CS 202 – Data Structures (Spring 2007)

Homework Assignment 2: Flexible Arrays

Assigned on 13 March, 2007 — Due by 23:00:00 on Thursday 29 March, 2007
Graded by Merve Çaylı (mervec@...)

PLEASE NOTE:

- The deadline is hard: No exceptions will be made.
- Solutions must be your own: No cooperation is permitted.
- Late solutions will be penalised by 10 points for each day of delay, but solutions that are late by more than 2 days will get 0 points.
- Solutions must be submitted via the WebCT server (whose clock may differ from yours): No other method of submission will be accepted.

1 Flexible Arrays

The array is a very useful data structure, as one can access or change any element in constant time, provided its index is known. For example, if one declares

```
int A[10000], i;
```

then one can use statements like:

```
A[920] = i;
...
i = A[549] + A[387];
```

However, arrays require that the memory needed to store all the elements is allocated contiguously and at once, as well as that the maximum array size is known in advance. A rather important consequence of this is that if for some reason one only uses a few cells in a large array, then one wastes a lot of memory. For example, one may need an array of 10,000 elements, but for some reason a program uses only the values at the indices 1, 3, 5, and 9970 in a given run. A declaration like

```
int A[10000];
```

would then allocate contiguous memory for 10,000 integers, but only a very tiny percentage of these would be used, which is very wasteful.

This is where the concept of flexible arrays comes in. A *flexible array* is like a normal array as far as how you use it is concerned: one can access the element at index i as easily as in a normal array. *Functionally*, there is no difference, but *computationally* there are differences in resource consumption. The first difference is that if one needs an array of 1,000,000 elements but really needs to use only a small number of its elements, then the flexible array requires a much smaller amount of memory. This flexibility comes at a price though: the second difference is that the access to an element of known index is not constant-time any more.

2 Representing Flexible Arrays

One can represent a flexible array as a linked list of chunks. The maximum size of the flexible array is not fixed in advance and can essentially be arbitrary. Each *chunk* is a small fixed-size array of the same type of objects as we want to store in the flexible array, plus some other information for managing the data structure. The `FlexibleArray` class is the analogue of the `List` class we studied in the lectures. It has some private fields, as shown here:

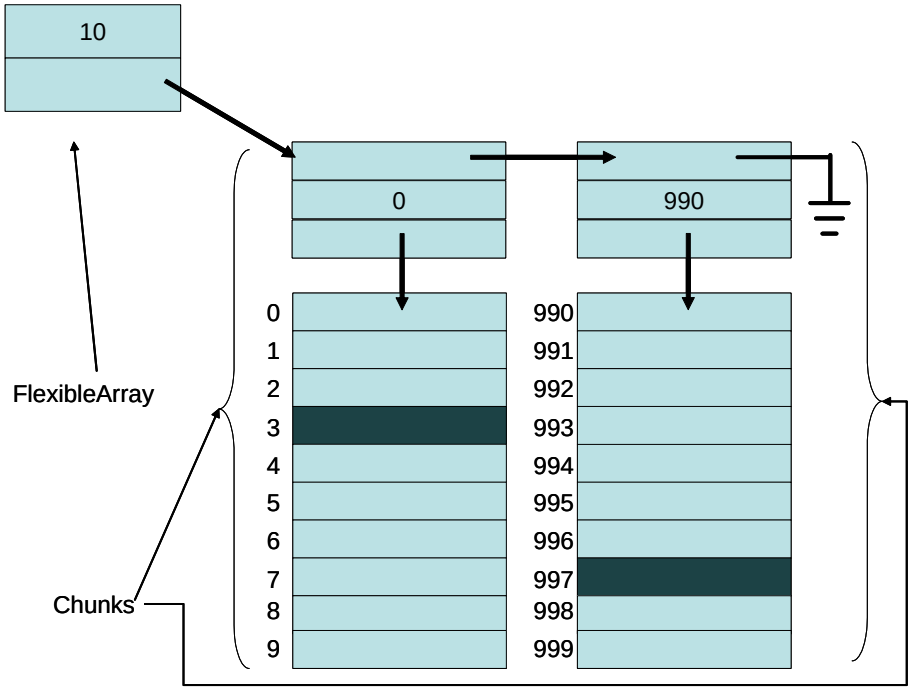
```
template <class Object>
class Chunk;

template <class Object>
class FlexibleArray
{...
private:
int ChunkSize;
Chunk<Object> *Chunks;
};
```

