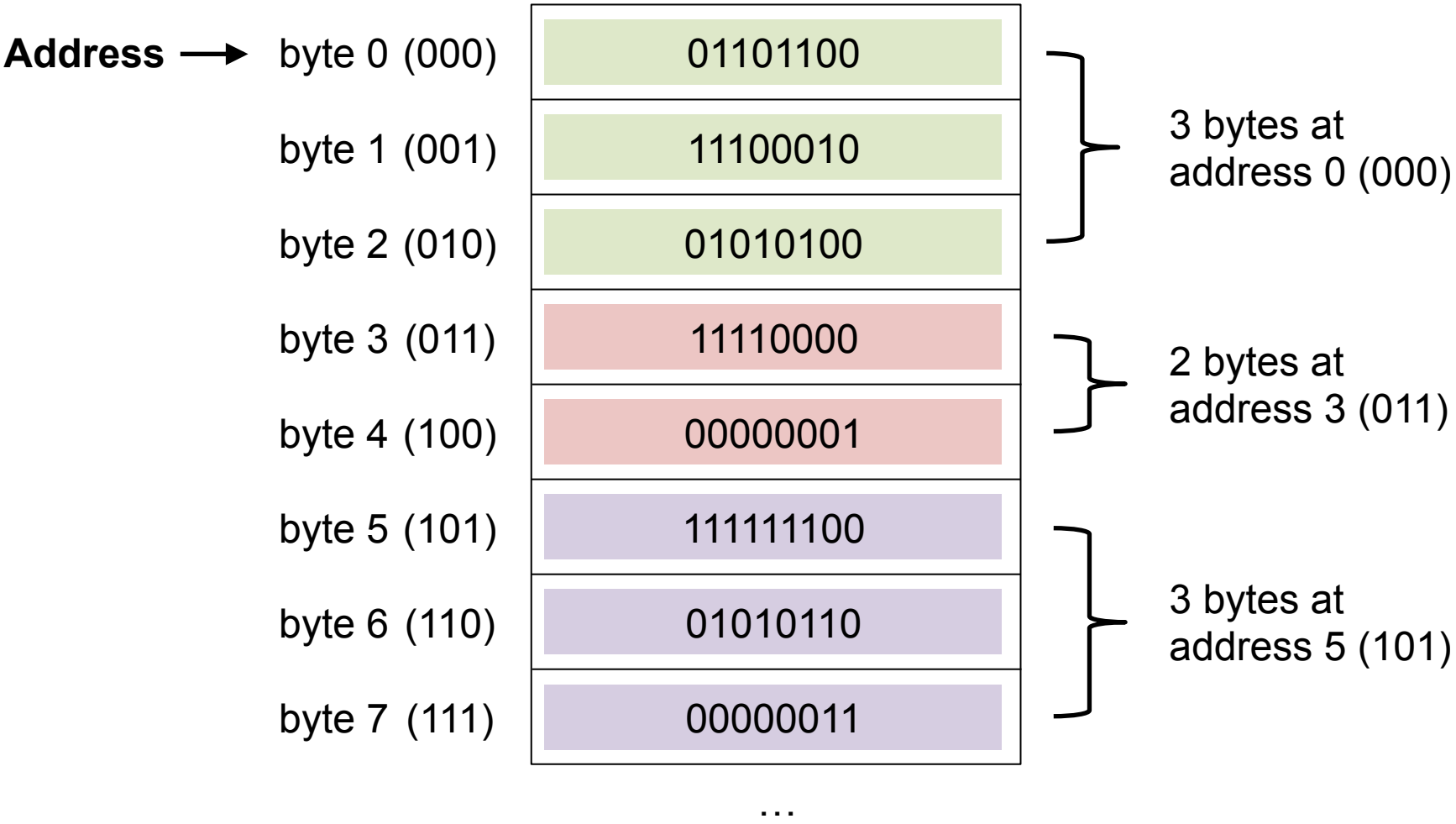
Pointers [and Pals]

Lecture 4

Outline

1. Pointers
2. Dynamic Arrays
3. Multidimensional Arrays

Quick Recap of Memory



What is a Pointer?

