

**Stefano Quer**

# **Advanced Programming and Problem-Solving Strategies in C**

**Part II: Algorithms and Data  
Structures**

**Second Edition**

**CLUT**

**Stefano Quer**

# **Advanced Programming and Problem-Solving Strategies in C**

**Part II: Algorithms and Data  
Structures**

**Second Edition**

**CLUT**

# Contents

<b>1</b>	<b>C Essentials and Problem Solving Basis</b>	<b>1</b>
1.1	Program structure .....	2
1.1.1	Blocks .....	2
1.1.2	Identifiers .....	2
1.1.3	Comments .....	2
1.1.4	Preprocessor Directive .....	4
1.1.5	Type and Cast .....	4
1.1.6	Typedef .....	5
1.1.7	Sizeof .....	5
1.1.8	Variables and Constants .....	5
1.1.9	Input .....	6
1.1.10	Output .....	6
1.2	Conditional Constructs .....	7
1.2.1	If and if-else .....	7
1.2.2	Switch .....	8
1.2.3	Conditional Expressions .....	8
1.3	Iteration Essentials .....	8
1.3.1	While .....	9
1.3.2	For .....	9
1.3.3	Do-While .....	9
1.3.4	Stop and Re-cycle .....	12
1.4	Board .....	13
1.5	Diamond .....	13
1.6	Arrays .....	14
1.6.1	Array definition and initialization .....	14
1.6.2	Array access .....	17
1.7	Binary Conversions .....	20
1.8	Rule-out Zero Values .....	22
1.9	Sub-Array Search .....	24
1.10	Horizontal Histogram .....	
1.11	Vertical Histogram .....	

1.12	Characters and Strings . . . . .	25
1.13	Anagram Verification . . . . .	26
1.14	String Rotation . . . . .	29
1.15	Spiral Print . . . . .	31
1.16	Local Minimum Search . . . . .	32
1.17	Sub-matrix Search . . . . .	34
1.18	Functions . . . . .	36
1.18.1	Definition and Declaration . . . . .	36
1.18.2	Function Call . . . . .	37
1.18.3	Arguments to the main program . . . . .	37
1.18.4	Function Exit . . . . .	38
1.18.5	String Manipulation . . . . .	38
1.18.6	Character Manipulation . . . . .	39
1.19	Sub-string Substitution . . . . .	41
1.20	Average Values . . . . .	43
1.21	Files . . . . .	43
1.21.1	Define a File Pointer Object . . . . .	44
1.21.2	Open a File . . . . .	44
1.21.3	Check a File Pointer . . . . .	44
1.21.4	Input (Output) from (to) File . . . . .	45
1.21.5	Close a File . . . . .	45
1.22	Text Alignment . . . . .	46
1.23	File Format . . . . .	48
1.24	Structures . . . . .	48
1.24.1	Reference to fields . . . . .	49
1.24.2	Advanced Usage . . . . .	49
1.25	Space Points . . . . .	51
1.26	Flights . . . . .	54
1.27	Sudoku . . . . .	59
<b>2</b>	<b>Memory, Variables, and Pointers</b>	<b>59</b>
2.1	Memory . . . . .	60
2.2	Variables and Padding . . . . .	62
2.3	Pointer Variables . . . . .	62
2.4	Pointer Operators . . . . .	62
2.4.1	The Indirection Operator . . . . .	63
2.4.2	The Address Operator . . . . .	64
2.4.3	Relationship between Indirection and Address Operators . . . . .	65
2.5	NULL and void Pointers . . . . .	65
2.6	Pointer Arithmetic . . . . .	66
2.7	Relationship between Pointers and Arrays . . . . .	71
2.7.1	Relationship between Pointers and Strings . . . . .	73
2.8	Relationship between Pointers and Structures . . . . .	74
2.9	Array of pointers . . . . .	75
2.10	Local arrays with variable size . . . . .	77
<b>3</b>	<b>Dynamic Memory Allocation</b>	<b>77</b>
3.1	Dynamic Memory Allocation . . . . .	77

3.1.1	Function <code>malloc</code> . . . . .	78
3.1.2	Function <code>calloc</code> . . . . .	79
3.1.3	Function <code>realloc</code> . . . . .	80
3.1.4	Function <code>free</code> . . . . .	81
3.2	Dynamically Allocated 1D Arrays . . . . .	82
3.2.1	Dynamic Memory Allocation and Modularity . . . . .	86
3.2.2	Dynamic String Allocation . . . . .	88
3.2.3	Dynamic Arrays of Structures . . . . .	90
3.3	Median Point . . . . .	93
3.4	Analytic Index . . . . .	95
3.5	Blood Donations . . . . .	99
3.6	String Evolution . . . . .	102
3.7	Broken lines . . . . .	106
3.8	Dynamically Allocated 2D Arrays . . . . .	110
3.8.1	2D Arrays Allocated as 1D Arrays . . . . .	110
3.8.2	2D Arrays Allocated as 2D Arrays . . . . .	111
3.8.3	Dynamic Memory Allocation and Modularity . . . . .	114
3.8.4	2D Arrays Summary . . . . .	116
3.9	Matrix Multiplication . . . . .	120
3.10	String Sort . . . . .	123
3.11	Cyclist Training . . . . .	126
3.12	Hotel Reservations . . . . .	130
3.13	Multi Merge . . . . .	134
3.14	Cross-puzzle . . . . .	137
3.15	Snakes . . . . .	141
3.16	Hospitals . . . . .	144
3.17	Counter-intelligence Agency . . . . .	149
3.18	Political Elections . . . . .	154
3.19	Library . . . . .	158
<b>4</b>	<b>Lists</b> . . . . .	<b>169</b>
4.1	General concepts . . . . .	169
4.2	Simple List . . . . .	170
4.2.1	C Representation . . . . .	170
4.2.2	Visit . . . . .	171
4.2.3	Search . . . . .	172
4.2.4	Extraction . . . . .	173
4.2.5	Insertion . . . . .	174
4.2.6	Free . . . . .	175
4.3	Lists with Sentinel and Dummy Elements . . . . .	176
4.4	Ordered List . . . . .	177
4.4.1	Search . . . . .	178
4.4.2	Extraction . . . . .	180
4.4.3	Insertion . . . . .	181
4.5	Circular Lists . . . . .	181
4.6	Doubly-Linked Lists . . . . .	182
4.7	List of Lists . . . . .	182
4.8	Ordered List of Integers . . . . .	182

		187
4.9	Words Frequency .....	190
4.10	Employees .....	193
4.11	Safari .....	197
4.12	List Sorting .....	202
4.13	Overlapped Lists .....	206
4.14	Formula 1 .....	210
4.15	Memory Allocator .....	213
4.16	Bi-linked Lists .....	216
4.17	Polygons .....	220
4.18	Lonely Hearts Club .....	225
4.19	Video club .....	225
<b>5</b>	<b>Recursion basis and simple recursive problems</b>	<b>231</b>
5.1	Recursion: Basic Concepts .....	231
5.1.1	Recurrences .....	232
5.2	Mutual Recursion .....	233
5.3	Recursion Tree and Stack .....	234
5.4	Recursion versus Iteration .....	235
5.5	Divide and Conquer .....	236
5.6	Factorial computation .....	237
5.7	Fibonacci computation .....	238
5.8	Power Computation .....	241
5.9	Array Visit .....	242
5.10	List Visit .....	244
5.11	Greatest Common Divisor .....	246
5.12	Map regions .....	248
5.13	Prefix (Polish) Notation .....	250
<b>6</b>	<b>Classic recursive problems</b>	<b>253</b>
6.1	Binary Search .....	253
6.2	Merge sort .....	255
6.3	Quick sort .....	258
6.4	Combinatorics .....	262
6.4.1	The multiplication method .....	262
6.4.2	Simple arrangements .....	262
6.4.3	Arrangements with repetitions .....	264
6.4.4	Simple permutation .....	265
6.4.5	Permutations with repetitions .....	267
6.4.6	Simple combinations .....	268
6.4.7	Combinations with repetitions .....	269
6.5	Binary Numbers .....	270
6.6	Football Pool .....	272
6.7	Anagrams Generation .....	273
6.8	The power-set .....	276
6.9	Partition of a set .....	279
6.10	The Knight Tour .....	281
6.11	The Eight Queen Problem .....	284
		288

<b>7 Recursion-based advanced problem solving tasks</b>	<b>301</b>
7.1 Towers of Hanoi . . . . .	301
7.2 Determinant . . . . .	304
7.3 The Magic Square . . . . .	307
7.4 The maze . . . . .	314
7.5 The Sudoku Puzzle . . . . .	319
7.6 Cross Checkers . . . . .	322
7.7 Bookshelves . . . . .	324
7.8 The knapsack problem . . . . .	329
7.9 Dominoes . . . . .	331
7.10 Shipping Firm . . . . .	335
7.11 Television programs . . . . .	338
7.12 The Hitori Puzzle . . . . .	342
7.13 Soccer competition . . . . .	345
7.14 String concatenation . . . . .	349
<b>8 Modularity, Multi-file Applications and Abstract Data Types</b>	<b>353</b>
8.1 Variables and Function Attributes . . . . .	353
8.1.1 Storage Class and Duration . . . . .	353
8.1.2 Scope . . . . .	359
8.1.3 Linkage . . . . .	360
8.1.4 A complete example . . . . .	360
8.2 Multiple-Source-File Programs . . . . .	362
8.2.1 Program Development Environment . . . . .	362
8.2.2 Memory Layout of C Programs . . . . .	364
8.3 Multiple-File Applications . . . . .	366
8.3.1 Functions and Variables Rules . . . . .	367
8.3.2 Further Hints . . . . .	369
8.3.3 Once-Only Headers . . . . .	370
8.4 Abstract Data Types (ADTs) . . . . .	372
8.4.1 ADT Summary Example . . . . .	374
<b>9 ADTs and Classic Data Structures</b>	<b>383</b>
9.1 A LIFO Stack . . . . .	383
9.1.1 Array implementation . . . . .	384
9.1.2 List Implementation . . . . .	385
9.2 A FIFO Queue . . . . .	387
9.2.1 Array Implementation . . . . .	388
9.2.2 List Implementation . . . . .	390
9.3 Functions as Function Arguments . . . . .	393
9.4 Macro Definition . . . . .	395
9.5 A Utility Library . . . . .	396
9.6 A First Class ADT Library LIFO Stack . . . . .	400
9.7 A First Class ADT Library FIFO Queue . . . . .	411
9.8 A List Library . . . . .	418
9.9 Date . . . . .	424
9.10 Plane Points . . . . .	426
9.11 Modular Binary Search . . . . .	430

9.12	Sorting Library . . . . .	432
9.13	Post-fix Computations . . . . .	446
<b>Index</b>		<b>449</b>
<b>Bibliography</b>		<b>453</b>

# Chapter 1

## C Essentials and Problem Solving Basis

Good software engineering is important, as it is key to making more manageable the task of developing the larger and more complex software systems we need. High performance is important as well, as it is key to realizing the systems of the future that will place ever greater computing demands on hardware. Unfortunately, these goals are often at odds with one another. For the vast majority of applications we are going to analyze in this book, we will consider good software engineering more important than programming for high performance.

In this chapter we will start this process by revising all C language constructs which are at the base of all subsequent chapters and topics.

