

'C' Programming Reference Notes

Version 0.1 created using OpenOffice 1.1.3 on Mandrake Linux 10.1 Richard Kay, March 2005.

Contents

1. About these notes
2. Glossary of terms
3. Organisation of source code
 - 3.1 Files
 - 3.2 Functions
 - 3.3 Blocks, statements and whitespace
 - 3.4 Preprocessing
 - 3.5 Variable and function names
 - 3.6 Keywords
 - 3.7 Comments
4. Primitive types, variable declarations and literals
 - 4.1 Integer types
 - 4.2 Floating point types
 - 4.3 Characters and strings
5. Expressions
 - 5.1 Arithmetic binary
 - 5.2 Arithmetic unary
 - 5.3 Assignments
 - 5.4 Comparisons
 - 5.5 Boolean
6. Control of execution flow
 - 6.1 if else branches
 - 6.2 top exit while loops
 - 6.3 bottom exit do while loops
 - 6.4 for loops
 - 6.5 using break for middle exit loops
 - 6.6 switch case

6.7 the ternary ?: branch expression

7. Arrays

7.1 Array declarations and literals

7.2 Array access and indexes

7.3 Array addresses as function parameters

7.4 Strings

7.5 Dynamic arrays

8. Functions

8.1 Example

8.2 Function prototype

8.3 Function call

8.4 Function definition

8.5 Function parameters

8.6 Return types

8.7 Localisation, duration and globalisation of data

9. Pointers

9.1 References and values

9.2 Address of operator

9.3 Pointer variable declarations

9.4 Indirection operator

10. Structures

10.1 struct declaration

10.2 Type definition

10.3 Structure variable declaration

10.4 Access to member variables

10.5 Use of structure pointers

10.6 Structures and arrays

11. Library functions

11.1 Standard input and output

11.2 String handling

11.3 Mathematical

11.4 miscellaneous

1. About these notes

These aim succinctly to describe the syntax of features of the 'C' Programming language likely to be required by students studying Data Structures and Algorithms, and related modules or content. They are also intended as an examination aid, in order to fulfill a similar role in written examinations to the formulae books customarily used in science and maths exams. These notes are intended to help consolidate, but not to substitute for, knowledge gained from lectures, background reading, experimentation, tutorial and course work.

Acknowledgements are given to Kernighan and Ritchie (The 'C' Programming Language 2e) some of whose information has been summarised in these notes, and to Paul Hobbs for assistance with proofreading.

2. Glossary of terms

Array: a collection of variables of the same type using a contiguous block of memory.

Assignment: a statement or expression which modifies the value of one or more variables e.g. $f = 2.5$; or $i++$;

Block: a set of statements which start and end with opening and closing curly braces `{}`. Note that not all 'C' source words enclosed in `{}` braces are blocks. A block is a unit of program code syntactically analogous to a paragraph in written English.

Call: The action of a program statement in executing or using a separate function. The calling function will wait for the called function to return.

Character: a single letter, or a variable capable of storing such, e.g. 'A'.

Comment: information added to source code which does not affect how the

program runs, but documents and describes functions, data and algorithms, authorship and change information etc.

Compiler: a program which converts the source code of a program into its executable machine code. In the 'C' language this involves preprocessing, compiling and linking object modules with library code.

Constant: a value used within a program which can not change e.g.

```
#define PI 3.14159
```

Declaration: a statement or part of a statement which associates a variable with a data type, e.g. float f; This statement associates the float data type with the declared variable named f.

Decrementation: subtracting a value from a variable, usually 1.

Element: a single data item contained within an array of similar elements.

Escape: a character encoding allowing interpolation of unprintable characters or parameters within a string. E.G. \n for a newline, or %d to substitute the decimal value of an integer parameter within an output string.

Expression: a statement or part of a statement which generates or returns a value, e.g. 2 + 2

Filehandle: the address of a file used for input/output access.

Float: a type of data (floating point) used for the storage of numbers which may contain fractions.

Function: a component of a program which may be defined by the programmer or accessed from a function library. See call and return.

Global: a property of data variables, type definitions and declarations when these are accessible to the program as a whole and are not localised inside any block.

Incrementation: adding a value to a variable, usually 1.

Integer: a type of data used for the storage of whole numbers.

Literal: data expressed within source code as an actual value, e.g. 42 or "hello".

Local: a property of data variables etc. when these are modularised within a block, see global.

Member: a named field variable within a record or structure.

Object: an identifiable entity within the real or a virtual world about which information and/or behaviour might be associated.

Operating system: software which runs on a physical computer capable of loading and executing other programs, interacting with physical hardware and supporting user interaction.

Parameter: a value passed or copied from a calling function to a called function.

Pointer: an address or a variable capable of storing an address. See reference.

Program: one or more source code or executable files capable of being executed as a unit by an operating system.

Prototype: a statement declaring the interface to or signature of a function, including its name, return type and the number, order, and types of its parameters.

Record: a collection of data about a particular object. See also structure.

Reference: another name or means of accessing an object, such that the same object can be accessed from different name spaces. See pointer.

Return: the action of a function or program in returning control of execution to a waiting calling function or program.

SOB: This is used as shorthand to describe programming syntax in places where either a statement or a block is required.

Statement: a unit of program code syntactically analogous to a sentence in English. Statements end in semicolons (;). They are grouped into 2 types, declarations and others. Declarations which allocate types to data variables must be in the upper part of any statement block, or in global program sections outside of any block.

Stream: an open file.

String: An array of characters e.g. "hello". Strings are conventionally terminated with a null character: '\0' and therefore require 1 more unit of storage than the number of characters contained within them.

Structure: a programmer-defined type collecting a set of fields or members into a record.

Type: a classification or class applied to data, e.g. determining whether it can be used for storage of floats, integers, strings etc or a programmer designed structured data type.

Typedef or Type Definition: The name given to a programmer defined type.

Variable: a data value used within a program which may change, or the storage or memory used for this value.

Whitespace: non-printable newlines, space or tab characters.

