Introduction to C Programming

A hands-on course on the C programming
language organised by the Professional Development
Programme and given by the Department of Computer Science.

Jim Bell
Jiannong Cao
Olivier de Vel

– May 1992 –

Introduction to C Programming

Programme Outline

1) History and background

2) Planning software for the C language

3) Getting started (my first C program)

4) Syntactic and semantic level

5) Types of data elements, operators, and expressions

6) Flow of Control

7) Functions and program structure

8) Pointers, arrays and strings

9) Static structures

10) Dynamic structures and list processing

11) Input/output, file access

12) The C library and UNIX system interface

Reference Book :

"*The C Programming Language (ANSI C)*" by Brian Kernighan and Dennis Ritchie, Second Edition, Prentice-Hall Publishers, 1988.

Programme Schedule

Day 1 (Tuesday, 16 June 1992)	
09:00 - 09:15	Introduction to course
09:15 - 10:45	Lectures & Hands-on (Olivier de Vel)
	History and Background, Planning Software for C, Getting Started
10:45 - 11:00	Tea/coffee break
11:00 - 12:30	Lectures & Hands-on (Olivier de Vel)
	Syntax & Semantics, Data Types, Operators and Expressions, Flow of Control
12:30 - 13:30	Lunch
13:30 - 15:00	Lectures & Hands-on (Olivier de Vel)
	Functions, Program Structure
15:00 - 15:15	Tea/coffee break
15:15 - 17:00	Lectures & Hands-on (Olivier de Vel)
	Functions, Program Structure

Day 2 (Wednesday, 17 June 1992)	
09:00 - 10:30	Lecture (Jiannong Cao)
	Pointers and Structures
10:30 - 10:45	Tea/coffee break
10:45 - 12:30	Lecture & Hands-on Exercises (Jiannong Cao)
	Pointers and Structures
12:30 - 13:30	Lunch
13:30 - 14:00	Program Presentation (Stuart Kemp)
14:00 - 15:00	Hands-on Exercises
15:00 - 15:15	Tea/coffee break
15:15 - 17:00	Hands-on Exercises

Day 3 (Thursday, 18 June 1992)	
09:00 - 10:30	Hands-on Exercises (continuation)
10:30 - 10:45	Tea/coffee break
10:45 - 12:30	Lecture (Jim Bell)
	Input/Output, Unix System Interface
12:30 - 14:00	Lunch
14:00 - 15:30	Hands-on Exercises
15:30 - 15:45	Tea/coffee break
15:45 - 17:00	Overview, Questions, Evaluation

Contents

Chapter 1

History and Background

1.1 The Origin of the C Language

C was invented/designed by Dennis Ritchie (Bell Labs, now called AT&T).

Ritchie wanted a high-level procedural language (rather than an assembler) suitable for writing an operating system - an OS which manages I/O, allocates data storage, schedules programs etc...
C was designed to be the systems language for the UNIX operating system.

The initial versions of UNIX were in fact designed and developed by Ken Thompson (the "father" of UNIX) using assembly language and the B language. B was based on the BCPL language as a typeless systems programming language. It made heavy use of pointers (which provided for machine-independent address arithmetic).

C was invented to overcome the limitations of B (by incorporating typing – e.g. integers, floating-point etc...).

Due to its origins as a systems language, C is therefore not a "very high-level" language, nor is it a "big" language (e.g. C has fewer keywords than Pascal).

About 80-90% of a typical UNIX operating system is written in C.

1.2 Why Use C ?

C is a <u>small language</u> – and small is beautiful in programming!

C gets its power by carefully including the right control structures and data types – and allowing their uses to be nearly unrestricted where meaningfully used.

Languages do not gain popularity on their own merits. The hidden secret of a language's success is the system environment.
e.g. C does not need to have embedded I/O constructs (e.g. "read" and "write" statements, as might be found in Pascal or FORTRAN) or complicated interrupt handlers – but instead relies on library routines for these functions.
e.g. C offers only single-thread control flow (e.g. loops, subprograms etc...) but no coroutines or synchronisation facilities (e.g. as in Modula-2, Ada etc...) – these are provided by the UNIX environment.

C is portable – by virtue of being small and initially being defined on a small machine (a PDP-11). The C compiler is itself very small.

C is terse. C has a powerful set of operators – indirection and address arithmetic (pointers) can be combined within a single expression. This is seen to be elegant to some (but obscure for others to read). This underlies the increased productivity of programmers.

C is modular. C supports one style of routine (the function) which calls parameters by value. Functions are declared as individual objects and they cannot be nested (unlike Pascal which considers a function to be a block which can be nested).
In fact, the main programme is just a function.

C is not without criticism :

- it is not strongly typed (as in Ada, Modula-2 etc...),
- it makes multiple use of the same symbols (e.g. "*" is used for multiplication and for pointers),
- some operators have the wrong precedence,
- it does not provide direct operations on strings, sets etc..., and
- the compiler is too liberal (e.g. it has no automatic array bounds checking, it reorders evaluation within parameter lists, etc...).

Therefore, C strives for functional modularity and effective minimalism.

1.3 Implementations of C

In the early 1980s there was no standard for the C language comparable to the FORTRAN or Pascal languages.

In fact the first edition of Kernighan & Ritchie's book (in 1978) served as a reference.

Despite the lack of an official standard, there was never much interest in "dialects" of C – which proves that that C does in fact meet a wide range of needs.

In 1983, the American National Standards Institute (ANSI) prepared a standard for release in 1986 (but actually only completed in 1988) – the ANSI C.

Tha ANSI C (the one used in this course) is based on the original 1978 K & R version of C – with a few enhancements (particularly at the level of function declaration and structures).

Chapter 2

Planning Software for the C Language

2.1 The Stages of Software Development

Programming is the activity of communicating algorithms to computers.

An algorithm is a computational procedure whose steps are completely specified (e.g. making a cake).

In English, we can easily give out instructions to other people who will carry them out (hopefully !). A certain degree of ambiguity and impreciseness is tolerable.

In programming, the process is similar – however computers have no tolerance for ambiguity and must have all steps specified in tedious detail.

That is, computers require simple, but very precise languages. The syntax and semantics of programming languages will therefore need to be very exact.

$$\begin{tikzcd}\text{SYNTAX} \arrow[r] & \text{SEMANTICS} \arrow[r] & \text{PRAGMATICS} \arrow[r] & \text{AXIOMATICS} \\ \text{(programming languages)} & & & \text{(real-world)}\end{tikzcd}$$

Many people think that software development is mainly ”programming”, by which they mean writing in some computer language.

However, software development is usually organised in projects, with explicit goals, assignments, and schedules.

Software projects typically include the following stages:

Software definition This is concerned with the external operation of the software. That is, "what" is to be done.

The objective is to specify the services that software should deliver (as required by the customer).

The end-product is a software specification describing those services in realistic detail.

Software design This is concerned with the internal organisation of the software. That is, "how" to do it.

Usually use step-wise refinement to produce a set of "segments" each performing a particular type of work.

The smallest segment corresponds to (in C) the function. Segments which perform closely related tasks are grouped into "modules" (e.g. segments that share data and conventions – a set of operations on a linked list).

Should try and keep the connections (interfaces) between software segments simple and direct. There are three major types of connections :

→ flow of control (by invoking C functions),

→ shared access to data (through global declarations), and

→ shared conventions (through preprocessor facilities, such as the **#define** statement and **#include** header files)

Software implementation The code itself.

Software validation This makes sure that an entire software system works and prove that it works reliably (precision and correctness).

Involves : module units testing, integration testing, system testing, and performance testing.

Software maintenance This is to produce documentation, modify existing features, add new features etc....

Chapter 3

Getting Started with C

In this chapter, we give an overview of the C programming language.

We present a few programs, explaining the basic elements of each program, without getting bogged down in details, rules and exceptions. These will come later ....

3.1 The First C Program – A Classic !

Using the vi editor, type in the following "hello world" program (also on page 6 of K & R) :

```
#include <stdio.h>

main()
{
        printf("hello, world\n")
}
```

The steps for writing and running a program are as follows :

Step 1) Create a text file with name ending in ".c" and type a C program into it :

vi hello.c

Step 2) Compile the program using the **gcc** (GNU C) compiler :

gcc hello.c

which produces an executable file "a.out", or type in,

gcc -o hello hello.c

which produces executable file "hello".

We can also use [optionally] the facility **lint** to check for syntax errors :

lint hello.c

Step 3) Execute the program as follows :

a.out (in the first **gcc** case)

hello (in the second **gcc** case)

Dissection of the Program :

#include <stdio.h>

This tells the compiler to include information (stored in a system file "stdio.h") about the standard input/output.

main()

Every program has a function named "main" where execution begins. The parentheses "()" indicate that it is a function.

{

Braces surround the body of function (braces can also be used to group statements together). The "{" brace opens the body of the "main" function.

printf()

This is a function from the standard C library (stdio.h) that prints on the screen the sequence of characters enclosed in the brackets.

"`hello, world\n`"

Sequence of characters enclosed in double quotes to be printed on screen. Notice that `\n` is the "new line" character (optional).

}

This brace matches the left brace above and ends the function "main".

[Exercise : Repeat the "hello, world" program, this time writing "Hello" and "World" on separate lines]

3.2 Variables, Assignment, and Expressions

Let us study the following simple program (Section 1.2 of K & R) :

```
#include <stdio.h>

/* print Fahrenheit–Celsius table for fahr = 0, 20, ..., 300 */

main()
{
        int fahr, celsius;
        int lower, upper, step;

        lower = 0;      /* lower limit of temperature table */
        upper = 300;    /* upper limit of temperature table */
        step = 20;      /* temperature step size */

        fahr = lower;
        while (fahr <= upper) {
            celsius = 5 * (fahr−32) / 9;
            printf("%d\t%d\n", fahr, celsius);
            fahr = fahr + step
        }
}
```

Variables :

Since C is a typed language, we must *declare* all variables before they are used. This is done usually at the beginning of the function.

A variable declaration announces the properties of variables. This is as follows :

type_name $\quad$ *variable*$_1$*[, variable*$_2$*, . . . , variable*$_n$*] ;*

such as (simple "scalar" data types shown only) :

int	fahr, celsius;
short	int avar;

```
long int  bvar;
float     kilometers;
char      c1, c2, c3;
double    distance;
```

The type "**int**" means that the variables listed are integers. Integers can be 16-bit (between -32768 and 32767) or 32-bit (-2^{31} to 2^{31}-1) words depending on the machine. 16-bit **int**s are called "short", 32-bit **int**s are called "long".

A "**float**"ing point number is typically 32-bits, with at least 6 significant digits (i.e. precision), and a magnitude (i.e. dynamic range) between approx. 10^{-38} and 10^{38}.

The type "**char**" is used for character constants.

The type "**double**" refers to double-precision floating-point numbers, with an approximate precision equal to 16, and a dynamic range of identical to a that of a "**float**".

We can initialise a variable while declaring it. For example :

```
char   esc = '\\';
int    limit = MAXLINE + 1; /*MAXLINE has been defined*/
long   day = 1000L*60L;
char   hello[ ] = "hello there";
char   *hello = "hello there";
int    vector[ ] = { 1, 2, 3 };
```

We can also qualify a variable declaration, by using "**const**". For example,

```
const char  msg[ ] = "Warning :  ";
```

which specifies that the value of "msg" will not change during execution (i.e. it is stored as read-only data).

There are other also *arrays, structures* and *unions* of these basic types. There are *pointers* to them, as well as *functions* that return them. We shall meet these in due course.

Assignments :

The executable code on page 19 begins with various assignment statements (which, in this example, serve the purpose of initialising variables). Statements are terminated by a semicolon ";" (it is a statement **terminator** and NOT a statement separator, as in Pascal).

Expressions :

Most of the work in the program on page 19 is done in the "**while**" loop. (Notice the identation – not obligatory, but suggested).

Expressions, such as

```
celsius = 5 * (fahr - 32) / 9;
```

are typically found on the right-hand side of assignment statements and as arguments to functions (such as might be found in a "printf" library function).