A **pointer** is a variable whose value is a *memory address* [of another variable]

– It "points" to objects in memory

```
int *my_pointer;
```

If you output the value of a pointer, you will get a memory address, typically represented in **hexadecimal** (prefixed with 0x; 0-9, A-F)

Reinforcing Humor ala XKCD

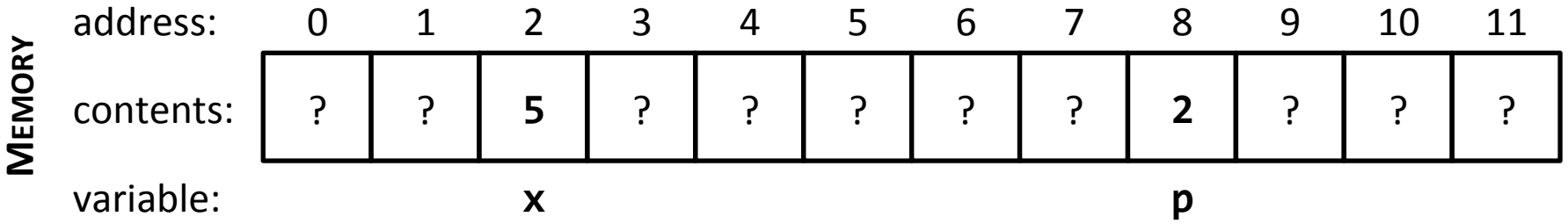
Useful Operators

`&variable` = "address of" variable

`*pointer` = "value at" address

– Termed **dereferencing** the pointer

Pointer Representation




```
int x;
x = 5;
cout << x;
cout << &x;
```

5

2

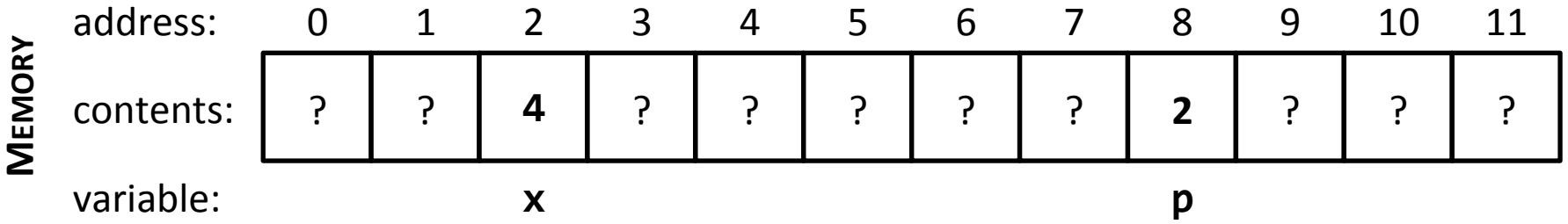
```
int *p;
p = &x;
cout << p;
cout << &p;
cout << *p;
```

2

8

5

Pointer Representation




```
x = 7;
cout << x;      7
cout << &x;     2
cout << p;      2
cout << &p;     8
cout << *p;     7
```

```
*p = 4;
cout << x;      4
cout << &x;     2
cout << p;      2
cout << &p;     8
cout << *p;     4
```

The `new` and `delete` Operators

- The new operator produces a new, nameless variable and returns a pointer to this new variable

```
int *p = new int;
```

- Variables created this way are termed **dynamic** (others are *automatic*/*ordinary*)

- Any variable created this way must be later de-allocated via **`delete`** (variables not created this way must *not* be deallocated using **`delete`**)

```
delete p;
```

- Do not call **`delete`** on memory that has already been deallocated

Memory Leak

Anytime you lose track of memory you allocated, it can no longer be used by your or other programs

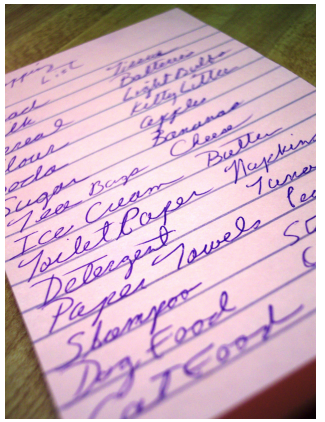
Example

```
int *p = new int;
p = new int;
```

