4 Functions

4.1 Introduction

We have already used library functions such as *printf*, *gets* and *sscanf* in our programs. We have used our own functions - *unsignedNumberRead*, *intNumberRead* and *doubleNumberRead* for example. In this chapter we design and write some functions for ourselves. We start with a simple example.

4.2 Designing and Implementing Functions

Let us suppose that a function which adds two *int* numbers together and gives us the result is useful. The result supplied by the function is the sum of two *int* numbers. So we name the function *intSum*. The function requires two *int* values to add together; these are to be supplied to the function from another source. We are not concerned with where the values come from, only with what to do with them once we have them. We shall name these two values *X* and *Y*. Now we can write the function prototype thus

```
int intSum(int X, int Y);
```

Reading from right to left, this function prototype tells us that the function requires two *int* values - named *X* and *Y* here, that its name is *intSum*, and that its result is an *int* value.

The items enclosed within brackets and separated by a comma, namely *int X* and *int Y*, are known as parameters.

We have to place some restrictions on the parameters. First, both *X* and *Y* must contain *int* values. Second, we cannot allow the value of *X* plus *Y* to be too large to fit into an *int* variable. We express these restrictions as pre-conditions thus

```
int intSum(int X, int Y);
/* pre-condition: X + Y <= INT_MAX */
```

The pre-condition *X and Y must both contain int values* is implied by the parameters defined in the function prototype. And so they do need to be explicitly mentioned.

If the values supplied meet the restrictions, then we will guarantee that the function will supply the correct result. We express this in a post-condition thus

```
int intSum(int X, int Y);
/* pre-condition: X + Y <= INT_MAX */
/* post-condition: returns X + Y   */
```

Pre-conditions describe what must be true on entry to a function if the function is to perform as required. Post-conditions describe what must be true on exit from a function when the function has completed its task. If pre-conditions are not met then the function result is not specified. A function template, together with a set of pre- and post-conditions, is known as a function specification.

Given a function's specification, we can use it without knowing or even caring how the function actually works. For example

(a)
```
int sumOfInts = intSum(2, 3);
```

(b)
```
int jockeysWeight = 96;
int saddleWeight = 7;
int totalWeight = intSum(jockeysWeight, saddleWeight);
```

When the function is used as in

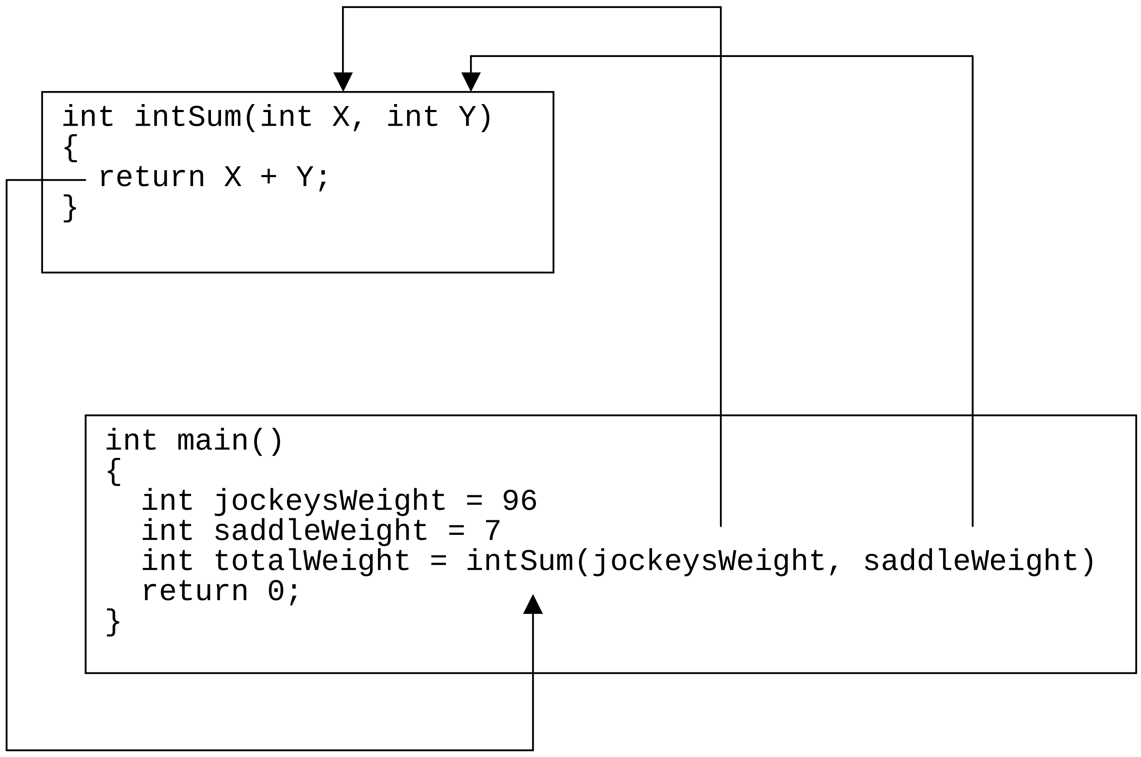
```
int sumOfInts = intSum(2, 3);
```

the value *2* is passed to *X* and the value *3* is passed to *Y*. The values *2* and *3* are examples of argument values. The order in which the arguments are written must match the order in which the parameters are written. So, when the function starts to perform its task, *X* contains *2* and *Y* contains *3*. When the function completes its task, the function result, *5* in this example, is stored in *sumOfInts*. We say that *a function returns a value*. The value returned may be assigned to a variable (as in this example) or used in an expression (as we shall see) or even ignored.