3. Organisation of source code

3.1 Files

A 'C' program is compiled from 1 or more application programmer-written source code text files, together with included header files and linked library functions. By convention, preprocessing statements, prototypes, structures, typedefs and global variables are placed at the top of the file, with function definitions placed below. Use the extern keyword when declaring global program variables for which the memory allocation is performed using another source file.

3.2 Functions

By convention function prototypes are declared above function calls, and function calls are coded above function definitions.

3.3 Blocks, statements and whitespace

A block is a set of statements enclosed between {} curly braces. Statements contained within a block are conventionally written each on a separate line and indented by a single tab compared to prior and subsequent code. e.g.

```
#include <stdio.h>
/* program to demonstrate code layout.
   Richard Kay, Mar 2005
*/

void foo(void); /* prototype of foo() */

int main(void){ /* main function. Program starts execution here */
    foo(); /* call to foo() function */
    printf("bar\n");
    return 0;
}
```

```
void foo(void){ /* function which prints "foo" */  
    printf("foo\n");  
}
```

Placement of empty lines between significant sections of code also improves readability.

3.4 Preprocessing

Preprocessing involves manipulation of the source code to be compiled prior to compilation. Special statements control this. These statements are written in a separate preprocessor language, and have a different format from other 'C' language statements. Preprocessor statements start with a # hash character.

#include statements are used to include other source files. e.g.

```
#include <stdio.h> /* includes the standard I/O 'C' library */  
#include "myheader.h" /* includes a programmer written header file */
```

#define symbol value statements are used to define program constants, so that during compilation proper, the symbol is replaced with the value. e.g.

```
#define MAX_RECORDS 100
```

constant names conventionally use UPPERCASE_LETTERS.

3.5 Variable and function names

Names of variables and functions must start with a letter, and may contain only letters, numbers and underscores (_). Reserved words (keywords) must not be used. Use descriptive names which match and identify the purpose of the variable, but avoid typing strain and errors by using overlong names.

Correct: alpha, catch_22, file_name

Incorrect: 22c (must not start with other than a letter), while (reserved

keyword), file name (contains prohibited space character).

3.6 Keywords

These can't be used for variable, structure, typedef, constant or function names:

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

Modern 'C' compilers are also used to compile the 'C++' language; therefore 'C++' specific keywords must also be avoided when naming 'C' variables:

asm	bool	catch	class	delete
export	explicit	false	friend	inline
mutable	new	namespace	operator	private
protected	public	template	this	true
try	virtual			

3.7 Comments

Use plenty of appropriate comments to document your program. You should state the purpose of each variable when it is declared if its name is not obvious. Comment the authorship, latest change date and purpose of the program as a whole. Comments should describe each function, and any other significant block of code, particularly where the purpose or operation of the code isn't obvious from the source itself. In 'C' comments are preceded by `/*` and ended with `*/`, e.g:

```
/* this is a comment */
```

C++ style comments (i.e. the rest of a line after a pair of forward slashes `//`)

are not standard 'C' , but these are likely to be compiled correctly, for the same reason that 'C++' keywords can't be used as object names.

4. Primitive types, variable declarations and literals

4.1 Integer types and arithmetic

Integer types involve whole numbers which can't contain a fraction. The result of an integer arithmetic expression is an integer, see examples below.

Variables declared as int e.g: int i; have a memory allocation based on the CPU architecture and operating system default integer handling. E.G. using Borland 'C' version 3.1 int variables use 2 bytes of storage and on more modern systems, 4 or 8 bytes. The compiler can be directed to allocate more or less storage to integers using types long and short.

Examples

```
int i=8; /* declaration of integer i, initial value 8 */
long bignum=1000000000L; /* appending l or L after literal for long int*/
short smallnum; /* less memory for small integers */
10/3 /* result of integer division expression is 3 */
10%3 /* use of modulus or remaindering operator, result is 1 */
```

4.2 Floating point types and arithmetic

Floating point variables can contain fractions. Arithmetic expressions including a floating point term generate a floating point result. Normal precision floating point variable storage size uses type float, while double precision floating point variables use type double. Examples

```
float radius=2.5; /* declaration of floating point radius, initial value 2.5 */
double circum, area, volume; /* double precision floating point variables */
10.0/3.0 /* result of expression is 3.333333 */
```

4.3 Characters and strings

A char variable stores a single character. A char literal is a single character or character code enclosed in single quotes e.g. 'A' or '\n'. A string is an array of chars e.g. "hello". String expressions including the name of a char array and string literals such as "hello" return the address of the string. Strings are conventionally terminated with a null character: '\0' and therefore require 1 more unit of storage than the number of characters. Examples :

```
char letter, line[1024]; /* storage for single char letter and string called line */
char c='A', address[]="Rose Cottage\nBull Lane,\nAmbridge\nBorsetshire\n";
/* storage allocated for letter and string containing a postal address with initial
values */
```

Declaration with initial values as above is permitted, but assignment to a string or comparison of 2 strings are not. Use strcpy() or strcmp() from string.h to copy or compare strings. Example:

```
char neighbour[30],address[30];
printf("enter address");
gets(address); /* input address */
if( strcmp(address,"10 Downing St") == 0) {
    printf("hello prime minister");
    strcpy(neighbour,"Gordon Brown at no. 11");
}
```

5. Expressions

5.1 Arithmetic binary

Operators

+ addition, - subtraction, * multiplication, / division, % modulus.

Examples: 2+2 (returns 4), 10/3 (returns 3), 10%3 (returns remainder of 1 on dividing 10 by 3), 10.0/3 (returns 3.333333), 10.0%3 (invalid - modulus only valid for integers), 10.0/0.0 (invalid divide by zero).

5.2 Arithmetic unary

incrementation of integer variable i:

`i++ ; /* adds 1 to i after i is used if part of another expression */`
`++i ; /* adds 1 to i before i is used if part of another expression */`

decrementation of integer variable i:

`--i ; /* subtracts 1 from i after i is used if part of another expression */`
`i-- ; /* subtracts 1 from i before i is used if part of another expression */`

- (minus) may also be used as a unary operator, which reverses the sign of the expression to which it applies.

5.3 Assignments

`variable = expression; /* uses result of expression to modify value of variable*/`

In place assignments

`variable op= value` is equivalent to `variable = variable op value`

where op is typically one of + - * / % .

