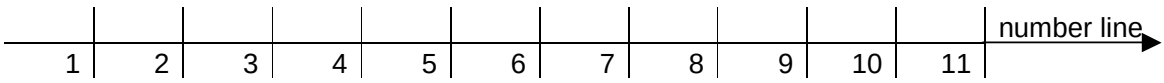
3 Selections

3.1 Boolean Expressions

We can represent whole numbers as points along a number line.



We can see from the number line that, for example, 3 is less than 4 and 4 is more than 3.

Is-less-than is represented by the symbol < and is-more-than is represented by >. We write 3 < 4 and 4 > 3.

The expression (3 < 4) has a value; this value is TRUE. Similarly, (4 > 3) has the value TRUE.

The expression (4 < 3) has the value FALSE. (How can four be less than three?). Similarly, (3 > 4) has the value FALSE.

Expressions which can only have either TRUE or FALSE as their values are known as Boolean expressions.

The symbols < and > are examples of relational operators. In C the relational operators are:

<	is less than
<=	is less than or equal to
>	is more than
>=	is more than or equal to
==	is equal to
!=	is not equal to

Figure 3.1 - the relational operators.

So, for example,

(4 <= 4) is TRUE
(4 == 4) is TRUE
(4 != 4) is FALSE

Notice that, where two symbols are used together, the = symbol is always the last one written, and that there are no spaces between them.

3.2 One-Way Selection

The purpose of the first program in this chapter is to input a number representing a mark obtained in an examination, and, if that mark is 40 or more, to output the message "Passed.". Here is the program.

```
/* program 3.1 - inputs an exam mark, outputs Pass if the */
/*               mark is 40 or more.                      */

#include <stdio.h>

int intNumberRead(char []);

enum { passMark = 40 };

int main()
{
  int mark = intNumberRead("Examination mark? ");
  if (mark >= passMark)
    printf("Passed.");
  printf("\n");
  return 0;
}

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

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

Three examples of program runs are

(1)
```
Examination mark? 41
Passed.
```

(2)
```
Examination mark? 40
Passed.
```

(3)
```
Examination mark? 39
```

The line

```
enum { passMark = 40 };
```

defines *passMark* to be the *int* value 40. Since the value of *passMark* cannot be subsequently changed by a program statement, it is an example of a constant. The C keyword *enum* stands for enumeration. An enumeration is just a list of named items, separated by commas, and enclosed within curly brackets or braces. So, *passMark* is an example of an enumerated constant. An enumerated constant is always an integer value.

The boolean expression *(mark >= passMark)*is either TRUE (when mark is 41 for example) or FALSE (when mark is 39 for example). So, the statement

```
if (mark >= passMark)
   printf("Passed.");
```

says that *Passed.* is displayed only if *(mark >= passMark)* is TRUE. If *(mark >= passMark)* is FALSE, then no consequent action is specified. This is why, in example 3 of the program run shown above, no result is shown when the value 39 was entered by the program user.

3.3 TWO-WAY SELECTION

The purpose of the next program is to display *Passed.* if the exam mark entered is 40 or more, but to output *Failed.* if the exam mark is less than 40.

```
/* program 3.2 - inputs an exam mark, outputs Pass    */
/*               if the mark is 40 or more, otherwise */
/*               outputs Fail.                        */

#include <stdio.h>

int intNumberRead(char []);

enum { passMark = 40 };

int main()
{
  int mark = intNumberRead("Examination mark? ");

  if (mark >= passMark)
    printf("Passed.");
  else
    printf("Failed.");
  printf("\n");
  return 0;
}

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

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

Three examples of program runs are

(1)
```
Examination mark? 41
Passed.
```

(2)
```
Examination mark? 40
Passed.
```

(3)
```
Examination mark? 39
Failed.
```

If the boolean expression *(mark >= passMark)* is TRUE then the message *Passed.* is displayed; but if *(mark >= passMark)* is FALSE then *Failed.* is displayed. Either one message or the other is selected for output depending on whether *(mark >= passMark)* is TRUE or FALSE.

```
if (mark >= passMark)
  printf("Passed.");    <------  done if (mark >= passMark) is TRUE
else
  printf("Failed.");    <------  done if (mark >= passMark) is FALSE
```

Both *if* and *else* are C keywords. *else* cannot be used without a corresponding *if*. Notice that a semi-colon does not follow a boolean expression such as *(mark >= passMark)*; neither does a semi-colon follow the else keyword.

We follow the convention that

- statements written below an *if* are indented by two spaces
- *else* is written directly in line underneath the *if*
- statements written below an *else* are also indented by two spaces

Indentation helps us to see logical structure "at a glance".

