# 8 Selection

## 8.1 MAKING CHOICES

Loop constructs permit the coding of a few simple numerical calculations; but most programs require more flexibility. Different input data usually have to be interpreted in different ways. As a very simple example, imagine a program producing some summary statistics from lots of data records defining people. The statistics required might include separate counts of males and females, counts of citizens and resident-aliens and visitors, counts of people in each of a number of age ranges. Although similar processing is done for each data record, the particular calculation steps involved will depend on the input data. Programming languages have to have constructs for *selecting* the processing steps appropriate for particular data.

The modern Algol family languages have two kinds of selection statement:

"if " statements

and

"switch" statements (or "case" statements)

There are typically two or three variations on the "if" statement. Usually, they differ in only minor ways; this is possibly why beginners frequently make mistakes with "if"s. Beginners tend to mix up the different variants. Since "if"s seem a little bit error prone, the "switch" selection statement will be introduced first.

### switch statement

The C/C++ switch statement is for selecting one processing option from among a choice of several. One can have an arbitrary number of choices in a switch statement.

Each choice has to be given an identifying "name" (which the compiler has to be able to convert into an integer constant). In simple situations, the different

choices are not given explicit names, the integer numbers will suffice. Usually, the choice is made by testing the value of a variable (but it can be an expression).

Consider a simple example, the task of converting a date from numeric form <day> <month> <year>, e.g. 25 12 1999, to text December 25th 1999. The month would be an integer data value entered by the user:

```
int day, month, year;
cout << "Enter date as day, month, and year" << endl;
cin >> day >> month >> year;
...
```

The name of the month could be printed in a switch statement that used the value of "month" to select the appropriate name. The switch statement needed to select the month could be coded as:

```
...
switch(month) {
case 1:
    cout << "January ";
    break;
case 2:
    cout << "February ";
    break;
…
…
case 12:
    cout << "December ";
    break;
    }
```

***Parts of a switch statement***

A switch statement is made up from the following parts:

```
switch
```

The keyword `switch`.

```
(month)
```

A parenthesised expression that yields the integer value that is used to make the choice. Often, as in this example, the expression simply tests the value of a variable.

```
    {
```

The `{` 'begin block' bracketing character.

```
case
```

The keyword `case`. This marks the start of the code for one of the choices.

```
1
```

The integer value, as a simple integer or a named constant, associated with this choice.

```
:
```

A colon, `':'`, punctuation marker.

```
cout << "January ";
```

The code that does the special processing for this choice.

```
break;
```

The keyword `break` that marks the end of the code section for this choice (or case).

```
case 2:
```

The code for the next case.

```
cout << "February ";
break;"
```

Similar code sections for each of the other choices.

```
}
```

Finally, the `}` 'end block' bracket to match the `{` at the start of the set of choices.

Be careful when typing the code of a switch; in particular, make certain that you pair your `case ... break` keywords.

You see, the following is legal code, the compiler won't complain about it:

```
case 1:
    cout << "January ";
case 2:
    cout << "February ";
    break;
```

But if the month is 1, the program will execute the code for both case 1 and case 2 and so print "`January February `".

***"case fall through"***

C/C++ allows "case fall through" (where the program continues with the code of a second case) because it is sometimes useful. For example, you might have a program that uses a `switch` statement to select the processing for a command entered by a user; two of the commands might need almost the same processing with one simply requiring an extra step :

```
/* Command 'save and quit' --- save data and close up */
case 101:
        SaveData();
        /* CASE FALLTHROUGH REQUIRED */
```

```
/* Command 'quit' --- close up */
case 100:
        CloseWindow();
        DisconnectModem();
        break;
```

(If you intend to get "case fall through", make this clear in a comment.)

Another example where case fall through might be useful is a program to print the words of a well known yuletide song:

```
cout << "On the " << day_of_christmas
   << " day of Christmas, " << endl;
cout << "My true love gave to me " <<endl;

switch(day_of_christmas) {
case 12:
     cout << "Twelve lords a leaping";
case 11:
     ...
     ...
case 3:
     cout << "Three French hens" << endl;
case 2:
     cout << "Two turtle doves, " << endl;
     cout << "and " << endl;
case 1:
     cout << "A partridge in a pear tree";
};
```

The "case labels" (the integers identifying the choices) don't have to be in sequence, so the following is just as acceptable to the compiler (though maybe confusing to someone reading your code):

```
     switch(month) {
case 4:
         cout << "April ";
         break;
case 10:
         cout << "October ";
         break;
      ...
case 2:
         cout << "February";
         break;
         }
```

In this simple example, it wouldn't be necessary to invent named constants to characterize the choices – the month numbers are pretty intuitive. But in more complex programs, it does help to use named constants for case labels. You would have something like:

***Defining named constants for case labels***

```
#include <iostream.h>

const int JAN = 1;
```

```
const int FEB = 2;
...
const int DEC = 12;

void main()
{
    int day, month, year;
    ...
    switch(month) {
case JAN:
       cout << "January ";
       break;
case FEB:
       cout << "February ";
       break;
...
case DEC:
       cout << "December ";
       break;
     }
    ...
```

***Another switch statement***

As well as printing the name of the month, the dates program would have to print the day as 1st, 2nd, 3rd, 4th, ..., 31st. The day number is easily printed ---

```
     cout << day;
```

but what about the suffixes 'st', 'nd', 'rd', and 'th'?

Obviously, you could get these printed using some enormous case statement

```
    switch(day) {
case 1:  cout << "st "; break;
case 2:  cout << "nd "; break;
case 3:  cout << "rd ", break;
case 4:  cout << "th "; break;
case 5:  cout << "th "; break;
case 6:  cout << "th "; break;
case 7:  cout << "th "; break;
...
case 21: cout << "st "; break;
case 22: cout << "nd "; break;
...
case 30: cout << "th "; break;
case 31: cout << "st "; break;
    }
```

***Combining similar cases***

Fortunately, this can be simplified. There really aren't 31 different cases that have to be considered. There are three special cases: 1, 21, and 31 need 'st'; 2 and 22 need 'nd'; and 3 and 23 need 'rd'. Everything else uses 'th'.