### 1.1 Program structure

Each program has the following basic structure:

```
<header inclusion>

int main (void) {
    <definitions>

    <instructions>

    return <val>;
}
```

Blank lines, space characters and tab characters (i.e., “tabs”) can be used to make programs easier to read. Together, these characters are known as white space. White-space characters are normally ignored by the compiler.

The line int main (void) indicates that the program includes a “main” function. C programs contain one or more functions, one of which must be main . Every program in C begins executing at the function main. Functions can return information. The keyword int to the left of main indicates that main “return” an integer (whole-number) value. Functions also can receive information when they are called upon to execute. The void in parentheses here means that main does not receive any information.

### 1.1.1 Blocks

A left brace, { . begins the body of every function. A corresponding right brace } ends each function. All statements contained within a pair of braces are called *compound statements* or *blocks* or *local blocks*. Local blocks can have any number of statements and are important program unit in C.

```
{ ... this is an instruction block ... }
```

For example, variables declared in a local block have local scope, i.e., they can be accessed within the block only.

### 1.1.2 Identifiers

In C, we use identifiers as names for user-defined constants, variables, types, and functions. Each identifier in a program may have other attributes, such as storage class, storage duration, scope and linkage. Several identifiers are reserved keywords. These keywords have special meaning to the C compiler, so you cannot use them as identifiers such as object names.

### 1.1.3 Comments

Comments are inserted to document programs and improve program readability. Comments do not cause the computer to perform any action when the program is run. Comments are ignored by the C compiler and do not cause any machine-language object code to be generated. In C there are two possibilities to insert comments: Multi-line comments (in which everything from /\* on the first line to \*/ at the end of the last line is a comment)

```
/*
 * C Style (multi line)
 */
```

and

```
// C++ Style (single line)
```

i.e., comments beginning with // that are shorter and eliminate common programming errors that occur with the previous comment style.

### 1.1.4 Preprocessor Directive

The C compiler has a preprocessor built into it. Lines beginning with # are processed by the preprocessor before compilation. They are called *preprocessing directives*.

#### ***The include directive***

The #include directive tells the preprocessor to insert the contents of another file into the source code at the point where the #include directive is found. Include directives are typically used to include the C header files for C functions that are held outside of the current source file. The syntax for this directive is either:

or

```
#include "header_file_name"
```

The difference between these two syntaxes is subtle but important. If a header file is included within `<>`, the preprocessor will search a predetermined directory path to locate the header file. If the header file is enclosed in `" "`, the preprocessor will look for the header file in the same directory as the source file.

Useful header files are the following: `stdio.h`, `stdlib.h`, `string.h`, `math.h`, `limits.h`, `float.h`, `ctype.h`, `assert.h`.

### **The `#define` directive**

The `#define` directive allows the definition of macros within your source code. These macro definitions allow constant values to be declared for use throughout your code. Macro definitions are not variables and cannot be changed by your program code like variables. You generally use this syntax when creating constants that represent numbers, strings or expressions in a symbolic way. Symbols can be easily changed.

Normally, all `#define` lines are placed at the beginning of the file. Notice that most C programmers define their constant names in uppercase, but it is not a requirement of the C Language. The syntax for creating a constant is:

```
#define <id> <value>
```

or

```
#define <id> <expression>
```

where `<id>` is the name of the constant, `value` is the value of the constant, and `expression` is the expression whose value is assigned to the constant. The expression must be enclosed in parentheses if it contains operator. See Section 9.4 for further details.

**Example 1.1** The following statements define an integer constant `value` and the  $\pi$  real constant:

```
#define SIZE 10
#define pi 3.1415
```

The following `#define` statements define a macro (i.e., `max`) which receives two parameters (`a` and `b`) and it returns their sum:

```
#define sum(a,b) (a+b)
```

### **Conditional compilation**

Conditional compilation enables the programmer to control the execution of preprocessor directives and the compilation of program code. Each conditional preprocessor directive evaluates a constant integer expression. The conditional preprocessor construct is much like the `if` selection statement:

```
#if <constant_value>
...
#endif
or
#if <constant_value>
...
#else
...
#endif
```

In this last case if the `constant_value` evaluates to true the of code between `#if` and `#else` is inserted, otherwise the piece of code between `#else` and `#endif` is inserted in the code.

The following example shows a possible technique and use of this strategy.

**Example 1.2** Let us suppose we want to insert or not insert a specific piece of code depending on whether we want to debug our code or not. For example, we may want to enable or disable specific printing instruction depending on the level of verbosity we may want to obtain running the program. One solution to this issue is shown by the following piece of code.

```
#define DEBUG 1

#if DEBUG
    // This piece of code is inserted when DEBUG is true (e.g., 1)
    ...
#else
    // This piece of code is inserted when DEBUG is false (e.g., 0)
    ...
#endif
...
```

By modifying the value of the constant DEBUG (and then we recompile the program) we enable or disable specific section of codes within our program. The main difference between using a standard if construct is that in that case the undesired code is not even inserted in the executable file, then the executable file is somehow smaller and faster.

**Example 1.3** In the following piece of code:

```
#if !defined(MY_CONSTANT)
#define MY_CONSTANT 0
#endif
```

if MY\_CONSTANT is defined, i.e., it has already appeared in an earlier #define directive, the expression defined(MY\_CONSTANT) evaluates to true and !defined(MY\_CONSTANT) evaluates to false. Otherwise defined(MY\_CONSTANT) evaluates to false and !defined(MY\_CONSTANT) evaluates to true.  
 If !defined(MY\_CONSTANT) is true, the #if is entered and the constant defined.  
 If !defined(MY\_CONSTANT) is false, the body of the #define directive is skipped.

Notice that every #if construct ends with #endif.

Directives #ifdef and #ifndef are shorthand for #if defined(name) and #if !defined(name).

### 1.1.5 Type and Cast

C has a concept of “data types” which are used to define a variable before its use. The definition of a variable will assign storage for the variable and define the type of data that will be held in the location. The value of a variable can be changed any time. C has the following basic built-in data-types: char, int, float, double, void.

Please note that there is not a Boolean data type. Less useful types are: long, short, signed, unsigned.

Type casting is a way to convert a variable from one data type to another data type.  
 (<new\_type>) <expression>

which convert the result of the expression into the type new\_type.

### 1.1.6 Typedef

The keyword typedef provides a mechanism for creating synonyms (or aliases) for previously defined data types. Names for structure types are often defined with typedef.

to create shorter type names. The statement

**typedef** <type> <synonym>;

defines the new type name <synonym> as a synonym for type type. C programmers often use **typedef** to define a structure type, so a structure tag is not required.

### 1.1.7 **Sizeof**

The unary **sizeof** operator finds (and returns) the number of bytes needed to store and object. An expression of the form:

**sizeof** (<type>)

returns the size in bytes of the object representation of the indicated type. The same action can be performed by:

**sizeof** (<variable\_or\_type>)

**Example 1.4** In the following piece of code:

```
int 11, 12;
char c;
...
11 = sizeof (char);
12 = sizeof (c);
```

both 11 and 12 will be 1.

### 1.1.8 **Variables and Constants**

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable. A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

<type> <variable\_name\_1>, <variable\_name\_2>, ...;

where <type> must be a valid C data type.

A C constant is usually just the written version of a number. In C constants are usually defined with the **define** construct, but the **const** construct is also possible, such as:

<type> **const** <variable\_name> = <value>;

Notice that: 5 is an integer (**int**), '5' is a character (**char**), "5" is a string (i.e., it is an array of two characters).

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C allows both binary (=, +, -, \*, /,), and unary arithmetic operators (++, --). Moreover it allows relational operators (>, >=, ==, !=, <=, <), and logical (&&, ||). Furthermore it is also possible to use shift operators (i.e., >> and <<), and bit-field operators (~ NOT, | OR, & AND, ^ XOR).

### 1.1.9 **Input**

Most input and output operations can be performed using function **scanf** and **printf**.

Precise input formatting can be accomplished with `scanf`. Every `scanf` statement contains a format control string that describes the format of the data to be input. The format control string consists of conversion specifiers and literal characters:

```
#include <stdio.h>
scanf ("%<format>", <variable_pointer_1>, <variable_pointer_2>, ...);
```

Useful format: `%d, %f, %c, %s`.

### 1.1.10 Output

Function `int printf` sends formatted output to standard output:<

```
#include <stdio.h>
printf ("%<format>", <expression_1>, <expression_2>, ...);
```

where `<format>` is the string that contains the text to be written to standard output. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Useful format specifications are: `\n, \t`.

## 1.2 Conditional Constructs

Statements in a program are normally executed in sequence. However, most programs require alteration of the normal sequential flow of control. The `if`, `if-else`, `switch`, and conditional expression statements provide the mechanisms to generate alternative action flows.

These constructs typically require the evaluation of logical expressions, expressions that the programmer thinks of as being either true or false. In C, any non-zero value is considered to represent true, and any zero value is considered to represent false.

### 1.2.1 If and if-else

We might make a decision in a program using the `if` statement. Its simplest form is the following one:

```
if (<expr>
    <instr>;
```

The logical expression given inside the brackets after `if` is evaluated first. If `<expr>` is nonzero (true), then statement `<instr>` is executed; otherwise, it is skipped. It is important to recognize that any `if` statement, even though it contains an instruction part, is itself a single statement.

Moreover, the `if` statement can be extended with an `else` section:

```
if (<expr>
    <instr1>;
else
    <instr2>;
```

If the expression is true, then the statement `<instr1>` is executed. If the expression is false, the statements `<instr2>` is executed.

The same statements can be better written as:

```
if (<expr> {
    <instr1>;
} else {
    <instr2>;
}
```

where each single statement `<instr1>` and `<instr2>` can be substituted by any sequence of statements (i.e., a block of instructions).

Conditions in `if` statements are formed by using the equality operators and relational operators. In C, a logical expression such as `(i <= 5)` has value 1 (integer value, true) if `i` is less than or equal to 5, and has a value 0 (integer value, false) otherwise.

### 1.2.2 Switch

The `switch` construct is a multi branching control statement.

```
switch (<expr>) {
    case <const_1>: <instr_1>;
        break;
    case <const_2>: <instr_2>;
        break;...
    case <const_i>: <instr_i>;
        break;
    ...
    case <const_N>: <instr_N>;
        break;
    default : <instr_other>;
        break;
}
```

The execution of a switch statement begins by evaluating the expression inside the `switch` keyword brackets. The expression should be an integer (1, 2, 100, 57, etc.) or a character constant like 'a', 'b' etc. This expression's value is then matched with each case values. There can be any number of case values inside a switch statements block. If first case value is not matched with the expression value, program control moves to next case value and so on. When a case value matches with expression value, the statement (or statements) that belong to a particular case value is (are) executed. Notice that last set of lines that begins with `default`. The word `default` is a keyword in C/C++. When used inside switch block, it is intended to execute a set of statements, if no case values matches with expression value. So if no case values are matched with expression value, the set of statements that follow `default`: will get executed.

### 1.2.3 Conditional Expressions

The conditional operators ? or ternary operator statement is used in the following expressions: as follows:

$$<\text{expression1}> ? <\text{expression2}> : <\text{expression3}>;$$

It works as follows. The first operand `<expression1>` is implicitly converted to a Boolean value and it is evaluated. If it evaluates to true (not zero), the second operand is evaluated. If it evaluates to false (zero), the third operand is evaluated. The result of the conditional operator is the result of whichever operand is evaluated, the second or the third. Only one of the last two operands is evaluated.

