7 STRUCTURES

7.1 Introduction

Let us suppose that a person's bank account contains the following details: their account number, their name and the balance left in their account. We can describe these attributes in a single structure.

```
typedef struct {
   String number;
   String name;
   double balance;
}  BankAccount;
```

Here, we have defined a *struct* type and named it *BankAccount*. *struct* is a C keyword. It introduces a structure declaration. The variables *name*, *number* and *balance* are the component parts of the structure named *BankAccount*. The component parts of a structure are known as members.

Since we have declared a type, we can define a variable to hold values of this type.

```
BankAccount bankAccount;
```

How can we place some values into this variable? We could write

```
bankAccount.balance = 375.72;
```

So, to refer to a member of a particular structure, we use the form

StructureName.MemberName

We can display the contents of each member of a structure by using the dot member selector operator.

```
printf("%s %s %0.2f", bankAccount.accountNumber,
                      bankAccount.name, bankAccount.balance);
```

But the syntax is a little more complex when pointers to structures are involved, as we shall see.

Here is a program which fills a structure with a person's bank account details and then displays the details.

```
/* program 7.1 - a bank account structure. */

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

typedef char *String;

typedef struct {
  String number;
  String name;
  double balance;
} BankAccount;

BankAccount bankAccount();
/* post-condition: returns a bankAccount whose fields contain
     values. */

String memoryFor(String);
/* post-condition: returns memory initilaised with the given
     String item */

void printBankAccount(BankAccount *pBankAccount);

int main()
{
  BankAccount aBankAccount;

  aBankAccount = bankAccount();
  printBankAccount(&aBankAccount);
  return 0;
}

BankAccount bankAccount()
{
  BankAccount aBankAccount;

  aBankAccount.number = memoryFor("147309");
  aBankAccount.name = memoryFor("Marris");
  aBankAccount.balance = 150.0;
  return aBankAccount;
}

String memoryFor(String item)
{
  int storageRequired = (int)strlen(item) + 1;
  String memory = (String)malloc(storageRequired);
  strcpy(memory, item);
  return memory;
}


void printBankAccount(BankAccount *pBankAccount)
{
   printf("%s   %s   %0.2f", pBankAccount->number,
                   pBankAccount->name, pBankAccount->balance);
}
```

The result of running this program is that

```
147309   Marris   150.00
```

is displayed on the screen.

The implementation of *bankAccount* requires some explanation. It returns a structure with values assigned to each member.

```
BankAccount bankAccount()
{
  BankAccount aBankAccount;

  ...
  return aBankAccount;
}
```

Assigning a value to aBankAccount.balance is straightforward.

```
aBankAccount.balance = 150.0;
```

But assigning values to *String* (i.e. pointer to *char*) members requires

- allocating sufficient memory to include the end-of-string NULL character

```
int storageRequired = (int)strlen("147309") + 1;
String memory = (String)malloc(storageRequired);
```

- copying the *String* value into the malloced space in memory

```
strcpy(memory, "147309");
```

This is accomplished, for each *String* item, by the *memoryFor* function.

```
String memoryFor(String item)
{
  int storageRequired = (int)strlen(item) + 1;
  String memory = (String)malloc(storageRequired);
  strcpy(memory, item);
  return memory;
}
```

In the function call

```
printBankAccount(&aBankAccount);
```

the address of the structure variable *aBankAccount* is passed to the pointer parameter *pBankAccount* in the function

```
void printBankAccount(BankAccount *pBankAccount)
```

The value stored in each member of the *bankAccount* structure is displayed by

```
printf("%s   %s   %0.2f", pBankAccount->number,
                          pBankAccount->name,
                          pBankAccount->balance);
```

Since *pBankAccount* contain the address of the structure, each member is selected by the `->` operator.

