Current Assignments

- Start Reading Chapter 6
- Project 3 – Due Thursday, July 24

  Contact List Program
- Homework 6 – Due Sunday, July 20

  First part easy true/false questions about arrays.

  Second part implement a stack and a queue using an array.

Today

- Argv and Argc
- String conversion
- The Heap
- The new and delete commands
- Dynamic Arrays
- Structs

Argc and Argv

- Programs often take arguments just like functions.
- For example in unix you might type:

  cp file1.cpp file2.cpp
- The program takes the arguments file1.cpp and file2.cpp and does something with those arguments.
- It copies the file with name file1.cpp to file2.cpp

Argc and Argv

- Programs can take an unlimited number of arguments all of type string.
- The arguments are passed into the program via argc and argv
- Argv is an array of strings (the argument values)
- Argc is a integer and is the number of argument the user gave your program

Argc and Argv

```
#include <iostream>

using namespace std;

int main( int argc, char* argv[] )
{

    cout << "Argument list:" << endl;

    for ( int i = 0; i < argc; i++ )
    {

        cout << argv[i] << " " << endl;
    }

    return 0;
}
```

Input:
program1 argument1 2 3

Output:
Argument list:
program1
argument1
2
3

Argc and Argv

- Since program arguments are always strings we often have to do some work to extract the value we want from an argument.
- There are a number functions for converting strings into other types.
- Eg. atoi(*string*) converts the string into an integer.

Argc and Argv

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main( int argc, char* argv[] )
{
        int x = 0, y = 0;

        x = atoi( argv[1] );
        y = atoi( argv[2] );

        cout << x*y << endl;

        return 0;
}
```

Input:
program2 4 3

Output:
12

String conversion functions

Other string conversion functions:

long int atol(*char** string);

double strtod(*char** start, *char** end);

long int strtol(*char** start, *char** end);

unsigned long int strtoul(*char** start, *char** end, *int* base);

Page 1026 of Deitel and Deitel.

The Stack

- Until now all the memory locations we have used for our data existed on the stack.
- Stack memory is fixed when the program is compiled.
- We can't dynamically get more stack memory after the program starts.
- This is a problem.

The Heap

- If we don't know how much space we will need before the program starts we are in trouble.
- … but we can ask for memory on the heap even after the program starts.

The Heap

- When we ask for memory from the heap we are responsible for managing that memory.
- Before, the memory we needed to store data was created for us when we declared a variable
- And destroyed for us when the end of that variables scope was reached.
- We have to manually create and destroy the memory we use on the heap.

The New and Delete commands

- We ask for memory on the heap with the *new* command (in old C it was *malloc*).
- We return memory to the heap with the *delete* command (in old C it was *free*).
- Syntax for *new* and *delete*:

*type** variable_name = new *type*;

delete variable_name;

or

delete [] variable_name; // If deleting an array

New and Delete

```
int main()
{
    int* x = new int(0); // Initialize variable to 0
    float* y = new float(6.0);
    char* z = new char('z');

    cout << *x << " " << *y << " " << *z << endl;
    *x = 7;
    *y = 10.0;
    *z = 'k';

    cout << *x << " " << *y << " " << *z << endl;

    delete x; delete y; delete z;

    return 0;
}
```

Output:

0 6 z

7 10 k

New and Delete – Memory Leaks

- If the program keeps allocating memory and doesn't return it all, eventually all the memory in the system will be used up.
- For example:

```
for( int i = 0; i < size; i++ )  // the memory reserved for x is returned to the
{                                // stack when x goes out of scope "}"
        int x = 5;
}

for( int i = 0; i < size; i++ ) // The pointer goes out of scope but the memory
{                               // it pointed too is still reserved
        int* x = new int(5);    // But now we have lost its address, there is no
}                               // way to find it again – it is orphaned memory
```

- If this occurs enough times the system will crash for lack of unallocated memory.

- During GWI the patriot missile systems had to be shutdown and restarted every few hours because their control system had a memory leak.

Dynamic Arrays

```
int main()
{
    int size = 0;

    cin >> size;

    int array[size] = {0};

    for ( int i = 0; i < size; i++ )
    {
        array[i] = i;
        cout << array[i] << " ";
    }

    return 0;
}
```

Output:
error C2057: expected constant expression

Dynamic Arrays

```
int main()
{
    int size = 0;

    cin >> size;

    int* array = new int[size];

    for ( int i = 0; i < size; i++ )
    {
        array[i] = i;
        cout << array[i] << " ";
    }
    delete [] array;
    return 0;
}
```

Output:

10

0 1 2 3 4 5 6 7 8 9

Multidimensional Dynamic Arrays

- We can create multidimensional dynamic arrays too.
- They consist of dynamic arrays of pointers to other dynamic arrays.
- They are not quite like multidimensional static arrays since all the memory locations are not guaranteed to be contiguous.
- But they look the same from our point of view though they might be slightly slower under certain circumstances.

Multidimensional Dynamic Arrays

```
int main()
{

    const int d1size = 10, d2size = 10;

    int array[d1size][d2size];

    for ( int i = 0; i < d1size; i++ )
    {
        for ( int j = 0; j <d2size; j++ )
        {
            array[i][j] = (i+1)*(j+1);
            cout << array[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Multidimensional Dynamic Arrays

```
int main()
{
    int d1size = 0, d2size = 0;
    cin >> d1size >> d2size;
    int** array = new int*[d1size];

    for ( int k = 0; k < d1size; k++ )
    {
        array[ k ] = new int[d2size];
    }

    for ( int i = 0; i < d1size; i++ )
    {
        for ( int j = 0; j < d2size; j++ )
        {
            array[i][j] = (i+1)*(j+1);
            cout << array[i][j] << " ";
        }
        cout << endl;
    }
```

Input:
4 5

Output:
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20

Multidimensional Dynamic Arrays

```
    // Clean up
    for ( int k = 0; k < d1size; k++ )
    {
            delete [] array[k];
    }

    delete [] array;

    return 0;
}
```

Input:
4 5

Output:
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20