# STACKS

## WITH DYNAMIC MEMORY ALLOCATION

# ILLUSTRATING C REVISED EDITION

DONALD ALCOCK

In the previc
based on arr
depicted on
stack is introduce
dynamic storage
ay.



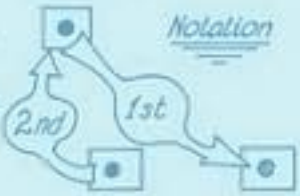
Here is the definition
eac

```
typedef struct Element
    {
        char c;
        struct Element   * next;
    }
        Element_Type, * Pointer_Type;
```

alias for 'struct Element'

alias for 'pointer to objects of type struct Element'

'List processing' is the art of diverting pointers by copying addresses from one pointer variable to another. To depict such operations we use the notation shown here. The fat arrow depicts a simple copying of contents in the direction of the arrow. The ordinal number (1st, 2nd, *etc.*) shows the order of operations needed to avoid overwriting.




Here is the definition of Push(). The copy operations are depicted opposite, together with sketches of the linkage before and after the copy operations:

pointer to objects of type Pointer_Type

```
void Push ( Pointer_Type * q, char ch )
{
    Pointer_Type p;
    p = ( Pointer_Type ) malloc ( sizeof ( Element_Type ) ) ;
    p -> c = ch ;
    p -> next = *q ;
    * q = p ;
}
```

2nd 1st and new element

An invocation of Push() demands two arguments of which the first nominates a pointer. For example, Push( &X, 'A' ) to push 'A' onto stack X.


ILLUSTRATING
C

# ILLUSTRATING C

## (ANSI/ISO VERSION)

Donald Alcock

Reigate Manual Writers

CAMBRIDGE UNIVERSITY PRESS

CAMBRIDGE
NEW YORK PORT CHESTER
MELBOURNE SYDNEY

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521468213



First published 1992
Reprinted (with corrections and in a larger format) 1993
Reprinted 1998
Re-issued in this digitally printed version (with corrections) 2008

*A catalogue record for this publication is available from the British Library*

ISBN 978-0-521-46821-3 paperback

# Acknowledgements

My warmest thanks to the following people without whom the job of writing this book would have been lonely and terrifying: Paul Burden, for patiently steering my rambling thoughts from nonsense to sense during many telephone conversations; Mike Ingham, for the same thing, and making it worth while to continue instead of throwing it all in the bin; Paul Shearing, for his enthusiasm and indispensable help with production; Andrew, my elder son, for help with just about everything.

# CONTENTS

# PREFACE

The original C programming language was devised by Dennis Ritchie. The first book on C, by Kernighan and Ritchie, came out in 1978 and remained the most authoritative and best book on the subject until their second edition, describing ANSI standard C, appeared in 1988. In all that time, and since, the availability and use of C has increased exponentially. It is now one of the most widely used programming languages, not only for writing computer systems but also for developing applications.

There are many books on C but not so many on ANSI standard C which is the version described here.

This book attempts three things:

- to serve as a text book for introductory courses on C aimed both at those who already know a computer language and at those entirely new to computing

- to summarize and present the syntax and grammar of C by diagrams and tables, making this a useful reference book on C

- to illustrate a few essential programming techniques such as symbol state tables, linked lists, binary trees, doubly linked rings, manipulation of strings, parsing of algebraic expressions.

For a formal appreciation of C ≈ its power, its advantages and disadvantages ≈ see the references given in the Bibliography. As an *informal* appreciation: all those I know who program in C find the language likeable and enjoy its power. Programming C is like driving a fast and powerful car. Having learned to handle the car safely you would not willingly return to the family saloon.

The hand-written format of this book has evolved over several years, and over six previous books on computers and programming languages. The pages contain the kind of diagram an able lecturer draws on the blackboard and annotates with encircled notes. Written text has been kept short and succinct. I have tried to avoid adverbs, cliches, jargon and unnecessarily formal language.

I hope the result looks friendly.

REIGATE

Surrey, U.K.