The case statement can be simplified to:

```
   switch (day) {
case 1:
case 21:
```

```
case 31:
        cout << "st ";
        break;
case 2:
case 22:
        cout << "nd ";
        break;
case 3:
case 23:
        cout << "rd ";
        break;
default:
        cout << "th ";
        break;
       }
```

Where several cases share the same code, that code can be labelled with all those case numbers:

```
case 1:
case 21:
case 31:
         cout << "st ";
         break;
```

(You can view this as an extreme example of 'case fall through'. Case 1 --- do nothing then do everything you do for case 21. Case 21 --- do nothing, then do everything you do for case 31. Case 31, print "st".)

***Default section of switch statement***

Most of the days simply need "th" printed. This is handled in the "default" part of the switch statement.

```
       switch (day) {
case 1:
case 21:
case 31:
          cout << "st ";
          break;
   ...
  ...
default:
          cout << "th ";
          break;
          }
```

The compiler will generate code that checks the value used in the switch test against the values associated with each of the explicit cases, if none match it arranges for the "default" code to be executed.

A default clause in a switch statement is often useful. But it is not necessary to have an explicit default; after all, the first example with the months did not have a default clause. If a switch does not have a default clause, then the compiler generated code will be arranged so as to continue with the next statement following after the end of the switch if the value tested by the switch does not match any of the explicit cases.

Usually a switch statement tests the value of an integer variable:

```
switch (month) {
...
}
```

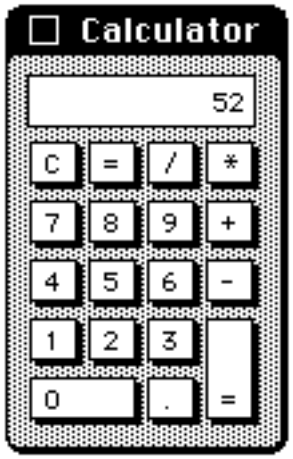
and

```
switch (day) {
...
}
```

But it can test the value of any integer expression.
A program can have something like `switch(n+7) { ... }` if this makes sense.

## 8.2 A REALISTIC PROGRAM: DESK CALCULATOR

Loop and selection constructs are sufficient to construct some quite interesting programs. This example is still fairly simple. The program is to simulate the workings of a basic four function calculator




Specification:

The calculator program is to:

1 Display a "current value".
2 Support the operations of addition, subtraction, multiplication, and division, operations are to be invoked by entering the character '+', '-', '*', or '/' in response to a prompt from the program.
3 When an operator is invoked, the program is to request entry of a second number that is to be combined with the current value; when given a number,

the program is to perform the specified calculation and then display the new current value.

4 The program is to accept the character 'C' (entered instead of an operator) as a command to clear (set to zero) the "current value".

5 The program is to terminate when the character 'Q' is entered instead of an operator.

Example operation of required calculator program:

```
Program initially displaying                0
prompt for operator                         +
                          '+' character entered
prompt for number                           1.23
                          addition operation 0+1.23
current value displayed                     1.23
prompt for operator                         *
                          '*' character entered
prompt for number                           6.4
                          multiplication operation performed
current value displayed                     7.872
prompt for operator                         C
                          clear operation
current value displayed                     0
prompt for operator                         Q
                          program terminates
```

## Design and Implementation

***Preliminary design:***

Programs are designed in an iterative fashion. You keep re-examining the problem at increasing levels of detail.

***First iteration through the design process***

The first level is simple. The program will consist of

- some initialization steps
- a loop that terminates when the 'Q' (Quit) command is entered.

***Second iteration through the design process***

The second iteration through the design process would elaborate the "setup" and "loop" code:

- some initialization steps:
  set current value to 0.0
  display current value
  prompt user, asking for a command character
  read the command character

- loop terminating when "Quit" command entered
  while command character is not Q for Quit do the following
  ( process command --- if input is needed ask for number then do
  ( operation
  ( display result
  ( prompt the use, asking for a command character
  ( read the command character

***Third iteration through the design process***

The part about prompting for a number and performing the required operation still needs to be made more specific. The design would be extended to something like the following:

```
set current value to 0.0
display current value
prompt user, asking for a command character
read the command character

while command character is not Q for Quit do
    examine command character
'C'         set current number to zero
            finished
'+'         ask for a number,
            read number,
            do addition
            finished
'-'         ask for a number,
            read number,
            do subtraction
            finished
'*'         ask for a number,
            read number,
            do multiplication,
            finished
'/'         ask for a number,
            read number,
            do division,
            finished

    display result
    prompt for another command,
    read command character
```

***Detailed design***

The preliminary design phase for a program like this is finished when you've developed a model of the various processing steps that must be performed.

You must then consider what data you need and how the data variables are to be stored. Later, you have to go through the processing steps again, making your descriptions even more detailed.

***Data***

Only three different items of data are needed:

- "the current displayed value" – this will be a double number;
- "the command character"– a character,

and

- another double number used when entering data.

These belong to the main program:

```
int main()
{
    double        displayed_value;
    double        new_entry;
```

```
    char          command_character;
```

***The while loop***

We can temporarily ignore the switch statement that processes the commands and concentrate solely on the while loop structure. The loop is to terminate when a 'Q' command has been entered. So, we can code up the loop control:

```
while (command_character != 'Q') {
    /* switch stuff to go here */
    ...
    /* get next command entered. */
    cout << "Value : " << displayed_value << endl;
    cout << "command>";
    cin >> command_character;
    }
```

The loop continues while the command character is *NOT 'Q'.* Obviously we had better read another command character each time we go round the loop. We have to start the loop with some command character already specified.

Part of the setup code prompts for this initial command:

```
#include <iostream.h>
#include <stdlib.h>

int main()
{
    double        displayed_value;
    double        new_entry;
    char          command_character;

    displayed_value = 0.0;

    cout << "Calculator demo program" << endl
           << "----" << endl;

    cout << "Enter a command character at the '>' prompt"
           << endl;
    ...

    cout << "Value : " << displayed_value << endl;
    cout << "command>";
    cin >> command_character;
    while (command_character != 'Q') {
           /* Switch statement to go here */
           ...
           cout << "Value : " << displayed_value << endl;
           cout << "command>";
           cin >> command_character;
    }
    return 0;
}
```

The code outlined works even if the user immediately enters the "Quit" command!

***Pattern for a loop processing input data***

The pattern

- setup code reads initial data value
- while loop set to terminate when specific data value entered
- next input value is read at the end of the body of the loop

is very common. You will find similar while loops in many programs.

