from newline) or characters from the extended set (those with integer values > 127), a warning message should be printed and the line should be discarded. If the addition of the line would make the total message exceed 1000 characters, a warning message should be printed and the program should terminate.

5. If the line is acceptable, the characters are appended to those already in a main message buffer. The newline character that ended the input line should be replaced by a space.

6. When the input loop terminates, the message now held in a main message buffer should be encrypted according to the mechanism described above, the encrypted message is to be written to the file.

7. The encrypt program is to close the file and terminate.

8. The decrypt program is to prompt the user for a filename, and should then open the file (the decrypt program terminates if the file cannot be opened).

9. The message contained in the file should read and decoded character by character. The decoded message should be printed on cout. Since newlines have been removed during the encrypting process, the decrypt program must insert newlines where needed to keep the output text within page bounds.

10. The two programs should share a "substitutions" array.

Design:

The encrypt program will be something like: ***First iteration***

```
Open output file
Get the message
Encrypt the message
Save message to the file
Close the file
```

while the decrypt program will be:

```
Open input file
Convert substitutions table to reverse table
Decrypt the message
```

The requirement that the two programs share the same `substitutions` array is easily handled. We can have a file that contains just the definition of the initialized `substitutions` array, and #include this file in both programs.

The principal "functions" are immediately obvious for both programs, but some (e.g. "Getthe message") may involve additional auxiliary functions.

The rough outline of the encrypt program now becomes: ***Second iteration***

```
Open file
    prompt user for filename
    read name
    try opening an output file with that name
    if fail give up

Get the message
    initialize message buffer
    get and check lines until the blank line is entered
           adding lines to buffer
```

```
Encrypt
    replace each character in buffer by its encryption

Save
    write string to file

Close
    just close the file

encrypt_main
    Open output file
    Get message
    Encrypt
    Save
    Close
```

while the decrypt program becomes:

```
Open file
    prompt user for filename
    read name
    try opening an input file with that name
    if fail give up

MakeRTable
    basics of algorithm given in problem statement, minor
           changes needed to incorporate the stuff about
           the changing offset

Decrypt
    initialize
    read first character from file
    while not end of file
           decrypt character
           print character
           maybe formatting to do
           read next character from file

decrypt_main
    Open intput file
    MakeRTable
    Decrypt
```

Some of these function might be "one-liners". The `Close()` function probably will be just something like `outfile.close();`. But it is worth keeping them in as functions rather than absorbing their code back into the main routine. The chances are that sooner or later the program will get extended and the extended version will have additional work to do as it closes up.

The only function that looks as if it might be complex is "Get the message". It has to read input, check input, add valid input to the buffer, check overall lengths etc. That is too much work for one function.

The reading and checking of a single line of input can be factored out into an auxiliary routine. This main "Get the message" routine can call this "Get and check line" from inside a loop.

An input line may be bad (illegal characters), in which case the "GetMessage" controlling routine should discard it. Alternatively, an input line may be empty; an empty line should cause "GetMessage" to terminate its loop. Otherwise, GetMessage will need to know the actual length of the input line so that it can check whether it fits into the buffer. As well as filling in an array of characters with input, function "GetAndCheckLine" can return an integer status flag.

The "Get the message" outline above should be expanded to:

```
GetAndCheckLine
    use getline() to read a complete line
    if linelength is zero (final blank input line) return 0

    check that all characters are "printable", if any are
           not then return -1 to indicate a bad line

    return line length for a good line

GetMessage
    initialize message buffer and count
    n = GetAndCheckLine()
    while n != 0
           if n > 0
                  (message is valid)
                  check if buffer can hold new line (allow
                         for extra space and terminating null)
                  if buffer can't then print warning and quit

                  copy message to buffer, add extra space
                         and null at end
           n = GetAndCheckLine()
```

The bit about "maybe formatting to do" in function `Decrypt()` needs to be tightened up a bit. We need new lines put in before we reach column 80 (assumed width of the screen), but we want to avoid breaking words. The easiest way is to output a newline if we have just output a space, and our position is already beyond column 60.

Third iteration

At this stage it is necessary to decide how to organize data.

