# Computer Programming: Skills & Concepts (CP1)
# Sorting II

4th November 2010



## Tuesday's lecture

- BubbleSort algorithm (from slides18.pdf).
- (on board) of running-time of BubbleSort.
- The `merge` function (for two sorted sub-arrays).

Due to time constraints, we did NOT finish the slides for Lecture 19 (MergeSort) . . . we finish these today.



## Today's lecture

- Review of `merge` function.
- The MergeSort Algorithm.
- Running time of MergeSort.
- Two features used in `mergesort`:
  - `calloc` for dynamically-sized arrays.
  - `++` expressions for incrementing.



## Trial run of `mergesort`

```
int main(void) {
  int i, sz, key[] = {4, 3, 1, 67, 0, 4, -5, 37, 7, 2, -1, 199};
  sz = sizeof(key)/sizeof(int);
  printf("Before mergesort: \n");
  wrt(key, sz);
  printf("\n");
  mergesort(key, sz);
  printf("After mergesort:\n");
  wrt(key, sz);
  return EXIT_SUCCESS;
}
```

## Results of Trial run

```
[fletcher]mcryan: ./a.out
Before mergesort:
   4   3   1  67   0   4  -5  37   7   2  -1 199

   3   4   1  67   0   4  -5  37   2   7  -1 199
   1   3   4  67  -5   0   4  37  -1   2   7 199
  -5   0   1   3   4   4  37  67  -1   2   7 199
  -5  -1   0   1   2   3   4   4   7  37  67 199
After mergesort:
  -5  -1   0   1   2   3   4   4   7  37  67 199
```

- 1st step: all length-2 blocks sorted;
- 2nd step: all (three) length-4 blocks sorted;
- 3rd step: block of length-8 sorted, end-block (length-4) unchanged;
- 4th step: length-8 block merged with the end-block.



## Features of `mergesort` implementation

A CHALLENGING PROGRAM

- Implemented in a "bottom-up" fashion (more standard implementation is via *recursion*).
- Uses the `calloc` function to *dynamically* allocate memory of a variable size.
- Uses the ++ operator for incrementing *inside* another expression ⇒ *complicated meaning*



## `calloc`

*Usually*, when defining arrays, we must specify the length of the array as a fixed value chosen in advance (when writing the program).

To define array size *dynamically*, use `calloc`:

- `calloc()` takes 2 arguments (of type `size_t`):

  `calloc(n,`*el_size*`)`

- This allocates (IF available) space for an array of length `n` of type *el* (each cell using *el_size* bytes).
  - `calloc` returns a pointer to the address of the start of the array in memory (assuming space is available)
  - If that space is NOT available, `calloc` returns a `NULL` pointer.
- Space created is initialized to all-bits-0.



## Examples of `calloc`

Testing our sorting program on arrays of varying lengths:

```
int i, sz, *key;
double start, stop, t;
printf("Input desired size of array: ");
scanf("%d", &sz);
printf("\n");
key = calloc(sz, sizeof(int));   /* Make array of this size */
if (key != NULL) {               /* check there was space */
  for(i = 0; i < sz; i++)      /* Fill array:
    key[i] = rand() % 1000;     * rand() returns 1 random int */
  start = (double)clock();
  mergesort(key, sz);
  stop = (double)clock();
  t = (stop-start)/CLOCKS_PER_SEC;
  printf("Time on array of length %d was %f sec.\n", sz, t);
}
```

## Incrementing/decrementing with ++

4 ways to increment a variable:

```
x = x+1;   x += 1;   ++x;   x++;
```

4 ways to decrement a variable:

```
x = x-1;   x -= 1;   --x;   x--;
```

These commands/expressions can appear *within* other expressions - the *semantics* (meaning/interpretation) is quite interesting in these cases.



## Side-effects

++x("pre-increment"):
Add 1, *then* return the result to the expression ++x; is appearing in.

```
int x = 10;
printf("%d\n", ++x);
```

will print 11 to standard output (here "the expression ++x is appearing in is ++x itself).

x++("post-increment"):
Return value of x to the expression ++x; appears in, then add 1 to x.

```
int x = 10;
printf("%d\n", x++);
```

will print 10 to standard output.



## Use of ++ in merge

```
while (i < m && j < n) {
  if (a[i] <= b[j])
    c[k++] = a[i++];
  else
    c[k++] = b[j++];
}
```

is equivalent to

```
while (i < m && j < n) {
  if (a[i] <= b[j]) {
    c[k] = a[i];
    i++; k++;
  }
  else {
    c[k] = b[j];
    j++; k++;
  }
}
```



## Homework

- Sections 6.8 and 6.9 of Kelley and Pohl (for sorting)
- Section 2.10 of Kelley and Pohl (for increment/decrement)
- Experiment with the code.
  - Run `mergesort.c` for arrays of length 50000, 100000, 200000, ... to see effect of size.
  - Add the code-fragment for dynamically creating arrays to `bubblesort.c` and test this on arrays of varying sizes.
  - Compare results for MergeSort against BubbleSort.