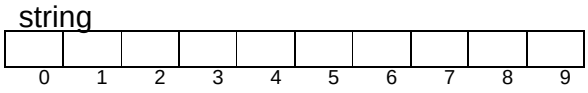
6 ARRAYS

6.1 Introduction

We have already met definitions similar to

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

This defines an object named string which has ten consecutive containers or cells numbered from zero up to nine.

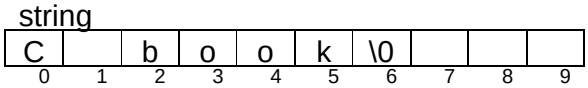


string is an example of an array. An array is a set of contiguous containers or cells, that is, each cell is joined to its neighbour and there are no gaps between them. The number of cells in an array is known as its size. An array with size equal to ten has cells numbered from zero up to nine inclusive.

If we initialised the array *string* thus

```
char string[10] = "C book";
```

then one character is stored in each cell.



\0 represents the end-of-string marker known as the NULL character. It is automatically placed in the array by the compiler when an array of characters is initialised in this way (provided, of course, there is enough room in the array).

Each cell in an array is known as an element. Each element number is known as an index or subscript. Each element of an array is referenced by enclosing its index within square brackets thus

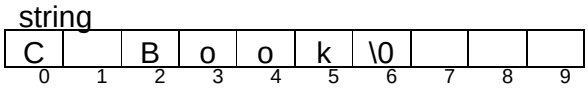
```
string[0], string[1], string[2], ... string[9].
```

The value stored in *string[2]* is *'b'*. *string[5]* contains *'k'*. The content of *string[7]* is undefined. *string[10]* does not exist.

To store the value *'B'* in *string[2]* we write

```
string[2] = 'B';
```

This overwrites the value previously stored in *string[2]*.



All the values stored in an array must be of the same type. In *string* for example, every element contains a value of type *char*. So, *string* is an array of type *char*.

Arrays can store values of other types such as *int*, *float* and enumerated values.

6.2 The Length of a String

In C, a string is a sequence of characters, terminated with the NULL character (denoted by \0). A string may be held in an array of type *char*. The library named *string.h* contains functions which enable us to determine the number of characters actually held in an array of *char* and to copy the contents of one *char* array into another. Let us see how these may be used and implemented.

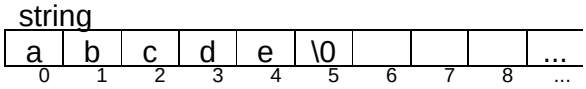
The following program finds the length of a string input by the user.

```
/* program 6.1 - finds the length of a string. */

#include <stdio.h>
#include <string.h>

int main()
{
  char string[BUFSIZ];
  int lengthOfString = 0;

  printf("String? ");
  gets(string);
  lengthOfString = (int)strlen(string);
  printf("Length of string entered is %lu", lengthOfString);
  return 0;
}
```

An example of a program run is

```
String? abcde
Length of string entered is 5
```

abcde would be stored in the array like this



What is stored in elements 6 onwards is not defined. *strlen* counts the number of characters up to (but not including) the first NULL character; this number is known as the length of the string. Let us look at some of the details in program 6.1.

string.h contains the prototype for the function *strlen*. The line

```
lengthOfString = (int)strlen(string);
```

says store the length of *string* in *lengthOfString*. The cast operator *(int)* ensures that a value of the right type (*int*) is assigned to *lengthOfString*.

It is instructive to see how *strlen* might be implemented. Our version of *strlen*, named *stringLength* and shown below, returns the number of characters in a string up to, but not including, the NULL character.

```
int stringLength(char string[])
/* pre-condition:  string contains a NULL-terminated sequence
                   of characters.
   post-condition: returns the number of characters in string.
*/
{
   int length = 0;

   while (string[length] != (char)NULL)
      length++;
   return length;
}
```

If *string* contains no characters (apart from the NULL character) the loop is not executed at all and zero is returned. If string contains two characters, the first being a non-NULL character and the second the NULL character, the loop is executed once and length is incremented (that is, increased by one). One is returned.

6.3 Copying a String

From time to time we need to make a copy of all or part of a string. *strncpy* from the string library copies a number of characters from one string to another. Its use is illustrated in program 6.2 shown below.

```
/* program 6.2 - copies a string. */

#include <stdio.h>
#include <string.h>
```

```
void copyNChars(const char string[], int nChars,
                char newString[]);
/* pre-condition:  string contains a null-terminated sequence
                   of characters, newString is sufficiently
                    large to contain string and NULL,
                    nChars >= NULL
   post-condition: copies nChars from string into newString,
                   appends NULL to newString. */

int main()
{
  char string[BUFSIZ], newString[BUFSIZ];

  printf("String? ");
  gets(string);
  copyNChars(string, 5, newString);
  printf("The new string is %s", newString);
  return 0;
}

void copyNChars(const char source[], int nChars,
                    char destination[])
{
   strncpy(destination, source, nChars);
   destination[nChars] = (char)NULL;
}
```

Three examples of program runs are

(1)
```
String? abcd
The new string is abcd
```

(2)
```
String? abcde
The new string is abcde
```

(3)
```
String? abcdef
The new string is abcde
```

Notice that in (3) above the *f* has not been copied. This is because the five characters must include the end string NULL character.

Inside main, we have defined two arrays of type char.

```
char string[BUFSIZ], newString[BUFSIZ];
```

To pass an array as an argument value to a function parameter, we supply just its name. Here, the arrays *string* and *newString* are passed to *copyNChars*.

```
copyNChars(string, 5, newString);
```

However, what is actually passed is the address of the first element of the array; this is done automatically by C.

