Introduction to C

N. Chr. Albertsen
Informatics and Mathematical Modelling
Technical University of Denmark
Building 305

June 1997, March 2006

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-TECHNICAL REPORT-2006-09
ISSN 1601-2321

Contents

1. General

The principles of C are very much like those of PASCAL, although the syntax differs slightly. The programs are entered in free format, which the programmer may utilise to make the text reader-friendly, and every sentence is ended with a semicolon. Comments can be entered at any point by enclosing them in /* and */. e.g.: /* a comment */. Note that the C compiler is case sensitive.

A C program may consist of several program segments. One of them must be the main program, the rest are denoted functions. In contrast to PASCAL the concept procedure does not exist. Every segment starts with a number of declarations, after which follows the executable code. As in PASCAL both global and local variables may be declared. It is possible to combine a program from several separate files, one of which contains the main program, the others functions. This leads to the concept of scope of global variables. To avoid this extra complication, we will in the following consider only C programs contained in a single file.

2. Anatomy of a C program

In contrast to FORTRAN, where it is possible, although not advisable, to write a complete program without declarations, C poses a number of formal demands on the program structure. All variables must de declared, and prototypes must be included for all functions to be used, even if they belong to C's standard library, which incidentally is far more comprehensive than FORTRAN's and include functions of a more complex nature, such as functions that allow calls to the operating system as illustrated in the example CPRINT. It is, on the other hand, impossible to write a useful C program without the use of functions, since even a fundamental process like input and output of data requires the use of functions.

A function prototype is an expression which defines the type of the function and the number and types of its arguments, but nothing about the purpose of the function. Prototypes for C's standard functions are contained in files, which are an integrated part of the system. The user can include these files into his program with an #include command. The program would thus typically start with:

#include <stdio.h>
#include <stdlib.h>

where stdio.h is a file, which contains prototypes for a series of input and output functions, especially printf, and stdlib.h contains amongst others a prototype for exit, which can be used to stop a program. The angle brackets inform the compiler to search for the files among the system files. Then follows a list of prototypes for the user written functions. Since the main program formally is considered a function in C (a function called from the operating system) there should in principle be a prototype for main (as the main program must be called), but in practice the function doubles as its own prototype. Since main is a function, it can take arguments, argc and argv, transferring information from the operating system to the program. An example of this can be seen in CPRINT. For now it is assumed that no arguments are being used, whence the (unnecessary) prototype becomes:

```
int main(void);
```

Here it is indicated that main is of type int (meaning that it can return an integer to the operating system via exit) and has no arguments. The main program could instead have been given the type void, in which case no value could have been returned (the program could then be stopped with a "return;" statement). After the function prototypes, declarations of global variables or global data structures follow. These are data that can be used anywhere, both in main and in the

functions.

The actual program begins with:

```
int main()
```

where the parenthesis must be included, even when the argument list is empty. Had there been arguments, each argument should have been prefixed by a type declaration. There follows a list of commands contained within a set of braces: {program segment} , where { corresponds to **begin** in PASCAL and } to **end**. The program segment starts with type declarations for the local variables used in this segment. Here we wish to use an integer *i* and three double precision variables (FORTRAN REAL*8) *x, x2* and *x3*:

```
{
int i;
double x,x2,x3;
```

It is generally possible to initialise variables in declarations. We could therefore have written:

```
int i = 1;
double x = 1, x2 = 2, x3 = 3;
```

instead. Then follows the program code that executes the commands desired, and finally a return is made to the operating system with a code that may be used to indicate possible errors. If there has been no errors, a zero code is returned:

```
exit(0);
}
```

To see everything in the proper context, a small example is shown here of a program that calculates the second and third powers of the numbers 1., 1.2, 1.4,.....,5.:

```
/* ---------------- C example  No. 1 --------------------- */
/*            2      3                                    */
/* calculate x  and x  for x from 1 to 5 in steps of 0.2   */
#include <stdio.h>      /* contains prototype for printf   */
#include <stdlib.h>     /* contains prototype for exit     */
```

```
/*  function prototypes:  */
int main(void);

int main()
{
  int i;
  double x,x2,x3;
  for(i=1;i<=21;i++)
     {
     x=1.+(i-1)*0.2;
     x2=x*x;
     x3=x*x2;
     printf("   x=%8.2lf  x2=%8.2lf  x3=%8.2lf\n",x,x2,x3);
     }
  exit(0);
}
```

The constructions *for(..;..;..)* and *printf(.....,..,..,..)* will be discussed in detail later, here it suffices to mention that the former dictates that the following command is to be executed 21 times, and the latter prints the result on the terminal screen in columns. Notice that the command to be repeated consists of a block of sentences, enclosed in the construction { ...}. Notice also that the dynamical assignment of a value is done using = as in FORTRAN (*not* := as in PASCAL). The special C operator ++, which will be discussed later, causes the variable, to which the operator is attached, in this case *i*, to be increased by one.

3. Data type declarations for scalars

Declaration	*Meaning*
char	8 bit signed/unsigned integer depending on the compiler (here unsigned)
signed char	8 bit signed integer -128 $\leq$ signed char $\leq$ 127
unsigned char	8 bit unsigned integer 0 $\leq$ unsigned char $\leq$ 255
short int	16 bit signed/unsigned integer depending on the compiler (here signed)
signed short int	16 bit signed integer -32768 $\leq$ signed short int $\leq$ 32767
unsigned short int	16 bit unsigned integer 0$\leq$ unsigned short int $\leq$ 65535
long int	32 bit signed/unsigned integer depending on the compiler

	(here signed)
signed long int	32 bit signed integer -2147483648 $\leq$ signed long int $\leq$ 2147483647
unsigned long int	32 bit unsigned integer 0 $\leq$ unsigned long int $\leq$ 4294967295
int	16/32 bit signed/unsigned integer depending on the compiler (here 32 signed)
signed int	16/32 bit signed integer depending on the compiler (here 32)
unsigned int	16/32 bit unsigned integer depending on the compiler (here 32)
float	32 bit floating point number, precision 6 significant digits, numerical value between 10^{-44} and 10^{39} .
double	64 bit floating point number, precision 15 significant digits, numerical value between 10^{-323} and 10^{309}
long double	128 bit floating point number, precision 33 significant digits, numerical value between 10^{-4965} and 10^{4932}

Notice that there are no logical variables. Wherever a logical variable is required, a variable of type int is used, and the value is interpreted as false if is zero, otherwise as true. If the value is generated through a logical expression, the value for true will be 1. All standard functions returning floating point numbers are declared with type *double*, making it expedient always to use the type *double* instead of *float*, unless it is imperative to save storage.

Notice furthermore that there are no complex variables.

4. Data type declarations for vectors

A vector is a data structure with several elements of the same type, e.g. vector[10], which contains the elements vector[0], vector[1], ... , vector[9]. Vectors are declared with a type declaration as e.g.:

```
int vector[10];
double matrix[10][6];
char line[80];
```

Here vector is a one-dimensional vector with 10 elements of type int, matrix is a two-dimensional vector with 10 * 6 = 60 elements of type double and line is a one-dimensional vector of type char with 80 elements, where each element consists of a single character. The characters can be addressed individually as line[i], or

line can be considered as a text string. In the latter case, the last character in line must be a null-byte (\0 in C notation).