The array with the letter substitutions will get included from its own file. The array can be defined at filescope. This extra file should hold other information that the encrypt and decrypt programs would share. They would both need to know the starting value for any 'k' used to change the encoding of letters and the value of the increment that changes k. All these could be defined in the one file: "table.inc":

```
const int  kSTART = 0;
const int  kINCREMENT = 1;

static int substitutions[] = {
'U', 'L', 'd', '{', 'p', 'Q', '@', 'M',
'-', ']', 'O', 'r', 'c', '6', '^', '#',
…
…
'E', 'g', 'm', ',', '[', 'H', '%', 'a',
's', 'n', 'I', 'X', '~', '"', 'G'
};

const int len = sizeof(substitutions) / sizeof(int);
```

(The development environment you use may have specific requirements regarding file names. You want a name that indicates this file is not a header file, nor a separately compiled module; it simply contains some code that needs to be #included into .cp files. A suffix like ".inc" is appropriate if permitted in your environment.)

The `ifstream` that is to look after the file in the decrypt program can be a filescope variable as can the `ofstream` object used in the encrypt program. The "reverse table" used in the decrypt program can also be filescope.

Other character arrays can be local automatic variables of specific routines, getting passed by reference if required. The encrypt program will require a character array capable of holding a thousand character message; this can belong to its `main()` and get passed to each of the other functions. Function `GetMessage()` will have to define an array long enough to hold one line (≈120 characters should be plenty); this array will get passed to the function `GetAndCheckLine()`.

The Open-file functions in both programs will have local arrays to store the filenames entered by the users. The `Openfile()` function from the last example should be easy to adapt.

Fourth Iteration

This involves nothing more than finalising the function prototypes and checking the libraries needed so that we remember to include the correct header files:

Program encrypt:

```
void OpenOutfile(void);

int GetAndCheckLine(char line[], int linelen);

void GetMessage(char txt[]);

void Encrypt(char m[]);

void SaveToFile(const char m[]);
```

```
void CloseFile(void);

int main()

i/o                                          iostream.h
files                                        fstream.h
termination on error                         stdlib.h
copying strings                              string.h
```

Program decrypt:

```
void OpenInputfile(void)

void MakeRTable(void)

void Decrypt(void)

int main()

i/o                                          iostream.h
files                                        fstream.h
termination on error                         stdlib.h
```

Implementation:

The first file created might as well be "table.inc" with its character substitution table. This could simply be copied from the text above, but it would be more interesting to write a tiny auxiliary program to create it.

"Scaffolding"

It is actually quite common to write extra little auxiliary programs as part of the implementation process for a larger program. These extra programs serve as "scaffolding" for the main program as it is developed. Most often, you write programs that produce test data. Here we can generate a substitution table.

The code is simple. The table is initialized so that each character is substituted by itself! Not much security in that, the message would be copied unchanged. But after this initialization, the data are shuffled. There are neater ways to do this, but a simple way is to have a loop that executes a thousand times with each cycle exchanging one randomly chosen pair of data values. The shuffled table can then be printed.

```
#include <iostream.h>
#include <stdlib.h>

const int NSYMS = 95;
const int NSHUFFLES = 1000;
int main()
{
    cout << "Enter seed for random number generator" << endl;
    int s;
```

```
    cin >> s;

    srand(s);

    char table[NSYMS];

    for(int i = 0; i < NSYMS; i++)
           table[i] = i + ' ';

    // shuffle
    for(int j = 0; j < NSHUFFLES; j++ ) {
           int ndx1 = rand() % NSYMS;
           int ndx2 = rand() % NSYMS;

           char temp = table[ndx1];
           table[ndx1] = table[ndx2];
           table[ndx2] = temp;
           }

    // Output
    j = 0;
    for(i = 0; i < NSYMS; i++) {
           cout << "'" << table[i] << "', ";
           j++;
           if(j == 8) { cout << endl; j = 0; }
           }
    cout << endl;
    return 0;
}
```

The output from this program has to be edited to deal with special characters. You have to change `'''` to `'\''` and `'\'` to `'\\'`. Then you can substitute the generated table into the outline given above for "table.inc" with its `const` definitions and the definition of the array `substitutions`.

Encrypt

Program encrypt would start with #includes for the header files describing the standard input/output libraries etc, and a #include for the file "table.inc".

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#include "table.inc"
```

#includes of system files and project files

Note that the #include statements are different. The system files (the headers that describe the system's libraries like iostream) are #included with the filename in pointy brackets <…>. A project file, like table.inc, is #included with the filename in double quotes "…". The different brackets/quotes used tell the compiler where to look for the file. If the filename is in quotes, then it should be in the same directory as the program code. If the filename is in <> brackets, then it will be located in some standard directory belonging to the development environment.

Program encrypt would continue with a definition of the maximum message size, and of a variable representing the output file. Then the functions would be defined.

