```
            double temp = 0.0;
            for (int k = 0; k < 5; ++k)
                temp += X[i][k] * Y[k][j];
            Z[i][j] = temp;
        }
```

In terms of matrices, the equivalent mathematical equations are much simpler. The addition is given by

$$C = A + B$$

and the multiplication by

$$Z = X * Y$$

However, it is only by introducing overloaded operators (in Chapter 9) that we can truly manipulate matrices, rather than matrix components, as objects.

Exercise

By assigning appropriately chosen integers to the elements of `A[][]`, `B[][]`, `X[][]` and `Y[][]`, check the correctness of matrix addition and multiplication as implemented above. You should display the calculated matrices as two-dimensional arrays.

Since a memory location is specified by a single address, the two or more dimensions of a multi-dimensional array must be mapped into the linear address space of physical memory. This mapping is often known as a *storage map*. In C++, two-dimensional arrays are stored by rows and a typical storage map is shown in Figure 6.4, where the symbol x_0 represents the array element `&x[0][0]`.

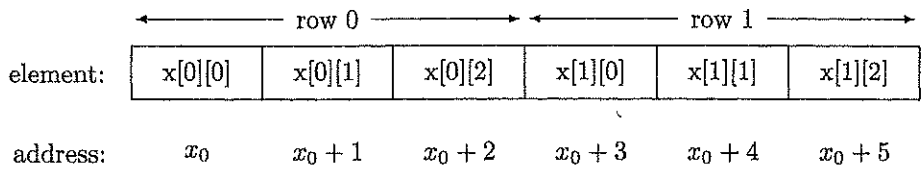

Figure 6.4: Storage map for `x[2][3]`.

6.5.1 Pointers and Multi-dimensional Arrays†

An array defined by:

```
int x[2][3];
```

is accessed via `x[i][j]`, which is equivalent to:

```
*(&x[0][0] + 3 * i + j)
```

In fact, two-dimensional arrays have no more significance than this equivalence. In this example, `x[0][0]` is the element at the low end of the array and therefore `&x[0][0]` is the base address of the array.

Great care should be exercised when using pointers to access multi-dimensional arrays. The constant, `x`, is also the base address of this array, but

```
*(x + 3 * i + j)
```

is not equivalent to `x[i][j]` since `(x + i)` is actually the base address of row `i`; for instance, `(x + 1)` is the base address of the row 1. A correct way of using `x`, rather than `&x[0][0]`, to access an element of the two-dimensional array is to use:

```
(*(x + i))[j]
```

The outer parentheses are required because `[]` binds tighter than the dereferencing operator. In fact, by studying the above expression, we can see how the notation for two-dimensional arrays arises. The expression, `*(x+i)`, is the same as `x[i]`, the base address of row `i`, so the whole expression is directly equivalent to `x[i][j]`. There are two other, equally devious, ways of rewriting `x[i][j]`; these are

```
*((*(x + i)) + j)
```

and

```
*(x[i] + j)
```

It is worth convincing yourself that these expressions really are equivalent to `x[i][j]`, although in practice it is best to stick to the more obvious notation.

Exercise

You have now encountered four different non-standard ways of accessing a two-dimensional array by using pointers. Write a program that uses each of these four techniques to write numbers to a 2×3 array of type `int`. Use the standard way of accessing the array to demonstrate that the correct values are assigned in each case.

For arrays of three dimensions and higher, the storage map is a straightforward extension of the two-dimensional case. For instance, if we make the definition:

```
float y[2][3][4];
```

then an element, `y[i][j][k]`, can equivalently be accessed by:

```
*(&y[0][0][0] + 3 * 4 * i + 4 * j + k);
```

Needless to say, there are devious ways of rewriting this expression in terms of y rather than `&y[0][0][0]`.

We don't need to know how arrays are stored in order to use them, but doing so can help us to understand (and even reduce) the overhead caused by indexing into multi-dimensional arrays. For instance, if a calculation involves going down columns, one step at a time, then it may be faster to rearrange the code so that the stepping is done along rows. In Section 7.3.2 we show how knowledge of the array storage map can speed up a typical numerical application.

It is worth emphasizing that in many situations it is better to keep to the standard array notation since a small increase in efficiency is not worth the risk of making a mistake with the pointer arithmetic. However, in situations where using pointer arithmetic would lead to a worthwhile increase in speed, then the more advanced techniques of C++ should be used to encapsulate the pointer arithmetic in isolated pieces of code that can be carefully checked.