Arithmetic operators including + - * / and % are combined with = to perform binary arithmetic using the value of variable as the left hand side of the binary arithmetic, with the result being assigned to variable.

Examples

```
int a=2, b; /* a is given initial value, but b is not. */
b= 3 * 3 ; /* b is assigned the value 3 times 3 which is 9 */
a+=3; /* adds 3 to previous value of a, which is assigned the value 5 */
b/=2; /* divides b by 2 (integer division), after this b becomes 9/2 which is 4 */
```

5.4 Comparisons

Comparisons return integers 1 if true and 0 if false and are used to compare numbers or characters. They are not usable for comparing more complex objects, e.g. strings, for which library functions should be used. Testing for equality of floating point variables is fraught by rounding errors.

The following comparison operators are used:

a == b	true if a and b are equal.
a != b	true if a and b are not equal
a < b	true if a is less than b
a <= b	true if a is less than or equal to b
a > b	true if a is greater than b
a >= b	true if a is greater than or equal to b

5.5 Boolean

Expressions returning true or false (non-zero or 0) results can be combined using boolean operators.

a&&b True if both a AND b are true. Expr. b is evaluated only if a is true.
a || b True if either a OR b or both are true. Expr. b not evaluated if a is true.
!a Read as NOT a. True if a is false, and false if a is true.

5.6 Order of evaluation

The table (Kernigan and Ritchie, "The 'C' Programming Language" 2e) lists rows of operators with equal precedence on the same row. Operators in higher rows are of higher precedence i.e. evaluated before those in lower rows. Within a row, associativity is either left to right (L to R) or right to left (R to L).

Operator Types	Operators	associativity
braces, array index, struct member connectors	() [] -> .	L to R
not, incr/decr, unary + -, indirection, address, cast, sizing	! ~ ++ -- + - * & (type) sizeof	R to L
arithmetic binary mult/div/mod	* / %	L to R
arithmetic binary add/minus	+ -	L to R
left and right bit shift	<< >>	L to R
comparison	< <= > >=	L to R
equality	== !=	L to R
bitwise and	&	L to R
bitwise exclusive or	^	L to R
bitwise inclusive or		L to R
boolean and	&&	L to R
boolean or		L to R
ternary branch expression	?:	R to L
assignment	= += -= /= %= &= ^= = <<= >>=	R to L
comma	,	L to R

Use () braces to force the order of evaluation. E.G to divide the sum of 3 and 6 by 3, you would write: (3 + 6)/3, which returns 3. Without braces, 3+6/3 returns 5, because / takes precedence over + .

5.7 Casts

Casts are used to convert data to a compatible type. The casting operator is a primitive or programmer created type enclosed in () braces and it applies to the expression to the right of the cast. If the type name is followed by an asterisk (*) the data is used for an address (pointer or reference) of that type.

Examples:

```
(float)10/3    /* returns 3.333333 . 10/3 would have returned 3. */
recptr=(RECORD*)malloc(sizeof(RECORD)); /* allocate memory for record
and
convert void pointer returned by malloc to RECORD pointer. */
```

6. Control of execution flow

6.1 if else branches

General form:

```
if(boolean expression)
    SOB1
else
    SOB2
```

SOB1 means statement or block 1, SOB2 means statement or block 2.

The else and subsequent SOB2 (the else clause) is optional. If the boolean expression is true, SOB1 is executed and SOB2 is not. If the boolean expression is false SOB1 is not executed and SOB2 is executed.

Example 1 :

```
if(i > 0)
    printf("i is positive");
else {
```

```
    printf("i is either zero ");
    printf("or i is negative");
}
```

In this example SOB1 is a single statement, SOB2 is a block. Note the positioning of the semicolons and {} braces to delimit the start and end of SOB1 and SOB2.

Example 2:

```
if(messages > 0)
    printf("You have mail\n");
```

In this example "You have mail" is printed if there are 1 or more messages. As there is no else clause, nothing happens if there are no messages.

Nested if else branches

Nested if else multi way branches can be constructed to execute one of a set of SOBs. Example 3:

```
int temp;
printf("input temperature between 0 and 100\n");
scanf("%d",&temp);
if(temp>30)
    printf("hot\n");
else if(temp>20)
    printf("warm\n");
else if(temp>10)
    printf("cool\n");
else
    printf("cold\n");
```

In this example exactly 1 of the printf statements will be executed.

6.2 top exit while loops

General form:

```
while(boolean expression)
    SOB
```

A loop is used to execute a statement or block (SOB) a number of times. A top exit while loop can execute its SOB 0 or more times, and continues to run while the boolean expression tested remains true.

Example 1

```
i=1;
while(i<4){
    printf("%d is less than 4\n",i);
    i++;
}
```

Example 2

```
i=1;
while(i<4)
    printf("%d is less than 4\n",i++);
```

Both examples print out:

```
1 is less than 4
2 is less than 4
3 is less than 4
```

Note that *i* is incremented after its value is used by `printf` in both examples. In the second example, as the SOB is only a single statement, braces are not required.

6.3 bottom exit do while loops

General form:

```
do
    SOB
while(boolean expression);
```

A bottom exit while loop can execute its statement or block (SOB) 1 or more times, while the boolean expression tested remains true. This form is often used when the body needs to be executed at least once, e.g. to create the conditions where the loop test is possible. Common usage is to re-prompt for invalid input. Example:

```
int i;
do {
    printf("enter a number between 1 and 10\n");
    scanf("%d",&i);
} while(i<1 || i>10);
```

6.4 for loops

General form 1:

```
for( initialisation ; continuation test ; incrementation)
    statement;
```

is equivalent to:

```
initialisation;
while(continuation test){
    statement;
    incrementation;
}
```

General form 2:

```
for( initialisation ; continuation test ; incrementation){  
    statement 1;  
    ...  
    statement n;  
}
```

is equivalent to:

```
initialisation;  
while(continuation test){  
    statement 1;  
    ...  
    statement n;  
    incrementation;  
}
```

For loops are preferred where initialisation and incrementation statements are considered an integral part of the loop control.