The simplest expression involves just a constant (such as "upper = 300;"), or just the name of a variable (such as "fahr = lower;").

Variables can be combined with operators such as $*$, / + etc...to form an expression. In this case the evaluation of an expression will involve conversion rules. In the example, the Celsius temperature involves integer operations so, if we had

```
celsius = (5 / 9) * (fahr - 32);
```

the result would have been zero (i.e. $5/9 = 0$).

Primo : Notice how we can define the field width and precision of the "printf" conversion specification. Examples are listed on page 13 of K & R.

Segundo : There are many ways of translating an algorithm into an actual program. In this case, we can write the "while" loop as a "for" loop (see bottom of page 13 of K & R).

Ultimo : One way of introducing data abstraction is by using symbolic constants. This is done by using

#define name *replacement_text*

which abstracts away the "replacement_text" with "name". The "**#define**" statement is a preprocessor macro. Symbolic constants are evaluated and expanded at compile-time. (Notice that there is no ";" at the end of the statement).

For an example on the use of symbolic constants, see the top of page 15 of K & R).

Other examples :

```
#define   EQ   ==
```

This allows us to use "EQ" in lieu of "==" in the program. That is :

if (i EQ 1)   *rather than*   **if** (i == 1)

Also,

```
#define   WRITE(X)   printf(#X " =%d\n", X)
```

replaces the program statement

WRITE(abc)   *with*   printf("abc" " =%d\n", abc);

All of these are examples of "syntactic sugar".

3.3 Character Input and Output

How do we process character data (for both input and output) ?

I/O is supported by the C library. The model for textual I/O is based on a stream of characters .

A text stream is a sequence of characters divided into lines – each line consisting of zero or more characters followed by a newline character (`\n`).

To read (write) one character at a time, use the standard input (output) library call "getchar" ("putchar"). Each time the function is called, as

```
c = getchar();
```

"getchar" reads the next input character from a text stream and returns that as its value in variable "c".

Variable "c" could be declared as a "**char**" data type. But one problem arises when we wish to distinguish the end-of-input from valid data. A special value needs to be defined that cannot be confused with any real character – this value is called end-of-file, or EOF (EOF is stored in "stdio.h" as -1). The variable "c" is therefore defined as "**int**".

Study the following program :

```
#include <stdio.h>

/* copy input to output; 1st version */

main()
{
        int c;

        c = getchar();
        while (c != EOF) {
           putchar(c);
           c = getchar();
        }
}
```

Compare the above program with the one on page 17 of K & R. [Exercise : Type in and excute the program on page 17 of K & R]

Notice the precedence of the operators "!=" over "=", hence the parentheses.

Prior to executing the program on page 17 of K & R, try using the UNIX command :

	stty cbreak	(makes each character available)
or,	**stty -cbreak**	(makes each character available after a newline is received)

Typically (in UNIX), an EOF is generated (from the keyboard) by doing a "`^D`" after a carriage return, with **stty -cbreak** having being set.

Now have a look at the program on page 20 of K & R (word counting). Notice :

- the initialisation (nl = nw = nc = 0)
- the "increment-by-one" prefix operator (++nc)
- use of the "**if**" conditional statement
- the blank (' ') and tab ('`\t`') characters
- the OR Boolean operator (with left-to-right evaluation)

Another input (output) function which we will useful at this stage is the formatted input (output) "scanf" ("printf") C library call.

The output function "printf" translates internal values to characters – it converts, formats, and prints its arguments on the standard output. It also returns the number of characters printed.

Standard output is, by default, the screen – but it can be redirected by means of "output redirection" (filters or pipes) at the UNIX command level :

e.g.	**hello > FileName**	(redirects to file "FileName")
e.g.	**hello | AnotherProgram**	(redirects to another program)

The basic format of "printf" is

printf (*format_string, arg1, arg2, . . .*)

The *format_string* typically involves ordinary characters and conversion specifications. The conversion specification begins with a "%" and ends with a conversion character. It may specify field width as well as precision.

Examples include :

```
printf ("The string is %s\n", string);   /*string is a character array */
printf ("Numbers are %d and %6d", max, min);   /* integer numbers */
printf ("Value is %10.3f\n", val);   /*floating point number */
```

Standard input is, by default, the keyboard – but can be redirected by means of input redirection (filters or pipes) at the command level.

The basic format of scanf" is

scanf (*format_string, arg1, arg2, . . .*)

"scanf" reads characters from the standard input, interprets them according to the specification in *format_string*, and stores the results through the remaining arguments. Parameters are passed by value, so these arguments MUST be the addresses of the variables i.e. they must be pointers.

The *format_string* typically involves conversion specifications which are used to control conversion of the input. "scanf" will read across white spaces to find its input (white spaces are e.g. blank, tab, newline, carriage return). The conversion specification includes a "%" followed by a (optional) maximum field width and ends with a conversion character. Examples include :

```
scanf ("%d", &day);     /* "day" is of type "int" */
scanf ("%lf", &sum);    /* "sum" is of type "double" float*/
```

Let us now concentrate on the details of the C language.

Chapter 4

Syntactic and Semantic Level

A C program is a sequence of characters that will be converted by a C compiler to a target language (usually machine language) on a particular machine.

A C program is constructed out of a sequence of characters that include :

lowercase letters	a b c ... z
uppercase letters	A B C ... Z
digits	0 1 2 3 4 5 6 7 8 9
special characters	+ = _ − ( ) * & % $ # ! | < >
	. , ; : " ' / ? { } ~ \ [ ] ^
nonprinting characters	white space i.e. blank, newline, tab

These characters are collected by the compiler into syntactic units called tokens, which comprise the basic vocabulary of the language (e.g. identifiers, keywords, constants, strings, operators, etc...).

The compiler then checks that the tokens can be formed into legal strings according to the syntax of the language. Further checks are made for semantic correctness.

4.1 Syntax rules

Typically, the syntax of C is described using a production rule system (i.e. a grammar) derived from the Backus-Naur Form (BNF).

The rules are written as productions of the form :

```
        syntactic_term   :
                              definition
```

where a "definition" is a sequence of objects (symbols).

The grammar rules are presented in Appendix A13 (pages 234 - 239) of K & R.

The grammar in K & R has undefined terminal symbols (such as *identifier*) which are defined in other appendices (e.g., A2.3 on page 192).

Chapter 5

Data Types, Operators, and Expressions

5.1 Data Types (K & R Section 2.2, pp 36-37)

As mentioned previously there are only a few fundamental data types in C :

char	a single byte, capable of holding one character in the local character set.
int	an integer (16 or 32 bits, machine dependent).
float	single-precision floating point.
double	double-precision floating point.

There are other variations of these types. For example :

short int	(or just simply "**short**")	– usually 16 bits
long int	(or just simply "**long**")	– at least 32 bits

We may also use the "unsigned" qualifier :

e.g. **unsigned short int**
unsigned char (values between 0 and 255)

For example, **#define** Cardinal **unsigned int**

Extended-precision floating-point is set using "long double".

5.2 Constants (K & R Section 2.3, pp 37-39)

There are several kinds of constants – each has a data type :

integer constant is a sequence of digits which is decimal by default, or octal if it begins with zero (0), or hexadecimal if it begins with 0X (or 0x).

e.g. -0XB (B_{16}) , 077 (77_8).

Integer constants may be suffixed by "U" (or "u") to specify it is **unsigned**, or "L" (or "l") to specify a **long**. e.g. 123456789L

character constant is an integer written as one character enclosed in single quotes. e.g. 'x'

To represent nonprinting characters (e.g. "newline") we use the escape sequence (\) followed by one to three octal or hex digits giving the integer value to be assigned.

e.g. '`\n`' for newline, '`\xB`' for bell, '`\x30`' for zero (see page 193 of K & R).

Note that the character constant '`\0`' is in fact just the *numeric value* zero (0). However, remember that '0' (numeric value equal to 48_{10}) is the *character constant* for the digit 0.

We can also write a *constant expression* involving only constants (e.g. **#define** MAXSIZE 100). These are evaluated at compile-time only.

A *string constant* is a sequence of zero or more characters enclosed in double quotes. e.g. "hello". It is stored as an array of characters, with a '`\0`' at the end - to find its length you must scan it until you find the terminal value '`\0`'.

Note that the 'x' (character constant) is NOT the same as the "x" (string constant), since "x" consists of an array of two characters ('x' and '`\0`')

The C library contains various functions for handling strings – not a *forté* of C ! – such as copying strings, comparing strings, token searching etc.... These are contained in the header file "string.h".

floating constant e.g. 123.4 or 1.234e2 or 123.4F or 123.4L

(If not suffixed, they are stored as **double**. If they are suffixed by F (or f) they are **float**, or by L (or l) they are **long double**.

enumeration constant is a list of constant integer values (like the enumerated type in Pascal).

e.g. **enum** boolean {NO, YES};

It is basically an array with the initial index value (starting at 0 by default) stored in the array (i.e. NO = 0, YES = 1).

Another example :

```
enum   day   {sun, mon, tue, wed, thu, fri, sat}   d1, d2;
```

The name "day" is called the *e_tag*.

Here, "d1" and "d2" are variables that can take on as values only the elements of the set. Thus,

```
d1 = fri;   /*assigns the value "fri" to "d1"*/
```

Like **#define**, **enum** constants are evaluated at compile-time. They should be used to aid programme clarity. It is yet another way of providing data abstraction.

5.3 Declarations (K & R Section 2.4, page 40)

You must declare all variables before use, specifying its type and a list of one or more variables – as we saw before.

We may also initialise variable declarations. The initialiser is preceded by "=", and is either a constant expression or a list of initialisers nested in brackets "{" and "}".

For example :

```
char hello[ ] =        "hello";                     /*string*/
char bye[ ] =          { 'b', 'y', 'e' };           /*character array*/
char *bye_again =      "byebye";                    /*pointer to string*/
const double e =       2.718281828;                 /*read-only data*/
float days[3] =        { 1.5e1, 2.34e–1, 3.2 };     /*vector*/
int matrix[2][3] =     {                            /*matrix*/
                       { 1, 2, 3 },
                       { 4, 5, 6 }
                       };
```

```
struct key    {                                /*structure*/
              char *word;
              int count;
              } keytab[ ] = {
                        {"auto", 0},
                        {"break", 1},
                        {"case", 2}
                        };
```

5.4 Operators (K & R, pp 41–51)

5.4.1 Arithmetic Operators :

We have the normal binary arithmetic operators

+	−	*	/	% (modulus)

The modulus operator is the remainder after integer division. [Be careful : do not confuse * with the indirection operator !!!]

The binary operators are grouped from left to right. That is,

1 − 2 + 3 − 4 − 77   is equal to   (((1 − 2) + 3) − 4) − 77

For example, see the definition of a leap year on page 41 of K & R.

Binary arithmetic operators + and − have <u>lower</u> precedence than the *, / and % operators.

There are also unary arithmetic operators :

+   −   ++   −−

The ++ (autoincrement) and −− (autodecrement) operators can only be applied to variables (**int** and **float**) and NOT to constants or expressions. They are grouped from <u>right-to-left</u> in an expression, and can be prefix or postfix operators.

Example :

```
If n is 5, then x = n++; gives x = 5
If n is 5, then x = ++n; gives x = 6
If n is 5, then x = n--; gives x = 5
If n is 5, then x = --n; gives x = 4

If x is 3.3, then y = ++x; gives y = 4.3
```

Also,	---a	is equivalent to	-(-(-a))
	- --a	is equivalent to	-(--a))
	f ++ - g	is equivalent to	(f ++) - g

Study the "squeeze" function on page 47 of K & R.

The ++ and -- operators can give rise to side-effects (unintended results). For example,

```
a = ++c + c;
```

The problem is that there is no guarantee on the effect of "c" on the outcome. That is, is the second "c" value incremented before adding to the first (incremented) "c" ?

It is better to write two statements as :

```
++c;
a = c + c;
```

Unary operators ++ and -- have higher precedence than unary operators - and + which in turn have higher precedence over the binary operators.