In the function declaration

```
void copyNChars(const char source[], int nChars,
                    char destination[])
```

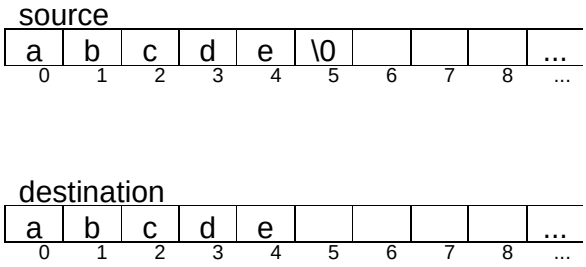
the array parameters are written *char source[]* and *char destination[]*. The empty brackets *[]* indicate that both source and destination are to contain the start address of an array.

Now, if a function parameter contains the address of a variable, then the function can change the contents of that variable. To prevent this from happening, the keyword *const* is used to qualify the parameter. The contents of arrays marked as *const* remain unchanged during function execution.

In the function call

```
strncpy(destination, source, nChars);
```

nChars are copied from *source* into *destination*. Suppose *nChars* contains five. If source contains *"abc"*, then *abc* is copied into destination followed by two NULL characters to make up the five. If source contains *"abcdef"* then just the first five characters are copied into destination; a terminating NULL character is NOT copied into the array, as shown in the following diagram.



Since the terminating NULL character is required by many functions which have string parameters, it is up to us as programmers to ensure that any string we create is properly terminated. That is why we write

```
destination[nChars] = (char)NULL;
```

in

```
void copyNChars(const char source[], unsigned nChars,
                    char destination[])
{
   strncpy(destination, source, nChars);
   destination[nChars] = (char)NULL;
}
```

Here we have used *nChars* to represent a number of characters AND a cell number; they are not the same thing. We need to place the terminating NULL character in the element numbered 5 in the destination array.

The destination array must be sufficiently large to contain the expected number of characters, otherwise, the behaviour of *strncpy* is undefined.

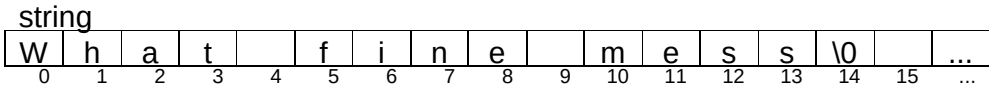
To display the contents of an array of *char*, ie a string, *%s* is used as the conversion specification in *printf*.

```
printf("The new string is %s", newString);
```

6.4 Testing a Character

Since we are dealing with arrays of characters, it may be reasonable at this point to examine characters a little more closely.

Sometimes we need to know whether a character is a letter or a space or a digit. Let us design a function which will count the number of spaces in a string.



Essentially, the method is: move through the array from its beginning up to the first NULL character inspecting the contents of each cell as you go; if the value stored in a cell is a space, then increment a number-of-spaces counter. Here is the function.

```
int numberOfSpacesInString(const char string[])
{
  int spaceCount = 0;
  int i = 0;

  while (string[i] != (char)NULL) {
    if (isspace(string[i]))
      spaceCount++;
    i++;
   }
   return spaceCount;
}
```

i refers to each element number in the array. It starts off with the value zero. Then, for as long as the character stored in the location numbered *i* is not the NULL character, if the location contains the space character, then increment *spaceCount*; move onto the next location. Finally, return the value stored in *spaceCount*.

isspace is defined in *ctype.h*. *isspace* returns one (ie TRUE) if its *char* argument is the space character, otherwise is returns zero (ie FALSE).

Here is a complete program which uses the function.

```
/* program 6.3 - counts spaces in a string. */
#include <stdio.h>
#include <ctype.h>

int numberOfSpacesInString(const char string[]);
/*  pre:  string is a null-terminated sequence of characters.
    post: returns the number of spaces in string. */

int main()
{
  char string[BUFSIZ];
  int nSpaces = 0;

  printf("String? ");
  gets(string);
  nSpaces = numberOfSpacesInString(string);
  printf("The number of spaces in the string is %d", nSpaces);
  return 0;
}

int numberOfSpacesInString(const char string[])
{
  int spaceCount = 0;
  int i = 0;
  while (string[i] != (char)NULL) {
    if (isspace(string[i]))
      spaceCount++;
    i++;
   }
   return spaceCount;
}
```

An example of a program run is

```
String? What a fine mess you have got me into.
Number of spaces in the string is 8
```

isspace is a function which returns TRUE if its character argument is the space character, otherwise, it returns FALSE. Other functions which work in a similar way are *isalpha* (checks whether a character is a letter), *isdigit* (checks whether a character is a digit) and *islower* (checks whether a character is a lower case character). The prototypes for these functions are found in *ctype.h*.

6.5 The Case Sensitivity Problem

Also defined in *ctype* is the function *toupper*. *toupper* converts a lower case character to its upper case equivalent. Here is its specification.

```
int toupper(int ch);
/* post-condition: returns upper case equivalent of ch if ch
   is a lower case letter, otherwise returns ch unchanged. */
```

Notice that the implied pre-condition is that the argument value is an *int* and that the function

returns an *int*. In C, values of type *char* can be stored in *int* variables. This is necessary because some character values, such as the end-of-file character, cannot be stored in a *char* variable. But this need not concern us here.

C is case sensitive. So the two string literals *Smith* and *SMITH* are not the same. In some circumstances we would want *Smith* and *SMITH* to represent the same item. So we need a function which will convert all the lower case characters in a string to upper case. Here is the specification of such a function.

```
void convertStringToUpperCase(const char string[],
                      char upperCaseString[]);
/* pre-condition:  string is a NULL-terminated sequence of
                   characters.
   post-condition: upperCaseString' contains the upper case
                   equivalent of string. */
```

An example of a call to this function is

```
char name[BUFSIZ], upperCaseName[BUFSIZ];

readString("Enter name: ", name);
convertStringToUpperCase(name, upperCaseName);
```

The implementation of *convertStringToUpperCase* is straightforward. Go through *string* from beginning to end: for each character in *string*, obtain its upper case equivalent and place it in *upperCaseString*.

```
void convertStringToUpperCase(const char string[],
                char upperCaseString[])
{
  int i = 0;

  while (string[i] != NULL) {
    upperCaseString[i] = (char)toupper(string[i]);
    i++;
  }
  upperCaseString[i] = NULL;
}
```

Exercise 6.1

1 Many telephone numbers contain eleven digit characters. Spaces are allowed between groups of digits and sometimes a group of digits is contained within brackets. Write a function which has a string argument and returns whether (or not) the string contains eleven digits.