***Switch statement***

Selection of the appropriate processing for each of the different commands can obviously be done with a switch statement:

```
switch (command_character) {
...
}
```

Each of the processing options is independent. We don't want to share any code between them. So, each of the options will be in the form

```
case ... :
    statements
    break;
```

The case of a 'Q' (quit command) is handled by the loop control statement. The switch only has to deal with:

'C' clear command, set current value to zero;

'+' addition, ask for input and do the addition;

'-' subtraction, ask for input and do the subtraction;

'*' multiplication, ask for input and do the multiplication;

'/" division, ask for input and do the division.

Each of these cases has to be distinguished by an integer constant and the switch statement itself must test an integer value.

Fortunately, the C and C++ languages consider characters like 'C', '+' etc to be integers (because they are internally represented as small integer values). So the characters can be used directly in the `switch() { }` statement and for case labels.

```
    switch(command_character) {
case 'C':
            displayed_value = 0.0;
            break;
case '+':
            cout << "number>";
            cin >> new_entry;
            displayed_value += new_entry;
            break;
case '-':
            cout << "number>";
```

```
            cin >> new_entry;
            displayed_value -= new_entry;
            break;
case '*':
            cout << "number>";
            cin >> new_entry;
            displayed_value *= new_entry;
            break;
case '/':
            cout << "number>";
            cin >> new_entry;
            displayed_value /= new_entry;
            break;
    }
```

The code for the individual cases is straightforward. The clear case simply sets the displayed value to zero. Each of the arithmetic operators is coded similarly – a prompt for a number, input of the number, data combination.

Suppose the user enters something inappropriate – e.g. an 'M' command (some calculators have memories and 'M', memorize, and 'R' recall keys and a user might assume similar functionality). Such incorrect inputs should produce some form of error message response. This can be handled by a "default" clause in the switch statement.

```
default:
            cout << "Didn't understand input!";
```

If the user has does something wrong, it is quite likely that there will be other characters that have been typed ahead (e.g. the user might have typed "Hello", in which case the 'H' is read and recognized as invalid input – but the 'e', 'l', 'l' and the 'o' all remain to be read).

***Removing invalid characters from the input***

When something has gone wrong it is often useful to be able to clear out any unread input. The `cin` object can be told to do this. The usual way is to tell `cin` to ignore all characters until it finds something obvious like a newline character. This is done using the request `cin.ignore(...)`. The request has to specify the sensible marker character (e.g. '`\n`' for newline) and, also, a maximum number of characters to ignore; e.g.

```
cin.ignore(100,'\n');
```

This request gets "cin" to ignore up to 100 characters while it searches for a newline; that should be enough to skip over any invalid input.

With the default statement to report any errors (and clean up the input), the complete switch statement becomes:

```
    switch(command_character) {
case 'C':
            displayed_value = 0.0;
            break;
case '+':
            cout << "number>";
```

```
            cin >> new_entry;
            displayed_value += new_entry;
            break;
    case '-':
            cout << "number>";
            cin >> new_entry;
            displayed_value -= new_entry;
            break;
    case '*':
            cout << "number>";
            cin >> new_entry;
            displayed_value *= new_entry;
            break;
    case '/':
            cout << "number>";
            cin >> new_entry;
            displayed_value /= new_entry;
            break;
    default:
            cout << "Didn't understand input!";
            cin.ignore(100,'\n');
        }
```

Since there aren't any more cases after the `default` statement, it isn't necessary to pair the `default` with a `break` (it can't "fall through" to next case if there is no next case) But, you probably should put a `break` in anyway. You might come along later and add another case at the bottom of the switch and forget to put the `break` in.

## 8.3 IF

Switch statements can handle most selection tasks. But, they are not always convenient.

You can use a switch to select whether a data value should be processed:

```
        // Update count of students who's assignment marks
        // and examination marks both exceed 25
        switch((exam_mark > 25) && (assignment_mark > 25)) {
    case 1:
            good_student++;
            break;
    case 0:
            // Yuk, ignore them
            break;
        };
```

The code works OK, but it *feels* clumsy.

What about counting the numbers of students who get different grades? We would need a loop that read the total course marks obtained by each individual student and used this value to update the correctly selected counter. There would

be counters for students who had failed (mark < 50), got Ds (mark < 65), Cs (<75), Bs (<85) and As. The main program structure would be something like:

```
int main()
{
    int A_Count = 0, B_Count = 0, C_Count = 0,
           D_Count = 0, F_Count = 0;
    int mark;
    cin >> mark;
    while(mark >= 0) {
           code to select and update appropriate counter
           cin >> mark;  // read next, data terminated by -1
           }
    // print counts
    …
    …
}
```

How could the selection be made?

Well, you *could* use a `switch` statement. But it would be pretty hideous:

```
    switch(mark) {
case 0:
case 1:
case 2:
…
case 49:
           F_Count++;
           break;
case 50:
case 51:
…
case 64:
           D_Count++;
           break;
…
…
           break;
case 85:
case 86:
case 100:
           A_Count++;
           break;
}
```

Obviously, there has to be a better way.

***if*** The alternative selection constructs use the `if` … and `if` … `else` … statements. A simple `if` statement has the following form:

```
if(boolean expression)
    statement;
```

Example:

```
if((exam_mark > 25) && (assignment_mark > 25))
    good_student++;
```

Often, an `if` will control execution of a compound statement rather than a simple statement. Sometimes, even if you only require a simple statement it is better to put in the { begin bracket and } end bracket:

```
if((exam_mark > 25) && (assignment_mark > 25)) {
    good_student++;
    }
```

This makes it easier if later you need to add extra statements –

```
if((exam_mark > 25) && (assignment_mark > 25)) {
    cout << "*";
    good_student++;
    }
```

(also, some debuggers won't let you stop on a simple statement controlled by an `if` but will let you stop in a compound statement). Note, you don't put a semicolon after the `}` end bracket of the compound statement; the `if` clause ends with the semicolon after a simple statement or with the `}` end bracket of a compound statement.