5. Data type declarations for struct's

It is possible in C to define a more complex structure denoted a *struct*. The easiest way to use a *struct* is first to define a prototype to inform the compiler of the internal structure of the *struct*. A *struct* suitable to contain a date could have the following prototype:

```
struct date {
int day;
int month;
int year;
char m_name[4];
};
```

This prototype is only an information to the compiler and does not reserve space in memory. Notice that a member of a *struct* can be a vector. It is now possible to introduce actual *struct* variables using the prototype, which we have named date, in the declaration. We could write, e.g.:

```
struct date hans,jesper,christian;
```

which would reserve memory space for three variables with a content as described in date. If we wish to store, e.g., the birthday of Christian as 16/1/1996 we can write:

```
christian.day = 16;
christian.month = 1;
christian.year = 1996;
strcpy(christian.m_name,"Jan");
```

Notice that it is only possible to store a text string with 3 characters in m_name, since there must be space for a null-byte at the end, and also that in C it is not possible to move a text string without invoking a *function* strcpy (se section on call of programs).

If there is a need for many *struct*'s it may be easier to introduce them as a vector of *struct*'s, e.g.:

```
struct date student[100];
```

where the birthdays of 100 students can be stored. An example of how the individual members of the *struct* can be addressed in this case is:

```
student[5].day = 1;
strcpy(student[5].m_name,"feb");
student[5].m_name[0]='F';
```

where the last line shows how to address a single element in a vector which is itself a member of a *struct* vector.

6. The pointer concept

In C the concept of a *pointer* is widely used. This is a variable that points to another variable. A pointer is, however, not just an address in memory. Each pointer must be associated with a specific type, i.e. the type of variable to which it is pointing. As with all other variables, a pointer must be declared. This is done with the operator *, denoted the *indirection* operator (not to be confused with the arithmetic operator for multiplication). E.g.:

```
int *pi;
double *px;
char *pc;
```

declares pi, px and oc as pointers to int, double and char, respectively. A pointer can be assigned a value using the address operator & as follows:

```
pi = &i;
px = &x;
pc = &c;
```

where i is an int, x a double and c a char variable. Now pi contains the address of i etc. The reverse operations are:

```
i = *pi;
x = *px;
c = *pc;
```

following which i is assigned the value of the cell to which pi points etc. As a consequence it is possible, anywhere in the program where an int variable (e.g. i) is required, to replace it with a pointer prefixed with * (e.g. *pi). It is also possible to perform certain arithmetical operations on a pointer. It is possible to add an int to a pointer, to subtract an int, or to subtract two pointers, provided the two pointers are of identical type. In the two first cases, the result is a new pointer of the same type, in the last case the result is an int. It is not possible to add two pointers, since it has no meaning, and it is not possible to use any other arithmetical operators on pointers.

A pointer is closely associated with the concept of a vector. Consider the following declarations:

```
int vek[10], *pvek;
```

where vek is declared as an int vector with 10 elements and pvek is declared as a pointer to int. We can now let pvek point to an arbitrary element in vek, e.g. the fourth (remember that vectors always start with element 0):

```
pvek = &vek[3];
```

If we specifically wish pvek to point to vek[0], it is sufficient to write:

```
pvek = vek;
```

since C regards the name of a vector as a pointer to its first element. Notice, however, that there is a difference between pvek and vek, inasmuch as pvek is a variable that can be changed dynamically, whereas vek is a constant that always points to the same cell in memory, and therefore cannot appear on the left hand side of a sentence with an assignment. Since a pointer is declared with a type, C can calculate the length of the elements to which it points. The consequence of increasing pvek with 1 is therefore to make it point to the next vector element, *not* to the next byte in memory. This is true also if a pointer to a *struct* is considered:

```
struct date *pstudent = student, *phans = &hans;
```

where pstudent and phans have been declared as pointers to a struct vector and to a single struct, respectively, and where pstudent furthermore is initialised to point to the first element of the vector student (se section on structs), while phans is initialised to point to the struct hans. If we wish to address the member of a struct with a pointer, this can be done in (at least) two ways. We wish to assign

the variable idag a value from day in date:

```
idag = (*phans).day;
idag = phans -> day;
```

where we in the first line use the '*' operator to change phans to hans, and then the operator '.' to address the element day. In the second line we have introduced the new operator '->' which points to an element in a struct. If we wish instead to select day from the fourth element in student, this can be done with pointer arithmetic. Since C knows the size of date, we will, as mentioned above, move the pointer one element forward every time we add 1 to it (and not one byte, e.g.). The result is therefore:

```
idag = (*(pstudent+3)).day;
idag = (pstudent+3) -> day;
idag = pstudent[3].day;
```

where we have again shown both forms. The third line may seem surprising, but as mentioned above, the name of a vector has the type of pointer, like pstudent, and we can therefore replace student[3] with pstudent[3], provided, of course, that pstudent has been assigned the value &student[0]. Notice that we still only have one pointer, not a vector of pointers. In more complex expressions it is usually advisable to insert parentheses to make the meaning of the expression quite clear, since the compiler will otherwise follow rules of priorities for the operators that may be difficult to remember. As an example of the problems that may arise, consider the following:

```
i = * pvek ++;
i = * (pvek ++); (i = * pvek; pvek = pvek + 1;)
i = (* pvek) ++; (vek[0] = vek[0] + 1; i = vek[0];)
```

The expression in the first line may theoretically be interpreted as shown in the second or the third line, and the compiler will choose that in the second line.

Similar to the introduction of vectors of struct's, we can of course introduce vectors of pointers. Just as a vector is naturally associated with a one-dimensional vector, a vector of pointers is naturally associated with a two-dimensional vector. Consider the following example:

```
int *pdvek[2];
int dvek[2][3];
```

```
int (*ppdvek)[3];
```

Here pdvek is a vector with two elements, both of which are pointers to int, and dvek is a two-dimensional vector, which can be considered as a matrix with two rows and three columns. Contrary to FORTRAN, which stores a matrix column by column, C stores it row by row, in other words, dvek is considered as a vector with two elements, each of which is a vector with 3 int. To increase the confusion, the third line shows a declaration which is similar to the one in the first line, but with a set of parentheses inserted. In fact this declares a single pointer, ppdvek, whose type is to point to an element consisting of 3 int; ppdvek is therefore *not* a vector of pointers. To illustrate the possibilities of the notation, a number of examples follow:

```
pdvek[0] = dvek[0]; /* initialises pdvek[0] */
pdvek[1] = dvek[1]; /* initialises pdvek[1] */
ppdvek = dvek; /* initialises pddvek */
i = * (pdvek[0]+1); /* i = dvek[0][1] */
i = (*(ppdvek+1))[2]; /* i = dvek[1][2] */
i = *(*(ppdvek+1)+2); /* i = dvek[1][2] */
```

In lines 1 and 2 we set the two pointers in pdvek to point at the first and second row of dvek, respectively. Notice that just as the name of a one-dimensional vector is regarded as a pointer, dvek[0] is regarded as a pointer to the first element in the first row of dvek. As it is apparent from line 3, ppdvek and dvek both have the same level of indirection, inasmuch as they are both pointers to a collection of 3 int. Comparing lines 1 and 3 may lead to the thought, that line 3 could be replaced by the expression: ppdvek = pdvek;, but this would be an illegal expression, since ppdvek is a pointer to 3 int, while pdvek is a pointer to a pointer.