```
const int  kMSGSIZE = 1000;

static ofstream   outfile;

void OpenOutfile(void)
{
    const int kNAMESIZE = 100;
    char   filename[kNAMESIZE];

    cout << "Enter name of message file" << endl;

    cin.getline(filename, kNAMESIZE-1, '\n');

    outfile.open(filename, ios::out);

    if(!outfile.good()) {
           cout << "Can't open that file. Quitting." << endl;
           exit(1);
           }
}
```

Function `OpenOutFile()` is a minor variation on the `Openfile()` function in the previous example.

Function `GetAndCheckLine()` is passed the line array that should be filled in, along with details of its length. The line is filled using the `cin.getline()` call, and the length of the text read is checked. If the length is zero, a terminating blank line has been entered, so the function can return. Otherwise the for loop is executed; each character is checked using `isprint()` from the ctype library; returns 1 (true) if the character tested is a printable character in the ASCII set. If any of the characters are unacceptable, `GetAndCheckLine()` returns -1 as failure indicator. If all is well, a 1 success result is returned.

```
int GetAndCheckLine(char line[], int linelen)
{
    cin.getline(line, linelen - 1, '\n');
    // Check for a line with no characters
    int actuallength = strlen(line);
    if(actuallength == 0)
```

```
            return 0;
        // Check contents
        // all characters should be printable ASCII
        for(int i = 0; i< actuallength; i++)
            if(!isprint(line[i])) {
                cout << "Illegal characters,"
                 " line discarded"; return -1; }
        return actuallength;
    }
```

Function `GetMessage()` is passed the array where the main message text is to be stored. This array doesn't need to be blanked out; the text of the message will overwrite any existing contents. The message will be terminated by a null character, so if the message is shorter than the array any extra characters at the end of the array won't be looked at by the encryption function. Just in case no data are entered, the zeroth element is set to the null character.

Function `GetMessage()` defines a local array to hold one line and uses a standard while loop to get and process data terminated by a sentinel. In this case, the sentinel is the blank line and most of the work of handling input is delegated to the function `GetAndCheckLine()`.

```
void GetMessage(char txt[])
{
    int    count = 0;
    txt[count] = '\0';
    const int ll = 120;
    char Line[ll];
    int n = GetAndCheckLine(Line,ll);
    while(n != 0) {
            if(n > 0) {
                    count += n;
                    if(count > kMSGSIZE-2) {
                            cout << "\n\nSorry that message is"
                                    "too long. Quitting." << endl;
                            exit(1);
                            }
                    strcat(txt, Line);
                    // add the space to the end of the message
                    txt[count] =' ';
                    // update message length
                    count++;
                    // and replace the null just overwritten
                    txt[count] = '\0';
                    }
            n = GetAndCheckLine(Line,ll);
            }
}
```

Function `GetMessage()` is the one where bugs are most likely. The problem area will be the code checking whether another line can be added to the array. The array must have sufficient space for the characters of the newline, a space, and a null. Hopefully, the checks are done correctly!

Beware of "out by 1" errors

It is easy to get errors where you are one out so that if you do by chance get a 999 character input message you will fill in `txt[1000]`. Of course, there is no element `txt[1000]`, so filling in a character there is an error. You change something else.

Finding such bugs is difficult. They are triggered by an unusual set of data (most messages are either definitely shorter than or definitely longer than the allowed size) and the error action will often do nothing more than change a temporary automatic variable that was about to be discarded anyway. Such bugs can lurk undetected in code for years until someone makes a minor change so that the effect of the error is to change a variable that is still needed. Of course, the bug won't show just after that change. It still will only occur rarely. But sometime later, it may cause the program to crash.

How should you deal with such bugs?

Prove correctness

The first strategy is never to make "out by 1" errors. There is an entire research area on "proof of program correctness" that comes up with mechanisms to check such code and show that it is correct. This strategy is considered elegant; but it means you spend a week proving theorems about every three line function.

Code carefully and test thoroughly

The second strategy is to recognize that such errors are possible, code carefully, and test thoroughly. The testing would focus on the special cases (no message, and a message that is just smaller than, equal to, or just greater than the maximum allowed). (To make testing easier, you might compile a version of the program with the maximum message size set to something much smaller, e.g. 25).

Avoid

A final strategy (of poor repute) is to eliminate the possibility of an error causing harm in your program. Here the problem is that you might be one out on your array subscript and so trash some other data. OK. Make the array five elements larger but only use the amount originally specified. You can be confident that that extra "slop" at the end of the array will absorb any over-runs (you know you won't be six out in your subscripting). Quick. Easy. Solved for your program. But remember, you are sending the contents of the array to some other program that has been told to deal with messages with "up to 1000 characters". You might well surprise that program by sending 1001.

The `Encrypt()` function is simple. If `kSTART` and `kINCREMENT` are both zero it will work as a simple substitution cipher. If `kINCREMENT` is non zero, it uses the more sophisticated scheme:

