CMSC 202	**Project 2**	Spring 1996
Sections 101, 102 & 103	**due: Tuesday, 23 April**	Handout #21

This assignment will give you more practice implementing an abstract data type (ADT). It will also expose you to the use of command–line arguments (*cf.* Kelley and Pohl text, page 258). The ADT you implement will be based on work you did in Homework 4. In Homework 4, you read *tuples* (fixed–length sequences of objects of different types) from a file. Each tuple was composed of an integer, a real, and a string. Your job was to store the tuples in an appropriate data structure (a *list* in one case, an *array* in the other) in sorted order. The order was determined by a *comparator*.

One major difference between the two data structures is that a list in principle has no size limit while an array's size is fixed. The data file contained information on the number of tuples to be stored; this information was not needed for the list implementation, but was crucial for allocating the array. In this project, you will generalize the work of Homework 4—You will implement the *Whamo* Abstract Data Type.

The Whamo ADT

Recall that an abstract data type specification has two parts: a description of the elements making up the type, and a description of the operations that may be performed on those elements and any constants that need to be provided. An ADT specification *never, ever, no–not–even–sometimes* mentions an implementation. For example, the *list* ADT might be specified as:

1. a homogeneous sequence of elements, with no limit on the length of the sequence; and
2. the operations `cons`, `first`, `rest`, and `empty_list`, and the constant `the_empty_list`.

Here is a specification for the *Whamo* ADT:

1. a homogeneous collection of elements, with no limit on the number of elements, together with a comparator between pairs of elements; and
2. the operations:
   - `whamo_type makeWhamo(comparator comp)` creates and returns a new empty `whamo_type` that has `comp` as its comparator. The type `comparator` is defined to be:

     `int (*comparator)(const void *, const void *)`
   - `void insertWhamo(whamo_item_type item, whamo_type whamo)` inserts `item` into `whamo`.
   - `whamo_item_type getfirstWhamo(whamo_type whamo)` returns the item in `whamo` that comes first on the basis of `whamo`'s comparator. The item is not removed from `whamo`.

- `whamo_item_type getlastWhamo(whamo_type whamo)` returns the item in `whamo` that comes last on the basis of `whamo`'s comparator. The item is not removed from `whamo`.
- `void transformWhamo(whamo_type whamo, transformer trans)` replaces each item in `whamo` with the result of applying `trans` to it. The type `transformer` is defined to be:

  ```
  whamo_item_type (*transformer)(whamo_item_type)
  ```
- `void printWhamo(FILE *fp, whamo_type whamo, printfunc print)` prints (to `fp`) all the items in the Whamo, in the order determined by the Whamo's comparator. Each item in the Whamo is printed by applying `print` to it. The type `printfunc` is defined to be:

  ```
  void (*printfunc)(FILE *, whamo_item_type)
  ```

These types and operations are declared in the interface file `whamo.h`, which can be found in the `lib202` directory in `~mayfield`. You should compile those files that include `whamo.h` using the `-I ~mayfield/lib202` flag.

Notice that there is no mention of an implementation in the Whamo specification. You may have begun to think about how you would implement a Whamo. Certainly it would be reasonable to consider a list implementation, because a Whamo is unlimited in capacity. On the other hand, an array implementation would also be possible; the implementation would simply create a larger array whenever one was needed. We'll take for granted that the list implementation is doable. Just to show that this ADT can be implemented in many different ways, we will implement the Whamo ADT using an array.

Assignment

There are two pieces to this project that will fit together into a single program:

1. You are to implement the Whamo ADT using an array. This means you must implement each of the Whamo operations in such a way that a Whamo can be used to store any given type of element (as with lists, the objects stored in a Whamo will be restricted to be no larger than a pointer).
2. You are to write a program that accepts command–line arguments and performs the following operations in the order given here:
   - check that the correct number of command–line arguments is present.
   - determine which comparator is to be used, based on the command–line arguments.
   - open the input file that was specified on the command–line.
   - construct an empty Whamo (using `makeWhamo`).
   - read the input file data and place them in the Whamo.

- print the first item in the Whamo (by using `getfirstWhamo`).
- print the last item in the Whamo (by using `getlastWhamo`).
- print each item in the Whamo (by using `printWhamo`).
- transform the Whamo using `transformWhamo` and a transformer function that doubles the integer values.
- transform the Whamo using `transformWhamo` and a transformer function that takes the square root of the real values (but that does not change negative numbers).
- transform the Whamo using `transformWhamo` and a transformer function that changes every `'a'` in the string to `'z'`.
- print every item in the Whamo (by using `printWhamo`).

