$Id: lab9c-voidstar-generic.mm,v 1.1 2012-03-08 18:57:51-08 - - $
/afs/cats.ucsc.edu/courses/cmps012b-wm/Labs-cmps012m/lab9c-voidstar-generic

1. Overview

In this lab, you will impement a generic sorting routing using the **`void*`** parameter declaration. This is similar to the C library function **`qsort`**(3). You will also review your knowledge of **`Makefile`**s and header files. Begin by studying the example programs **`cqsort-int.c`** and **`cqsort-string.c`**(**`Examples/wk08-lec22-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`**.
`ci`	depends on all source files and runs both **`ci`** and **`checksource`**
`submit`	depends on source files and submits them.
`lint`	runs lint on the source code.

`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[1000]`** 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 1000 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[1000]`** 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 **`#include`**s 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: **`base`** is the base address of the array, **`nelem`** is the number of elements (length) of the array, **`size`** is the number of bytes used by a single array element, and

```
// 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;
   }
}
```

Figure 1. Java function **`insertion_sort`**

`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 of 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`**, as shown in Figure 1.

(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

In this lab, eliminiate ***all*** warnings from both **`gcc`** and **`lint`**. For **`lint`**, the following should not produce any output at all:

```
lint -Xa -fd -m -u -x -errchk=%all numsort.c inssort.c
lint -Xa -fd -m -u -x -errchk=%all linesort.c inssort.c
```

Also, use the following options when running **`gcc`**:

```
gcc -g -O0 -Wall -Wextra -std=gnu99
```

4. What to submit

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