Donald Alcock

February 1992

# 1

# INTRODUCTION

The introduction starts with the concept of a stored program. The concept is second nature to anyone who has programmed anything on any computer in any language, but to a complete novice it can be difficult to grasp. So a simple program is written in English and then translated into C.

The chapter explains principles of running a C program on the computer. The explanation is sketchy because each implementation of C has different rules for doing so. Check the manuals for your own installation.

Finally the program is dissected, statement by statement.

# CONCEPTION

## *THE CONCEPT OF A STORED PROGRAM*

If you ask to borrow £5,ØØØ at 15.5% compound interest over 5 years, the friendly bank manager works out your monthly repayment, M, from the compound interest formula:

$$M = \frac{P \times R \times (1+R)^N}{12\,((1+R)^N - 1)}$$

Where:

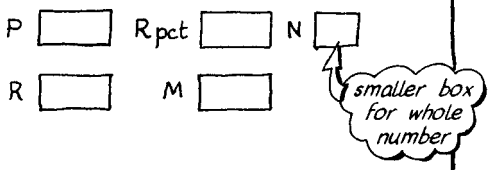
P represents the principal (£5ØØØ in this case)

R represents the rate of interest (Ø.155 is the *absolute* rate in the case of 15.5%)

N represents the number of years (5 in this case)

To work this out the friendly bank manager might use the following 'program' of instructions:

1. Get math tables or calculator ready
2. Draw boxes to receive values for P, Rpct, N. Also a box for the absolute rate, R, and a box for the repayment, M




3. Ask the client to state the three values: Principal (P), Rate percent (Rpct), Number of years (N)
4. Write these values in their respective boxes
5. Write in box R the result of Rpct/1ØØ. For Rpct use the value to be found in box Rpct (don't rub out the content of box Rpct)
6. Write in box M the result of the compound interest formula. Use for the terms P, R, N the values to be found in boxes P, R, N respectively (don't change anything in boxes P, R, N)
7. Confirm to the client the values in boxes P, Rpct, N and the monthly installment read from box M
8. Work out (12 × value in box M × value in box N) to tell tell the client how much will have to be repaid.

This program is good for any size of loan, any rate of interest, any whole number of years. Simply follow instructions 1 to 8 in sequence.

A computer can be made to execute such a program, but first you must translate it into a language the computer can understand. Here is a translation into the language called C.