5.4.2 Relational and Logical Operators :

The relational operators are :

```
>        >=        <        <=
```

which have lower precedence than arithmetic operators. That is,

i < lim - 1	is equivalent to	i < (lim – 1)
a < b < c	is parsed as	(a < b) < c

There are also the equality operators == and != which are used in the context of relational expressions. They are of lower precedence to the above relational operators. Beware that

```
if (i = 1) ...
```

is wrong !!

The logical operators are

!	&&	||
(unary)		(binary)

They yield either the integer value 0 or 1.

For example, 3 + !x evaluates to 3 if x is non-zero, and 4 if x is zero.

These are evaluated left-to-right and evaluation stops as soon as the truth or falsehood is known. See example on bottom of page 41 of K & R.

The precedence of && is higher than || (but lower than all unary, arithmetic and relational operators) – i.e. && is evaluated before ||. For example,

'A' <= c && c <= 'Z' || x   equals   (('A' <= c) && (c <= 'Z')) || x

5.4.3 Bitwise Operators (K & R pp 48–49)

Bit manipulation is possible :

&	|	^	<<	>>	~
(and)	(or)	(xor)	(shift left)	(shift right)	(1s compl)

For example,

```
x << 3;           /*shift x left by 3 bits, injecting 3 zeros at right*/
y = y & 0XFF;     /*mask lower byte*/
```

Be careful when shifting right, because you must have declared the variable to be **unsigned** to avoid the problem of injecting the sign bit into the left-hand side.

Exercise 2-7 of K & R : Write a set of expressions that results in the *n* bits that begin at position *p* of a variable "x" inverted.

```
        unsigned int x, y, mask;
        int p, n;

/*      generate mask n bits from position p */

        mask = ((~ ( ~0 << n)) << p);

/*      one's complement of x inside mask*/

        y = ( ~x ) & mask;
        y = (( x ) & (~mask)) & y;
```

Hands-on exercise : Type in the following code used for packing four characters into an **int** :

```
main () {
        char  a, b, c, d;    /* will need to initialise these, e.g. '\1' */
        int p;

        p = a;               /* promotion of char to int (see later) */
        p = (p << 8) | b;
        p = (p << 8) | c;
        p = (p << 8) | d;
/*      display p */
}
```

and for unpacking an int into four characters :

```
main() {
        char a, b, c, d;
        int p;

        d = p & 0xFF;
        c = (p & 0xFF00) >> 8;
        b = (p & 0xFF0000) >> 16;
        a = (p & 0xFF000000) >> 24;
}
```

5.4.4 Assignment Operators and Expressions (K & R pp. 50-51)

We can write an expression like

```
i = i + 2;
```

in a compressed form, as :

```
i += 2;
```

The += is an assignment operator. Others exist, in the form "operator="

```
+=   -=   *=   /=   %=   <<=   >>=   &=   ^=   |=
```

(N.B. : Don't confuse the equality operator "!=" with other assignment operators)

Note that	k *= 3 + x	is equivalent to	k = k * (3 + x)
		rather than	k = k * 3 + x

5.4.5 The Comma Operator (K & R, page 62)

The comma operator finds a use in the "for" statement.

A pair of expressions separated by a comma is evaluated left-to-right, and the type and value of the result are the type and value of the right operand.

For example : sum = 0, i = 1

(If "i" has been declared as "**int**", then this comma expression has value 1 and type "**int**").

e.g. If i, j, and k = 3 are type "**int**" then the expression

```
i = 1, j = 2, ++k + 1
```

evaluates to the integer value 5.

5.4.6 Conditional Expressions (K & R, pp 51-52)

We can write the "if-else" statement (see later)

```
if (expr1)
      expr2;
else
      expr3;
```

as a ternary expression :

```
x = expr1 ? expr2 : expr3;
```

i.e. If "expr1" is true (non-zero) then "expr2" is evaluated, and that is the value of the conditional expression ("x") ; otherwise "expr3" is evaluated, and that is the value of "x".

It is a way of writing succint code. For example :

```
z = (a > b) ? a : b;
```

Another example :

```
for (i = 0; i < n; i++)
      printf("%d%c", a[i], (i%10==9 || i==n–1) ? '\n' : ' ');
```

which prints "n" elements of an array, 10 per line, with each column separated by one blank, and each line terminated by a newline ("\n").

5.5 Type Conversions and Casts (K & R, pp 42-46)

An arithmetic expression such as "x + y" computes a value of a certain type – depending on the type of the variables x and y. If x and y are different types then x + y is a mixed expression. The operands are then automatically converted to a common type according to a set of rules.

This automatic conversion process goes under various names :

implicit conversion coercion promotion widening

In general, no loss of information occurs when converting a "narrower" operand into a "wider" one.

E.g. : converting an **int** to a **float** in an expression like f + i.

However, a narrowing or demotion can lose information. For example,

- assigning a **long int** to a **short int**
- assigning a **float** to an **int**

(they are not illegal - but may draw warnings !)

There is one subtle point about widening characters to integers.

The problem is that **char** does not specify whether variables of type **char** are signed or unsigned.

That is : If the **char** value is negative, then the higher-order bits can be sign-extended (filled with 1s here) or just padded with 0s. This is machine-dependent !

Moral of the story – specify "**signed char**" or "**unsigned char**" if non-character data is to be stored in **char** variables.

Arithmetic conversions for expressions are based on "promoting" the lower type operand to the higher type operand. The exact rules for arithmetic conversions are given in Section A6.5 of K & R (page 198).

Conversions can also take place across an assignment – the value of the right-hand side is converted to the type of the left-hand side. The same promotion and demotion rules apply. Demotion again involves the possible loss of information (e.g. truncation of the fractional part when assigning a **float** variable to an **int** variable).

In addition to the above implicit conversions, there are explicit conversions where the expression is coerced to a given type – called *casting*. For example,

k = (**int**) ((**int**) x + (**double**) i + j);

5.6 Precedence and Order of Evaluation (K & R, pp 52-54)

C has fully defined rules for forming expressions and how they are evaluated. These rules are precedence and associativity.

Expressions inside parentheses are evaluated first – parentheses can be used to clarify the order in which operations are performed.

Consider the expression

```
1 + 2 * 3
```

In C, the operator $*$ has higher precedence than $+$, so multiplication is performed first. i.e. it's equivalent to $1 + (2 \ * 3)$. You must use parentheses to perform $(1 + 2) * 3$. For example,

```
if ( x & mask == 0) ...
```

is interpreted as

```
if ( x & (mask == 0)) ...
```

since & has lower precedence than ==. You should make sure that you fully parenthesise to give proper results :

```
if ( (x & mask) == 0) ...
```

Look at the precedence and associativity of operators on page 53 of K&R.

Beware that these rules do NOT dictate the order in which parts of expressions are evaluated. Expressions involving one of the commutative operators (such as + , * , & , ^ , |) can be reordered at the convenience of the compiler even though they have been parenthesised in the program ! (The only exception to this are the "&&", "||", "?:", and "," operators which have a well-defined evaluation order).

For example :

```
x = f() + g();      /* () is the notation for a function */
```

can be summed by the compiler in some unspecified order (i.e. f is evaluated before g, or after) - this can be a problem if f and g mutually depend on a common variable. If you want to make sure that f() is evaluated before g(), you must write

```
temp = f();
x = temp + g();
```

Other nasties (side-effects) can arise if some variable is changed as a by-product of the evaluation of an expression – this is particularly true with arguments in function calls. For example :

```
printf("%d  %d\n", ++n, power(2, n));
```

produces different results depending on whether ++n is evaluated before or after "power(2, n)". Best to write :

```
printf("%d  %d\n", n, power(2, n));    ++n;
```

Moral – don't write code that depends on the order of evaluation.

Chapter 6

Flow of Control

We have already briefly covered the use of the "if-then-else" statement and loop constructs. Here, we look at them in more detail.

6.1 Statements and Blocks (K & R page 55)

We have seen that when an expression is followed by a ";", it becomes a statement. For example ;

```
x = 0;
i++;
```

As we saw previously, the semicolon indicates the termination of a statement (and not the separation of statements like in Pascal).

We can group various statements and declarations together to form a compound statement (sometimes called a block if the declarations come at the beginning of a block) by using braces "{" and "}".

There is no ";" after the "}" brace that ends a block.

For example : A compound statement :

```
{
    a = 7;
    b = a + 3;
    printf("\na + b = %d and a - b = %d", a + b, a - b);
}
```

or, as a <u>block</u> :

```
{
        int a, b;

        a = 7;
        b = a + 3;
        printf("\na + b = %d and a - b = %d", a + b, a - b);
}
```

Blocks can be nested and/or organised in parallel. For example :

```
{
        a = 1;
        b = 2;
        {
                c = 3;
                d = 4;
        }
}
```

The definition of blocks is important when considering scoping (see the topic of functions later).

6.2 If-elseif-else (K & R pp 55-58)

To represent a multiway decision we use the if-else if-else statement. Its syntax is :

```
if (expression_1)
        statement_1
else if (expression_2)
        statement_2
else if (expression_3)
        statement_3
...
else
        statement_n
```

The "else if" and "else" parts are optional. A statement can be a simple or compound statement.

The semantics are :

- The expressions are evaluated in order (top-down).
- If any expression is true, the statement associated with it is executed, and this terminates the whole chain.

Although the "else" case is optional, it should always be included (as a default "catch-the-impossible condition").

Look at the code for counting the number of occurrences of digits, blanks etc.. on page 22 of K & R.

6.3 The Switch (K & R pp 55-58)

This is a multi-way conditional statement generalising the if-else conditional statement.

Like the if-else statement, "switch" operates in a cascade fashion. Switch, however, tests whether the expression matches one of a number of constant integer values. (The if-else statement, remember, evaluates the truth or falsehood of the expression).

The syntax is :

```
switch (expression) {
      case constant_expr : statements
      case constant_expr : statements
      . . .
      default : statements
}
```

Each case is labeled by one or more integer-valued constants or constant expressions.

If a "case" matches the expression values, execution starts at that case. This continues on in a cascade manner until it "falls off the end". All case expressions must be different.

The "default" case is optional. If it isn't present no action is taken. It is used as a "catch-all" option.

To forcefully exit at any stage from within a case label (and thereby get out of the cascade effect), use the "**break**" statement. The "break" statement causes the immediate exit from within a case label.

Look at the switch code on page 59 of K & R.

Exercise : Try modifying the code to also count newlines. [insert a set of statements after "case '`\n`':"]

6.4 Loops (K & R pages 60-66)

6.4.1 The While and For Loops

Repetition is a common action undertaken in an algorithm. To perform repetition, some loop control mechanism are available in C.

The "**while**" loop :

```
while (expression)
      statement
```

[N.B. : There's no "do" token - like in other languages]

The expression is evaluated each time, before entry to each loop. If the expression evaluates to non-zero (i.e. TRUE) , "*statement*" is executed. If the expression is zero (i.e. FALSE), execution of the loop terminates.

Beware that the loop doesn't repeat *ad nauseam* (infinite loop) – you must make sure that the "*expression*" changes from outside the program (such as reading from a file until EOF or some other condition – see bottom of page 60 of K & R), or from within the loop (by changing the value of variable(s) used in the expression). For example :

```
int n;
...
while (-- n) {
      ...
}
```

This has the following problem :

What happens if "n" gets assigned to be < 0 before entering the "while" loop ?

It is best to rewrite it as :

```
int n;
...
while (--n > 0) {
      ...
}
```

The "for" loop :

> **for** (*expr*$_1$; *expr*$_2$; *expr*$_3$)
>   *statement*

which is equivalent to

> *expr*$_1$;
> **while** (*expr*$_2$) {
>   *statement*
>   *expr*$_3$;
> }

For example :

```
for ( i = 1 ; i <= n ; i++ )
      factorial *= i;
```

Note that the index "i" can be altered from within the loop. It also retains its value when the loop terminates.

All the "*expr*$_n$" are optional. In fact an infinite loop (not recommended in most circumstances) can be written as :

```
for ( ; ; ) {
      . . .
}
```

which is equivalent to :

```
while (1) {
      . . .
}
```

We often use the comma operator within "for" loops (see bottom of page 62 of K & R). Both "while" and "for" loops can be nested.