The following example prints a 12 times table.

```
int i;  
for(i=1;i<=12;i++)  
    printf("12 x %d = %d\n",i,i*12);
```

output:

1 x 12 = 12

2 x 12 = 24

...

12 x 12 = 144

6.5 using break for middle exit loops

General form:

```
while(1){ /* 1 is always true */
    1 or more statements;
    if(exit condition)
        break;
    1 or more statements;
}
```

The break statement exits the innermost enclosing loop. This design pattern allows for clearer coding in cases where the condition requiring exit from the loop is detected in the middle of it. Example:

```
while(1){
    printf("enter name or * to quit");
    gets(name);
    if(name[0] == '*')
        break; /* exit loop now */
    /* do things with name, and maybe related fields */
}
```

To avoid a middle exit as shown above, the loop body would require a flag variable in the if block for use in a top or bottom continuation test, and putting the part of the loop body skipped when a loop exit is required into an else branch, obscuring the flow of control.

6.6 switch case

This construct is used as a multi way branch as an alternative to a set of nested if - else branches. The variable used to control the switch must be an integer or character type. Flow of control from the switch will either be to a case label matching the switch variable or to the default case, if no label matches. Break statements are required to prevent execution continuing sequentially.

Example

```
int day, index; /* day of month and suffix string index, 0 - 3 */
char *suffix[]={"th","st","nd","rd"}; /* for 1st, 2nd, 3rd 4th etc of month. */
printf("enter day of month\n");
scanf("%d",&day);
switch (day){
    case 1: case 21: case 31: /* numbers having suffix st */
        index=1;
        break;
    case 2: case 22: /* numbers having suffix nd */
        index=2;
        break;
    case 3: case 23: /* numbers having suffix rd */
        index=3;
        break;
    default: /* all other numbers have suffix th */
        index=0;
}
printf("it is the %d%s of the month\n",day,suffix[index]);
```

output

enter day of month

22

it is the 22nd of the month

6.7 the ternary ?: branch expression

This may be used as an alternative to an if - else branch in some cases. It is useful when coding minor expression choices when these should have limited visual impact on the overall layout of the algorithm.

general form:

```
expr 1?expr 2:expr 3
```

If expr 1 is true, the entire expression returns expr 2, otherwise it returns expr 3.

Example:

```
printf("how many wolves ?\n");
scanf("%d",&nw);
printf("there %s %d %s", nw==1?"is":"are", nw, nw==1?"wolf":"wolves" );
```

output

how many wolves ?

1

there is 1 wolf

how many wolves ?

3

there are 3 wolves

7. Arrays

7.1 Array declarations and literals

An array is a contiguous block of memory for the storage of a set of variables all of the same type. Compiled array declarations are similar to those for single variables, but requiring square brackets after the array name, and information enabling the compiler to calculate the number of elements, e.g:

```
#define MAX_POSITIONS 10
```

```
int i,positions[MAX_POSITIONS];
```

The memory allocation for the array is equal to the size of the element type multiplied by the number of elements.

Test and fixed data may be coded in the form of array literals. Here the compiler can size the array by counting the literal values supplied e.g:

```
float sortdata[]={65.2, 98.3, 32.4, 88.8, 12.7, 56.1, 9.2, 76.3};
```

7.2 Array access and indexes

The array name, as used in data processing statements without [] brackets, is the address of the start of the array. Individual array elements are referenced using pointer arithmetic, or can be accessed directly using square brackets with the relevant integer index. For the N elements in an array, the valid integer indexes are in the range 0 .. N-1 , e.g. if N is 10, valid indexes are in the range 0-9 inclusive. Incorrect uses of invalid array indexes are likely to result in unexpected program runtime behaviour. Example

```
#define MAX 10
int ia[MAX], i;
printf("enter %d numbers\n",MAX);
for(i=0;i<MAX;i++)
    scanf("%d",ia+i); /* array element addresses using pointer arithmetic */

printf("the numbers entered are:\n");
for(i=0;i<MAX;i++)
    printf("%d\n",ia[i]); /* index in square brackets for direct element access
*/
```

7.3 Array addresses as function parameters

The whole array is passed using call by reference rather than by value. Hence the address of the array is passed by the calling function to a pointer variable within the called function. Example:

```
#define MAX 10

void printarray(int array[],int size); /* prototype. array[] same as *array */

int main(void){
    int ia[MAX], i;
    printf("enter %d numbers\n",MAX);
    for(i=0;i<MAX;i++)
        scanf("%d",ia+i);
```

```
    printarray(ia,MAX); /* function call passing array address and size */
    return 0;
}

void printarray(int array[],int size){ /* definition of printarray */
    printf("the numbers entered are:\n");
    for(i=0;i<size;i++)
        printf("%d\n",array[i]);
}
```

7.4 Strings

Strings are arrays of char variables. The 'C' language supports a simple means of specifying string literals by double quoting, e.g. "hello". 'C' strings require 1 extra storage unit, in addition to the number of characters within the useful part of the string array, for marking the end of the string. The special 0 or NULL character: '\0' is used for this purpose. European languages conventionally use 1 byte per char, and Unicode strings are likely to require 2, but these sizes depend upon the encoding. The 'C' input/output and string handling library functions all use the value '\0' to indicate the end of a string.

To process arbitrary binary data in strings which might require \0 as a valid datum, standard 'C' string functions can not be used, and separate function parameters for string sizes are needed.

