RAMAIAH UNIVERSITY OF APPLIED SCIENCES

Assignment

Course Code	CSC305A
Course Name	Programming Language Principles
Programme	B.Tech
Department	CSE
Faculty	FET

Name of the Student	Satyajit Ghana
Reg. No.	17ETCS002159
Semester/Year	05/2019
Course Leader(s)	Ms. Chaitra S

Declaration Sheet

Student Name	Satyajit Ghana		
Reg. No	17ETCS002159		
Programme	B.Tech	Semester/Year	05/2019
Course Code	CSC305A		
Course Title	Programming Language Principles		
Course Date		to	
Course Leader	Ms. Chaitra S		

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Contents

List of Tables

1 Question 1

Solution to Question No. 1

1.1 Implementation of the application using C-Programming

main.c

```c
#include <stdio.h>

#include "array_list.h"
#include "gst_item.h"

#define MAX_ITEMS 100000

int main(int argc, char **argv) {
    // testing
    array_list *list = init_array_list(free_gst_item);

    /**
     * A Single loop of 100000 objects, sequencial access
    */
    for (int i = 0; i < MAX_ITEMS; i++) {
        int item_id = i;
        int gst_slab_id = i % 5;
        double price = 1000.5f;

        gst_item *item = create_gst_item(item_id, gst_slab_id, price);

        list->add(list, item);
    }

    double total_amt = 0.0f;

    // calculate the GST'ed price
    for (int i = 0; i < MAX_ITEMS; i++) {
        gst_item *item = list->at(list, i);
        total_amt += get_mrp(item);
    }

    free_array_list(list);

    printf("TOTAL_ITEMS : %d\nTOTAL AMOUNT : %.10f\n", MAX_ITEMS, total_amt);

    fflush(stdout);

    return 0;
```

```
}
```

array_list.h

```c
#pragma once

struct array_list {
    void** data;
    void* (*at)(struct array_list*, int idx);
    int (*add)(struct array_list*, void* data);
    void (*free_item)(void* data);
    int length;
    int buffered_length;
};

typedef struct array_list array_list;

array_list* init_array_list(void (*)(void*));

void* array_list_at(struct array_list*, int idx);

int array_list_add(struct array_list*, void* data);

void free_array_list_items(array_list*);

void free_array_list(array_list*);
```

array_list.c

```c
#include "array_list.h"

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

// set the buffer to 256 elements
#define BUFFER 256

array_list* init_array_list(void (*free_item)(void*)) {
    array_list* mylist;
    if ((mylist = malloc(sizeof *mylist)) != NULL) {
        mylist->at = array_list_at;
        mylist->add = array_list_add;
        mylist->buffered_length = BUFFER;
        mylist->free_item = free_item;
        mylist->data = malloc(mylist->buffered_length * sizeof(*mylist->data));
        if (mylist->data == NULL) {
            return NULL;
        }
```

```c
        return mylist;
    }

    return NULL;
}

void* array_list_at(struct array_list* mylist, int idx) {
    if (mylist != NULL) {
        if (idx >= mylist->length) {
            perror("idx out of bounds\n");
            return NULL;
        }

        return mylist->data[idx];
    }

    return NULL;
}

int array_list_add(struct array_list* mylist, void* data) {
    if (mylist != NULL) {
        if (mylist->length >= mylist->buffered_length) {
            mylist->buffered_length += BUFFER;
            mylist->data = realloc(mylist->data, (mylist-
>buffered_length) * sizeof(*mylist->data));
        }

        mylist->data[mylist->length++] = data;

        return 1;
    }

    return -1;
}

void free_array_list_items(array_list* list) {
    for (int i = 0; i < list->length; i++) {
        list->free_item(list->at(list, i));
    }
    list->length = 0;
    list->buffered_length = 0;
}

void free_array_list(array_list* list) {
    free_array_list_items(list);
    free(list);
}
```

gst_item.h

```c
#pragma once

struct gst_item {
    int item_id;
    int gst_slab_id;
    double price;
};

typedef struct gst_item gst_item;

static float slab_tax[5] = {0.0, 0.05, 0.12, 0.18, 0.28};

float get_slab_tax(int slab_id);

double get_mrp(gst_item* item);

void* create_gst_item(int, int, double);

void free_gst_item(void*);
```

gst_item.c

```c
#include "gst_item.h"

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

float get_slab_tax(int slab_id) {
    return slab_tax[slab_id];
}

double get_mrp(gst_item* item) {
    assert(item != NULL);

    return item->price + get_slab_tax(item->gst_slab_id) * item->price;
}

void* create_gst_item(int item_id, int slab_id, double price) {
    gst_item* item;
    if ((item = malloc(sizeof *item)) != NULL) {
        item->item_id = item_id;
        item->gst_slab_id = slab_id;
        item->price = price;
        return item;
```

```
    }

    return NULL;
}

void free_gst_item(void* item) {
    gst_item* myitem = item;

    free(myitem);
}
```