Exercise 3.1

1 Write a program which will prompt the user to enter an integer value to represent a person's age, input their response and output whether or not the person is entitled to hold a provisional driving licence. Assume that the minimum age for a provisional licence holder is 17. Your program should use an enumerated constant for the minimum age.

2 To become a member of my millionaires club you have to have about one million pounds, or more, in your bank account. Write a program which will input a number to represent the amount of money in a person's bank account and to output whether that person is eligible for membership of my club.

3 The area of green algae which covers a pond doubles every day. Write a program which will input the area of the pond and the area currently covered by the green algae, and will either output the warning "The pond will be completely covered tomorrow!" if the area currently covered is more than half of the area of the pond, or output the reassuring message "Nothing to worry

about!" if otherwise. Use integer values to represent the area of the pond and green algae.

3.4 MULTI-WAY SELECTION

Grades are awarded to students depending on the percentage mark they obtain in a test. Students who obtain 60% or more gain a Merit. Students who obtain between 40% and 59% gain a Pass. Students who obtain less than 40% are graded Refer. The purpose of the next program is to input an integer value to represent a percentage mark and to output the corresponding grade: Merit, Pass, or Refer.

```
/* program 3.3 - inputs a percentage mark, outputs the */
/*               corresponding grade.                  */

#include <stdio.h>

int intNumberRead(char []);

enum { passMark = 40, meritMark = 60 };

int main()
{
  int mark = intNumberRead("Examination mark? ");

  if (mark >= meritMark)
    printf("Merit.");
  else if (mark >= passMark)
    printf("Passed.");
  else
    printf("Failed.");
  printf("\n");
  return 0;
}

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

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

Let us examine the program to see what happens for various values of *mark*.

Suppose *mark* has the value 65. Then *(mark >= meritMark)* is TRUE and *Merit.* is displayed.

Suppose now *mark* has the value 50. Then *(mark >= meritMark)* is FALSE but *(mark >= passMark)* is TRUE. So, *Pass.* is displayed.

And again, suppose *mark* now has the value 30. Then, *(mark >= meritMark)* is FALSE and *(mark >= passMark)* is FALSE. Therefore, *Refer.* is displayed.

The following table shown below in Figure 3.2 illustrates the point.

case	mark	(mark >= meritMark)	(mark >= passMark	Outcome
1	65	TRUE	not tested	Merit
2	50	FALSE	TRUE	Pass
3	30	FALSE	FALSE	Refer

Figure 3.2 - if mark is 65, then the first boolean expression, (mark >= meritMark) is TRUE and consequently no other boolean expression is evaluated. If mark is 50 then the first boolean expression is FALSE but the second boolean expression (mark >= passMark) is TRUE and consequently Pass is displayed. If mark is 30, then none of the boolean expressions are TRUE and Refer is displayed.

So, understanding a sequence of else if's is easy: look down the sequence of Boolean expressions, in the order written, and stop at the first one which is TRUE, then do the consequent action(s), and then skip to the end of the if ... else if ... else ... construction; if none of the boolean expressions is TRUE, then do the actions following the else (if any).

3.5 Boundary Testing

Let us look again at the selection statement in program 3.3.

