Background

So far, all your programs have used **static memory allocation** to reserve space in memory for variables of known memory size. This means that when the code is compiled, available memory is located and allocated to declared variables for use. However, in many instances, you may not know the proper size of memory needed or you may have a constraint on the amount of memory you can use.

Dynamic memory allocation allows you to allocate memory for variables whose sizes are determined during the run time of the program instead of at compile time. In this way, memory can actually be allocated and freed, allowing for very efficient memory management in critical pieces of software. There are four main dynamic memory allocation functions in C, and we will focus on the two simplest in this assignment: `malloc()` and `free()`.

The `malloc()` function (which stands for **m**emory **alloc**ation) takes as its input the number of **bytes** of memory to allocate and outputs a pointer to the beginning of that allocated memory. This is why the size of memory does not have to be known at run time. An example of using `malloc()` to allocate memory for 3 integers is shown below. Note the `(int *)` in front of the function. This is called **type casting** and is necessary to use `malloc()` the way we want in this example. Also, note that the `sizeof()` function is used only within the `malloc()` function to reserve the correct amount of space for variables of type `int`.

```
int *pointer;
pointer = (int *) malloc(3*sizeof(int));
```

The `free()` function takes as its input a pointer to a dynamically allocated section of memory and opens up that block of memory to be used again by the program. If allocated memory is not freed, then calling `malloc()` over and over again will eventually lead to running out of memory to use. An example of using `free()` is shown below.

```
free(pointer);
```

Since `malloc()` returns a pointer, it is necessary to be able to manipulate pointers to fully make use of dynamic memory allocation. Thus, recall that you can increment a pointer by an integer to change the location in memory to which it points (pointer arithmetic) and that you can use a pointer to access both the address and stored value.

If there is not sufficient contiguous memory available when `malloc()` is called, the memory allocation fails and the function returns `NULL`.

Problem Statement

In this extension assignment, you will create a simple **data packet**. A data packet is a unit of data which can be transmitted between processors for communication purposes. It is a sequence of values/data/information which the processors can understand. In general, a data packet has three main components: a **head**, a **body**, and a **tail**. The body is the actual data being transmitted which has some value beyond the communication. The head and the tail may contain meta information about the data packet itself which helps the processors understand the body data. So, when a processor receives a data packet, it first reads the head to understand what the form of the body data, then it reads in all the body data to store and manipulate, and then it reads the tail to know it has reached the end of the body and possibly check its work.

For this assignment, the data packet you create will have all three parts. The body will be a sequence of integer values entered by the user. The head will have just one value which is the number of integers entered by the user. The tail will also have just one value which is the sum of all the integers entered by the user. All of these values will be stored in order (head, body, tail) and in consecutive locations in memory.

To do this, you will use a single loop counter variable and a single pointer. **Thus, you are only allowed to declare the two variables listed below, and you may only use one ampersand (&) in your code to read the initial input of the number of integers.**

```
int i;
int *ptr;
```

Instructions

- Complete the C28-1 assignment which introduces pointers in C.
- Create a new program and complete the following:
  - Declare the two variables above.
  - Prompt the user for the desired number of integers to store. You may scan this value into the variable **`i`**.
  - Allocate the proper number of bytes to the pointer for the entire data packet (including the head and tail) based on the user input. You may use the variable **`i`** which has the number of integers to do the dynamic memory allocation.
  - Check to see if the memory allocation succeeded.
  - Create the entire data packet with reference to the pointer:
    - The first value is the number of integers.
      - This should be retrieved from the value stored in the variable **`i`**. After this point, you will have to reference the first value in the data packet when you want to know how many integers there are.
    - The middle values are the individual integers which the user will be prompted to enter.
      - You may use the variable **`i`** as a loop counter variable.
    - The last value is the sum of the integers.
      - You may use the variable **`i`** as a loop counter variable.
  - Print the address and value of each element in the entire data packet to the screen.
    - You may use the variable **`i`** as a loop counter variable.
  - You must use pointer arithmetic with the pointer and the variable **`i`** to store, access, and update all data packet values.
  - Free the allocated memory.