

CMPT 225 Lab 2 - Dynamically Allocated Arrays

Exercise 1 - Array of integers

Problem Statement and Requirements

In this Exercise 1, you are to write a C++ program that sums integers.

NOTE: You will be given more specific instructions in the section **Solution Recipe** below, so, for now, simply read the problem statement outlined in this section.

Your solution (i.e., program) to this problem (described above) will require only one file and you will not need to create any classes. Your program must contain a few functions and one of them will be the **main()** function.

More specifically, your program must:

1. Prompt the user to enter the total number of values to be summed.
2. Create a dynamically allocated array of this size.
3. Prompt the user to enter each integer value, one at a time, read the value and store it into the array until the array is full.
4. Sum these values and print this sum. More specifically,

- The sum of the values in the array must be calculated using a separate function with this header:

```
int sumArray(int arr[], unsigned int arrSize);
```

- This function must use a loop to calculate the sum of the array values and must work with an array of any size.

In this program, the number of elements stored in the array is actually equal to the array size. Therefore, the parameter **arrSize** not only represents the array size, but it also represents the number of elements stored in the array.

However, do remember that this is not always the case: in general, the size of an array (i.e., the number of cells in the array, also known as its **capacity**) does not always equate the number of elements stored in the array.

5. Delete the dynamically allocated array before terminating the program.
 - Make sure you know the difference between the **delete[]** operator and the **delete** operator.
6. Have the **main()** function return the sum of the values.

Question: Once your program has finished executing and the execution flow has returned to the command line, which Linux command must you use (at the command line) in order to print the sum the **main()** function returned?

Solution Recipe

Steps:

- Create a directory within your **\$fhome/cmpt-225** directory (command: **mkdir Lab2**), then within **Lab2**, create a directory called **Ex1**.
- Make **Ex1** your current directory.
- Use a text/code editor to create a file which you must call **sum_array.cpp** and save it in this directory.

- Create your solution to the problem stated in the section **Problem Statement and Requirements** above, i.e., write your code in **sum_array.cpp** following the 6 steps outlined in the section above. If you are not familiar with the syntax one must use to create heap-allocated arrays, you may want to read the section **C++ Syntax Notes** below.

- Compile **sum_array.cpp** into an executable called **sum_array** as follows:

```
g++ -Wall -o sum_array sum_array.cpp
```

Note the **-Wall** option here (**-W** stands for warnings, **all** for all types of warnings). This option tells the compiler to be extra picky, and give you warnings about potentially hazardous bits of code. It is a good idea to always use this option. If you have been using other options when using the compiler command, please, feel free to add them to the above command. As long as they do not clash with the suggested options in the above command.

- Download [this zip file](#) and unzip it in your **Ex1** directory. This zipfile contains three test cases: three test data files (**1.in**, **2.in**, **3.in**), their corresponding expected result files (**1.er**, **2.er**, **3.er**) and a test script (**test.py**).
- Run the test script as follows:

```
uname@hostname:~/sfuhome/cmpt-225/Lab2/Ex1: ~$ ./test.py
```

If you have correctly built the executable **sum_array** and if your program solves the problem stated above, you will see:

```
Running test 1... passed
Running test 2... passed
Running test 3... passed
Passed 3 of 3 tests
```

on the computer monitor screen.

If your program passed all 3 tests, please, ask a TA to take a look at your source code (**sum_array.cpp**) to make sure you have successfully satisfied the requirements of this section of the lab and that you are using **GPS** (Good Programming Style) then move on to the next section of this lab. Otherwise, debug your program and test again!

C++ Syntax Notes

Stack-allocated (automatically allocated) Arrays

When a program executes, the system allocates a section of the memory segment called the **stack** to the executing function. This allocated section of stack memory is referred to as the **stack frame** of the executing function.

When you declare an array as follows (here, an array of integers):

```
constexpr unsigned int ARR_SIZE = 4;
int arr[ARR_SIZE];
```

the system automatically allocates some of the executing function's stack frame to the array named **arr**, giving this array sufficient amount of stack memory such that it can accommodate **ARR_SIZE** integers (int).

One could replace the code fragment above (2 lines) with the code fragment below (1 line). However, the code fragment below:

```
int arr[4];
```

does not follow the good programming style (GPS) used in this course. This is to say that one must not hard-code the array size (or array capacity) in the array declaration, but declare this array size as either a variable or a constant (the latter is exemplified above) and use this variable or constant in the array declaration as illustrated in the code fragment above (2 lines).

Here are some characteristics of stack-allocated memory:

- This segment of the memory is managed efficiently by the microprocessor: it does not become fragmented.
- One does not have to explicitly allocate and free variables allocated on the stack memory segment.

- Used for local variables and parameters, i.e., data that only needs to be available during the execution of a function.
 - Variables cannot be resized.
 - Stack memory segment is limited in size (OS-dependent).
-

Heap-allocated (dynamically allocated) Arrays

For this lab, you need to create a **dynamically allocated** array. To do this you need to ...

1. declare a pointer variable,
2. find out how big the array must be (i.e., its size or capacity) and
3. allocate this new array on the heap memory.

When you implement your solution to the problem stated above, you may find these few lines of code helpful.

```
int * arr = nullptr;           // "arr" is a pointer variable, pointing to an array of int
unsigned int arrSize = 0;      // "arrSize" holds the number of cells in the array
                               // Right now, initialized to 0 - User-defined

// Get the array size from the user using cin:
cin >> arrSize;
arr = new int[arrSize];        // Create a heap-allocated (dynamically allocated) array
...
```

Note that having the user enter the size of the array is one way to obtain this size. One can also use a constant, as we have done in the **Stack-allocated Arrays** section, and use it to initialize **arrSize**.