When the function is used, as in

```
int jockeysWeight = 96;
int saddleWeight = 7;
int totalWeight = intSum(jockeysWeight, saddleWeight);
```

a copy of the value stored in *jockeysWeight* (namely *96*) is passed to *X* and a copy of the value stored in *saddleWeight* (namely *7*) is passed to *Y*. The following diagram, Figure 4.1, illustrates the point.



```
int intSum(int X, int Y)
{
  return X + Y;
}
```

```
int main()
{
  int jockeysWeight = 96
  int saddleWeight = 7
  int totalWeight = intSum(jockeysWeight, saddleWeight)
  return 0;
}
```

Figure 4.1 - initially, jockeysWeight contains 96 and saddleWeight contains 7. Then a call to intSum is made. X receives a copy of jockeysWeight contents and Y receives a copy of saddleWeight contents; consequently, X contains 96 and Y contains 7. Then intSum returns its result and 103 is placed in totalWeight.

When the function *intSum* begins to perform its task, *X* contains *96* and *Y* contains *7*. When the function completes its task, the value returned by the function, namely 96 + 7 == 103, is stored in *totalWeight*. We say that a function returns a value in its name.

Argument and parameter names need not be the same, but the order in which they are written must match each other. Consequently, a single function may be used in many different contexts.

Let us see how the function *intSum* actually works; its implementation is straightforward.

```
int intSum(int X, int Y)
{
  return X + Y;
}
```

The C keyword *return* returns you to the point where the function was called, and, in this case, the value of *X* + *Y* is sent there also. Let us see how the function would be used in a complete program.

```
/* program 4.1 - uses intSum function. */

#include <stdio.h>

int intNumberRead(char prompt[]);
/* post-condition: returns int value from the keyboard.     */

int intSum(int X, int Y);                   ←———    function template with
/* pre-condition: X + Y <= INT_MAX */                two int parameters
/* post-condition: returns X + Y   */

int main()
{
  int a = intNumberRead("Number? ");
  int b = intNumberRead("Another number? ");
  int c = intSum(a, b);                     ←———    function call with
                                                     arguments a and b.
  printf("Their sum is %d\n", c);
  return 0;
}

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

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

int intSum(int X, int Y)                    ←———    function heading with
{                                                    parameters X and Y
  return X + Y;
}
```

An example of a program run is

```
Number? 3
Another number 2
Their sum is 5
```

A function implementation is known in C as a function definition. The functions which make up the program may be written in any order. But we follow the convention that *main* is the first function written, and that the other functions are written in alphabetical order. A function prototype must come before its implementation and call.

In general, a function takes some argument values, stores them in parameter variables, performs some task on the parameter values and returns a single value as its result. The values that may be returned by a function include those of type *char*, *int* and *double*. Let us look at another simple example.

Value Added Tax (VAT) is charged at different rates for different goods or services. Let us write a simple function which will receive two argument values - one representing the cost of some goods or services and one representing the rate of VAT to be charged. The function is to return the total cost of the goods. We shall name the function *costPlusVat*. Here is the function specification.

```
double costPlusVat(double cost, double vatRate);
/* pre-condition: cost + cost * vatRate < DBL_MAX     */
/* post-condition: returns cost with VAT added to it. */
```

Given the following variable definitions and assignments

```
double cost = 100.00;
double generalVatRate = 0.175;  /* 17.5% */
```

the function may be used like this

```
double costWithVat = costPlusVat(cost, generalVatRate);
```

The argument values, *100.00* and *0.175*, are passed to, and stored in, the function parameter variables *cost* and *vatRate*. The function adds VAT to the cost and returns the total cost, which is then stored in the variable *costWithVat*.

The implementation of *costPlusVat* is straightforward.

```
double costPlusVat(double cost, double vatRate)
{
  return cost + cost * vatRate;
}
```

And a final example: students are graded on their exam performance as either satisfactory if they obtain 40% or more, and unsatisfactory otherwise. We shall write a function which receives, as its argument value, an integer to represent a percentage mark, and which returns *u* if the mark is less than 40 or *s* if the mark is 40 or more. We shall not place any special restrictions on the argument value except that it should not be negative. We shall name the function *gradeFromMark*. Here is its specification.

```
char gradeFromMark(int mark);
/* post-condition: returns u if mark < 40,
                    otherwise returns s. */
```

An example of its use is

```
int mark = 50;
char grade = gradeFromMark(mark);

if (grade == 'u')
  printf("Include student on unsatisfactory list.\n");
else if (grade == 's')
  printf("Include student on satisfactory list.\n");
```

In C, *char* constant values are written within single quotation marks. So the character constants *u* and *s* are written in C as *'u'* and *'s'*.

An implementation for *gradeFromMark* is

```
char gradeFromMark(int mark)
{
  if (mark < 40)
    return 'u';
  else
    return 's';
}
```

The problem with using *u* to represent unsatisfactory is that we have to remember what *u* means. It would be better to use a more descriptive term such as the word *unsatisfactory* itself.

C allows us to define our own constant enumerated values (see program 3.1 for example). We shall extend this to define our own type named *Grade*.

```
typedef enum { unsatisfactory, satisfactory } Grade;
```

typedef is a C keyword. It is used to provide an alternative name for an existing data type. Here, we have given the name *Grade* to the *enum* data type { *unsatisfactory*, *satisfactory* }. Having defined a data type we can define variables of this type.

```
Grade grade;
```

And we can declare functions to return a value of this type.

```
Grade gradeFromMark(int mark);
```

Here, we have followed the convention that type names chosen by ourselves begin with an upper case letter. (Variable and function names always begin with a lower case letter so we can see at a glance which is a type and which is a variable or function.)