Your program is to be invoked with a command–line that looks like:

`proj2 {integer | real | string} {increase | decrease}` *filename*

The first two arguments on the command–line specify the comparator to be used, while the third specifies the name of the data file. For example, to run your program using data contained in the file `proj2.dat`, and placing those data in a Whamo that orders its elements in decreasing order of the real number elements of the tuples, you would use the command:

```
proj2 real decrease proj2.dat
```

A script produced by the program is shown below. The data file used for this script is the same file that was used in Homework 4. Homework 4 data files should be directly useable in this project (your program should just ignore the "number–of–elements" line in the data file).

The Whamo's array is to start off with a capacity of `WHAMO_INITIAL_CAPACITY` (defined in `whamo.h`). As the array becomes filled, its capacity is to be increased in increments of `WHAMO_CAPACITY_INCREMENT` (also defined in `whamo.h`). Each time the array is increased in capacity, a report is to be made to `stderr`, stating both the old and the new capacity. You *may not* use `realloc` to increase the size of the array; you must use `calloc` to allocate a new array, copy the old data over to the new array, then `free` the old array.

Error Conditions

Your program must send its output to `stdout`, except for error reports and capacity–increase messages (described above), which are to be written to `stderr`. You may assume that the input file is well formed. Its specification is exactly the same as for Homework 4, including the line indicating how many items are in the file (even though that line is not needed for and must not be used in this project).

Errors in the command–line arguments must be detected and reported. The program is to terminate on any command–line error, after issuing a message to `stderr`. The command–line arguments must conform to the following restrictions to be acceptable:

1. There must be exactly 3 command–line arguments (not including the name of the program).
2. The first argument must be one of `integer`, `real` or `string`.
3. The second argument must be one of `increase` or `decrease`.
4. The third argument must be the name of a readable file (it is not necessary to test that the contents of the file are of the correct form—assume that they are).

Suggestions

Overall Design

You will be playing two roles in this project, that of the ADT implementor, and that of the ADT user. The common meeting ground of these two roles is the ADT definition, as given in `whamo.h`. You must honor this specification from both points of view.

As the Whamo implementor, you will write `whamo.c`, in which you implement each of the functions described in `whamo.h` (you are of course allowed to include auxiliary functions that are not specified in `whamo.h` as well). You have no idea what data type the user might declare for `whamo_item_type`. All you know is that it will be no larger than a pointer. Your best bet is to assume it to be a `void` pointer. `Whamo.h` defines `whamo_type` to be `struct whamo_tag *`. You must define what a `struct whamo_tag` is in `whamo.c`. You will be wise to look at the way lists were done in `lists.h` and `lists.c` for guidance on how you should write `whamo.c` (these files are available from the course web page).

As the Whamo user, you don't see any of the implementor's code; you are unconcerned with the implementation, you just go ahead and use it. As with lists, you may declare `whamo_item_type` to be whatever type you like (as long as it is no larger than a pointer). You, the Whammo user, must not define `struct whamo_tag`; rather you should just use the `whamo_type` given you in the `whamo.h` interface file.

Comparators

As the Whamo user, you must provide a comparator (of the type used for `qsort`) that will work with your `whamo_item_type`. Remember that such comparators take as arguments two *pointers* to whamo_item_types, not whamo_item_types themselves. For example, if whamo_item_type is defined to be int, your comparator would be of the form:

```
int
comp(int *x, int *y)
{
  return *x - *y;
}
```

When handed as argument to makeWhamo, the comparator must be coerced to be of type comparator:

```
my_whamo = makeWhamo((comparator) comp);
```

Suppose now that whamo_item_type has been defined to be a *pointer* to an int. In this case the above style of comparator will not work, because two pointer dereferences are needed (one to dereference the pointer to the whamo_item_type, the other to dereference the pointer to the int). Thus, the comparator in this case must be of the form:

```
int comp(int **x, int **y)
{
  return(**x - **y);
}
```

Transformers

The Whamo user must write transformers for use with transformWhamo. Here is an example of a transformer for use when whamo_item_type is pointer to double:

```
whamo_item_type trans(whamo_item_type x)
{
  whamo_item_type temp = malloc(sizeof(double));

  *temp = *x * 2.0;
  return temp;
}
```

Printfuncs

The user must also provide a printfunc that determines how the elements of a Whamo are to be printed when printWhamo is called. For example, if whamo_item_type is pointer to double, the printfunc might be:

```
void printdata(FILE * fp, whamo_item_type x)
{
  fprintf(fp, "%f ", *x);
}
```

Submission

Submission of your program will be electronic, using the 'submit' program. To submit your files, use `proj2` as the project name. Thus, to submit the files `whamo.c`, `proj2.c` and `Makefile`, you should enter:

```
submit cs2021 proj2 whamo.c proj2.c Makefile
```

You must submit all files necessary to successfully compile your program, along with a Makefile. The executable must be named `proj2`, but you may name files as you wish. It is traditional to name the implementation file for an ADT like the interface file. Thus, the traditional name for your Whamo implementation file would be `whamo.c`; you are encouraged to name it so.