6.4.2 The Do-While Statement

We can also test the truth or falsehood of an expression after the loop – i.e. after each pass of the loop.

This is done using the "do-while" statement :

```
do
    statement
while ( expression );
```

This is like the “repeat-until” statement in Pascal.

The do-while loop is used much less than the while-loop (experience tells us that it is generally better to test a condition before doing something).

6.5 Forced Exiting from a Loop

It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom of the loop.

The “**break**” and “**continue**” statements provide us with a mechanism for doing such a thing.

A “break” causes the innermost enclosing loop to be exited immediately (similar to the situation in a “switch” statement).

“Break” can be used in “for”, “while” and “do” loops (as well as in “switch”).

For example :

A typical use of “break” is to terminate an otherwise infinite loop upon a given condition :

```
...
while (1) {
    scanf("%lf", &x);
    if (x < 0.0)
        break;        /* exit loop if value is negative */
    printf("\n%f", sqrt(x));
    ...
}
...                   /* "break" jumps to here */
```

The “continue” statement causes the current iteration of a loop to stop and causes the next iteration of the loop to begin immediately.

For example : The following code processes all characters except digits :

```
for (i = 0; i < TOTAL; ++i)  {
   c = getchar ();
   if ( '0' <= c && c <= '9' )
      continue;
   ...                       /* process other characters */
/* "continue" transfers control to here for next iteration */
}
```

The "continue" statement may only occur in "for", "while" and "do" loops. It does not apply to the "switch" statement (it can exist inside a "switch" inside a loop, but it just causes the next loop iteration).

Exercise : Rewrite the following pieces of code to avoid using "break" or "continue" :

Example Code 1 :

```
while  (c = getchar() )  {
   if (c == 'E')
      break;
   ++count;
   if (c >= '0' && c <= '9')
   ++digitCount;
}
```

Example Code 2 :

```
i = -5;
n = 50;
while (i < n)  {
   ++i;
   if (i == 0)
      continue;
   total += i;
   printf("\ni is %d and total is %d", i, total);
}
```

Chapter 7

Functions and Program Structure

People are unable to solve large problems without special problem-solving techniques.

Abstraction is one of these techniques used to manage the complexity of large programs.

It relies upon emphasizing important concepts, while hiding (i.e. abstracting away) unnecessary detail.

Top-down design using step-wise refinement is an example which relies upon control abstraction. It is based upon problem decomposition.

We can conveniently implement stepwise refinement using control abstraction – i.e. each sub-task is written as a separate routine.

For example, a recipe for cooking.

There are advantages to using routines :

- Easier to read and understand the overall algorithm
- Program is modular in design and implementation
- Hides away unnecessary details
- Sub-tasks can be re-used (reusability)
- Easier to debug, document and maintain

The C language includes just one type of routine – the function. (Pascal has both functions and procedures).

In fact a C program is just a collection of objects – which are either functions or variables.

A typical C program generally consists of many small functions ("main" is itself a function), which may reside in one or more source files.

Source files may be compiled separately and loaded together – along with previously compiled C library functions (such as "printf" etc...).

A C program is just a set of definitions and functions.

Communication between the functions is by arguments and values returned by the functions, and through external variables.

The functions can occur in any order in the source file, and the source program can be split into multiple files, so long as no function is split. Functions may NOT, however, be nested (unlike Pascal).

A function definition has a name and a parenthesis pair containing zero or more formal parameters and a body (block).

```
return_type function_name ( argument declarations )
{
        declarations and statements
}
```

For each parameter there should be a corresponding declaration that occurs before the body.

In ANSI C, each parameter declaration should include the type explicitly – such a declaration is called a function prototype.

The "return" statement is the mechanism for returning a value from the called function to its caller. Any expression can follow the "return" statement.

```
return (expression); /* parentheses are optional */
```

The "*expression*" will be converted to the type of the function (*return_type*) if necessary.

There is in fact no need to have an expression after "return" (if no value needs to be returned), or no need to have even "return" (flowing off the end of a function (by reaching the "}" brace) is equivalent to a return with no expression).

It is always good programming practice to return at least 0 if it is not a void function - as in “main” :

```
int  main()    /* int is optional here */
{
      int  done = 0;
      ...
      return done;
}
```

The simplest ANSI C function definition is :

```
void nothing (void) {
      return;      /* optional line */
 }
```

This has no arguments and returns no value to the calling program (i.e. for void functions “return” may optionally exist, but cannot be written with an expression, such as “return 0” – which is returning an integer value equal to 0).

(This can also be written as :

```
nothing () {
 }
```

but this is not recommended in ANSI C)

Functions may return an arithmetic type, void (also a “struct”ure, a “union”, or a pointer), but not a function or array.

An example function definition that returns a type “**int**” is

```
int max ( int a, int b, int c)
{
        int m;

        m = (a > b) ? a : b;      /* m is max of a and b */
        return (int) (( m > c) ? m : c);    /* max of c and m */
}
```

The variable “m” is called a local variable – it only exists when the function is active (i.e. “m” is created when the function is called, and is destroyed when return to the calling program is effected).

Here, local variables are called automatic variables because they are automatically created when the function is invoked – they are NOT permanent variables.

Next, the calling routine (say, "main ()", in this case) must know that "max" returns an "**int**" value. This is ensured by declaring "max" explicitly in the calling routine main :

```
#include <stdio.h>

main ()
{
        int max(int, int, int);     /* function declaration */
        int x, y, z, ans;
        ...

/*      function invoked where actual parameters are x, y and z */

        ans = max(x, y, z);
        ...
        return 0;
}
```

The function can also be declared before "main" as :

```
#include <stdio.h>
int max(int, int, int);     /* function declaration */

main ()
{
        int x, y, z, ans;
        ...

/*      function invoked where actual parameters are x, y and z */

        ans = max(x, y, z);
        ...
        return 0;
}
```

We will return to this when considering scope rules.

Functions are invoked by writing their name and an appropriate list of actual arguments within parentheses. These actual arguments will match in number and type with the formal parameters in the function definition.

All arguments are passed by "call-by-value" – i.e. a copy of the actual parameter is made and passed to the function. The actual parameters can therefore not be changed by the function.

It is possible to pass arguments using "call-by-reference" – by passing the address rather than the value. This does allow the function to modify the values of the actual parameters in the calling environment.

This is effected by using the "&" address unary operator (see later with pointers).

Explain why the following function will not work :

```
void exchange (double a, double b)
{
        double temp;

        temp = a;
        a = b;
        b = temp;
        return;
}
```

This requires call-by-reference by using pointers.

7.1 External (Global) Variables (K & R pp 73-79)

Passing arguments by means of a parameter list is not the only way of communicating data between functions.

It is possible to use "external (global) variables" which are globally accessible and accessed by name (just like variables in Fortran COMMON blocks and variables in the outer most block of Pascal programs).

Unlike automatic variables, global variables are permanent (during the execution of the program).

The use of external variables is frowned upon – but can be useful in situations where many functions have to share a large number of variables or structures.

Consider the reverse Polish calculator program on pages 76 and 77 of K & R. The essentials of the program are as follows :

```
#includes
#defines

/* function declarations for "main" */

main() {
...
}

/ * external variables for "push" and "pop" functions */

void push(double f) {
...
{
double pop(void) {
...
}

int getop(char s[ ]) {
...
}

/* routines called by "getop" */
```

Here the stack (the array "val") and its associated variables (stack pointer "sp") are declared external to both the "pop" and "push" routines because they are shared by "pop" and "push" and not used by "main".

Note that the external variables for "push" and "pop" are declared after "main" – since "main" does not refer to them. "main" does not "see" these variables. If "main" were to use them, they would have to be declared before "main" – or declare them as external (by using the "**extern**" qualifier – more of this shortly).

Also, the functions "getop", "push" and "pop" are declared before "main" as they are used by "main".

7.2 Scope Rules (K & R pp 80-82)

C is not a block-structured language in the sense of Pascal – because functions cannot be nested i.e. they cannot be defined within other functions.

On the other hand, variables can be defined in a block-structured fashion within a function. Variable declarations (including initialisations) may follow the left brace that introduces any compound statement.

The scope of a name is the part of the program within which the name can be used. We have already seen that the scope of local variables (such as automatic variables and function parameters) is the function in which the name of the variable is declared. We have also seen that external variables cannot be referred to before they have been declared textually - i.e. the scope of external variables is from the point of declaration to the end of the file being compiled. Their scope can be extended to the entire file (and, as we shall see, to other files) if they are qualified with the word "**extern**".

The basic rule of scoping is that identifiers are accessible *only within the block in which they are declared.* They are unknown outside the boundaries of that block. If an inner block redefines an identifier, the equivalent outer block identifier is hidden, or masked, from the inner block. All other variables used an the outer block are also visible in an inner block (if they haven't been redefined). For example :

```
{ /* block 1 */
  int   a = 5, b = −3, c = −7;          /* a  b  c  */

  printf("\n%d  %d  %d", a, b, c);       /* 5 −3 −7 */
  { /* block 2 */
    int    b = 8;
    float  c = 9.9;

    printf("\n%d  %d  %.1f", a, b, c); /* 5 8 9.9 */
    a = b;
    { /* block 3 */
      int   c;

      c = b;
      printf("\n%d  %d  %d", a, b, c); /* 8 8 8 * /
    }
    printf("\n%d  %d  %.1f", a, b, c);  /* 8 8 9.9 */
  }
  printf("\n%d  %d  %d", a, b, c);       /* 8 −3 −7 */
}
```

Try and find out what is printed out with the following program :

```
main()
{
        int   a = 1, b = 2, c = 3;

        ++a;
        c += ++b;
        printf("\nfirst :  %5d %5d %5d", a, b, c);
        {
           float   b = 2.0;
           int     c;

           c = b * 3;
           a += c;
           printf("\nsecond:  %5d %5.1f %5d", a, b, c);
        }
        printf("\nthird:  %5d %5d %5d\n\n", a, b, c);
}
```

The functions and external variables that make up a C program need not all be compiled at the same time ; the source text of the program may be kept in several files.

So the problem now arises of the scope of the different variables and functions in the different files.

It is important to distinguish between the declaration of an external variable and its definition. Using, say,

int sp;

outside of any function defines the external variable “sp”, declares “sp” for the rest of the file, and causes storage to be set aside for it.

However, the line

extern int sp;

declares “sp” for the rest of the source file, but it does not create the variable nor reserve storage for it. Its actual definition must be made in somewhere else (perhaps in another file).

There must only be one definition of an external variable among all the files that make up the source program – all other files that use that external variable must use the "**extern**" qualifier.

We can organise the shared variables and functions of a large program (which is divided into various files) by having one or more "header files" and use the **#include** directive in all files that use the functions and shared-variables (see K & R page 82).

7.3 Storage Classes (K & R pp 83-85)

Every variable has two attributes :

- type, and
- storage class

The storage class relates to the way memory storage is allocated for the variable.

There are four storage classes :

- **auto**(matic)
- **extern**
- **register**, and
- **static**

These words are used to qualify a variable (by prefixing) as, for example :

```
static int x;
```

7.3.1 Auto(matic) Storage Class

We have already seen the automatic storage class.

Automatic variables are created upon entry to a compound statement block and are destroyed upon exit.

If the block is re-entered, storage is once again allocated – but previous values (which were destroyed) are unknown.

Variables declared within function bodies are (by default) automatic.

7.3.2 Register Storage Class

Here, the programmer is requesting (if possible) to the compiler to allocate storage for a given variable in a high-speed register.

This is particularly useful for variables that are used often.

The compiler, of course, is limited to how many registers it may use (this depends on the CPU design) – so your request may be ignored.

See the example on page 84 of K & R.

Both register and automatic storage classes have undefined (i.e. garbage) initial values - you have to initialise them.