So, if we have a pointer to a structure, we refer to a particular member by writing

PointerToStructure->MemberName

Whatever structure identifier we choose, the members are always referred to by the same name. For example, given the two variable definitions

```
BankAccount, aBankAccount, anotherBankAccount;
```

we write

```
aBankAccount.number, aBankAccount.name, aBankAccount.balance
```

and

```
anotherBankAccount.number, anotherBankAccount.name,
anotherBankAccount.balance
```

A structure may be returned by a function. The address of a structure may be passed to a function parameter. And an entire structure can be passed as an argument value to a parameter - as shown below in another version of *printBankAccount*.

```
void printBankAccount(BankAccount bankAccount)
{
  printf("%s   %s   %0.2f", bankAccount.number,
                            bankAccount.name,
                            bankAccount.balance);
}
```

An example of a call to this function is

```
BankAccount bankAccount;
...
printBankAccount(bankAccount);
```

BankAccount is an example of a data structure sometimes known as a record. The component parts of a record are known as fields. However, in C, a record is known as a structure and a field is called a member.

7.2 Structure Arrays

A bank looks after many accounts. We can model a collection of accounts with an array in which each element is a structure.

```
BankAccount bankAccounts[5];
```

To refer to the *name* member of the i'th element we would write

```
bankAccounts[i].name;
```

where *i* can be any value between 0 and 4 inclusive.

Here is a program which shows how arrays of structures may be used.

```
/* program 7.2 - an array of bank records. */

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

typedef char *String;

typedef struct {
  String number;
  String name;
  double balance;
} BankAccount;

String memoryFor(String);
/* post-condition: returns memory containing a String item */

void buildScenario(BankAccount bankAccounts[]);
/* post-condition: each element of bankAccounts contains a
value. */

void printAccounts(BankAccount bankAccounts[], int lastIndex);
/* pre-condition: lastIndex is the highest index bankAccounts.
*/

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

  BankAccount bankAccounts[arraySize];
  buildScenario(bankAccounts);
  printAccounts(bankAccounts, lastIndex);
  return 0;
}

String memoryFor(String item)
{
  int storageRequired = (int)strlen(item) + 1;
  String memory = (String)malloc(storageRequired);
  strcpy(memory, item);
  return memory;
}
```

```
void buildScenario(BankAccount bankAccounts[])
{
  bankAccounts[0].number = memoryFor("147309");
  bankAccounts[0].name = memoryFor("Marris");
  bankAccounts[0].balance = 150.00;

  bankAccounts[1].number = memoryFor("246890");
  bankAccounts[1].name = memoryFor("Gupta");
  bankAccounts[1].balance = 255.00;

  bankAccounts[2].number = memoryFor("135797");
  bankAccounts[2].name = memoryFor("Green");
  bankAccounts[2].balance = -630.50;

  bankAccounts[3].number = memoryFor("754811");
  bankAccounts[3].name = memoryFor("Smith");
  bankAccounts[3].balance = 200.00;

  bankAccounts[4].number = memoryFor("135797");
  bankAccounts[4].name = memoryFor("Jones");
  bankAccounts[4].balance = 30.80;
}

void printAccounts(BankAccount bankAccounts[], int lastIndex)
{
  int i = 0;
  while (i <= lastIndex) {
    printf("%s   %s   œ%0.2f\n", bankAccounts[i].number,
         bankAccounts[i].name, bankAccounts[i].balance);
    i++;
  }
}
```

The output from this program is

```
147309   Marris   £150.00
246890   Gupta   £255.00;
135797   Green   £-630.50;
754811   Smith   £200.00;
797531   Jones   £30.80;
```

This time, it is an array which is passed as an argument value to both *buildScenario* and *printAccounts*. So the . (dot) operator is used in both functions to select a member.

7.3 Member Arrays

A Small Leisure Centre has a single squash court. Owing to the popularity of the court during Monday and Friday lunchtimes, a booking system is used for these times. Members may book the court up to six days in advance.

To help us to appreciate the situation, let us look at an example of a booking process dialogue between a Member and the Receptionist.

Member: I would like to book a squash court.

Receptionist: Certainly. Which day?