***if … else***

The `if` … `else` … control statement allows you to select between two alternative processing actions:

```
if(boolean expression)
           statement;
else
           statement;
```

Example:

```
if(gender_tag == 'F')
    females++;
else
    males++;
```

***if … else if … else if … else …***

You can concatenate `if` … `else` control statements when you need to make a selection between more than two alternatives:

```
if(mark<50) {
    F_Count++;
    cout << "another one bites the dust" << endl;
    }
else
if(mark<65)
    D_Count++;
else
if(mark<75)
    C_Count++;
else
```

```
if(mark<85)
    B_Count++;
else {
    A_Count++;
    cout << "*****" << endl
    }
```

You have to end with an `else` clause. The statements controlled by different `if` clauses can be either simple statements or compound statements as required.

***Nested ifs and related problems***

You need to be careful when you want to perform some complex test that depends on more than one condition. The following code is OK:

```
if(mark<50) {
    F_Count++;
    if((exam_mark == 0)  && (assignment_count < 2))
           cout << "Check for cancelled enrollment" << endl;
    }
else
if(mark<65)
    D_Count++;
else
    …
```

Here, the `{` and `}` brackets around the compound statement make it clear that the check for cancelled enrollments is something that is done only when considering marks less than 50.

The `{ }` brackets in that code fragment were necessary because the `mark<50` condition controlled several statements. Suppose there was no need to keep a count of students who scored less than half marks, the code would then be:

```
if(mark<50) {
    if((exam_mark == 0)  && (assignment_count < 2))
           cout << "Check for cancelled enrollment" << endl;
    }
else
if(mark<65)
    …
```

Some beginners might be tempted to remove the `{ }` brackets from the `mark<50` condition; the argument might be that since there is only one statement "you don't need a compound statement construction". The code would then become:

***Buggy code!***

```
if(mark<50)
    if((exam_mark == 0)  && (assignment_count < 2))
           cout << "Check for cancelled enrollment" << endl;
else
if(mark<65)
```

Now, you have a bug.

An `if` statement can exist on its own, without any `else` clause. So, when a compiler has finished processing an `if` statement, it gets ready to deal with any kind of subsequent statement assignment, function call, loop, switch, …. If the

compiler encounters the keyword `else` it has to go back through the code that it thought that it had dealt with to see whether the preceding statement was an `if` that it could use to hang this `else` onto. An `else` clause gets attached to an immediately preceding `if` clause. So the compiler's reading of the buggy code is:

```
if(mark<50)
    if((exam_mark == 0)  && (assignment_count < 2))
           cout << "Check for cancelled enrollment" << endl;
    else
    if(mark<65)
           D_Count++;
    else
           …
```

This code doesn't perform the required data processing steps. What happens now is that students whose marks are greater than or equal to 50 don't get processed – the first conditional test eliminates them. A warning message is printed for students who didn't sit the exam and did at most one assignment. The count `D_Count` is incremented for all other students who had scored less than 50, after all they did score less than 65 which is what the next test checks.

It is possible for the editor in a development environment to rearrange the layout of code so that the indentation correctly reflects the logic of any ifs and elses. Such editors reduce your chance of making the kind of error just illustrated. Most editors don't provide this level of support; it is expected that you have some idea of what you are doing.

Another legal thing that you shouldn't do is use the comma sequencing operator in an `if`:

```
// If book is overdue, disable borrowing and
// increase fines by $3.50
if(Overdue(book_return_date))
    can_borrow = 0, fine += 3.50;
```

Sure this is legal, it just confuses about 75% of those who read it.

## Abbreviations again: the conditional expression

You might expect this by now – the C and C++ languages have lots of abbreviated forms that save typing, one of these abbreviated forms exist for the `if` statement.

Often, you want code like:

```
// If Self Employed then deduction is 10% of entry in box 15
// else deduction is $450.
if(Employ_Status == 'S')
    deduction = 0.1*b15;
else
    deduction = 450.0
```

or

```
// If new temperature exceeds maximum
// recorded, update the maximum
if(temperature > maximum)
    maximum = temperature;
```

or

```
// print label "male" or "female" as appropriate
if(gender_tag == 'f')
    cout << "Female : ";
else
    cout << "Male   : ";
```

All can be rewritten using a conditional expression.

This has the form

```
(boolean expression) ? result_if_true : result_if_false
```

The conditional expression has three parts. The first part defines the condition that is to be tested. A question mark separates this from the second part that specifies the result of the whole conditional expression for the case when the condition evaluates to true. A colon separates this from the third part where you get the result for cases where the condition is false.

Using this conditional expression, the code examples shown above may be written as follows:

```
// If Self Employed then deduction is 10% of entry in box 15
// else deduction is $450.
deduction = (Employ_Status == 'S') ?
                    0.1*b15       :
                    450.0;

maximum = (temperature > maximum) ?
                    temperature   :
                    maximum;

cout << ((gender_tag == 'f') ? "Female : " : "Male   : ");
```

***Caution – watch out for typos related to == operator***

Note: remember to use `==` as the equality test operator and not the `=` assignment operator! The following is legal code:

```
cout << ((gender_tag = 'f') ? "Female : " : "Male   : ");
```

but it changes the value of `gender_tag`, then confirms that the new value is non zero and prints "`Female : `".

## 8.4 TERMINATING A PROGRAM

Sometimes, you just have to stop. The input is junk; your program can do no more. It just has to end.

***exit() function***

The easy way out is to use a function defined in stdlib. This function, `exit()`, takes an integer argument. When called, the `exit()` function terminates the program. It tries to clean up, if you were using files these should get closed. The integer given to `exit()` gets passed back as the return status from the program (like the 0 or `EXIT_SUCCESS` value returned by the main program). A typical IDE will simply discard this return value. It is used in scripting environments, like Unix, where the return result of a program can be used to determine what other processing steps should follow. The normal error exit value is 1; other values can be used to distinguish among different kinds of error.