Also, you can initialise register and automatic variables by using i) constants ii) expressions involving previously defined values, and even iii) function calls. For example :

```
int binsearch (int x, int y[ ], int n)
{
      int low = 0;
      int high = n -1;
      int mid;
      . . .
}
```

7.3.3 Static Storage Class

Static declarations have two important uses :

- allows a local variable to retain its previous value when the block in which it resides is reentered. This is in contrast to automatic variables.
- provides a “privacy” mechanism very important for program modularity.

<u>The first use</u> :

Consider the following "random number generator" example :

```
#define MULTIPLIER    25173
#define MODULUS       65536
#define INCREMENT     13849
#define INITIAL_SEED  17

int random()
{
        static  int   seed = INITIAL_SEED;

        seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
        return  (int)  seed;
}
```

Here, the variable "seed" has a static storage class – it will retain its value upon exiting from the function – so upon reentry (next time the function is invoked) the value allocated to "seed" will be the same as it was when the function previously exited.

<u>Second use</u> :

The "static" declaration, applied to an external variable or function limits the scope of that object from the point of declaration to the rest of the source file being compiled (and only in that file).

You may ask – Isn't that like the case of external variables ?

The difference is that external variables can be made available in another file by using "extern". A static variable only has a scope of the file in which it's defined – the use of the same variable in another file refers to a different variable.

Static variables and functions (if the functions haven't been "**extern**"ed in another file) are therefore only visible <u>inside</u> the file in which they are declared. Files can therefore be used as private program modules.

Static variables are guaranteed to be initialised to zero.

7.4 Recursion (K & R pp 86-87)

One of the most important attribute in programming is the ability to perform recursion (self- and mutual-recursion).

C functions may be used self-recursively.

Study the following function for calculating n factorial ; that is,

$$n! = n(n-1)(n-2)\ldots 3.2.1 \qquad \text{for} \quad n > 0$$

```
int factorial (int n)
{
        if (n <=1)
        return (1);
        else
        return (int) (n * factorial(n-1));
}
```

Explain in detail what the following program does (try it !):

```
#include <stdio.h>

main ()
{
        void try_me (void);

        printf("\nWhat is your favourite line ?  ");
        try_me();
        printf("\n\n");
        return 0;
}

void try_me (void)
{
        int c;

        if ((c = getchar()) != '\n')
           try_me();
        putchar(c);
        return;
}
```

Chapter 8

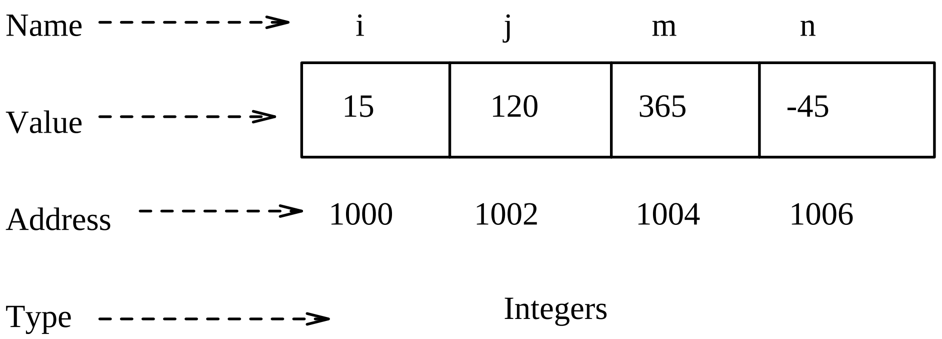
Pointers and Arrays

In this chapter, we introduce the basic concepts of pointers and arrays. The chapter is organised as follows:

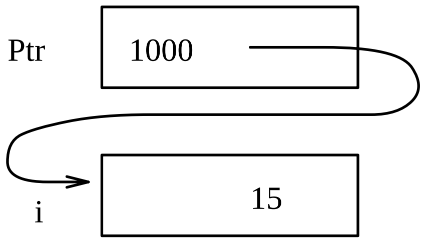
1. What is a Pointer ?
2. Declaring a Pointer Variable
3. Initializing a Pointer variable
4. Using a Pointer Variable
5. Pointers as Function Parameters
6. Relationship between Pointers and Arrays
7. Strings and Pointers
8. More on Pointers
    - Pointers to Functions
    - Pointers to Pointers
    - Pointers and Multi-dimensional Arrays
    - Array of Pointers
    - Complex Declarations

8.1 What is a Pointer ?

- A pointer is a memory address : an address of a variable. Its value indicates where a variable is stored, not what is stored.
- A variable has four attributes: Name, Type, Value, and Address. The address is stored as the value of a pointer variable.
- Address operator & : an unary operator which returns the address of its operand. The operand can be a variable of any type, but not a constant, a register, an array, or a function.



(1) &i : Address of variable i (= 1000)
(2) Value of i = 15
(3) Ptr = &i : stores 1000, which is the address of i, in the variable Ptr. Ptr is called a pointer variable. By storing an address, it points to an object.



8.2 Declaring a Pointer Variable

The following declaration declares a pointer variable `Ptr`:

```
                    type_T      *Ptr;
```

- A pointer variable is declared with $*$.
- A pointer variable must have a base type `T`: `Ptr` is a pointer to a variable of type `T`.
- Memory is allocated for the pointer variable `Ptr`, not for the variable it points to. A pointer consumes the same amount of storage as an **int** variable.

```
-- Examples:

int   *intPtr;     /* pointer to integer   */
float *floatPtr;   /* pointer to float     */
char  *charPtr;    /* pointer to character */
```

8.3 Initializing a Pointer Variable

- Pointers declared outside a function or declared as static inside a function are automatically initialized to zero.
- Pointers declared inside a function (automatic variables) are not initialized to any address. You must initialize such a pointer before use it.
- Pointers can be initialized to an expression involving the addresses of previously defined variables of appropriate type.

```
-- Examples:

float *floatPtr;   /* automatically initialised to 0 */
int   i, j[10];
int   *ip = &i;
int   *jp = &j[0] + 3;
int   *fptr = func();  /* func() returns a pointer to int */

int *func()
{ ... }
```

8.4 Using a Pointer Variable

8.4.1 Assignment

- Assigning addresses to pointers. Pointers and integers are not interchangeable, except zero. A constant integer value can be assigned to a pointer variable by using cast to make sure that the pointer must be the same type.
- Assign one pointer to another. The two pointers must be the same type. If you assign a pointer of one type to a pointer of another type by using cast, the address you are assigning will be rounded to boundary suitable for the data type pointed to, and the result might be an unexpected address.
- `NULL`: Defined as zero – a special pointer points to nowhere. `Null` can be used for any type and as a unique sentinel for terminating dynamic structures.

8.4.2 Dereferencing

- Accessing the underlying storage location through a pointer (get the value stored at the address pointed by the pointer)
- Dereferencing operator $*$: an unary operator which returns the value pointed by its operand. The operand must be a pointer variable.

```
-- Examples

int x = 1, y = 2, z[10],i;
int *ip;
float *fp;

ip = (int *) 1000; /* assign a constant int to ip */
ip = &z[0]; /* ip now points to z[0] */
ip = &x;    /* ip points to x */
fp = (float *) ip; /* assign different type ptr */

y = *ip;    /* y is now 1 */
*ip = 10;   /* x is now 10 */

-- Pointer increment

/* &, *, and ++ associate right to left. */
i = *ip++   is equivalent to {i=*ip; ip++}
```

```
i = *++ip   is equivalent to {ip++; i=*ip}
i = (*ip)++ is equivalent to {i=*ip; *ip=*ip+1}
i = {++*ip} is equivalent to {*ip=*ip+1; i=*ip}
```

8.4.3 Pointer Arithmetic

Pointer arithmetic is performed in terms of the underlying base type of the pointers. All the pointer manipulations automatically take into account the size of the object pointed to.

- Add integers to / sub integers from pointers.
- Pointer subtraction: the two pointers must point to elements of the same array.
- Relational comparisons of pointers: a pointer can be compared for ==, ! =, < , <=, >, or >= with another pointer or with zero, but the comparisons are portable only if the pointers access the same array (There is one exception: the address of the first element past the end of an array can be used in pointer arithmetic)
- No pointer addition, multiplication, division, or arithmetic involving real numbers.

Given the following declarations:

```
int   *ip1, *ip2;
float *fp;
char  *cp;
```

we can interpret the following expressions

```
ip + 3          -- 3 integers after the integer pointed by ip
                   (6 bytes increment)
cp + 3          -- 3 chars after the char pointed by cp
                   (3 bytes increment)
ip2-ip1+1       -- number of ints between ip1 and ip2 incl.
(ip1+ip2)/2     -- illegal
ip1+(ip2-ip1)/2 -- OK
```

8.4.4 Where Pointers are used ?

Pointers can be used in the following cases:

- Parameter passing in functions
- Using pointers to access arrays
- Dynamic storage allocation and linked data structures

HANDS-ON EXERCISE:

```
#include <stdio.h>

int main (void)
{
 int count = 10, x;
 int *int_pointer;  /* defines a pointer to int */

 printf ("int_pointer contains nothing, NULL = %i\n", int_pointer);

 int_pointer = &count;   /* assign addr of count to int_pointer */
 printf ("Now int_pointer contains the address of count = %i\n",
          int_pointer);

 x = *int_pointer;  /* assign the var pointed by int_pointer to x */

 printf("count = %i, x = %i\n", count, x); /* count and x contains
                                               the same value 10 */

 *int_pointer = 20; /* change the value of the var pointer by
                                                int_pointer */
 printf("count = %i, x = %i\n", count, x); /* count is changed but
                                       x remains the same value 10 */
 return (0);
}
```

8.5 Pointers as Function Parameters

There are primarily two ways in which parameters can be passed to subprograms :

1. *Call-by-Value* : In C, all parameters are passed "call by value", that is, each real parameter is evaluated and its value is used locally in place of the formal parameter. If a variable is passed to a function, the stored value of that variable will not be changed in the calling environment. Thus, in C, there is no direct way for the called function to alter a variable in the calling function.

2. *Call-by-reference* : the address of the real parameter is passed to the function ; the function then can be made to change the value of the addressed variable. This method is used when you want a function to return more than one output values (C's return-value mechanism permits a function to return only a single value).

```
-- Example of parameter passing

swap(int a, int b)
{
  int temp;

  temp = a;
  a    = b;
  b    = temp;
}

Suppose we have two integer variables i and j and we make
a call: swap(i,j). Now after return from swap(i,j), i and
j remain unchanged.

swap_1(int *a, int *b)  /* pointer version */
{
  int temp;

  temp = *a;
  *a   = *b;
  *b   = temp;
}

Now after swap_1(i,j) is called, i and j will be exchanged.
```

Call by reference can be accomplished in the following way:

- Declare a function formal parameter to be a pointer
- Using the dereferenced pointer in the function body to access the parameter

- Passing in an address as a real parameter when the function is called.

8.6 Relationship Between Pointers and Arrays

Pointers in C are intimately associated with arrays.