Member: Monday please.

Receptionist consults the bookings made so far.

Receptionist: The court is free at 12 and at 1 pm on that day.

Member: What about 2?

Receptionist: Sorry. The court is booked at 2. Would you like another time or another day?

Member: What about Friday?

Receptionist consults bookings made so far.

Receptionist: The court is free at 2 pm on Friday.

Member: That will suit me very nicely.

Receptionist: What name please?

Member: Smith

Receptionist: The court is now booked for you.

We can model the bookings diary as an array in which each element is a structure or record, one for each array index or time slot.

Array Index	Day	Time	Name
0	Mon	12	
1	Mon	1	
2	Mon	2	Patel
3	Fri	12	Jones
4	Fri	1	French
5	Fri	2	Smith

The court is free on Monday at 12 and at 1 pm, but is booked at all other times. The bookings data structure is easily defined in C.

First, we define the structure of one booking, that is, one entry in the bookings diary.

```
typedef char *String;

typedef struct {
   String day;
   int time;
   String name;
}  BookingEntry;
```

But the diary is made up of several such entries.

```
typedef struct {
  int lastIndex;
  BookingEntry bookingsArray[6];
} BookingsDiary;
```

Here, we have combined the array and its last index value into a single structure named *BookingsDiary*. Why? Suppose, for example, we want to display the entire contents of the bookings diary. Two possible calls to a function to perform this task are

(1) *showAllBookings(bookingsDiary, lastIndex);*

and

(2) *showAllBookings(bookingsDiary);*

In the first example, we pass an array together with its last index value. In the second example, we pass a structure which contains the array together with its last index. The second example is preferred because it has fewer arguments than the first example. However, lastIndex must be initialised with a value. This can be done in the function which sets up the initial contents of the bookings diary.

```
BookingsDiary emptyBookings(void)
{
  BookingsDiary bookings;

  bookings.bookingsArray[0].day = (String)malloc(10);
  strcpy(bookings.bookingsArray[0], "Mon";)
  bookings.bookingsArray[0].time = 12;
  bookings.bookingsArray[0].day = (String)malloc(100);
  strcpy(bookings.bookingsArray[0].name, blank);

    ...

  bookings.lastIndex = 5;
  return bookings;
}
```

blank is defined to be the space character " ". Since we need to use blank in various functions throughout the program, we write

```
#define blank " "
```

near the top of the program just before the implementation of the first function (*main* in our case). In the compiler's pre-translation stage, every occurrence of the word *blank* in a statement is replaced with " " (that is, quotes space quotes.) So, for example, the pre-processor part of the compiler replaces

```
  strcpy(bookings.bookingsArray[0].name, blank);
```

with

```
  strcpy(bookings.bookingsArray[0].name, " ");
```

The substitution of blank with " " does not occur inside comments such as

```
/* blank out the name. */
```

or within quoted strings such as

```
printf("Replace blank with quotes-space-quotes");
```

Notice that there is no space between # and *define* and the line is not terminated with a semi-colon. *#define* is an example of a pre-processor directive.

The user's view of the program is a menu which provides the following operations

```
1  make a booking
2  show all bookings
3  at the end of the day, clear the day's bookings
```

The specifications of the main functions are

```
BookingEntry aBooking(String day, unsigned time, String name);
/* post-condition: returns a booking assembled from day, time
          and name. */

BookingsDiary bookingsAfterBookingMade(BookingsDiary
bookings);
/* post-condition: returns bookings with (possibly) */
                 a new booking included. */

BookingsDiary bookingsClearedForDay(BookingsDiary bookings,
          String day);
/* post-condition: returns bookings with all names for day
          blanked. */

BookingsDiary emptyBookings(void);
/* post-condition: every name in bookings is blank. */

int isBooked(BookingsDiary bookings, int time, String day);
/* post-condition: returns true if name for day and time is
          not blank, otherwise returns false. */

void showAllBookings(BookingsDiary bookings);
/* prints the entire bookings diary. */

BookingsDiary updatedBookings(BookingsDiary bookings,
                              BookingEntry booking);
/* post-condition: returns bookings with booking included. */

unsigned vacanciesOnDay(BookingsDiary bookings, String day);
/* post-condition: returns the number of vacant, unbooked
          slots on day. */
```