We enhance the specification by writing

```
Grade gradeFromMark(int mark);
/* post-condition: returns unsatisfactory if mark < 39, */
/*                 otherwise returns satisfactory.      */
```

Program 4.2 shown below displays either *Place student on unsatisfactory list* or *Place student on satisfactory list* depending on the value returned by the *gradeFromMark* function.

```
/* program 4.2 - grades a student as either satisfactory */
/*               or unsatisfactory.                      */

#include <stdio.h>

typedef enum { unsatisfactory, satisfactory } Grade;

Grade gradeFromMark(int mark);
/* post-condition: returns unsatisfactory if mark < 40, */
/*                 otherwise returns satisfactory.      */
```

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

int main()
{
  Grade grade;
  int mark = intNumberRead("Student's mark? ");

  grade = gradeFromMark(mark);
  if (grade == unsatisfactory)
    printf("Place student on unsatisfactory list.\n");
  else
    printf("Place student on satisfactory list.\n");
  return 0;
}

Grade gradeFromMark(int mark)
{
  if (mark < 40)
    return unsatisfactory;
  else
    return satisfactory;
}

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

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

Two examples of program runs are

(1)
```
Student's mark? 39
Place student on unsatisfactory list.
```

(2)
```
Student's mark? 40
Place student on satisfactory list.
```

Exercise 4.1

1 A Building Society will lend up to about three times the annual salary of a person applying for a mortgage. Design, write and test a function which will take two argument values - a salary and a multiplier - and return the maximum loan allowed on that salary.

2 The cost of materials used in manufacturing a simple double-glazed, non-opening rectangular window depends on its size. Design, write and test a function which will take four argument values - the length and height in metres of a window, the cost of the frame material per metre and the cost of the glass per square metre, and which will return the total cost of the materials.

3 Competitors in a marathon are categorised, according to their age, as either junior, regular or senior. Design, write and test a function which will receive, as an argument value, a competitor's age and return junior if age is less than 17, regular if age is between 17 and 39 inclusive and senior if age is 40 or more.

4 Write a function which will calculate and return the car parking charge payable if the charge is £0.75 per hour (or part hour). Your function should have three parameters - the arrival and departure times in 24 hour clock notation, and the unit charge.

4.3 Some Useful Functions

4.3.1 YesOrNo

We often need to ask a program user a question which invites a yes or no answer. For example: Married (y/n)? Are you currently employed (y/n)? Aged over 40 (y/n)? Let us design and write a function to do the task. First, we name the function *yesOrNo* since we require the function result to represent either yes or no (or TRUE or FALSE). The function parameter - only one in this case - is going to be text, that is, a string literal. So the parameter is a sequence of *char*. We do not need to place any special restrictions on the argument value.

Now we can write the function specification.

```
int yesOrNo(char prompt[]);
/* post-condition: returns either 0 for FALSE or No,       */
/*                 1 for TRUE or Yes.                      */
```

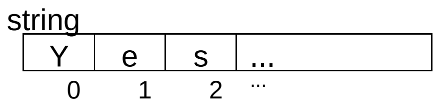
An example of its use is

```
int hasAGrant;
int isAFullTimeStudent =
          yesOrNo("Are you a full time student (y/n)? ");

if (isAFullTimeStudent)
  hasAGrant = yesOrNo("Do you have a grant (y/n)? ");

if ((isAFullTimeStudent) && (hasAGrant))
  printf("Loan application accepted.\n");
else
  printf("Loan application rejected.\n");
```

Let us see how the function may be implemented.

```
int yesOrNo(char prompt[])
{
  char string[BUFSIZ];

  prints("%s", prompt);
  gets(string);
  return (string[0] == 'y') || (string[0] == 'Y') ||
         (string[0] == 't') || (string[0] == 'T');
}
```

string is a variable which can store a sequence of up to *BUFSIZ* (*BUFSIZ*, defined in *stdio.h*, is usually *255*) characters. The place where each character is stored in *string* is identified by a number. The first storage location is always numbered zero. So the string literal *Yes* would be stored in the variable string like this



We are only interested in the first character stored. We identify the first character stored in *string* by writing *string[0]*. In our example here, *string[0]* contains the *char Y*. In C, *char* constant values are enclosed within single quotes. So we can write

```
string[0] == 'Y'
```

The statement

```
return (string[0] == 'y') || (string[0] == 'Y') ||
       (string[0] == 't') || (string[0] == 'T');
```

says return TRUE (that is, one) if the first character stored in *string* is either *y* or *Y* or *t* or *T*, otherwise, return FALSE (that is, zero).

The *yesOrNo* function is used in the following program.

```
/* program 4.3 - uses the yesOrNo function. */
#include <stdio.h>

int yesOrNo(char prompt[]);
/* post-condition: returns either 0 for FALSE or No, */
/*                 1 for TRUE or Yes.                */

int main()
{
  int canFindError =
              yesOrNo("Can you find your error (y/n)? ");

  if (canFindError)
    printf("Then fix it!\n");
  else
    printf("Perhaps you are looking in the wrong place.\n");
  return 0;
}
```

```
int yesOrNo(char prompt[])
{
  char string[BUFSIZ];

  printf("%s", prompt);
  gets(string);
  return (string[0] == 'y') || (string[0] == 'Y') ||
         (string[0] == 't') || (string[0] == 'T');
}
```

Two examples of program runs are

(1)
```
Can you find your error (y/n)? y
Then fix it.
```

(2)
```
Can you find your error (y/n)? N
Then perhaps you are looking in the wrong place.
```