Several of the later examples use calls to `exit()` as an easy way to terminate a program if the data given are found to be invalid. For example, the square root program (7.2.2) could check the input data as follows:

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>

int main()
{
    const double SMALLFRACTION = 1.0E-8;
    double x;
    double r;
    cout << "Enter number : ";
    cin >> x;

    if(x <= 0.0) {
           cout << "We only serve Positive numbers here."
                          << endl;
           exit(1);
           }
    …
```

## 8.5 EXAMPLE PROGRAMS

### 8.5.1 Calculating some simple statistics

Problem:

You have a collection of data items; each data item consists of a (real) number and a character (m or f). These data represent the heights in centimetres of some school children. Example data:

```
140.5      f
148        m
137.5      m
133        f
```

```
129.5       m
156         f
…
```

You have to calculate the average height, and standard deviation in height, for the entire set of children and, also, calculate these statistics separately for the boys and the girls.

Specification:

1. The program is to read height and gender values from `cin`. The data can be assumed to be correct; there is no need to check for erroneous data (such as gender tags other than 'f' or 'm' or a height outside of a 1 to 2 metre range). The data set will include records for at least three boys and at least three girls; averages and standard deviations will be defined (i.e. no need to check for zero counts).

2. The input is to be terminated by a "sentinel" data record with a height 0 and an arbitrary m or f gender value.

3. When the terminating sentinel data record is read, the program should print the following details: total number of children, average height, standard deviation in height, number of girls, average height of girls, standard deviation of height of girls, number of boys, average height of boys, standard deviation of height of boys.

***Watch out for the fine print***

Note that this specification excludes some potential problems.

Suppose the specification did not mention that there would be a minimum number of boys and girls in the data set?

It would become *your* responsibility to realize that there then was a potential difficulty. If all the children in the sample were boys, you would get to the end of program with:

```
number_girls            0
sum_girls_heights       0.0
```

and would need to work out

```
average_girl_height = sum_girls_heights / number_girls;
```

Execution of this statement when `number_girls` was zero would cause your program to be terminated with arithmetic overflow. You would have to plan for more elaborate processing with `if` statements "guarding" the various output sections so that these were only executed when the data were appropriate.

Remember, program specifications are often drawn up by teams including lawyers. Whenever you are dealing with lawyers, you should watch out for nasty details hidden in the fine print.

***"sentinel" data***

The specification states that input will be terminated by a "sentinel" data record (dictionary: *sentinel* – 1) soldier etc posted to keep guard, 2) Indian-Ocean crab

with long eye-stalks; this use of sentinel derives from meaning 1). A sentinel data record is one whose value is easily recognized as special, not a normal valid data value; the sentinel record "guards" the end of input stopping you from falling over trying to read data that aren't there. Use of a sentinel data record is one common way of identifying when input is to stop; some alternatives are discussed in Chapter 9 where we use files of data.

## Program design

***Preliminary design***

First, it is necessary to check the formulae for calculating averages (means) and standard deviations!

Means are easy. You simply add up all the data values as you read them in and then divide by the number of data values. In this case we will get something like:

```
average_height = sum_heights / number_children;
```

The standard deviation is a little trickier. Most people know (or at least once knew but may have long forgotten) the following formula for the standard deviation for a set of values *xi*.

$$S = \sqrt{\frac{\sum_{i=1}^{N}\left(x_i - \bar{x}\right)^2}{N-1}}$$

| | |
|---|---|
| S | standard deviation |
| N | number of samples |
| $x_i$ | sample i |
| $\bar{x}$ | average value of samples |

i.e. to work out the standard deviation you sum the squares of the differences between sample values and the average, divide this sum by N-1 and take the square root.

The trouble with that formula is that you need to know the average. It suggests that you have to work out the value in a two step process. First, you read and store all the data values, accumulating their total as you read them in, and working out their average when all are read. Then in a second pass you use your stored data values to calculated individual differences, summing the squares of these differences etc.

Fortunately, there is an alternative version of the formula for the standard deviation that lets you calculate its value without having to store all the individual data elements.

$$S = \sqrt{\frac{\sum_{i=1}^{N}x_i^2 - N\bar{x}^2}{N-1}}$$

This formula requires that as well as accumulating a sum of the data values (for calculating the average), we need to accumulate a sum of the squares of the data values. Once all the data values have been read, we can first work out the average and then use this second formula to get the standard deviation.

***First iteration through design process***

If we are to use this second formula, the program structure is roughly:

```
initialize count to zero, sum of values to zero
    and sum of squares to zero
read in the first data element (height and gender flag)
while height ≠ 0
    increment count
    add height to sum
    add square of height to sum of squares
    read next data element

calculate average by dividing sum by number
use second formula to calculate standard deviation
```

Of course, we are going to need three counts, three sums, three sums of squares because we need to keep totals and individual gender specific values. We also need to elaborate the code inside the loop so that in addition to updating overall counts we selectively update the individual gender specific counts.

***Second iteration through design process***

We can now begin to identify the variables that we will need:

***Data***

```
    three integers      children, boys, girls
    three doubles       cSum, bSum, gSum
    three doubles       cSumSq, bSumSq, gSumSq

    double              height
    char                gender_tag

    double              average
    double              standard_dev
```

The averages and standard deviations can be calculated and immediately printed out; so we can reuse the same variables for each of the required outputs.

***Loop code***

The code of the loop will include the selective updates:

```
while height ≠ 0
    increment count
    add height to sum
    add square of height to sum of squares
    if(female)
           update girls count, girls sum, girls sumSq
    else
           update boys count, boys sum, boys sumSq
    read next data element
```

and the output section needs elaboration:

***Output code***

```
calculate overall average by dividing cSum by children
use second formula to calculate standard deviation
    using values for total data set
```

```
Output details for all children

calculate girls' average by dividing gSum by girls
use second formula to calculate standard deviation
    using values for girls data set

Output details for girls

… similarly for boys
```

Some simple data would be needed to test the program. The averages will be easy to check; most spreadsheet packages include "standard deviation" among their standard functions, so use of a spreadsheet would be one way to check the other results if hand calculations proved too tiresome.

## Implementation

Given a fairly detailed design, implementation should again be straightforward. The program needs the standard `sqrt()` function so the math library should be included.

