# Programming with C

Terry Marris December 2010

## *15 Arrays of Numbers*

In the previous chapter we looked at arrays of *char*. Now we look at arrays of numbers such as *ints* and *doubles*.

### 15.1 Arrays of Int

We have already met an array of *char*:

```
char string[10];
```

An array of integers is declared in much the same way:

```
int number[10];
```

This declaration says that *number* is an array that can contain up to 10 integers.

number

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 33 | 16 | 11 | 35 | 25 | 39 | | | | |

The size of the array is 10, it has 10 cells or elements. The length of the sequence of numbers stored there is six, the number of integers held in the array.

The integers 0..9 are known as indices. Array indices always start with zero. Each index addresses its own element in the array. So:

*number[0] == 33;*
*number[1] == 16;*
*numbers[5] == 39;*

*number[6]* has no defined value. And *number[10]* does not exist.

How can the end of the sequence of values in an array of integers be determined? There are several possibilities.

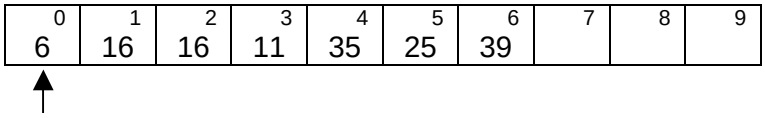
1. use a special number that is not part of the sequence. for example, if you are storing positive integers only, you could use -1 to signal the end of the sequence.

number

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 33 | 16 | 11 | 35 | 25 | 39 | -1 | | | |

2. reserve the first element in the array for a count of the values held

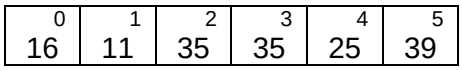
number




number of integers stored

3. fill the array entirely. Then the length of the sequence of numbers is also the size of the array, one more than the last index value.

number




Which strategy we use depends on the situation.

## 15.2 Sum Values in an Array

The next program sums the integers held in an array. Here, we use strategy #2, the first element contains the number of integers to be summed.

One way to give an array its values is to assign them along with the declaration of the array.

```
int array[10] = { 6, 3, 4, 1, 5, 2, 7 };
```

Six is the number of values to be summed. We expect the sum to be 3 + 4 + 1 + 5 + 2 + 7 = 22.

The array is passed to the *sum()* function with

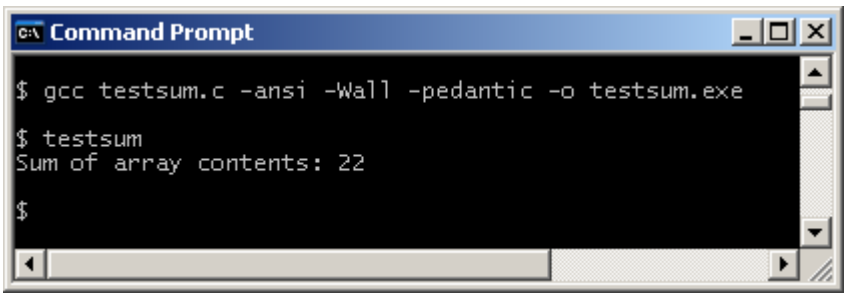
```
sum(array)
```

The *sum()* function receives the array in *(int *array)*. Then, starting at index number 1, we add each value to the *total*.

```
/* sum: returns the sum of the integers in array */
int sum(int *array)
{
  int total = 0;
  int i;

  for (i = 1; i <= array[0]; i++)
    total = total + array[i];
  return total;
}
```

Here is the entire program and its run.

```
/* testsum.c: sums an array of integers */

#include <stdio.h>

/* sum: returns the sum of the integers in array */
int sum(int *array)
{
  int total = 0;
  int i;

  for (i = 1; i <= array[0]; i++)
    total = total + array[i];
  return total;
}

int main()
{
  int array[10] = { 6, 3, 4, 1, 5, 2, 7 };

  printf("Sum of array contents: %d\n", sum(array));
  return 0;
}
```




## 15.3 Arrays as Accumulators

Fifteen students rated the quality of chips from their refectory, on a scale from 0 (appalling) to 5 (brilliant). We need to record the ratings and print a summary of the results.