#### Example 1.5 The code

```
if (x < y)
    z = x;
else
    z = y;
```

assign to  $z$  the minimum of  $x$  and  $y$ . This can be accomplished by:

```
 $z = (x < y) ? x : y;$ 
```

**Example 1.6** The following line define a macro which returns the maximum between two values:

```
#define MAX(x,y) ((x>y)?x:y)
```

which can be called as  $\text{MAX}(2, 3)$ ,  $\text{MAX}(a, b)$ , etc.

## 1.3 Iteration Essentials

Most programs involve repetition, or looping. A loop is a group of instructions the computer executes repeatedly while some loop-continuation condition remains true. There are usually two kind of control constructs: Counter-controlled repetition, and Sentinel-controlled repetition.

Counter-controlled repetition is sometimes called definite repetition because we know in advance exactly how many times the loop will be executed.

Sentinel-controlled repetition is sometimes called indefinite repetition because it is not known in advance how many times the loop will be executed.

Such controls are realized in C using the following constructs.

### 1.3.1 While

The general form of a while statement is

```
{ while (<expr>) {
    <instr>;
}
```

where  $<\text{instr}>$  is either a simple statement or a compound statement. When the while construct is executed,  $<\text{expr}>$  is evaluated. If it is nonzero (true), then statement is executed and control passes back to the beginning of the while loop. This process continues until  $\text{expr}$  has value 0 (false). At this point, control passes on to the next statement.

### 1.3.2 For

The for statement is an entry controlled loop. Traditionally, the difference between while and for is in the number of repetitions. The for loop was originally used for counter-controlled repetitions, whereas while was used for sentinel-controlled repetitions. This difference does not exist in C language.

```
{ for (<instr1>; <expr>; <instr2>) {
    <instr>;
}
```

The program control enters the for loop. At first it execute the statements given as initialization statements  $<\text{instr1}>$ . Then the condition statement  $<\text{exp}>$  is evaluated. If the condition is true, then the block of statements inside curly braces is executed. After executing curly brace statements fully, the control moves to the “iteration” statements  $<\text{instr2}>$ . After executing iteration statements, control comes back to condition statements. Condition statements are evaluated again for true or false. If true the curly brace statements are executed. This process continues until the condition turns false.

### 1.3.3 Do-While

Unlike while, do-while is an exit controlled loop:

```
do {  
    <instr>;  
} while (<expr>);
```

Here the set of statements inside braces are executed first. The condition inside while is checked only after finishing the first time execution of statements inside braces. If the condition is true, then statements are executed again. This process continues as long as condition is true. Program control exits the loop once the condition turns false.

#### 1.3.4 Stop and Re-cycle

The `break` statement terminates the loop (any `for`, `while` and `do-while`) immediately when it is encountered. It is often used with decision making statement such as `if`.

The `continue` statement skips some statements inside the loop. Like the previous one, it is often used with decision making statement such as `if`.

## 1.4 Board

## Specifications

Write a program able to:

- ▷ Read an integer value  $n$  from standard input.
  - ▷ Draw a board of size  $n$  as represented in the following examples.

**Example 1.7** Let  $n$  be equal to 5. The program has to draw (using characters “-”, “|”, “#”, white spaces, and “new lines”):

| # # # |  
| # # # |  
| # # # |  
| # # # |  
| # # # |

**Example 1.8** Let  $n$  be equal to 10. The program has to draw (using characters “-”, “/”, “#”, white spaces, and “new lines”):

A 10x10 grid of hash symbols (#) arranged in ten rows and ten columns, centered on the page.

## Solution 1

This first version uses `while` cycles to solve the problem. To print the right symbol at each iteration we use the following strategy. Within the integer domain, the division between two integer numbers  $N$  (numerator) and  $D$  (denominator) produces a quotient  $Q$  and a remainder  $R$ , i.e.;

$$\frac{N}{D} = Q + \frac{R}{D}$$

In C, quotient and remainder are generated by the operators `/` and `%`, respectively. For example, the following expressions hold:

$9/2 = 4$ $10/2 = 5$ $17/4 = 4$	$9\%2 = 1$ $10\%2 = 0$ $17\%4 = 1$
---------------------------------------	--

As a consequence, we use operation `r%2` and `c%2` to understand which symbol we have to print.

```

1 #include <stdio.h>
2
3 int main(void) {
4     int n, r, c;
5
6     printf("Number of frames: ");
7     scanf("%d", &n);
8
9     /* First line */
10    r=0;
11    while (r<=n+1) {
12        printf ("\"");
13        r++;
14    }
15    printf ("\n");
16
17    r=1;
18    while (r<=n) {
19        printf ("|");
20        c=1;
21        while (c<=n) {
22            /* Check element:
23             odd on odd rows, and even even rows */
24            if ((r%2==1 && c%2==1) || (r%2==0 && c%2==0)) {
25                printf ("#");
26            } else {
27                printf (" ");
28            }
29            c++;
30        }
31        printf ("|\n");
32        r++;
33    }
34
35    /* Last line */
36    r=0;
37    while (r<=n+1) {
38        printf ("\"");
39        r++;

```

```

40     }
41     printf ("\n");
42
43     return 1;
44 }
```

## Solution 2

The second solution includes the following variants with respect to the previous one:

- ▷ The input value `n` is received from the command line, not from standard input. For more details on main arguments see Section 1.18.3 which summarizes the topic with some more details.
- ▷ The `while` constructs are substitutes by `for` ones.
- ▷ Output messages are differentiated to be placed on standard output or on standard error (for error or warning messages). This may seem useless, but it is good practice for more complex programs which generate substantial output and warning messages.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[]) {
5     int n, r, c;
6
7     /* Get program argument */
8     if (argc!=2) {
9         /* Use stdout and stderr for controlled output */
10        fprintf (stderr, "Run as: <pgrm> integerValue\n");
11        return (0);
12    }
13    /* Conversion function: from ASCII to Integer */
14    n = atoi (argv[1]);
15
16    /* Use for cycle to be more compact */
17    for (r=0; r<=n+1; r++) {
18        fprintf (stdout, "-");
19    }
20    fprintf (stdout, "\n");
21
22    for (r=1; r<=n; r++) {
23        fprintf (stdout, "|");
24        for (c=1; c<=n; c++) {
25            if ((r%2==1 && c%2==1) || (r%2==0 && c%2==0)) {
26                fprintf (stdout, "#");
27            } else {
28                fprintf (stdout, " ");
29            }
30        }
31        fprintf (stdout, "|\\n");
32    }
33
34    for (r=0; r<=n+1; r++) {
35        fprintf (stdout, "-");
36    }
37    fprintf (stdout, "\n");
38 }
```

```
39     return 1;
40 }
```

## 1.5 Diamond

### Specifications

Write a program able to:

- ▷ read an odd integer value  $n$  from the program command line.
- ▷ draw a board of size  $n$  with an inscribed diamond as represented in the following examples.

**Example 1.9** Let  $n$  be equal to 5. The program has to draw (using characters “-”, “+”, and “new lines”):

```
--+--
-+-+-
+---+
-+-+-
--+--
```

**Example 1.10** Let  $n$  be equal to 7. The program has to draw (using characters “-”, “+”, and “new lines”):

```
----+----
---+-+-
-+---+-
+-----+
-+---+-
--+-+-
----+--
```

### Solution

In the following version note the use of the “%” and the “/” operators applied between integer values. Moreover, notice the use of the two constants EXIT\_SUCCESS and EXIT\_FAILURE as return values within `exit` and `return` instructions.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[]) {
5     int n, r, c, left, right;
6
7     /* Get program argument */
8     if (argc!=2) {
9         /* Use stdout and stderr for controlled output */
10        fprintf (stderr, "Run as: <pgm> oddIntegerValue\n");
11        return (EXIT_FAILURE);
12    }
13
14    /* Conversion function: from ASCII to Integer */
15    n = atoi (argv[1]);
16    if (n%2 == 0) {
```

```

17     fprintf (stderr, "Input value has to be odd.\n");
18     return (EXIT_FAILURE);
19 }
20
21 /* Explicit cast operation (int) ... redundant */
22 left = right = (int) (n/2);
23 for (r=0; r<n; r++) {
24     for (c=0; c<n; c++) {
25         if (c==left || c==right) {
26             fprintf (stdout, "+");
27         } else {
28             fprintf (stdout, "-");
29         }
30     }
31     if (r < ((int) (n/2))) {
32         left--;
33         right++;
34     } else {
35         left++;
36         right--;
37     }
38     fprintf (stdout, "\n");
39 }
40
41 return EXIT_SUCCESS;
42 }
```

## 1.6 Arrays

In programming, an array is a way of storing several items (such as integers). These items must have the same type (only integers, only strings, etc.) because an array can not store different kinds of items. Every item in an array has a number so the programmer can get the item by using that number. This number is called the index. In C language, the first item has index 0.

### 1.6.1 Array definition and initialization

To define an array, a programmer specifies the type of the elements and the number of elements required by an array. For single-dimensional arrays the definition is the following one:

`<type> <array_name>[<array_size>];`

The `array_size` must be an integer constant (a numeric or a symbolic constant) greater than zero<sup>1</sup>. Type can be any valid C data type. For n-dimensional arrays the definition is the following one:

`<type> <array_name>[<array_size_1>] [<array_size_2>] ... [<array_size_n>];`

Array can be initialized using a single statement as follows:

`<type> <array_name>[<array_size>] = {<value_0>, <value_1>, ...};`

---

<sup>1</sup> Notice, that in this book we will never allow `array_size` to be a variable, i.e., to assume a value defined only at run-time. This case will be explicitly managed through dynamic memory allocation. In other words, in all cases `array_size` will be a value known at compilation time.

The number of values between braces {} cannot be larger than the number of elements that we declare for the array between square brackets [ ]. If you omit the size of the array, an array just big enough to hold the initialization is created. For multi-dimensional arrays initialization is similar:

```
<type> <array_name>[<array_size_1>][<array_size_2>] =
{ {<value_0_0>, <value_0_1>, ...},
  {<value_1_0>, <value_1_1>, ...},
  ...
};
```

### 1.6.2 Array access

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array:

`<array_name>[<index>]`

and for multi-dimensional arrays:

`<array_name>[<index1>]...[<indexN>]`

## 1.7 Binary Conversions

### Specifications

Write a C program able to read a positive integer number from standard input and to print out its corresponding binary, octal, and hexadecimal forms.

**Example 1.11** Let us suppose that the input number is  $n$ . The corresponding binary number can be found using the *division* method, i.e., at each step:

- ▷ A bit *bit* of the binary number is given by taking the remainder of the division of the number  $n$  by the base 2, i.e.,  $bit = n \% 2$ .
- ▷ The number is substituted by the quotient of the number divided by the base 2, i.e.,  $n = n / 2$ .