2 A telephone number is perhaps best stored in two arrays, one for the area code and one for the number. Write a function which receives a full telephone number in one parameter and returns the number in its two constituent parts in two other parameters.

3 Write a program to test the *convertStringToUpperCase* function described in

section 6.5 above.

4 Design, write and test a function named *stringEqual*. The function is to have two string parameters and is to return one if the two parameter values are the same; otherwise it is to return zero. The function should use the *strcmp* function defined in the string library. Its specification is

```
int strcmp(const char s1[], const char s2[]);
/* Compares two strings character by character until
either two characters are found which are not identical
or the end of a string is reached.  Returns a number less
than zero if the first character of the pair comes before
the second one in "character table order", returns a
number greater than zero if the first character of the
pair comes after the second one, returns zero if the two
strings are identical. */
```

A commonly used "character table" is the American Standard Code for Information Interchange (ASCII) table. Part of this table is shown below.

<space> 0 1 2 3 4 5 6 7 8 9
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z

So, for example, *A* comes before *B* in ASCII table order sequence but *a* does not come before *B*.

The ASCII collating (ie character ordering) sequence is similar to, but not the same as, alphabetical or dictionary order. In dictionary order, the letters would be ordered like this

A a B b C c D d ...

So perhaps the argument values to *strcmp* should be upper-case strings. An example of a call to *strcmp* is

```
char string1[BUFSIZ], string2[BUFSIZ];
char upperCaseString1[BUFSIZ], upperCaseString2[BUFSIZ];

readString("String? ", string1);
readString("Another string? ", string2);
convertStringToUpperCase(string1, upperCaseString1);
convertStringToUpperCase(string2, upperCaseString2);

if (strcmp(upperCaseString1, upperCaseString2)) == 0)
  printf("Equal\n");
else if strcmp(upperCaseString1, upperCaseString2)) < 0)
  printf("First string comes before the second.\n");
else if strcmp(upperCaseString1, upperCaseString2)) > 0)
  printf("first string comes after the second.\n");
```

5 Write a function which will return TRUE if the first of its two string arguments comes before the second in dictionary order, or FALSE if otherwise. So, for

example, if the function was named *stringLessThan*

```
stringLessThan("apple", "zoo") == TRUE;
stringLessThan("zoo", "apple") == FALSE;
```

6 Nearly every new book published is assigned an International Standard Book Number (ISBN); no two books have the same ISBN. An ISBN contains ten digits; the last one is the check digit. An example of an ISBN is 0131103628. In this example, the check digit is 8.

The check digit is calculated from the first nine digits using the modulus 11 weighted method, as shown in the following example.

First, each digit is multiplied by its weight starting with 10.

partISBN	0	1	3	1	1	0	3	6	2
weight	x10	x9	x8	x7	x6	x5	x4	x3	x2
partSum	0	9	24	7	6	0	12	18	4

Then the part sums are totalled.

total = 0 + 9 + 24 + 7 + 6 + 0 + 12 + 18 + 4 = 80

Then the total is divided by 11 to obtain the remainder.

80 / 11 = 7 remainder 3

The remainder is subtracted from 11 to obtain the checkDigit.

checkDigit = 11 - 3 = 8

If the checkDigit turns out to be 10, then X is used to represent it.

Write a program which contains the following functions:

```
void getISBN(char ISBN[]);
/* post-condition: ISBN' contains a NULL terminated ISBN
                   number entered at the keyboard.  */

char checkDigitFromISBN(const char ISBN[]);
/* pre-condition:  ISBN is a NULL-terminated sequence of
                   digit characters.
   post-condition: returns the last digit (or letter)
                   character contained in ISBN. */

void getPartISBNFromISBN(const char ISBN[],
                         char PartISBN[]);
/* pre-condition:  ISBN contains a NULL terminated ISBN
                   number.
   post-condition: PartISBN' contains all the digit
                   characters contained in ISBN except
                   the check digit character. */
```

```
char calculatedCheckDigit(const char PartISBN[]);
/* pre-condition:  PartISBN contains all the digit
                   characters in an ISBN except for the
                   check digit character.
   post-condition: returns the check digit calculated
                   from the PartISBN using the modulus 11
                   weighted method. */

int checkDigitsAreEqual(char calculatedCheckDigit,
                                char theCheckDigit);
/* post-condition: returns TRUE if calculatedCheckDigit =
                   theCheckDigit, otherwise, returns
                   FALSE. */
```

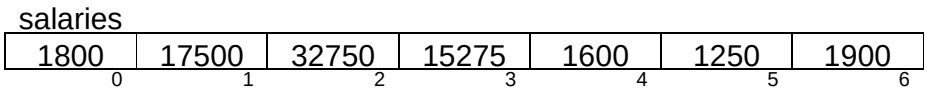
Your program should input ten characters to represent an ISBN number. The program should then calculate, from the first nine digit characters entered, what the check digit should be. Then your program should compare the check digit entered with the one calculated; if they are different the program should output "The number entered is not a valid ISBN", otherwise, the program should output "ISBN is OK". You might find the following two functions useful

```
unsigned digitFromCharacter(char ch)
/* pre-condition:  char is a digit character.
   post-condition: returns the number equivalent of ch.
                   If ch contains the character '8' then
                   digitFromCharacter returns the integer
                   8. The normal arithmetic operations
                   can be performed on unsigned int
                   values. (We do not normally perform
                   arithmetic with chars.)  An example of
                   a call to this function is
     integerDigit = digitFromCharacter(partISBN[index]);
*/
{
   return (unsigned)ch - '0';
}

char characterFromDigit(unsigned digit)
/* pre-condition:  digit is one of 0,1,2,3,4,5,6,7,8,9
   post-condition: returns the character equivalent of
                   digit. characterFromDigit is the
                   converse of digitFromCharacter.
                   An example of a call to this function
                   is
    checkDigitCharacter = characterFromDigit(checkDigit);
*/
{
   return (char)digit + '0';
}
```

7 Design, write and test a function which will convert an integer number into a string.

6.5 Arrays of Numbers

A financial director wishes to know the largest salary in a group of salaries. Suppose the salaries are contained in an array.