4.3.2 doubleEqual

Now we consider another example. Deciding whether two *double* values are equal or not is a bit of a problem. For example, in some circumstances we might agree that 3.999999 is the same as 4.000000. And in some circumstances we might agree that these two values are not the same. To complicate matters a little more, we cannot guarantee that values of type *double* (or *float*) are stored exactly (because they are converted into binary format before they are stored in memory, and for some double values there is no exact binary equivalent). In all circumstances we shall agree that two *double* values are the same provided they differ by no more than an agreed amount (known as the tolerance). So, if the tolerance is 0.01, then 3.999 would be the same as 4.000 because 4.000 - 3.999 < 0.01. (0.001 is less than 0.01.) However, if the tolerance is 0.0005 then 4.000 would not be the same as 3.999 because 4.000 - 3.999 < 0.0005. (< means is-not-less-than.) We shall name the function *doubleEqual*. The function shall have two double parameters to represent the two values to be compared, and a third double parameter to represent the tolerance. We shall place no special restriction on the argument values except that they be values of type *double*. The function will return an *int* result to represent either TRUE or FALSE. Here is its specification.

```
int doubleEqual(double X, double Y, double tolerance);
/* post-condition: returns 1 if absolute(X - Y) <= tolerance
                   otherwise returns 0. */
```

What is this *absolute(X - Y)*? It is the difference between the two values *X* and *Y*; this difference is always positive (or zero) irrespective of whether *X* is greater than *Y* or *Y* is greater than *X*. So

```
doubleEqual(3.999, 4.000, 0.01) == TRUE;
```
and
```
doubleEqual(4.000, 3.999, 0.01) == TRUE;
```

Another example of a call to *doubleEqual* is

```
double actualBallBearingSize = 3.906;
double requiredBallBearingSize = 3.900;
double tolerance = 0.005;

if doubleEqual(actualBallBearingSize,
                    requiredBallBearingSize, tolerance)
  printf("Accept.\n");
else
  printf("Reject.\n");
```

The implementation of *doubleEqual* is straightforward.

```
int doubleEqual(double X, double Y, double tolerance)
{
  return fabs(X - Y) <= tolerance;
}
```

fabs returns the absolute double value of its *double* argument. So, for example,

fabs(-0.03) == 0.03 and *fabs(0.03) == 0.03*.

fabs is defined in the *math* library. The function *doubleEqual* returns TRUE (that is, one) if the value returned by *fabs* is less than or equal to *tolerance*, otherwise, *doubleEqual* returns FALSE (that is, zero).

The following program, program 4.4, uses *doubleEqual* to determine whether the amount of money in a supermarket check-out till is close enough to what should be in the till.

```
/* program 4.4 - uses doubleEqual. */
#include <stdio.h>
#include <math.h>

int doubleEqual(double X, double Y, double tolerance);
/* post-condition: returns 1 if absolute(X - Y) <= tolerance
                   otherwise returns 0.                */

double doubleNumberRead(char prompt[]);
/* post-condition: returns number entered by the user at the
     keyboard.                                         */

int main()
{
  double actualMoneyInTill =
            doubleNumberRead("Money actually in till is £");
  double calculatedMoneyInTill =
            doubleNumberRead("Money in till should be £");
  double tolerance = 0.50;

  if (doubleEqual(actualMoneyInTill,
                   calculatedMoneyInTill, tolerance))
    printf("OK\n");
  else
    printf("Money in till does not match expected amount.\n");
  return 0;
}
```

```
int doubleEqual(double X, double Y, double tolerance)
{
  return fabs(X - Y) <= tolerance;
}

double doubleNumberRead(char prompt[])
{
  char string[BUFSIZ];
  double number = 0.0;

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

Two examples of program runs are

(1)
```
Money actually in till is £100.00
Money in till should be £99.50
OK
```

(2)
```
Money actually in till is £100.00
Money in till should be £99.49
Money in till does not match expected amount.
```

4.4 Addresses as Arguments to Function Parameters

Up until now, argument values have been passed strictly in one direction - from the point where the function was called to the function parameters; a function could only return a single value such as an *int*, *char* or *enum* value to its caller. In this section we see how a C function can return several values back to its caller.

We have already written several general purpose functions such as *intNumberRead* and *doubleNumberRead*. What we need is a function which will read, from the keyboard, a sequence of characters such as somebody's name, and store it in a suitable variable. An appropriate variable definition would be

```
char personsName[BUFSIZ];
```

Here, the variable identifier is *personsName* and the variable is of type *array of char*. An array is just a named, continuous sequence of numbered storage locations.

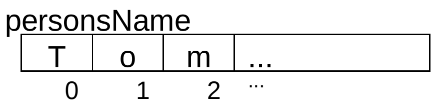


Figure 4.2 - an array is a named sequence of numbered storage locations.

Each storage location in an array of *char* can hold a single *char* value. In C, a sequence of characters stored in an array of *char* is known as a string. So, a person's name, stored in memory, is an example of a string.

Given the variable definition

```
char personsName[BUFSIZ];
```

a function call to input a string value from the keyboard and store it in *personsName* might be

