**How do I allocate multidimensional arrays using new?**

**(From C++ FAQ)**

There are many ways to do this, depending on how flexible you want the array sizing to be. On one extreme, if you know all the dimensions at compile-time, you can allocate multidimensional arrays statically (as in C):

class Fred { /\**...*\*/ };

void someFunction(Fred& fred);

void manipulateArray()

{

const unsigned nrows = 10; // *Num rows is a compile-time constant*

const unsigned ncols = 20; // *Num columns is a compile-time constant*

Fred matrix[nrows][ncols];

for (unsigned i = 0; i < nrows; ++i) {

for (unsigned j = 0; j < ncols; ++j) {

// *Here's the way you access the (i,j) element:*

someFunction( matrix[i][j] );

// *You can safely "return" without any special delete code:*

if (today == "Tuesday" && moon.isFull())

return; // *Quit early on Tuesdays when the moon is full*

}

}

// *No explicit delete code at the end of the function either*

}

More commonly, the size of the matrix isn't known until run-time but you know that it will be rectangular. In this case you need to use the heap ("freestore"), but at least you are able to allocate all the elements in one freestore chunk.

void manipulateArray(unsigned nrows, unsigned ncols)

{

Fred\* matrix = new Fred[nrows \* ncols];

// *Since we used a simple pointer above, we need to be VERY*

// *careful to avoid skipping over the delete code.*

// *That's why we catch all exceptions:*

try {

// *Here's how to access the (i,j) element:*

for (unsigned i = 0; i < nrows; ++i) {

for (unsigned j = 0; j < ncols; ++j) {

someFunction( matrix[i\*ncols + j] );

}

}

// *If you want to quit early on Tuesdays when the moon is full,*

// *make sure to do the delete along ALL return paths:*

if (today == "Tuesday" && moon.isFull()) {

delete[] matrix;

return;

}

*...code that fiddles with the matrix...*

}

catch (...) {

// *Make sure to do the delete when an exception is thrown:*

delete[] matrix;

throw; // *Re-throw the current exception*

}

// *Make sure to do the delete at the end of the function too:*

delete[] matrix;

}

Finally at the other extreme, you may not even be guaranteed that the matrix is rectangular. For example, if each row could have a different length, you'll need to allocate each row individually. In the following function, ncols[i] is the number of columns in row number i, where i varies between 0 and nrows-1 inclusive.

void manipulateArray(unsigned nrows, unsigned ncols[])

{

typedef Fred\* FredPtr;

// *There will not be a leak if the following throws an exception:*

FredPtr\* matrix = new FredPtr[nrows];

// *Set each element to NULL in case there is an exception later.*

// *(See comments at the top of the try block for rationale.)*

for (unsigned i = 0; i < nrows; ++i)

matrix[i] = NULL;

// *Since we used a simple pointer above, we need to be*

// *VERY careful to avoid skipping over the delete code.*

// *That's why we catch all exceptions:*

try {

// *Next we populate the array. If one of these throws, all*

// *the allocated elements will be deleted (see catch below).*

for (unsigned i = 0; i < nrows; ++i)

matrix[i] = new Fred[ ncols[i] ];

// *Here's how to access the (i,j) element:*

for (unsigned i = 0; i < nrows; ++i) {

for (unsigned j = 0; j < ncols[i]; ++j) {

someFunction( matrix[i][j] );

}

}

// *If you want to quit early on Tuesdays when the moon is full,*

// *make sure to do the delete along ALL return paths:*

if (today == "Tuesday" && moon.isFull()) {

for (unsigned i = nrows; i > 0; --i)

delete[] matrix[i-1];

delete[] matrix;

return;

}

*...code that fiddles with the matrix...*

}

catch (...) {

// *Make sure to do the delete when an exception is thrown:*

// *Note that some of these matrix[...] pointers might be*

// *NULL, but that's okay since it's legal to delete NULL.*

for (unsigned i = nrows; i > 0; --i)

delete[] matrix[i-1];

delete[] matrix;

throw; // *Re-throw the current exception*

}

// *Make sure to do the delete at the end of the function too.*

// *Note that deletion is the opposite order of allocation:*

for (unsigned i = nrows; i > 0; --i)

delete[] matrix[i-1];

delete[] matrix;

}

Note the funny use of matrix[i-1] in the deletion process. This prevents wrap-around of the unsigned value when i goes one step below zero.

Finally, note that [pointers and arrays are evil](http://www.parashift.com/c%2B%2B-faq-lite/arrays-are-evil.html). It is normally much better to encapsulate your pointers in a class that has a safe and simple interface. [The following FAQ](http://www.parashift.com/c%2B%2B-faq-lite/multidim-arrays2.html) shows how to do this.