7.5 Dynamic arrays

If the size of array required is not known at compilation time, memory allocations can be dynamically taken from and returned to the heap using the malloc function, with the memory allocation computed at runtime. The parameter to malloc is the number of bytes required for the array. Malloc returns NULL if the request is unsuccessful. Otherwise it returns a void pointer to a memory allocation of the size requested. Conventionally this address is cast to the pointer type of array required. Example:

```
RECORD *array; /* pointer for the array address but not yet the storage */
int nrecs;
```

```
printf("how many records ?\n");
scanf("%d",&nrecs);
if((array=(RECORD*)malloc(nrecs*sizeof(RECORD)) ) == NULL){
    printf("allocation unsuccessful.");
    exit(1); /* quit program now */
}
/* if still here, storage is now allocated so can use array of type RECORD with
valid elements between array[0] and array[nrecs -1] */

program statements using array ...
free(array); /* no longer need space so free memory allocation in heap */
```

8. Functions

8.1 Example

```
#include <stdio.h>

void foo(void); /* prototype of foo function */

int main(void){ /* start of definition of main function */
    printf("before foo\n");
    foo(); /* call to foo function from within main*/
    printf("after foo\n");
    return 0; /* value returned by main function */
} /* end of definition of main function */

void foo(void){ /* start of definition of foo function */
    printf("inside foo\n");
} /* end of definition of foo function */
```

output

```
before foo
inside foo
after foo
```

Explanation

Execution starts at the beginning of the function called main, which conventional 'C' programs must have. Statements in main execute sequentially until foo is called, when execution jumps to the foo function. On completing the foo function and returning from it, execution resumes within the main function after the call.

8.2 Function prototype

This is a statement declaring the interface to or signature of a function, including its name, return type and the number, order, and types of its parameters. Function prototypes are declared above the places in the code where functions are called. The compiler will not validate a call which is incompatible with the prototype.

8.3 Function call

This is the action of a program statement within 1 function (the calling function) in executing or using another function (the called function). The calling function will wait for the called function to return.

8.4 Function definition

This is the coding which causes the function to perform its task. Design of a function typically involves first defining the interface and documenting what the function will do (see prototype). At this stage the function can be thought of as a black box with a known interface, such that program components which use this component can be designed. At a later stage, the function will then need to be coded, which involves designing the function internals and deciding how the function will do what it is intended to do.

8.5 Function parameters

Parameters are most commonly passed by reference. The exception is that they may be passed by value only for single variables (or single array

elements) to which the function needs read access but not write access. For whole arrays (including strings) and for single variables to which the function needs write access, call by reference must be used. See **7.3 Array addresses as function parameters** for an example of an array reference parameter.

Call by value and reference example

```
void copy_int(int from,int *to);    /* prototype of copy_int function
                                   * from is called by value,
                                   * to is called by reference.
                                   */

int main(void){
    int a=3,b;
    copy_int(a,&b); /* a is passed by value, b by reference */
    printf("a: %d b: %d\n",a,b);
    return 0;
}

void copy_int(int from,int *to){ /* definition of copy_int function */
    *to = from; /* copies from value, to storage addressed by to */
}
```

8.6 Return types

Function of void type do not require a return statement. However, these can be used to return from the function as an alternative to using other flow control approaches causing execution to finish at the last statement written in the body of the function. Functions returning other (non void) types require an object in return statement/s compatible with the return type. E.G:

```
int main(void){ /* do nothing except return a 1 to the operating system */
    return 1;
}
```

Here the object returned (1) is of the same type (int) as the main function.

The return statement can return only a single variable. However, this may be a structured type defined by the programmer. When a function returns a value, the function call may be considered as being replaced by an expression which returns the same value. Example:

```
#include <stdio.h>

float square(float); /* prototype of square function */

int main(void){
    float a=2.0,b;
    b=square(a); /* assign the square of a to b */
    printf("a: %f b: %f\n",a,b);
    return 0; /* value returned by main function */
}

float square(float value){ /* definition of square function */
    return value * value; /* compute and return the square */
}
```

output
a: 2.000000 b: 4.000000

8.7 Localisation, duration and globalisation of data

It is good design practice to modularise data as well as functionality. Variables declared inside function blocks are considered local to the functions in which they are declared. This includes parameter variables. Normally local variables are automatically created and destroyed when the function starts and ends executing. This might not be the required behaviour:

- a. Either if modular design requires that data defined within a function continues to be available to other program units after that function has returned, e.g. through a returned pointer variable,
- b. Or if the function can usefully maintain state information between invocations e.g. so that successive calls to the same function can iterate

through a collection of data without having to restart from the beginning each time.

In these situations, static definition of local data is desirable to enable this data to persist beyond the function return. Example:

```
char *weekday(int day){
    /* function which returns pointer to static weekday string */
    static char *days[]={"Invalid Input", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday", "Sunday"}; /* static declaration */
    if(day>=1 && day<=7)
        return days[day]; /* pointer to weekday string */
    else /* not a valid number of day in week */
        return days[0];
}
```

Global data is declared outside of any function, typically at the top of the source file or within header files. Use of global variables can violate modularity of design and explicitness of declared interfaces, so should be carefully controlled. The situations where global data is of benefit are likely to include program constants and structured type definitions.

Global variable data can also benefit program design in situations where an entire program is designed to access a particular data structure, e.g. an array of structure records. In this situation, declaring this structure within main and having additional parameters to every, or most functions in the program to access this data structure would violate the design objective of simplicity.

9. Pointers

9.1 References and values

Without references, values would either have to be global, or would have to be cloned (copied) in order to be usable in different modules within a program. Inappropriate use of global data violates modularity leading to bad design. References allow more than one name, or means of access, to exist

for the same object, which does not have to be copied. Having copies made from the same object in 2 or more places is frequently undesirable, partly due to the waste of memory, but more importantly because updates to an object accessed from one place have to be made usable elsewhere to avoid confusion and error.

The 'C' language uses pointer variables to store references, and various means to obtain and manipulate data through the use of references.

9.2 Address of operator

A reference can be obtained from a single object using the & (address of) operator together with the object to be referenced. Example:

```
int i; /* declare storage for integer object i */
printf("enter a number\n");
scanf("%d",&i); /* pass address of i (by reference) to scanf */
```

9.3 Pointer variable declarations

Preceding the name of an object with an asterisk when it is declared indicates this object will store a reference, or pointer, or address of the required type. Pointer variables can be dynamically assigned the addresses of objects of the declared type, or whole array references, or references to segments or elements of an array of the same type, e.g.1 pointer variable declaration:

```
int *ia; /* ia is a pointer variable so can be assigned an address of an int */
```

e.g.2: prototype reference parameter declaration:

```
void swap(int *a, int *b); /* swaps 2 variables in caller. Parameters a and
    b are declared as pass by reference/pointer variables local to swap */
```

9.4 Indirection operator

Data processing statements involving pointers use the indirection or dereferencing * (asterisk) operator to mean the storage or value pointed at by the pointer variable. This is as different a meaning from use of the * asterisk in a data declaration as both are from use of * asterisk to indicate multiplication.

