Making Decisions - Chapter Summary

1 The `if` Statement: Form 1

The `if` statement in C has the general form:

```
if (expression)
     if_body
```

If the expression evaluates to true, i.e. is non-zero, the if_body is executed, otherwise control passes on immediately to the first statement following the `if` statement, and the if_body is not executed. As with loop bodies, the if_body may be either a simple or a compound statement, and in the latter case, begins and ends with opening and closing braces { and } thus:

```
if (days > 365)
{
     year++;
     days = 0;
     printf("\nIt is now %d. Happy New Year!",year);
}
```

2 The if Statement, Form 2: the `if else` Construct

The `if else` construction is used for mutually-exclusive conditions and has the general form:

```
if (expression)
     if_body
else
     else_body
```

If the expression evaluates to true, if_body is executed, otherwise else_body is executed. The `else` part of the construction is equivalent to writing:

```
if (!expression)
     else_body
```

The `if else` construction is used in contexts where there are only two states possible, for instance true or false, on or off, odd or even, male or female:

```
if (male)
     gender = 'm';
else
     gender = 'f';
```

3 Nesting if Statements

An `if` statement may include loops or other nested `if` statements. It is important, however, to use successive levels of indentation to keep track of which statements depend on which conditions. Braces { and } can also be useful for marking off an if_body, even if the if_body does not consist of a compound statement, so that the braces are not strictly required:

```
     if (employed)
     {
          if (!holiday)
               if (!weekend)
               {
                    if (!sick)
                         printf("\nGet up and go to work!");
                    else
                         printf("\nTake an aspirin!");
               }
     }
     else
          printf("\nGo get a job!");
```

As a precaution, it is always worth checking that there are as many closing braces } as there are opening braces {.

4 The `if` Statement, Form 3: the `else if` Construct

Although the `if else` construction can only be used for mutually-exclusive conditions, the `else if` construction can be used for choosing between several alternatives. It has the general form:

```
     if (expression1)
          if_body
     else if (expression2)
          if_else_body1
```

```
else if (expression3)
     if_else_body2
...
else
     else_body
```

If expression1 is true, if_body is executed. If expression1 is false, but expression2 is true, if_else_body1 is executed. If both expression1 and expression2 are false, if_else_body2 is executed, etc.

In practice, there may be as many `else if` statements as required.

The following construction selects the name of one of the six so-called noble gasses on the basis of its atomic number:

```
if (noble_gas)
{
     if (atomic_number == 2)
          printf("Helium");
     else if (atomic_number == 10)
          printf("Neon");
     else if (atomic_number == 18)
          printf("Argon");
     else if (atomic_number == 36)
          printf("Krypton");
     else if (atomic_number == 54)
          printf("Xenon");
     else
          printf("Radon");    /* atomic number 86 */
}
```

5 The `switch` Statement

C offers another, more elegant, construction for choosing between multiple options: the `switch` statement. This has the general form:

```
switch (expression)
{
     switch_body
}
```

The expression evaluates to an integer or character value. The `switch_body` contains a list of `case`s, one for each of the possible values of the expression, and each of the general form:

```
case constant_value:
        statement;
        statement;
        break;
```

The `constant_value` will be a possible value for the expression, and each `case` will have a different constant. Note that the constant is followed by a colon `:.` There may be any number of statements to be executed for a given `case`, but a `switch` is often easier to read if function calls are used rather than multiple statements. (Functions calls are described in a following chapter.)

The final statement for each `case` is usually `break`; , which causes immediate escape from the `switch` statement.

If the `break` statement is not present at the end of a `case` statement, execution will automatically continue to the next `case`. This can be very useful, as it allows a single group of statements to be associated with several consecutive `case` constants if this is required.

One of the `case` statements in the `switch` statement may be an optional `default :`, which specifies the statements to be executed if none of the other options apply. Putting a `break` statement at the end of each `case` ensures that if any of the previous `cases` are true, the `default` statement will not be reached.

It is not an error if none of the `cases` match the current value of the expression. If there is no match and no `default` option, execution simply moves on to the next statement after the `switch` statement.

The `switch` statement in the following example might accompany a system menu offering the user five numbered options. Each of the first three `cases` uses a function call to load the particular program offered in the menu. If option 4 is selected, there is a match with `case 4`, but execution falls through to `case` 5, and the message **Access Denied.** will be displayed. Since there is no `break` statement at the end of `case 5`, execution will fall through to the `default` option, which displays the additional message on the next line, **Please try again**.

```
     switch (menu_opt)
     {
          case 1:   printf("\nLoading Word Processor.");
                    wordp();
                    break;

          case 2:   printf("\nLoading Spreadsheet.");
                    spreadsh();
                    break;

          case 3:   printf("\nLoading C Compiler.");
                    ccomp();
                    break;

          case 4:   /* Print default message also */
          case 5:   printf("\nAccess Denied.");

          default:  printf("\nPlease try again.");
                    break;
}
```

6 The Conditional Operator ?:

The conditional operator `?:` is a ternary operator, in that it takes three operands, each of which is an expression:

expression1 ? expression2 : expression3

If `expression1` is true, i.e. non-zero, the value of the whole expression is the value of `expression2`. If `expression1` is false, i.e. zero, the value of the whole expression is the value of `expression3`.