```
readString("Name? ", personsName);
```

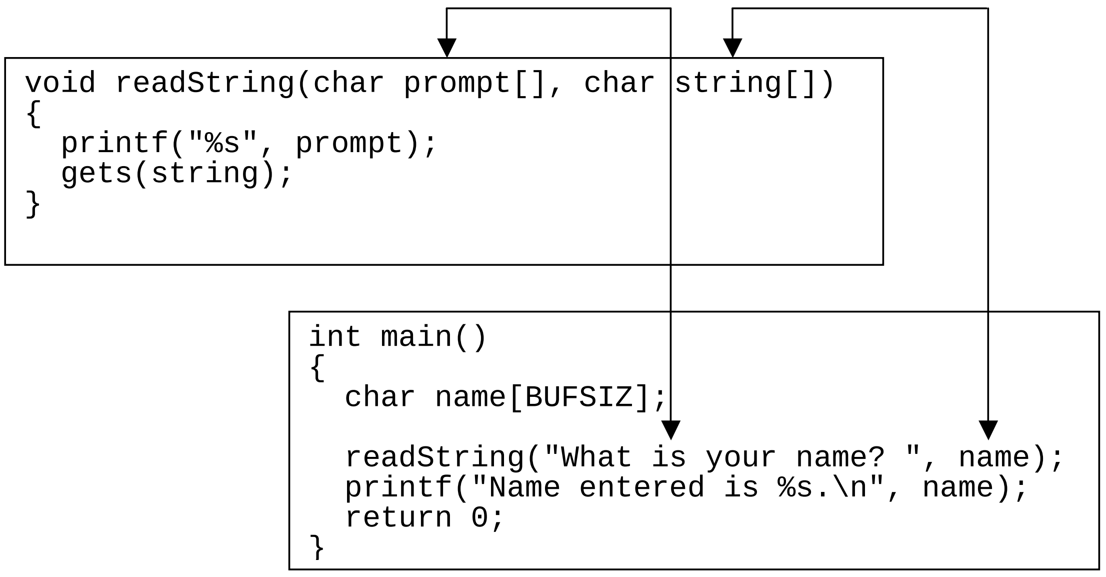
Here is the function's specification.

```
void readString(char prompt[], char string[]);
/* Displays prompt on the screen, inputs string from keyboard.
   post-condition: string' contains characters entered by the
                   user at the keyboard. */
```

The square brackets *[]* declares the parameter to be an array type.

Incidentally, *void* is the type with no values. Think of an empty bag; there is nothing in it. A function with the *void* return type returns nothing.

In making the function call, *Name?* is passed to *prompt* and the array *personsName* is passed to *string*. Passing arrays as argument values is a special case in C because, what is actually passed is the address at which the array is located (rather than a copy of the value of a variable, as has been the case up until now). Therefore, in this example, *personsName* and *string* both refer to the same location in memory. The value stored in this array is known as *personsName* in the function call, and as *string* inside the function itself.



```
void readString(char prompt[], char string[])
{
  printf("%s", prompt);
  gets(string);
}
```

```
int main()
{
  char name[BUFSIZ];

  readString("What is your name? ", name);
  printf("Name entered is %s.\n", name);
  return 0;
}
```

Figure 4.3 - the text *What is your name?* is sent to prompt, and the value retrieved from the keyboard by gets(string) is sent to name in main.

Here is a simple program which uses readString.

```
/* program 4.5 - inputs a string, then displays it. */
#include <stdio.h>

void readString(char prompt[], char string[]);
/* post-condition: string' contains character sequence entered
                   by the user at the keyboard. */

int main()
{
  char name[BUFSIZ];

  readString("What is your name? ", name);
  printf("Name entered is %s.\n", name);
  return 0;
}

void readString(char prompt[], char string[])
{
  printf("%s", prompt);
  gets(string);
}
```

An example of a program run is

```
What is your name? Peter Green
Name entered is Peter Green.
```

Now let us look at another example. Suppose we need a function which will obtain some personal details such as name, gender and age from the keyboard. Given the following type and variable definitions

```
typedef enum { male, female } Gender;

char name[BUFSIZ];
Gender gender;
int age;
```

an appropriate function call might be

```
getDetails(name, &gender, &age);
```

This time we have to explicitly pass the address of both *gender* and *age* as argument values because neither *gender* nor *age* are arrays. We do this by prefixing the variable name with the *&* symbol. The *&* symbol is known as the address operator. Let us look at the corresponding function specification.

```
void getDetails(char name[], Gender *pGender, int *pAge);
/* pre-condition: pGender contains the address of a Gender
     variable, pAge contains the address of an int variable.
   post-condition: name' contains a sequence of characters,
     *pGender' contains either male or female,
     *pAge' contains an int value. */
```

If an address is passed as an argument value, then the corresponding parameter must be capable of storing that address. The parameter declaration is made by prefixing the parameter name with an asterisk, as in *double *pAge* for example. A variable which stores an address is known as a pointer. So we follow the convention that variables which store addresses have their name prefixed with the letter *p*.

The * used in a variable declaration indicates that the variable is a pointer; it is known in this context as the pointer punctuator.

The * used with a variable refers to the contents of that variable; it is known as the indirection operator. For example, *pGender* contains the address of a location in store (or memory). But **pGender* refers to the value stored at that location. We explore the point further as we consider the implementation of *getDetails*.

```
void getDetails(char name[], Gender *pGender, int *pAge)
{
  char string[BUFSIZ];

  readString("Name? ", name);
  *pGender = genderRead("Male or female (m/f)? ");
  *pAge = unsignedNumberRead("Age? ");
}
```

genderRead returns either *male* or *female*. Now, *pGender* contains an address of a location in store, that is, a variable. The line

```
*pGender = genderRead("Male or female (m/f)? ");
```

places either the value *male* or the value *female* into this location.

And again, *pAge* contains the address of a location in store. **pAge* refers to the contents of that storage location. So the statement

```
*pAge = unsignedNumberRead("Age? ");
```

places the value returned by *unsignedNumberRead* into that location. A simple program illustrates the main ideas.

```
/* program 4.6 - uses pointers. */
#include <stdio.h>

void getAge(unsigned int *pAge);
int intNumberRead(char prompt[]);

int main()
{
  int age;
  printf("Address of variable age is %u.\n\n", &age);
  getAge(&age);
  printf("Variable age contains %d.\n", age);
  return 0;
}
```

```
void getAge(int *pAge)
{
  printf("parameter pAge contains %u.\n", pAge);
  *pAge = intNumberRead("Person's age? ");
  printf("*pAge contains %d.\n\n", *pAge);
}

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

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

An example of a program run is

```
Address of variable age is 65508.

Parameter pAge contains 65508.
Person's age? 26
*pAge contains 26.

age contains 26.
```

The events are summarised in Figure 4.4 below.