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

int main(void)
{
    float P, Rpct, R, M;
    int N;
    printf ( "\nEnter: Principal, Rate%, No. of yrs.\n" );
    scanf ("%f %f %i", &P, &Rpct, &N );
    R = Rpct / 100;
    M = P * R * pow(1+R, N ) / ( 12 * ( pow(1+R, N) - 1));
    printf ("\n£%1.2f, @%11.2f %% costs £%1.2f over %i years", P, Rpct, M, N);
    printf ("\nPayments will total £%1.2f", 12*M*N );
    return 0;
}
```

(1) #include <stdio.h> #include <math.h>

(main program starts) int main(void) {

(2) float P, Rpct, R, M; int N;

(3) printf ( "\nEnter: Principal, Rate%, No. of yrs.\n" );

(4) scanf ("%f %f %i", &P, &Rpct, &N );

(5) R = Rpct / 100;

(6) M = P * R * pow(1+R, N ) / ( 12 * ( pow(1+R, N) - 1));

(7) printf ("\n£%1.2f, @%11.2f %% costs £%1.2f over %i years", P, Rpct, M, N);

(8) printf ("\nPayments will total £%1.2f", 12*M*N );

return 0; (main program ends) }

The above is a *program*. This particular program comprises:

- a set of *directives* to a *preprocessor*; each directive begins #
- a *function* called main() with one *parameter* named void.

A function comprises:

- a *header* conveying the function's name (main) followed by
- a *block*

A block { enclosed in braces } comprises:

- a set of *declarations* ('drawing' the little boxes)
- a set of *statements* (telling the processor what to do)

Each declaration and each statement is terminated with a semicolon.

The correspondence between the English program opposite, and the C program above, is indicated by numbers 1 to 8.

The C program is thoroughly dissected in following pages.

# REALIZATION

The program on the previous page should work on any computer that understands C.

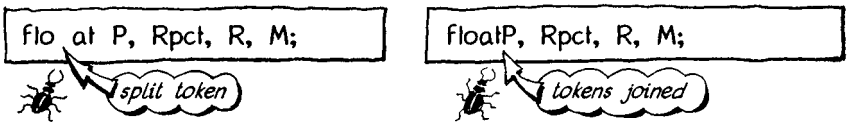
Unfortunately not all computer installations go about running C programs the same way; you have to have some understanding of the *operating system*, typical ones being Unix and DOS. You may be lucky and have an *integrated development environment* (IDE) such as that which comes with Turbo C or Microsoft C. In this case you do not have to learn much about Unix or DOS. You control Turbo C with mouse and menus; it really is easy to learn how.

Regardless of environment, the following essential steps must be taken before you can run the C program on the previous page.

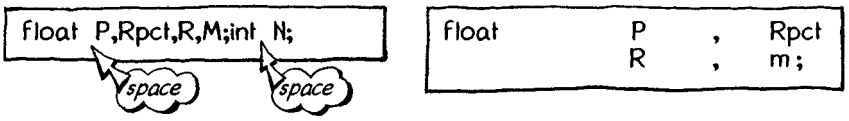
- Type. Type the program at the keyboard using the editing facilities available. If these are inadequate, discover if it is feasible to use your favourite word processor.

  When typing, don't type **main** as **MAIN**; corresponding upper and lower case letters are distinct in the C language (except in a few special cases).

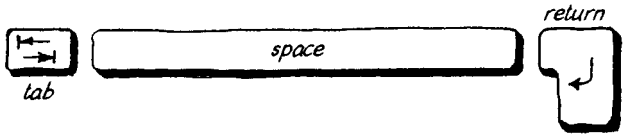
  Be sensible with spacing; don't split adjacent *tokens* and don't join adjacent tokens if both are words or letters;




  Apart from that you may cram tokens together or spread them out over several lines if you like:




  To separate tokens, use any combination of whitespace keys:




- Store. Store what you type in a file, giving the file a name such as WOTCOST.C (The .C is added automatically in some environments; it signifies a file containing a C program in character form, the .C being an *extension* of the name proper.)

- Compile. Compile the program which involves translating your C program into a code the computer can understand and obey directly.

  This step may be initiated by selecting Complie from a screen menu, or typing a command such as cc wotcost.c (Unix) and pressing the Return key. It all depends on your environment.

  The compiler reports any errors encountered. A good IDE displays the statements in which the errors were discovered, and locates the cursor at the point where the correction should be made.

- Edit. Edit the .C file and recompile as often as necessary to correct the errors discovered by the compiler. The program may still have *logical* errors but at least it should compile.

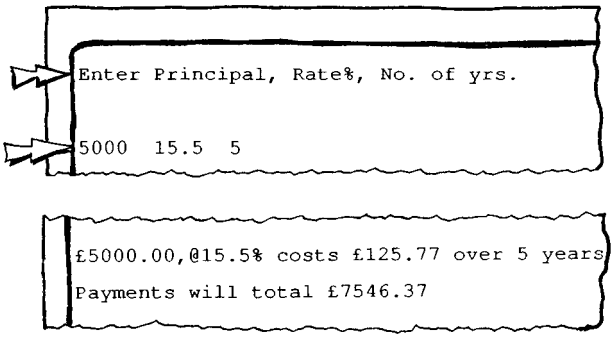
  You have now created a new file containing *object code*. The file of object code has a name related to the name of the original file. In a DOS environment it might have the name WOTCOST.OBJ (compiled from WOTCOST.C). In a Unix environment, if you compiled **wotcost.c** your object code would be stored in **a.out**.

- Link. In many environments a simple C program may be compiled and linked all in one go (type a.out, press Return, and away we go!). In other environments you must link the program to functions in the standard libraries (pow, printf, scanf are functions written in C too). The resulting file might have the name WOTCOST.EXE (linked from WOTCOST.OBJ).

- Run. Run the executable program by selecting Run from a menu or enterng the appropriate command from the keyboard.

- Execution. The screen now displays:

  Enter three items separated space, tab or new line. End by pressing Return.




  The program computes and sends results to the standard output file (named std.out). This 'file' is typically the screen.

# DISSECTION *OF A C PROGRAM, PIECE BY PIECE*

Here is the compound interest program again ≈ with a title added for identification.