To tell which is which, all data declarations are in the top part of a block and associate parameter variables with data types. Data processing statements follow below in the block.

```
void swap(int *a, int *b) { /* declares a and b as pointer parameter variables */
    int temp; /* local storage of type needed for data swap */
    temp = *a; /* assign contents referenced by a to temp */
    *a = *b; /* assign contents referred to by b
              to storage pointed at by a */
    *b = temp; /* assign value of temp to storage pointed at by b */
}
```

9.5 Arrays as references

An example is given in section 7.3 (Array addresses as function parameters). For all arrays in 'C' the name of the array used in a data processing statement (e.g. a function call) evaluates to the address at which the array is located. Declaring a pointer parameter as array[] in a data declaration is syntactically equivalent to declaring it as *array, however, the first form may be preferred to give a visual clue that the variable is an array.

9.6 Pointer arithmetic

The expression array + i is equivalent to the expression &array[i]. In other words it yields the address of the i'th element of the array counting from 0. This is possible because pointer variables are typed. A pointer variable stores 2 datums, the address and the size of its data type. A void pointer only stores the address. An example is given in section 7.2 (Array access and indexes).

10. Structures

10.1 struct declaration

These declarations are most likely to be global. Arrays organise a set of data objects all of the same type. Struct declarations define a record which may be composed of different types. Examples:

```
struct student {
    char sno[12]; /* student number */
    char name[30]; /* student name */
    float mark; /* module mark */
};
```

```
struct coordinate {
    float x;
    float y; /* x and y members of coordinate object */
};
```

In the second example, the members `x` and `y` are of the same type. The reason for use of a structure in preference to an array is because this results in clearer naming and access to coordinate objects and members.

10.2 Type definition

The structure declarations above are commonly supplemented by or enclosed in type definitions, to enable the new types to be expressed as one word rather than two. Examples:

```
typedef struct student STUDENT; /* struct student must exist for this to work */
```

```
typedef struct coordinate {
    float x;
    float y; /* x and y members of coordinate object */
} COORDINATE ; /* you can define structure and type together */
```

By convention, typedefs use UPPER_CASE.

10.3 Structure variable declaration

In data declaration statements the type precedes a list of variable names separated by commas. Examples:

```
struct student me, group[20]; /* single variable me, and array of 20 students.
```

```
COORDINATE origin, cursor, graph[10]; /* preferred */
```

Use of one word typedefs is preferred to having 2 for the type.

10.4 Access to member variables

Direct access to members involves use of the variable name, followed by a dot (.) followed by the member name. Examples:

```
strcpy(me.sno,"ABC12345678"); /* copy student number string */  
me.mark=50.0; /* assign mark */
```

```
for(i=0;i<10;i++){  
    printf("input x and y for coordinate %d\n",i+1);  
    scanf("%f%f",&graph[i].x,&graph[i].y);  
}
```

10.5 Use of structure pointers

Where ptr is a pointer variable storing the address of a structured variable, and mem is a member field, the syntax ptr->mem directly accesses the value of member mem of the structure variable whose address is stored in ptr.

Examples:

```
STUDENT me,group[20],*sptr;  
int i;
```

```
sptr=&me;
strcpy(sptr->sno,"ABC12345678"); /* sptr->sno is same as me.sno */

sptr=group; /* make sptr store address of group array */
for(i=0;i<20;i++)
    (sptr+i)->mark=0.0; /* make all marks in group 0 at start of semester */
```

In the example above () braces were needed around the expression `sptr+i` because addition (as in pointer arithmetic or elsewhere) takes a lower precedence to the `->` structure member indirection operator.

10.6 Structures and arrays

See `graph` and `group` variables in examples above for usage examples of arrays of structures. The `struct student` typedef also includes string arrays as members within a structure. Array elements within member arrays can be accessed by placing the array [] indexing brackets after the member name.

Example:

```
printf("The initial letter of the first student in the group is: %c \n",
      group[0].name[0]);
```

Clearly the first [0] indexes the first student within the group, and the second indexes letters within the name member of the first student.

11. Library functions

Information in this section was summarised using information taken from Kernigan and Ritchie, "The 'C' Programming Language" 2e. Only a selection of functions available in the 1988 ANSI 'C' standard are included. Selection is made based on the probability of these being useful in coursework, or in an examination requiring the student to write source code in 'C'.

11.1 Standard input and output

11.1.1 Header include

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

11.1.2 printf

```
int printf(const char *format, ... )
```

Prints arguments in a formatted manner on standard output.

printf(...) is equivalent to fprintf(stdout, ...)

example

```
printf("name: %s weight: %.2f age: %d\n", "Fred", 54.123, 37);
```

output

```
name: Fred weight: 54.12 age: 37
```

Escapes

%s is used to convert string arguments, %f for float, %d for int, %c char, %e double, %ld long int, %x for hexadecimal output. Interpolating numbers between the % and the following letter can specify output field width and precision e.g. %10.2f specifies a field width of 10 and a precision of 2 decimal places for decimal floating point output. Escapes in format are matched to the

following parameters sequentially and it is invalid to specify escapes for which parameters are not supplied and vice-versa.

Backslash (\) is used to prefix non-printable or inconvenient character escapes in format, these include: \n for newline, \t for tab, \\ for a single backslash, \" for double quote, \xhh for hex number: hh. A literal % can be escaped as %%.

