Declaration of fixed-sized vectors and matrices

In the program below we discuss some essential features of vector and matrix handling where the dimensions are declared in the program code.

In **line a** we have a standard C++ declaration of a vector. The compiler reserves memory to store five integers. The elements are vec[0], vec[1],....,vec[4]. Note that the numbering of elements starts with zero. Declarations of other data types are similar, including structure data.

The symbol vec is an element in memory containing the address to the first element vec[0] and is a pointer to a vector of five integer elements.

In **line b** we have a standard fixed-size C++ declaration of a matrix. Again the elements start with zero, matr[0][0], matr[0][1], ..., matr[0][4], matr[1][0],.... This se- quence of elements also shows how data are stored in memory. For example, the element matr[1][0] follows matr[0][4]. This is important in order to produce an efficient code and avoid memory stride.

There is one further important point concerning matrix declaration. In a similar way as for the symbol **vec**, **matr** is an element in memory which contains an address to a vector of three elements, but now these elements are not integers. Each element is a vector of five integers. This is the correct way to understand the declaration in **line b**. With respect to pointers this means that matr is pointer-to-a-pointer-to-an-integer which we can write ∗∗matr. Furthermore ∗matr is a-pointer-to-a-pointer of five integers. This interpretation is important when we want to transfer vectors and matrices to a function.

In **line c** we transfer vec[] and matr[][] to the function sub_1(). To be specific, we trans- fer the addresses of vec[] and matr[][] to sub_1().

In **line d** we have the function definition of subfunction(). The **int** vec[] is a pointer to an integer. Alternatively we could write **int** ∗vec. The first version is better. It shows that it is a vector of several integers, but not how many. The second version could equally well be used to transfer the address to a single integer element. Such a declaration does not distinguish between the two cases.

The next definition is **int** matr[][5]. This is a pointer to a vector of five elements and the compiler must be told that each vector element contains five integers. Here an alternative version could be int (∗matr)[5] which clearly specifies that matr is a pointer to a vector of five integers.

```
 int main()
{
  int k,m, row = 3, col = 5;
  int  vec[5];   // line a
  int  matr[3][5]; // line b
  // Fill in vector vec
  for (k = 0; k < col; k++) vec[k] = k;
  // fill in matr
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++) matr[m][k] = m + 10*k;
  }
  // write out the vector
  cout << `` Content of vector vec:'' << endl;
  for (k = 0; k < col; k++){
     cout << vec[k] << endl;
  }
  // Then write out the matrix
  cout << `` Content of matrix matr:'' << endl;
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++){
       cout << matr[m][k] << endl;
     }
  }
  subfunction(row, col, vec, matr); // line c
  return 0;
} // end main function

void subfunction(int row, int col, int vec[], int matr[][5]); // line d
{
  int k, m;
  // write out the vector
  cout << `` Content of vector vec in subfunction:'' << endl;
  for (k = 0; k < col; k++){
     cout << vec[k] << endl;
  }
  // Then write out the matrix
  cout << `` Content of matrix matr in subfunction:'' << endl;
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++){
       cout << matr[m][k] << endl;
     }
  }
} // end of function subfunction
```

There is at least one drawback with such a matrix declaration. If we want to change the dimension of the matrix and replace 5 by something else we have to do the same change in all functions where this matrix occurs.

There is another point to note regarding the declaration of variables in a function which includes vectors and matrices. When the execution of a function terminates, the memory required for the variables is released. In the present case memory for all variables in main() are reserved during the whole program execution, but variables which are declared in subfunc- tion() are released when the execution returns to main().

Runtime Declarations of Vectors and Matrices in C++

We change thereafter our program in order to include dynamic allocation of arrays. As men- tioned in the previous subsection a fixed size declaration of vectors and matrices before compilation is in many cases bad. You may not know beforehand the actually needed sizes of vectors and matrices. In large projects where memory is a limited factor it could be important to reduce memory requirement for matrices which are not used any more. In C an C++ it is possible and common to postpone size declarations of arrays untill you really know what you need and also release memory reservations when it is not needed any more. The following program shows how we could change the previous one with static declarations to dynamic allocation of arrays.

In **line a** we declare a pointer to an integer which later will be used to store an address to the first element of a vector. Similarily, **line b** declares a pointer-to-a-pointer which will contain the address to a pointer of row vectors, each with col integers. This will then become a matrix with dimensionality [col][col]

In **line c** we read in the size of vec[] and matr[][] through the numbers row and col.

Next we reserve memory for the vector in **line d**. In **line e** we use a user-defined function to reserve necessary memory for matrix[row][col] and again matr contains the address to the reserved memory location.

The remaining part of the function main() are as in the previous case down to **line f**. Here we have a call to a user-defined function which releases the reserved memory of the matrix. In this case this is not done automatically.

In **line g** the same procedure is performed for vec[]. In this case the standard C++ library has the necessary function.

Next, in **line h** an important difference from the previous case occurs. First, the vector declaration is the same, but the matr declaration is quite different. The corresponding parameter in the call to sub_1[] in **line g** is a double pointer. Consequently, matr in **line h** must be a double pointer.

```
 int main()
{
  int k,m, row = 3, col = 5;
  int  vec[5];   // line a
  int  matr[3][5]; // line b

  cout << `` Read in number of rows'' << endl; // line c
  cin >> row;
  cout << `` Read in number of columns'' << endl;
  cin >> col;

  vec = new int[col];                // line d
  matr = (int **)matrix(row,col,sizeof(int)); // line e
  // Fill in vector vec
  for (k = 0; k < col; k++) vec[k] = k;
  // fill in matr
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++) matr[m][k] = m + 10*k;
  }
  // write out the vector
  cout << `` Content of vector vec:'' << endl;
  for (k = 0; k < col; k++){
     cout << vec[k] << endl;
  }
  // Then write out the matrix
  cout << `` Content of matrix matr:'' << endl;
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++){
       cout << matr[m][k] << endl;
     }
  }
  subfunction(row, col, vec, matr); // line f
  free_matrix((void **) matr); // line g
  delete vec[];
  return 0;
} // end main function

void subfunction(int row, int col, int vec[], int matr[][5]); // line h
{
  int k, m;
  // write out the vector
  cout << `` Content of vector vec in subfunction:'' << endl;
  for (k = 0; k < col; k++){
     cout << vec[k] << endl;
  }
  // Then write out the matrix
  cout << `` Content of matrix matr in subfunction:'' << endl;
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++){
       cout << matr[m][k] << endl;
     }
  }
} // end of function subfunction
```

As an alternative, you could write your own allocation and deallocation of matrices. This can be done rather straightforwardly with the following statements. Recall first that a matrix is represented by a double pointer that points to a contiguous memory segment holding a sequence of double* pointers in case our matrix is a double precision variable. Then each double* pointer points to a row in the matrix. A declaration like double** A; means that A[i] is a pointer to the i + 1-th row A[i] and A[i][j] is matrix entry (i, j). The way we would allocate memory for such a matrix of dimensionality $n \times n$ is for example using the following piece of code

```
int n;
double ** A;

A = new double*[n]
for ( i = 0; i < n; i++)
   A[i] = new double[N];
```

When we declare a matrix (a two-dimensional array) we must first declare an array of double variables. To each of this variables we assign an allocation of a single-dimensional array. A conceptual picture on how a matrix **A** is stored in memory.

Allocated memory should always be deleted when it is no longer needed. We free memory using the statements

```
for ( i = 0; i < n; i++)
   delete[] A[i];
delete[] A;
```

delete[]A;, which frees an array of pointers to matrix rows.