```
#include <iostream.h>
#include <math.h>

int main()
{
    int           children, boys, girls;
    double        cSum, bSum, gSum;
    double        cSumSq, bSumSq, gSumSq;

    double        height;
    char          gender_tag;

    children = boys = girls = 0;
    cSum = bSum = gSum = 0.0;
    cSumSq = bSumSq = gSumSq = 0.0;

    cin >> height >> gender_tag;

    while(height != 0.0) {
           children++;
           cSum += height;
           cSumSq += height*height;

           if(gender_tag == 'f') {
                  girls++;
                  gSum += height;
                  gSumSq += height*height;
                  }
           else {
                  boys++;
                  bSum += height;
```

***Conditional updates of separate counters***

```
                        bSumSq += height*height;
                        }

                cin >> height >> gender_tag;
                }
```

***Variables declared in the body of the block***

```
        double          average;
        double          standard_dev;

        cout << "The sample included " << children
                        << " children" << endl;
        average = cSum / children;
        cout << "The average height of the children is "
                        << average;

        standard_dev = sqrt(
                        (cSumSq - children*average*average) /
                        (children-1));

        cout << "cm, with a standard deviation of " <<
                        standard_dev << endl;

        cout << "The sample included " << girls
                        << " girls" << endl;

        average = gSum / girls;
        cout << "The average height of the girls is "
                        << average;

        standard_dev = sqrt(
                        (gSumSq - girls*average*average) /
                        (girls-1));

        …
        …

        return 0;
    }
```

Note, in this case the variables `average` and `standard_dev` were not defined at the start of `main()`'s block; instead their definitions come after the loop at the point where these variables are first used. This is typical of most C++ programs. However, you should note that some people feel quite strongly that all variable declarations should occur at the beginning of a block. You will need to adjust your coding style to meet the circumstances.

***Expression passed as argument for function call***

Function `sqrt()` has to be given a double value for the number whose root is needed. The call to `sqrt()` can involve any expression that yields a double as a result. The calls in the example use the expression corresponding to the formula given above to define the standard deviation.

***Test run***

When run on the following input data:

```
135         f
140         m
139         f
```

```
151.5       f
133         m
148         m
144         f
146         f
142         m
144         m
0           m
```

The program produced the following output (verified using a common spreadsheet program):

```
The sample included 10 children
The average height of the children is 142.25cm, with a
standard deviation of 5.702095
The sample included 5 girls
The average height of the girls is 143.1cm, with a standard
deviation of 6.367888
The sample included 5 boys
The average height of the boys is 141.4cm, with a standard
deviation of 5.549775
```

***Dubious trailing digits!***

The standard deviations are printed with just a few too many digits. Heights measured to the nearest half centimetre, standard deviations quoted down to about Angstrom units! The number of digits used was determined simply by the default settings defined in the iostream library. Since we know that these digits are spurious, we really should suppress them.

***Formatting***

There are a variety of ways of controlling the layout, or "format", of output if the defaults are inappropriate. They work, but they aren't wildly convenient. Most modern programs display their results using graphics displays; their output operations usually involve first generating a sequence of characters (held in some temporary storage space in memory) then displaying them starting at some fixed point on the screen. Formatting controls for sending nice tabular output to line printers etc are just a bit *passe*.

***iomanip***

Here we want to fix the precision used to print the numbers. The easiest way is to make use of an extension to the standard `iostream` library. The library file `iomanip` contains a number of extensions to standard iostream. Here, we can use `setprecision()`. If you want to specify the number of digits after the decimal point you can include `setprecision()` in your output statements:

```
cout << setprecision(2) << value;
```

This would limit the output to two fraction digits so you would get numbers like 12.25 rather than 12.249852. If your number was too large to fit, e.g. 35214.27, it will get printed in "scientific" format i.e. as 3.52e4.

***precision is a "sticky" format setting***

Once you've told `cout` what precision to use, that is what it will use from that point on. This can be inconvenient. While we might want only one fraction digit for the standard deviations, we need more for the averages (if we set `setprecision(1)`, the an average like 152.643 may get printed as 1.5e2). Once you've started specifying precisions, you are committed. You are going to have to do it everywhere. You will need code like:

```
cout << "The average height of the children is "
          << setprecision(3) << average;

standard_dev = sqrt(
          (cSumSq - children*average*average) /
          (children-1));

cout << "cm, with a standard deviation of " <<
          setprecision(1) << standard_dev << endl;

cout << "The sample included " << girls <<
          " girls" << endl; // integer, no precision
```

(You must `#include <iomanip.h>` if you want to use `setprecision()`.) On the whole, it is best not to bother about these fiddly formats unless you really must.

### 8.5.2 Newton's method for roots of polynomials

Problem:

You have to find the root of a polynomial; that is given some function of x, like $13.5*x^4 - 59*x^3 - 28*x^2 + 16.5*x + 30$, you must find the values of x for which this is 0.

The method is similar to that used earlier to find square roots. You keep guessing, in a systematic controlled way, until you are happy that you are close enough to a root. Figure 8.1 illustrates the basis for a method of guessing systematically. You have to be given two values of x that bracket (lie on either side of) a root; for one ("posX") the polynomial has positive value, the value is negative for the other ("negX"). Your next guess for x should be mid-way between the two existing guesses.

If the value of the polynomial is positive for the new x value, you use this x value to replace the previous posX, otherwise it replaces the previous negX. You then repeat the process, again guessing halfway between the updated posX, negX pair. If the given starting values did bracket a single root, then each successive guess should bring you a little closer.

Specification:

1. The program is to find a root of the polynomial $13.5*x^4 - 59*x^3 - 28*x^2 + 16.5*x + 30$.

2. The program is to take as input guesses for two values of x that supposedly bracket a root.

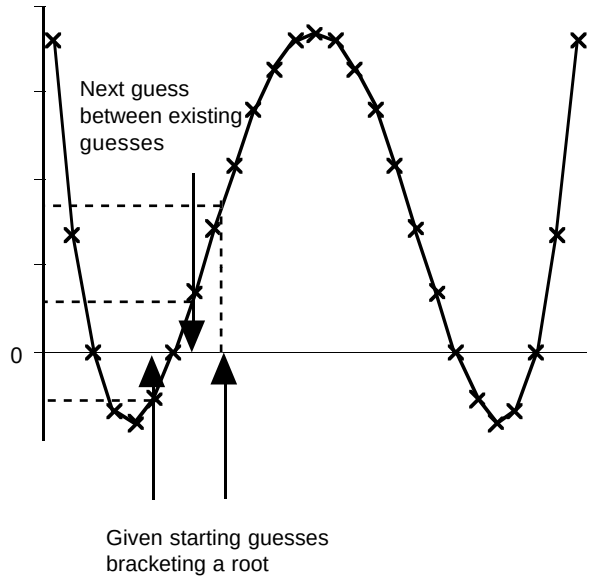



Figure 8.1 Principles of root finding!

3. The program is to verify that the polynomial does have values of opposite sign for the given x values. If the polynomial has the same sign at the two given values (no root, or a pair of roots, existing between these values) the program is to print a warning message and then stop.

4. The program is to use the approach of guessing a new value mid way between the two bracketing values, using the guessed value to replace one of the previous values so that it always has a pair of guesses on either side of the root.

5. The program is to print the current estimate for the root at each cycle of the iterative process.

6. The loop is to terminate when the guess is sufficiently close to a root; this should be taken as meaning the value for the polynomial at the guessed x is less than 0.00001.

## Program design

A first iteration through the design process gives a structure something like the following:

***Preliminary design***