The scale, 0..5, form the indices of an array.

tally

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 2 | 4 | 5 | 1 | 2 |

The values held in the array represent the number of students who scored each rating. So, for example, 1 student awarded 0 marks for the quality of chips,, two students awarded 1 mark, four students awarded 2 marks, ... Notice that the entire array is filled and so we can use strategy #3.

We start with the *main()* function, which, in a sequence of function calls, outlines *what* needs to be done.

```
int main()
{
  enum { size = 6, nStudents = 15, min = 0, max = 5 };
  int tally[size] = { 0, 0, 0, 0, 0, 0 };

  collectResults(tally, nStudents, min, max);
  printResults(tally, size);
  return 0;
}
```

The *size* of the array is 6; there are six elements indexed 0..5. The number of students, *nStudents*, is 15. *min* is the least rating, *max* is the largest rating.

Initially, each element in the array has the value zero. We want to collect the results from the students and store them in the array *tally*. Then we want to print the results contained in *tally*.

Now we turn to the functions that describe *how* it is done. The function *collectResults()* looks like this:

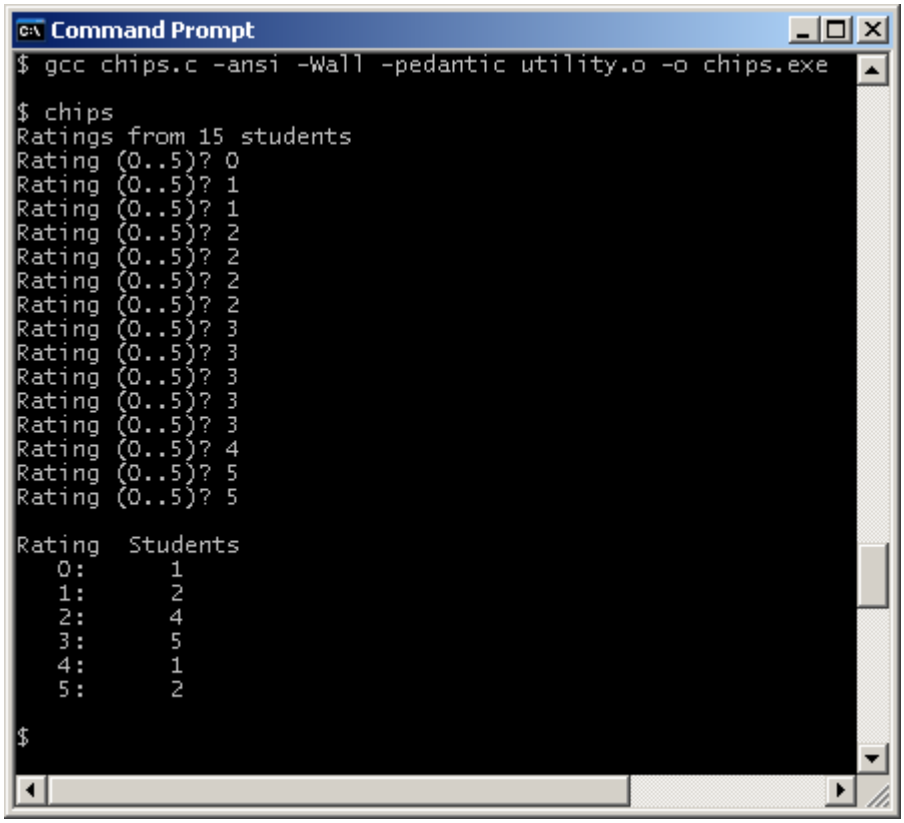
```
/* collectResults: stores nStudents ratings of chips on a scale
   min..max */
int collectResults(int tally[], int nStudents, int min, int max)
{
  int rating, i;

  printf("Ratings from %d students\n", nStudents);
  for (i = 0; i < nStudents; i++) {
    printf("Rating (%d..%d)? ", min, max);
    rating = getInt();
    if (rating >= min && rating <= max)
      tally[rating]++;
  }
  printf("\n");
  return 0;
}
```

We loop for each of the 15 students surveyed, and input each student's rating. Only if the rating is within bounds, i.e. within 0..5, is the element in the array updated. For example, *tally[0]*++ adds 1 to the contents of the array in element zero. *tally[5]*++ adds 1 to the contents in element five of the array. It would be an error to attempt to update an array element that does not exist, hence the guard *if (rating >= min && rating <= max) ...*

The function *printResults()* is straightforward.

