Passing multidimensional dynamic arrays to C++ functions: Make simple with example

Summary:

Multidimensional arrays are important for many applications especially in signal processing and engineering fields. However, they are a source of headache and loss of generality in C++ programming especially in multi-platform or cross-platform development environments. In this article a light will be shed on the correct way of passing multidimensional arrays to functions in C++. Complexity is avoided and a comprehensive example code is provided for more convenience.

Functions accepting multidimensional array:

Generally, there are two possible forms of function definition that accept multidimensional arrays. In the following discussion a 2Dim arrays will be discussed, but it can easily be extended to any dimension. The first form of function definition may appear like this

```cpp
void myFun1(int A[][2], int nrow)
```

The above function accepts an array of fixed number of columns (2 columns) but dynamic number of rows. That means myFun1 accepts an array with direct indexing. For example when it is required to access \( A[i][j] \), the CPU will do the following:

1. Take the value of pointer \( A \)
2. Add \( i \) to it
3. Add \( 2 \times j \) to it
4. Access memory location at the result

Of course, the function can also be defined as (recommended for clarity):

```cpp
void myFun1(int (*A)[2], int nrow)
```

For example, when passing a 3x2 array to myFun1, the function expects the following array structure:

```
A : int(*)[2]
```

- Element-1 Element-2 first row
- Element-1 Element-2 second row
- Element-1 Element-2 third row

In this case, the array \( A[][2] \) is semi-dynamic because the number of columns must be 2. The array can be defined in any of the following ways:

```cpp
int A[3][2]; //fixed ncol and nrow
```
or:
```c
int (*A)[2]=new int[nrow][2]; // fixed ncol, but dynamic nrow
```

where `nrow` must be defined previously either as a constant or a variable with assigned value.

The other method of defining a function is using the following forms:

```c
void myFun2(int **A, int nrow)
```

In this case, `myFun2` accepts a fully dynamic 2Dim array or an array of pointers in which each pointer points to another pointer that in turn points to a row. The row can be of any size allowing any NxM array to be passed. While `myFun2` exhibit the fully dynamic feature, one must strictly be aware of passing the correct array. In fact, the arrays defined in the first method above cannot be passed to `myFun2` even when using type casting to overcome compiler error. The possible way to define the valid array to be passed to `myFun2` is by using array or pointers to pointers:

```c
int **A=new int*[nrow];
for(i=0;i<nrow;i++) A[i]=new int[ncol]; //dynamic ncol and nrow
```

where both `nrow` and `ncol` must be defined and assigned either statically or dynamically. For example, `myFun2` will expect the following structure when passing 3xM array:

```
A: int**

| int* | Element-1 | Element-2 | --- | --- | first row |
| int* | Element-1 | Element-2 | --- | --- | second row |
| int* | Element-1 | Element-2 | --- | --- | third row |
```

So, when it is required to access element `A[i][j]`, the CPU will do the following:

1. Take the value of pointer `A`
2. Add `i` to it to obtain the memory location holding the required pointer
3. Read the content of the above memory
4. Add `j` to the value read in (3)
5. Access memory location at the result

**Closing Remarks:**

1. If you do not need dynamic number of columns, using the first method is faster, more safe and less cumbersome because it accepts statically defined arrays.
2. If your function need to be more general and accept any dimension, then using the second method is the only choice.
3. If you want to pass statically defined arrays to myFun2, you will need to build a surrogate array of pointers to pointers and pass it to the function.
4. Finally, you must clear any arrays that were allocated by new clause to avoid memory leak.

The following code provides comprehensive example about using both of the above methods. This code is executed under VS2008 using C++ console application. Similar code was executed under Qt 4.8 for Linux.

```cpp
#include "stdafx.h"

void myFun1(int (*A)[2], int nrow){
    int i,j;
    for (i=0;i<nrow;i++)
        for (j=0;j<2;j++)
            printf("A[%i][%i]= %i \n",i,j,A[i][j]);
    printf("\n");
}

void myFun2(int **A, int nrow){
    int i,j;
    for (i=0;i<nrow;i++)
        for (j=0;j<2;j++)
            printf("A[%i][%i]= %i \n",i,j,A[i][j]);
    printf("\n");
}

int _tmain(int argc, _TCHAR* argv[])
{
    int i,j;
    int nrow=3;
    int ncol=2;
    int A[3][2];
    int (*B)[2]=new int[nrow][2];
    //(*B) is a collection of nrow-pointers defined by new clause
    //each element of (*B) points to a row of 2-elements of int
    //if nrow is const, this is exactly equ. to: int B[nrow][2];
    int **C=new int*[nrow];
    for(i=0;i<nrow;i++) C[i]=new int[ncol]; // dynamic ncol and nrow

    for(i=0;i<3;i++)
        for(j=0;j<2;j++)
            A[i][j]=i+j; B[i][j]=i+j; C[i][j]=A[i][j];

    myFun1(A,nrow); //works
    //myFun1(B,nrow); //works
    //myFun1(C,nrow); //works
    myFun2(A,nrow); //compile error
    //myFun2(B,nrow); //compile error
    //myFun2(C,nrow); //compile error
    myFun2((int**)A,nrow); //compile but array flattenning, runtime error
    //myFun2((int**)B,nrow); //compile but array flattenning, runtime error
    //myFun2((int**)C,nrow); //compile and run, but incorrect results
}```
delete[] B;
for(i=0;i<3;i++) delete[] C[i];

while(1);
return 0;
}