It is also possible to define a pointer to a function. In this way, the name of one function may be transferred to another function through a call. E.g. the declaration:

```
int (*psub)(double , int);
```

defines psub as a pointer to a function, requiring a double and an int variable as input, and returning an int value to the caller. An example of the use of such a pointer may be found in the section on function calls.

A good introduction to more complex C declarations may be found in [3].

7. Initialisation and assignment

Every variable in C addresses an area in the computer memory. The size of the area can vary from 1 byte (= 8 bits) for e.g. a char variable to many millions of bytes for e.g. multi-dimensional vectors. All variables therefore have a value at all times, corresponding to the content of the memory cells, but if the program has not specifically defined the value, it will be random.

A variable can be assigned a value statically (initialisation), so that the variable has this value when the program starts executing, or it can be assigned a value dynamically with an executable expression. Variables that have not been assigned a value are in principle undefined when the program starts. An optimistic assumption, that they have the default value zero is invalid.

Notice that vectors can only be initialised if they are global or if their declarations are prefixed with the key word static (see section on function calls).

Initialisation of a variable can be done in a declaration as e.g.:

```
int i = 5, j = 12;
int iv[3] = 1, 2, 3, jv[] = 4, 5, 6;
double x = 10.e5, pi = 3.1415927;
double xv[2] = 10.e5, 10.e3, yv[] = 3., 4.;
char c = 'A';
char lamf[5] = "Lamf", kursus[ ] = "Kursus 310";
```

where the first line initialises i and j to 5 and 12, respectively, the second line initialises 3 elements in the iv to 1, 2, and 3, and 3 elements in jv to 4, 5, and 6. Notice that the compiler automatically creates jv as jv[3] by counting the number of parameters in the initialisation. In the following line x and pi are initialised to 10 and π, in the next line the elements of the vectors xv and yv are initialised, and yv is created as yv[2]. In the next line, c is assigned the value 'A', and notice that, since c is a single char, no null-byte is required. In contrast lamf is declared with 5 elements to make room for a null-byte at the end. In particular for text strings it is useful to utilise the [ ] construction as with kursus, since the compiler will then creates the vector with the correct number of elements (here 11 since \0 is automatically added at the end). The string could also have been initialised as:

```
char lamf[ ] = {'L','a','m','f','\0'};
```

in analogy with the arithmetical initialisations.

A two-dimensional vector is initialised row by row, as e.g.:

```
int dvek[2][3]={{1,2,3},{4,5,6}};
char *pc[ ]={"alfa","beta","gamma"};
int *pdvek[ ]={dvek[0],dvek[1]};
```

where the second line shows how, in the case of a vector of pointers to char, it is possible to initialise the text strings pointed to at the same time as vector elements. A vector of pointers to int can be initialised as in line 3.

A variable is typically assigned a value dynamically with an command, where the variable appears on the left hand side of the equal sign, while on the right hand side there can be a number, another variable, or an expression. Notice that a variable can change value, even if it appears on the right hand side of the equal sign, by means of the operators '++' and '–'. Since a text string is regarded as a vector of characters, this is also true for the individual members of the vector. It is, however, not possible in a single command to move an entire text string without the use of either a standard function (*strcpy*) or a construction with *do*, *while*, or *for*.

Notice that C several variables to be assigned a value in the same command, e.g.:

```
a = b = 10. * c;
```

where first b is assigned the value 10.*c, after which a is assigned the same value. It is also possible to join several commands to a single command using the comma operator, e.g.:

```
((a = 1.) , (b = 2.) , (c = 3.));
```

where first a is assigned the value 1., then b the value 2. and finally c the value 3. If the meaning remains clear (as in this case), the parentheses can be removed. In other cases (e.g. parameter lists in call to functions), where ',' also serves other purposes the parentheses will be necessary. The comma operator is particularly useful in places, where normally only one command is allowed. It is, e.g., possible to perform several initialisation in a *for* construction in this way.

8. Operators

Symbol	*Meaning*	*Example of use*
+	addition	a = b + c;
–	subtraction	a = b – c;
*	multiplication	a = b * c;

/	division NB: if both b and c are type int the result is truncated even if a is of type double	a = b / c;
%	remainder NB: both b and c must be of type int	a = b % c;
<	less than	a = b < c;
<=	less than or equal to	a = b <= c;
>	greater than	a = b > c;
>=	greater than or equal to	a = b >= c;
==	equal to	a = b == c;
!=	not equal to	a = b != c;
&&	AND	a = b && c;
||	OR	a = b | | c;
+=	addition assignment	a += b;
−=	subtraction assignment	a −= b;
∗=	multiplication assignment	a ∗= b;
/=	division assignment	a /= b;
%=	remainder assignment NB: both b and c must be of type int	a %= b;
<<	bitwise left shift NB: both b and c must be of type int, corresponds to multiplication with $2^c, c \geq 0$	a = b < < c;
>>	bitwise right shift NB: both b and c must be of type int, corresponds to division with $2^c, c \geq 0$	a = b > > c;
&	bitwise AND	a = b & c;
|	bitwise inclusive OR	a = b | c;
^	bitwise exclusive OR	a = b ^ c;
<<=	bitwise left shift assignment NB: b must be of type int	a <<= b;
>>=	bitwise right shift assignment NB: b must be of type int	a <<= b;
&=	bitwise AND assignment	a &= b;
|=	bitwise inclusive OR assignment	a |= b;
^=	bitwise exclusive OR assignment	a ^= b;

!	NOT	a = ! b;
∼	bitwise complement	a = ∼b;
–	negation	a = – b;
+	no effect	a = + b;
∗	value in address	a = ∗ p;
&	address of value	p = & a;
++	increment, prefixed	a = ++ b;
++	increment, postfixed	a = b ++;
– –	decrement, prefixed	a = – – b;
– –	decrement, postfixed	a = b – –;
?:	conditional operator	a>b ? c=0 : c=1;