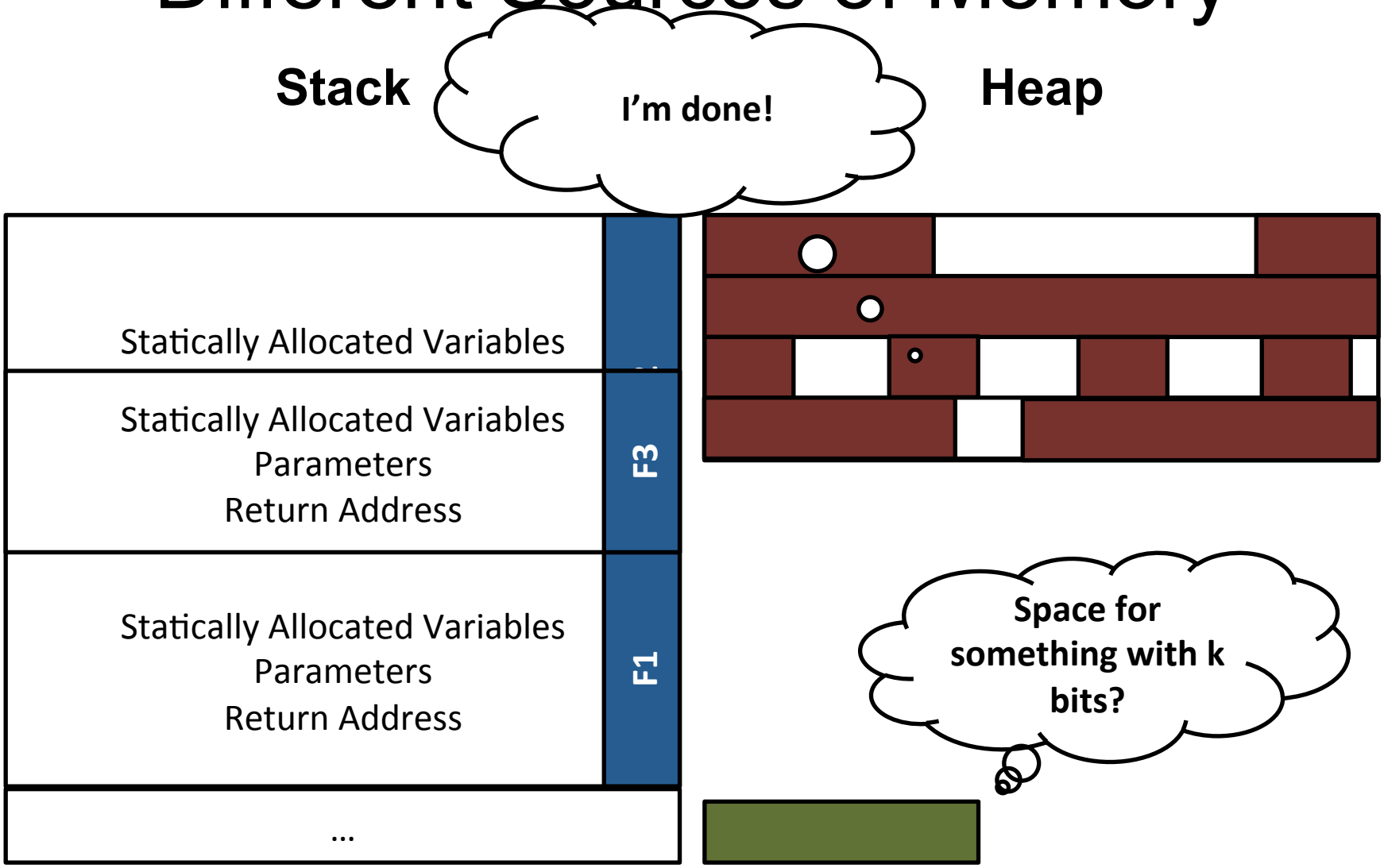
Why Dynamic Allocation?