If  $n = 23_{10}$  we have:

$$\begin{array}{r} 23 : 2 \\ 1 \overline{)11} : 2 \\ 1 \overline{)5} : 2 \\ 1 \overline{)2} : 2 \\ 0 \overline{)1} : 2 \\ 1 \overline{)0} \end{array}$$

As the first bit is the least significant one, the binary number is then  $10111_2$ .  
A similar technique can be used to find the octal number  $27_8$ , and the hexadecimal one  $17_{16}$ . For example to convert  $n_{10}$  to the base 8, we have:

$$\begin{array}{r} 23 : 8 \\ 7 \overline{)2} : 8 \\ 2 \overline{)0} \end{array}$$

**Example 1.12** Let us suppose  $n = 40_{10}$ . We can compute:

$$\begin{array}{r}
 40 : 2 \\
 0 \mid 20 : 2 \\
 0 \mid 10 : 2 \\
 0 \mid 5 : 2 \\
 1 \mid 2 : 2 \\
 0 \mid 1 : 2 \\
 1 \mid 0
 \end{array}$$

That is:  $101000_2$  in binary format. This value can then be used to obtain the numbers in base 8 by grouping bits in group of 3 starting from the least significant bit, i.e.,

$$101000_2 = 101\ 000 = 50_8$$

Analogously, we can obtain the number base 16:

$$101000_2 = 10\ 1000 = 0010\ 1000 = 28_{16}$$

### Solution 1

This solution converts the decimal number into base 2, 8, and 16 directly adopting three distinct iteration on the original number  $n$ . An array is used to store hexadecimal digits 'A'÷'B'. Notice that all numbers are printed-out from the Least Significant Bit or Digit (LSB or LSD).

```

1 #include <stdio.h>
2
3 int main(void) {
4     int n, tmp, bit, oct, hex;
5     char c[] = {'A', 'B', 'C', 'D', 'E', 'F'};
6
7     /* read number */
8     do {
9         printf("Decimal number (>=0): ");
10        scanf("%d", &n);
11        if (n < 0) {
12            printf("Negative value, re-read!\n");
13        }
14    } while (n < 0);
15
16    /* compute the binary number (from LSB) */
17    tmp = n;
18    printf("Binary number (from LSD) = ");
19    do {
20        bit = tmp % 2;
21        printf("%d", bit);
22        tmp = tmp / 2;
23    } while (tmp != 0);
24    printf("\n");
25
26    /* compute the octal number (from LSD) */
27    tmp = n;
28    printf("Octal number (from LSD) = ");
29    do {
30        oct = tmp % 8;
31        printf("%d", oct);
32        tmp = tmp / 8;
33    } while (tmp != 0);

```

```

34     printf("\n");
35
36     /* compute the hexadecimal number (from LSD) */
37     tmp = n;
38     printf("Hexadecimal number (from LSD) = ");
39     do {
40         hex = tmp % 16;
41         if (hex>=10) {
42             printf("%c", c[hex-10]);
43         } else {
44             printf("%d", hex);
45         }
46         tmp = tmp / 16;
47     } while (tmp != 0);
48     printf("\n");
49
50     return 0;
51 }
```

## Solution 2

In this solution we convert the number to  $b = 2$ , and then from this representation to  $b = 8$  and  $b = 16$ . To do that, we store bits in an array. We suppose integer numbers stored on 64 bits, then we allocate an array of 64 elements. This strategy also allows us to print-out numbers in a more natural form, i.e., from the Most Significant Bit or Digit.

```

1 #include <stdio.h>
2
3 /* Max number of digits in b=2, 8, 16 */
4 #define N_B 64
5 #define N_O 22
6 #define N_H 16
7
8 int main(void) {
9     int n, nb, no, nh, i, j, p;
10    int bit[N_B], oct[N_O], hex[N_H];
11    char c[] = {'A', 'B', 'C', 'D', 'E', 'F'};
12
13    /* read number */
14    do {
15        printf("Decimal number (>=0): ");
16        scanf("%d", &n);
17        if (n < 0) {
18            printf("Negative value, re-read!\n");
19        }
20    } while (n < 0);
21
22    /* compute the binary number (from MSB) */
23    printf("Binary number (from MSB) = ");
24    nb = 0;
25    do {
26        bit[nb] = n % 2;
27        n = n / 2;
28        nb++;
29    } while (n != 0);
30    for (i=nb-1; i>=0; i--) {
31        fprintf(stdout, "%d", bit[i]);
```

```

32 }
33 fprintf(stdout, "\n");
34
35 /* compute the octal number (from MSD) */
36 printf("Octal number (from MSB) = ");
37 for (no=0, i=0; i<nb; no++, i+=3) {
38     oct[no] = 0;
39     for (p=1, j=i; j<i+3 && j<nb; j++, p*=2) {
40         oct[no] = oct[no] + bit[j] * p;
41     }
42 }
43 for (i=no-1; i>=0; i--) {
44     fprintf(stdout, "%d", oct[i]);
45 }
46 fprintf(stdout, "\n");
47
48 /* compute the hexadecimal number (from MSD) */
49 printf("Hexadecimal number (from MSD) = ");
50 for (nh=0, i=0; i<nb; nh++, i+=4) {
51     hex[nh] = 0;
52     for (p=1, j=i; j<i+4 && j<nb; j++, p*=2) {
53         hex[nh] = hex[nh] + bit[j] * p;
54     }
55 }
56 for (i=nh-1; i>=0; i--) {
57     if (hex[i]>=10) {
58         printf("%c", c[hex[i]-10]);
59     } else {
60         printf("%d", hex[i]);
61     }
62 }
63 fprintf(stdout, "\n");
64
65 return 0;
66 }

```

## 1.8 Rule-out Zero Values

### Specifications

Write a program which:

- ▷ Reads an array of DIM integer values. DIM is a pre-defined constant.
- ▷ Eliminates all elements equal to zero from the array and shift all other values on the left, i.e., toward the beginning of the array.
- ▷ Prints-out the resulting array (only meaningful elements).

**Example 1.13** Let the following be one the input array of size  $\text{DIM} = 10$ . This array has to be

0	1	2	3	4	5	6	7	8	9
0	15	5	10	25	0	0	5	15	10

transformed as the following: where the elements in position 7, 8 and 9 are undefined. Notice that it is not possible to use another array.

0	1	2	3	4	5	6	7	8	9
15	5	10	25	5	15	10	?	?	?

## Solution 1

In this solution we use two cycles to solve the problem. The outer one searches for all zeros in the array. The inner one shifts left of one position all elements following each zeros. In the worst case, the complexity of the algorithm is then quadratic, i.e.,  $O(n^2)$  if  $n$  is the number of elements in the array.

```

1 #include <stdio.h>
2
3 #define DIM 10
4
5 int main(void) {
6     int i, j, n;
7     int array[DIM];
8
9     /* read in the array */
10    printf("Input array:\n");
11    for (i=0; i<DIM; i++) {
12        printf("array[%d] = ", i);
13        scanf("%d", &array[i]);
14    }
15
16    /* computation */
17    n = DIM;
18    i = 0;
19    while (i < n) {
20        if (array[i] == 0) {
21            /* left-shift all remaining elements by one position */
22            for (j=i+1; j<n; j++) {
23                array[j-1] = array[j];
24            }
25            /* decrease the array logic size */
26            n = n - 1;
27        } else {
28            /* move right only when there is no shift */
29            i = i + 1;
30        }
31    }
32
33    /* print the resulting array */
34    printf("Output array:\n");
35    for (i=0; i<n; i++) {
36        printf("array[%d] = %d\n", i, array[i]);
37    }
38    return 0;
39 }
```

## Solution 2

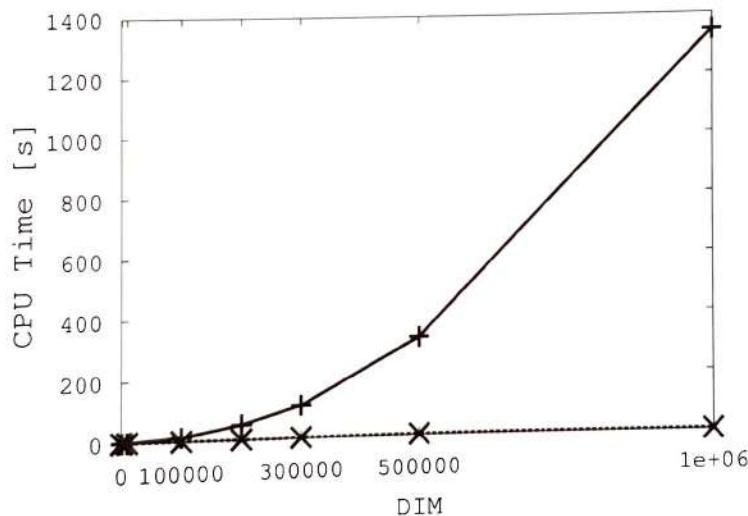
If  $n$  is the size of the array, the previous solution has a  $O(n^2)$  time cost. The following solution, on the contrary, has a cost equal to  $O(n)$ .

The following table plots the wall clock time<sup>2</sup> for the two programs with increasing value of DIM. In bot cases the arrays are randomly generated.

DIM	CPU Time for Solution 1	CPU Time for Solution 2
100	0.75	0.68
1000	0.89	0.82
10000	1.49	0.99
100000	13.26	1.01
200000	50.36	1.59
300000	108.46	1.69
500000	323.96	1.79
1000000	1342.86	1.89

**Table 1.1** A time comparison between solution 1 and solution 2.

Those data are also plotted in Figure 1.1.



**Figure 1.1** A time comparison between solution 1 (quadratic growth) and solution 2 (linear growth).

```

1 #include <stdio.h>
2
3 #define DIM 10
4
5 int main(void) {
6     int i, j;
7     int array[DIM];
8
9     /* read in the array */

```

<sup>2</sup> The wall clock time is the time necessary to the (mono-thread or multi-thread) process to complete the task, i.e., the difference between the time at which the task finishes and the time at which the task started. For this reason, the wall clock time is also known as *elapsed time*.

```

10   printf("Input Array:\n");
11   for (i=0; i<DIM; i++) {
12     printf("array[%d] = ", i);
13     scanf("%d", &array[i]);
14   }
15
16   /* computation */
17   for (i=0, j=0; i<DIM; i++) {
18     if (array[i] != 0) {
19       array[j] = array[i];
20       j++;
21     }
22   }
23
24   /* print the resulting array */
25   printf("Output array:\n");
26   for (i=0; i<j; i++) {
27     printf("array[%d] = %d\n", i, array[i]);
28   }
29
30   return 0;
31 }
```

## 1.9 Sub-Array Search

### Specifications

Write a program to:

- ▷ Read two arrays of integer values v1 and v2, of size DIM1 and DIM2, respectively. DIM1 and DIM2 are pre-defined constants, and DIM1 > DIM2.
- ▷ Understand whether the sequence of values stored in v2 appears in v1 as a subsequence. In such a case the program has to print-out the starting index.

Correctly deal with the case in which the v2 sequence appears in v1 more than once.

**Example 1.14** Given  $\text{DIM1} = 5$ ,  $\text{DIM2} = 2$  and:

```
v1 = 1 2 3 4 5
v2 = 2 3
```

The sequence “2 3” stored in v2 belongs to v1 starting from the element of index 1.

**Example 1.15** Given  $\text{DIM1} = 5$ ,  $\text{DIM2} = 2$  and:

```
v1 = 1 2 1 2 1
v2 = 1 2
```

The sequence “1 2” stored in v2 appears in v1 twice, starting from elements of index 0 and 2.

