Using Pointers to Save Time and Space

1. **Using a pointer to simulate an array.**

   One way to declare an array's length at *run-time*, that is, when the space-allocation statement executes, is to use a feature called "variable length arrays (VLA's)" that the most modern version of C (called C99) provides. Here is an example:

```
int length;

printf("How many temperatures will you enter? ");
fflush(stdout);
scanf("%i", &length);

double temperatures[length];
```

This statement cannot be placed earlier in the code – it must appear only after the variable *length* has been assigned a meaningful value.

   In the above, temperatures is a full-fledged array. However, If you have an older compiler (pre-C99), or if you are in one of the (infrequent) circumstances in which VLA's are not allowed in C99, or if you need an array-like object that can change its length as execution progresses, the above VLA solution is not available. Instead, the solution is to ***simulate an array using a pointer***. Here is an example:

```
int length, k;
double *temperatures;

printf("How many temperatures will you enter? ");
fflush(stdout);
scanf("%i", &length);


temperatures = (double*) malloc(length * sizeof(double));


for (k = 0; k < length; ++k) {
    ... temperatures[k] ...
}
```

The *malloc* function allocates the requested amount of space and returns a pointer to that space.

The *malloc* function returns a *pointer* to the space – note the asterisk.

Allocate space for this many entries in the simulated array.

In this example, each entry of the simulated array will hold a *double*.

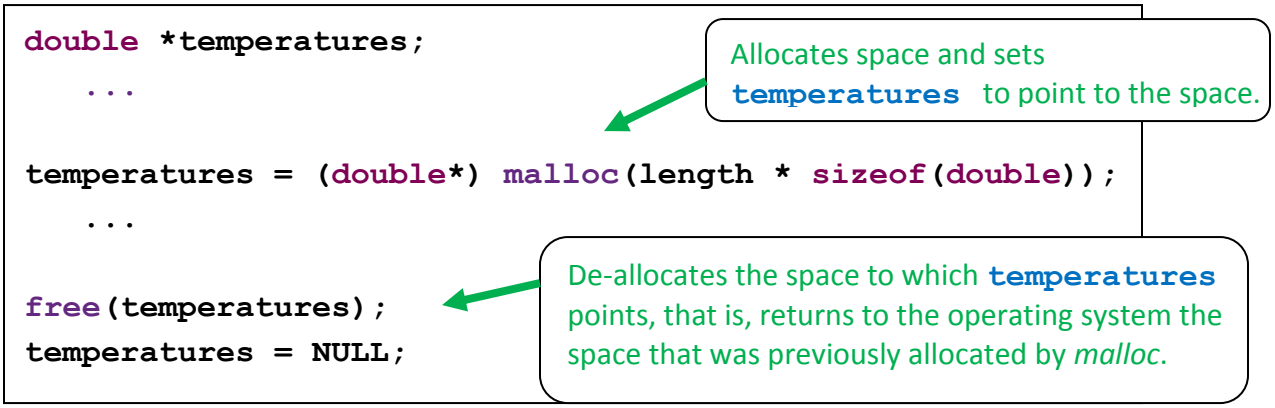
After the *malloc* statement, treat the pointer as if it were an array, that is, use the bracket notation that arrays use.

   Technically, `temperatures` in the above code is a *pointer* and not an *array*. However, you can (and should) use `temperatures` with the same bracket notation that you use for arrays, as shown in the example. Thus, the pointer ***simulates*** an array.

There are several important differences between a pointer that simulates an array and an actual array:

1. Whenever you use *malloc* to allocate space, you are responsible for de-allocating that space when you are done with it, that is, to return it to the operating system. Failing to do so is called a ***memory leak*** – as the program runs, space "leaks" out of the operating system into the program until the operating system runs out of space and the program crashes (or the system grounds to a crawl).

   You use the *free* function to de-allocate space, as in this example:



   After the *free* statement, `temperatures` still points to the same space to which it previously pointed, but the contents of that space may change in whatever way the operating system chooses. Thus, it is safer to reset `temperatures` to the special value `NULL`, which means that `temperatures` no longer points to anything. You have to `#include <stdio.h>` for `NULL` to be defined.