How can we find the largest value in the array? We look at the contents of the first cell; it is the largest salary seen so far. Then we look at the contents of each of the following cells in turn: if we find a salary which is larger than the one seen so far, then that salary becomes the largest seen so far.

```
double largestSalaryInSalaries(double salaries[],
               int lastIndex)
{
  int i = 0;
  double largestSalary = salaries[i];

  while (i <= lastIndex) {
    if (salaries[i] > largestSalary)
      largestSalary = salaries[i];
    i++;
  }
  return largestSalary;
}
```

We are obliged to let the *largestSalaryInSalaries* function know what the last array index value is because, otherwise, it would have no way of knowing where the array ends. It would be an error if we attempted to refer to an element which does not exist. So, for example, we do not use index *i* if it contains a value greater than *lastIndex*.

The next program, program 6.4, uses the *largestSalaryInSalaries* function.

```
/* program 6.4 - finds the largest value in an array of
                 numbers. */

#include <stdio.h>

void buildScenario(double salaries[]);
/* post-condition: salaries contains a sequence of salary
                   values. */

double largestSalaryInSalaries(const double salaries[],
                    int lastIndex);
/* pre-condition: lastIndex is the last index value in
                  salaries.
   post-condition: returns the largest value in salaries.
*/

void showLargestSalary(double salary);
```

```
int main()
{
  enum { lastIndex = 6, arraySize = 7 };

  double salaries[arraySize];
  double largestSalary;

  buildScenario(salaries);
  largestSalary = largestSalaryInSalaries(salaries,
                    lastIndex);
  showLargestSalary(largestSalary);
  return 0;
}

void buildScenario(double salaries[])
{
  salaries[0] = 18000;
  salaries[1] = 17500;
  salaries[2] = 23750;
  salaries[3] = 14275;
  salaries[4] = 16000;
  salaries[5] = 12500;
  salaries[6] = 19000;
}

double largestSalaryInSalaries(const double salaries[],
                    int lastIndex)
{
  int i = 0;
  double largestSalary = salaries[i];

  while (i <= lastIndex) {
    if (salaries[i] > largestSalary)
      largestSalary = salaries[i];
    i++;
  }
  return largestSalary;
}

void showLargestSalary(double salary)
{
   printf("The largest salary is £%0.2f", salary);
}
```

When the program is executed

```
The largest salary is £23750.00
```

is shown on the screen.

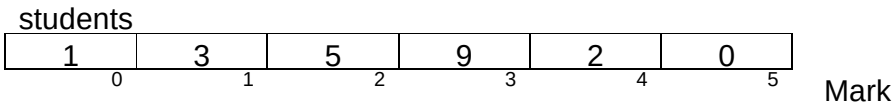
Let us look at *buildScenario* shown above. In *buildScenario* we, as programmers, are storing explicit values in the array *salaries*. This means that these values do not have to be entered every time the program is run.

6.6 Arrays As Accumulators

Twenty students were asked to rate the quality of food supplied by their refectory on a scale from zero (appalling) to five (excellent). The results are summarised in the following table:

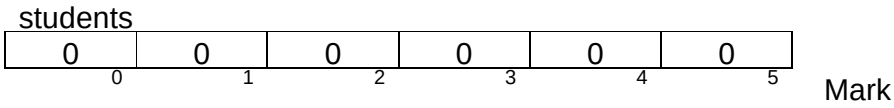
Number of students who awarded	0 marks:	1
	1 mark:	3
	2 marks:	5
	3 marks	9
	4 marks:	2
	5 marks:	0

Out of the twenty students, only one gave zero marks for quality, three gave one mark for quality, five gave two marks, and so on. Can we use an array to hold the results of the poll? Yes we can, thus

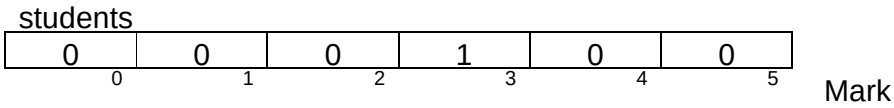


The index values 0, 1, 2, 3, 4, and 5 represent each of the possible ratings of food quality. One student gave zero marks for food quality, three students gave one mark and five students gave two marks, and so on.

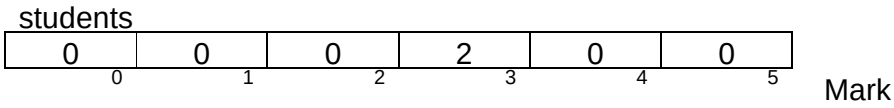
Initially, before the poll starts, no students have given any marks for the quality of food.



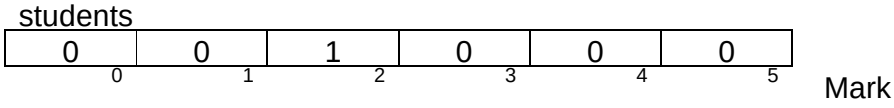
Suppose the first student polled awarded 3 marks. Then we need to increment *students[3]*.



Suppose now that the second student polled also gave three marks for food quality. Then, again, we need to increment *students[3]*.



And again. Suppose the next student awarded 2 marks, then we need to increment *students[2]*.