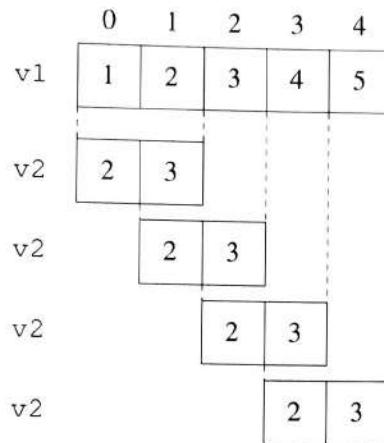
**Example 1.16** Given  $\text{DIM1} = 10$ ,  $\text{DIM2} = 3$  and:

```
v1 = 1 2 1 2 1 2 1 2 1 2
v2 = 1 2 1
```

The sequence “1 2 1” may be considered as appearing in v1 starting from position 0, 2, 4 and 6 or just 0 and 4 if we do not consider overlapping sub-sequences.

## Solution

The program logic is represented by the following picture, in which the second array is overlapped to the first one in all possible overlapping positions. For each position a



comparison of the small array and the corresponding section of the large one solve the problem.

```

1 #include <stdio.h>
2
3 #define DIM1 5
4 #define DIM2 3
5 #define OVERLAPPING 1
6
7 int main(void) {
8     int v1[DIM1], v2[DIM2], i, j, flag;
9
10    /* read in the first array */
11    printf("Input v1 (%d elements).\n", DIM1);
12    for (i=0; i<DIM1; i++) {
13        printf(" v1[%d]: ", i);
14        scanf("%d", &v1[i]);
15    }
16
17    /* read in the second array */
18    printf("Input v2 (%d elements).\n", DIM2);
19    for (i=0; i<DIM2; i++) {
20        printf(" v2[%d]: ", i);
21        scanf("%d", &v2[i]);
22    }
23
24    /* computation */
25    i = 0;
26    while (i <= DIM1-DIM2) {
27        j = 0;
28        flag = 1;
29        while (j<DIM2 && flag==1) {
30            if (v1[i+j] != v2[j]) {
31                flag = 0;
32            }
33            j++;
34        }

```

```

35     if (flag == 1) {
36         printf("Sub-sequence found from element %d.\n", i);
37         if (OVERLAPPING == 0) {
38             /* find out NON-overlapping sub-sequences */
39             i = i + DIM2;
40         } else {
41             /* find out overlapping sub-sequences */
42             i++;
43         }
44     } else {
45         i++;
46     }
47 }
48
49 return 0;
50 }
```

## 1.10 Horizontal Histogram

### Specification

Write a program that:

- ▷ Reads a sequence of integer numbers until a value strictly negative or larger than 99 is introduced.
- ▷ Prints-out an histogram made up of 10 rows of characters '#', where the numbers of characters in the.
  - ◊ First row indicates the number of values previously introduced in the range [0, 9].
  - ◊ Second row indicates the number of values previously introduced in the range [10, 19].
  - ◊ etc.
  - ◊ The tenth (and last) row indicates the number of values previously introduced in the range [90, 99]

**Example 1.17** Let us suppose to introduce the following sequence:

1 2 3 4 5 10 11 12 20 30 40 50 60 62 64 66 68 70 80 90 98 -1

the program has to print-out the following histogram:

```

0- 9 #####
10-19 ####
20-29 #
30-29 #
40-29 #
50-59 #
60-69 #####
70-79 #
80-80 #
90-99 ##
```

## Solution

To solve this problem, the most important issue to understand is that it is not necessary to store all values introduced from standard input in an array as each value can be manipulated as soon as it has been introduced. Moreover, the number of values introduced is unknown, thus it would be impossible to allocate a proper array to store them all. On the contrary, it is compulsory to store a counter for each histogram class, counting the number of values belonging to the class itself. Several possibility can be used to understand which counter it is necessary to increment, such as a sequence of **if** construct, or a single **if** construct within a cycle. In the first case the code would appear like the following:

```
if (val>=0 && val<=9)
    classes[0]++;
else if (val>=10 && val<=19)
    classes[1]++;
...
else if (val>=90 && val<=99)
    classes[9]++;
```

In the second case, it would appear like:

```
for (i=0; i<10; i++) {
    if (val>=(i*10) && val<=(i*10+9))
        classes[i]++;
}
```

The solution illustrate a shortest strategy, in which, in line 26 the operator / is used to compute the integer quotient of the integer division  $\text{val} / 10$ . For example, if  $\text{val}=13$  then  $\text{val} / 10 = 1$ , if  $\text{val}=45$  then  $\text{val} / 10 = 4$ , etc.

Notice that lines 9–12 are used to initialize the array **classes**, i.e., all class counters. A possible alternative is to initialize it during the definition by substituting line 7 with the following: as

```
int classes[SIZE] = {0};
```

Even if this is possible, we will rarely use this strategy in this book.

```
1 #include <stdio.h>
2
3 #define SIZE 10
4
5 int main(void) {
6     int i, j, val, flag;
7     int classes[SIZE];
8
9     /* init array: zero frequency */
10    for (i=0; i<SIZE; i++) {
11        classes[i] = 0;
12    }
13
14    /* input phase: read numbers and collect statistics */
15    printf("Input sequence:\n");
16    flag = 1;
17    do {
18        printf("Value = ");
19        scanf("%d", &val);
20
21        /* check the value for termination */
22        if (val<0 || val>=SIZE*SIZE) {
```

```

23     flag = 0;
24 } else {
25     /* increase the class frequency */
26     classes[val/SIZE]++;
27 }
28 } while (flag == 1);
29
30 /* output phase */
31 for (i=0; i<SIZE; i++) {
32     printf("[%2d-%2d] ", i*SIZE, (i+1)*SIZE-1);
33     for (j=0; j<classes[i]; j++) {
34         printf("#");
35     }
36     printf("\n");
37 }
38
39 return 0;
40 }
```

## 1.11 Vertical Histogram

### Specifications

Write a program that:

- ▷ reads a sequence of integer numbers till the introduction of a negative or strictly larger than 99 value
- ▷ prints-out an histogram made up of 10 columns of characters '#', where the number of characters in the
  - ◊ first column indicates the number of values previously introduced in the range [0, 9]
  - ◊ second column indicates the number of values previously introduced in the range [10, 19]
  - ◊ etc.
  - ◊ the tenth (and last) column indicates the number of values previously introduced in the range [90, 99]

**Example 1.18** Let us suppose to introduce the following sequence:

1 2 3 4 5 10 11 12 20 30 40 50 60 62 64 66 68 70 80 90 98 -1

the program has to print-out the following:

```
#####
##  #  #
##  #
#  #
#  #
```

### Solution

Notice that standard output is managed in textual format, not with graphical features. This implies that it is only possible to move the cursor (and print each new character) from left to right and from top to bottom.

```

1 #include <stdio.h>
2
3 #define SIZE 10
4
5 int main(void) {
6     int i, val, flag;
7     int classes[SIZE];
8
9     /* init array: zero frequency */
10    for (i=0; i<SIZE; i++) {
11        classes[i] = 0;
12    }
13
14    /* input phase: read numbers and collect statistics */
15    printf("Input sequence:\n");
16    flag = 1;
17    do {
18        printf("Value = ");
19        scanf("%d", &val);
20
21        /* check the value for termination */
22        if (val<0 || val>=SIZE*SIZE) {
23            flag = 0;
24        } else {
25            /* increase the class frequency */
26            classes[val/SIZE]++;
27        }
28    } while (flag == 1);
29
30    /* output phase */
31    flag = 1;
32    while (flag == 1) {
33        flag = 0;
34        for (i=0; i<SIZE; i++) {
35            if (classes[i] > 0) {
36                printf("#");
37                classes[i]--;
38                flag = 1;
39            } else {
40                printf(" ");
41            }
42        }
43        printf("\n");
44    }
45
46    return 0;
47 }
```

## 1.12 Characters and Strings

Characters are the fundamental building blocks of source programs. The value of a character constant is the integer value of the character in the machine's character set. For example, 'z' represents the integer value of z, and '\n' the integer value of newline (122 and 10 in ASCII, respectively).

A string is a series of characters treated as a single unit. A string may include letters, digits and various special characters. Strings are actually one-dimensional

array of characters terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.

Strings can be defined and initialize in different ways, such as:

```
char str1[20] = {'s', 't', 'r', 'i', 'n', 'g', '\0'};
char str2[20] = "string";
char str3[] = "string";
char *str4 = "string";
```

## 1.13 Anagram Verification

### Specifications

Write a program able to:

- ▷ read from command line two strings made up of alphabetic characters only.
- ▷ understand whether the two strings are anagrams, that is, they are made by the same set of letters placed in different position. Write a message in case the two strings are anagrams.

Capital and small letters have to be consider as equivalent.

### Solution 1

In this solution we organize the process using two nested cycle: For each character of the first string (outer loop) we look for an equal character in the second string (inner loop). As each character must be counted only once, each matching character in the second string is substituted with a dash symbol.

Notice that this solution is quite inefficient, and with similar considerations as the one reported for Exercise 1.8 we state that it has a quadratic cost. In the next subsection, we present a linear solution.

```
1 #include <stdio.h>
2 #include <ctype.h>
3 #include <string.h>
4
5 #define MAX_LEN 31
6
7 int main(int argc, char *argv[]) {
8     char s1[MAX_LEN], s2[MAX_LEN];
9     int i, j, found;
10
11    /* manage parameters */
12    if (argc != 3) {
13        printf("Parameter error.\n");
14        printf("Run program as: %s <string1> <string2>\n", argv[0]);
15        return 1;
16    }
17    strcpy(s1, argv[1]);
18    strcpy(s2, argv[2]);
19
20    /* check for anagrams */
21    for (found=1, i=0; i<strlen(s1) && found==1; i++) {
22        for (found=0, j=0; j<strlen(s2) && found==0; j++) {
23            if (s2[j]==s1[i]) {
24                s2[j] = '-';
```

```

25         found = 1;
26     }
27 }
28 }
29
30 if (found==0) {
31     printf("The two words are NOT anagrams.\n");
32 } else {
33     for (found=0, i=0; i<strlen(s2) && found==0; i++) {
34         if (s2[i] != '-') {
35             found = 1;
36         }
37     }
38     if (found==0) {
39         printf("The two words are anagrams.\n");
40     } else {
41         printf("The two words are NOT anagrams.\n");
42     }
43 }
44
45 return 0;
46 }
```

## Solution 2

In this solution, we associate a counter to each alphabetic characters. Then, we use these counters to count-up the number of characters in the first string and to count down the number of characters in the second string. At the end of the process, all counters have to be equal to zero. To have one counter for each alphabetic character we define an array of 31 elements, and then we associate each alphabetic letter to one element. To do that, we proceed as follow.

Table 1.2 reports the ASCII representation for all small and capital letters of the Latin alphabet. It can be noticed that consecutive letters have consecutive codes.