   For every *malloc*, there should be a corresponding *free*.

2. A pointer that simulates an array can be reassigned, where an array cannot. So, if the size of the simulated array needs to change, you can do so like this:

```
temperatures = realloc(temperatures, newLength * sizeof(double));
```

   If there is danger that *realloc* cannot assign the new space (e.g., if the requested amount of space is larger than the operating system has available), then you should guard against this as described in http://c-faq.com/malloc/realloc.html.

3. With an actual array, you can initialize the contents of the array like this:

```
float weights[] = {45.7, 900.4, 32.8, 74.3};
char name[] = "Nelson Mandela";
```

   No such notation is available for a pointer that simulates an array.

2. **"Ragged" two-dimensional arrays – an application of using pointers to simulate arrays.**

A "ragged" two-dimensional array is one in which the lengths of the rows vary, like this:

```
99  40  17  88
15   8  45
66  48  22  29  30  95  76  80
10  74   5  99
```

Here is how one codes a ragged array:

```
int numberOfRows, numberOfColumns, j, k;
int* rowLengths;
double** matrix;


printf("How many rows will you need? ");
fflush(stdout);
scanf("%i", &numberOfRows);


rowLengths = (int*) malloc(numberOfRows * sizeof(int));

matrix = (double**) malloc(numberOfRows * sizeof(double*));


for (j = 0; j < numberOfRows; ++j) {
    printf("How many columns do you need in row %i? ", j);
    fflush(stdout);
    scanf("%i", &numberOfColumns);

    matrix[j] = (double*) malloc(
                        numberOfColumns * sizeof(double));
}


for (j = 0; j < numberOfRows; ++j) {
    for (k = 0; k < rowLengths[j]; ++k) {
        matrix[j][k] = 0;
    }
}
```

A one-dimensional array (simulated by a pointer) whose entries specify the lengths of the rows of the two-

The two-dimensional array, really an array (the rows, simulated by a pointer) of arrays (for each row, the columns, simulated by pointers).

Allocate space for the array of row lengths.

Allocate space for the rows in the two-dimensional array.

For each row, ask the user how many columns are needed for that row.

Then allocate space for those columns. Note that `matrix[j]` is the *j*th row of the two-dimensional array.

An example showing how you can use the (simulated) two-dimensional array with the bracket notation.

This example continues on the next page.

```
   ...

for (j = 0; j < numberOfRows; ++j) {
    free(matrix[j]);
}

free(rowLengths);
free(matrix);
```

When you are done with the two-dimensional array, you must "free" (i.e., return to the operating system) all of the array's space that was allocated with *malloc*.

3. **Passing an array to a function – another application of using pointers to simulate arrays.**

   When you send an array to a function, the compiler actually sends a *pointer* whose value is the address of the first element in the array. This has three important consequences:

   1. The function call is fast – even with an array of many thousands of elements, only a *single* thing (the address of the first element in the array) is copied and sent to the function.
   2. The function can modify the elements of the array, and those modifications will still be in effect after control returns to the caller.
   3. Usually, you also pass the *length* of the array to the function, since arrays do not know their length.

   You don't need to know anything else about pointers to pass arrays to functions, since the compiler allows you to continue to use array notation (square brackets), like this:

```
void printArray(float numbers[], int length) {
     int k;

     for (k = 0; k < length; ++k) {
          printf("%f\n", numbers[k]);
     }
}
```

Indicate that the parameter is an array by putting square brackets after the array name.

Don't put anything inside the square brackets – the compiler ignores anything there, so putting something there only misleads the reader.

You use the array using the usual bracket notation, even though this is really a pointer simulating an array.

If you will loop through the array in the function, you need to pass the array's length, as in this example.

```
int lengthOfWeightsArray = 400;
float weights[lengthOfWeightsArray];
   ...
printArray(weights, lengthOfWeightsArray);
```

Pass the array by putting its name as the argument, WITHOUT brackets. The name of the array is really a pointer.