MidExam More Examples Spring 2018

#1 Option 1 Assume the following **Java** declaration within a **method**:

String name = "Peg Board";

List and explain its **data characteristics**. You may assume that integers are 4 bytes and pointers are 8 bytes.

location?

automatic for the variable, dynamic (or literal pool) for string

data type?

string

structure?

object containing len, offset, pointer to string

size?

string object = 16 bytes (since Java stores len, offset and string ptr take sizeof(len) + sizeof(offset)+ sizeof(pointer))

string literal – literal = 9\*2 bytes

length = 4 bytes

additional overhead for garbage collection and storage management?

value?

"Peg Board",

value of variable can change, string cannot

length set to 9

#1 Option 2 Offset calculations

For the following data structure, assume int and float are 4 bytes and must be aligned on an offset that is a multiple of 4.

typedef struct

{

char szName[18];

char szStudentId[8];

float gradeM[20];

} Student;

typedef struct

{

int iStudentCnt;

Student studentM[20];

} StudentData;

StudentData info;

a. What is the size of each of the following?

Student 18 + 8 + 2 (slack to get an offset mult of 4) + 4\*20 = 108

StudentData 4 + 108 \* 20 = 2164

b. Given the origin of the array, how would you compute (show a formula) the address of each of the

info.studentM[i].szStudentId[j]

origin + 4 + 108\*i + 18 + 1\*j

info.studentM[i].gradeM[j]

origin + 4 + 108 \* i + 28 + 4 \* j

#2. List the **operational characteristics** of the **C** += operator. (e.g., x += y)

|  |
| --- |
|  |
| syntax - infix |
| precedence is needed |
| binary (two operands) |
| explicit operation and operands |
| works for many data types including numerics and pointers  doesn't work for struct and array |
| generic |
| has side effect of assigning a value |
| compound operation |

#7. Describe all bindings involved with that variable declaration statement at the following times:

Assume the following **Java** declaration within a **method**:

String name = "Peg Board";

a. language implementation

size of pointers

use of UTF-16

format of int values used in objects

b. compile or load

data type of variable

object attributes for variable

length of string literal

c. execution

memory location in stack on method entry ; freed on method exit

allocate string object at runtime in the heap

string literal placed in literal pool

value assigned to variable on method entry

#8. Suppose a **variable-sized string** is represented in **C** using the following typedef:

typedef struct

{

short shMaxLength; // maximum length of the string (this is the size we malloc)

short shLength; // current number of characters in the string

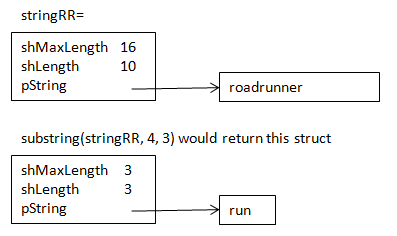
char \*pString; // pointer to dynamic memory containing a contiguous array

// with a size of shMaxLength. It is not zero terminated.

} VarString;

Show code for the C function, **VarString substring(VarString string, int iBeg, int iLgth**), which should return a substring of the specified string beginning at iBeg for a length of iLgth. Notes:

* The first character in the string will be at subscript 0. The last character will be at subscript shLength-1.
* If the request went outside the range of characters, use errExit to terminate with an appropriate message.
* Allocate memory (use malloc) for the contiguous array of iLgth characters. If the malloc fails, use errExit to terminate with an appropriate message.
* It is not necessary to allocate a VarString structure using malloc since we are simply returning a structure.
* Return the VarString structure.
* The VarString typedef is defined in "VarString.h". You do not have to include the standard C include files.
* See the example in the extras.:



Invoking code snippet:

#include "VarString.h"

void someFunc(VarString stringTeam)

{

VarString stringPart;

…

stringPart = substring(stringTeam, 4, 3);

…

}

VarString substring(VarString string, int iBeg, int iLgth)

{

VarString vsRet;

vsRet.shMaxLength = 3;

vsRet.shLength = 3;

vsRet.pString = (char\*)malloc(iLgth \* sizeof(char));

memcpy(string.pString + iBeg \* sizeof(char), vsRet.pString, iLgth)

return vsRet;

//Long’s code

if(iBeg > string.shLength && iBeg <0)

errExit(“fuck u”);

VarString a;

a.pString = (char\*)(malloc(iLgth));

if(a.pString == NULL)

errExit(“can not allocate …”);

//else

a.shMaxLength = (short)iLgth;

a.shLength = (short)iLgth;

memcpy(string.pString+iBeg,a.pString,iLgth);

return a;

}

#8 Suppose a **homogeneous** **unbounded** array of any data type is represented in **C** using the following typedef:

typedef struct

{

short shElementSize; // size of a single element in the array

short iAllocatedElements; // number of allocated elements

int iHighestPopulatedElement; // highest subscript of a populated element

void \*pArray; // pointer to memory containing a contiguous array

} ARRAYSTRUCT;

Show code for the C function, **int reAllocate**(ARRAYSTRUCT \*pStruct), which should

* allocate memory (use malloc or calloc) for the new array, doubling its current size
* copy the old data into the new array
* set the attributes in the structure
* functionally returns TRUE if the reallocate was successful; otherwise it functionally returns FALSE
* what else is needed?