```
prompt for and read in guesses for the two bracketing x
    values
calculate the values of the function at these two xs
if signs of function are same
    print warning message
    exit
initialize negX with x value for which polynomial was -ve
```

```
initialize posX with x value for which polynomial was +ve

pick an x midway between posX and negX
evaluate polynomial at x

while value of polynomial exceeds limit
    print current guess and polynomial value
    if value of polynomial is negative
           replace negX by latest guess
    else replace posX by latest guess
    set x midway between updated posX, negX pair
    evaluate polynomial at x

print final guess
```

***Detailed design***

The data? Rather a large number of simple variables this time. From the preliminary design outline, we can identify the following:

```
posX, negX          the bracketing values
g1, g2              the initial guesses for bracketing values
x                   the current guess for the root

fun1, fun2          the values of the polynomial at g1 and g2

fun3                the value of the polynomial at x
```

all of these would be doubles and belong to the main routine.

How to evaluate the polynomial?

Really, we want a function to do this! After all, the polynomial is "a function of x". But since we can't yet define our own functions we will have to have the code written out at each point that we need it. Exercise 2 in Chapter 7 presented some alternative ways of evaluating polynomial functions. Here, we will use the crudest method! The value of the polynomial for a specific value of x is given by the expression:

```
13.5*pow(x,4) - 59*pow(x,3) - 28*pow(x,2) + 16.5*x + 30
```

What is the simplest way of checking whether `fun1` and `fun2` have the same sign?

You don't need a complex boolean expression like:

```
(((fun1 < 0.0) && (fun2 < 0.0)) ||
           ((fun1 > 0.0) && (fun2 > 0.)))
```

All you have to do is multiply the two values; their product will be positive if both were positive or both were negative.

## Implementation

The implementation is again straightforward. The code would be:

```
#include <stdlib.h>
#include <iostream.h>
#include <math.h>

int main()
{
    double posX, negX;

    double fun1, fun2, fun3;

    cout << "Enter guesses that bracket a root" << endl;
    double g1, g2;
    cin >> g1;
    cin >> g2;
    double x = g1;
    fun1 = 13.5*pow(x,4) - 59*pow(x,3)- 28*pow(x,2)
                + 16.5*x + 30;
    x = g2;
    fun2 = 13.5*pow(x,4) - 59*pow(x,3)- 28*pow(x,2)
                + 16.5*x + 30;

    if((fun1*fun2) > 0.0) {
           cout << "Those values don't appear to bracket a"
                           " root" << endl;
           exit(1);
           }
    negX = (fun1 < 0.0) ? g1 : g2;
    posX = (fun1 > 0.0) ? g1 : g2;

    x = 0.5*(negX + posX);
    fun3 = 13.5*pow(x,4) - 59*pow(x,3)- 28*pow(x,2)
                + 16.5*x + 30;
    while(fabs(fun3) > 0.00001) {
           cout << x << ", " << fun3 << endl;
           if(fun3 < 0.0) negX = x;
           else posX = x;
           x = 0.5*(negX + posX);
           fun3 = 13.5*pow(x,4) - 59*pow(x,3)- 28*pow(x,2)
                           + 16.5*x + 30;
           }

    cout << "My final guess for root : " << endl;
    cout << x << endl;

    return EXIT_SUCCESS;
}
```

Convince yourself that the code does initialize `posX` and `negX` correctly from the guesses `g1` and `g2` with its statements:

```
    negX = (fun1 < 0.0) ? g1 : g2;
    posX = (fun1 > 0.0) ? g1 : g2;
```

## 8.6 WHAT IS THE LARGEST? WHAT IS THE SMALLEST?

In example 8.5.1, the statistics that had to be calculated were just mean values and standard deviations. Most such problems also require identification of minimum and maximum values.

Suppose we needed to find the heights of the smallest and tallest of the children. The program doesn't require much extension to provide these statistics. We have to define a couple more variables, and add some code in the loop to update their values:

```
int main()
{
    int           children, boys, girls;
    double        cSum, bSum, gSum;
    double        cSumSq, bSumSq, gSumSq;

    double        height;
    char          gender_tag;

    double        smallest, tallest;

    children = boys = girls = 0;
    …

    cin >> height >> gender_tag;

    while(height != 0.0) {
          children++;
          cSum += height;
          cSumSq += height*height;

          smallest = (height < smallest) ?
                              height :
                                    smallest;

          tallest = (height > tallest) ?
                              height :
                                    tallest;
```

Of course, we need `smallest` and `tallest` to have some initial values. The initial value of `tallest` would have to be less than (or equal) to the (unknown) height of the tallest child, that way it will get changed to the correct value when the data record with the maximum height is read. Similarly, `smallest` should initially be greater than or equal to the height of the smallest child.

How should you chose the initial values?

***Initialize minima and maxima with actual data***

Generally when searching for minimum and maximum values in some set of data, the best approach is to use the first data value to initialize both the minimum and maximum values:

```
    cin >> height >> gender_tag;
```

```
smallest = tallest = height;

while(height != 0.0) {
        children++;
        …
```

This initialization guarantees that initial values are in appropriate ranges.

It would have been a little harder it we had needed the heights of the smallest boy and of the smallest girl. These can't both be initialized from the first data value. The first value might represent a small boy, smaller than any of the girls but not the smallest of the boys; if his height was used to initialize the minimum height for girls then this value would never be corrected.

***Initialize from constants representing extreme values***

If you can't just use the first data value, then use a constant that you "know" to be appropriate. The problem specification indicated that the children were in the height range from 1 metre to 2 metres; so we have suitable constants:

```
// Children should be taller than 100 cm
// and less than 200 cm
const double LowLimit = 100;
const double HiLimit = 200;
…
…
smallest = HiLimit;
tallest = LowLimit;

cin >> height >> gender_tag;
while(height != 0.0) {
    …
```

Note: `smallest` gets initialized with `HiLimit` (it is to start *greater* than or equal to smallest value and `HiLimit` should be safely greater than the minimum); similarly `tallest` gets initialized with `LowLimit`.

You don't have to define the named constants; you could simply have:

```
smallest = 100;
tallest = 200;
```

but, as noted previously, such "magic numbers" in the code are confusing and often cause problems when you need to change their values. (You might discover that some of the children were aspiring basket ball players; your 200 limit might then be insufficient).

***Don't go guessing limits***

In this example, the specification for the problem provided appropriate values that could be used to initialize the minimum and maximum variables. Usually, this is not the case. The specification will simply say something like "read in the integer data values and find the maximum from among those that ...". What data values? What range? What is a plausible maximum?

Very often beginners guess.

They guess badly. One keeps seeing code like the following:

```
min_val = 10000;
max_val = 0;
```

```
cin >> val >> code_char;
while(code_char != 'q') {
    min_val = (val < min_val) ? val : min_val;
    …
```

Who said that 10000 was large? Who said that the numbers were positive? Don't guess. If you don't know the range of values in your data, initialize the "maximum" with the least (most negative) value that your computer can use; similarly, initialize the "minimum" with the greatest (largest positive) value that your computer can use.

***Limits.h***

The system's values for the maximum and minimum numbers are defined in the header file limits.h. This can be #included at the start of your program. It will contain definitions (as #defines or const data definitions) for a number of limit values. The following table is part of the contents of a limits.h file. `SHRT_MAX` is the largest "short integer" value, `LONG_MAX` is the largest (long) integer and `LONG_MIN` (despite appearances) is the most negative possible number (-2147483648). If you don't have any better limit values, use these for initialization.

```
#define SHRT_MIN          (~32767)
#define SHRT_MAX          32767
#define USHRT_MAX         0xFFFF

#define LONG_MIN          (~2147483647L)
#define LONG_MAX          2147483647L
#define ULONG_MAX         0xFFFFFFFFL
```

Once again, things aren't quite standardized. On many systems but not all, the limits header file also defines limiting values for doubles (the largest, the smallest non-zero value etc).

***Unsigned values***

The list given above included limit values for "unsigned shorts" and "unsigned longs". Normally, one-bit of the data field used by an integer is used in effect for the ± sign; so, 16-bits can hold at most a 15-bit value (numbers in range -32768 to +32767). If you know that you only need positive integers, then you can reclaim the "sign bit" and use all 16 bits for the value (giving you a range 0…65535).

Unsigned integers aren't often used for arithmetic. They are more often used when the bit pattern isn't representing a number – it is a bit pattern where specific bits are used to encode the true/false state of separate data values. Use of bit patterns in this way economises on storage (you can fit the values of 16 different boolean variables in a single unsigned short integer). Operations on such bit patterns are illustrated in Chapter X.

The definitions for the "`MIN`" values are actually using bit operations. That character in front of the digits isn't a minus sign; it is ~ ("tilde") – the bitwise not operator. Why the odd bit operations? All will be explained in due time!

## EXERCISES

Loops and selection statements provide considerable computational power. However, the range of programs that we can write is still limited. The limitation is now due mainly to a

lack of adequate data structures. We have no mechanism yet for storing data. All we can do is have a loop that reads in successive simple data values, like the children's heights, and processes each data element individually combining values to produce statistics etc. Such programs tend to be most useful when you have large numbers of simple data values from which you want statistics (e.g. heights for all year 7 children in your state); but large amounts of data require file input. There are just a few exercises here; one or two at the end of Chapter 9 are rather similar but they use file input.

1. Implement a working version of the Calculator program and test its operation.

2. Extend the calculator to include a separate memory. Command 'm' copies the contents of the display into the memory, 'M' adds the contents of the display to memory; similar command 'r' and 'R' recall the memory value into the display (or add it to the display).

3. Write a program that "balances transactions on a bank account" and produces a financial summary.

   The program is to prompt for and read in the initial value for the funds in the account. It is then to loop prompting for and reading transactions.
   Transactions are entered as a number and a character, e.g.

```
128.35   u
 79.50   f
 10.60   t
 66.67   r
  9.80   e
213.50   j
 84.30   f
 66.67   r
 …
```

   The character codes are:

   debits (expenditures):

| | |
|---|---|
| u | utilities (electricity, gas, phone) |
| f | food |
| r | rent |
| c | clothing |
| e | entertainment |
| t | transport |
| w | work related |
| m | miscellaneous |

   credits (income)

| | |
|---|---|
| j | job |
| p | parents |
| l | loan |

   Credit entries should increase available funds, debit entries decrease the funds.
   Entry of the value 0 (zero) is to terminate the loop (the transaction code with a 0 entry will be arbitrary).

When the loop has finished, the program is to print the final value for the funds in the account. It is also to print details giving the total amounts spent in each of the possible expenditure categories and also giving these values percentages of the total spending.
The program should check that the transaction amounts are all greater than zero and that the transaction codes are from the set given. Data that don't comply with these requirements are to be discarded after a warning message has been printed; the program is to continue prompting for and processing subsequent data entires.

4. Extend the childrens' heights program to find the heights of the smallest boy and the smallest girl in the data set.

5. Modify the heights program so that it can deal correctly with data sets that are empty (no children) or have children who all are of the same gender. The modified program should print appropriate outputs in all cases (e.g. "There were no girls in the data given").

6. Write a program to find a root( for polynomial of a given maximum degree 4, i.e. a function of the form $c4\ x^4 + c3\ x^3 + c2\ x^2 + c1\ x + c0$ for arbitrary values of the coefficients c4, c3, c2, c1, and c0.

   The formula for evaluating the polynomial at a given value of x is

   val = (((c4 * x + c3)*x + c2)*x + c1)*x + c0

   The program is to prompt for and read the values of the coefficients.

   The program should then prompt for a starting x value, an increment, and a final x value and should tabulate the value the polynomial at each successive x value in the range. This should give the user an idea of where the roots might be located.

   Finally, the program should prompt for two x values that will bracket a root and is to find the root (if any) between these given values.