```
/* WOTCOST; Computes the cost of a loan */        <- comment
#include <stdio.h>                                <- directives - no semicolon
#include <math.h>
int main (void)                                   <- header
{                                                 <- block
    float P, Rpct, R, M;
    int N;
    printf ("\nEnter: Principal, Rate%, No. of yrs.\n");
    scanf ("%f %f %i", &P, &Rpct, &N );
    R = Rpct / 100;
    M = P*R*pow(1+R, N) / (12*(pow(1+R, N) - 1 ) );
    printf ("\n£%1.2f, @%11.2f%% costs £%1.2f over %i years",P,Rpct,M,N);
    printf ("\nPayments will total £%1.2f", 12*M*N );
    return 0;
}
```

`/* WOTCOST;  loan */` Any text between /* and */ is treated as commentary. Such commentary is allowed wherever whitespace is allowed, and is similarly ignored by the processor.

`#include <stdio.h>`
`#include <math.h>` The # (which must be the first non-blank character on the line) introduces an instruction to the *preprocessor* which deals with organizational matters such as including standard files. The standard libraries of C contain many useful functions; to make such a function available to your program, tell the preprocessor the name of its *header file*. In this case the header files are stdio.h (standard input and output) and math.h (mathematical). The header files tell the linker where to find the functions invoked in your program.

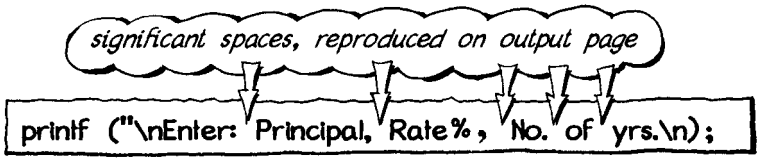
`int main (void)` A C program comprises a set of functions. Precisely one must be named **main** so the processor knows where to begin. The **int** and **void** are explained later; just accept them for now. The declarations and statements of the functions follow immediately; they are enclosed in braces, constituting a *block*. There is no semicolon between header and block.

`float P,Rpct,R,M;`
`int N;` The 'little boxes' depicted earlier are called *variables*. Variables that hold decimal numbers like 15.5 are of a different *type* from variables that hold only whole numbers. These two statements declare that the variables named P, Rpct, R, M are of type **float** (short for floating point number) and the variable named N is of type **int** (short for integer). Other types are introduced later.

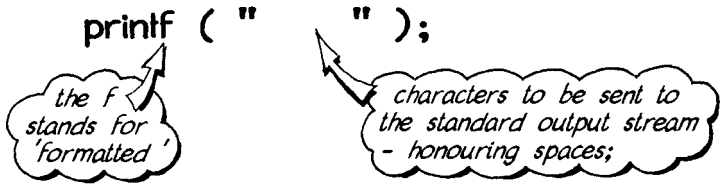
*Declarations*, such as those above, must all precede the first *statement*.

Each declaration and each statement is terminated by a semicolon. A *directive* is neither a declaration nor a statement; it has no semicolon after it.

You have freedom of layout. Statements may be typed several to a line, one per line, one to several lines. To the C compiler a space, new line, Tab, comment, or any number or combination of such things between statements ≈ or between the tokens that make up a statement ≈ are simply *whitespace*. One whitespace is as good as another, but not when between quotation marks as we see here.




This is an invocation of printf(), a much-used library function for printing. In some environments the processor includes standard input and output automatically ≈ without your having to write #include <stdio.h>

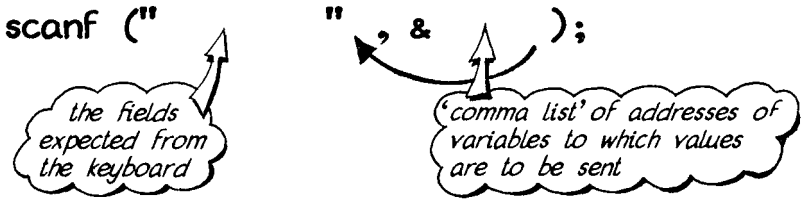



When printing, the processor watches for back-slash. On meeting a back-slash the processor looks at the next character for guidance: n says start a new line. \n is called an *escape sequence*. There is also \t for Tab, \f for form feed, \a for ring the bell (or beep) and others.

(It's no good pressing the Return key instead of typing \n. Pressing Return would start a new line on the screen, messing up the syntax and layout of your program. You don't *want* a new line *in the program*, you want your program to generate one when it obeys printf().)