So now we have a method for accumulating the marks awarded by the students for food quality. The function to accomplish the task is very simple:

```
void incrementStudentsWhoGaveMark(int mark, int students[])
{
  students[mark]++;
}
```

The line

```
students[mark]++;
```

increases the value contained in *students[mark]* by 1. So, if *mark* contained 2 then the array cell with index 2 has its value incremented. Here is the complete program.

```
/* program 6.5 - analyses marks awarded for food quality. */
#include <stdio.h>

void incrementStudentsWhoGaveMark(int mark, int students[]);
/* pre-condition:  mark is within the bounds of students.
   post-condition: students'[mark] = students[mark] + 1   */

void initialiseArray(int students[], int lastIndex);
/* post-condition: every element in students contains zero. */

int intNumberRead(char prompt[]);
/* post-condition: returns the number entered at the keyboard.
*/

int markRead(int min, int max);
/* post-condition: returns a value between min and max
          inclusive. */

void showResults(const int students[], int lastIndex);

int main()
{
  enum { minMark = 0, maxMark = 5, lastIndex = 5,
              arraySize = 6,studentsPolled = 20 };
  int students[arraySize];
  int mark = 0;
  int studentsProcessed = 0;

  initialiseArray(students, lastIndex);
  while (studentsProcessed < studentsPolled) {
    mark = markRead(minMark, maxMark);
    incrementStudentsWhoGaveMark(mark, students);
    studentsProcessed++;
  }
  showResults(students, lastIndex);
}
```

```
void incrementStudentsWhoGaveMark(int mark, int students[])
{
  students[mark]++;
}

void initialiseArray(int students[], int lastIndex)
{
  int i = 0;
  while (i <= lastIndex) {
    students[i] = 0;
    i++;
  }
}

int intNumberRead(char prompt[])
{
  char string[BUFSIZ];
  int number = 0;

  printf("%s", prompt);
  gets(string);
  sscanf(string, "%d", &number);
  return number;
}

int markRead(int min, int max)
{
  int mark = intNumberRead("Mark? ");

  while ((mark < min) || (mark > max)) {
    printf("The mark must lie between ");
    printf("%d and %d inclusive.\n", min, max);
    mark = intNumberRead("mark? ");
  }
  return mark;
}

void showResults(const int students[], int lastIndex)
{
  int mark = 0;

  printf("The results of the survey are: \n\n");
  printf("Grade   Number of Students\n\n");
  while (mark <= lastIndex) {
    printf("  %d            %d\n", mark, students[mark]);
    mark++;
  }
}
```

Exercise 6.2

1. Test program 6.5 shown above.

2. Create an array which contains the annual salary of about five employees.

Then write and test functions which will find (a) the range of salaries (where we define range as being the largest salary minus the smallest salary) and (b) the average salary.

3 Create an array which contains the number of days in each month for a non leap year; store zero in the first array element so that index values ranging from one to twelve can be used to represent the month numbers.

daysInMonth

0	31	28	31	30	31	30	31	31	30	31	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12

month

Then write a function (or functions) which will input a date in the form *dayNumber, monthNumber*.

An array which contains the days in each month can be used to check whether a *dayNumber* and a *monthNumber* can be a valid combination. For example, if day contains 31 and month contains 4 then day cannot be in month because there are only thirty days in April. Write and test a function to the following specification

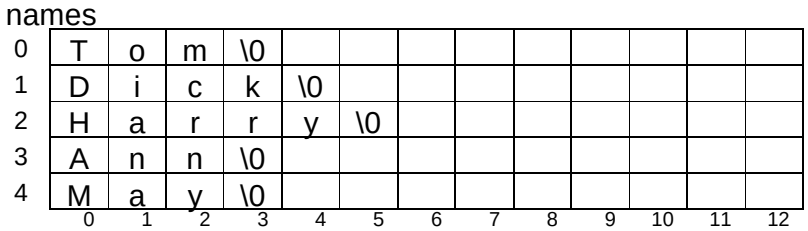
```
int dayIsInMonth(int day, int month,
                         const int daysInMonth[]);
/* pre-condition : month is in 0..12,
          daysInMonth contains the days in each month and
          daysInMonth[0] contains zero.
   post-condition: returns 1 if
     Day <= daysInMonth[month]; otherwise returns 0. */
```

Now write and test a function which will convert a day in a month to a day in the year. For example, 1st January would be day 1 in the year, 2nd January would be day 2, 3rd January day 3, ... , 31st January day 31, 1st February day 32, ... and 31st December would be day 365 in the year. The 7th March would be day 66 (66 = 31 + 28 + 7) in the year. A specification for this function might be

```
int dayInYearNumber(int day, int month,
                         const int daysInMonth[]);
/* pre-condition: dayIsInMonth(day, month)
                 daysInMonth contains the days in each
                 month and daysInMonth[0] contains zero.
   post: returns day-in-year-number for day in month. */
```

6.7 Arrays of Strings

Suppose we need to store some names in an array. For example, suppose we need to store up to five names and that we allow up to twelve characters per name, then we need an array of five elements in which each element is itself an array of thirteen elements.



We could define the type *String* as an array of thirteen character elements.

```
typedef char String[13];
```

And then define the array of names as

```
String names[5];
```

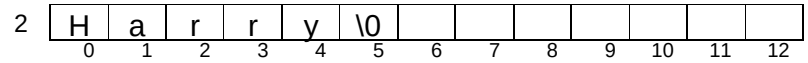
This definition says that *names* is an array of five *String* elements, where a *String* element is itself an array of 13 character elements. To print the contents of the array of strings is easy.

```
void printNames(String names[], int lastIndex)
{
  int i = 0;
  while (i <= lastIndex) {
    printf("%s \n", names[i]);
    i++;
  }
}
```

And to fill the array with names is just as easy.

```
void getNames(String names[], int lastIndex)
{
  int i = 0;

  while (i <= lastIndex) {
    printf("Name? ");
    gets(names[i]);
    i++;
  }
}
```

If *i* contains 2 and *names[i]* contains *Harry\0* then we can imagine the situation as shown in the picture below.



names[2][0] contains '*H*', *names[2][4]* contains '*y*' and *names[2][5]* contains '*\0*'.