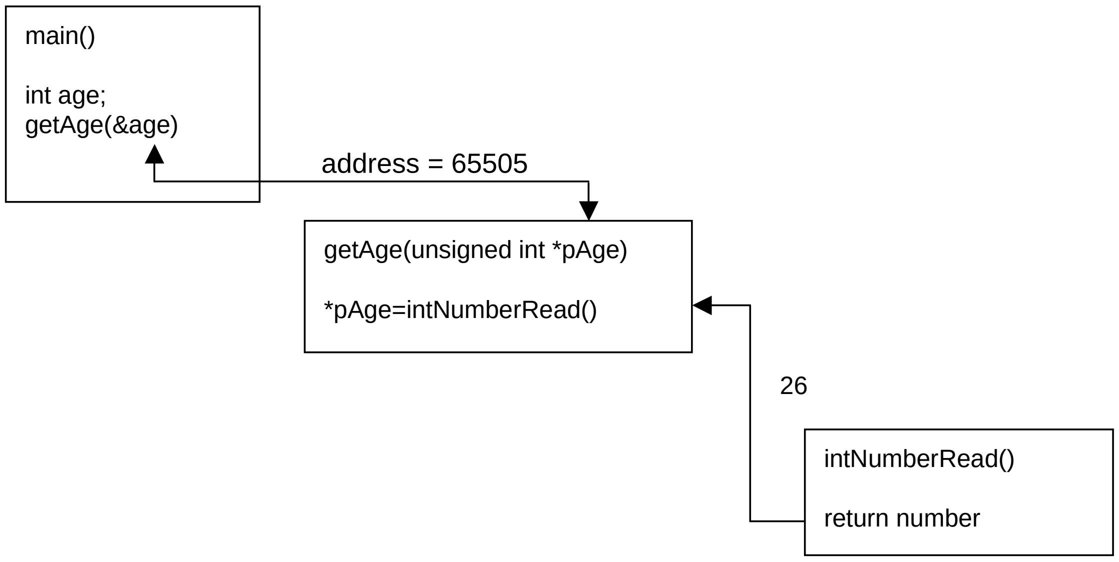


Figure 4.4 - The variable age is declared in main. The address of age is passed to *pAge in getAge; this opens a two-way communication channel. *pAge receives a value e.g. 26, entered by the user and returned by unsignedNumberRead. The value is stored directly in the variable age in main.

Here, we have used * for two different purposes. In the function header, the
* in *getAge(int *pAge)* means that *pAge* is to contain an address. In the statement
**pAge = intNumberRead("Person's age? ")* the * refers to the value stored at an address.

You may have been wondering about the prime symbol, ' in the post-conditions. A parameter identifier without the prime symbol refers to its value on entry to the function. A parameter identifier with the prime symbol refers to its updated value on exit from the function.

4.5 Classical Top Down Design

We begin by looking at a simple problem. A program is required which will maintain a person's bank account. The program is to input a transaction, either a withdrawal or a deposit, update the balance in the account and to display the new balance.

We start with a statement of the problem in a single word.

maintainAccount

We then add more detail by refining this statement into a sequence of function calls.

```
maintainAccount
  balance = currentBalance();
  transaction = transactionInput();
  balance = updatedBalance(transaction, balance);
  displayBalance(balance);
```

currentBalance returns the present balance in the account. *transactionInput* returns the transaction, either withdrawal or deposit, from the keyboard. *updatedBalance* returns the new balance after it has been updated by the transaction. *displayBalance* returns nothing; it merely prints its argument on the screen.

Sometimes, a picture is useful. We describe the structured collection of function calls in a diagram known as a program structure chart.

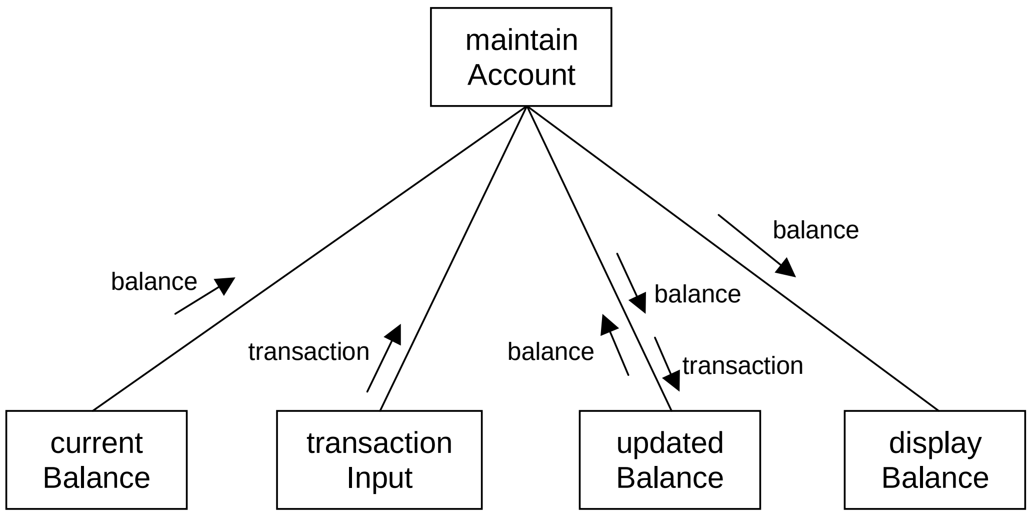


Figure 4.5 - program structure chart showing that currentBalance returns balance to maintainAccount, transactionInput returns transaction to maintainAccount, updatedBalance receives balance and transaction and returns balance and displayBalance receives balance and returns nothing.