Shown below is the complete program and an example run.

```
/* chips.c: analyses students rating of chips */

#include <stdio.h>
#include "utility.h"

/* collectResults: stores nStudents ratings of chips on a scale
   min..max */
int collectResults(int tally[], int nStudents, int min, int max)
{
  int rating, i;

  printf("Ratings from %d students\n", nStudents);
  for (i = 0; i < nStudents; i++) {
    printf("Rating (%d..%d)? ", min, max);
    rating = getInt();
    if (rating >= min && rating <= max)
      tally[rating]++;
  }
  printf("\n");
  return 0;
}

/* printResults: prints summary of students' ratings */
int printResults(int tally[], int size)
{
  int i;

  printf("Rating  Students\n");
  for (i = 0; i < size; i++)
    printf("%4d: %6d\n", i, tally[i]);
  return 0;
}

int main()
{
  enum { size = 6, nStudents = 15, min = 0, max = 5 };
  int tally[size] = { 0, 0, 0, 0, 0, 0 };

  collectResults(tally, nStudents, min, max);
  printResults(tally, size);
  return 0;
}
```




## 15.4 Statistics

As part of the analysis of student's project grades, each lecturer involved in the marking supplies the highest mark and the lowest mark they awarded, along with the number of projects they marked, and the average mark. We write a program that will input a sequence of marks, each in the range 0..100, for an unspecified number of students, and output the required statistics. Each lecturer marks the work of typically ten students, and, in any case, not more than 15. We can use -1 to signal the end of input because -1 is not a valid project mark.

mark

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 55 | 65 | 60 | 35 | 45 | 50 | 55 | -1 | | | | | | | | |

Here, we can see with our eyeballs that the lowest mark is 35, the highest is 65, and that the marks for seven students work are entered.

We start with the *main()* function, which, in a sequence of function calls, outlines *what* needs to be done.

```
int main()
{
  enum { size = 16 };
  int *marks;
  int min, max, nStuds;
  double average;

  marks = getMarks(size);
  min = getMin(marks, size);
  max = getMax(marks, size);
  nStuds = getNStuds(marks, size);
  average = getAverage(marks,size);
  printStats(min, max, nStuds, average);

  return 0;
}
```

The size of the array is fixed as containing 16 elements, indexed 0..15. *marks* is declared as a pointer to *int*. The intention here is to reserve space for the array in the *getMarks()* function. A call to *getMin()* retrieves the least mark, and a call to *getMax()* retrieves the largest mark stored in *marks*. A call to *getNStuds()* gets the number of student marks stored. A call to *getAverage()* returns the average mark. Finally, the statistics are printed with a call to *printStats()*.

Now we turn to the functions that describe *how* it is done.

```
/* getMarks: stores up to size-1 project marks */
int *getMarks(int size)
{
  int *marks = malloc(sizeof(int) * size);
  int i;

  for (i = 0; i < size; i++) {
    printf("Mark (-1 to end)? ");
    marks[i] = getInt();
    if (marks[i] < 0)
      break;
  }
  marks[i] = -1;
  return marks;
}
```

We reserve space for the array with a call to *malloc()*. We loop for as long as there is space in the array to be filled. We invite the user to enter a mark, or to enter -1 if there are no more student marks to be entered. If -1 is entered, we break out of the *for* loop. Finally, we ensure that the sequence of marks stored is terminated with -1 and we return the completed array of marks.

Now, we look at *getMin()*.

