Question 1 [10 marks]

An infinite geometric series is described by the formula:

$$\sum_{n=0}^{\infty} ar^n = a + ar + ar^2 + \ldots$$

The partial sum of this series is described by the the formula:

$$S_k = \sum_{n=0}^{k} ar^n = a + ar + ar^2 + \ldots + ar^k$$

(a) [5 marks] Write an *iterative* C function named `partial_sum` that calculates and returns the partial sum of this geometric series for specified values of *a*, *r* and *k*. The function prototype is:

```
double partial_sum(double a, double r, int k);
```

Assume that parameter `k` will always be greater than or equal to 0. (Your function does not have to check this).

Your function must use a loop. No marks will be awarded for a recursive solution.

(b) [5 marks] Write a *recursive* C function named `partial_sum_r` that calculates and returns the partial sum of this geometric series for specified values of *a*, *r* and *k*. The function prototype is:

```
double partial_sum_r(double a, double r, int k);
```

Assume that parameter `k` will always be greater than or equal to 0. (Your function does not have to check this).

Your function must not call the `partial_sum` function you wrote for part (a). No marks will be awarded for a solution that uses a loop.

Question 2 [10 marks]

Here are the formulas for calculating the two roots of the quadratic equation $ax^2 + bx + c = 0$:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \; x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Here is the prototype for a function that uses these formulas to compute the roots of the quadratic equation with the specified coefficients *a*, *b* and *c*:

```
_Bool quadratic_roots(double a, double b, double c,
                      double *root1, double *root2);
```

Parameters `root1` and `root2` point to the variables where the function will store the roots it calculates. This function returns `true` if the roots were computed, and returns `false` if no roots were computed because coefficient *a* was zero. (To simplify things, you can assume that the term under the square root sign will never be negative, so the function will never have to calculate roots that are complex numbers.)

(a) [5 marks] Write the complete implementation of function `quadratic_roots`.

(b) [5 marks] Consider these test cases:

Case 1: $x^2 - x - 12 = 0$. Expected roots are 4 and -3, `quadratic_roots` should return `true`.
Case 2: $2x^2 - 8x + 6 = 0$. Expected roots are 1 and 3, `quadratic_roots` should return `true`.
Case 3: $6x + 12 = 0$. No roots are computed, `quadratic_roots` should return `false`.

Write a function named `test_quadratic_roots` that tests your `quadratic_roots` function. Here is the function prototype:

```
void test_quadratic_roots(double a, double b, double c);
```

The three parameters are the three coefficients that will be passed to `quadratic_roots`.

Here are some examples of what `test_quadratic_roots` should print if the function it tests (`quadratic_roots`) is correct. Unlike the test functions you wrote in the lab, `test_quadratic_roots` should only print the actual results computed by `quadratic_roots`. Instead, to simplify your function, it will not print the expected results for us to compare to the actual results.

Test Case 1: This function call:

```
test_quadratic_roots(1.0, -1.0, -12.0);
```

prints:

```
Calling quadratic_roots with coefficients: 1.0 -1.0 -12.0
Return value: true
Calculated roots: 4.0 -3.0
```

This question continues on the next page.

Test Case 3: This function call:

```
test_quadratic_roots(0.0, 6.0, 12.0);
```

prints:

```
Calling quadratic_roots with coefficients: 0.0 6.0 12.0
Return value: false
Calculated roots: none
```

Don't worry about getting the formatting of the real numbers exactly as it is shown here. It doesn't matter if your function would print more than one digit after the decimal point.

Question 3 [5 marks]

Here are the measurements (in meters) of a northern town's annual snowfall over the past nine years:

6.0, 5.0, 5.3, 6.4, 5.3, 5.6, 5.8, 6.3, and 6.9

The values vary from year to year, obscuring any trend.

One way to smooth out the fluctuations is to calculate the *sliding average* or *running average* of the measurements. The sliding average is defined as the average of each measurement and its immediate neighbours. To illustrate, here is how the sliding averages for some of the measurements listed above are calculated:

- Sliding average of the first measurement (6.0): (6.0 + 5.0) / 2
- Sliding average of the second measurement (5.0): (6.0 + 5.0 + 5.3) / 3
- Sliding average of the third measurement (5.3): (5.0 + 5.3 + 6.4) / 3
- Sliding average of the last measurement (6.9): (6.3 + 6.9) / 2

Note that the calculations for the first and last measurements is different than the calculations for the second through next-to-last measurements.

Write a function called `sliding_average` that calculates the sliding averages of the first *n* measurements stored in an array. The function prototype is:

```
void sliding_average(double a[], int n, double sliding[]);
```

You can assume that there will always be two or more measurements in the array. (Your function does not have to check this.)