Now we can easily implement each function as shown in program 4.7 below.

```
/* program 4.7 - maintains a person's bank account. */

#include <stdio.h>

double currentBalance();
/* post-condition: returns balance currently in a bank
     account. */

void displayBalance(double balance);

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

double transactionInput(void);
/* post-condition: returns transaction input at the keyboard.
*/

double updatedBalance(double transaction, double balance);
/* pre-condition: transaction + balance < DBL_MAX */
/* post-condition: returns transaction + balance  */

int main()  /* maintainAccount */
{
  double balance = currentBalance();
  double transaction = transactionInput();

  balance = updatedBalance(transaction, balance);
  displayBalance(balance);
  return 0;
}
```

```
double currentBalance()
{
  return doubleNumberRead("Current balance £");
}

void displayBalance(double balance)
{
  printf("Current balance is £%0.2f\n", balance);
}

double doubleNumberRead(char prompt[])
{
  char string[BUFSIZ];
  double number = 0.0;

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

double transactionInput()
{
  return doubleNumberRead(
             "Transaction (- withdrawal, + deposit) £");
}

double updatedBalance(double transaction, double balance)
{
  return transaction + balance;
}
```

Here is another example. A program is required which will process applications for The Marathon Race. The program is to input applicants details, put them into categories, assign a competitor number to each person, and to print their details. We shall deal with just one application. We start with a statement of the problem in just one word.

processApplicant

We refine this statement by adding more detail in the form of the main function calls required. For each function call, we show its arguments and return value - if any.

```
processApplicant
  inputApplicants(name, dateOfBirth, gender, age);
  category = applicantsCategory(gender, age);
  number = competitorNumber();
  printCompetitor(name, category, number);
```

Then we choose any one of these function calls and expand on it by adding more detail. Let us look at *inputApplicants* for example.

```
inputApplicants(char name[], char dateOfBirth[], Gender *pGender, int *pAge)
  readName(name);
  readDate(dateOfBirth);
  *pGender = genderRead();
  *pAge = ageRead();
```

We draw the structure chart accordingly as shown in Figure 4.6.

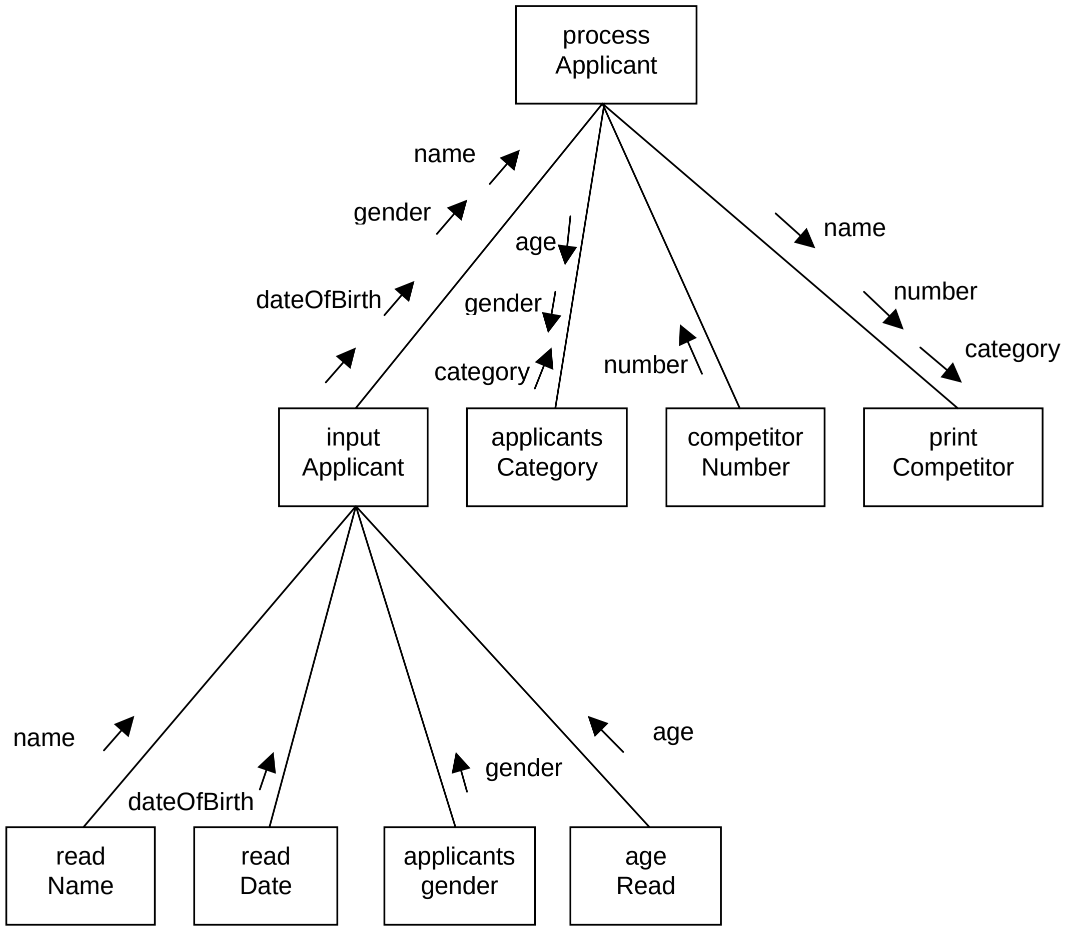


Figure 4.6 - processApplicant calls inputApplicants, applicantsCategory, competitorNumber and printCompetitor. inputApplicants calls readName, readDate, applicantsGender and applicantsNumber.

We repeat this process of refining one function at a time until each function can be easily implemented.