```
void Encrypt(char m[])
{
    int k = kSTART;
    for(int i=0; m[i] != '\0'; i++) {
           char ch = m[i];
           m[i] = substitutions[(ch-32 +k ) % len];
           k += kINCREMENT;
    }
}
```

Functions `SaveToFile()` and `CloseFile()` are both tiny to the point of non existence. But they should still both be functions because they separate out distinct, meaningful phases of the overall processing.

```
void SaveToFile(const char m[])
{
    outfile << m;
}

void CloseFile(void)
{
    outfile.close();
}
```

A good `main()` should be simple, little more than a sequence of function calls. Function `main()` declares the array `msg` and passes it to each of the other functions that needs it.

```
int main()
{
    char msg[kMSGSIZE];
    OpenOutfile();
    cout << "Enter text of message:" << endl;
    GetMessage(msg);
    cout << "\nEncrypting and saving." << endl;
    Encrypt(msg);
    SaveToFile(msg);
    CloseFile();
    return 0;
}
```

Decrypt

Program decrypt is a bit shorter. It doesn't need quite as many system's header files (reading header files does slow down the compilation process so only include those that you need). It #includes the table.inc file to get the chosen substitution table and details of the modifying `kSTART` and `kINCREMENT` values. It defines filescope variables for its reverse table and its input file.

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

#include "table.inc"

static int rtable[len];
static ifstream infile;
```

The program has an `OpenInputfile()` function that is not shown here; this function is similar to that in the letter counting example. Function `MakeRTable()` makes up the reverse substitution table from the data in the #included file with the definition of array substitutions. The code here is slightly different from that illustrated in the problem description. Rather than have rtable hold the character, it holds the index number of the character. This is done to simplify the arithmetic in the `Decrypt()` function.

```
void MakeRTable(void)
{
    for(int i = 0;i < len; i++) {
           int tem = substitutions[i];
           tem -= ' ';
           rtable[tem] = i;
    }
}
```

Function `Decrypt()` reads characters of the encrypted message from the file and outputs the original message characters. Newlines are inserted at convenient points. The decryption mechanism reverses the encryption. The crypto character is looked up in the array, getting back the index of the character selected in the `Encrypt()` routine. This isn't the original character from the clear text message because it will have been offset by the k term. So this offset is reversed. That process gives the index number of the original character, adding `' '` converts this index into the ASCII code.

```
void Decrypt(void)
{
    int ch;
    int linepos = 0;
    int k = kSTART;
    ch = infile.get();
    while(ch != EOF) {
           int t = rtable[ch - 32];
           t -= k;
           while(t<0) t+= len;
           cout << char(t + ' ');

           if((t==0) && (linepos > 60))
                  { cout << endl; linepos = 0; }
           k += kINCREMENT;
           ch = infile.get();
    }
    cout << endl;
}
```

The `main()` routine is simply a sequence of function calls:

```
int main()
{
    OpenInputfile();
    MakeRTable();
    Decrypt();
    return 0;
}
```

11.8.3 Simple image processing

Problem

Most pictures input to computers are simply retouched, resized, recoloured, and then displayed or printed. But in some areas like "Artificial Intelligence" (AI), or "Pattern Recognition" (PR), or "Remote Sensing", programs have to be written to interpret the data present in the image.