```
if (mark >= meritMark)
  printf("Merit");
else if (mark >= passMark)
  printf("Passed.");
else
  printf("Failed.");
printf("\n");
```

Take the first Boolean expression *(mark >= meritMark).* We know that *meritMark == 60*. *meritMark* represents a boundary value because if *mark* is just over 60, then *Merit.* would be displayed, but if *mark* is just under 60, then *Pass.* would be displayed.

And again. Take the second Boolean expression *(mark >= passMark).* We know that *passMark == 40*. If *mark* is just over 40, then *Pass.* would be displayed; if *mark* is just under 40, then *Refer.* would be displayed. *passMark* is an example of a boundary value.

A boundary occurs whenever a small change in the value of a variable causes a large change in the behaviour of a program. For example, the small change in *mark* from 39 to 40 causes the message to be displayed to be changed from *Refer.* to *Pass.*

The purpose of testing a program is to locate errors. A useful strategy is to choose data values just under, just on and just over each boundary (because experience has shown that errors sometimes occur around these points). The data values chosen to test a program are

documented in a test plan. A test plan is shown in Figure 3.3.

TEST PLAN

Author: Terry Marris **Date:** 13 February
Program: Program 3.3

Test Number	Test Data	Reason	Expected Result
1	mark = 61	just over meritMark	Merit. shown
2	mark = 60	just on meritMark	Merit. shown
3	mark - 59	just under meritMark	Pass. shown
4	mark = 41	just over passMark	Pass. shown
5	mark = 40	just on passMark	Pass. shown
6	mark = 39	just under passMark	Refer. shown

Figure 3.3 - test plan for program 3.3.

This test plan shows that program 3.3 is to be tested 6 times. In the first test, the value input to *mark* is going to be 61 because that is just over the *meritMark* boundary; the anticipated result is that *Merit.* is going to be displayed on the screen. And similarly for tests 2, 3, 4, 5 and 6. Knowing what results to expect is essential, because, otherwise, how would you know whether your program is working correctly?

3.6 Test Logs

A test plan outlines your intentions for testing a program. A test log records what actually happened when you tested your program according to your test plan. An example of a test log is shown in Figure 3.4.

TEST LOG

Author: Terry Marris **Date:** 13 February
Program: Program 3.3
Time: 10:30

Test Number	Actual Result
1	Merit. shown
2	Merit. shown
3	Pass. shown
4	Pass. shown
5	Pass. shown
6	Refer. shown

Figure 3.4 - test log for program 3.3.

The test numbers correspond to the test numbers used in the test plan for program 3.3.

A test log is a factual, truthful, honest, historical account of your program testing, errors and all! If your actual results do not match the expected results, then a mistake has been made

somewhere. If you discover a mistake then you either document it (by discussing it in a report or mentioning it where it can be seen) or you correct it. It is far better to acknowledge errors in your program than to pretend they never occurred.

Exercise 3.2

1. Explain, with the aid of examples, the difference between the two C operators = and ==.

2. Amend program 3.3 so that if a mark greater than 100 or less than 0 is entered, the message "Invalid mark - mark should be between 0 and 100 inclusive." is displayed. Construct a test plan for the program then test your program according to your test plan. Correct your program if necessary. Do not forget to complete a test log as you test your program.

3.7 Boolean Variables and Logical Operators

The Right One is a dating agency which attempts to introduce each client to a compatible partner of the opposite sex. One particular client is looking for a partner who is between the ages of 25 and 30 inclusive. The purpose of the next program is to input a potential partner's age details and to output whether they match the age requirements.

```
/* program 3.4 - decides whether a person would make a */
/*               suitable partner on the basis of age. */

#include <stdio.h>

int intNumberRead(char []);

enum { minimumAge = 25, maximumAge = 30 };

int main()
{
  int age = intNumberRead("How old are you? ");
  int ageIsOK = (age >= minimumAge) && (age <= maximumAge);

  if (ageIsOK)
    printf("You will do!");
  else
    printf("Sorry - you are not suitable.");
  printf("\n");
  return 0;
}

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

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

Examples of program runs are:

```
(1)  How old are you? 24
     Age is not suitable.

(2)  How old are you? 25
     Age is OK.

(3)  How old are you? 26
     Age is OK.

(4)  How old are you? 29
     Age is OK.

(5)  How old are you? 30
     Age is OK.

(6)  How old are you? 31
     Age is not suitable.
```

Let us examine the statement

```
int ageIsOK = (age >= minimumAge) && (age <= maximumAge);
```

Working from right to left: *(age <= maximumAge)* is a Boolean expression with value either TRUE or FALSE. *(age >= minimumAge)* is also a Boolean expression. The double ampersand, *&&*, means and-at-the-same-time. So, the whole expression

```
(age >= minimumAge) && (age <= maximumAge)
```

which says that *age* is greater than (or equal to) *minimumAge* and-at-the-same-time *age* is less than (or equal to) *maximumAge*, is also a Boolean expression with value either TRUE or FALSE.

In C, TRUE is represented by a non-zero value, usually one; FALSE is represented by zero. So we can assign the value of the boolean expression *(age >= minimumAge) && (age <= maximumAge)* to a variable of type *int*. This is what we have done here:

```
int ageIsOK = (age >= minimumAge) && (age <= maximumAge);
```

Then we can go on to write

```
if (ageIsOK)
  printf("You will do!");
else
  printf("Sorry - you are not suitable.");
```

where the statement is read as if ageIsOK is TRUE then printf "You will do" otherwise printf "Sorry - you are not suitable." ageIsOK is a Boolean expression with value either TRUE (that is, one) or FALSE (that is, zero).

We follow the convention that variables intended to represent boolean values include the

word *is* in their name. So, for example, *ageIsOK* is either TRUE, or it is not.