- Statically allocated variables are automatically managed by C++ via the **stack**
  - Memory [de]allocation is fast!
  - Memory is released once the function is done

- But sometimes you don't know how much memory you are going to need until the program runs!
  - Typical with user interaction/external data
  - Memory allocated via the **heap** and remains until released



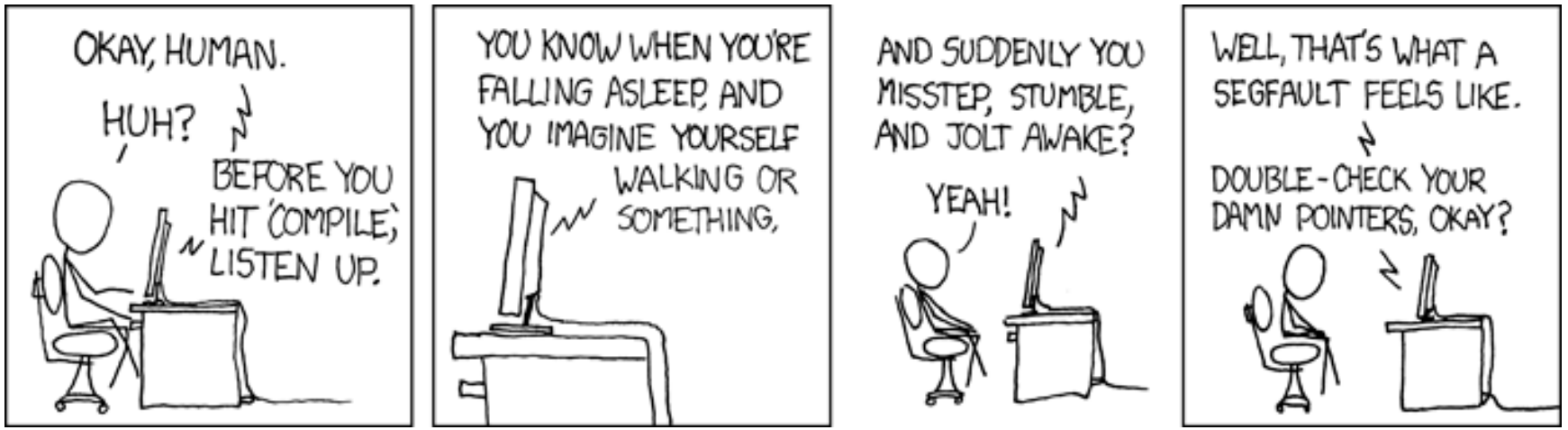
Different Sources of Memory



NULL

- The memory address `0` is a special address and never indicates valid memory
- Many libraries (e.g. `cstdlib`, `cstdio`) provide the constant `NULL` for this address
  - This use of NULL is a memory address, and shouldn't be confused with the NULL character for C strings
- Dereferencing NULL always results in a **segmentation fault** (segfault)

More Reinforcing Humor ala XKCD



Pointer-Array Duality

- In C++, an array variable is actually a pointer that points to the first indexed variable of the array

- Each array element has a unique address
  - `&arr[0]` for element `a[0]`
  - `&arr[1]` for element `a[1]`
  - …

- But C++ has a shortcut
  - `a` is the same as `&a[0]`
  - `(a+1)` is the same as `&a[1]`
  - …

Array Element Values

The bracket operators (`[]`) are a shortcut for dereferencing array element values

- `a[0]` for `*(a)`
- `a[1]` for `*(a+1)`
- `a[2]` for `*(a+2)`

…

Arrays as Parameters

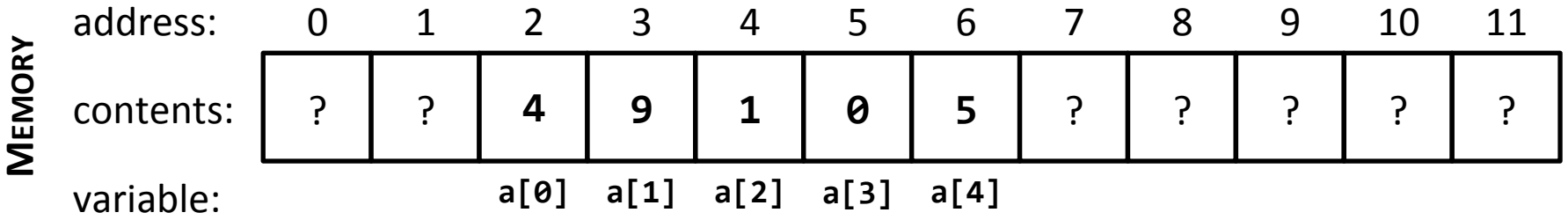
- Formerly we learned one way to pass arrays as arguments

```
void func(int arr[], int size);
```

