COMP0002 Programming Principles

Exercises 4 Example Answers to Some Questions

Q4.3 Write a version of the string copy function that takes a single string argument and creates a new copy in dynamic (heap) memory. The function signature should be:

```
char *stringCopy(char *s)
```

Answer:

```
char *stringCopy(char *string)
{
  char *copy = malloc(sizeof(char) * strlen(string) + 1);
  char *copyPointer = copy;
  while ((*copyPointer++ = *string++));
  return copy;
}
```

This shows a typical way of writing the function, but there are many variations possible. Items to note are:

- Strings are stored in character arrays but no array (square bracket) syntax is used, just pointers.
- The `malloc` function call allocates the amount of heap memory needed to store the copy of the string. The argument to `malloc` calculates the number of bytes required. The `sizeof` operator is used to find the number of bytes needed to store a single `char` value, and multiplying by the length of the string to be copied gives the amount of memory to store the string excluding the null (zero) at the end of the string. The `+1` adds the extra character needed for the null. It is a common mistake to forget to add the `+1` leading to unexpected errors when the program is run.
- The memory for the copy must be allocated on the heap, as a local array declared using square bracket syntax (e.g., `char copy[strlen(string) + 1];`) would be allocated memory on the stack that would deallocated when the `stringCopy` function returned. This would lead to the copy being overwritten when the stack memory is allocated to another function call.
- The while has an empty body, just the semi-colon denoting an empty statement, as all the work is done in the boolean expression. The post-increment ++ operators take care of moving the pointers forward along each string.
- The while loop copies the characters in the string and the `null` (`'\0'`) at the end, as the assignment of `null` takes place before any ++ operators, and the value of an assignment expression is that the value that has been assigned. When `null` is assigned, the value of the boolean expression is zero, which is false, and the loop terminates.
- The `malloc` function can return `0` (`null`) if there is no memory available[1]. A programmer could add an extra statement:
  `if (!copy) return 0; // Return null if the memory allocation failed`
  This returns `null` to the code calling the `stringCopy` function, but means that the calling code has to check for `null` and implement a strategy for handling the failed memory allocation,

[1] This is only sort of true! The way `malloc` behaves is rather more complicated, notably on Linux systems where `malloc` does lazy memory allocation and will appear to be able to allocate much more memory than actually exists.

making it more complicated. If all calls to `malloc` are checked for `null` it starts to create a big overhead, and makes the design of the program more complex. In practice many programmers never check for `null` and assume that the program will not run out of memory. Of course, if the program does actually run out of memory it will crash in an unpredictable way.

Q4.4 Write a function with the signature `int strend(char *s, char *t)` that returns true (`1`) if the string pointed to by `t` occurs at the end of the string pointed to by `s`. Otherwise the function returns false (`0`).

Example answer:

```
int strend(char *s, char *t)
{
   if (strlen(s) < strlen(t))
   {
    return 0;
   }
   int length = strlen(t);
   s += strlen(s);
   t += length;
   while (length-- >= 0)
   {
     if (*s-- != *t--)
     {
       return 0;
     }
   }
   return 1;
}
```

This code works by first checking that the string being searched for is not longer that than the string being searched. Then the rest of the function compares the two strings in reverse, starting with the last character in each string and moving to the previous character until either there are no more characters to compare or a mismatch is found.

Note that the function has three return statements and, hence three exit points. The two `return 0;` (false) statements are used to terminate the function early as soon as the result is known to be false. This style of writing a function with multiple exits is widely used and most programmers would follow the style, which often results in shorter code. However, there are some that would advocate there should be only one exit point at the end of the function, and would modify the code to achieve this even if it gets a bit more complicated. The reason is that is can make code easier to analyse and verify as correct.

The parameter variable names are `s` and `t`, which in hindsight are not at all descriptive and probably make the code harder to read. Can you think of better names that would make the code more readable for you?

Q4.5 Write a function with the signature `int *sort(int *n)` that takes a pointer to an array of integers, and returns a pointer to a new array containing the integers in sorted order.

Example answer:

```
#define SIZE 10


int compareInt(const void *a, const void *b)
{
```

```
  int *x = (int *) a;
  int *y = (int *) b;
  return *x - *y;
}

int *sort(int *array)
{
  int *copy = malloc(sizeof(int) * SIZE);
  int *copyPointer = copy;
  int count = 0;
  while (count++ < SIZE)
  {
    *copyPointer++ = *array++;
  }
  qsort (copy, SIZE, sizeof(int), compareInt);
  return copy;
}
```

This answer uses the `qsort` function from the standard C library to do the actual sorting. The `qsort` function has four arguments: a pointer to the array to sort, the size of the array, the size of an `int`, and a pointer to a function to compare integers. `qsort` is a general purpose sorting function that can sort arrays of any type. This is why the size of the values in the array is one of the parameters, and why a pointer to a comparison function is passed as another parameter since a different function is needed for each type.