Here is the complete program.

```
/* program 6.6 - arrays of fixed length strings. */
#include <stdio.h>
#include <string.h>

enum { stringSize = 13 };

typedef char String[stringSize];

void getNames(String names[], int lastIndex);
/* post-condition: every element in names contains a value
                   input by the user at the keyboard. */

void printNames(String names[], int lastIndex);
/* pre-condition: each element in names is a NULL-terminated
                  sequence of characters */

int main()
{
  enum { lastIndex = 4, arraySize = 5 };
  String names[arraySize];

  getNames(names, lastIndex);
  printNames(names, lastIndex);
  return 0;
}

void getNames(String names[], int lastIndex)
{
  int i = 0;

  while (i <= lastIndex) {
    printf("Name? ");
    gets(names[i]);
    i++;
  }
}

void printNames(String names[], int lastIndex)
{
  int i = 0;

  printf("The contents of the array are: \n");
  while (i <= lastIndex) {
    printf("%s \n", names[i]);
    i++;
  }
}
```

An example of a program run is

```
Name? Tom
Name? Dick
Name? Harry
Name? Ann
Name? May
The contents of the array are:
Tom
Dick
Harry
Ann
May
```

The problem with arrays of fixed length strings is that they use a lot of storage space; much of this space is wasted because we try to ensure that an array is large enough to contain any expected value. Since storage space is finite, we need a strategy for using only as much space as each individual string value needs. Let us see how this may be done.

First, we define a string type as a pointer to a *char*. (Remember that a pointer is a variable which stores an address.)

```
typedef char *String;
```

This allows us to define a variable of type *String* thus:

```
String name;
```

name is variable which is to contain the address of a single *char*. But we need to refer to a sequence of characters. So we define

```
char characterArray[BUFSIZ];
```

And then use *gets* as before to input the user-chosen string value.

```
printf("Name? ");
gets(characterArray);
```

Now we can determine how much storage is required for the value stored in *characterArray*.

```
storageRequired = (unsigned)strlen(characterArray) + 1;
```

(+ 1 for the end of string character, '\0'). And then claim storage of just the right size to contain the contents of *characterArray* and place the start address of this store in name.

```
name = (String)malloc(storageRequired);
```

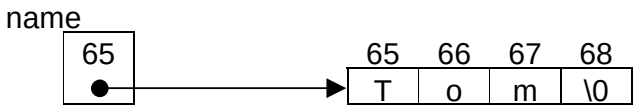
malloc allocates memory or storage. Here, it returns the address of a block of storage of size *storageRequired*. The cast operator *(String)* ensures that *malloc* returns storage suitable for values of type *String*.

Finally, we copy the contents of *characterArray* into the area pointed to by *name*.

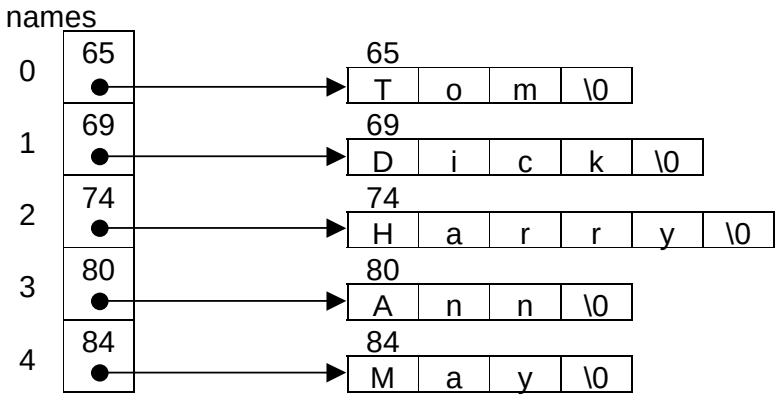
```
strcpy(name, characterArray);
```

strcpy copies every character in *characterArray*, including the NULL character.

If the value stored in *characterArray* is *Tom\0* and *name* contains 65 (say), that is, the start address of the region which is large enough to contain the *Tom\0*, then we can imagine the picture to be thus



If we had an array of such pointers to strings



then we use only as much storage as is required. The address at which each string is stored is determined automatically by the C programming system via the memory allocation function, *malloc*. *malloc* is found in *stdlib.h*.

Here is the complete program.

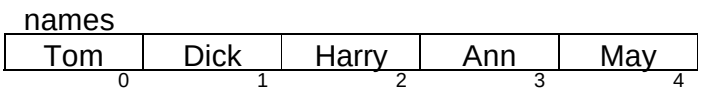
```
/* program 6.7 - arrays of variable length strings. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef char *String;

void getNames(String names[], int lastIndex);
/* post-condition: every element in names contains a
                   NULL-terminated string value, lastIndex is
                   the highest element number in names. */

void printNames(const String names[], int lastIndex);
/* pre-condition: every element in names contains a
                  NULL-terminated string value. */
```

```
int main()
{
  enum { lastIndex = 4, arraySize = 5 };

  String names[arraySize];

  getNames(names, lastIndex);
  printNames(names, lastIndex);
  return 0;
}

void getNames(String names[], int lastIndex)
{
  char characterArray[BUFSIZ];
  String name;
  int storageRequired;
  int i = 0;

  while (i <= lastIndex) {
    printf("Name? ");
    gets(characterArray);
    storageRequired = (int)strlen(characterArray) + 1;
    name = (String)malloc(storageRequired);
    strcpy(name, characterArray);
    names[i] = name;
    i++;
  }
}

void printNames(const String names[], int lastIndex)
{
  int i = 0;

  while (i <= lastIndex) {
    printf("%s \n", names[i]);
    i++;
  }
}
```

6.8 Searching

Searching is the process of looking for an item. Common applications of searching include looking up the balance in a person's bank account, looking for a patient's hospital records and attempting to find the best candidate for a vacancy.