The main function makes calls to *emptyBookings*, *bookingsAfterBookingMade*, *showAllBookings* and *bookingsClearedForDay*.

```
int main()
{
  enum { false = 0, true };

  BookingsDiary bookings;
  BookingEntry aBooking;
  String reply, day;
  int done = false;

  bookings = emptyBookings();
  while (!done) {
    reply = stringRead(
        "\nBookings menu: (M)ake, (S)how, (C)lear, (Q)uit: ");
    reply[0] = toupper(reply[0]);
    switch (reply[0]) {
    case 'M':
      bookings = bookingsAfterBookingMade(bookings);
      break;
    case 'S':
      showAllBookings(bookings);
      break;
    case 'C':
      day = dayRead();
      bookings = bookingsClearedForDay(bookings, day);
      break;
    case 'Q':
      done = true;
      break;
    default: /* unexpected value in reply[0]. */
      break;
    }
  }
}
```

Here we have used a simple menu; the user is invited to enter the first letter of their choice. This is stored in *reply*. Then, we used a new control structure, the *switch* or *case* construct, to process the user's choice.

The *switch* construct functions in a way similar to, but different from, a sequence of *else if*s.

```
if (reply[0] == 'M')
  bookings = bookingsAfterBookingMade(bookings);
else if (reply[0] == 'S')
  showAllBookings(bookings);
else if (reply[0] == 'C') {
  day = dayRead();
  bookings = bookingsClearedForDay(bookings, day);
}
else if (reply[0] ==  'Q')
  done = true;
else
  /* unexpected character in reply[0]. */
```

The switch statement selects a case for execution. The case selected depends on the value *contained in reply[0]*. So, for example, if *reply[0]* contains *'S'* then

```
case 'S':
  showAllBookings(bookings);
  break;
```

is executed.

The *break* statement prevents the cases which follow case *'S'*: from being executed. The *default* case in the *switch* construction corresponds in its function to the trailing *else* clause in a sequence of *else if*'s; it caters for non of the preceding cases. The *default* case is optional just as the trailing *else* is optional. The *return* statement is also used to prevent fall through to the next case, as shown below in the *bookingsAfterBookingsMade* function.

```
BookingsDiary bookingsAfterBookingMade(BookingsDiary bookings)
{
  enum { false, true, identical = 0 };

  typedef enum { day, time, name } State;

  String aDay, aName;
  int aTime;
  BookingEntry aNewBooking;
  int vacancies;
  State state = day;

  while (true) {
    switch(state) {
    case day:
      aDay = dayRead();
      if (strcmp(aDay, "None") == identical)
        return bookings;
      vacancies = vacanciesOnDay(bookings, aDay);
      if (vacancies == 0)
        state = day;
      else
        state = time;
      break;

    case time:
      aTime = timeRead();
      if (aTime == 0)
        return bookings;
      if (isBooked(bookings, aTime, aDay)) {
        printf("\nThe time is booked - choose another.\n");
        state = time;
      }
      else
        state = name;
      break;
```

```
    case name:
      aName = nameRead();
      aNewBooking = aBooking(aDay, aTime, aName);
      bookings = updatedBookings(bookings, aNewBooking);
      if (isBooked(bookings, aTime, aDay))
        printf("Booking confirmed\n");
      else
        printf("Error in booking.\n");
      return bookings;
    }
  }
}
```

The implementation of *bookingsAfterBookingMade* is a little tricky because the steps involved in making a booking are not necessarily the same for every booking made. We need to reflect the Receptionist's way of working. First, we recognise that there are three different situations or states; these are defined in the enum

```
typedef enum { day, time, name } State;
```

(In the absence of specific values, the compiler automatically assigns zero to day, one to time and two to name.)