One simple form of processing needed in some AI and PR programs involves finding the outlines of objects in an image. The image data itself will consist of an array; the elements in this array may be simple numbers in the range 0 (black) … 255 (white) for a "grey scale" image, or they could be more complex values representing the colour to be shown at each point in the image (grey scale data will be used to simplify the example). A part of an image might be:

```
220 220 218 216 222 220 210  10  12  10   8 …
220 222 214 218 220 214 216   8   8   8 …
230 228 220 220 218 216  40   8   7 …
228 226 226 220 221 200  16  10   4 …
220 224 218 220 220  60  10   8   5 …
218 220 226 220  54  16   9   6 …
215 218 220 200  22  11   8 …
210 220 214  48  10   8   7 …
208 216 202  12   8   4 …
206 210 104   8   8   6 …
208 212 164  16   8    8 …
210 216 200  60  20   9 …
212 220 218 200  32   8    6 …
…
…
```

This image has a jagged "edge" running approximately diagonally leftwards to a corner and then starting to run to the right. Its position is marked by the sudden change in intensities.

Finding and tracing edges is a little too elaborate to try at this stage. But a related simpler task can be performed. It is sometimes useful to know which row had the largest difference in intensities in successive columns (need to know row, column

number of first of column pair, and intensity difference) and, similarly, which column had the largest difference between successive rows.

These calculations involve just searches through the array considering either rows or columns. If searching the rows, then for each row you find the largest difference between successive column entries and compare this with the largest difference found in earlier rows. If you find a larger value, you update your records of where the maximum difference occurred.

The searches over the rows and over the columns can be handled by rather similar functions – a `ScanRows()` function and a `ScanCols()` function. The functions get passed the image data. They initialize a record of "maximum difference" to zero, and then use a nested double loop to search the array.

Multiple results from a function

There is one slightly odd aspect to these functions – they need to return three data values. Functions usually compute a value i.e. a single number. But here we need three numbers: row position, column position, and maximum intensity difference. So, there is a slight problem.

Later (Chapters 16 and 17) we introduce `struct` types. You can define a `struct` that groups together related data elements, like these three integer values (they don't really have independent meaning, these value belong together as a group). A function can return a `struct`, and that would be the best solution to the current problem.

However, there is an alternative approach that, although not the best, can be used here. This mechanism is required in many other slightly more elaborate problems that will come up later. This approach makes the functions `ScanRows()` and `ScanCols()` into procedures (i.e. `void` return type) and uses "reference" arguments to communicate the results.

Pass by reference again

As explained in section 11.4, arrays are always "passed by reference". The calling function places the address of the start of the array in the stack. The called function takes this address from the stack and places it in one of the CPU's address registers. When the called function needs to access an element of an array, it uses the contents of this address register while working out where required data are in memory. The data value is then fetched and used, or a new value is stored at the identified location.

Simple types as reference arguments

Simple types, like integers and doubles, are usually passed by value. However, you can request the "pass by reference" mechanism. If you specify pass by reference, the calling function places details of the address of the variable in the stack (instead of its value). The called function, knowing that it is being given an address, can proceed in much the same way as it does when passed an array. Its code arranges to load the address of the argument into an address register. When the argument's value is needed the function uses this address register to help find the data (just like accessing arrays except that there is no need to add on an "index" value).

Since the function "knows" where the original data variables are located in memory, it can change them. This makes it possible to have a function that "returns" more than one result. A function can have any number of "reference" arguments. As it runs, it can change the values in those variables that were passed by the calling function. When the function finishes, its local stack variables all disappear, but the changes it made to

the caller's variable remain in effect. So, by changing the values of several reference arguments, a function can "return" several values.

Specifying pass by reference

When you define the function prototype, you chose whether it uses pass by value or pass by reference for simple variables (and for `structs`). If don't specify anything, you get pass by value. If you want pass by reference you write something like:

```
void MultiResultFunction(int& ncount, double& dmax);
```

Reference types

Here, `&` is used as a qualifier that turns a simple type, e.g. `int`, into a "reference type", e.g. `int&`. Apart from a few very rare exceptions, the only place you will see variables of reference types being declared is in function prototypes.

In the C++ source code for `MultiResultFunction()`, the reference variable `ncount` and `dmax` will be treated just like any other `int` and `double` variables:

```
void MultiResultFunction(int& ncount, double& dmax)
{
    int m, d;
    cout << "Current count is " << ncount << endl;
    …
    cin >> …;
    …
    ncount += 2;
    …
    dmax = 0;
    cin >> d;
    …
    dmax = (d > dmax) ? d : dmax;
    return;
}
```

Because `ncount` and `dmax` are references, the compiler generates instructions for dealing with them that differ from those used to access ordinary variables like `m` and `d`. The compiler "knows" that the value in the reference variables are essentially addresses defining where to find the data. Consequently, the generated machine code is always of the form "load address register from reference variable, use address register's contents when accessing memory to get data value, use data value, (store back into memory at place specified by contents of address register)".