```
scanf ("%f %f %i", & P, & Rpct, & N );
```

This is an invocation of the scanf() function for input. For brevity, most examples in this book use scanf(). Safer methods of input are discussed later.

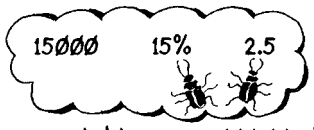



There is more about scanf() overleaf.

To obey the scanf() instruction the processor waits until you have typed something at the keyboard and pressed the return key (( 'something' means three values in this example )). The processor then tries to copy values, separated by whitespace, from the keyboard buffer. If you type fewer than three values the processor stays with the instruction until you have pressed Return after entering the third. If you type more, the processor reads and ignores the excess.

The processor now tries to interpret the first item as a floating point number (( %f )). If the attempt succeeds, the processor sends the value to the address of variable P (( & P )) ≈ in other words stores the value in P. The second value from the keyboard is similarly stored in Rpct. Then the processor tries to interpret the third item from the keyboard as a whole number (( %i )) and stores this in variable N.

What happens if you type something wrong? Like:




where the 15ØØØ is acceptable as 15ØØØ.ØØ, but the second item involves an illegal sign, the third is not a whole number.

The answer is that things go horribly wrong. In a practical program you would *not* use scanf().

Why the '&' in &P, &Rpct, &N ? Just accept it for now. The art of C, as you will discover, lies in the effective use of:

& 'the address of...' or 'pointer to...'

* 'the value pointed to by...' or 'pointee of...'

```
R=Rpct/1ØØ;
M=P*R*pow(1+R,N)/(12*pow(1+R,N)-1));
```

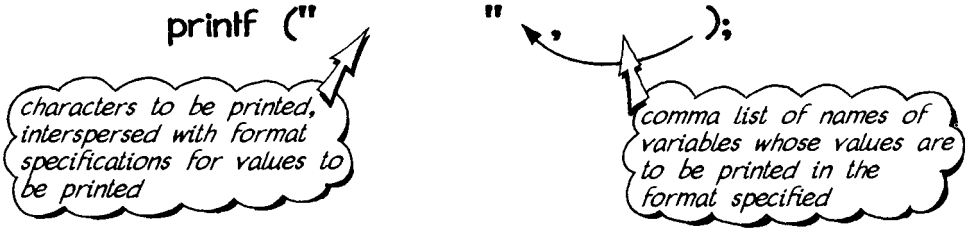
These statements specify the necessary arithmetic: Rpct/1ØØ means divide the value found in Rpct by 1ØØ. The /,*,+,- mean respectively: divide by, multiply by, add to, subtract from. They are called *operators*, of which there are many others in C.

pow(1+R,N) is a *function* which *returns* the value of (1+R) raised to the power N. If you prefer to use logs you could write exp(log(1+R)*N) instead. The math library (( #include <math.h> )) would be needed in either case; exp(), log(), pow() are all math.h functions.

The terms 1+R and N are *arguments* (( *actual* arguments )) for the function pow(), one for each of that function's *parameters* (( *dummy* parameters )). In some books on computing the terms *argument* and *parameter* are used interchangeably.

