MA400: Financial Mathematics

Introductory Course

Lecture 5: Pointers and Arrays

Pointers

One-dimensional arrays

Multi-dimensional arrays

The address-of operator: &

Whenever we define or declare a variable in a program, the C++ compiler reserves an appropriate amount of memory for the storage of that variable.
Each variable therefore has an address in memory, from which its allocated amount of bytes starts at.

We can access this address by use of the **address-of operator**, &, which can be read as "return the address of . . . ".

E.g.

```
&x;          // Return the address of variable x
&income      // Return the address of variable income
```

The address-of operator: &

For example

```
int i = 1;
double x = 3.0;
float z = 4.0;
cout << "Address" << "\t\t" << "Size" << '\t' << "Value"
     << '\n';
cout << &i << '\t' << sizeof(int) << '\t' << i
     << '\n';
cout << &x << '\t' << sizeof(double) << '\t' << x
     << '\n';
cout << &z << '\t' << sizeof(float) << '\t' << z
     <<'\n';
```

produces

```
Address          Size    Value
0xbffff7fc       4       1
0xbffff7f0       8       3
0xbffff7ec       4       4
```

The address-of operator: &

Note that both the numbers (of course) and the format of the preceding output is dependent on your compiler and system. E.g.

- addresses are given in hexadecimal format.
- addresses differ by the size of the types they store.

You cannot expect this in general.

The address-of operator: &

You can take the address of any variable you define.

Furthermore, you can take the address of any constants that you define:

```
const double pi = 3.142;
&pi;              // This returns a valid address.
```

However, you cannot take the address of a numbers:

```
&3.142            // Wrong!
```

Nor can you take the address of an expression:

```
&(pi + 1.0);      // Wrong!
```

Dereferencing, or indirection, operator: $*$

Given an address, we can return the value stored at that address using the **dereferencing**, or **indirection**, **operator** `*`.
Given a memory location as an argument (on the right), the dereferencing operator returns the value stored at that address.

Thus, given the definition:

```
int i = 1;
```

then `i` has data type `int` and value 1. Furthermore:

- the address in memory where `i` is stored is `&i`,
- the value stored at that address is 1, and can be accessed either by `i` or `*&i`.

Pointers

At times, it may be easier, and more efficient, to work with the addresses of our stored variables, rather than with them directly. To do this, we use **pointers**.

For each variable, a pointer must indicate:

- the starting address in memory for the variable, and
- the number of subsequent bytes used by the variable.

Pointers are thus variables themselves: storing the addresses of other variables.
To indicate how much memory is used by the variable, we associate a different pointer type for each variable data type.

Declaring pointers

```
int i, count, no_of_kangaroos;
double x, pi;
float theta;

int *pt_1;       // pt_1 is a pointer to the type int
double *pt_2;    // pt_2 is a pointer to the type double
float *pt_3;     // pt_3 is a pointer to the type float
```

With the above declarations:

- the pointer `pt_1` can hold the address of any integer variable;
- the pointer `pt_2` can hold the address of any double variable;
- the pointer `pt_3` can hold the address of any float variable.

Declaring pointers

Pointers for one data type cannot be used to hold the addresses of variables of other data types.
There is no implicit recasting of types.

If we tried to make the pointer `pt_3`, which has type `float`, point to the address of the `double` variable `x`, i.e. `&x`, then our compiler should fail with a message along the lines of

```
cannot convert float* to double in assignment
```

Declaring pointers

As with variables, multiple declarations can be made on a single line:

```
int *pt_i, *p1, *j;  // Three pointers to the type int
```

Note that while there is nothing wrong with the following:

```
int i, j, *pt_a, *pt_i;
```

it might be easier, and less error prone, to keep variable and pointer declarations separate.

```
int i, j;
int *pt_a, *pt_i;
```

Using pointers

Dereferenced pointers can be used in exactly the same was a variables of their corresponding type would be:

```
int i, j, k;
int *pt_i, *pt_j;
pt_i = &i;              // Assigns address of i to pt_i
pt_j = &j;              // Assigns address of j to pt_j
i = 1;
j = 2;
k = *pt_i + *pt_j;      // *pt_i behaves like i, and
                        // *pt_j behaves like j, so
                        // 3 is assigned to k, i.e. 1 + 2
*pt_i = 10;             // 10 is assigned to i
```

Exercise: Using pointers

Rewrite the quadratic program given previously so that as many operations as possible are performed using dereferenced pointers.