The `compareInt` function does the integer comparison used by `qsort`. As `qsort` can work with arrays of any type the comparison function takes two parameters of type `void*`, which are then cast to the actual type needed to make the comparison. This is a common technique in C but it does rely on the programmer getting the types correct, and the compiler cannot detect errors with incorrect cast expressions that muddle up types.

A pointer of type `void*` can be thought of as a general purpose pointer that can point to anything but carries no type information about what actual value the pointer is pointing at. Note that in the call to `qsort` the pointer to the compareInt function is obtained by simply giving the function name. Within the `qsort` implementation the pointer is then used to call the function with two arguments.

To make the sort function more cohesive it would be better to extract the code to copy the argument array into its own function:

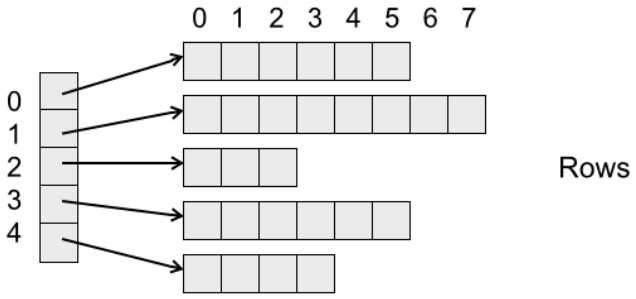
```
int *copyIntArray(int *array)
{
  int *copy = malloc(sizeof(int) * SIZE);
  int *copyPointer = copy;
  int count = 0;
  while (count++ < SIZE)
  {
    *copyPointer++ = *array++;
  }
  return copy;
}

int *sort(int *array)
{
  int *copy = copyIntArray(array);
  qsort (copy, SIZE, sizeof(int), compareInt);
  return copy;
```

```
}
```

A bigger issue, though, is that the code above can only work with arrays of size 10 as specified by the `#define SIZE 10`. To make the code more flexible each function needs an additional parameter giving the size of the array being sorted. This will mean more work for the programmer to avoid making mistakes over array sizes.

Q4.6 a) Write a function to create and return a ragged 2D array data structure. A ragged 2D array has rows of different lengths. The basic data structure has an array of pointers that point to each row, each of which is an array of integers:



The function should take as a parameter the number of rows and an array that specifies the length of each row.

Example Answer:

To get started the example program below provides functions to create and free a ragged array, along with helper functions to access the array elements and print out a ragged array. The `main` function provides examples of how the functions are used.

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

int *createRow(int size)
{
  return calloc(sizeof(int), size);
}

int **createRaggedArray(int numberOfRows, int *rowLengths)
{
  int **newRagged = calloc(sizeof(int*), numberOfRows);
  for (int row = 0 ; row < numberOfRows ; row++)
  {
    *(newRagged + row) = createRow(*(rowLengths + row));
  }
  return newRagged;
}

void freeRaggedArray(int **raggedArray, int numberOfRows) {
  for (int row = 0 ; row < numberOfRows ; row++) {
    free(*(raggedArray + row));
  }
  free(raggedArray);
}
```

```
void printRow(int* row, int length)
{
  for (int n = 0 ; n < length ; n++)
  {
    printf("%d ", *(row + n));
  }
  printf("\n");
}

void printRaggedArray(int **raggedArray, int numberOfRows, int *rowLengths)
{
  for (int row = 0 ; row < numberOfRows ; row++) {
    printRow(*(raggedArray + row), *(rowLengths  + row));
  }
  printf("\n");
}

void setValue(int **raggedArray, int aRow, int aColumn, int value)
{
  *(*(raggedArray + aRow) + aColumn) = value;
}

int getValue(int **raggedArray, int aRow, int aColumn) {
  return *(*(raggedArray + aRow) + aColumn);
}

int main(void)
{
  int rowLengths[] = {5,7,2,10,4,20,3,6};
  int **raggedArray = createRaggedArray(8, rowLengths);
  printRaggedArray(raggedArray, 8, rowLengths);
  setValue(raggedArray, 1, 1, 100);
  setValue(raggedArray, 4, 3, 200);
  printRaggedArray(raggedArray, 8, rowLengths);
  printf("1,1 = %d\n", getValue(raggedArray,1,1));
  printf("4,3 = %d\n", getValue(raggedArray,4,3));
  printf("5,10 = %d\n", getValue(raggedArray,5,10));
  freeRaggedArray(raggedArray, 8);