- You can also use pointer notation

```
void func(int *arr, int size);
```

Exercise




```
int a[] = {4,8,1,0,5};
cout << a;          2
cout << (a+3);      5
cout << *(a+4);     5
```

```
*(a+1) = 9;
cout << a[1];       9
cout << (a+1);      3
cout << *(a+1);     9
```

Dynamic Arrays

- So far we've only been able to create arrays of a fixed size
  - Could be too small or too large for a given problem

- The **`new`** operator allows you to create a dynamically sized array

```
int x;
cin >> x;
int *p = new int[x];
```

Deallocating Dynamic Arrays

- Use the **`delete []`** operator to release a dynamic array

```
delete [] p;
```

- Forgetting the `[]` only releases the first element, resulting in a memory leak

- Do not use the `[]` on an address that was not allocated as an array

- Do not use **`delete []`** with statically allocated arrays

Exercise

Ask the user for an integer from the keyboard. Create an array of that size and populate it with the numbers 2(i+1), where i is the array index. Then output the contents of the array to the screen and deallocate the dynamic memory. Do NOT use the brackets ([ ]) to get/set array values.

Answer

```
#include <iostream>
using namespace std;

int main()
{
     int size;
     int *x;

     cin >> size;
     x = new int[ size ];

     for ( int i=0; i<size; i++ )
          *( x + i ) = ( 2 * ( i + 1 ) );

     for ( int i=0; i<size; i++ )
          cout << *( x + i ) << endl;

     delete [] x;

     return 0;
}
```

Multidimensional Arrays

- C++ allows you to have an array with more than one index

```
int arr[ x ][ y ];
```

- In the example above, **`arr`** is an array of size **`x`**, where each element is an array of **`y`** integers

Example

```
#include <iostream>
using namespace std;

int main()
{
     double arr[3][4] = {
          { 0.0, 0.1, 0.2, 0.3 },
          { 1.0, 1.1, 1.2, 1.3 },
          { 2.0, 2.1, 2.2, 2.3 }
     };

     for ( int row=0; row<3; row++ )
     {
          for ( int col=0; col<4; col++ )
          {
               cout << "row=" << row
                    << ", col=" << col
                    << ": " << arr[ row ][ col ]
                    << endl;
          }
     }

     return 0;
}
```

```
row=0, col=0: 0
row=0, col=1: 0.1
row=0, col=2: 0.2
row=0, col=3: 0.3
row=1, col=0: 1
row=1, col=1: 1.1
row=1, col=2: 1.2
row=1, col=3: 1.3
row=2, col=0: 2
row=2, col=1: 2.1
row=2, col=2: 2.2
row=2, col=3: 2.3
```

Multidimensional Array Parameters

When passing multidimensional arrays as arguments, you must leave the first set of brackets empty (as with regular arrays), but provide sizes for each additional dimension

```
void func(int arr[][100], int s1, int s2);
```

Example (1)

```
#include <iostream>
using namespace std;

void output(int arr[][2], int rows, int cols);
void m_add(int arr1[][2], int arr2[][2], int r[][2], int rows, int cols);

int main()
{
    int m1[2][2] = { { 1, 2 }, { 3, 4 } };
    int m2[2][2] = { { 5, 5 }, { 6, 6 } };
    int m3[2][2] = { { 0, 0 }, { 0, 0 } };

    output( m1, 2, 2 );
    cout << "PLUS" << endl;
    output( m2, 2, 2 );
    cout << "EQUALS" << endl;
    m_add( m1, m2, m3, 2, 2 );
    output( m3, 2, 2 );

    return 0;
}
```

Example (2)

```
void m_add(int arr1[][2], int arr2[][2], int r[][2], int rows, int cols)
{
     for ( int i=0; i<rows; i++ )
          for ( int j=0; j<cols; j++ )
               r[i][j] = arr1[i][j] + arr2[i][j];
}

void output(int arr[][2], int rows, int cols)
{
     for ( int i=0; i<rows; i++ )
     {
          for ( int j=0; j<cols; j++ )
               cout << arr[i][j] << " ";

          cout << endl;
     }
}
```

Multidimensional Dynamic Arrays

- To create multidimensional arrays, first allocate memory for the first dimension, then loop for the second (and additional nested looping for any additional dimensions)
  - Don't forget to **`delete []`** at each level when you are done!