Input, output, and input-output parameters

Variables passed by values are sometimes referred to as "input parameters"; they provide input to a function but they allow only one way communication. Reference variables allow two way communication. The function can use the values already in these variable (as above where the value of `ncount` is printed) or ignore the initial values (as with `dmax` in the code fragment above). Of course they allow output from a function – that was why we introduced them. So here, `ncount` is an "input output" (or "value result") parameter while `dmax` is an output (or "result") parameter.

There are some languages where the "function prototypes" specify whether a reference argument is an output (result) or an input-output (value result) parameter.

C++ does not have any way of indicating this; but this kind of information should form part of a function's description in the program design documentation.

When passing arrays, we sometimes used `const` to indicate that the function treated the array as "read only" (i.e. the array was an "input parameter"), so why is the following inappropriate?

```
void Testfunction(const int& n)
{
    …
}
```

This is legal, but it is so much simpler to give `Testfunction()` its own copy of a simple variable like an `int`. The following would be much more appropriate:

```
void Testfunction(int n)
{
    …
}
```

When we get to `structs`, we will find that these can be passed by value or by reference. Passing a struct by value means that you must make a copy. As this copying process may be expensive, structs are often passed by reference even when they are "input parameters". In such circumstances, it is again appropriate to use the `const` qualifier to flag the role of the argument.

Now, to return to the original problem of the image processing program, we can achieve the required multi-result functions `ScanRows()` and `ScanCols()` by using pass by reference. These functions will be given the image data as const array argument along with three integers that will be passed by reference (these will be "output" or "result" arguments because their initial values don't matter).

Specification:

1. Write a program that will analyze "image data" to identify the row with the maximum difference in intensity between successive columns, and the column with the maximum difference in intensity between successive rows.

2. For the purposes of this program, an image is a rectangular array of 40 rows and 50 columns. The intensity data can be encoded as short integers (or as unsigned short integers as all data values should be positive).

3. The program is to prompt the user for a filename; the image data will be read from this file. The file will contain 2000 numbers; the first 50 values represent row 0, the next 50 represent row 1 and so forth. The image data can be assumed to be "correct"; there will be 2000 numbers in the file, all numbers will be positive and in

a limited range. The format will have integer values separated by spaces and newlines.

4. The program is identify the positions of maximum difference and print details in the following style:

```
The maximum difference in a row occurred in row 7
    between column 8 and column 9, where the difference was 156
The maximum difference in a column occurred in column 8
    between row 3 and row 4, where the difference was 60
```

Design:

First iteration

The program will be something like:

```
Open input file and load image from file
Check rows for maximum
Print row details
Check columns for maximum
Print column details
```

A program with three functions – `LoadImage()`, `ScanRows()`, and `ScanColumns()` – along with `main()` seems appropriate. The output statements should be simple and can be handled in `main()` rather than made into separate output routines.

Second iteration

The specification essentially guarantees that the input will be valid, so there will be no need to check the data read. Provided the file can be opened, a double `for` loop should be able to read the data. The array size, and loop limits, should be defined by `const` data elements; that will make things easier to change (and to test, using and a small array 8x10 rather than 40x50).

The `ScanRows()` and `ScanCols()` will have very similar structures. They have to initialize the "maximum difference" to zero (and the position to 0, 0). Then there will be an outer loop over the number of rows (columns) and an inner loop that will run to one less than the number of columns (rows). In order to find a difference, this inner loop has to compare elements '`[0]`' to '`[N-2]`' with elements '`[1]`' to '`[N-1]`'. The structure for both these routines will be something like the following (using rows for this example):

```
scan
    initialize max_difference, row, col all to zero
    for row 0 to maxrows - 1 do
           for col = 0 to maxcols -2 do
                  difference = image[row][col+1] -
                                image[row][col]
                  difference = abs (difference);
                  if( difference > max_difference)
                         replace max_difference, row, col
```

Third iteration

As this is a simple problem, we have already finished most of the design. work. All that remains is deciding on data representations and function prototypes.

Here it seems worth while using a "typedef" to introduce the name `Image` as name for a two dimensional array of short integers:

"Type Image"

```
const int kROWS = 40;
const int kCOLS = 50;

typedef short  Image[kROWS][kCOLS];
```

This will make the function prototypes a little clearer – we can specify an `Image` as an argument.