1.2 Implementation of the application using JAVA-Programming using OOP features

App.java

```java
package app;

import java.util.ArrayList;

public class App {

    public static int MAX_ITEMS = 100000;

    public static void main(String[] args) throws Exception {
        // testing
        ArrayList<GSTItem> list = new ArrayList<>();

        /**
         * A Single loop of 100000 objects, sequential access
         */
        for (int i = 0; i < MAX_ITEMS; i++) {
            int item_id = i;
            int gst_slab_id = i % 5;
            double price = 1000.5f;

            GSTItem item = new GSTItem(item_id, gst_slab_id, price);

            list.add(item);
        }

        double total_amt = 0.0f;

        // calculate the GST'ed price
        for (int i = 0; i < MAX_ITEMS; i++) {
            GSTItem item = list.get(i);
            total_amt += item.get_mrp();
        }

        System.out.printf("TOTAL_ITEMS : %d\nTOTAL_AMOUNT : %.10f\n", MAX_ITEMS, total_amt);
    }
```

```
}
```

GSTItem.java

```java
package app;

/**
 * GSTItem
 */
public class GSTItem {
    private int item_id;
    private int gst_slab_id;
    private double price;

    GSTItem(int item_id, int gst_slab_id, double price) {
        this.item_id = item_id;
        this.gst_slab_id = gst_slab_id;
        this.price = price;
    }

    public static float slab_tax[] = { 0.0f, 0.05f, 0.12f, 0.18f, 0.28f };

    public double get_mrp() {
        return this.price + slab_tax[this.gst_slab_id] * this.price;
    }
}
```

1.3 Discussion on ease of writing program in C in comparison with that in JAVA

C is a bare bones language, i.e. it is much more primitive and has very low-level access to hardware, the inbuilt libraries that come with C is limited and most of the memory allocation is done manually by the programmer, this gives more control over the memory used by the program.

When it comes to Java there are a lot of inbuilt libraries that come with it, which makes it easier to write code, the memory management is done by JVM and deletion of variables is not given to the programmer.

For example, in the program that was implemented in 1.2 and 1.3, a data structure was required to store the GST Items, such as an ArrayList, by default an arraylist is not available in C, an array is available, but it is not generic, i.e. the array cannot change the data that it holds at runtime, therefore we implemented a custom array_list Abstract Data Type that can store any kind of data, now the memory management and the functions associated with this ADT was written manually by the programmer, which takes a lot of effort and the programmer needs to be very careful about the memory leaks that might happen if the ADT is not disposed properly. When it came to JAVA, we already have a proper implementation of ArrayList which can be directly imported from the library and used, all the basic functions are already defined, hence our program source code is much more smaller than C.

1.4 Discussion on the amount of changes required to introduce a new tax slab 25% and removing 28% slab in both the languages

The slab tax is stored in form of an array in both the languages, since the slabs are already predefined and fixed. Now if we have to introduce a new tax and remove the existing one, the number can be simply changed in the array. The new lines of code that will make the necessary changes:

In C

```
static float slab_tax[5] = {0.0, 0.05, 0.12, 0.18, 0.25};
```

In JAVA

```
public static float slab_tax[] = { 0.0f, 0.05f, 0.12f, 0.18f, 0.25f };
```

1.5 Discussion on the efficiency in terms of CPU and MEM usage using tools in both the languages

Tests were conducted using two tools, common for both Java and C

`Memory Profiling: memusage`

`CPU Profiling: perf`

Below is the RAW statistics provided by these tools

Language: C

CPU PERFORMANCE (PERF)

```
TOTAL_ITEMS : 100000
TOTAL AMOUNT : 112656300.1282616258

 Performance counter stats for 'gst-c/build/gst-c':

              9.54 msec task-clock:u              #    0.967 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
             1,031      page-faults:u             #    0.108 M/sec
```

```
        13,126,242      cycles:u                  #    1.376 GHz
        46,010,443      instructions:u            #    3.51  insn per cycle
         9,413,655      branches:u                #  986.741 M/sec
             4,664      branch-misses:u           #    0.05% of all branches

       0.009869029 seconds time elapsed

       0.003299000 seconds user
       0.006539000 seconds sys

# Overhead  Command  Shared Object     Symbol
    34.66%  gst-c    libc-2.29.so      [.] _int_malloc
    12.91%  gst-c    libc-2.29.so      [.] _int_free
    10.70%  gst-c    [unknown]         [k] 0xffffffff8a800b07
    10.13%  gst-c    libc-2.29.so      [.] malloc
     6.74%  gst-c    libc-2.29.so      [.] malloc_consolidate
     6.33%  gst-c    libc-2.29.so      [.] cfree@GLIBC_2.2.5
     5.28%  gst-c    gst-c             [.] main
     4.76%  gst-c    gst-c             [.] get_mrp
     4.14%  gst-c    gst-c             [.] create_gst_item
     2.87%  gst-c    gst-c             [.] array_list_at
     1.03%  gst-c    ld-2.29.so        [.] strcmp
     0.46%  gst-c    ld-2.29.so        [.] __GI___tunables_init
```

