

---

\$Id: lab8c-voidstar-generic.mm,v 1.1 2014-05-19 16:04:05-07 - - \$

PWD: /afs/cats.ucsc.edu/courses/cmcs012b-wm/Labs-cmcs012m/lab8c-voidstar-generic

URL: http://www2.ucsc.edu/courses/cmcs012b-wm/:/Labs-cmcs012m/lab8c-voidstar-generic/

---

## 1. Overview

In this lab, you will implement a generic sorting routine using the `void*` parameter declaration. This is similar to the C library function `qsort(3)`. You will also review your knowledge of **Makefiles** and header files. Begin by studying the example programs in `wk09a-cqsort/`. Also study `misc/voidstar.c`.

## 2. Programs to write

Write the following programs and files, each as described here :

### Makefile

Write a **Makefile** with the following targets, and in each case, provide the appropriate actions.

```
all:          should build the two binaries numsort and linesort.

numsort:      depends on numsort.o and inssort.o.

linesort:     depends on linesort.o and inssort.o.

%.o:          depends on %.c. All C compilations should be done with the command
               gcc -g -O0 -Wall -Wextra -std=gnu11
               Note specifically the use of the -c and -o options from previous
               Makefiles.

ci:           depends on all source files and runs both ci and checksource.

submit:       depends on source files and submits them.
```

### numsort.c

This utility reads in `double` numbers from `stdin`, sorts them, then prints them.

- (i) Write a program which will create an array `double array[0x1000]` and use `scanf` to read numbers into this array.
- (ii) It stops reading when the first of the following happens: end of file, any invalid input not recognized by `scanf`, or the array is full.
- (iii) The numbers are then passed to the function `inssort`, along with a suitable comparison function. The numbers are sorted in increasing order.
- (iv) The numbers are then printed one per line using the format `"%20.15g\n"`.

### linesort.c

This utility reads in lines from `stdin` into an array, sorts them, then prints them.

- (i) Allocate an array of `0x1000` pointers to character strings, read in each character string from `stdin` and `strdup` each line into the array. Plug the newline at the end of each line with a `'\0'`, but don't error out if there is no newline. Use `char buffer[0x1000]` as an input buffer. The program stops at end of file, or when the array is full.

- (ii) It then calls `inssort` to sort the strings using a suitable comparison function. The lines are sorted into increasing lexicographic order.
- (iii) The lines are then printed, one per line of output.

**inssort.h**

This file is the header file to be included by both `numsort.c` and `linesort.c` and it is important that both of these programs call the same function. Do not write a separate `double` sorter and a separate `char*` sorter. Using proper style, provide file guards and necessary `#includes` to prototype the following function:

```
void inssort (void* base, size_t nelem, size_t size,
             int (*compar) (const void*, const void*));
```

The parameters are as follows:

- (i) `base` is the base address of the array,
- (ii) `nelem` is the number of elements (length) of the array,
- (iii) `size` is the number of bytes used by a single array element, and
- (iv) `compar` is a comparison function which produces the usual results, i.e., a negative number if the first argument is less than the second, zero if equal, and a positive number if greater.

**inssort.c**

Before beginning your program, you may wish to use the library function `qsort(3)` to debug your main programs, but be sure to delete all references to `qsort` before submitting your program.

- (i) Your program should be a direct line-for-line translation of the Java function `insertion_sort`:

```
static <elem_t extends Comparable <? super elem_t>>
void insertion_sort (elem_t[] array, int nelem) {
    for (int sorted = 1; sorted < nelem; ++sorted) {
        int slot = sorted;
        elem_t copy = array[slot];
        for (; slot > 0; --slot) {
            int cmp = copy.compareTo (array[slot - 1]);
            if (cmp > 0) break;
            array[slot] = array[slot - 1];
        }
        array[slot] = copy;
    }
}
```

- (ii) Inside the function, you must use byte offsets from the base of the array in order to compute data movements.
- (iii) Cast addresses from `void*` to `char*` in order to do address arithmetic. An array element `i` is at location `base + i * size`.

- (iv) Pass the address of each pair of elements to the comparison function. The comparison function accepts addresses of elements, not elements themselves.
- (v) Use the function `memcpy(3)` to copy parts of the array from one location in memory to another.
- (vi) To allocate space for the temporary `element` variable, use `malloc(3)`. Don't forget to `free(3)` this temporary before returning from the function.

### 3. Eliminate all warnings and submit

Eliminate all warnings that `gcc` with the `-Wall` and `-Wextra` options may produce, ensure `checksource` does not complain, and eliminate all messages from `valgrind --leak-check=full`.

Submit `README`, `Makefile`, `numsort.c`, `linesort.c`, `inssort.h`, `inssort.c`. Also, if you are doing pair programming, submit the required pair programming files.