- The array name is associated with a pointer to the first element of the array: an array name is a pointer
- When an array is passed to a function, instead of the entire array, only the first location in the array is passed. So the array is not copied and changes to the array in the called function affect the array in the calling function: an array is passed call by reference.
- An array can be accessed through a pointer: It is faster since it uses a single machine instruction. In fact, the compiler translates all array subscripting into pointer dereferences.
- It is not necessary to declare the size of one–dimension arrays passed to functions, since it is the address of the first element that is copied to the function. The compiler knows how much storage to allocate — the amount necessary to hold a single pointer. In the calling function the compiler actually allocated sufficient storage to hold the entire array.
- It is possible to pass part of an array to a function: by passing a pointer to the beginning of the sub-array.
- Finally, it worth noting that a pointer is a variable but an array name is not a variable. An array name is a constant so that it cannot be assigned a value.

```
-- Example: Pointers and arrays

    int a[100];
    int *aptr = a;  /* aptr points to &a[0] */

Now:
    aptr <=> a <=> &a[0]
    aptr+1 <=> &a[1] <=> a+1
    aptr+n <=> &a[n] <=> a+n <=> &a[0]+n
    *aptr <=> a[0]
    *(aptr+n) <=> a[n]
```

```
        a = aptr;   /* Illegal */
        a++;        /* Illegal */
        &aptr = a;  /* Illegal */

    -- Example: Pointer operations and arrays

    1. Print an array using a pointer.
        int a[100];
        int *aptr;

        for (aptr = a; aptr <= &a[99]; aptr++)
             printf("%d\n", *aptr);

    2. Print an array in reverse order using a pointer.
        int a[100];
        int *aptr;

        for (aptr = &a[99]; aptr >= a; aptr--)
          printf("%d\n", *aptr);
```

HANDS-ON EXERCISE:

```
#include <stdio.h>

int main (void)
{
 int i1=10, i2;
 int a[10]={1,2,3,4,5,6,7,8,9,0};
 int *p1=&i1, *p2;  /* defines two pointers to int */
                    /* p1 is initialized to point to i */


 i2 = *p1 / 2 + 10; /* the value referenced by p1 can be used in
                       arithmetic expression */

 /* insert a printf here to print out values of i2 */

 p2 = a; /* p2 points to the array a */

 /* insert a printf here to print out values of p2, a and &a[0] */
 /* insert a printf here to print out values of *p2, *a and a[0] */
 p1 = &a[9];       /* p1 points to the last element of a */
 /* insert a printf here to print out number of elements between
    p2 and p1, using pointer arithmetic */

 /* using a while loop to print out elements in a, using pointer */

 while (p2 <= p1) {printf ("%i\n", *p2); p2++;}

 return(0);
}
```

8.7 Strings and Pointers

- A string is an array of characters.
- Constant strings:
  - Declared between double quotation marks
  - Always terminated with an extra character, the null character '
    0'.
- When we build a string ourself (eg. by reading from terminal), we have to supply the terminating null character. This is because that C has a library of useful string functions, all of which expect a string to be terminated with '
  0' (including printf() using %s)
- String assignment:
- String functions from standard library: C does not provide operators that work on an entire string directly, instead it provides a set of build-in functions in the standard library. These functions are declared in the system include file `string.h` which must be included in your program if you want to use them.
  - strcmp (**const char** *s1, **char** *s2) /* comparison */
  - strcat (**char** *s1, **char** *s2) /* concatenation */
  - strcpy (**char** *s1, **char** *s2) /* copy */
  - strlen (**char** *s1, **char** *s2) /* length */

```
-- Examples: Strings and pointers

(1)
```

```
      char line[80];
      char sentence[];
      char *s = "Hello there"; /* 12 chars, including /0 */
      (A string constant is interpreted as a pointer to the
       first char of the string)

   (2)
      while ((line[i] = getch()) != '\n') i++;
      line[++i] = '\0';  /* insert the terminating null character */

   (3)
      line = "illegal";  /* line is a constant pointer */
      line[5] = 'a';
      s = line;
      s[5] = 'b';
      s = "ok";
      *s = "illegal";    /* type mismatch */
```

8.8 More on Pointers

8.8.1 Pointers to Functions

- In C, a function itself is not a variable. How can we assign functions as values to variables, how can we pass functions as parameters to functions, and how can a function be returned as a value of another function: Using pointers to functions.
- Pointers to functions are especially useful when we write a function to evaluate different functions.
- A pointer to a function can be thought of as address of the code executed when the function is called.
- Declaring a pointer to a function:

```
  type_T  (* fun_ptr)();
  fun_ptr is a pointer to a function returning a value of type T.
```

- To obtain a pointer to a function, use the function name without following it with a parenthesized parameter list.

```
          double  (*fun_ptr)(),  exp();
          fun_ptr = exp;
```

- To call the function, dereference the pointer, followed by the parameter list:

```
(* fun_ptr)(10.0, power)  <=> exp(10.0, power)
```

- Before you pass a pointer to a function as a parameter, the compiler must know the type of the function's return value.

8.8.2 Pointers to Pointers

- When a pointer is passed to a function, actually what is passed is an address of the location currently pointed by the pointer.
- If you use the pointer to access and modify the location within the function, the content of the location is changed. However, the pointer itself passed to the function remains unchanged outside the function.
- In order for the pointer itself to be modified, you need to pass a pointer to that pointer.

8.8.3 Array of Pointers

- Pointers can be stored as elements in an array. All the pointers in an array have the same base type.
- An array of pointers can be declared in the following way:

```
type_T  *ptr_array[100];
```

  `ptr_array` is an array of 10 elements, each element of which is a pointer to type T

  - `ptr_array[1]` is a pointer to type _T
  - ∗`ptr_array[1]` is a value of type T

```
-- Example: Array of pointers

   char *days[] =
        { "Sunday", "Monday", "Tuesday", "Wednesday",
          "Thursday", "Friday", "Saturday"};
```

The array `days` is defined to contain seven entries, each a pointer to character string. We could display the name of the third weekday, for example, with the following statement:

```
printf ("%s\n", days[3]);}
```

HANDS-ON EXERCISE:

```
/* print the names and days of the months */

char *months[] =
    {
    "January", "February", "March",
    "April", "May", "June",
    "July", "August", "September",
    "October", "November", "December"
    };

int days[] =
    {
    31, 28, 31, 30, 31, 30,
    31, 31, 30, 31, 30, 31
    };


#include <stdio.h>

int main(void)
{
    int i;

    for (i = 0; i < 12; i++)
        printf("%s, %i\n", months[i], days[i] );

    return(0);
}
```

8.8.4 Complex Declarations

- Rules for deciphering a declaration:
  - The array operator (`[]`)and function operator(`()`) have a higher precedence than the pointer operator ($*$)
  - The array and function operators group from left to right, whereas the pointer operator groups from the right to left
- Specifying types in casts: if you want to cast a variable to a particular data type, you need to specify the data type which may be complicated.

```
-- Examples: Complex declarations

char  **argv; /*pointer to pointer to char */
int   (*daytab)[13]  /* pointer to array [13] of int */
int   *daytab[13]    /* array[13] of pointers to int */
int   *comp()        /* function returning pointer to int */
void  (*comp)()      /* pointer to function returning void */
char  (*(*x())[])()  /* function returning pointer to array
                        of pointers to function returning char */
char  (*(*x[3])())[5] /* array[3] of pointers to function
                        returning pointer to array[5] of char */
```

8.8.5 Command Line Arguments

- `main()` is a function that takes arguments. When you run a program, the host environment is responsible to supply two argument.

```
% int argc     -- argument count
% char *argv[] -- pointer to argument vector
                (array of pointers or a pointer to pointer to char)
```

- `argc` is larger than one because `argv[0]` is always the program name.
- All the arguments are string constants. If the arguments are intended to represent numerical data, you must explicitly convert them by using runtime library functions such as `atoi()` , `atof()`, etc.
- In `main()`, use `argc` to check correct number of arguments and then to use `argv` to access these arguments.

```
  /* Echo the command line arguments */

int main (int argc, char **argv)
    {
      if (argc < n+1)  /* n is the desired number of arg */
      {
        printf("Wrong number of arguments \n");
        exit(1);
      }
      while (--argc > 0)
        printf("%s", *++argv);

      retunr(0);
    }
```

8.8.6 Two-dimensional Arrays

- Declared by enclosing the bounds of each dimension separately in brackets.
- Index scheme: row–dominant. The elements of row zero are stored consecutively in memory; then the elements of row one and so on. The compiler calculates the element's position from the two subscripts and the number of columns in each row.

```
&array[i][j] = &array[0][0] + i * Col_num * sizeof(int) + j
```

- So when we pass a two–dimensional array to a function, the second dimension is needed but the first is not.
- As in one–dimensional array, whenever an element in a 2–dimensional array is accessed, the compiler converts the array access into equivalent pointer expression.

```
array[i][j]  <=> *(*(array+i)+j)
```

Let's look at some examples: Suppose we have a two dimensional array. `int is[3][4]`. This is really an array of three elements, where each element is an array of four integers. Abstractly, this array would look like this

ia[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]
[2][0]	[2][1]	[2][2]	[2][3]

The first thing to understand is exactly what the name `ia` represents here. Since it is an array, `ia` is the base address, and it points to the zeroth element of the array. In this case, `ia` is not a pointer to an `int`, but rather a pointer to an array of 4 `ints`. This is important when pointer arithmetic is used. let's look at some expressions.

```
ia -- Address of zeroth array element, same as &ia[0].
ia[1] -- First element of array. Abstractly, this is referring to
         row 1 of the two dimensional array above. Since ia[1] is
         itself an array, it represents the base address of this
         array. That is, ia[1] is the same as &ia[1][0], or ia+1
*ia -- Since ia is the address of the zeroth array element, this
       is the zeroth element. That is, this is the zeroth row
```

```
          of the two dimensional array.
    (*ia)[2] -- This is the element ia[0][2] (same as *(*ia+2))
    ia+2 -- This is the address of the second element of ia. That is,
            the address of the second row of ia.
    *(ia+2) -- This is the second row of ia.
    *(ia+1)[2] -- This is element ia[2][2] (same as *(*(ia+2)+2))
```

Suppose we have a two dimensional array representing test scores for students. `Max_students` and `Max_tests` are the maximum number of students and tests, respectively. The following code prints the scores using conventional array subscripting.

```
void print_score(scores)
  int scores[][Max_tests];
  {
    int student, test;
    for (student = 0; student < Max_students; student++)
      {
        for (test = 0; test < Max_tests; test++)
          printf("%d", scores[student][test]);
        putchar('\n');
      }
  }
```

We can use pointer to print the two dimensional array:

```
void print_score(scores)
  int scores[][Max_tests];
  {
    int test, (*row_ptr)[Max_tests];  /* pointer to an array */
    for (; row_ptr < scores[Max_students]; row_ptr++)
      {
        for (test = 0; test < Max_tests; test++)
          printf("%d", *row_ptr[test]);
        putchar('\n');
      }
  }
```

Chapter 9

Structures

In this chapter, we introduce the basic concepts of the structure construct in C. The chapter is organised as follows:

- Defining a New Data Type (Typedef)
- What Is a Structure ?
- Declaring a Structure variable
- Initializing a Structure Variable
- Using a Structure Variable
- Structures as Function Parameters
- Array of Structures
- Nested Structures
- Linked Data Structures

9.1 Defining a New Data Type

- C allows programmers to create their own names for data types with the **typedef** keyword.
- (2) Syntactically, a `typedef` is exactly like a variable declaration except that the declaration is preceded by the `typedef` keyword. Semantically, the variable name becomes a synonym for the data type rather than a variable that has memory allocated for it.

- A `typedef` can appear outside any function and data types created with `typedef` are in effect from the typdef until the end of the file.
- A `typedef` definition must appear before it is used in a declaration.
- `Typedef` does not create brand new data types. It is useful for providing a shorthand for complex type, eg., structures and for abstracting global types that can be used throughout a program.

```
 -- Examples: Typedef

 (1)  typedef  int   *ptr_to_int;
      ptr_to_int     iptr1, iptr2;

 (2)  typedef  int   stack[2][3];
      stack          stk1 = {1,2,3,4,5,6}, *sptr=&stk1;

 (3)  typedef enum {true, false} BOOLEAN;
      BOOLEAN   flag;
```

9.2 What is a Structure

- A structure is used to keep different pieces of information together as a single data record
- A structure is like an array except that each element can have a different data type, and the elements in a structure have names instead of subscript values.

9.3 Declaring a Structure variable

- To declare a structure, use the keyword **struct** followed by an optional name and declarations of each field. Field declarations are enclosed in braces.
- If we give the structure a name, we can declare variables to be the structure type by using the name.
- If we want to declare a single structure type to be used in one place only, you can declare it without using a name.
- We can declare a structure type and variables of the type together, by giving the variable names following the declaration.