MEMORY USAGE (MEMUSAGE)

```
Memory usage summary: heap total: 2404904, heap peak: 2400808, stack peak: 2112
         total calls   total memory   failed calls
 malloc|     100003        1606184              0
realloc|        390         798720              0  (nomove:342, dec:0, free:0)
 calloc|          0              0              0
   free|     100001        1600040
Histogram for block sizes:
   16-31         100000  99% ==================================================
   32-47              1  <1%
 2048-2063            1  <1%
 4096-4111            2  <1%
 6144-6159            1  <1%
 8192-8207            1  <1%
10240-10255           1  <1%
12288-12303           1  <1%
14336-14351           1  <1%
16384-16399           1  <1%
18432-18447           1  <1%
20480-20495           1  <1%
22528-22543           1  <1%
24576-24591           1  <1%
26624-26639           1  <1%
28672-28687           1  <1%
30720-30735           1  <1%
32768-32783           1  <1%
34816-34831           1  <1%
36864-36879           1  <1%
38912-38927           1  <1%
40960-40975           1  <1%
43008-43023           1  <1%
45056-45071           1  <1%
47104-47119           1  <1%
```

```
49152-49167          1  <1%
51200-51215          1  <1%
53248-53263          1  <1%
55296-55311          1  <1%
57344-57359          1  <1%
59392-59407          1  <1%
61440-61455          1  <1%
63488-63503          1  <1%
   large           360  <1%
```

Language: JAVA

CPU PERFORMANCE (PERF)

```
TOTAL_ITEMS : 100000
TOTAL_AMOUNT : 112656300.1282616300

 Performance counter stats for 'java -cp GSTJava/bin app.App':

           212.06 msec task-clock:u              #    1.459 CPUs utilized
                0      context-switches:u        #    0.000 K/sec
                0      cpu-migrations:u          #    0.000 K/sec
            4,752      page-faults:u             #    0.022 M/sec
      359,432,031      cycles:u                  #    1.695 GHz
      411,960,222      instructions:u            #    1.15  insn per cycle
       78,327,254      branches:u                #  369.371 M/sec
        2,874,409      branch-misses:u           #    3.67% of all branches

       0.145300220 seconds time elapsed

       0.189254000 seconds user
       0.026494000 seconds sys

# Overhead  Command          Shared Object       Symbol
     2.14%  java             libjvm.so           [.] 0x0000000000d14785
     1.14%  java             libjvm.so           [.] 0x0000000000d147b1
     0.82%  java             [unknown]           [k] 0xffffffff8a800b07
     0.78%  java             libjvm.so           [.] 0x0000000000b7a168
     0.63%  java             libjvm.so           [.] 0x0000000000b7a5fb
     0.63%  java             libjvm.so           [.] 0x0000000000d147ae
     0.54%  C1 CompilerThre  ld-2.29.so          [.] __tls_get_addr
     0.52%  java             libjvm.so           [.] 0x0000000000d147b7
     0.52%  java             [JIT] tid 1347      [.] 0x00007f9ab89654c4
     0.47%  java             libjvm.so           [.] 0x0000000000d14b69
     0.44%  java             libc-2.29.so        [.] __vfprintf_internal
     0.43%  java             ld-2.29.so          [.] do_lookup_x
     0.43%  java             [JIT] tid 1347      [.] 0x00007f9ab8957096
     0.39%  java             libjimage.so        [.] 0x0000000000002d15
     0.39%  java             [JIT] tid 1347      [.] 0x00007f9ab89780fb
     0.39%  java             libjvm.so           [.] 0x0000000000c6a4e0
     0.38%  java             [JIT] tid 1347      [.] 0x00007f9ab89654c0
     0.38%  java             [JIT] tid 1347      [.] 0x00007f9ab8974702
     0.38%  java             libjvm.so           [.] 0x0000000000bb074a
     0.38%  java             libjvm.so           [.] 0x00000000008246cc
     0.38%  java             [JIT] tid 1347      [.] 0x00007f9ab8965516
     0.37%  java             libjimage.so        [.] 0x0000000000003a1f
     0.37%  java             ld-2.29.so          [.] _dl_relocate_object
```

```
  0.37%  java             libjvm.so          [.] 0x00000000008246c7
  0.36%  C1 CompilerThre  libc-2.29.so       [.] __vfscanf_internal
  0.36%  java             [JIT] tid 1347     [.] 0x00007f9ab896bd88
  0.35%  java             libc-2.29.so       [.] _int_malloc
  0.30%  C1 CompilerThre  libc-2.29.so       [.] __memmove_avx_unaligned_erms
  0.29%  java             ld-2.29.so         [.] strcmp
  0.27%  C1 CompilerThre  [JIT] tid 1347     [.] 0x00007f9ab8957096
  0.27%  java             [JIT] tid 1347     [.] 0x00007f9ab8969568
  0.26%  C1 CompilerThre  libc-2.29.so       [.] cfree@GLIBC_2.2.5
```