The conditional operator offers an elegant and succinct alternative to an `if else`. The following two constructions, for instance, are equivalent:

```
     (sales > costs) ? profit : loss;
     if (sales > costs)
          profit;
     else
          loss;
```

The conditional operator is typically used in contexts where a variable has to take either the greater or the smaller of two values. Note that brackets around `expression1`, the test expression, are not essential, but do improve readability.

```
main()
{
     double    best_price, cost1 = 19.99,
               cost2 = 18.70, cost3 = 17.99;

     best_price = (cost1 < cost2) ? cost1 : cost2;
     best_price = (best_price < cost3)
                        ? best_price : cost3;

     printf("\nThe cheapest price is %4.2f."
               , best_price);

}
```

When compiled and run, the above program will print out:

```
          The cheapest price is 17.99.
```

Note that the conditional operator has a higher precedence than the assignment operator =, so that in the second conditional operator statement, the old value of the variable `best_price` is compared with the value of `cost3` before the conditional operator evaluates the whole expression to the right of the assignment operator to the value of `cost3`, which is the new value that is then assigned to `best_price`.

7 Statements Affecting Flow of Control: `break`, `continue`, and `goto`

7.1 The break Statement

The `break` statement is used with loops and `switch` statements, and forces immediate exit from the loop or `switch` statement in which it is enclosed. The flow of control passes straight on to the next statement after the loop or `switch`.

If a `break` statement occurs in a nested loop or `switch`, the flow of control only passes outside the immediate loop or `switch`, and not out of the whole nested construction.

In addition to its common use in `switch` statements, the `break` statement is typically used with error-trapping within loops. In the following example, `break` statements are used to exit from the `for` loop before errors can cause incorrect or meaningless output:

```
for (min = low, max = hi, asc = min; asc <= max; asc++)
{
     if (asc < 0)
     {
          printf("\nError 1:");
          printf("ASCII code cannot be < 0 !");
          break;
     }
     if (min > max)
     {
          printf("\nError 2: minimum > maximum!");
          break;
     }
     printf("\nASCII code %d");
     printf(" corresponds to: %c",asc, asc);
}
```

While occasionally a useful tool, the `break` statement should be used sparingly to force exits from loops, as it can easily produce code that is hard to read and prone to errors.

7.2 The `continue` Statement

The `continue` statement can be used in loops, but unlike `break`, cannot be used in `switch` statements.

A `continue` statement causes the current iteration of a loop to be aborted, missing out all the remaining statements in the loopbody. In `while` and `do while` loops, the flow of control passes to the loop condition, ready for the next iteration. In `for` loops, however, flow of control passes to the loop-update term first, and then on to the loop condition.

The `continue` statement is typically used for introducing exceptions into loop structures. This allows the exceptions, errors for instance, to be discounted, without stopping the running of the loop for the other, acceptable cases.

In the following example, the `continue` statement aborts the loop if `number` is divisible by three. The `for` loop thus prints out all the numbers between a given minimum and maximum that are not divisible by three:

```
for (number = minimum; number <= maximum; number++)
{
     if (number % 3 == 0)
          continue;
     printf("\n%d", number);
}
```

7.3 goto and Labels

Although it does not strictly need one, C has a `goto` statement that causes the flow of control to jump unconditionally to a labelled statement. The rules for label names are the same as those for variable names, which are described in the summary to the Data Types chapter. Labels are separated from the statement they are marking by a colon `:`, and the labelled statement may occur anywhere in the function, either above or below the `goto` statement.

In the following example, the program jumps to the statement labelled `exclamation` if the input character is an exclamation mark `!`:

```
exclamation:printf("Help!  I have encountered a goto!");
     printf("Please type in a integer.");
     scanf("%d", &chosen);

     if (chosen == -1)
          goto exclamation;
```

7.4 Structured Programming

C is designed to encourage structured programming, which helps to make error trapping, maintenance and modification of programs all much easier. The `break`, `continue` and `goto` statements undoubtedly have their uses, but go against the principles of structured programming by causing flow of control to jump around the program in ways that are frequently hard to follow. They should therefore be employed sparingly, if at all. There are often other ways of achieving the same results without resorting to their use.