The arithmetical operators +, -, *, /, and % should be self-explanatory. As a speciality of C, they can be combined with = to form a more compact notation. Consider the example: a += b; Here the left hand operator first operates on a and b, after which a is assigned the resulting value, hence the command is equivalent to: a = a + b; The operators <, <=, >, >=, ==, and != operate on arithmetical variables or expressions and generate a logical result (true = 1 or false =0). The operators && and || operate on variables, assumed to have a logical value (false =0 otherwise true) and generate a logical result (true = 1 or false =0). The operators << and >> operate on bit level, and move the all bits in the first variable a number of places specified by the second variable. Bits that are moved outside the space reserved for the first variable are lost. The operators &, | and ˆ also operate on bit level, such that every bit in a is the result of the specified operation on the corresponding bit in b and c, respectively. It is therefore imperative that the lengths of the three variables are identical. These operators can, similar to the arithmetical operators, be combined with a = creating a more compact notation. As an example, the command a <<= 2 will result in a being multiplied by 4. The operator ! changes the logical value of a variable, while ∼ changes all bits in the variable, zeroes to ones and vice versa. The operator - changes the sign of a variable, while + no effect has. The operator * assigns the value of the memory cell, to which the *pointer* p points, to the variable on the left hand side, while &, on the contrary, calculates the pointer to the memory cell where a is stored. The operators ++ and - - will result in an increment or decrement of 1, respectively, of the associated variable. These operators can be either prefixed or postfixed. Being prefixed will cause the operation to be performed before the variable is used, being postfixed after the variable has been used. Hence a = ++ b; means that b is increased by 1, after which the value is assigned to a, while a = b - -; means that a is assigned the value of b before 1 is subtracted from b. Notice that these operators influence the value of variables, not only on the left hand side of

=, but also on the right hand side. As a consequence it is possible in C to write a command without the use of =, since

```
i ++;
```

is a legitimate command, which increases the value of i by 1. The use of the operator ?: is best described by an example:

```
a = (logical expression) ? (expression 1) : (expression 2);
```

If the logical expression is true, a will be assigned the value of expression 1, else the value of expression 2.

The variables that appear on the right hand side of the equal sign in the examples, may be replaced by expressions enclosed in parentheses.

9. Logical instructions for program control

If a sequence of commands are to be performed a number of timed, it is possible to use a construction of the form *do ... while*, *while*, or *for*, as illustrated in the following three examples:

```
do
{
.
. (commands to be performed)
.
} while (logical expression);
```

Here the commands enclosed in {} will be performed until the logical expression is false. Since while appears at the end of the construction, the commands will always be performed at least once.

```
while (logical expression)
{
.
. (commands to be performed)
.
}
```

Here the commands enclosed in {} will also be performed until the logical expression is false. Since while now appears at the beginning of the construction, the commands will not always be performed at least once.

```
for (command 1 ; logical expression ; command 2)
{
.
. (commands to be performed)
.
}
```

In this construction, command 1 is performed first. Then a loop is started in which the logical expression is tested, and, if true, the commands enclosed in {} are performed, after which command 2 is performed. Since command 1 is only performed once, it can be used to initialise a counter, which is subsequently tested in the logical expression, and finally is increased or decreased in command 2. Notice that if the logical expression is true when it is tested the first time, command 2 will always be the last command performed before the program continues.

To make the execution of a number of commands dependent on the result of a previous calculation, an *if* construction can be used:

```
if(logical expression)
{
.
. (commands to be performed)
.
}
```

where the commands enclosed in {} will be executed if the logical expression is true only. The construction can be supplemented with *else if* and/or *else*:

```
if(logical expression No.1)
{
. (commands to be performed only if logical
. expression No.1 is true)
}
else if(logical expression No.2)
{
. (commands to be performed only if logical
. expression No.1 is false and
```

```
. expression No.2 is true)
}
else
{
. (commands to be performed only if both logical
. expressions No.1 and 2 are false)
}
```

Normally the program commands are executed sequentially. If it is absolutely necessary to jump to another part of the same program segment, this can be done with the *goto* instruction:

```
goto label
.
.
.
label: .....
```

where label is a text string. Notice the use of ':'. Using a *switch* construction it is possible to perform a selective jump depending on the value of an integer parameter or an integer expression, e.g.:

```
switch(i)
{case 2:
.
. (commands to be performed if i=2)
break;
case 4:
.
. (commands to be performed if i=4)
break;
default:
.
. (commands to be performed if i is neither 2 nor 4)
}
```

There can be an arbitrary number of *case...*: expressions with different values, and default: is optional. If not included, no action will be taken if i does not match any case value. The insertion of *break* as the last command in a *case* segment

effects a jump to '}'. If left out, the commands in the following *case* segment (if any) also be executed.

10. Call of subprograms

C operates with a type of subprograms denoted functions. A subprogram is allocated a name, e.g. func, and may be called from the main program or from any other function. In contrast to FORTRAN, a function may call itself recursively, which opens up wide ranges of possibilities for complicated constructions. Recursion is made possible by the fact, that C uses "call by value" in function calls, which means that a memory cell is created in the function for each argument in the call, and the value of the argument in the program calling the function is copied to the function. Another advantage, in addition to being able to use recursion, is that it is allowed to use numbers in calls to functions, since the value of the number cannot be affected by commands in the function. Functions can return one value to the caller using a return(x) command, where x is the value to be returned. As in FORTRAN the returned value will be inserted at the place where func appears. The bane func is not a variable and can therefore not be assigned a value, but it is possible to transfer the name of a function as a parameter in the call to another function, where it must be declared as a pointer to a function.

A disadvantage of the "call by value" construction is that only one value can be returned from a function in contrast to e.g. a SUBROUTINE in FORTRAN, which can return an arbitrary number of values via the argument list. It is, however, possible to circumvent this limitation by replacing variables with pointers to variables in the function call. If one of the parameters in particular is the name of a vector, this will in itself be a pointer, as mentioned previously. It is thus possible, from the function, to address all variables in the calling program, change their values. Also in this case the "call by value" convention is used, but it is now pointers, whose values are being transferred, i.e. it is possible make arbitrary arithmetical operations on them in the function, since they are only copies of those specified in the call.

Before a function can be used, a prototype must be declared, specifying the type of the function and the types of all the parameters used in calls to it, e.g.:

```
double power(double , int);
double vmul(int *, double [ ] , double [ ]);
double xpower(double (*)(double,int));
```

where line 1 shows the prototype for a function named power, returning a double value and requiring one input parameter of type double and another of type

int, line 2 shows the prototype for a function that returns a double and takes a pointer to an int and two double vectors as input, and line 3 shows the prototype for a function returning a double value and requiring one input parameter of type pointer to a function, itself returning a double and requiring a double and an int as input. As an example of the use of these functions consider:

```
int i;
double x, y, vx[10], vy[10];
.
.
x = power (y , i);
.
i = 10;
vmul(&i , vx , vy);
x = xpower(power);
```

showing that it is allowed to neglect the value returned by e.g. vmul, if the purpose of the function only is to manipulate the elements in the vector vx, and place the results in the vector vy. It is possible to define a function with no parameters, but the parentheses must not be omitted. In this case the parameter list in the prototype is specified as (void). If the function does not return a value, as vmul, its type can be specified as void in the prototype. For each function introduced by the user, a program segment must be included. These can be placed after the main program. For the above example this may look as:

```
double power(double x, int n)
{
int i;
double p=1.0;
for(i=1 ; i <= n ; ++i)
p*=x;
return(p);
}
```

where the first line contains the function name and a list of the local parameters to which values of the variables, appearing in the call, are copied. In the two next lines local parameters are declared. Notice that local parameters in C are reallocated in each call, whence the value from previous calls are lost. To save the value of a local parameter from one call to the next, it must be declared static, e.g.:

```
static int i;
```

It is, of course, important, that both the function and its arguments are declared exactly as in the calling program. The last command associates power with the value to be returned, and causes a jump back to the calling program. The second program could have the following form:

```
double vmul(int *pi, double vx[ ], double vy[ ])
{
int j;
.
.
j = *pi;
vx[0] = 1.;
.
.
return (0.);
}
```

The last program could look like:

```
double xpower(double (*pf)(double, int))
{
double z,x=4.;
.
z = pf(x,3);
.
return(z);
}
```

where pf is declared as a pointer to a function, which, when called from the main program with the command shown, represents power(x,n), and is set to power(4.,3).