- We can also use `typedef` to define a name for the structure type and then use the name to define variables of that type.
- All fields are stored consecutively in the order they are declared. Contiguity, however, is not required.

```
-- Examples: Declaring a structure

(1)  struct P_Record /* creates a template only - defines type */
      {
        char  name[20];
        short age;
      }

     struct P_Record  My_record, Pr_array[100];
     /* Now, allocates memory space for My_record and Pr_array */

(2)  struct
      {
        char   name[20];
        short  age;
      } pr;

(3)  struct
      {
        char  name[20];
        short age;
      } My_record, Pr_Array[100], *Pr_ptr;

(4)  typedef struct
      {
        char  name[20];
        short age;
      } P_Record;

     P_Record Pr_array[100], My_record;

(5)  typedef struct P_R
      {
        char  name[20];
        short age;
      } P_Record;      /* Now, struct P_R <=> P_Record */
```

9.4 Initializing a Structure Variable

- To initialize a structure, follow the structure variable name with an equal sign, followed by a list of initializers enclosed in braces.
- Only structure variables can be initialized. You cannot initialize a template since no storage is allocated for it.

```
-- Example: Initialize a structure

struct P_Record My_record =
                { "Jiannong Cao",
                  32
                }
```

9.5 Using a Structure Variable

- **Referencing Structure Fields**: We need a way to access the fields of a structure. There are two methods, depending on whether we have the structure itself or a pointer to structure.
  - The structure itself: use the field selection operator '.'. To select a field we enter the structure variable name and the field name separated by '.'.
  - A pointer to the structure: use the right arrow operator '$->$', a dash followed by a right bracket. Actually, $->$ is a shorthand for dereferencing the pointer and then using the dot operator:

    ```
    ptr -> field_name   <=>   (*ptr).field_name
    ```

  - A field of a structure can be any type a variable can be. The field names cannot be identical, but a field name can be the same as an existing variable name, including the structure variable name containing the field.
- **Assigning structures**: We can assign a structure to another structure, provided they share the same structure type.
- **Comparing structures**: We cannot compare two structure variables.
- **Input/output of structures**: When you use `scanf()/printf()`, you must read/write a structure field by field. You can use `fread/()/fwrite()` to read/ write an entire structure.

```
-- Examples: Using structures

(1)  name_len = strlen(My_record.name);
     Pr_array[10].age = 20;
     pr_ptr->age = 20;   /* equivalent to (*pr_ptr).age = 20 */

(2)  struct ss {
               int  a;
               float  b;
            } s1, s2, s3[3], a, f(), *p;
     FILE *fp;
     int nread;

     s1 = s2;
     s2 = f();
     a = *p;
     p = &s1;

     scanf("%d", &s1.a);
     scanf("%d", &a.a);
     scanf("%s", My_record.name);

     fp = fopen("RE", "r");  /* RE is the file name *?
     nread = fread(s3, sizeof(struct ss), 3, fp);
     fwrite(s3, sizeof(s3), nread, stdout);
```

HANDS-ON EXERCISE

```
#include <stdio.h>

main()
{
  struct Date
   {
     int month;
     int day;
     int year;
   };

  struct Date today;
  today.month = 11;
  today.day   = 28;
  today.year  = 1991;

  printf ("Today's date is %i/%i/%i.\n",
           today.day, today.month, today.year);
}
```

HANDS-ON EXERCISE

```
#include <stdio.h>

main()
{
  typedef struct Date  /* define DateType as a type of this struct */
   {
     int month;
     int day;
     int year;
   } DateType ;

  DateType today = {11, 28, 1991};  /* define and initialize today */

  struct Date *date_pointer = &today;  /*define a pointer to struct
                                                             Date */


  printf ("Today's date is %i/%i/%i.\n",
           today.day, today.month, today.year);

  date_pointer -> day = 29;
  printf ("Tomorrow's date is %i/%i/%i.\n",
           today.day, today.month, today.year);
}
```

9.6 Structures as Function Parameters

- The entire structure is copied to the function, i.e., a structure is passed by value.
- A pointer to a structure can be passed to achieve call-by-reference.

- Individual field of a structure can be passed as a real parameter to a function.
- It is possible for a function to return a structure or a pointer to a structure. If a structure is returned, it must be external or static. Otherwise it will disappear once the function returns.

```
 -- Examples:

(1) Passing an entire structure

     void func(P_Record Pr);
      { ... }

     P_Record     My_record;
     func(My_record);  /* changes to Pr within func() are not
                          reflected in the actual parameter */

(2) Passing a pointer to a structure

     void func(P_Record *Pr)
      { ... }

     P_Record My_record;
     func(&My_record);  /* Passing the address of a struct is
                          faster, changes to Pr are reflected
                                              in My_record */
```

9.7 Array of Structures

- An array of structure is declared by preceding the array name with the structure type name:

```
          struct   str_type_name   str_array_name[ ]
```

- How to select a field? : First use the usual array accessing method to reach individual structures and then use the dot operator to reach fields.
- An arrays of structures can be initialized at declaration time by initializing each structure in the array.

```
    -- Examples
```

```
        struct P_Record Pr_array[] =
                        {
                          {"Tom", 20},
                          {"Smith", 45}
                        }
```

HANDS-ON EXERCISE

```
/* print the names and days of the months */
struct
{
    char *month;
    int day;
} md[] =
    {
        { "January",    31 },
        { "February",   28 },
        { "March",      31 },
        { "April",      30 },
        { "May",        31 },
        { "June",       30 },
        { "July",       31 },
        { "August",     31 },
        { "September",  30 },
        { "October",    31 },
        { "November",   30 },
        { "December",   31 }
    };


#include <stdio.h>

main()
{
    int i;

    for (i = 0; i < 12; i++)
        printf("%s, %i\n", md[i].month, md[i].day );
}
```

9.8 Nested Structures

- When one of the fields of a structure is itself a structure, it is called a nested structure.
- Declaring a nested structure
  - declare a structure directly inside a structure
  - use typedef
- Initializing a nested structure: enclose each structure in braces, like initializing a multi-dimensional array.
- Accessing fields of inner structures: use multiple dot operators or combination of $->$ and '.'.

```
-- Examples:

struct P_Record
       {
         char  name[20];
         struct
             {
               short day, month, year;
             } birthday;
       } Pr, *p;

Pr.birthday.day = 12;
p -> birthday.day = 12;
```

9.9 Linked Data Structures

9.9.1 Dynamic Memory Allocation

- Memory are automatically allocated by compiler for a variable only if the compiler knows how much memory is required ahead of time. However, frequently the amount of memory needed by a program depends on the input.
- We can pick a maximum value for the input but this has two problems. First: there may be future time when you want to exceed this limit; Second: the higher the maximum, the more memory is wasted.

- A solution is to use runtime library functions to allocate the memory on the fly.
  - bytes in memory ; returns a pointer to the beginning of the allocated block. The argument to `malloc()` is the size in bytes of the block of memory to be allocated.
  - `free()` : Frees up memory that was previously allocated with `malloc()`. It takes a pointer to a contiguous block of storage and returns the storage to the storage pool.
  - `sizeof()` : returns the size in bytes of a variable of any type.
- If you declare a pointer to a structured data type, like an array or a structure, you must first allocate memory for the structured variable before you access it through the pointer.

```
-- Example: Dynamic memory allocation

(1)  int main (void)
         {
          int list[100], j, n;

          printf("How many values to be read?\n");
          scanf("%d", &n);
          for (j = 0; j < n; j++)
            scanf("%d", &list[j]);
         }

(2)  int main (void)
         {
          int *list, j, n;

          printf("How many values to be read?\n");
          scanf("%d", &n);
          list = (int *) malloc(n*sizeof(int));
          for (j = 0; j < n; j++)
            scanf("%d", list+j);
         }
```

9.9.2 Self-referential structures

- A structure may not contain instances of itself, but it may contain pointers to instances of itself.

- So we can declare pointers to structures that have not yet been declared. This permits us to create self-referential structures, as well as mutual-referential structures.
- However, forward references are not permitted within typedef

9.9.3 Linked lists

Together with storage allocation, using self-referential structures, we can create a linked list.

A linked list is a chain of structures that are linked one to another. In the simplest scheme, each structure contains an extra field which is a pointer to the next structure in the list.

In a typical linked-list application, you need to perform the following operations:

- create a list element (node)
- find a value in the list
- insert a value into the list
- delete a value from the list
- print a list

HANDS-ON EXERCISE

```
#include <stdio.h>

main ()
 {
   struct Entry  /* define a node in the linked list */
     {
       int            value;
       struct Entry   *next;
     };

   struct Entry  n1, n2, n3;
   struct Entry  *list_pointer = &n1;

   n1.value = 100;
   n1.next = &n2;  /* n1 is linked to n2 */

   n2.value = 200;
   n2.next = &n3;  /* n2 is linked to n3 */

   n3.value = 300;
   n3.next = NULL; /* n3 terminates the linked list */

   while (list_pointer != NULL)
    {
     printf (" %i\n", list_pointer -> value);
     list_pointer = list_pointer -> next;
    }
 }
```

Making a Node :

a) allocate memory for the node

b) assign the value to the node

c) return a pointer to the node

Finding a Value :

a) sequential search

b) binary search

Delete a value from a list :

a) find the node in the list with the value

b) remove the node from the list

c) free the memory occupied by the node

Insert a value into a list :

a) find the place where the node goes

b) make a node

c) link to node into the list

Print a list :

a) traverse the list and print the data in each node

b) until the list is NULL

Chapter 10

The Standard Library

The ANSI standard defines a set of standard library routines which perform frequently required tasks such as input/output, string manipulation, math functions, etc....

These routines are defined in the following standard header files :

assert.h	float.h	math.h	stdarg.h	stdlib.h
ctype.h	limits.h	setjmp.h	stddef.h	string.h
errno.h	locale.h	signal.h	stdio.h	time.h

If you use any of the routines defined in these files you should include that file at the top of your source file.

For example : If your program uses "printf", which is defined in "stdio.h" you would have the line :

#include <stdio.h>

Note the angle brackets surrounding the header file name. This indicates that the file may be found in the standard directory for headers. On UNIX this is typically /usr/include. If you include a file which is not in the standard include directory, it is necessary to surround the file name by double quotes.

For example : To include your own definitions from the file mydefs.h in the current directory you would put the following line in your source code :

#include "mydefs.h"

About 30% of all library routines are related to input/output.

A full description of the routines provided in the standard libraries is given in Appendix B of K&R. These notes will provide a summary of the routines available and demonstrate their use by means of examples.

10.1 The Libraries

10.1.1 Character Class Tests <ctype.h>

This library provides routines for classifying or inverting the case of characters.

Function definitions :

```
int     isalnum(int c)
int     isalpha(int c)
int     iscntrl(int c)
int     isdigit(int c)
int     isgraph(int c)
int     islower(int c)
int     isprint(int c)
int     ispunct(int c)
int     isspace(int c)
int     isupper(int c)
int     isxdigit(int c)

int     tolower(int c)
int     toupper(int c)
```

10.1.2 Input/Output <stdio.h>

The input/output library manages streams. A stream is a source/destination of data, and two types, binary and text, are supported. A text stream is a sequence of lines terminated by newlines. A binary stream is a sequence of bytes. By opening/closing a stream, we associate/disassociate that stream with a file.

Function Definitions :

```
FILE    *fopen(const char *filename, const char *mode)
FILE    *freopen(const char *filename, const char *mode, FILE *stream)
int     fflush(FILE *stream)
int     fclose(FILE *stream)
int      remove(const char *filename)
int      rename(const char *oldname, const char *newname)
FILE     *tmpfile(void)
char     *tmpnam(char s[L_tmpnam])
int      setvbuf(FILE *stream, char *buf, int mode, size_t size)
void     setbuf(FILE *stream, char *buf)

int      fprintf(FILE *stream, const char *format, ...)
int      printf(const char *format, ...)
int      sprintf(char *s, const char *format, ...)
int      vprintf(const char *format, va_list arg)
int      vfprintf(FILE *stream, const char *format, va_list arg)
int      vsprintf(char *s, const char *format, va_list arg)
int      fscanf(FILE *stream, const char *format, ...)
int      scanf(const char *format, ...)
int      sscanf(char *s, const char *format, ...)

int      fgetc(FILE *stream)
char     *fgets(char *s, int n, FILE *stream)
int      fputc(int c, FILE *stream)
int      fputs(const char *s, FILE *stream)
int      getc(FILE *stream)
int      getchar(void)
char     *gets(char *s)
int      putc(int c, FILE *stream)
int      putchar(int c)
int      puts(const char *s)
int      ungetc(int c, FILE *stream)

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

int      fseek(FILE *stream, long offset, int origin)
long     ftell(FILE *stream)
void     rewind(FILE *stream)
int      fgetpos(FILE *stream, fpos_t *ptr)
int      fsetpos(FILE *stream, const fpos_t *ptr)

void     clearerr(FILE *stream)
int      feof(FILE *stream)
int      ferror(FILE *stream)
void     perror(const char *s)
```

10.1.3 Common Definitions <stddef.h>

This header contains some commons definitions such as the **size_t** type and NULL.