The `Chunk` class has the following general structure:

```
template <class Object>
class Chunk
{
...
Chunk<Object> *Next;
int ChunkBeginsAt;
Object *Array;
};
```

A flexible array with two chunks of ten elements and with only the elements at indices 3 and 994 being used can be depicted as follows:



Note that we have allocated 20 elements of the flexible array although it seems to have at least 1,000 elements. If we need to store a value at index 6, then we first go to the initial chunk (which holds the elements at indices 0 to 9) and then store that value at index 6 in that chunk. If we need to store a value at index 994, then we first find the chunk where `ChunkBeginsAt` $\leq 994 <$ `ChunkBeginsAt` + `ChunkSize` and then store that value at index 4 in that chunk. In this case, we have such a chunk, namely the last one. If, on the other hand, we need to store a value at index 556, then we first create a new chunk, link it in after the first chunk, and then store that value at index 6 in that new chunk. Chunks in the list must be ordered by their `ChunkBeginsAt` fields. We assume you now get the general idea of how flexible arrays work.

3 Work To Be Done

Complete the definitions of the `Chunk` and `FlexibleArray` classes outlined above. Provide the relevant constructor methods, but you are not required to provide destructor methods. For the `FlexibleArray` class, provide the necessary public methods to access or modify the element at any index of the flexible array, as well as the necessary private methods to add chunks as needed. Since we will not delete any elements or chunks, there is no need to keep a dummy header node like for lists. Your classes must work on elements of any other class.

Write a simple main program that will be used to test your classes. It reads a series of commands from the standard input and where necessary prints results on the standard output. Each input command is of one of the following forms:

- C *integer*: create a new flexible array of integers with chunk size *integer*.
- S *index integer*: store *integer* at *index*.
- G *index*: get and print the value at *index*.
- A $index_1$ $index_2$: add up the elements at $index_1$ and $index_2$ and print that sum.
- Q: quit the program.

For example, for the input

```
C 5
S 1 350
S 42 50
A 1 42
S 890 1000
A 890 1
G 890
Q
```

the output is

```
400
1350
1000
```

Assume that you *never* have to access an array location into which no value has been stored before. Also assume that all input commands are syntactically correct.

What and How to Submit When?

Your solution must be in a single file. Take our warning on plagiarism **very seriously**. We assume that by submitting your solution, you are certifying that you are submitting your own work. Take the following steps:

1. Name the folder containing your files as *LastnameName-StudentID-hw2*. Do not use any special Turkish characters in the folder name. Remove any executables (debug or release), as they take up too much space.

2. Compress your folder, giving *MehmetogluAli-15432-hw2.zip* for example. Make sure it uncompresses properly, reproducing your folder exactly, and actually corresponds to assignment 2.

3. Submit this compressed file via WebCT by the deadline given on the first page.

Your solution will be graded in the following way:

- If your program does not use the classes as outlined above, then you will get 0 points, even if your program works correctly otherwise.

- If the program does not compile, then you get 0 points, otherwise you get 30 points.

- We will run n tests on your homework. Some of these check boundary conditions. Your outputs must be *exactly* in the format described above, without a single additional character: otherwise your program is considered to function incorrectly. Each fully correct test result will earn you $50/n$ points.

- The style of your program will be graded on clarity, comments, etc. This will cover 20 points. You will however not get any of these points if your program fails in *all* the tests we perform. A nice looking but useless program is just a useless program, no matter how nice it is.

For late submissions, we will first grade your solution as indicated above and then discount accordingly. So, assuming you got 90 points, if you were late at least one second but at most one day, then your actual grade will be 80 points; if you were late more than one day but at most two days, then your actual grade will be 70 points; if you were late more than two days, then your actual grade will be 0 points and your solution will not be graded.

Have fun!