We illustrate the principles of the searching process by looking for a particular name in an array of names. Suppose the array of names is



and the name we are looking for is *Harry*. Basically, we look at each name in the array in turn; if a name in the array matches the one we are looking for, then we terminate the search.

If we get to the end of the array without having found a match, then we conclude the name we want is not in the array. Here is the function definition.

```
int nameIsInArray(const String name, const String names[],
                          int lastIndex);
/* pre-condition: names is initialised with pointers to NULL-
                 terminated character sequences, lastIndex is
                 the highest index of array names name is a
                 NULL-terminated value.
   post-condition: returns 1 (ie TRUE) if name is in names,
                 otherwise, returns 0 (ie FALSE). */
```

And here is the implementation.

```
int nameIsInArray(const String name, const String names[],
                    int lastIndex)
{
  enum { false = 0, true = 1, identical = 0 };

  int i = 0;
  while (i <= lastIndex) {
    if (strcmp(names[i], name) == identical)
      return true;
    i++;
  }
  return false;
}
```

We look at the i'th element in the array. (*i* can be any value from zero up to four in this example.) If we compare the i'th element with the item we are looking for (*strcmp* compares strings and returns zero if they are identical) and find that they are identical, we return true (*name* is in *names*). However, if we reach the end of the array without returning, it is because *name* is not in *names*; we return false.

Here is a program which uses the search function.

```
/* program 6.8 - searches for a particular name in an array of
names. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef char *String;

void buildNames(String names[]);
/* post-condition: returns an array in which each element
                   points to a NULL-terminated string value.*/

int nameIsInArray(const String name, const String names[],
                          int lastIndex);
/* pre-condition: every element in names is initialised with a
                 pointers to NULL-terminated character
                 sequence lastIndex is the highest index of
                 array names name is a NULL-terminate value.
```

```
  post-condition: returns 1 (ie TRUE) if name is in names,
                  otherwise, returns 0 (ie FALSE).     */

String stringRead(char prompt[]);
/* post-condition: returns a pointer to a string entered at
                   the keyboard. */
int main()
{
  enum { lastIndex = 4, arraySize = 5 };
  String names[arraySize];
  String name = stringRead("Search for which name? ");
  buildNames(names);
  if (nameIsInArray(name, names, lastIndex))
    printf("Found.\n");
  else
    printf("Name not found.\n");
  return 0;
}

void buildNames(String names[])
{
  names[0] = "Tom";
  names[1] = "Dick";
  names[2] = "Harry";
  names[3] = "Ann";
  names[4] = "May";
}

int nameIsInArray(const String name, const String names[],
                    int lastIndex)
{
  enum { false = 0, true = 1, identical = 0 };
  int i = 0;
  while (i <= lastIndex) {
    if (strcmp(names[i], name) == identical)
      return true;
    i++;
  }
  return false;
}

String stringRead(char prompt[])
{
  char characterArray[BUFSIZ];
  int storageRequired;
  String string;

  printf("%s", prompt);
  gets(characterArray);
  storageRequired = (unsigned)strlen(characterArray) + 1;
  string = (String)malloc(storageRequired);
  strcpy(string, characterArray);
  return string;
}
```

Three examples of program runs are

(1)
```
Search for which name? Tom
Found.
```

(2)
```
Search for which name? May
Found.
```

(3)
```
Search for which name? Mary
Name not found.
```

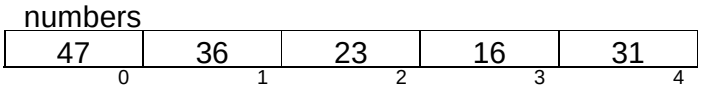
Incidentally, the function *stringRead* is interesting. You remember that a function could only return a value of a fundamental type such as a *char*, *int* or *double*? Well, a function can also return an address. Since *String* is defined to be a pointer to a *char*, values of type *String* can be returned by a function. Hence the function prototype

```
String stringRead(char prompt[]);
```

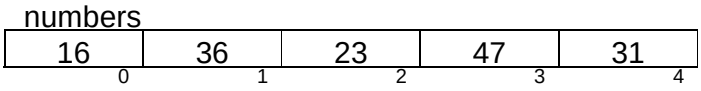
is valid.

6.9 Sorting

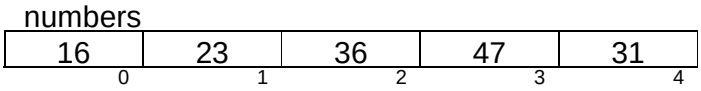
Sorting is the process of placing items (such as peoples names) into some kind of order (such as alphabetical order). We use an array of integers to illustrate the principles. We aim to sort the array shown below into ascending numerical order, namely 16 23 31 36 47.



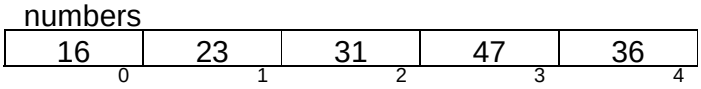
The location for the smallest value in numbers is zero; the value 47 is stored there. The smallest value is found in location 3. So we exchange the position of these two values.



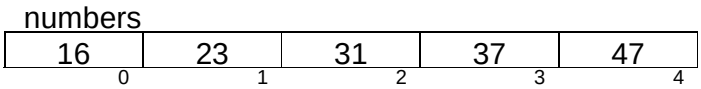
The location for the next smallest value is now one; the value 36 is stored there. We search from this position onwards for the next smallest value: it is 23 in location 2. We exchange the positions of these two values.



The location for the next smallest value is now two; the value 36 is stored there. We search from this position onwards for the next smallest value: it is 31 in location 4. We exchange the positions of these two values.



The location for the next smallest value is now three; the value 47 is stored there. We search from this position onwards for the next smallest value: it is 36 in location 4. We exchange the positions of these two values.