The program only needs one `Image`. This `Image` could be made a filescope variable but it is generally better style to have it defined as local to `main()` and arrange that it is passed to the other functions. (It doesn't make much difference in this simple program; but eventually you might need to generalize and have more than one `Image`. Then you would find it much more convenient if the `ScanRows()` and `ScanCols()` functions used an argument to identify the `Image` that is to be processed.)

Function prototypes

Following these decisions, the function prototypes become:

```
void LoadImage(Image picy);

void ScanRows(const Image picy, int& row, int& col, int& delta)

void ScanCols(const Image picy, int& row, int& col, int& delta)

int main();
```

Obviously `LoadImage()` changes the `Image` belonging to `main()`; the "scan" functions should treat the image as read-only data and so define it as `const`.

The two scan functions have three "reference to integer" (`int&`) arguments as explained in the problem introduction.

The header files will just be iostream and fstream for the file input, and stdlib (needed for `exit()` which will be called if get an open failure on the image file).

Implementation:

There is little of note in the implementation. As always, the program starts with the #includes that load the header files that describe the library functions employed; then there are const and typedef declarations:

```
#include <iostream.h>
#include <fstream.h>
```

```
#include <stdlib.h>

const int kROWS = 40;
const int kCOLS = 50;

typedef short  Image[kROWS][kCOLS];
```

In this example, the file opening and input of data are all handled by the same routine so it has the `ifstream` object as a local variable:

```
void LoadImage(Image picy)
{
    const int kNAMESIZE = 100;
    char   filename[kNAMESIZE];

    cout << "Enter name of image file" << endl;

    cin.getline(filename, kNAMESIZE-1, '\n');
    ifstream infile;

    infile.open(filename, ios::in | ios::nocreate);
    if(!infile.good()) {
                cout << "Couldn't open file." << endl;
                exit(1);
          }
    for(int i = 0; i < kROWS; i++)
          for(int j = 0; j < kCOLS; j++)
                infile >> picy[i][j];
    infile.close();
}
```

Generally, one would need checks on the data input. (Checks such as "Are all values in the specified 0…255 range? Have we hit the end of file before getting all the values needed?".) Here the specification said that no checks are needed.

The input routine would need to be changed slightly if you decided to save space by making `Image` an array of `unsigned char`. The restricted range of intensity values would make it possible to use an `unsigned char` to represent each intensity value and this would halve the storage required. If you did make that change, the input routine would have to use a temporary integer variable: `{ unsigned short temp; infile >> temp; picy[i][j] = temp; }`. If you tried to read into an "`unsigned char`" array, using `infile >> picy[i][j];`, then individual (non-blank) characters would be read and interpreted as the intensity values to initialize array elements!

The `ScanRows()` function is:

```
void ScanRows(const Image picy, int& row, int& col, int& delta)
{
    delta = 0;
    row = 0;
```

```
    col = 0;

    for(int r = 0; r < kROWS; r++)
           for(int c = 0; c < kCOLS-1; c++) {
                  int d = picy[r][c+1] - picy[r][c];
                  d = fabs(d);
                  if(d>delta) {
                         delta = d;
                         row = r;
                         col = c;
                  }
           }

}
```

Function `ScanCols()` is very similar. It just switches the roles of row and column.

Apart from the function calls, most of `main()` consists of output statements:

```
int main()
{
    int    maxrow, maxcol, maxdiff;
    Image thePicture;
    LoadImage(thePicture);
    cout << "Image loaded" << endl;

    ScanRows(thePicture, maxrow, maxcol, maxdiff);
    cout << "The maximum difference in a row occurred in row "
                  << maxrow
           << "\n\tbetween column " << maxcol
                  << " and column "  << (maxcol+1)
           << ", where the difference was "
                  << maxdiff << endl;

    ScanCols(thePicture, maxrow, maxcol, maxdiff);
    cout << "The maximum difference in a column occurred in"
                  "column " << maxcol
           << "\n\tbetween row " << maxrow
                  << " and row "  << (maxrow+1)
           << ", where the difference was "
                  << maxdiff << endl;

    return 0;
}
```

You would have to test this program on known test data before trying it with a real image! The values of the consts kROWS and kCOLS can be changed to something much smaller and an artificial image file with known data values can be composed. As noted in the previous example, it is often necessary to build such "scaffolding" of auxiliary test programs and test data files when developing a major project.