In the *day* state we establish the day required by the Member and offer an alternative day if needed. We remain in the *day* state until either a day is chosen, or no days are chosen. If no days are chosen (that is, *None*) then we exit from the function via the *return* statement. If a day is chosen, then we pass onto the time state.

We remain in the *time* state if the chosen day and time are already booked; we offer an alternative time. If there is no time slot available, then we exit from the function via the *return* statement. If an agreeable time slot has been found, then we pass on to the *name* state.

We remain in the *name* state just to obtain the Member's name and to make and confirm the booking. Then we exit from the function via the *return* statement.

Now, the expression

```
while (true) {
  ...
}
```

means loop for ever. But we break out of the loop by exiting (that is, returning) from the function at appropriate times.

The completion of the program is left as an exercise.

An example of a program run is:

```
Bookings menu: (M)ake, (S)how, (C)lear, (Q)uit: M

Day (Mon, Fri or None)? Mon

Current bookings for Mon are:

Mon 12
Mon  1
Mon  2 Jones

Time (12, 1, 2, or 0)?  0

Bookings menu: (M)ake, (S)how, (C)lear, (Q)uit: M

Day (Mon, Fri or None)? Fri

Current bookings for Fri are:

Fri 12 French
Fri  1 Kline
Fri  2

Time (12, 1, 2 or 0)? 2

Name? Smith

Current bookings for Fri are:

Fri 12 French
Fri  1 Kline
Fri  2

Bookings menu: (M)ake, (S)how, (C)lear, (Q)uit: Q
```

Exercise 7.1

1 From time to time the current balance in a particular account is required. Design a function which, if given an account number, returns the balance in the account with that number - if it exists. A specification for this function could be

```
double balanceInAccount(const BankAccount bankAccounts[],
                        int lastIndex, String accountNumber);
/* pre-condition: lastIndex is the highest index in
                  bankAccounts. accountNumber is in
                  bankAccounts.
   post-condition: returns balance in account for which
                  bankAccounts[i].number == accountNumber.
*/
```

However, since you cannot retrieve a balance from a bank account which does not exist, you should also provide a function which returns true if there is an account for a given account number, false otherwise. Include the functions in program 7.2 shown above and, after making suitable amendments to main,

test them.

2 The balance in most accounts does not stay the same for long; withdrawals are made to pay for goods and services and from time to time deposits are made to increase the balance. Design, write and test a function, which if given an account number and a transaction updates the balance held in the account for the given account number.

3 A Direct Computer Supplies Mail Order business maintains a price list of computer hardware it offers for sale. An entry in the price list contains

a catalogue code - a six character number eg MAR299
an item name - eg Marris Magic Printer
a description - eg 24-pin dot matrix printer
a price - eg £99.99

(a) Design a suitable data structure to contain about seven entries in the price list.

(b) State the specification for a function which will "build the scenario", that is, which will fill the data structure you designed in part (a) above with sample test data. Then write the function implementation.

(c) State the specification for a function which will display the entire price list. Then write and test the function.

(d) Every price in the list is to be increased by, say, ten percent. State the specification for a function which will return the required percentage increase input by the person running the program. Then write the function.

(e) Write the specification for a function which will increase every price in the price list by the percentage increase entered by the user; the percentage increase value must be a parameter to the function. Then write and test the function.

4 A airline maintains a record of the seats available on each of its scheduled flights. The following information is recorded.

flight number
maximum number of seats available
number of seats currently booked

Design, write and test a C program which will simulate the booking activities of the airline. When a reservation is made, the number of seats currently booked is increased by the requested number of seats but only if there are enough seats available. When a cancellation is made, then the number of seats booked is decreased.

5 A small Leisure Centre has a single squash court. Owing to the popularity of

the court during Monday and Friday lunchtimes, a booking system is used for these times. Members may book the court up to six days in advance. Create a bookings diary.