C has a long list of built-in functions (see section about standard functions). These are used exactly as user defined functions, but do not demand that the user writes a prototype and, obviously, no program code. Instead they demand that the appropriate header files with the prototype declaration for the functions used, are included.

11. Input/Output commands

It is essential for a computer program to be able to communicate with its surroundings. This communication takes place via the operating system, and is in C accomplished by standard functions declared in the header file stdio.h. There is a long series of functions for this purpose, but we shall consider only the most basic. The simplest input operation is to read a single character from the terminal, which is done with the getchar() function. The function has no arguments and returns the character as an int. The opposite process, to write a character on the terminal, is done with the function putchar(c), where the argument c is the desired character stored in an int.

To communicate with a file, it is necessary first to open the file. This is done with the function fopen(s1,s2), where the arguments are two text strings. The first string, s1, contains the name of the file, e.g. s1 = "minfil.data", and the second string contains information about the operation to perform, s2 = "r" if the file is to be opened for read operation, s2 = "w" if it is to be opened for write from the start of the file and s2 = "a" to append data to the end of the file. The function returns a pointer to a struct named FILE, which is defined in stdio.h. This pointer is used in all following read/write commands to address the file. The function getc(FILE *) reads a character from a file and putc(c,FILE *) writes a character (c of type int) to a file. Finally the file is closed with the function fclose(FILE *). An example of a program segment, which opens two files and copies a character from one file to the other:

```
FILE *fpinp,*fpout; /* pointers to I/O files */
.
.
fpinp=fopen("input.fil","r");
fpout=fopen("output.fil","w");
i = getc(fpinp); /* read 1 character from file */
putc(i,fpout); /* and write it in the output file */
fclose(fpinp); /* close the input file */
fclose(fpout); /* close the output file */
```

Formatted printout of the values of variables on the terminal is done with the function printf(char *, arg1, arg2 ,...), where the first argument is a text string with formatting instructions, while arg1, arg2,... are the variables, whose values are to be printed. The format string contains the text to be printed, including special formatting codes, which are all a combination of \ and another character. The most important are shown in the table below. Furthermore the string must

contain a conversion specification for each argument after the comma. These conversion specifications are all a combination of % and another character as shown in the table below (to print a % it must be repeated: %%).

symbol	*represents*
\n	line shift
\t	horizontal tabulator stop
\v	vertical tabulator stop
\b	backspace
\r	carriage return
\f	page shift
\a	alarm
\\	\
\?	?
\’	’
\”	”
\ooo	octal number, ooo represents 1, 2 or 3 digits in the interval $0 \leq o \leq 7$
\xhh	hexadecimal number, hh represents 1 or 2 digits in the interval $0 \leq h \leq f$
%d %i	int, decimal representation
%u	unsigned int, decimal representation
%o	int, octal representation
%x	int, hexadecimal representation
%c	char
%s	char *, text string ending in a \0
%f	double
%e	double, exponential notation
%g	double, as %e if the exponent is <-4 or greater than the number of decimals specified
%p	hexadecimal address in pointer

If a more precise formatting is required, it is possible, between % and the following character, to insert: 1) et - if the symbol is to be left adjusted, 2) n.m where n is an integer indicating the minimal number of columns the symbol should span and m is an integer indicating the precision of the symbol, i.e. the number of decimals to include if the symbol is of type double, the minimum number of digits if the symbol is of type int, and the maximum number of characters is the symbol is a text string, and finally 3) an h if an int should be written as short and an ℓ if

it is to be written as long. The use of printf is exemplified in the section with examples.

Formatted input from the terminal is done with a similar function, scanf(char *, parg1, parg2,...), where the first argument is a format string, as discussed for printf, while the arguments after the comma now must be pointers to the variables, whose values are to be input, since the function scanf, as all other functions, use "call by value", see the section about call of subprograms.

Formatted input/output from/to a file are performed by two functions, fprintf(FILE *, char *, arg1, arg2,...) and fscanf(FILE *, char *, parg1, parg2,...), which are similar to printf and scanf, but have a pointer to struct FILE as their first argument, just as getc and putc.

12. Standard functions

Here follows a list of the most important standard functions in C. The complete list may be found in the C manual.

Name	*Function*	*Type*	*Argument type*
<math.h>:			
abs(i)	absolute value	int	int
acos(x)	arccos	double	double
asin(x)	arcsin	double	double
atan(x)	arctan	double	double
atan2(y,x)	Arg(x+iy)	double	double, double
ceil(x)	closest larger integer	double	double
cos(x)	cos	double	double
cosh(x)	cosh	double	double
erf(x)	error fct.	double	double
erfc(x)	1-erf(x)	double	double
exp(x)	e^x	double	double
fabs(x)	absolute value	double	double
floor(x)	closest smaller integer	double	double
gamma(x)	log ($\|\Gamma(x)\|$)	double	double
j0(x)	$J_0(x)$	double	double
j1(x)	$J_1(x)$	double	double
jn(n,x)	$J_n(x)$	double	int, double
log(x)	$\log_e(x)$	double	double
log10(x)	$\log_{10}(x)$	double	double

pow(x,y)	x^y	double	double, double
sin(x)	sin	double	double
sinh(x)	sinh	double	double
sqrt(x)	$\sqrt{x}$	double	double
tan(x)	tan	double	double
tanh(x)	tanh	double	double
y0(x)	$Y_0(x)$	double	double
y1(x)	$Y_1(x)$	double	double
yn(n,x)	$Y_n(x)$	double	int, double
<stdio.h>:			
fclose(fp)	close file	int	FILE *
fopen(s1,s2)	open file	FILE *	char *, char *
fprintf(fp, s,...)	write formatted to file	int	FILE *, char *,...
fscanf(fp, s,...)	read formatted from file	int	FILE *, char *,...
getc(fp)	read 1 character from file	int	FILE *
getchar()	read 1 character from terminal	int	void
printf(s,..)	write formatted to terminal	int	char *,...
putc(c,fp)	write 1 character to file	int	int, FILE *
putchar(c)	write 1 character to terminal	int	int
scanf(s,...)	read formatted from terminal	int	char *,...
<stdlib.h>:			
exit(i)	exit to OS	void	int
system(s)	system call	int	char *
<string.h>:			
strcmp(s,t)	compares two text strings	int	char *, char *
strcpy(s,t)	copies text string t to s	char *	char *, char *

13. Preprocessor directives

Before a C file is sent to the compiler, it is automatically processed by a "preprocessor", which reacts to the presence of certain "directives" in the file. We will here only consider some of the most useful:

```
#include "fn.ft"
#include <filnavn.h>

#define VAR 5

#if VAR == 5
(program segment 1)
#elif VAR == 0
(program segment 2)
#else
(program segment 3)
#endif
```

We have already seen the #include directive used earlier. This has two forms as shown. In the first line, a file named fn.ft is included from the disc at the place in the C file, where the directive is placed. In the second line, a system file named filnavn.h is included at the place, where the directive is placed. The suffix .h is used for all C's system header files. Since the user normally does not know where on the disc these files reside, the name is written with <> instead of "", indicating that the preprocessor knows where to find them.