Constants and pointers

Pointers to const types are allowed, but must be specified with the const type as well.

```
int variable = 0;
const int constant = 1;
int *pt_v;
const int *pt_c;
```

When pointing to a constant type:

```
pt_c = &constant;      // Correct
pt_v = &constant;      // Wrong!
```

The following are valid, and sensible:

```
pt_v = &variable;
*pt_v = 1;
```

Constant pointers can point to variables, but suffer restrictions.

```
pt_c = &variable;      // Valid.
*pt_c = 1;             // Wrong!  Address is read-only.
```

Pointers and pointers

Pointers are variables that hold addresses to other variables. As such, we can also define pointers to pointers.

```
int i;      // Memory defined to store an integer
int *pt;    // Memory defined to store the
            //   address of an integer
int **pt_pt;  // Memory defined to store the
            //   address of a pointer to an integer
```

Given the above declarations and the following assignments:

```
i = 1;
pt = &i;
pt_pt = &pt;
```

we have that `i`, `*pt` and `**pt_pt` all refer to the same value.

Example

```
double x          , y;
double *pt_x      , *pt_y;
double **pt_pt_x, **pt_pt_y;

   pt_x = &x      ,    pt_y = &y     ;
pt_pt_x = &pt_x  , pt_pt_y = &pt_y  ;
      x = 11.11  ,       y = 9.89   ;
```

You should complete the code to check that the following expressions return the same value:

```
x * y;
*pt_x * *pt_y;
**pt_pt_x * **pt_pt_y;
```

Whitespace and the & and * operators

Recall that C++ tends to forget whitespace, so that the following are equivalent pairs:

```
double *pt_x;
double * pt_x;

pt_pt_x = &x;
pt_pt_x = & x;
```

Furthermore, * and & have higher precedence than *most* arithmetical operators, so the following are equivalent:

```
**pt_pt_x * **pt_pt_y;
**pt_pt_x***pt_pt_y;
```

Initializing pointers

Pointers, as with any other variable, can be initialized at the time of declaration. However, the syntax is a little different.

For an integer variable `i`, the following are equivalent:

```
int *pt_i;
pt_i = &i;
```

and

```
int *pt_i = &i;
```

Thus the `*` occurs in just two places:

- when declaring a pointer variable, and
- when dereferencing a pointer variable.

Initializing pointers

As with all variables, you must take care to **initialize pointers before you use them**.
In fact, since they refer to addresses in memory, the failure to ensure you use initialize pointers can be quite disastrous.

```
int i, j, k;
int *pt_i, *pt_j;
pt_j = &j;      // Forget to assign &i to pt_i here
i = 1;
j = 2;
k = *pt_i + *pt_j;     // WRONG: pt_i is arbitrary
// k should be 3, but is arbitrary, e.g. -1125351075
*pt_i = 10;
```

However, the real danger is the final line, which tries to write the value 10 to an arbitrary location in memory.

Null pointers

The **null pointer** is a special constant pointer that is guaranteed not to be a valid memory address.

```
int *pt_i = 0, *pt_j;
```

When trying to dereference this null pointer (e.g. in the calculation of `k`), a `run-time error` will result, causing the program to terminate with an error message.

Arrays

Arrays set aside a contiguous block of memory to store a fixed number of objects of the same data type.
That is, they are more than just a collection of similar objects.

To define arrays of fixed size (and data type) we use a single integer parameter in square brackets:

```
int i[10];
double x[100];
char c[80];
```

As well as numbers, any expression that evaluates to an integral constant **at compile time** may also be used to specify the number of elements in an array.

```
const int array_length = 10;
double prices[2 + array_length];
```

Accessing elements in an array

```
int i[10];   // Define an array of 10 integers
```

We can access the elements of this array using the same square bracket notation.
However, the number inside `[ ]` now acts as the index, or position, we are interested in.

Note: For an array of length n, the indexing starts from 0 and terminates at $n - 1$.

The above example then has the following 10 elements:

$$i[0] \quad i[1] \quad i[2] \quad i[3] \quad i[4] \quad i[5] \quad i[6] \quad i[7] \quad i[8] \quad i[9]$$

It is very easy to get confused about this – e.g. start the indexing from 1 and not 0 – so take care.
In particular, `i[10]` does not refer to any element in the array above!