MEMORY USAGE (MEMUSAGE)

```
Memory usage summary: heap total: 14905189, heap peak: 12809891, stack peak: 30688
         total calls   total memory   failed calls
 malloc|      11386       14668887              0
realloc|         38           1120              0  (nomove:0, dec:0, free:0)
 calloc|        110         235182              0
   free|       4545        2169338
Histogram for block sizes:
    0-15            383   3% ===
   16-31           5356  46% ==================================================
   32-47           2258  19% =====================
   48-63           1224  10% ===========
   64-79            375   3% ===
   80-95            153   1% =
   96-111            56  <1%
  112-127           167   1% =
  128-143           111  <1% =
  144-159            22  <1%
  160-175            27  <1%
  176-191            64  <1%
  192-207            18  <1%
  208-223            10  <1%
  224-239            37  <1%
  240-255            54  <1%
  256-271            25  <1%
  272-287            53  <1%
  288-303            68  <1%
  304-319            28  <1%
  320-335            18  <1%
  336-351           128   1% =
  352-367             2  <1%
  368-383            62  <1%
  384-399             9  <1%
  400-415             2  <1%
  432-447            65  <1%
  464-479             2  <1%
  480-495             2  <1%
  496-511             2  <1%
  512-527             3  <1%
  528-543             2  <1%
  544-559           147   1% =
```

1.5.1 Performance results analysis

Both the programs were run for the exact same data for the exact same number of iterations, i.e. 100000 items with varying types of GST Tax slabs and the MRP for each of these commodities is calculated by the program.

Here is a table that describes the summary of the performance parameters

Table 1 Performance Comparison

Parameter	**C**	**JAVA**
Execution Time	0.009869s	0.145300s
IPC	3.51 ins/cycle	1.15 ins/cycle
Instructions	46,010,443 ins	411,960,222 ins

We can clearly observe in this scenario C is approximately 15 times faster than Java, this is without the optimizations done in C, which will make it even faster. Since Java is a interpreted language it is bound to be slower, the byte code generated is run on a JVM that uses the JIT Compiler to compile and run the code.

Here is a table that compares the memory usage of the two languages

Table 2 Memory Comparison

Parameter	**C**	**JAVA**
malloc	1568.53 kB	14325.08 kB
realloc	780 kB	1.09 kB
calloc	0 kB	229.67 kB
free	1562.53 kB	2118.49 kB
heap total	2348.53 kB	14555.84 kB
stack peak	2.06 kB	29.96 kB

Similarly, as observed previously, C is approximately 7 times less memory consuming, comparing the malloc and stack peak. `malloc` is the main routine in Linux `libc` that allocates memory for the program at runtime, both our programs make runtime memory allocation to make this comparison fair. Another thing to note is that the amount of memory allocated by malloc and then freed is important, since this shows the memory leaks that can happen in the program.

Another interesting thing to note in the memusage dump is that C allocated 99% of its memory chunks of the size 15-31 bytes, i.e. most of the memory that is used by the program was only for the gst item objects that we allocated at runtime. While in Java the allocation is distributed only about 46% of its memory was 15-31 byte chunks, this is because of the other objects that are associated with the gst item that also need to be allocated along with it.

In C the difference of malloc and free is 6kB while in JAVA it is 12206kB, which means that this memory wasn't freed at runtime, it must have been freed either by the JVM or by the OS when the program terminated.

The conclusion from this analysis is that JAVA has much more overhead than C since it is interpreted and has many inbuilt complex data structures that require a lot of runtime overhead, this contributes to the slower runtime and the higher memory usage. This can be seen in the Overhead displayed in perf analysis, Java several calls to `[JIT] tid <tid>`.

Bibliography