The contents of *numbers* is now in ascending numerical order. This sorting method, known as the insertion sorting algorithm, is used in the *sortIntArray* function shown below.

```
void sortIntArray(int numbers[], int lastIndex)
{
  int smallest = 0;
  int locationContainingSmallest = 0;
  int locationForNextSmallest = 0;

  while (locationForNextSmallest < lastIndex) {
    smallest = smallestInArray(
               numbers, locationForNextSmallest, lastIndex);
    locationContainingSmallest =
               locationOfSmallest(numbers, smallest,
               locationForNextSmallest, lastIndex);
    swap(&numbers[locationForNextSmallest],
         &numbers[locationContainingSmallest]);
    locationForNextSmallest++;
  }
}
```

Program 6.9 shown below sorts an array of integers into ascending order.

```
/* program 6.9 - selection sort on an array of integers. */
#include <stdio.h>

void buildIntArray(int numbers[]);
/* post-condition: every element in numbers contain a value.
*/

int locationOfSmallest(const int numbers[], int smallest,
               int startIndex, int lastIndex);
/* pre-condition: smallest is in numbers,
                  startIndex <= lastIndex,
                  lastIndex is highest index in numbers.
   post-condition: returns location of smallest in numbers.
*/
```

```
int smallestInArray(const int numbers[], int startIndex,
                int lastIndex);
/* pre-condition: startIndex <= lastIndex,
                  lastIndex is highest index in numbers.
   post-condition: returns smallest value stored in array
                  between startIndex and lastIndex inclusive
*/

void sortIntArray(int numbers[], int lastIndex);
/* pre-condition: lastIndex is the highest index in numbers.
   post-condition: items in numbers are in ascending numerical
                  order */

void swap(int *pX, int *pY);
/* post-condition: *pX' = *pY,
                   *pY' = *pX  */

void printIntArray(int numbers[], int lastIndex);
/* pre-condition: lastIndex is the highest index in numbers.
*/

int main()
{
  enum { lastIndex = 4, arraySize };
  int numbers[arraySize];
  buildIntArray(numbers);
  sortIntArray(numbers, lastIndex);
  printIntArray(numbers, lastIndex);
  return 0;
}

void buildIntArray(int numbers[])
{
  numbers[0] = 47;
  numbers[1] = 36;
  numbers[2] = 23;
  numbers[3] = 16;
  numbers[4] = 31;
}

int locationOfSmallest(const int numbers[], int smallest,
                       int startIndex, int lastIndex)
{
  int i = startIndex;

  while (i <= lastIndex) {
    if (smallest == numbers[i])
      return i;
    i++;
  }
  return -1;  /* error */
}
```

```
int smallestInArray(const int numbers[], int startIndex,
                    int lastIndex)
{
  int i = startIndex;
  int smallest = numbers[i];

  while (i <= lastIndex) {
    if (numbers[i] < smallest)
      smallest = numbers[i];
    i++;
  }
 return smallest;
}

void sortIntArray(int numbers[], int lastIndex)
{
  int smallest;
  int locationContainingSmallest;
  int locationForNextSmallest = 0;

  while (locationForNextSmallest < lastIndex) {
    smallest = smallestInArray(
               numbers, locationForNextSmallest, lastIndex);
    locationContainingSmallest =
               locationOfSmallest(numbers, smallest,
               locationForNextSmallest, lastIndex);
    swap(&numbers[locationForNextSmallest],
         &numbers[locationContainingSmallest]);
    locationForNextSmallest++;
  }
}

void swap(int *pX, int *pY)
{
  int hold = *pX;

  *pX = *pY;
  *pY = hold;
}

void printIntArray(int numbers[], int lastIndex)
{
  int i = 0;

  while (i <= lastIndex) {
    printf("%d  ", numbers[i]);
    i++;
  }
}
```

When run, this program displays

```
16   23   31   36   47
```

on the screen.

There are a few points to note. The specification for function *swap* is

```
void swap(int *pX, int *pY);
/* post-condition: *pX' = *pY,
                   *pY' = *pX  */
```

pX and *pY* contain the addresses of two different *int* variables. The post-condition

**pX' = *pY*

says that, on exit from the function, the value stored at the address contained in *X* is the value stored at the address contained in *Y* on entry to the function.

The post-condition

```
*pY' = *pX
```

says that, on exit from the function, the value stored at the address contained in *Y* is the value stored at the address contained in *X* on entry to the function.

The effect of the function is to exchange the values stored in two variables. An example of a call to this function is

```
swap(&numbers[locationForNextSmallest],
     &numbers[locationContainingSmallest]);
```

Here, the values stored in *numbers[locationForNextSmallest]* and *numbers[locationContainingSmallest]* would be exchanged. The function's implementation is straightforward.

```
void swap(int *pX, int *pY)
{
  int hold = *pX;

  *pX = *pY;
  *pY = hold;
}
```

hold preserves the value stored in **pX* before it is overwritten with the value stored in **pY*.

The next point to note is the enumeration in *main*.

```
enum { lastIndex = 4, arraySize };
```

If an explicit value is not assigned to an enumeration constant, C automatically assigns the next number in sequence to it. So here, for example, C assigns the value 5 to *arraySize*. This is useful because if we were to change the size of the array, then we need only change one value - the value of *lastIndex*; C takes care of the *arraySize* for us.

6.10 Programming Principles

Do not refer to elements of an array which do not exist. For example, do not use an index value which is outside the bounds of an array. The bounds of an array are its first and last element numbers and the first element is always numbered zero.

Always ensure that any string values you create are terminated with the NULL character, otherwise, string handling functions may not perform as you expect them to. For example, *printf* may print garbage if you expect it print a sequence of characters that has not been properly terminated.

The first and last elements in an array represent its boundaries. Therefore, when devising test plans which involves searching through an array for example, remember to include as search values the values contained in the first and last elements.

Always initialise a pointer with an address.

Exercise 6.3

1. Devise and test a function which will sort an array of string values into alphabetical order.