Accessing elements in an array

Given the array of values:

$$i[0] \quad i[1] \quad i[2] \quad i[3] \quad i[4] \quad i[5] \quad i[6] \quad i[7] \quad i[8] \quad i[9]$$

we can assign these elements values via:

```
i[6] = 24;
```

This assigns the value 24 to the element with index 6, i.e. the seventh element, in the array.
Accessing the stored values is done in a similar way:

```
int j;
j = i[6];   // Assigns the value of i[6], i.e. 24, to j
```

Initializing arrays

As well as filling up arrays using the subscript notation, we can initialize them when they are defined using an **initialization list**.

```
int i[6] = {3, 2, 7, 4, 6, 3};
```

If the array size is not specified, then this is taken to be the number of elements in the list provided:

```
double temp[] = {0.0, 100.0};   // I.e. double temp[2]
```

If the array size is larger than the list, then it is the first elements of the array that are filled: the remainder are **initialized as zero**:

```
int t[5] = {5, 3, 11};  // t[3] and t[4] are set to 0
```

Finally, the number of elements in the initialization list cannot be greater than the size of the array:

```
double p[2] = {1.0, 3.8, 2.7}; // Wrong.
```

Arrays and vectors

While there are certain similarities, and we can consider the following:

```
double coords[3];
```

to define a 3-dimensional vector, say, C++ does not come with any functions that we would want for a vector class.

For example, there are no notions of:

- the addition of two arrays, or
- the scalar multiplication of an array by a scalar number.

Of course, there is no reason why a class of vector objects could not be defined in C++ and, in fact, this is what you will see later in the course.

The relationship between pointers and arrays

Recall, an array is a **contiguous** region of memory storage. So

```
int i[10];
```

prompts the compiler to set aside 10 **consecutive** addresses in memory sufficient to hold 10 integers.
The starting address will be the address of the first integer, `i[0]`:

`&i[0]`.

The address of the next integer, `i[1]`, i.e.

`&i[1]`,

is also the same as the next address after `&i[0]`, i.e.

`&i[0] + 1`.

In general, `i[i]` is equivalent to:

`*(&i[0] + i)`.

Arithmetic on pointers and arrays

Because of this relationship between arrays and pointers, some useful arithmetic procedures are available to us:

```
double *pt;
double a[10];
pt = &a[0];          // pt points to element 0
++pt;                // pt points to element 1
pt += 4;             // pt points to element 5
pt = pt - 2;         // pt points to element 3
```

Note that our pointer, `pt`, has been defined to point to type `double`.
That is, the compiler knows exactly how many bytes it should advance the pointer under incrementation to point to the **next** `double`.

Arithmetic on pointers and arrays

As well as incrementing and decrementing pointers, there is a notion of subtraction of pointers.

A meaningful notion of subtraction here would be that the difference between, say, the 7^{th} element and the 4^{th} element, **in the same array**, is 3.
It would make no sense to try to compare element positions from different arrays.

Using the `ptrdiff_t` data type defined in `<cstddef>`, C++ can do this.

Arithmetic on pointers and arrays

```
double a[10], b[20];
double *pt_1, *pt_2;
ptrdiff_t diff;              // diff is of type ptrdiff_t

pt_1 = &a[1];
pt_2 = &a[4];
diff = pt_2 - pt_1;          // Assigns 3 to diff
diff = pt_1 - pt_2;          // Assigns -3 to diff

pt_1 = &a[0] + 10;           // Equivalent to &a[10]
pt_2 = &a[0] + 9;            // Equivalent to &a[9]
diff = pt_1 - pt_2;          // Correctly assigns 1

pt_1 = &b[19];
diff = pt_1 - pt_2;          // WRONG: undefined
```

Arithmetic on pointers and arrays

Finally, we can also compare pointers although, they must point to elements in the same array.

```
double *pt_1, *pt_2;
double x[10];

pt_1 = &x[9];
pt_2 = &x[3];

bool b1 = pt_2 > pt_1;    // Assigns b1 the value false
bool b2 = pt_2 <= pt_1;   // Assigns b2 the value true

bool b3 = pt_1 != pt_2;   // Assigns b3 the value true
bool b4 = pt_1 == pt_2;   // Assigns b4 the value false
```

Arithmetic on pointers and arrays

Given the declaration

```
double x[10];
```

assume we have defined pointers `pt_i` to each of the `x_i`.
Then we have

x[0]		x[1]		x[2]		...		x[9]		x[10]
pt_0	<	pt_1	<	pt_2	<	...	<	pt_9	<	pt_10

That is, it can be valid to use the array size as an array index: **for those operations that do not write to or call its value**.