```
printf ("\n%8.2f,@%.2f%% costs £%.2f over %i years", P,Rpct,M,N);
```

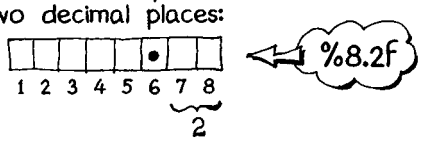
This is like the earlier printf( ) invocation; a string between quotes in which \n signifies *Start a new line on the output screen.*




But this time the string contains four *format* specifications: %8.2f, %.2f, %.2f, %i for which the values stored in variables P, Rpct, M, N are to be substituted in order. You can see this better by rearranging over two lines using whitespace:

```
printf ("\n£%8.2f @%.2f%% costs £%.2f over %i years"
        ,P      ,Rpct        ,M       ,N      );
```

Take %8.2f as an example. The % denotes a format specification. f denotes a field suitable for a value of type **float** ≈ in other words a number with a fractional part after a decimal point. The 8 specifies eight character positions for the complete number. The .2 specifies a decimal point followed by two decimal places:




A single percentage sign introduces a format specification as illustrated. So how do you tell the processor to *print* an ordinary percentage sign? The answer is to write %% as demonstrated in the printf( ) statement above.

The second ( and subsequent ) format specification is %.2f. How can the field be zero characters wide if it has a decimal point and two places after? This is a dodge; whenever a number is too wide, the processor widens the field rightwards until the number just fits.

```
printf ("\nPayments will total £%.2f", N * 12 * M );
```

This is another printf( ) invocation with an 'elastic' field. This time the value to be printed is given by an *expression*, n*12*M, rather than the name of a variable. The processor evaluates the expression, converts the resulting value (if necessary) to a value of type **float**, and prints that value in the specified field.