The define directive is used to assign a symbolic name to e.g. a number. It is customary to write these names in capital letters to remind the reader that they are not variables, hence they cannot be redefined with an assignment command. Such a symbol can be used anywhere the compiler expects to find a number of the type in question, but it can also be used to exclude parts of the program from compilation by combining it with one of the following directives. The expression in line 4 is true only if VAR = 5. The directive causes the following part of the file, denoted program segment 1, to be neglected by the compiler if the expression is false. The directive in line 6 has the meaning, that if the preceding segment was not included, the following, segment 2, will be included if VAR = 0, and if none of the preceding expressions were true, #else will cause segment 3 to be included. These directives are in force until an #endif directive. It is optional whether the #elif and/or #else directives are included.

An example of the use of these directives may be found in example No.2.

14. Internal representation of data

To understand the limits that apply to accuracy and range of variables in a computer, it is necessary to look at the internal representation of data. The smallest logical unit in a computer memory is a *bit*, which may be 0 or 1. Since a single bit very seldom can contain useful information, it is unnecessary and impractical to introduce an address for each single bit. The smallest addressable unit has therefore been chosen to be an ensemble of 8 bits, an octet or *byte*. In C a byte can represent a single character (*char*). For the representation of an *int* variable, C requires a minimum of 2 bytes, while normally 4 bytes are used. For the representation of a *float* variable (single precision), 4 bytes are necessary, for a *double* (double precision) 8 bytes and for a *long double* (extended precision) 16 bytes.

It is advisable to introduce a system of numbering for the bits in a data unit, and we will therefore define the least significant bit as number 0, i.e. the bits are counted from the right.

We will first consider a variable of the type *char*. Since precisely one character is stored in each byte, it is possible to distinguish between $2^8 = 256$ different characters. The two most important character sets for computers are ASCII (American Standard Code for Information Interchange) and EBCDIC (Extended Binary Coded Decimal Interchange Code), primarily used by IBM. In both sets there appear both printable and non-printable symbols. The non-printable symbols can normally not be entered from the keyboard and are used in the communication between the computer and its peripheral units. In the ASCII code, the first 32 symbols are non-printable, the next 96 printable and the remaining 128 undefined. In the EBCDIC code the first 64 symbols are non-printable and the remaining 224 contain the same printable symbols as the ASCII code, evenly distributed, and a number of undefined symbols. The value of a byte is normally given in binary, octal or hexadecimal form. We will distinguish between these by appending to the number a b (for binary), an o (for octal) or an h (for hexadecimal). In the hexadecimal system the numbers from 0 to 9 are supplemented by a = 10, b = 11, c = 12, d = 13, e = 14 and f = 15.

As an example, the ASCII value for a blank (sp in the tables) is:

00100000b = 040o = 20h = 32

We will now consider a variable of type *short int*. Since the 16 bits used to store it must be able to contain both positive and negative integers, the most significant bit (here number 15) is reserved as the sign bit. If bit 15 is 0, the integer is positive, otherwise negative. For a positive integer the bits numbered 0 to and including 14 contain the value of the number in binary form. The largest

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	sp	!	”	#	$	%	&	’	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Table 1: ASCII codes XYh, where X is row and Y is column number (sp denotes a blank)

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	NUL	SOH	STX	ETX	PF	HT	LC	DEL	GE	RLF	SMM	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	TM	RES	NL	BS	IDL	CAN	EM	CC	CUI	IFS	IGS	IRS	IUS
2	DS	SGS	FS		BYP	LF	EOB	ESC			SM	CU2		ENQ	ACK	BEL
3			SYN		PN	RS	UC	EOT				CU3	DC4	NAK		SUB
4	sp										ç	.	<	(	+	|
5	&										!	$	*	)	;	^
6	-	/									|	,	%	_	>	?
7										‘	:	#	@	’	=	”
8		a	b	c	d	e	f	g	h	i						
9		j	k	l	m	n	o	p	q	r						
a		~	s	t	u	v	w	x	y	z				[		
b														]		
c	{	A	B	C	D	E	F	G	H	I						
d	}	J	K	L	M	N	O	P	Q	R						
e	\		S	T	U	V	W	X	Y	Z						
f	0	1	2	3	4	5	6	7	8	9						EO

Table 2: EBCDIC codes XYh, where X is row and Y is column number (sp denotes a blank)

positive number that can be stored in 2 bytes thus consists of 15 ones and equals 2^{15} -1. Negative numbers are stored in two's complement form, i.e. the number 2^{16} is added to the number, which is then stored in binary form. An example:

0000000000000001b = 0001h = 1 , 1111111111111111b = ffffh = -1
0000000000000111b = 0007h = 7 , 1111111111111001b = fff9h = -7

The numerically largest number that can be stored is therefore 8000h = -2^{15}. For the type *int* the sign bit is number 31 and the largest positive number that can be stored is 2^{31} -1, and the numerically largest negative is -2^{31}. The length of type *long int* is, on the computers we consider, the same as for *int*. Variables of type *unsigned* have no sign bit and can therefore store numbers from 0 to 2^{16} -1 (*unsigned short int*) and from 0 to 2^{32} -1 (*unsigned int* and *unsigned long int*).

Finally we will consider the representation of decimal numbers, first *float*. To be able to store as large a range of numbers as possible, such "floating" numbers are stored as exponent and mantissa. The precise structure varies from compiler to compiler. Here we shall only mention the IEEE standard.

First we shall see how to code an arbitrary real number, X, as *float* in the IEEE standard. If X≡0. the representation consists of all 0 bits. Otherwise the first step is to write X as a product:

$$X = Xs\ Xm\ 2^N\ ,\ Xs = \pm 1\ ,\ 2. > Xm \geq 1.$$

where Xs is the sign, Xm is the mantissa and N is the exponent. Then the mantissa is rewritten as a binary decimal fraction:

$$Xm = 1.xxxxxxxxxxxxxxxxxxxxxxx_2$$

where x can be 0 or 1. The first digit after '.' represents 2^{-1} , the next 2^{-2} , ..., and the last 2^{-23}. It is thus not possible to distinguish between two numbers for which Xm deviates less than 2^{-23}, corresponding to a precision of between 6 and 7 decimals in the decimal system. Since N can be both positive and negative, an M is introduced as M = N +127, which must always be a positive integer that can be represented by 8 bits. This means that N must comply with:

$$-126 \leq N \leq 128$$

giving the following range of numbers:

$$-(2-2^{-23})\ 2^{128} \leq X \leq -2^{-126} \text{ or}$$

$2^{-126} \leq X \leq (2\text{-}2^{-23}) \, 2^{128}$ plus 0.