10.1.4 String Manipulation <string.h>

These routines manipulate blocks of **char**. A string is a sequence of "char"s terminated with a `\0`. The parameters below are defined as :

char	*s, *t
const char	*cs, *ct
size_t	n
int	c

Function definitions :

char	*strcpy(s,ct)
char	*strncpy(s,ct,n)
char	*strcat(s,ct)
char	*strncat(s,ct,n)
int	strcmp(s,ct)
int	strncmp(s,ct,n)
char	*strchr(cs,c)
char	*strrchr(cs,c)
size_t	strspn(cs,ct)
size_t	strcspn(cs,ct)
char	*strpbrk(cs,ct)
char	*strstr(cs,ct)
size_t	strlen(cs)
char	*strerror(n)
char	*strtok(s,ct)
void	*memcpy(s,ct,n)
void	*memmove(s,ct,n)
int	memcmp(cs,ct,n)
void	*memchr(cs,c,n)
void	*memset(s,c,n)

10.1.5 Mathematical Routines <math.h>

This group of routines define mathematical functions. All the functions return double. Parameters x and y are **double**, n is **int**, angles are expressed in radians.

Function definitions :

sin(x)	sinh(x)	sqrt(x)
cos(x)	cosh(x)	ceil(x)
tan(x)	tanh(x)	floor(x)
asin(x)	exp(x)	fabs(x)
acos(x)	log(x)	ldexp(x,n)
atan(x)	log10(x)	frexp(x, **int** *exp)
atan2(y,x)	pow(x,y)	modf(x, **double** *ip)
		fmod(x,y)

10.1.6 Diagnostic Routines <assert.h>

The "assert.h" header file defines the macro :

void assert(**int** expr)

which can be helpful in debugging programs. If the expression evaluates to zero when assert is called, then an appropriate diagnostic message is output containing the expression, filename and line number. Assertions become inoperable if NDEBUG is defined when assert.h is included.

10.1.7 Utilities <stdlib.h>

The file "stdlib.h" contains definitions of miscellaneous utility functions, such as text-to-numeric conversion routines, memory allocation routines, random number generator, environment command execution and process termination routines.

Function definitions :

double	atof(**const char** *s)
int	atoi(**const char** *s)
long	atol(**const char** *s)
double	strtod(**const char** *s, **char** **endp)
long	strtol(**const char** *s, **char** **endp, **int** base)
unsigned long	strtoul(**const char** *s, **char** **endp, **int** base)
int	rand(**void**)
void	srand(**unsigned int** seed)
void	*calloc(**size_t** nobj, **size_t** size)

void	*malloc(**size_t** size)
void	*realloc(**void** *p, **size_t** size)
void	free(**void** *p)
void	abort(**void**)
void	exit(**int** status)
int	atexit(void (*fcn)(**void**))
int	system(**const char** *s)
char	*getenv(**const char** *name)
void	*bsearch(**const void** *key, **const void** *base, **size_t** n, **size_t** size, **int** (*cmp)(**const void** *keyval, **const void** *datum))
void	qsort(**void** *base, **size_t** n, **size_t** size, **int** (*cmp)(**const void** *, **const void** *))
int	abs(**int** n)
long	labs(**long** n)
div_t	div(**int** num, **int** denom)
ldiv_t	ldiv(**long** num, **long** denom)

10.2 Variable Argument Lists <stdarg.h>

Occasionally it is useful to provide a function whose arguments are permitted to vary both in type and number. The printf/scanf family of functions are a primary example of this situation. Variable argument lists are facilitated by the routines defined in <stdarg.h>. Each function with a variable argument list must call "va_start" first and pass the name of the last non-variable argument. Before the function exits it must call "va_end" to perform the clean up of the argument list. In order to retrieve each argument in its appropriate type, "va_arg" is called, specifying the type of argument expected. The printf family of routines use the conversion specifications to guide argument conversion.

Function definitions :

(macro)	va_start(**va_list** ap, #lastarg#)
#atype#	va_arg(**va_list** ap, #atype#)
void	va_end(**va_list** ap)

10.3 Error Definitions <errno.h>

This header file contains the definitions of error codes which might be loaded into the global integer "errno", also defined here.

10.4 Non-Local Jumps <setjmp.h>

This library provides routines to allow fast exit from a deeply nested function call. "Setjmp" saves the program state information, and "longjmp" can be used to restore the program execution to that state at a later time. Variable values are NOT restored.

Function definitions :

```
int     setjmp(jmp_buf env)
void    longjmp(jmp_buf, int val)
```

10.5 Signals <signal.h>

The signal library provides facilities for handling exceptions. Certain events cause signals to be sent to a process. The raise function permits a process to signal itself. The signal routine defines how signals are handled : either by the default handler, by a custom handler, or by ignoring them. Various types of signals are defined including:

SIGABRT	abnormal termination
SIGFPE	floating point exception
SIGILL	illegal instruction
SIGINT	interactive interrupt
SIGSEGV	segmentation violation
SIGTERM	termination request

Function definitions :

```
void    (*signal (int sig, void (*handler)(int)))(int)
int     raise(int sig)
```

10.6 Date/Time Functions <time.h>

Routines are provided for manipulating date and time information.

Function definitions :

clock_t	clock(**void**)
time_t	time(**time_t** *tp)
double	difftime(**time_t** time2, **time_t** time1)
time_t	mktime(**struct tm** *tp)
char	*asctime(**const struct tm** *tp)
char	*ctime(**const time_t** *tp)
struct tm	*gmtime(**const time_t** *tp)
struct tm	*localtime(**const time_t** *tp)
size_t	strftime(**char** *s, **size_t** smax, **const char** * fmt, **const struct tm** *tp)

10.7 Implementation-defined Limits <limits.h> <float.h>

These header files contain definitions relating to implementation dependent size and magnitude limits for the standard integer (including **short** and **char**) and **floa**t (including **double**) types.

10.8 Locale Definitions <locale.h>

This library contains routines for setting and interrogating local definitions such as currency unit characters, thousands separators in numbers and other minor conventions.

Function definitions :

struct lconv	*localeconv(**void**)
extern char	*setlocale(**int** _category, **const char** *_locale)

Examples :

```
/********************************
* Program to invert the case of
* alphabetic characters.  input
* from stdin, output to stdout.
********************************/
#include <stdio.h>
#include <ctype.h>

main() {
  int   c;

  while ( (c=getchar()) != EOF )
    if (isupper(c)) putchar(tolower(c));
    else if (islower(c)) putchar(toupper(c));
    else putchar(c);
}


/**************************************
* This program demonstrates the use of
* setjmp and longjmp to exit quickly
* from deep nesting.  Run as is, and
* then after uncommenting the longjmp
* call to observe the effects.
**************************************/
#include <setjmp.h>

jmp_buf env;
int     lev=0;

main() {
  if (setjmp(env)==0)
    nest(20);
  printf("%d\\n",lev);
}

int nest(int l) {
  lev++;
  if (l) nest(l-1);
  lev--;
  /* longjmp(env); */
}
```

```
/*********************************
* This program loops forever and
* will ignore interrupt and term-
* inate request signals.
*********************************/
#include <signal.h>

main() {
  signal(SIGTERM,SIG_IGN);
  signal(SIGINT,SIG_IGN);
  while (1);
}

/*******************************************
* Program to insert and search stock records
* in a binary file.  Demonstrates i/o, conv-
* ersion routines and string manipulation.
*******************************************/
#include <stdio.h>
#include <stdlib.h>
#define NLEN 20

typedef struct {
  int   code;
  char  name[NLEN];
  float price;
} itemrec;

FILE    *stockfile;

main() {
  char menuselect();

  stockfile = fopen("stock.dat","a+b");
  while (1)
    switch ( tolower( menuselect() ) ) {
      case 'i': insert();      break;
      case 's': search();      break;
      case 'q': quit();        break;
      default : ;
    }
}

char menuselect() {
  char select,s[NLEN];

  do {
    printf("Select Function:\\n");
    printf("  (I)nsert\\n");
    printf("  (S)earch\\n");
    printf("  (Q)uit\\n");
    printf("Enter letter> ");
    select = tolower(gets(s)[0]);
  } while (select != 'i' && select != 's' && select != 'q');
  return select;
}

insert() {
  itemrec       item;
  char          s[NLEN];

  do {printf("Enter item code:     ");
  } while ((item.code=atoi(gets(s))) < 0);  do {printf("Enter item name:
");
  } while (strlen(gets(item.name))==0);
  do {printf("Enter item price:  ");
  } while ((item.price=atof(gets(s))) < 0.0);

  fseek(stockfile, 0, SEEK_END);
  fwrite(&item,sizeof(item),1,stockfile);
  fflush(stockfile);
}

search() {
  itemrec       check,want;
  int           searchby;
  char          s[NLEN];

  printf("Search by:\\n");
  printf("  (C)ode\\n");
  printf("  (N)ame\\n");
  printf("  (P)rice\\n");
  printf("Enter letter> ");
  searchby = tolower(gets(s)[0]);
  if (searchby != 'c' && searchby != 'n' && searchby != 'p') return;
  printf("Search value> "); gets(s);
  switch (searchby) {
    case 'c': want.code=atoi(s); break;
    case 'n': strncpy(want.name,s,NLEN); break;
    case 'p': want.price=atof(s); break;
    default : ;
  }

  fseek(stockfile, 0, SEEK_SET);
  while (!feof(stockfile)) {
    fread(&check,sizeof(check),1,stockfile);
    if ( (searchby=='c' && check.code==want.code)
    ||   (searchby=='n' && strncmp(check.name,want.name,NLEN)==0)
    ||   (searchby=='p' && check.price==want.price)) {
      printf("Found!!\\n");
      printitem(check);
      return;
    }
  }
  clearerr(stockfile);
  printf("Not found.\\n");
}

quit() {
  printf("bye\\n");
  exit(0);
}

printitem(itemrec item) {
  printf("Code:     %d\\n",item.code);
  printf("Name:     %s\\n",item.name);
  printf("Price:  %.2f\\n",item.price);
  printf("\\n");
}

/************************************
* Program to demonstrate the use of
* the routines in the time library.
************************************/
#include <stdio.h>
#include <time.h>
#define themost 100

main() {
  char thetime[themost];
  size_t most=themost;
  struct tm *ttt;
  time_t tt;

  tt=time(NULL);
  ttt=localtime(&tt);
  strftime(thetime,most,"%a %d of %b - %Y, day %j - %I%p\\n", ttt);
  printf(thetime);
}
```

Exercises :

1. Write a program which prints out it's arguments one per line.
2. Modify the program which inverts the case of alphabetic characters so that input is taken from files argv[1] through argv[argc-2], and output is written to argv[argc-1].
3. Write a program to calculate the average word length read on stdin. Assume a word is an uninterrupted sequence of alphabetic characters.
4. Write a program to count the frequencies of each character read from stdin. Print the non-zero frequencies and associated characters on stdout.
5. Modify the program from exercise 4 so that input and output go to files as specified by argv.
6. Modify the binary file stock program given so that the user may browse both forward and backward through the file of stock items.
7. Count word frequencies on stdin. Print out the 10 most commonly occuring words to stdout.
8. Obtain a listing of the current directory from within a c program.
9. Print out the date and time in any format you desire.

Index