Arithmetic on pointers and arrays

Using these basic arithmetical properties, we can replace rewrite operations on arrays in terms of pointer arithmetic.
Thus, consider summing the entries of an array of size 10: `x[10]`.

```
for ( int i = 0 ; i < 10 ; ++i )
  sum += x[i];
```

In pointer arithmetic, this becomes:

```
double *pt, *pt_end;
pt = &x[0];
pt_end = pt + 10;      // -> one element beyond the array
while ( pt < pt_end )
  sum += *pt++;
```

Note we can also declare and initialize the pointers in one statement:

```
double *pt = &x[0], *pt_end = pt + 10;
```

The base address of an array

Recall that the relationship between arrays and pointers means that, for some array `x[ARRAY_SIZE]`, the following are equivalent:

`x[i]`, `*(&x[0] + i)`.

A notational shorthand that is heavily used with arrays is that `x` is defined to be the base address of the array.
That is, the following are equivalent:

`&x[0]`, `x`.

That is, we can also access the element `x[i]` using:

`*(x + i)`.

Multi-dimensional arrays

C++ is also capable of dealing with multi-dimensional arrays.

```
double A[2][3];     // Defines a 2 x 3 array of doubles
int J[2][4][6];     // Defines a 2 x 4 x 6 array of ints
```

As before, we can access, and assign values to, the elements of these arrays using the subscript method:

```
A[0][0] = 1.0;              // Very first entry of an array
J[1][2][2] = 10;
```

Once again, care must be taken that we do not get the indexing confuse and try to write to areas outside the allocated memory:

```
A[2][3] = 0.0;              // WRONG
```

Initializing multi-dimensional arrays

Multi-dimensional arrays can be initialized using comma-separated, nested braces, in a similar fashion to one-dimensional arrays:

```
int A[2][3] = { {1, 2, 3} , {4, 5, 6} };
```

Again, assignment of values from the list is from the low-end to the high-end of the array, so that partial lists are allowed:

```
int B[2][3] = { {1, 2} , {4} };
```

Here, `B[0][2]`, `B[1][1]` and `B[1][2]` are initialized as zero.

Finally, the use of initialization lists allow us to implicitly define the first bound:

```
int C[][3][3] = { {}, {1}, {1, 2} };
```

Multidimensional arrays and matrices

In the two-dimensional case, one can identify the first index as labelling rows, and the second columns.
While one may represent matrices with C++ arrays, there are no matrix operations defined within C++.

For example, there are no notions of:

- the addition of two $n \times m$ matrices, or
- the multiplication of an $i \times k$ and $k \times j$ matrix.

We must define these operations ourselves.

A first attempt at matrix multiplication using arrays

Consider the multiplication of a 3×5 matrix X, and a 5×4 matrix Y. This yields a 3×2 matrix Z, where:

$$Z_{ij} = \sum_{k=0}^{5} X_{ik} Y_{kj}, \qquad i = 1, 2, 3, \quad j = 1, 2, 3, 4$$

The following code fragment should perform the necessary calculations:

```
double X[3][5], Y[5][4], Z[3][4];
for ( int i = 0 ; i < 3 ; ++i ) {
  for ( int j = 0 ; j < 4 ; ++j ) {
    double temp = 0.0;
    for ( int k = 0 ; k < 5 ; ++k)
      temp += X[i][k] * Y[k][j];
    Z[i][j] = temp;
  }
}
```

Exercise: Adding two matrices using arrays

Once you have understood the previous code fragment, write a program to add two 4×5 matrices together.

You should define your matrices as appropriate arrays of doubles, but you can verify your method by filling them with suitable integers.

Your program should display the calculated matrices as two-dimensional arrays.
Hint: Use the tab and newline character literals `'\t'` and `'\n'`.

The storage map

Regardless of the dimensions of your arrays, they are always mapped to a linear address space. Thus, while the array `X[2][3]` can be represented by the matrix:

$$\begin{pmatrix} \texttt{X[0][0]} & \texttt{X[0][1]} & \texttt{X[0][2]} \\ \\ \texttt{X[1][0]} & \texttt{X[1][1]} & \texttt{X[1][2]} \end{pmatrix}$$

its storage map is in fact:

`X[0][0]`	`X[0][1]`	`X[0][2]`	`X[1][0]`	`X[1][1]`	`X[1][2]`
X_0	$X_0 + 1$	$X_0 + 2$	$X_0 + 3$	$X_0 + 4$	$X_0 + 5$

Here, the X_0 represents $\&X[0][0]$.