Returning to the Right One Dating Agency, another client is looking for a partner who is either rich or good looking (or possibly both rich and good-looking). The purpose of the next program is to input a potential partner's financial situation and appearance, and to output whether they meet the requirements for a suitable partner.

```
/* program 3.5 - decides whether a person would make a
     suitable partner on the basis of money and looks. */
#include <stdio.h>

int main()
{
  int isRich;
  int isGoodLooking;
  char reply[BUFSIZ];

  printf("Are you rich? ");
  gets(reply);
  isRich = (reply[0] == 'y') || (reply[0] == 'Y');

  printf("Are you good looking? ");
  gets(reply);
  isGoodLooking = (reply[0] == 'y') || (reply[0] == 'Y');

  if ((isRich) || (isGoodLooking))
    printf("You will do!");
  else
    printf("Sorry - I am not interested.");
  return 0;
}
```

Some examples of program runs are

(1)
```
Are you rich? y
Are you good looking? y
You will do!
```

(2)
```
Are you rich? Y
Are you good looking? n
You will do!
```

(3)
```
Are you rich? n
Are you good looking? Y
You will do!
```

(4)
```
Are you rich? N
Are you good looking? N
Sorry - I am not interested.
```

The lines

```
if ((isRich) || (isGoodLooking))
  printf("You will do!");
else
  printf("Sorry - I am not interested.");
```

say that if either *isRich* is TRUE OR *isGoodLooking* is TRUE (or both *isRich* and *isGoodLooking* are TRUE) then print *You will do*, otherwise, if neither *isRich* nor *IsGoodLooking* are TRUE, then print *Sorry - I am not interested*.

3.8 Documentation

Statements sandwiched between *if* and *else* are to be indented by two character spaces to help the human reader understand the structure at a glance. *else* is to be written in line directly below the *if*. If a statement sequence comprises several statements, then the braces could be positioned as shown below:

if (boolean-expression) {
statement;
statement;
...
}
else {
statement;
statement;
...
}

For example

```
if (currentTransaction > creditLimit) {
  printf("Credit limit exceeded - transaction refused.\n");
  printf("Inform customer.\n");
}
else {
  printf("Proceed with transaction.\n");
  printf("Check signature.\n");
}
```

Test Plans - see Figure 3.3 - are used to design and specify data values to test programs in a systematic manner. They are used to record items such as data values intended to show that the program works according to its specification, reasons for using particular test data values and the expected results of running the program with the test data.

Test Logs - see Figure 3.4 - are used to record the results of actually running programs with the test data. If the results actually obtained do not match the expected results, then a mistake has been made somewhere. Test logs must be written by hand at the time of testing and should include details such as test run number, time, date and actual result.

3.9 Programming Principles

Use enumerated constants and boolean expressions wherever their use makes the program logic clearer to the human reader.

Remember that program users might input characters in either lower or upper case; they should be allowed to do so. It is up to the programmer to take appropriate action.

With a sequence of else if's arrange the boolean expressions in the correct order to obtain the desired results.

Exercise 3.5

Remember to include test plans and test logs with every program you write.

1 An important principle is "if it is working, then do not fix it!". Writeand test a C program which will ask the user whether their program is working correctly, and to output appropriate advice.

2 The Right One is a dating agency which attempts to introduce each client to a compatible partner. One particular client is looking for a partner who is a non-smoker and who does not eat meat. Design, write and test a program which will input a potential partner's smoking and eating habits and output whether the person would make a suitable partner.

3 A television rental company will rent a tv to a customer if the customer is not on the bankrupt list AND the customer's name and address appear in the register of eligible voters AND the customer is currently employed AND has been in full-time employment for the past two years. Write a program which will ask the user to reply to questions such as "Is customer on the bankrupt list (y/n)?", "Is customer on the electoral role (y/n)?", "Is the customer currently employed (y/n)" and output whether a tv should be rented to the customer or not.

4 Design, write and test a program to the following specification. The program is to be a model for processing credit card transactions. Inputs: credit limit, credit used, bill for goods or services. Outputs: "OK" if credit used + bill is not greater than credit limit, otherwise "Transaction refused".

5 A year is a leap year if it is divisible by 4 but not by 100 except that years divisible by 400 are leap years. So, for example, 1992 is a leap year (divisible by 4) but 1900 is not a leap year (divisible by 4 and divisible by 100); but 2000 is a leap year (divisible by 400). Design, write and test a program which will display "Is a leap year" or "Is not a leap year" depending on the value of an integer input to represent a year. Hint: a number is divisible by 4 if the remainder after division by 4 is 0.

6 Electricity meter readings are five-digit numbers. If a meter reaches 99999,

then it restarts at 00000. Every three months the meter is read. The number of electricity units used in the previous three months is calculated from the present and previous meter readings. Usually, the previous meter reading is less than the present meter reading; however, when the meter reaches 99999 and restarts at 00000, the previous meter reading could be greater than the present meter reading. Design, write and test a program which will input the present and previous meter readings and output the number of units used together with the cost of the electricity used if the charge per unit is £0.08 and the standing charge is £9.50. (The standing charge is payable irrespective of the amount of electricity used.)