Thus, given a small alphabetic character  $c$  the value of  $c - 'a'$  correspond to the ordinal position of that character within the small alphabetic alphabet.

```

1 #include <stdio.h>
2 #include <ctype.h>
3 #include <string.h>
4
5 #define MAX_LEN 31
6
7 int main(int argc, char *argv[]) {
8     char s1[MAX_LEN], s2[MAX_LEN];
9     int occurrences[26], i, flag;
10
11    /* manage parameters */
12    if (argc != 3) {
13        printf("Parameter error.\n");
14        printf("Run program as: %s <string1> <string2>\n", argv[0]);
15        return 1;
16    }
17    strcpy(s1, argv[1]);
18    strcpy(s2, argv[2]);
19
20    /* check for anagrams */
21    for (i=0; i<26; i++) {
```

Character	Decimal	Octal	Hexadecimal	Character	Decimal	Octal	Hexadecimal
a	97	141	61	À	65	101	41
b	98	142	62	È	66	102	42
c	99	143	63	Ç	67	103	43
d	100	144	64	Ð	68	104	44
e	101	145	65	È	69	105	45
f	102	146	66	È	70	106	46
g	103	147	67	È	71	107	47
h	104	150	68	È	72	110	48
i	105	151	69	È	73	111	49
j	106	152	6a	È	74	112	4a
k	107	153	6b	È	75	113	4b
l	108	154	6c	È	76	114	4c
m	109	155	6d	È	77	115	4d
n	110	156	6e	È	78	116	4e
o	111	157	6f	È	79	117	4f
p	112	160	70	È	80	120	50
q	113	161	71	È	81	121	51
r	114	162	72	È	82	122	52
s	115	163	73	È	83	123	53
t	116	164	74	È	84	124	54
u	117	165	75	È	85	125	55
v	118	166	76	È	86	126	56
w	119	167	77	È	87	127	57
x	120	170	78	È	88	130	58
y	121	171	79	È	89	131	59
z	122	172	7a	È	90	132	5a

Table 1.2 ASCII codes for all alphabetic letters.

```

22     occurrences[i] = 0;
23 }
24 for (i=0; i<strlen(s1); i++) {
25     occurrences[tolower(s1[i]) - 'a']++;
26 }
27 for (i=0; i<strlen(s2); i++) {
28     occurrences[tolower(s2[i]) - 'a']--;
29 }
30 flag = 1;
31 for (i=0; i<26 && flag; i++) {
32     if (occurrences[i] != 0) {
33         flag = 0;
34     }
35 }
36
37 /* output result */
38 if (flag) {
39     printf("The two words are anagrams.\n");
40 } else {
41     printf("The two words are NOT anagrams.\n");
42 }
43
44 return 0;
45 }
```

### Solution 3

In this solution the two parameters are not copied into `s1` and `s2`, but are directly manipulated by the program.

```

1 #include <stdio.h>
2 #include <ctype.h>
3 #include <string.h>
4
5 int main(int argc, char *argv[]) {
6     int occurrences[26], i, flag;
7
8     /* check parameters */
9     if (argc != 3) {
10         printf("Parameter error.\n");
11         printf("Run program as: %s <string1> <string2>\n", argv[0]);
12         return 1;
13     }
14
15     /* check for anagrams */
16     for (i=0; i<26; i++) {
17         occurrences[i] = 0;
18     }
19     for (i=0; i<strlen(argv[1]); i++) {
20         occurrences[tolower(argv[1][i]) - 'a']++;
21     }
22     for (i=0; i<strlen(argv[2]); i++) {
23         occurrences[tolower(argv[2][i]) - 'a']--;
24     }
25     flag = 1;
26     for (i=0; i<26 && flag; i++) {
27         if (occurrences[i] != 0) {
28             flag = 0;
29         }
30     }
31
32     /* output result */
33     printf("The two words are %s anagrams.\n", flag ? "" : " NOT");
34
35     return 0;
36 }
```

## 1.14 String Rotation

### Specifications

Write a program that:

- ▷ Reads from standard input a C string (no spaces) of at most 100 characters.
- ▷ Prints-out all strings generated by the input one by rotating its characters (left or right-rotations) of one single character at the time.

**Example 1.19** Let “examination” be the input string. The program has to print-out the following strings:

```
examination
xaminatione
aminationex
```

```

minationexa
inationexam
nationexami
ationexamin
tionexamina
ionexaminat
onexaminati
nexaminatio

```

## Solution 1

In this solution the string is “physically” rotated to create a new string which is then directly printed-out.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 #define MAX 100
5
6 int main(void) {
7     char string[MAX+1], c;
8     int i, j, length;
9
10    printf("Input string: ");
11    scanf("%s", string);
12
13    length = strlen(string);
14    for (i=0; i<length; i++) {
15        /* save the first char */
16        c = string[0];
17
18        /* left shift all chars */
19        for (j=0; j<length-1; j++) {
20            string[j] = string[j+1];
21        }
22
23        /* copy the saved char in the final position */
24        string[length-1] = c;
25
26        printf("Rotated string (rotation %d): %s\n", i+1, string);
27    }
28
29    return 0;
30 }

```

## Solution 2

In this solution the rotation is only “logically” realized, i.e., the original string is not modified. The rotated string is generated directly by the printing procedure.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 #define MAX 100
5
6 int main(void) {
7     char string[MAX+1];
8     int i, j, k, length;

```

```

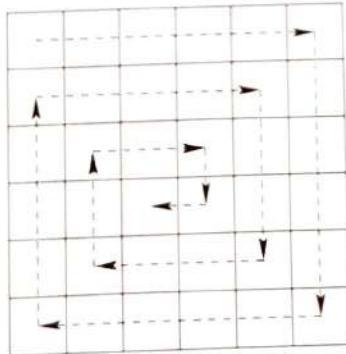
9     printf("Input string: ");
10    scanf("%s", string);
11
12    length = strlen(string);
13    for (i=0; i<length; i++) {
14        printf("Rotated string (rotation %d): ", i+1);
15        for (j=0; j<length; j++) {
16            k = (i+j) % length;
17            printf("%c", string[k]);
18        }
19        printf("\n");
20    }
21
22    return 0;
23 }
24 }
```

## 1.15 Spiral Print

### Specifications

Write a program which is able to:

- ▷ Read from standard input a square matrix of integer values, whose maximum size is  $N \times N$ , where  $N$  is a pre-defined constant.
- ▷ Print all elements of such a matrix (one for each row) following a spiral order, starting from element  $(0, 0)$  and proceeding clock-wise as represented by the following picture.



### Solution

The following solution adopts four `for` statements to print-out a row from left to right, a column from top to bottom, a row from right to left, and, finally, a column from bottom to top. One more `for` construct repeats the entire process a number of times which depends on the matrix size.

Notice, that a little care is required to understand whether the “central” element has to be generated and written-out (always, for odd-sized matrices) after the outer cycle terminates.

```

1 #include <stdio.h>
2
3 #define N 5
4
5 int main (void) {
6     int i, j, matrix[N][N];
7
8     /* input matrix */
9     for (i=0; i<N; i++) {
10         for (j=0; j<N; j++) {
11             printf("Element [%d, %d]: ", i, j);
12             scanf("%d", &matrix[i][j]);
13         }
14     }
15
16     /* print matrix elements */
17     for (i=0; i<N/2; i++) {
18         /* from left to right */
19         for (j=i; j<N-1-i; j++) {
20             printf("Element [%d, %d]: %d\n", i, j, matrix[i][j]);
21         }
22         /* from top to bottom */
23         for (j=i; j<N-1-i; j++) {
24             printf("Element [%d, %d]: %d\n", j, N-1-i, matrix[j][N-1-i]);
25         }
26         /* from right to left */
27         for (j=N-1-i; j>i; j--) {
28             printf("Element [%d, %d]: %d\n", N-1-i, j, matrix[N-1-i][j]);
29         }
30         /* from bottom to top */
31         for (j=N-1-i; j>i; j--) {
32             printf("Element [%d, %d]: %d\n", j, i, matrix[j][i]);
33         }
34     }
35
36     if (N % 2) {
37         printf("Element [%d, %d]: %d\n", N/2, N/2, matrix[N/2][N/2]);
38     }
39
40     return 0;
41 }
```

## 1.16 Local Minimum Search

### Specifications

Write a program that:

- ▷ Reads a matrix of R rows and C columns forcing each elements to be strictly positive (i.e., re-read the value otherwise)
- ▷ Prints-out the coordinate (row and column) of each element whose value is smaller than the (at most) eight adjacent elements in the matrix.

**Example 1.20** Let us suppose to have the following matrix of 5 rows and 5 columns:

10	3	2	4	50
34	4	2	25	9

```

10 33 4 23 2
 9 5 3 24 6
91 5 33 7 4

```

The program has to print out the coordinate of the three elements whose value is smaller than all adjacent elements:

```

Local Minimum (value=2) at row=3, column=5.
Local Minimum (value=3) at row=4, column=3.
Local Minimum (value=4) at row=5, column=5.

```

## Solution

For each element within the matrix it is necessary to consider all (at most 8) adjacent values. Notice anyway, that elements on the matrix frame have less than 8 adjacent elements.

```

1 #include <stdio.h>
2
3 #define R 3
4 #define C 4
5
6 int main(void) {
7     int matrix[R][C];
8     int r, c, i, j;
9     char flag;
10
11    /* read matrix; each element must be positive */
12    for (r=0; r<R; r++) {
13        for (c=0; c<C; c++) {
14            do {
15                printf("Element [%d] [%d] = ", r+1, c+1);
16                scanf("%d", &matrix[r][c]);
17            } while (matrix[r][c]<=0);
18        }
19    }
20
21    /* computation: for each matrix element ... */
22    for (r=0; r<R; r++) {
23        for (c=0; c<C; c++) {
24
25            /* flag = 1 means that element (r, c) is a local minimum */
26            flag = 1;
27
28            /* ... check the 8 adjacent elements out ... */
29            for (i=r-1; i<=r+1 && flag==1; i++) {
30                for (j=c-1; j<=c+1 && flag==1; j++) {
31                    if ((i!=r || j!=c) && i>=0 && i<R && j>=0 && j<C) {
32                        if (matrix[r][c] >= matrix[i][j]) {
33                            flag = 0;
34                        }
35                    }
36                }
37            }
38
39            /* ... and print the result */
40            if (flag == 1) {
41                printf("Local Minimum (value=%d) ", matrix[r][c]);
42                printf("at row=%d, column=%d.\n", r, c);

```

```

43     }
44   }
45 }
46
47 return 0;
48 }
```

## 1.17 Sub-matrix Search

### Specifications

Let  $\text{DIM1}$  and  $\text{DIM2}$  be two integer constants, with  $\text{DIM1}$  larger than  $\text{DIM2}$ . Write a program able to:

- ▷ Read from standard input two integer matrices,  $m_1$  of size  $\text{DIM1} \times \text{DIM1}$ , and  $m_2$  of size  $\text{DIM2} \times \text{DIM2}$ , storing only characters '\*' and '#'.  
▷ Find in  $m_1$  all sub-matrices equivalent to  $m_2$ .
- ▷ Copy all sub-matrices equal to  $m_2$  in  $m_1$  into a new matrix  $m_3$ , in which all other characters are white elements (blanks).
- ▷ Print-out  $m_3$ .