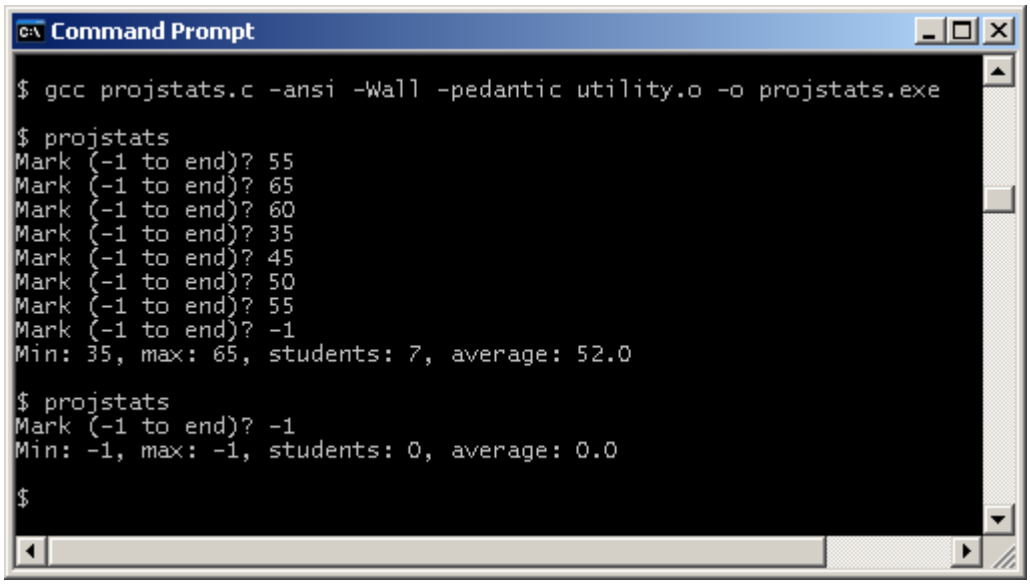
```
/* getMin: returns the least mark in marks */
int getMin(const int *marks, int size)
{
  int min = marks[0];
  int i;

  for (i = 0; marks[i] >= 0 && i < size; i++)
    if (marks[i] < min)
      min = marks[i];
  return min;
}
```

The first mark stored in the array is the lowest mark seen so far. We loop though each value stored in the array until -1 is reached. For each value we check whether it is less the least mark found so far. If it is then that mark becomes the least so far. Finally, we return the least mark.

The program is left unfinished for you to complete.

```
/* projstats.c: inputs project marks, outputs statistics */

#include <stdio.h>
#include <stdlib.h>
#include "utility.h"

/* getMarks: stores up to size-1 project marks */
int *getMarks(int size)
{
  int *marks = malloc(sizeof(int) * size);
  int i;

  for (i = 0; i < size; i++) {
    printf("Mark (-1 to end)? ");
    marks[i] = getInt();
    if (marks[i] < 0)
      break;
  }
  marks[i] = -1;
  return marks;
}

/* getMin: returns the least mark in marks */
int getMin(const int *marks, int size)
{
  int min = marks[0];
  int i;

  for (i = 0; marks[i] >= 0 && i < size; i++)
    if (marks[i] < min)
      min = marks[i];
  return min;
}

/* getMax: returns the largest mark in marks */
int getMax(const int *marks, int size)
{
...
}
```

```
/* getNStuds: returns the number of students */
int getNStuds(const int *marks, int size)
{
...
}

/* getAverage: returns the average mark */
double getAverage(const int *marks, int size)
{
...
}

/* printStats: prints min, max and nStuds */
int printStats(int min, int max, int nStuds, double average)
{
...
}

int main()
{
  enum { size = 16 };
  int *marks;
  int min, max, nStuds;
  double average;

  marks = getMarks(size);
  min = getMin(marks, size);
  max = getMax(marks, size);
  nStuds = getNStuds(marks, size);
  average = getAverage(marks,size);
  printStats(min, max, nStuds, average);

  return 0;
}
```



## Exercise 15.1

1. Complete the *getMax(), getNStuds()*, *getAverage()* and *printStats()* functions of *projstats.c* shown above. Ensure the output is sensible if no marks are entered.

2. Write and test a program that stores the percentage exam marks for n students, and outputs how many scored 0..19, how many scored 20..39, how many scored 40..59,how many scored 60..79, and how many students scored 80..100 marks.

**We have** looked at arrays of numbers. **Next** we continue our study of arrays looking at searching and sorting techniques.

## Bibliography

Kernighan B and Ritchie D *The C Programming Language* Prentice Hall 1988
Mark Williams Company *ANSI C A Lexical Guide* Prentice Hall 1988