We will automatically compile and run your program on a fixed set of data. You may submit your program more than once, but only the last version you submit will be graded. Submissions made by midnight on the due date (Tuesday, 23 April) will receive the on–time bonus. Don't forget to include a comment at the top of your program stating your sources of help (as described in the Prospectus). You must submit all files necessary for successful compilation of your project, including your Makefile. You must submit a Makefile that correctly compiles your source files on the gl machines upon issue of the command `make`. The executable resulting from the compilation must be named `proj2`, and it must execute on the gl machines.

Sample Script

Here is a trace of a correct version of `proj2` being applied to the datafile that appeared in the Homework 4 handout (Handout #15).

```
umbc10[101] proj2 integer increase p2_data1.dat
Increasing size from 5 to 10
Increasing size from 10 to 15
Increasing size from 15 to 20

First value is (133, 4032.933105, Clinton)

Last value is (8504, 1492.899048, pickle)
```

```
(133, 4032.933105, Clinton)
(389, 4890.900879, warbler)
(574, 2386.268066, facetiously)
(690, 3183.364014, titular)
(806, 9371.234375, facetiously)
(1217, 4701.037109, hobo)
(1637, 4729.735840, circus)
(2383, -458.020996, kiss)
(2647, 6319.262207, grotesque)
(2796, 5462.934082, Placido)
(3086, 9614.063477, yogurt)
(3260, 527.440979, frenulum)
(3579, 8749.060547, Enrico)
(5436, 3918.111084, posse)
(6118, 1993.095947, warbler)
(6910, 9848.871094, grotesque)
(7811, 4597.146973, empath)
(7931, 7904.910156, Clinton)
(8504, 1492.899048, pickle)

(266, 63.505379, Clinton)
(778, 69.934975, wzrbler)
(1148, 48.849442, fzcetiously)
(1380, 56.421307, titulzr)
(1612, 96.805138, fzcetiously)
(2434, 68.564110, hobo)
(3274, 68.773071, circus)
(4766, -458.020996, kiss)
(5294, 79.493790, grotesque)
(5592, 73.911667, Plzcido)
(6172, 98.051331, yogurt)
(6520, 22.966084, frenulum)
(7158, 93.536415, Enrico)
(10872, 62.594818, posse)
(12236, 44.644104, wzrbler)
(13820, 99.241478, grotesque)
(15622, 67.802261, empzth)
(15862, 88.909561, Clinton)
(17008, 38.638050, pickle)
umbc10[102] exit
```

Here is the `main` function that was used to produce the script shown above. You may use this main function if you wish or you may write your own. In any event, the main function must perform the operations described above. This `main` is available from the course web page.

```
#include <stdio.h>
#include <stdlib.h>

#include "p2.h"

void main(int argc, char *argv[])
{
  whamo_type whamo;
  comparator comp;
  FILE * fp;

  /* Check number and composition of command line arguments; exit if wrong */
  CheckCmdLineArgs(argc, argv);

  /* Determine comparator to use */
  comp  = DetermineComparator(argv[1], argv[2]);

  /* Open input data file; exit if can't open it */
  fp = OpenInputFile(argv[3]);

  /* Make a new Whamo, and read the file into it */
  whamo = makeWhamo(comp);
  ReadFileIntoWhamo(fp, whamo);

  /* Print info from the Whamo */
  fprintf(stdout, "\nFirst value is ");
  printdata(stdout, getfirstWhamo(whamo));
  putc('\n', stdout);
  fprintf(stdout, "Last value is ");
  printdata(stdout, getlastWhamo(whamo));
  putc('\n', stdout);
  printWhamo(stdout, whamo, printdata);
  putc('\n', stdout);

  /* Do the transforms */
  transformWhamo(whamo, doubleint);
  transformWhamo(whamo, sqrtreal);
  transformWhamo(whamo, a2z);
  printWhamo(stdout, whamo, printdata);
}
```