- Function arguments can be all *'s or one can be left as []

  `int *arr[]` same as **`int **arr`**

  `int **arr[]` same as **`int ***arr`**

Revisiting Program Arguments

- You can now reinterpret the arguments to the main function

```
int main(int argc, char *argv[]);
```

- This can be equivalently written with only pointers

```
int main(int argc, char **argv);
```

- Why is there not a second size argument?

Example

```
#include <iostream>
using namespace std;

void output(int **arr, int rows, int cols);

int main()
{
	int rows, cols;
	cin >> rows >> cols;

	int **m = new int*[ rows ];

	for ( int i=0; i<rows; i++ )
	{
		m[i] = new int[ cols ];
		for ( int j=0; j<cols; j++ )
		{
			cin >> m[i][j];
		}
	}
	output( m, rows, cols );

	for ( int i=0; i<rows; i++ )
		delete [] m[i];
	delete [] m;

	return 0;
}
```

```
void output(int **arr, int rows, int cols)
{
	for ( int i=0; i<rows; i++ )
	{
		for ( int j=0; j<cols; j++ )
			cout << arr[i][j] << " ";

		cout << endl;
	}
}
```

```
> 2 2
> 1 2 3 4
1 2
3 4
```

Putting It All Together!

Make a matrix addition program. At the command line, require two integer arguments (first=# rows, second=# cols). Now input two matrices with these dimensions and store them in 2D arrays. Finally, add them together and output the result. Example:

```
> ./a 2 2
1 2 3 4
5 5 9 8
1 2
3 4
PLUS
5 5
9 8
EQUALS
6 7
12 12
```

Answer (1)

```
#include <iostream>
#include <cstdlib>
using namespace std;

void m_add(int **m1, int **m2, int **r, int rows, int cols)
{
     for ( int i=0; i<rows; i++ )
          for ( int j=0; j<cols; j++ )
               r[i][j] = m1[i][j] + m2[i][j];
}

void output(int **m, int rows, int cols)
{
     for ( int i=0; i<rows; i++ )
     {
          for ( int j=0; j<cols; j++ )
               cout << m[i][j] << " ";

          cout << endl;
     }

}
```

Answer (2)

```
int main(int argc, char **argv)
{
	if ( argc != 3 )
	{
		cout << "Usage: " << argv[0] << " <rows> <cols>" << endl;
		return 0;
	}

	int rows = atoi( argv[1] );
	int cols = atoi( argv[2] );
	if ( ( rows < 1 ) || ( cols < 1 ) )
	{
		cout << "Usage: " << argv[0] << " <rows> <cols>" << endl;
		return 0;
	}

	int **m[3];
	for ( int i=0; i<3; i++ )
	{
		m[i] = new int*[ rows ];

		for ( int j=0; j<rows; j++ )
		{
			m[i][j] = new int[ cols ];

			if ( i != 2 )
				for ( int k=0; k<cols; k++ )
					cin >> m[i][j][k];
		}
	}
```

Answer (3)

```
        m_add( m[0], m[1], m[2], rows, cols );
        output( m[0], rows, cols );
        cout << "PLUS" << endl;
        output( m[1], rows, cols );
        cout << "EQUALS" << endl;
        output( m[2], rows, cols );

        for ( int i=0; i<3; i++ )
        {
                for ( int j=0; j<rows; j++ )
                        delete [] m[i][j];

                delete [] m[i];
        }

        return 0;
}
```

Wrap Up

- Pointers are variables that store memory addresses [of variables]
- The **`new`** operator dynamically allocates memory from the heap, **`delete`** releases that memory
- Array variables are actually pointers with some useful shortcuts
- Dynamically sized arrays can be created via **`new`**, released via **`delete []`**
- Multidimensional arrays can be accessed via multiple indexes via brackets