11.1.3 fprintf

```
int fprintf(FILE *stream, const char *format, ... )
```

Prints arguments in a formatted manner on specified output file. The returned value is the number of characters written or negative on error. Output formatting is the same as for printf. The difference is that fprintf requires an extra (initial) parameter to specify the output filehandle.

example:

```
FILE *out; /* declare filehandle out */
out=fopen("output.txt","w"); /* open output.txt file for writing */
fprintf(out,"hello text file world\n"); /* write text to output.txt file */
```

11.1.4 sprintf

```
int sprintf(char *s, const char *format, ... )
```

This is a similar function to fprintf and printf, the difference being that instead of printing to a filehandle or stdout, formatted output is directed to a string. The string address is specified as the first parameter. This is useful for converting from binary numeric format for numbers into human readable decimal representations in string format. Example:

```
char buffer[1024], name[]="Fred";
int age=42;
float weight=55.6;
sprintf(buffer,"Name: %s Age: %d Weight: %.2f \n",name,age,weight);
```

```
printf(buffer);
```

output:

Name: Fred Age: 42 Weight: 55.60

11.1.5 scanf

```
int scanf(const char *format, ... )
```

Reads text from standard input and converts data for use in program variables. All variables to which scanf provides input must be passed by reference. Single variables must either be pointers or use the address of (&) operator. The name of a string variables, being an array, is its address in 'C'.

scanf(...) is equivalent to fscanf(stdin, ...)

Example

```
int age;
float weight;
char name[30];
printf("enter age, weight and name\n");
scanf("%d%f%s",&age,&weight,name);
```

Note that in this example scanf uses white space to delimit input fields, e.g spaces, tabs and newlines. String input using scanf is not suited to fields containing embedded whitespace. The same conversion escapes in connection with data types are used as by printf.

11.1.6 fscanf

```
int fscanf(FILE *stream, const char *format, ... )
```

Reads arguments in a formatted manner from specified input file. Input formatting is the same as for scanf, fscanf requires an extra (initial) parameter to specify the input filehandle. Example:

```
FILE *in; /* declare filehandle in */
int i;
in=fopen("input.txt","r"); /* open input.txt file for reading */
fscanf(in,"%d",&i); /* read integer from start of input.txt (must be there) */
```

The return value of `fscanf` is the number of items read or EOF on end of file or error. This may be used to determine when all input data has been read.

Example:

```
while(fscanf(in,"%s%d",name,&i) != EOF){
/* read data until end of file */
    ... /* process data */
}
```

`fscanf` is fragile, in the sense that if the data present in the file is not compatible based on its format specification this is likely to result in program crashes or bugs.

11.1.6 `sscanf`

```
int scanf(char *s, const char *format, ... )
```

This is a similar function to `fscanf`, the difference being that instead of reading from a filehandle, formatted input is read from a string. The string address is specified as the first parameter. This is useful for converting from human readable decimal data in strings into binary numeric format. It enables untrusted data to be read as strings and validated prior to conversion to binary variables. Example:

```
char buff[]="42 56.5 Fred",name[30];
int age;
float weight;
sscanf(buff,"%d%f%s",&age,&weight,name);
```

11.1.7 `fflush`

```
int fflush(FILE *stream)
```

Used to flush files. Return EOF on error. Often used to flush unread data from input files. e.g.

```
fflush(stdin);
```

See `getchar` for further explanation and example.

11.1.8 `getchar`

```
int getchar(void)
```

Equivalent to `getc(stdin)`. This function reads and returns a single character from standard input e.g:

```
char option;  
printf("do you want to continue ? y/n \n");  
option=getchar();
```

A programmer making use of `getchar` in connection with repeated input of chars and other variables using various input functions, e.g. `scanf`, should be aware of the need to flush unread newline '\n' characters from standard input using `fflush` prior to using `getchar`. Example:

```
int i;  
char c;  
printf("input a number\n");  
scanf("%d",&i);  
printf("input a character\n");  
fflush(stdin); /* need to flush the leftover \n from the input stream */  
c=getchar();
```

11.1.9 `getc` and `fgetc`

```
int getc(FILE *stream)
```

Reads and returns the next character read converted to an unsigned int from

stream, or EOF if error or end of file occurs. `fgetc` is similar. Example:

```
FILE *in,*out; /* in and out are pointers of type FILE */
char c;
in=fopen("master.txt","rt"); /* initialise in FILE pointer */
out=fopen("backup.txt","wt"); /* initialise out FILE pointer */
while((c=getc(in)) != EOF) /* copy all chars in to out until EOF */
    putc(c,out); /* writes char c to FILE *out */
fclose(in); fclose(out);
```

11.1.10 `putc` and `fputc`

```
int putc(int c, FILE *stream)
```

Writes character `c` to stream. Returns EOF on error. `fputc` is similar. See `getc` for example.

11.1.11 `puts`

```
int puts(const char *s)
```

Used to write a string and a newline to standard output. Returns EOF on error. Example:

```
puts("hello world");
```

11.1.12 `gets`

```
char *gets(char *s)
```

Reads the next input line into string `s`, replacing the ending `\n` with `\0`. Returns the pointer `s` or `NULL` in error or at end of file. Example:

```
char name[30];
printf("input name\n");
```

```
gets(name);
```

11.1.13 fgets

```
char *fgets(char *s, int n, FILE *stream)
```

Reads up to n-1 chars from stream or until a newline is read. All chars read including the newline are stored in s. Returns NULL if end of file or error occurs. fgets is commonly used for secure input, as ensuring n is equal to or less than the storage addressed by s prevents overflow. Example:

```
FILE *in;
char buffer[1024];
in=fopen("c:\\autoexec.bat","r"); /* open in read mode */
fgets(buffer,1024,in); /* read up to 1023 chars or
                        first \n from in filehandle */
printf("The first line of the AUTOEXEC.BAT file is:\n%s\n",buffer);
fclose(in);
```

11.1.14 fopen

```
FILE *fopen(const char *filename, const char *mode)
```

Used to open filename for reading, writing or both, based on mode. Returns NULL if unsuccessful, or address (i.e. the filehandle) of the file opened.

Simpler modes include "r" to read a file, "w" to write at start of file discarding any previous contents and "a" to write at end of file. See fgets for example.

11.1.15 fclose

```
int fclose(FILE *stream)
```

fclose flushes unwritten output data to the file, discards unread input data and closes the file. Returns EOF if unsuccessful. See fgets for example.