Pointers and multi-dimensional arrays

Recall that for the one-dimensional array defined by:

```
int x[10];
```

`x` was identified with the base address of the array, giving the equivalence of the following three forms:

`x[i]`, `*(&x[0] + i)`, `*(x + i)`.

Defining a multi-dimensional array with:

```
int X[2][3];
```

we also have the identification of `X` with the base address of the array, which gives us the equivalence of:

`X[i][j]`, `*( &X[0][0] + 3 * i + j )`.

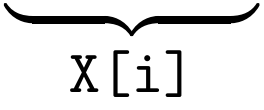
Pointers and multi-dimensional arrays

However, the following are **not equivalent**:

`X[i][j]`, `*(X + 3 * i + j)`.

In fact, `(X + i)` is the base address of row `i`, which is itself a one-dimensional array (in our example).
Therefore, it is the following that are equivalent:

`X[i][j]`, `( *(X + i) )[j]`



We need the outer parentheses, as `[]` binds more tightly than `*`

Pointers and multi-dimensional arrays

In fact, we have **four** equivalent ways of writing `X[i][j]`:

```
X[i][j]

(*(X + i))[j]

*( X[i] + j )

*( (*(X + i)) + j )
```

Pointers and multi-dimensional arrays

In an analogous manner, for the array defined by:

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

the element `y[i][j][k]` can also be accessed by:

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

With of course, a number of intermediary forms using `y` instead of `y[0][0][0]`.

Pointers and multi-dimensional arrays

Ultimately, these equivalent forms merely point out that, regardless of the dimensions of our array, its data is still stored in a linear address space.
In the case of two-dimensional arrays, we have seen that the elements are stored row-wise.

This insight gives us another way to initialize a two-dimensional array: using a single, flat list.
The following are therefore equivalent:

```
int A[2][3] = { {1, 2, 3}, {4, 5, 6} };
int A[2][3] = { 1, 2, 3, 4, 5, 6 };
```

Pointers and multi-dimensional arrays

It also allows us to identify potential inefficiencies in our code.

For example, in writing a program to add two 3×4 matrices, say A and B, together, the natural method would be to use nested loops. However, using the storage map, we could actually perform this with just one loop (consisting of 20 iterations).

`A[0][0]`	`A[0][1]`	`A[0][2]`	`A[0][3]`	`A[1][0]`	...
A_0	$A_0 + 1$	$A_0 + 2$	$A_0 + 3$	$A_0 + 4$	...
`B[0][0]`	`B[0][1]`	`B[0][2]`	`B[0][3]`	`B[1][0]`	...
B_0	$B_0 + 1$	$B_0 + 2$	$B_0 + 3$	$B_0 + 4$	...

i.e.

$$*A_0 + *B_0, \quad *(A_0+1) + *(B_0+1), \quad \ldots, \quad *(A_0+19) + *(B_0+19).$$

Arrays of pointers

As well as building up arrays from the fundamental data types, we can define arrays of pointers.
The notation should be familiar to us by now:

```
int *pt_i[10];   // Stores 10 addresses of integers
double *pt_x[5]; // Stores 5 addresses of doubles
```

Here, the index operator `[]`, has higher precedence than the dereferencing operator `*`.
Thus this defines an array of pointers, rather than a pointer to an array.
The latter is accomplished by using parentheses:

```
float (*pt_a)[5];
```

Example: Arrays of pointers

```
double x, y, z;

double *pt[3];      // An array of 3 pointers to doubles
pt[0] = &x;
pt[1] = &y;
pt[2] = &z;

x = 1.11;
y = 2.22;
z = 3.33;

double **p, **p_end;
p = pt;
p_end = p + 3;

while (p < p_end)
  cout << "data = " << **p++ << "\n";
```

Example: Arrays of pointers

```
data = 1.11
data = 2.22
data = 3.33
```

Ragged arrays

Arrays of pointers can be used to create **ragged arrays**, that is arrays whose rows have different lengths.
A simple example would be the use of a binomial tree for pricing an option.
This gives rise to the tree of values:

```
S[0][0]
S[1][0]   S[1][1]
S[2][0]   S[2][1]   S[2][2]
```

What is the best structure to hold these values?

Ragged arrays

A square array of dimensions $T \times T$ may not be the most efficient way of storing these.