```
return 0;
```

Just accept it for now: the opening **int main ( void )** and closing **return 0** are described later.

# EXERCISES

1 Implement the loans program. This is an exercise in using the tools of your particular C environment. It can take a surprisingly long time to master a new editor and get to grips with the commands of an unfamiliar interface. If all else fails, try reading the manual.

# 2

# CONCEPTS

One of the few troubles with C is that you can't formally define concept A without assuming something about concept B, and you can't define B without assuming something about A. Books on C have a bit in common with the novel Catch 22.

The aim of this chapter is to introduce, informally, enough simple concepts and vocabulary to make subsequent chapters comprehensible.

This chapter introduces decisions, loops, characters, arrays, functions, scope of variables, and recursion. Complete programs are included to illustrate the aspects introduced.

# DECISIONS

## LOGICAL VALUES IN C
## THERE ARE NO BOOLEAN VARIABLES

If, in your program, Profit is greater than Loss ( Profit and Loss being names of variables holding values ) you may want the program to do one thing, otherwise another. The expression Profit > Loss is *true* if the value in Profit is greater than that in Loss; *true* is represented by 1. Conversely, if the value in Profit is *not* greater than that in Loss the expression is *false* and takes the value ø.

Thus 9.5 > ø.ø takes the value 1 ( *true* ); 9.5 < ø.ø takes the value ø ( *false* ). A few other logical operators are shown here: ⇒ Operators are defined in Chapter 3 and briefly summarized on page 196.

| | |
|---|---|
| < | less than |
| >= | greater than or equal to |
| == | equal to |
| != | not equal to |
| && | logical *and* |

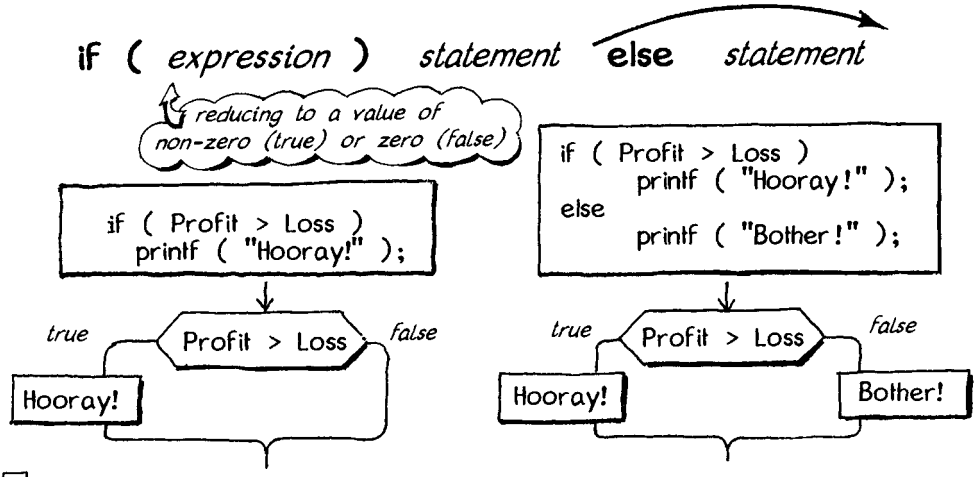
There are no Boolean variables in C ⇒ you have to make do with integers; a value of zero represents *false*; *any non-zero value represents true.*

Statements concerned with the flow of control ( if, while, do for ) are based on values of logical expressions: non-zero for *true*, zero for *false*.

# IF ELSE

## A SELECTION STATEMENT

The if statement may be used to select a course of action according to the logical value ( true or false ) of a parenthesized expression:




The *statement* is typically a compound statement or *block*. Anywhere a *statement* is allowed a *block* is also allowed. A block comprises an indefinitely long sequence, in braces, of declarations ( optional ) followed by statements. Some of the statements may be if statements ⇒ a *nested* pattern.

Be careful when nesting 'if' statements. Try to employ the pattern resulting from 'else if' rather than 'if if' which leaves 'elses' dangling in the brain. A sequence of 'if if' makes it difficult to match the associated 'elses' that pile up at the end.

In the illustration below, the operator ! means *not*. Thus if variable Lame holds the value ø (( false )) then the expression !Lame takes the value 1 (( true )). Conversely, if Lame holds a non-zero value (( true )) then the expression !Lame takes the value ø ((false)).

```
if ( Lame )
  Walk ( ø );
else
  if ( SoSo )
      Trot ( ø );
      else
          if ( Quiet );
              Canter ( ø );
              else
                  Gallop ( ø );
```

OK

```
if ( !Lame )
  if ( !SoSo )
      if ( !Quiet )
          Gallop ( ø );
      else
          Canter ( ø );
  else
      Trot ( ø );
else
  Walk ( ø );
```

confusing

Each 'else' refers to the closest preceding 'if' that does not already have an 'else', paying due respect to parentheses. Careful indentation shows which 'else' belongs to which 'if', but remember that *the processor* pays no attention to indentation. Careless indentation can present a misleading picture.

Here is a program that uses a *block* in the 'if' statement as discussed opposite. The program does the same job as the introductory example but first checks that all items of data are positive.

Complicated logic based on 'if else' can be clumsy; we meet more elegant methods of control later.

```
/* WOTCOST with data check */
#include <stdio.h>
#include <math.h>
int main (void)
{
  float P, Rpct, R, M;
  int N;
  printf ("\nEnter: Principal, Rate%, No. of yrs.\n");
  scanf ("%f %f %i", &P, &Rpct, &N );
  if ( ( P>ø ) && ( Rpct > ø ) && ( N > ø ) )
  {
      R = Rpct / 1ØØ;
      M = P*R*pow(1+R, N) / (12*(pow(1+R, N) - 1 ) );
      printf ("\n£%.2f, @%.2f%% costs £%.2f over %i years",P,Rpct,M,N);
      printf ("\nPayments will total %1.2f", 12*M*N );
  }
  else
      printf ( "Non-positive Data" );
  return ø;
}
```

*the initial 'int' means the program returns an integer to its environment as a signal of success or failure;* return ø; *(below) indicates success*

*&& says 'and'*

*block*