As an example of how this function is used, here is a C statement declares an array named `annual`, and initializes it with the nine annual snowfall measurements:

```
double annual[] = {6.0, 5.0, 5.3, 6.4, 5.3, 5.6, 5.8, 6.3, 6.9};
```

The sliding averages of these measurements are to be stored in an array named `averages`:

```
double averages[9];
```

The `sliding_averages` function would be called this way:

```
sliding_average(annual, 9, averages);
```

Question 4 [15 marks]

The C program shown here may not make much sense, but it is a perfectly legitimate program. which will compile and run without errors.

```
#include <stdio.h>

int do_this(int first, int *second)
{
    int third;

    third = first - 3;
    first = *second + third;
    *second = first * third;

    /* Point A */

    return third;
}

void do_that(int p[], int n, int *this_one, int that_one)
{
    int the_other;

    the_other = 2;
    p[that_one * 2] = 1;
    *this_one = p[n-1] - p[1];
    that_one = *this_one / the_other;

    /* Point C */

    return;
}

int main(void)
{
    int a[4];
    int j, m, n;

    a[0] = 2; a[1] = 4; a[2] = 8; a[3] = 16;

    m = 6;
    n = 5;
    j = do_this(m, &n);

    /* Point B */

    m = 4;
    n = 1;
    do_that(a, 4, &m, n);

    /* Point D */

    return 0;
}
```

This question continues on the next page.

Answer the following questions using the notation used in the lectures and labs.

(a) Draw a memory diagram that depicts the program's stack frames when execution reaches `Point A`, immediately before the `return` statement in `do_this` is executed.

(b) Draw a memory diagram that depicts the program's stack frames when execution reaches `Point B`, immediately after the call to `do_this` returns.

(c) Draw a memory diagram that depicts the program's stack frames when execution reaches `Point C`, immediately before the `return` statement in `do_that` is executed.

(d) Draw a memory diagram that depicts the program's stack frames when execution reaches `Point D`, immediately after the call to `do_that` returns.

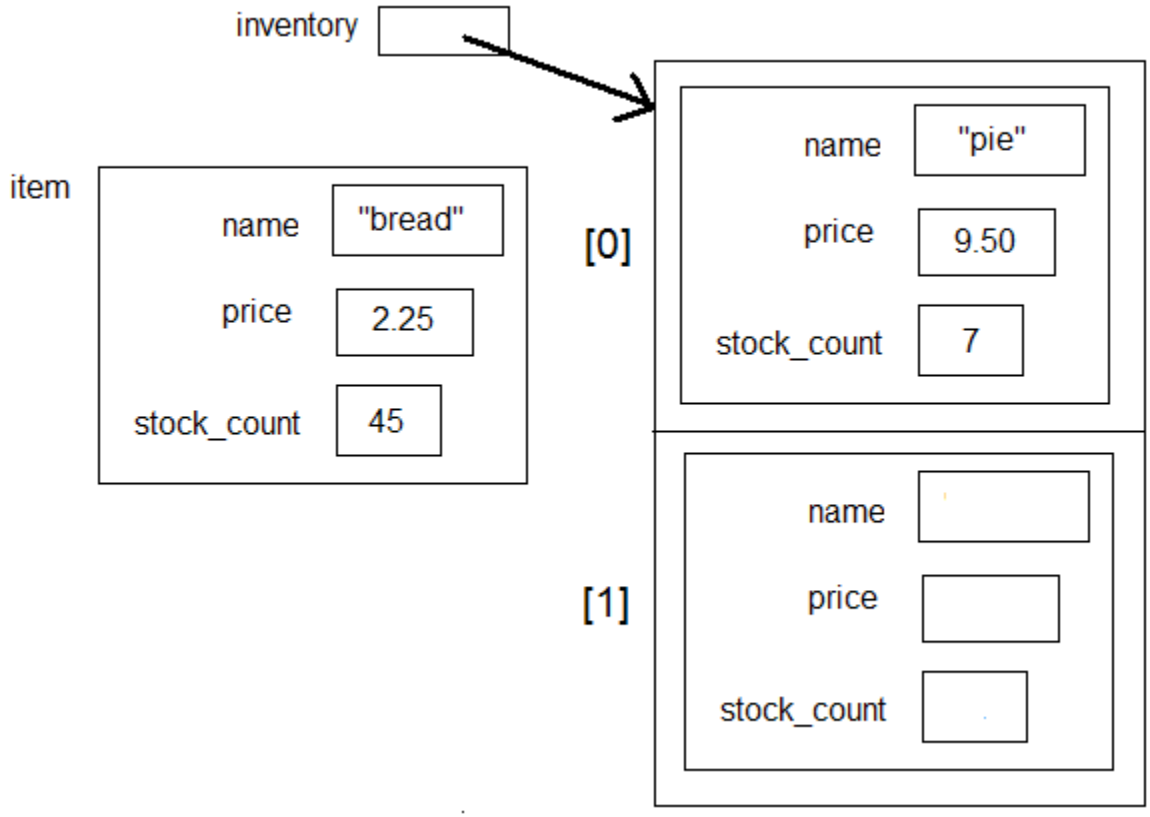
Question 5 [10 marks]

Here is the declaration for a structure that describes products in a bakery's inventory:

```
struct product {
    char *name;
    double price;
    int stock_count;
};

typedef struct product PRODUCT;
```

For parts (a) and (b), you will write the code that, when executed, would result in this memory diagram:



This question continues on the next page.