T	nodes	$T \times T$	blanks
2	3	4	1
3	6	9	3
4	10	16	6
5	15	25	10

T	$\frac{1}{2}T(T+1)$	T^2	$\frac{1}{2}(T-1)T$

Ragged arrays

The following code fragment defines an appropriate ragged array for this example:

```
// Define arrays of just sufficient size to store
// stock prices at each time period, t = 0, 1, 2
double T0[1] = {20};
double T1[2] = {18,22};
double T2[3] = {16.2,19.8,24.2};

// Define an array of pointers to the data arrays:
double *S[3];
S[0] = T0;
S[1] = T1;
S[2] = T2;
```

Ragged arrays

The following code fragment prints out the content of your ragged array.

```
for (int i = 0 ; i < 3 ; ++i) {
  for (int j = 0 ; j <= i ; ++j) {
    cout << "S[" << i << "][" << j << "] = "
         << S[i][j] << '\t';
  }
  cout << '\n';
}
```

This produces:

```
S[0][0] = 20
S[1][0] = 18    S[1][1] = 22
S[2][0] = 16.2  S[2][1] = 19.8  S[2][2] = 24.2
```

Ragged arrays

To get an idea of the total amount of storage used in our ragged array example:

```
sizeof(S);   // Returns 3 * 4 = 12
sizeof(T0);  // Returns 1 * 8 = 8
sizeof(T1);  // Returns 2 * 8 = 16
sizeof(T2);  // Returns 3 * 8 = 24
```

The total footprint of this ragged array is then:

$$12 + 8 + 16 + 24 = 60$$

For the size of a 3×3 array:

```
double array[3][3];
sizeof(array);
```

The `sizeof()` function will return

$$3 \times 3 \times 8 = 72.$$

Ragged arrays

It should be easy to calculate storage requirements for cases other than $T = 2$.

$$(2 \times 4) + (8 + 16) = 32 \quad \text{(S2)}$$
$$(3 \times 4) + (8 + 16 + 24) = 60 \quad \text{(S3)}$$
$$(4 \times 4) + (8 + 16 + 24 + 32) = 96 \quad \text{(S4)}$$
$$(5 \times 4) + (8 + 16 + 24 + 32 + 40) = 140 \quad \text{(S5)}$$

Comparing against the sizes of a simple $T \times T$ array, we have:

T	$T \times T$ storage	ragged
2	4	32	32
3	9	72	60
4	16	128	96
5	25	200	140

Exercise

Given a set of real numbers, $x_1, \ldots, x_N$, the mean, $\bar{x}$ is defined by:

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i,$$

the variance, Var, by:

$$Var(x_1 \ldots x_N) = \frac{1}{N-1} \sum_{i=1}^{N} (x_i - \bar{x})^2,$$

and the standard deviation by:

$$\sigma(x_1 \ldots x_N) = \sqrt{Var(x_1 \ldots x_N)}.$$

Write a program that calculates these three values from a list of numbers stored as a one-dimensional array.
For test purposes you can use the `rand()` function, declared in `<cstdlib>`, to generate the list. (You should find that the mean is approximately 0.5.)
Try lists of increasing length. Is there any point in generating more than `RAND_MAX` random numbers?

Exercise

Write a program that **initializes** the `int` array `a[2][3][4]`, with values corresponding to:

$$a_{ijk} = (1 - i)(2 - j)(3 - k)$$

by using nested, comma-separated lists.
Elements that have the value zero should be initialized by default and **not** explicitly.
Check your program by comparing the initialization actually achieved with the expected values of `a[i][j][k]`.

Exercise

Use nested, comma-separated lists to initialize the array `a[2][3][4]` so that it corresponds to:

$$a_{0ij} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

and

$$a_{1ij} = \begin{pmatrix} 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \end{pmatrix}$$

Using only pointer arithmetic and the array name (that is `a`, but **not** `a[0][0][0]` or `a[i][j][k]`), list the array elements and hence verify your method of access.

Exercise

P and Q are polynomials in x and y, given by:

$$P = \sum_{i=0}^{3} \sum_{j=0}^{3} p_{ij} x^i y^j, \qquad Q = \sum_{i=0}^{3} \sum_{j=0}^{3} q_{ij} x^i y^j,$$

where p_{ij} and q_{ij} are integers.
Use two-dimensional arrays to add P and Q for various values of p_{ij} and q_{ij}.
The output should be given in terms of a polynomial, rather than a list of coefficients.