The binary code, stored in memory, consists of: bit 31 is set according to Xs, 0 if Xs is 1, 1 if Xs is -1. The positive number M is stored in binary form in the bits 30 up to and including 23, and the 23 digits marked x in the binary decimal fraction are stored in the bits 22 to 0. Actually the numerically smallest number, that can be stored, is less than indicated above, since the IEEE standard operates with so-called "denormal" numbers. For these numbers the representation of Xm changes to:

$Xm = x.xxxxxxxxxxxxxxxxxxxxxxx_2$

and N is set to -127. Furthermore numbers with N=128 have a special meaning (infinite, NaN, or indefinite). The full range therefore becomes:

$-(2\text{-}2^{-23}) \, 2^{127} \leq X \leq -2^{-149}$ or
$2^{-149} \leq X \leq (2\text{-}2^{-23}) \, 2^{127}$ plus 0.

For double precision (*double*) the sign bit is number 63, M is defined as M = N + 1023 and stored in the bits 62 to 52 and the digits marked x (of which there are now 52), are stored in the bits 51 to 0. This means that N must now comply with:

$-1022 \leq N \leq 1024$

but, due to the above mentioned peculiarities, the range becomes:

$-(2\text{-}2^{-52}) \, 2^{1023} \leq X \leq -2^{-1074}$ or
$2^{-1074} \leq X \leq (2\text{-}2^{-52}) \, 2^{1023}$ plus 0.

The range is thus much larger than for single precision and the accuracy is increased to more than 15 decimals in the decimal system.

For extended double precision (*long double*) the sign bit is number 127, M is defined as M = N + 16383 and stored in the bits 126 to 112 and the digits marked x (of which there are now 112), are stored in the bits 111 to 0. This means that N must now comply with

$-16382 \leq N \leq 16384$

but, due to the above mentioned peculiarities, the range becomes

$-(2-2^{-112})\ 2^{16383} \leq X \leq -2^{-16494}$ or
$2^{-16494} \leq X \leq (2-2^{-112})\ 2^{16383}$ plus 0.

The range is thus still larger than for double precision and the accuracy increased to about 33 decimals in the decimal system.

Nothing has so far been mentioned about the physical storage of the bits in the computer memory. It is done in one of two ways. Either the number is stored as it is, i.e. the 8 first bits are stored in e.g. address px, the next 8 bits in px+1, et cetera such that the least significant byte is allocated the highest address ("big-endian" principle), or the number is stored in reverse order such that the last 8 bits are stored in px, the last but one byte in px+1 et cetera ("little-endian" princip). Normally it is of no interest which method is used, but if by mistake a parameter in a call to a function is declared with different lengths in the calling program and in the function, bits from different variables get mixed, and the result will naturally depend on the physical storage method of the bits.