11.1.16 fread

size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)

Used for direct binary file reading. Reads (nobj x size) bytes from stream, storing these at memory range addressed by ptr. Returns no. of objects read. size_t is an unsigned integer type suitable for sizing file objects. Example:

```
FILE *in;
int ia[100],bytes,i=0; /* we know or hope there are less than 100 ints in the file
*/
in=fopen("ints.bin","rb"); /* read and binary mode */
while((bytes=fread(ia+i,sizeof(int),1,in))>0)
    /* reads integers one at a time into ia[] until EOF */
    printf("integer %d is %d\n",i+1,ia[i++]);
printf("there were %d integers in the file\n",i);
```

11.1.17 fwrite

size_t fwrite(void *ptr, size_t size, size_t nobj, FILE *stream)

Used for direct binary file writing. Writes (nobj x size) bytes to stream, using data taken from location addressed by ptr. Returns no. of objects written.

E.G:

```
int ia[]={2,4,8,16};
FILE *out;
fopen("ints.bin","wb");
fwrite(ia,sizeof(int),sizeof(ia)/sizeof(int),out);
/* writes contents of ia[] to binary file */
```

11.1.18 fseek

int fseek(FILE *stream, long offset, int origin)

Sets position of cursor in file for start of next read or write operation. This is typically required when files are not accessed sequentially, but at a program determined position in the file. origin is provided from an integer constant, symbolic values are `SEEK_SET` (start), `SEEK_CUR` (current position) or `SEEK_END` (end position of file), and offset is in bytes relative to origin. Returns non-zero on error. Example:

```
fseek(in,100L,SEEK_SET);
/* positions file cursor 100 bytes from the start of file */
```

11.1.19 rewind

```
void rewind(FILE *stream)
```

Positions cursor at start of file and clears any error indicators for stream.

11.1.20 fgetpos

```
int fgetpos(FILE *stream, fpos_t *ptr)
```

Type `fpos_t` is an integer type suitable for storing file cursor positions. `fgetpos` causes the position of the file cursor in stream to be written to the variable addressed by `ptr`. Returns non-zero on error.

11.1.21 fsetpos

```
int fsetpos(FILE *stream, const fpos_t *ptr)
```

`fsetpos` causes position of file cursor in stream to be changed to the value of variable addressed by `ptr`. Normally this variable would have been set previously using `fgetpos`. Returns non-zero on error.

11.2 String handling

11.2.1 header

```
#include <string.h>
```

11.2.2 strcpy

```
char *strcpy(char *target,const char *source)
```

Copies source string to target string, stopping at \0 . Example:

```
char name[6]; /* extra byte needed for \0 */  
strcpy(name,"Harry"); /* name is now "Harry" */
```

11.2.3 strncpy

```
char *strncpy(char *target,const char *source, size_t n)
```

Copies no more than n chars from source to target, padding target with \0 chars. Returns target. Example:

```
char name[6]; /* extra byte needed for \0 */  
strncpy(name,"Desdemona",5); /* name is now "Desde" */
```

11.2.4 strcat

```
char *strcat(char *target,const char *source)
```

Concatenates (joins) source string to the end of target string, returns target. Example:

```
char hello[20]="Hello";  
printf(strcat(hello,"World")); /* outputs HelloWorld */
```

11.2.5 strncat

```
char *strncat(char *target,const char *source,size_t n)
```

Concatenates (joins) at most n chars of source string to the end of target string. Returns target. Example:

```
char hello[20]="Hello";  
printf(strncat(hello,"Worldview",5)); /* outputs HelloWorld */
```

11.2.6 strcmp

```
int strcmp(const char *s1, const char *s2)
```

Compares strings s1 and s2. Returns negative integer if s1 less than s2, positive integer if s1 greater than s2 and returns 0 if they are identical. Example:

```
char passwd[30];  
printf("enter password\n");  
scanf("%s",passwd);  
if(strcmp(passwd,"sesame") == 0) /* check for correct password */  
    printf("your wish is my command\n");  
else  
    printf("no entry\n");
```

11.2.7 strlen

```
size_t strlen(const char *s)
```

Returns no. of chars in string s prior to first \0. Example:

```
printf("%d\n",strlen("hello")); /* outputs 5 */
```

11.3 Mathematical

header

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

All functions below have double parameters and return double.

$\sin(x)$, $\cos(x)$, $\tan(x)$, $\log(x)$ \sin , \cos , \tan of x , with x in radians.

$\log(x)$, $\log_{10}(x)$ natural log of x and log base 10 of x .

\sqrt{x} square root

$\text{fabs}(x)$ absolute value of x

$\text{pow}(x,y)$ x raised to the power y

11.4 miscellaneous

11.4.1 header

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

11.4.2 malloc

```
void *malloc(size_t size)
```

This returns a pointer to a dynamically allocation of size bytes of memory from the heap, or NULL on failure. For an example see 7.5 Dynamic arrays.

11.4.3 free

```
void free(void *ptr)
```

This frees a heap allocation made by malloc, calloc or realloc addressed by ptr.

11.4.4 atof

```
double atof(const char *s)
```

Converts string *s* containing decimal to double precision float.

11.4.5 atoi

```
int atoi(const char *s)
```

Converts string *s* containing decimal to integer.

11.4.6 rand

```
int rand(void)
```

Returns a pseudo-random integer in the range 0 to RAND_MAX . Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* required for time function */
int main(void){
    int i;
    srand(time(NULL)); /* seconds since epoch is too predictable
                        a seed value for cryptographic use. */
    printf("RAND_MAX: %d\n",RAND_MAX);
    for(i=0;i<10;i++) /* generate and print 10 random ints */
        printf("%d\n",rand());
    return 0;
}
```

output

```
RAND_MAX: 2147483647
706401704
999954375
448005111
1353913345
312400476
720169393
406591281
```

873707680

721815575

535261251

11.4.7 srand

`void srand(unsigned int seed)`

Seeds the rand random number generator. Uses seed integer as the seed which determines the starting point of a new pseudo-random sequence to be generated using rand. See rand example above, where srand uses a changing value based on the system clock to seed the sequence.

11.4.8 exit

`void exit(int status)`

exit is used to terminate execution of a program. It may be used in any function, but its use within the main function is similar to the effect of a return statement. Its use is a shortcut for stopping the program, and is typically used if an error condition is detected within a function. Use of exit avoids the need for more complex flow of control within the calling function (or stack of returns) which might otherwise be required. The status is returned to the operating system.