The essence of the classical top-down design method is

- start with a one-word statement of the problem
- refine the statement into a sequence of function calls
- consider refining each of these functions into another sequence of calls
- repeat the refine-a-function process until each function can be easily implemented

Classical top down design results in a program structured as a logical hierarchy of functions.

4.6 Documentation

Program Structure Charts show the hierarchy of function calls and the data items passed between them; see Figures 4.5 and 4.6 for example. You can have several levels of function calls as shown in Figure 4.6. Strive for a consistent level of detail at each level in the hierarchy.

Functions can appear in any order in a program. But we shall follow the convention that the main function is the first one in a program, and all other functions are shown in alphabetical order by function name. This helps us to quickly understand what the program does and to quickly find any function. Without a logical order, a particular function could be visually hard to find in a large program.
A pre-condition states what must be true about a parameter variable on entry to a function. A post-condition states what must be true on exit from a function, provided all the pre-conditions have been met. A post-condition is a relationship between the parameter variables on entry and the parameter variables on exit from a function. A parameter variable on exit from a function is documented with the prime symbol, '. For example, the post-condition

```
/* post-condition: X' = X + Y */
```

says that the value of *X* on exit from the function is equal the original value of *X* plus the origianl value of *Y* on entry to the function. Pre- and post-conditions form the basis of a contract between a function and its user. If the user does not meet the pre-conditions, then the behaviour of the function is not defined - we cannot guarantee that the function will behave as required. Functions should have their pre- and post-conditions specified along with their prototypes; this informs the person reading or using the function about its purpose and how it should be used.

Programmer defined type identifiers should begin with an uppercase letter. Functions which return a value should have an identifer which describes the nature of that value. Functions which do not explicitly return a value should have an identifer which describes the function's purpose. For example, *int sumOfTwoNumers(int X, int Y)* returns the sum of two numbers, but *void addTwoNumbers(int X, int Y, int *Z)* adds two numbers and places the result in **Z.*

If a function does not explicitly return a value, but returns a value (or values) in its parameters, then place them at the end of the parameter list, as shown in *addTwoNumbers* above.

If a function is to be used in just one, specific situation, then choose parameter names which are the same as the argument names. If a function is a general-purpose one, to be used in several different contexts, then use general-purpose parameter names (which are not necessarily the same as the argument names).

4.7 PROGRAMMING PRINCIPLES

Design your programs so that

- a main control function which defines the whole of WHAT is to be done
- they are made up of a hierarchy of functions
- each function does one small task
- each function has a well-defined interface, that is, parameter list
- values are passed between functions only through their interface and return values

Make each function as self contained as possible. Pass the minimum number of argument values from one function to another, that is, just enough to do the job.

If you have a choice of either explicitly returning a value from a function or passing an address as an argument to a parameter variable, return the value; this is simpler and less prone to error.

Provide pre- and post-conditions for every function you write. Whenever you make a call to a function, remember that it is up to you to ensure that its pre-conditions are met.

Exercise 4.2

Remember to document your functions with pre- and post-conditions, and to document your programs with structure charts.

1 A veterinary surgeon charges farmers on the basis of hours spent treating an animal, from the moment the vet leaves his or her home to the moment the vet returns. The vet charges to the nearest hour and the minimum charge is for one hour's work. Design, write and test a function, which is to be part of a billing program, which has parameters for the hours worked and rate per hour, and which returns the charge payable.

2 A garden centre orders bags of compost to replace those sold in the past month. Design, write and test a function which is to be part of an automated re-order system. The function is to have parameters for the number of bags sold in the past month and the maximum number of bags that can be stored on the premises. The function is to return the number of bags to be re-ordered.

3 Design, write and test a function to be used in a college timetabling program. The function is to have parameters for room capacity and anticipated number of students in a class. The function is to return whether the class will fit in the room.

4 A hotel guest is charged extra for certain goods and services. The extra charge is added to the guest's bill. A function is required which will be part of a billing program. The function is to have parameters for the current bill, the cost of extra services and the percentage service charge to be added to every goods or services provided. The function is to return the new, increased bill.

5 Design, write and test function named doubleLessThan. The function is to have three parameters of type double and is to return TRUE if the first parameter is less than the second or FALSE if the second parameter is less than the first, within the tolerance specified by the third parameter.

6 Design, write and test a function named doubleGreaterThan. The function should use doubleEqual and doubleLessThan in its implementation.

7 Design and write a program which can be used to calculate an employee's gross pay. The program is to input the standard and overtime rates of pay per hour, and the number of hours actually worked in a week. The standard hours worked in a week is to be held as an enumerated constant set to, typically, 37. Any hours worked in excess of 37 is to be counted as overtime. The program is to calculate and output the gross pay.

8 Design and write a program which might be used by the local constabulary to help determine whether a suspect for a crime is worth investigating further. The program is to input a suspects details - whether or not they have a motive, past form or an alibi - and to output whether or not the suspect meets any two of the three criteria: has a motive, has past form or has no alibi.

9 Design and write a program which might be used by a friendship agency. The program is to input details of two clients - their name, age and whether they prefer to spend their time outdoors or within the home and garden; the program is to output whether the two clients might make a suitable match. A match is made if the two clients' ages are within five years of each other and they both share the same interests.

10 Design and write a program which might be used by a book publisher to help determine the cost of editing, printing and publishing a book. The cost of editing an author's typescript and producing camera ready copy is about £5.00 per page. The cost of printing is about £2.00 for each 250 (or part of 250) pages. The cost of binding and distribution is about £2.00 per book. The program is to input the number of pages in a book, the number of books to be printed, and the costs of editing and producing camera-ready pages, printing, binding and distribution. The program is to output the total cost of editing, printing and publishing the book.