15. Examples of C programs

```
Examples of C programs
/* ------------------ C example  No. 2 ----------------------- */
/*            2      3                                        */
/* calculate x  and x  for x from 1 to 5 in steps of 0.2       */
#include <stdio.h>          /* contains prototype for printf   */
#include <stdlib.h>         /* contains prototype for exit     */
#define EXAMPLE  2          /* preprocessor directive          */

/*  function prototypes:  */
int main(void);

int main()
{
      int i;
      double x,x2,x3;
      i=0;
      printf("Table of 2. and 3. powers of x\n");
#if EXAMPLE == 1
      while (i < 21)
        {
        i=i+1;
```

```
        x=1.+(i-1)*0.2;
        x2=x*x;
        x3=x*x2;
        printf("     x=%8.2lf  x2=%8.2lf  x3=%8.2lf\n",x,x2,x3);
        }
#elif EXAMPLE == 2
      do
        {
        i=i+1;
        x=1.+(i-1)*0.2;
        x2=x*x;
        x3=x*x2;
        printf("     x=%8.2lf  x2=%8.2lf  x3=%8.2lf\n",x,x2,x3);
        } while(i <= 20);
#else
      printf("Error in preprocessor directive\n");
#endif
      exit(0);
}

/* ------------------ C example  No. 3 ----------------------- */
/*          example of call to a function                     */
#include <stdio.h>           /* contains prototype for printf  */
#include <stdlib.h>          /* contains prototype for exit    */
/*  function prototypes:  */
int main(void);
double power(double ,int );

int main()
{
    int i=0;
    double x,y,z1,z2;
    x=2.0;
    y=-3.;
    printf("i,x,y= %d %lf %lf\n",i,x,y);
    for(i=0;i<10;++i)
       { z1= power(x,i);
         z2= power(y,i);
         printf("%d %lf %lf\n",i,z1,z2);   }
    exit(0);
```

```
}
double power(double x, int n)
{
    int i;
    double p=1.0;
    for(i=1;i<=n;++i)
        p*=x;
    return(p);
}

/* ------------------ C example  No. 4 ----------------------
Composed of examples from: Kernighan og Ritchie: "The C programming
language".
The program reads a file named testfil.c and counts the occurrence
of certain words which are
stored in the struct keytab
-----------------------------------------------------------------*/
#include <stdio.h>          /* contains prototype for printf    */
#include <stdlib.h>         /* contains prototype for exit      */
/* definition of symbolic constants:  */
#define MAXWORD 20
#define LETTER 'a'
#define DIGIT  '0'
/*  struct prototype:  */
struct key { char *keyword;  /* pointer to text string        */
             int keycount;   /* int to count occurrences      */
           };
/*  struct definition with initialisation:  */
struct key keytab[] ={{"break",0},
                      {"case",0},
                      {"char",0},
                      {"continue",0},
                      {"default",0},
                      {"unsigned",0},
                      {"while",0}};
/* definition of symbol for number of elements in
   vector keytab:                                    */
#define NKEYS (sizeof(keytab)/sizeof(struct key))
/* definition of pointer to file:  */
FILE *stream;
```

```
/*  function prototypes:  */
int main(void);
int binary(char *,struct key [],int);
int type(int );
int getword(char *,int );
int getchx(void);
int ungetch(int );
/*-----------------------------------------------------------------
main program
-----------------------------------------------------------------*/
int main()
{
    int n,t;                              /* declaration of local */
    char word[MAXWORD];                   /*    variables         */
    stream=fopen("testfil.c","r");
    while((t=getword(word,MAXWORD)) != EOF)
    { if(t==LETTER)
         if((n=binary(word,keytab,NKEYS)) >=0)
             keytab[n].keycount++;
    }
    for(n=0; n<NKEYS; n++)
         if(keytab[n].keycount>0) printf("%4d %s\n",
            keytab[n].keycount,keytab[n].keyword);
    exit(0);
}
/*-----------------------------------------------------------------
function to find word i tab[0]...tab[n-1] if possible
-----------------------------------------------------------------*/
int binary(char *word, struct key tab[], int n)
{
    int low,high,mid,cond;      /* declaration of local var     */
    low=0;
    high=n-1;
    while(low <= high)
       { mid=(low+high)/2;
         if((cond=strcmp(word,tab[mid].keyword)) < 0) high=mid-1;
         else if(cond > 0) low=mid+1;
         else return(mid);
       }
    return(-1);
```

```
}
/*-------------------------------------------------------------
function returning type of input character
-------------------------------------------------------------*/
int type(int c)
{
    if(c>='a' && c<='z' || c>='A' && c<='Z') return LETTER;
    else if(c>='0' && c<='9') return DIGIT;
    else return(c);
}
/*-------------------------------------------------------------
function returning the next word from input file
-------------------------------------------------------------*/
int getword(char *w, int lim)
{
    int c,t;
    if(type(*w++=c=getchx()) != LETTER)
       { *w='\0';';
         return(c);
       }
    while(--lim>0)
       { t=type(*w++=c=getchx());
         if(t != LETTER && t != DIGIT)
            { ungetch(c);
              break;
            }
       }
    *(w-1)='\0';
    return(LETTER);
}
/*-------------------------------------------------------------
definition of symbol BUFSIZE and variables buf[] and bufp, which
will be global, but only in the rest of the file from this point
-------------------------------------------------------------*/
#define BUFSIZE  100
int  buf[BUFSIZE];   /* buffer for ungetch, cannot be declared as
                        char for EBCDIC characters, nor as unsigned
                        char since the program tests whether character
                        is EOF, represented by -1                  */
int bufp=0;          /* points to next free position in buf     */
```

```
/*-----------------------------------------------------------------
function to get character from buffer or from input file
-----------------------------------------------------------------*/
int getchx()
{
    return((bufp>0) ? buf[--bufp] : getc(stream));
}
/*-----------------------------------------------------------------
function to place character in buffer
-----------------------------------------------------------------*/
int ungetch(int c)
{
    if(bufp>=BUFSIZE) printf("ungetch: too many characters");
    else buf[bufp++]=c;
}
/*-----------------------------------------------------------------
end of C example No. 4
-----------------------------------------------------------------*/

/* ------------------ C example  No. 5 -----------------------
cprint.c - program to print a C source file on an IBM laser printer.
Unfortunately the hexadecimal codes for certain special symbols
used by the C compiler do not correspond to the codes an IBM laser
printer requires for the same symbols. The program reads the input
file whose name is specified on the command line when the program
is executed, converts the symbols in question and write a new file.
If the original file was e.g.: minfil.c, the new file will be called
minfil.CPRINT. The program then gives a print command to the system.
The command line to start the program: cprint minfil c [enter].
-----------------------------------------------------------------*/
#include <stdio.h>           /* definitions for IO              */
#include <stdlib.h>          /* definitions for file functions  */

FILE *fpinp,*fpout;          /* pointers to I/O files           */

/*--- function prototypes: -------------------------------------*/
int main(int ,char *[]);
int conversion(int);

/*--- global data:  --------------------------------------------*/
```

```
char infil[80];                  /* name of input file         */
char outfil[80];                 /* name of output file        */
char outfil_ft[]="CPRINT";       /* type of output file        */
char sysarg[80]="lp -c ";        /* argument for system call   */
/*--- data use:  ------------------------------------------------
 sysarg : input to unix
 outfil : output file name (fn.CPRINT)
 infil  : input  file name (fn.ft)
-----------------------------------------------------------------*/
int main(argc,argv)
int argc;            /* number of arguments in call to cprint   */
char *argv[];        /* pointers to arguments                   */
{
    int m,n,t;       /* local variables in main                 */
    char *farg;      /* local pointer to string                 */
/* echo arguments to terminal:                                  */
    n=argc;
    while(--n>=0)                /* loop over fn ft             */
       {
       printf("%4d %s\n",n,argv[n]);   /* print arguments       */
       }
/* construct infil:                                             */
    n=0;
    for(m=1 ; m<argc ; m=m+1)          /* loop over arguments   */
       {
       farg=argv[m];             /* farg pointer to argument    */
       while(*farg != '\0')      /* loop over characters        */
            {
            t=*farg++;           /* get 1 character             */
            infil[n++]=t;        /* store in input file name    */
            }
       infil[n++]='.';           /* replace null byte with '.' */
       }
    infil[--n]='\0';            /* end file name with null byte*/
/* show name for input file on terminal:                        */
    printf("input file name:%s\n",infil);
/* construct outfil:                                            */
    n=0;
    for(m=1 ; m<argc ; m=m+1)                /* as for infil  */
       {
```

```
       if(m != 2) farg=argv[m];
       else farg=outfil_ft;
       while(*farg != '\0')
          {
          t=*farg++;
          outfil[n++]=t;
          }
       outfil[n++]='.';
       }
    outfil[--n]='\0';
/* show name for output file on terminal:                          */
    printf("output file name:%s\n",outfil);
/* construct argument to system call:                              */
    n=0;
    while((t=outfil[n]) != '\0')    /* insert output file name  */
       sysarg[6+n++]=t;             /* in argument to system    */
    sysarg[6+n]='\0';
/* show argument to system on terminal:                            */
    printf("system argument:%s\n",sysarg);
/* open input file:                                                */
    fpinp=fopen(infil,"r");
    if(!fpinp)                      /* check for errors         */
       {
       printf("File does not exist\n");
       exit(0);
       }
/* open output file:                                               */
    fpout=fopen(outfil,"w");
    if(!fpout)                      /* check for errors         */
       {
       printf("Cannot open output file\n");
       exit(0);
       }
    while((t=getc(fpinp)) != EOF)   /* read 1 character from    */
       putc(conversion(t),fpout);   /* file convert is and write*/
                                    /* it in output file        */
                                    /* continue to EOF          */
    fclose(fpinp);                  /* close input file         */
    fclose(fpout);                  /* close output file        */
    t=system(sysarg);               /* call operating system    */
```

```
    if(t != 0)                        /* print error code if any  */
       printf("system call returns:%d\n",t);
    exit(t);
}
/*-----------------------------------------------------------------
Function conversion converts certain hexadecimal characters from
the value demanded by the C compiler to the value demanded by the
laser printer
-----------------------------------------------------------------*/
int conversion(int c)
{
if(c == 0xAD) return(0x9E);    /* ' [ ' */
if(c == 0xBD) return(0x5A);    /* ' ] ' */
if(c == 0xC0) return(0x9C);    /* ' { ' */
if(c == 0xD0) return(0x47);    /* ' } ' */
if(c == 0x6A) return(0x70);    /* ' | ' */
if(c == 0x5A) return(0x4F);    /* ' ! ' */
if(c == 0x7B) return(0x4A);    /* ' # ' */
if(c == 0x5B) return(0x67);    /* ' $ ' */
if(c == 0x7C) return(0x80);    /* ' @ ' */
if(c == 0xA1) return(0x5F);    /* ' ~ ' */
return(c);
}
/*-----------------------------------------------------------------*/
```

Bibliography

[1] Brian W. Kernighan and Dennis M. Ritchie, "The C programming language", 2.ed., Prentice Hall Software Series, 1988.

[2] Robin Jones and Ian Stewart, "The art of C programming", Springer Verlag, 1987.

[3] Greg Comeau, "A guide to understanding even the most complex C declarations", Microsoft Systems Journal, Vol.3, No.5, September 1988, pp. 10-20.