Pointers Contd.

Causing a Memory Leak

```c
int *ptr = new int;
*ptr = 8;
int *ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;
```

Leaving a Dangling Pointer

```c
int *ptr = new int;
*ptr = 8;
int *ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;
delete ptr2;
ptr2 = NULL;
```
Dynamic Array Example

```cpp
sortFromInput( ) {
  float *array;
  int size, idx;
  cin >> size;
  array = new float[size];
  for (idx=0; idx<size; idx++)
    cin >> array[idx];
  Sort(array, size);
  OutputSortedArray(array, size);
  delete[] array;
}
```

Pointer Arithmetic

- Int *p:
  - p = new int[10]
  - p[i]: 0 <= i <= 9
  - What are *p, *p++ etc..

Algorithm Efficiency and Sorting Algorithms
Measuring the Efficiency of Algorithms

- **Analysis of algorithms** is the area of computer science that provides tools for contrasting the efficiency of different methods of solution.
  - Concerns with significant differences

How To Do Comparison?

- Implement the algorithms in C++ and run the programs
  - How are the algorithms coded?
  - What computer should you use?
  - What data should the programs use?
- Analyze algorithms independent of implementations

The Execution Time of Algorithms

- Count the number of basic operations of an algorithm
  - Read (get), write (put), compare, assignment, jump, arithmetic operations (increment, decrement, add, subtract, multiply, divide), shift, open, close, logical operations (not/complement, AND, OR, XOR), ...
The Execution Time of Algorithms

- Counting an algorithm’s operations

```
int sum = item[0];
int j = 1;
while (j < n) {
    sum += item[j];
    ++j;
}
```

- 1 assignment
- 1 assignment
- n comparisons
- n-1 plus/assignments
- n-1 plus/assignments

Total: 3n operations

Algorithm Growth Rates

- Measure an algorithm’s time requirement as a function of the problem size
  - Number of elements in an array

  Algorithm A requires \(\frac{n^2}{5}\) time units
  Algorithm B requires 5n time units

- Algorithm efficiency is a concern for large problems only

Common Growth-Rate Functions - 1
Common Growth-Rate Functions - II

Big O Notation

Algorithm A is order $f(n)$-denoted $O(f(n))$-if constants $k$ and $n_0$ exist such that A requires $\leq k \cdot f(n)$ time units to solve a problem of size $n \geq n_0$.

- $n^2/5$
  - $O(n^2)$: $k=1/5$, $n_0=0$
- $5^n$
  - $O(n)$: $k=5$, $n_0=0$

More Examples

- How about $n^2-3n+10$?
  - $O(n^2)$ if there exist $k$ and $n_0$ such that
    * $kn^2 \geq n^2-3n+10$ for all $n \geq n_0$
    * $3n^2 \geq n^2-3n+10$ for all $n \geq 2$; so $k=3$, $n_0=2$
Properties of big-Oh

- Ignore low-order terms
  - \( O(n^3+4n^2+3n) = O(n^3) \)
- Ignore multiplicative constant
  - \( O(5n^3) = O(n^3) \)
- Combine growth-rate functions
  - \( O(f(n)) + O(g(n)) = O(f(n)+g(n)) \)

Worst-case vs. Average-case Analyses

- An algorithm can require different times to solve different problems of the same size.
- Worst-case analysis (find the maximum number of operations an algorithm can execute in all situations)
  - is easier to calculate and is more common
- Average-case (enumerate all possible situations, find the time of each of the \( m \) possible cases, total and divide by \( m \))
  - is harder to compute but yields a more realistic expected behavior