To learn more about **Dynamic Allocation of Arrays**, please, refer to Section C2.5 of the C++ Interlude 2 in the **Data Abstractions & Problem Solving with C++, Walls and Mirrors (6th or 7th Edition)** textbook. You may also find Section C2.3 helpful.

Here are some characteristics of heap-allocated memory:

- This kind of memory allocation is done dynamically, as the program executes, as opposed to automatically when the variables (such as array **arr**) are declared.
 - One must manage this memory segment by explicitly allocating and freeing space. This signifies that there is no guarantee that this memory segment is used efficiently: it may become fragmented over time as blocks of memory are allocated then freed.
 - The size of the heap memory segment is not as limited as the size of the stack memory segment.
 - Used for variables containing large amount of data such as arrays.
 - Used for data that needs to persist (i.e., accessible) beyond the execution of a function.
 - Variables (arrays) can be resized. To know more about **Resizable Arrays** (also called **Expandable Arrays**), please, refer to the file called **Expandable Array** posted under Lecture 3 on our course web site.
-

Exercise 2 - Array of Circles

Now, let's store pointers to Circle objects into a dynamically allocated array:

```
Circle ** circleArray = nullptr;
```

You may wish to create a second subdirectory within **Lab2**, perhaps called **Ex2**.

Again, to solve this problem, you will only need one file which will contain the **main()** function. Call this file **circle_array.cpp**. You will also need the **Circle** class you created in Lab 1.

Your program must:

1. Prompt the user to enter the number of Circle objects to create.

Similarly to Exercise 1 above, in this program, the number of elements (i.e., pointers to Circle objects) stored in the array is actually equal to the array size. Therefore, this value not only represents the array size, but it also represents the number of elements stored in the array.

2. You may wish to validate this number: what if the user entered a negative number?
3. Create a dynamically allocated array of that size.
4. For each cell in the array ...
 - Prompt the user to enter the centre coordinates (x,y) and the radius of a circle.
 - Instantiate a Circle object with these values, then store the pointer (memory address) to this Circle object in the array.
5. At this point in your program, feel free to manipulate each/some of these Circle objects by calling their appropriate methods. For example, you could "move" the Circle objects around and see if they intersect.
6. Finally, print the content of the entire array, i.e., for each cell:
 - Print the value of the index of the cell.
 - Print the Circle object to which the pointer stored in this cell is pointing using the appropriate Circle method.
 - Print the area of this Circle object, setting precision to 6 figures.
7. Delete the dynamically allocated array before terminating the program.
8. This program returns 0.

Code Reuse

As you are writing your solution to this Exercise 2, feel free to ...

- recycle sections of the **sum_array.cpp** you wrote in the first exercise of this lab when creating **circle_array.cpp**.
- use the **Circle.h** and **Circle.cpp** you wrote in Lab 1. If you designed and implemented your **Circle** class as an **ADT** (Abstract Data Type), no modification should be required. In this Exercise 2, observe how your **circle_array.cpp** becomes a client code of your **Circle** class.

Compiling and Testing

Use this [Makefile](#) to compile your **circle_array.cpp**.

Once your code compiles, let's test it! In this course, we use the [white-box testing strategy](#) (thank you, Wiki!). Using this testing strategy, our goal is to execute each of the code statements in our program. Often, this cannot be done using only one (1) test case. Therefore, we need to create several test cases in order to execute all of our code.

OK! Let's give this white-box testing strategy a go by creating test cases to test your program **circle_array.cpp**, i.e., by defining, for each test case, its test data and expected results.

How many test cases must you create? Good question! Again, look at your code and create as many as you need in order to execute each statement in your program.

For example:

1. Test Case 1 - test data: Enter zero (0) as the array size. What kind of result would you expect?

2. Test Case 2 - test data: Enters -7 as the array size. This test case would test your user input validation code. Again, what kind of result would you expect?
3. Test Case 3 - test data: Enters 4 as the array size. Then, insert 4 Circle objects (their associated pointers) into your array. In this test case, you would need to define the test data required in order to create these 4 Circle objects. As part of your test case, you may print the content of your array. What kind of result would you expect?
4. etc.

When creating test cases for a program such as **circle_array.cpp**, i.e., a program with the **main()** function, as opposed to a class, the idea is to ensure that all of the code (statements) has been executed at least once.

When creating test cases for a class, the idea is to instantiate various objects (for example, Circle objects having different values assigned to their data members, in other words, Circle objects having different "states") and to call each of the class' private and public methods at least once using each of these objects.

Lastly, make sure that your test cases are distinct, i.e., they test different statements of your program. However, it is OK! if your test cases do execute some of the same statements in your program. The bottom line is that you do not want to have test cases testing exactly the same statements. That would not be efficient!

Observation

Notice that the programs you were asked to write, namely **sum_array.cpp** in Exercise 1 and **circle_array.cpp** in Exercise 2, were assigned more than one responsibility. Indeed, each of these programs were responsible ... :

1. for interacting with the user (prompt the user, read in what the user enters at the command line, validate what the user enters, etc.), and
2. for managing the data, i.e., managing the data structure **array** (creating the array, inserting data into the array, etc.).

How can this design be improved? By assigning only one responsibility to each of our class/program. This is what we did when we designed the solution to our FriendsBook problem in class. The **FriendsBook program** was assigned the responsibility of interacting with the user, the **Profile class** was assigned the responsibility of modelling a user of the social network and the **List data collection ADT class** was assigned the responsibility of managing the data.

Assigning only one responsibility to each of our class/program is the design we shall follow when solving problems in this course. Stay tuned!

Enjoy!

CMPT 225 - [School of Computing Science](#) - [Simon Fraser University](#)