  return 0;
}
```

The code above uses pointer notation and pointer operators all the way through. This version of the data structure does not store any information in the data structure itself about the length of each row or the number of rows. Hence, a separate array of integers is required to record the length of each row, and is used by the `createRaggedArray` and `printRaggedArray` functions. In addition, the number of rows has to be maintained separately and passed to functions as needed.

The type of a pointer to a 1D array of int is `int *`, the type of a pointer to an `int *` array is `int **` (pointer to a pointer to `int`). This is why the type of a ragged array as created by the `createRaggedArray` function is `int **`. If you have trouble following the use of pointers in the program then try sketching pictures of the arrays and pointers to see how they fit together.

b) It would be better if the ragged array data structure also stored information about the number of rows and the length of each row. Modify your data structure so that it can store the size information, and also provide functions to create and free a ragged array, along with helper functions to get and set values in a ragged array and print out a ragged array.

Example Answer:

The program below shows how this can be done.

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

typedef struct Row {
  int length;
  int *contents;
} Row;

typedef struct RaggedArray
{
  int numberOfRows;
  Row *rows;
} RaggedArray;

void initialiseRow(Row *row, int length)
{
  row->length = length;
  row->contents = calloc(sizeof(int), length);
}

RaggedArray *createRaggedArray(int numberOfRows, int *rowLengths)
{
  RaggedArray *newRagged = malloc(sizeof(RaggedArray));
  newRagged->numberOfRows = numberOfRows;
  newRagged->rows =  malloc(sizeof(Row) * numberOfRows);
  for (int n = 0 ; n < numberOfRows ; n++)
  {
    initialiseRow(newRagged->rows + n, *(rowLengths + n));
  }
  return newRagged;
}

void freeRowContents(Row *rows, int numberOfRows) {
  for (int n = 0 ; n < numberOfRows; n++) {
    free((rows + n)->contents);
  }
}

void freeRaggedArray(RaggedArray *raggedArray) {
  freeRowContents(raggedArray->rows, raggedArray->numberOfRows);
```

```
  free(raggedArray->rows);
  free(raggedArray);
}

void printRow(Row *aRow)
{
  for (int n = 0 ; n < aRow->length ; n++)
  {
    printf("%d ", *(aRow->contents + n));
  }
  printf("\n");
}

void printRaggedArray(RaggedArray *raggedArray)
{
  for (int n = 0 ; n < raggedArray->numberOfRows ; n++)
  {
    printRow(raggedArray->rows + n);
  }
  printf("\n");
}

void setValue(RaggedArray *raggedArray, int aRow, int aColumn, int value)
{
  int* rowContents = (*(raggedArray->rows + aRow)).contents;
  *(rowContents + aColumn) = value;
}

int getValue(RaggedArray *raggedArray, int aRow, int aColumn) {
  int* rowContents = (*(raggedArray->rows + aRow)).contents;
  return *(rowContents + aColumn);
}

int main(void)
{
  int rowLengths[] = {5,7,2,10,4,20,3,6};
  RaggedArray *raggedArray = createRaggedArray(8, rowLengths);
  printRaggedArray(raggedArray);
  setValue(raggedArray, 1, 1, 100);
  setValue(raggedArray, 4, 3, 200);
  printRaggedArray(raggedArray);
  printf("1,1 = %d\n", getValue(raggedArray,1,1));
  printf("4,3 = %d\n", getValue(raggedArray,4,3));
  printf("5,10 = %d\n", getValue(raggedArray,5,10));
  freeRaggedArray(raggedArray);

  return 0;
}
```

To allow the ragged array to store information about its size, two structs are used. Each struct has two parts, a size and a pointer to data. For a row the struct is:

```
typedef struct Row {
  int length;
  int *contents;
} Row;
```

The struct is combined with a typedef to make it easier to use the struct type, as the typedef allows a new type name to be defined based on an existing type. In this case the type name will be Row, which can be used instead of struct Row. Note that it is a convention to use the same name for the struct and typedef (i.e., Row) to reduce the number of distinct names needed.

The Row struct illustrates a common solution in C for providing arrays that know their own size. Unlike C-Strings that use zero (null or '\0') to mark the end of the string in a character array, we cannot use zero to mark the end of an int array, as zero is a valid number that might be stored in the array. In this respect, character arrays are a special case as the the value zero in ASCII is defined as the null character, which can be safely used as an end of string marker without any confusion with being a character that is part of the string.

Given the Row struct, a RaggedArray struct can be defined, again combined with a typedef:

```
typedef struct RaggedArray
{
  int numberOfRows;
  Row *rows;
} RaggedArray;
```

Each row is now represented by a Row struct, which in turn contains the pointer to the int array.

With the structs in place, the required functions can be written fairly straightforwardly. The example code uses pointer operations only, but square bracket syntax might make the code a bit more readable in places. Making sure that the pointer operations are all correct, and that memory is used correctly, requires careful checking. Really this should be done by having a comprehensive set of unit tests, but there is a tool called valgrind (http://valgrind.org) that can be used to check that only valid memory is accessed and all memory is freed correctly. Valgrind currently works best on Linux, and is a good excuse for setting up a Linux virtual machine to do your programming on.