(a) **[3 marks]** Write the C code to declare variable `item` and initialize it as shown; i.e., with information about loaves of bread. Don't write a complete function; just write the required statements.

(b) **[5 marks]** Write the C code to dynamically allocate an array from the heap that can store two products, with the first product initialized as shown; i.e., with information about pies. Don't write a complete function.

(c) **[2 marks]** Modify the memory diagram on this page (don't redraw the diagram in your answer booklet) so that is shows what memory would look like after this code fragment is executed:

```
PRODUCT *p;
p = &inventory[1];
*p = item;
```

Question 6 [15 marks]

Towards the end of term, we examined the implementation of a list collection that uses a dynamically allocated array as the underlying data structure. Here are the declarations:

```
struct intlist {
    int     *elems;
    int     capacity; // Maximum number of elements in the list.
    int     size;     // Current number of elements in the list.
};

typedef struct intlist IntList;
```

We call `intlist_construct` to create new, empty lists; for example,

```
// Create a list with capacity 10
IntList *list;
list = intlist_construct(10);
```

Recall that `intlist_construct` will allocate two blocks of memory from the heap. One block is the instance of the `IntList` structure, and the other is the array of 10 integers.

Here is function `intlist_append`, which stores an element at the end of the list:

```
_Bool intlist_append(IntList *p, int element)
{
    assert(p != NULL);
    if (p->size == p->capacity) {
       return false;     /* The list is full */
    }
    p->elems[p->size] = element;
    p->size += 1;
    return true;  /* Element successfully inserted. */
}
```

The list has a fixed capacity, and the function returns `false` if we attempt to add an integer to a full list. We would like to remove this limitation.

(a) [11 marks] Write a function named `increase_capacity` that attempts to enlarge a list's capacity to the specified value. Here is the function prototype:

```
void increase_capacity(IntList *p, int new_capacity);
```

You can assume that parameter `new_capacity` is greater than the list's current capacity (your function does not have to check this). The function should not change the order of the integers stored in this list; for example, suppose a list contains [1 7 3 -2 9] when `increase_capacity` is called. After the function returns, the list will contain the same integers, in the same order (1 is stored at index 0, 7 is stored at index 2, etc.) It's up to you to decide how the function should handle any errors that occur while it is executing.

(b) [4 marks] In your answer booklet, write a new implementation of `intlist_append` that, if the list is full, doubles the list's capacity before appending the element. The function's return type and parameter list must not be changed. Your solution must call your `increase_capacity` function.

Selected Functions from the C Standard Library

string.h

```
char *strcpy(char destination[], const char source[]);
```
Copies the C string pointed by *source* into the array pointed by *destination*, including the terminating null character.

```
char *strcat(char destination[], const char source[]);
```
Appends a copy of the *source* string to the *destination* string. The terminating null in *destination* is overwritten by the first character in *source*.

```
int strlen(const char str[]);
```
Returns the length of string *str*. The length does not include the string's terminating null character.

```
int strcmp(const char str1[], const char str2[]);
```
Compares strings *str1* and *str2*. A return value of zero value indicates that both strings are equal. A value greater than zero indicates that the first character that does not match has a greater value in *str1* than in *str2*. A value less than zero indicates the opposite.

math.h

```
float powf(float x, float y);
double pow(double x, double y);
```
Returns the value of `x` raised to the power of `y`.

```
float sqrtf(float x);
double sqrt(double x);
```
Returns the square root of `x`.

stdlib.h

```
void *malloc(int size);
```
Allocates a block of memory with the specified size from the heap. Returns a pointer to the block. Returns the `NULL` pointer if the block cannot be allocated.

```
void *calloc(int num, int size);
```
Allocates space for an array of `num` elements, where each element consists of `size` bytes. Each element is set to zero. Returns a pointer to the array. Returns the `NULL` pointer if the array cannot be allocated.

```
void free(void *ptr);
```
Cause the space pointed to by `ptr` to be deallocated (made available for subsequent allocation by `malloc` or `calloc`.) The behaviour of this function is undefined if `ptr` is not a pointer that was previously returned by `malloc` or `calloc`. The behaviour of this function is undefined if the memory pointed to by `ptr` was previously deallocated by a call to `free`.

This is the last page of the question paper.