**Example 1.21** Let  $\text{DIM1} = 5$  and  $\text{DIM2} = 3$ . If  $m_1$  and  $m_2$  are the following:

```
*#***#
##**#
*#*#*
#*###
#***#*
```

and

```
*#*
###
*#*
```

the program has to generate the following matrix  $m_3$  (and print it out):

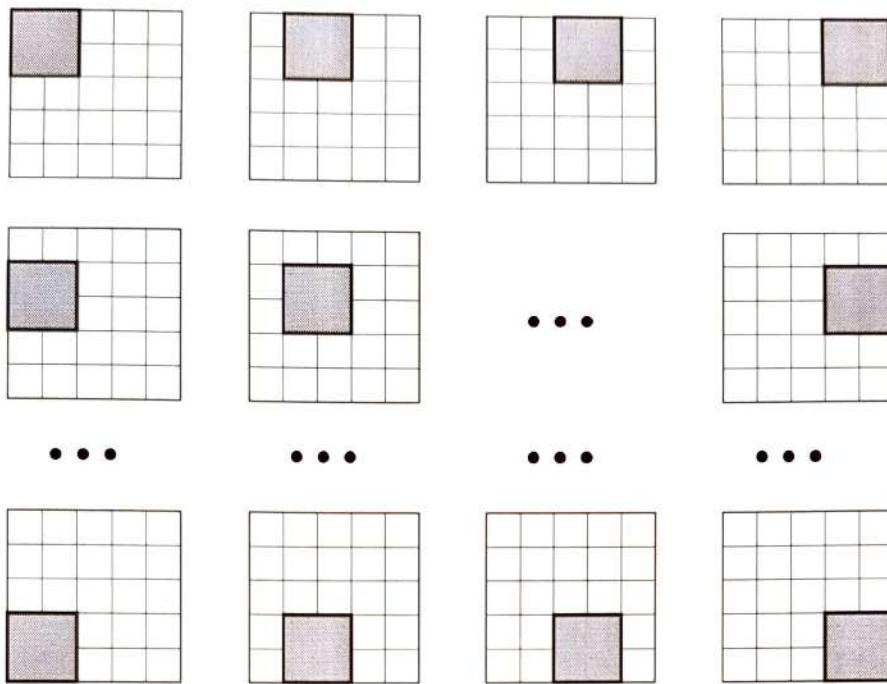
```
*#*
###
*#*#*
###
*#*
```

### Solution

Finding a smaller matrix as a sub-matrix of a larger one, is an interesting problem. We can solve it as follows. The smaller matrix can be logically “overlapped” to the larger one in all possible positions as illustrated by the following picture. Then, for each overlapping position, we organize two nested cycles to compare all elements of the smaller matrix to all overlapped elements of the larger matrix. Appropriate care has to be taken to avoid bound errors.

```

1 #include <stdio.h>
2
3 #define DIM1 5
```



```

4 #define DIM2 3
5
6 int main(void) {
7     int i, j, r, c, flag;
8     char m1[DIM1][DIM1], m2[DIM2][DIM2], aux[DIM1][DIM1];
9
10    /* read in the first matrix */
11    printf("Input matrix 1\n");
12    for (i=0; i<DIM1; i++) {
13        for (j=0; j<DIM1; j++) {
14            printf("Element [%d, %d] : ", i, j);
15            scanf("%c%c", &m1[i][j]);
16            aux[i][j] = 0;
17        }
18    }
19
20    /* read in the second matrix */
21    printf("Input matrix 2\n");
22    for (i=0; i<DIM2; i++) {
23        for (j=0; j<DIM2; j++) {
24            printf("Element [%d, %d] : ", i, j);
25            scanf("%c%c", &m2[i][j]);
26        }
27    }
28
29    /* computation */
30    for (i=0; i<=DIM1-DIM2; i++) {
31        for (j=0; j<=DIM1-DIM2; j++) {
32            flag = 1;
33            for (r=0; r<DIM2 && flag==1; r++) {
34                for (c=0; c<DIM2 && flag==1; c++) {
35                    if (m1[i+r][j+c] != m2[r][c]) {

```

```

36         flag = 0;
37     }
38 }
39 }
40 if (flag == 1) {
41     for (r=0; r<DIM2; r++) {
42         for (c=0; c<DIM2; c++) {
43             aux[i+r][j+c] = 1;
44         }
45     }
46 }
47 }
48 }
49 }
50 /* output result */
51 for (i=0; i<DIM1; i++) {
52     for (j=0; j<DIM1; j++) {
53         printf("%c", (aux[i][j]==1) ? m1[i][j] : ' ');
54     }
55     printf("\n");
56 }
57 }
58 return 0;
59 }
```

## 1.18 Functions

Most computer programs that solve real-world problems are quite large. Experience has shown that the best way to develop and maintain a large program is to construct it from smaller pieces or modules, each of which is more manageable than the original program. This technique is called *divide-and-conquer*.

Modules in C are called functions. C programs are typically written by combining new functions you write with prepackaged functions available in the C standard library. The C standard library provides a rich collection of functions for performing common mathematical calculations, string manipulations, character manipulations, input/output, and many other useful operations. This makes your job easier, because these functions provide many of the capabilities you need. For the remaining ones, the user is obliged to write his/her own functions.

### 1.18.1 Definition and Declaration

A function is a group of statements that together perform a task. Every C program has at least one function, which is `main`, and all the most trivial programs can define additional functions. You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function *declaration* tells the compiler about a function's name, return type, and parameters. A function *definition* provides the actual body of the function. A function definition in C programming consists of a function header and a function body:

```
<return_type> <function_name> (<formal parameters>) {  
    <local definitions>
```

```

<function body>

return <val>;
}

```

A function may return a value. The `return_type` is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the `return_type` is the keyword `void`.

The `function_name` is the actual name of the function. The function name with the parameter list constitute the function signature.

A parameter is like a placeholder. Parameters are optional; that is, a function may contain no parameters. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters can be passed *by value* and *by reference*. In the call by value, the actual value of an argument is copied into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. In the call by reference the address of an argument is copied into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. By default, C uses call by value to pass arguments. In general, this means that the code within a function cannot alter the arguments used to call the function. Anyhow, array are automatically passed by reference to a function, as the name of the array is the pointer to the first element of the array.

Variables are local to the function. The function body contains a collection of statements that define what the function does.

A function declaration (or prototype) tells the compiler about a function name and how to call the function:

```
<return_type> <function_name> (<formal parameters>);
```

The actual body of the function can be defined separately.

## 1.18.2 Function Call

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value:

```
<val> = <function_name> (<actual parameters>);
```

Integer (or void) return values can be ignored:

```
<function_name> (<actual parameters>);
```

## 1.18.3 Arguments to the main program

For those writing programs which will run in a hosted environment, arguments to main provide a useful opportunity to give parameters to programs. Typically, this facility is

used to direct the way the program goes about its task, for example it's particularly common to provide file names to a program through its arguments. The declaration of main looks like this:

```
int main(int argc, char *argv[]);
```

This indicates that main is a function returning an integer, and with at least two arguments. The first of these (argc) is a count of the arguments supplied to the program. The second is an array of pointers to the strings which are those arguments (its type is an “array of pointer to char”). Those arguments can then be used or just printed-out as:

```
int i;
printf ("Command line: ");
for (i=0; i<argc; i++) {
    printf("%s ", argv[i]);
}
printf("\n");
```

#### 1.18.4 Function Exit

Functions can be exited (i.e., global return):

```
#include <stdlib.h>
...
exit (<expr>);
```

It is possible to use constants:

- ▷ EXIT\_SUCCESS (often equal to 0, but implementation dependent) to return a success program termination code to the operating system.
- ▷ EXIT\_FAILURE (often equal to 1 but implementation dependent) to return an error program termination code to the operating system.

Those constant are usually defined in the stdlib.h standard library.

#### 1.18.5 String Manipulation

Useful function prototypes:

```
#include <string.h>
...
int sscanf (char *str, char *format, ...);
int sprintf (char *str, char *format, ...);
int strlen (char *str);
char *strcpy (char *dest_name, char *src_name);
char *strncpy (char *dest_name, char *src_name, int n);
char *strcat (char *dest_name, char *src_name);
char *strncat (char *dest_name, char *src_name, int n);
int strcmp (char *str1, char *str2);
int strncmp (char *str1, char *str2, int n);
char *strchr (char *str, char c);
char *strstr (char *str1, char *str2);
```

#### 1.18.6 Character Manipulation

C supports a wide range of functions that manipulate null-terminated strings. Useful function prototypes:

```
#include <ctype.h>
...
int isdigit (int c);
int islower (int c);
int isupper (int c);
int isalpha (int c);
int isspace (int c);
int toupper (int c);
int tolower (int c);
```

## 1.19 Sub-string Substitution Specifications

Write a program that:

- ▷ Receives three strings on the command line. Each string is at most 40 characters long.
- ▷ Substitutes into the first string all occurrences of the second string with the third string.
- ▷ Prints the resulting string on standard output.

**Example 1.22** Let us suppose that the command line is the following one:

abcdefghilmnopqrstuvwxyz mnopq #####

The program has to print-out:

abcdefgil#####rstuvwxyz

### Solution 1

Search and substitution are done by explicitly manipulating string characters.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define DIM 50
6
7 /* function prototypes */
8 int searchSubStr (char str1[], char str2[], int from);
9
10 /*
11  * main program
12 */
13 int main (int argc, char *argv[]) {
14     char str1[DIM+1], str2[DIM+1], str3[DIM+1];
15     char strOut[DIM*DIM+1];
16     int i, idx, from, to, endFlag;
17
18     /* check parameters */
19     if (argc != 4) {
20         printf("Parameter error.\n");
21         printf("Run program as: %s <string1> <string2> <string3>\n", argv[0]);
22         return 1;
23     }
```

```

24     strcpy(str1, argv[1]);
25     strcpy(str2, argv[2]);
26     strcpy(str3, argv[3]);
27
28     /* computation and output */
29     endFlag = from = idx = 0;
30
31     do {
32         to = searchSubStr(str1, str2, from);
33         if (to < 0) {
34             to = strlen(str1);
35             endFlag = 1;
36         }
37         for (i=from; i<to; i++) {
38             strOut[idx] = str1[i];
39             idx++;
40         }
41         if (endFlag == 0) {
42             for (i=0; i<strlen(str3); i++) {
43                 strOut[idx] = str3[i];
44                 idx++;
45             }
46         }
47         from = to+strlen(str2);
48     } while (endFlag != 1);
49
50     strOut[idx] = '\0';
51     printf("%s\n", strOut);
52     return 0;
53 }
54
55 /*
56 *   search a sub-string
57 */
58 int searchSubStr(char str1[], char str2[], int from) {
59     int i, j, flag;
60
61     i = from;
62     while (i < strlen(str1)-strlen(str2)) {
63         flag = j = 0;
64         while (j<strlen(str2) && flag==0) {
65             if (str1[i+j] != str2[j]) {
66                 flag = 1;
67             }
68             j++;
69         }
70         if (flag == 0) {
71             return i;
72         }
73         i++;
74     }
75     return -1;
76 }
```

## Solution 2

Standard C functions (such as `strstr`) are used to help the programmer to perform the required steps.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define DIM 50
6
7 int main (int argc, char *argv[]) {
8     char str1[DIM+1], str2[DIM+1], str3[DIM+1];
9     char strOut[DIM*DIM+1], *start, *end;
10
11    /* check parameters */
12    if (argc != 4) {
13        printf("Parameter error.\n");
14        printf("Run program as: %s <string1> <string2> <string3>\n", argv[0]);
15        return 1;
16    }
17
18    strcpy(str1, argv[1]);
19    strcpy(str2, argv[2]);
20    strcpy(str3, argv[3]);
21
22    /* computation and output */
23    strOut[0] = '\0';
24    start = str1;
25    do {
26        end = strstr(start, str2);
27        if (end == NULL) {
28            strcat(strOut, start);
29        } else {
30            strncat(strOut, start, end-start);
31            strcat(strOut, str3);
32            start = end + strlen(str2);
33        }
34    } while (end != NULL);
35
36    printf("%s\n", strOut);
37
38    return 0;
39 }
```

## 1.20 Average Values

### Specifications

Write a program that:

- ▷ Reads a matrix (named `matrixIn`) of real values and size equal to `MAX` rows and `MAX` columns.
- ▷ Computes a second matrix (named `matrixAvg`) in which each element  $(i, j)$  is equal to the average of all elements (at most 8) adjacent to element  $(i, j)$  of `matrixIn`.