EXERCISES

1 The sending of encrypted text between parties often attracts attention – *"What do they want to hide?"*!
If the correspondents have a legitimate reason for being in communication, they can proceed more subtly. Rather than have M. Jones send K. Enever the message

```
'q%V"Q \IH,n,h^ps%Wr"XX)d6J{cM6M.bWu6rhobh7!W.NZ..Cf8
qlK7.1_'Zt2:q)PP::N_jV{j_XE8_i\$DaK>yS
```

(i.e. "Kathy Darling, the wife is going to her mother tonight, meet me at Spooners 6pm, Kisses Mike")

this secret message can be embedded in a longer carrier message that appears to be part of a more legitimate form of correspondence. No one is going to look twice at the following item of electronic mail:

```
From: M. Jones
To: K. Enver

Re: Report on Library committee meeting.

Hi Kate, could you please check this rough draft report that I've typed; hope
you can call me back today if you have any problems.

--------
Meeting of the Kibrary Committee, Tuesday Augast 15th,
Present: D. Ankers, Rt Bailey, J.P. Cauroma, …
…
```

Letters from the secret message are substituted for letters in the body of the original (the underlines shown above mark the substitutions, of course the underlining would not really be present). Obviously, you want to avoid any regularity in the pattern of "typing errors" so the embedding is done using a random number generator.

The random number generator is first seeded (the seed is chosen by agreement between the correspondents and then set each time the program is run). The program then loops processing characters from the carrier message and characters from the secret text. In these loops, a call is first made to the random number generator – e.g. `n = (rand() % IFREQ) + 3;`, the value obtained is the number of characters from the carrier message that should be copied unchanged; the next character from the secret text is then substituted for the next character of the carrier text. The value for `IFREQ` should be largish (e.g. 50) because you don't want too many "typing errors". Of course, the length of the carrier text has to be significantly greater than `(IFREQ/2)` times the secret text. "Encrypting" has to be abandoned if the carrier text is too short (you've been left with secret text characters but have hit end of file on the carrier). If the carrier is longer, just substitute a random letter at the next point where a secret text letter is required (you wouldn't want the apparent typing accuracy to

suddenly change). These extra random characters don't matter; the message will just come to an end with something like "Kisses MikePrzyqhn Tuaoa". You can rely on an intelligent correspondent who will stop reading at the point where the extracted secret text no longer makes sense.

Write the programs needed by your good fiends Mike and Kate.

The "encrypting" program is to take as input:
an integer seed for the random number generator,
the message (allow them a maximum of 200 characters),
the name of the file containing the original carrier text,
the name of the file that is to contain the adjusted text.
The encrypting program should print a warning "Bad File" if the carrier text is too short to embed the secret text; otherwise there should be no output to the screen.

The "decrypting" program is to take as input:
an integer seed for the random number generator,
the name of the file containing the incoming message,
it should extract the characters of the secret text and output them on cout.

2. Change the encrypt and decrypt programs so that instead of sharing a fixed `substitutions` table, they incorporate code from the auxiliary program that generates the table.

   The modified programs should start by prompting for three small positive integers. Two of these define values for variables that replace `kSTART` and `kINCREMENT`. The third serves as the seed for the random number generator that is used to create the `substitutions` table.

3. Write a program that will lookup up details of distances and travel times for journeys between pairs of cities.

   The program is to use a pair of initialized arrays to store the required data, one array to hodl distances, the other travel times. For example, if you had just three cities, you might have a distances array like:

```
const int NCITIES = 3;
const double distances[NCITIES][NCITIES] = {
     { 0.0, 250.0, 710.0 },
     { 250.0, 0.0, 525.0 },
     { 710.0, 525.0, 0.0 }
};
```

   with a similar array for travel times.

   The program is to prompt the user to enter numbers identifying the cities of interest, e.g.

```
Select starting city:
1) London,
2) Paris,
3) Venice
CIty Number :
```

   Given a pair of city identification numbers, the program prints the details of distance and travel time. (Remember, users generally prefer numbering schemes that start at 1 rather

than 0; convert from the users' numbering scheme to an internal numbering suitable for zero based arrays.)

You should plan to use several functions. The city names should also be stored in an array.

4. The arrays of distance and travel time in exercise 3 are both symmetric about the diagonal. You can halve the data storage needed by using a single array. The entries above the diagnoal represent distances, those below the diagonal represent travel times.

   Rewrite a solution to assignment 3 so as to exploit a single travel-data array.