Implement the solution with the following two functions:

- ▷ `avg`: to compute the average value of the adjacent elements for each element  $(i, j)$ .
- ▷ `valid`: as “border” elements do not have 8 adjacent elements, call this function from function `avg` to establish whether any element is inside the matrix or not.

**Example 1.23** Let us suppose matrixIn is the following:

```
1 2 3
4 5 6
7 8 9
```

matrixAvg will be the following one:

```
3.66 3.80 4.33
4.60 5.00 5.40
5.66 6.20 6.33
```

where, for example,  $3.66 = \frac{2+4+5}{3}$ ,  $3.80 = \frac{1+4+5+6+3}{5}$ , etc.

## Solution

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MAX 3
5 #define TRUE 1
6 #define FALSE 0
7
8 /* function prototypes */
9 float average(float matrixIn[MAX][MAX], int x, int y);
10 int valid(int x, int y);
11
12 /*
13 * main program
14 */
15 int main(void) {
16     float matrixIn[MAX][MAX], matrixAvg[MAX][MAX];
17     int x, y;
18
19     /* read in the input matrix */
20     printf("Input matrix:\n");
21     for (x=0; x<MAX; x++) {
22         for (y=0; y<MAX; y++) {
23             printf("Element [%d][%d] = ", x, y);
24             scanf("%f", &matrixIn[x][y]);
25         }
26     }
27
28     /* compute the output matrix */
29     for (x=0; x<MAX; x++) {
30         for (y=0; y<MAX; y++) {
31             matrixAvg[x][y] = average(matrixIn, x, y);
32         }
33     }
34
35     /* print out result */
36     printf("Output matrix:\n");
37     for (x=0; x<MAX; x++) {
38         for (y=0; y<MAX; y++) {
39             printf("%6.2f", matrixAvg[x][y]);
40         }
41         printf ("\n");
42     }
43
44 return 0;
```

```

45 }
46
47 /*
48 * compute the average of adjacent elements
49 */
50 float average (float matrixIn[MAX][MAX], int x, int y) {
51     int i, j, count;
52     float sum;
53
54     sum = count = 0;
55     for (i=-1; i<=1; i++) {
56         for (j=-1; j<=1; j++) {
57             if ((i!=0 || j!=0) && valid(x+i, y+j)==TRUE) {
58                 count++;
59                 sum += matrixIn[x+i][y+j];
60             }
61         }
62     }
63     return sum/count;
64 }
65
66 /*
67 * check whether the element [x][y] is inside (belongs to) the matrix
68 */
69 int valid (int x, int y) {
70     if (x<0 || x>=MAX) {
71         return FALSE;
72     }
73     if (y<0 || y>=MAX) {
74         return FALSE;
75     }
76
77     return TRUE;
78 }

```

## 1.21 Files

Storage of data in variables and arrays is temporary, i.e., such data is lost when a program terminates. Files are used for permanent retention of data. Computers store files on secondary storage devices, such as hard drives, CDs, DVDs and flash drives. We will consider only sequential-access file processing.

C views each file simply as a sequential stream of bytes. Each file ends either with an end-of-file marker or at a specific byte number recorded in a system-maintained, administrative data structure. When a file is opened, a stream is associated with it. Three files and their associated streams are automatically opened when program execution begins, i.e., the standard input, the standard output and the standard error. The standard library provides many functions for reading data from files and for writing data to files.

Always adopt the following five steps to deal with text files.

### 1.21.1 Define a File Pointer Object

When accessing files through C, the first necessity is to have a way to access the files. For C File you need to use a FILE pointer, which will let the program keep track of the file being accessed. A file pointer variable can be defined as:

```
FILE *<file_pointer_name>;
```

### 1.21.2 Open a File

To open a file you need to use the `fopen` function, which returns a `FILE` pointer.

```
FILE *fopen (char *<file_name>, const char *<access_mode>);
```

In our analysis the `access_mode` will be “`r`”, for reading the file, or “`w`” to write the file. The `file_name` can be a constant string, as in:

```
FILE *fp;
fp = fopen ("name.txt", "r");
```

a symbolic constant:

```
#define NAME "name.txt"
FILE *fp;
```

```
fp = fopen (NAME, "r");
```

or a string:

```
FILE *fp;
char name[L];
strcpy (name, "name.txt");
fp = fopen (name, "r");
```

### 1.21.3 Check a File Pointer

It is appropriate to check the result of the `fopen` function:

```
if (<file_pointer_name> == NULL) {
    fprintf(stderr, "Error opening file (%s).\\n", name);
    exit (1);
}
```

### 1.21.4 Input (Output) from (to) File

Once the file has been opened, you can use the `FILE` pointer to let the compiler perform input and output functions on the file. Use `fscanf` of `fgets` to read a file. While reading always check the end of file<sup>3</sup>. The check can be performed in different ways, when using `fscanf`:

```
while (fscanf(<file_pointer_name>, ...) != EOF) { ... }
```

or

```
while (fscanf(<file_pointer_name>, ...) != n) { ... }
```

if we want to make sure exactly `n` values have been read from file.

When using function `fgets`:

```
while (fgets(..., <file_pointer_name>) != NULL) { ... }
```

If we suppose there is enough space on the physical device on which we are writing, there is no need to perform any check after `fprintf`. Use function `fprintf` to write on a file, as:

```
for (...) {
    ...
    fprintf (<file_pointer_name>, ...);
```

---

<sup>3</sup> Do not use function `eof` in a wrong way to check for file termination, such as is while `(!eof(<file_pointer_name>)) { ... }`. In fact, `eof(<file_pointer_name>)` may be false, but the file may end (that is, EOF reached) during the next `fscanf` operation.

Notice that stdio.h also includes the definition of three pointers, namely stdin, stdout, and stderr, that can be used to re-direct input and output operations. In particular

```
fscanf (stdin, ...);
fprintf (stdout, ...);
```

are equivalent to scanf and printf, respectively. stderr is often used to logically differentiate outputs on standard output from the one from standard error.

### 1.21.5 Close a File

Always close file as soon it is not needed:

```
fclose (<file_pointer_name>);
```

## 1.22 Text Alignment

### Specifications

Write a program able to.

- ▷ Read from standard output a file name storing a generic ASCII text.
- ▷ Read the text from file and print each row of the file centered with respect to a row size of 80 columns.

**Example 1.24** Let the text be the following one:

Everyone is a genius at least once a year.  
 The real geniuses have their bright ideas closer together.  
 George Christoph Lichtenberg

It has to be written as follows:

Everyone is a genius at least once a year.  
 The real geniuses have their bright ideas closer together.  
 George Christoph Lichtenberg

### Solution

```

1 #include <stdio.h>
2 #include <string.h>
3
4 #define MAX_LINE 82 /* 80 + \n + \0 */
5
6 int main(void) {
7     char name[MAX_LINE], line[MAX_LINE];
8     int i, num_space;
9     FILE *fp;
10
11    /* open file */
12    printf("Input text file name: ");
13    scanf("%s", name);
14    fp = fopen(name, "r");
15    if (fp == NULL) {
16        printf("Error opening the input file.\n");
17        return 1;

```

```

18     }
19
20     /* read and print text */
21     printf("Centered text:\n");
22     while (fgets(line, MAX_LINE, fp) != NULL) {
23         num_space = (MAX_LINE-strlen(line))/2;
24         for (i=0; i<num_space; i++) {
25             printf(" ");
26         }
27         printf("%s", line);
28     }
29
30     /* close file */
31     fclose(fp);
32     return 0;
33 }
```

## 1.23 File Format

### Specifications

Write a program that:

- ▷ Receives two file names on the command line.
- ▷ Read from the first file an ASCII text of unknown length.
- ▷ Store all lines in the second file with the following rules:
  1. Each alphabetic characters coming after a space should be transformed into a capital letter.
  2. All white spaces have to be ruled-out.
  3. All lines (but the last one) have to be reduced to a length of exactly 20 characters.

**Example 1.25** If the input file is the following one:

Watch your thoughts ; they become words .  
 Watch your words ; they become actions .  
 Watch your actions ; they become habits .  
 Watch your habits ; they become character .  
 Watch your character ; it becomes your destiny .  
 Lao-Tze

the output one has to have the following format:

WatchYourThoughts;Th  
 eyBecomeWords.WatchY  
 ourWords;TheyBecomeA  
 ctions.WatchYourActi  
 ons;TheyBecomeHabits  
 .WatchYourHabits;The  
 yBecomeCharacter.Wat  
 chYourCharacter;ItBe  
 comesYourDestiny.Lao  
 -Tze

## Solution

Notice that in line 39 we use function `toupper` to transform the character `c` into its corresponding uppercase character. An alternative method, is the following one:

```
if (c>='a' && c<='z') [
    c = (char) ((int) c - (int) 'a' + (int) 'A');
```

which transform the character into its corresponding integer value (ASCII value), add to it the ASCII “distance” between the lowercase and uppercase alphabet, and it transforms it back into a character.

```
1 #include <stdio.h>
2 #include <ctype.h>
3
4 #define ROW_LENGTH 20
5
6 int main(int argc, char *argv[]) {
7     int i;
8     char c, blank;
9     FILE *fin, *fout;
10
11    if (argc < 3) {
12        printf("Error: missing parameter.\n");
13        printf("Run as: %s <input_file> <output_file>.\n", argv[0]);
14        return 1;
15    }
16
17    /* open the files */
18    fin = fopen(argv[1], "r");
19    if (fin == NULL) {
20        printf("Error opening the input file.\n");
21        return 1;
22    }
23    fout = fopen(argv[2], "w");
24    if (fout == NULL) {
25        printf("Error opening the output file.\n");
26        return 1;
27    }
28
29    /* computation */
30    i = 0;
31    blank = 0;
32    while (fscanf(fin, "%c", &c) != EOF) {
33        if (c != '\n') {
34            if (c == ' ') {
35                blank = 1;
36            } else {
37                if (blank) {
38                    c = toupper(c);
39                    blank = 0;
40                }
41
42                fprintf(fout, "%c", c);
43                i++;
44
45                if (i == ROW_LENGTH) {
46                    fprintf(fout, "\n");
47                    i = 0;
48                }
49            }
50        }
51    }
52}
```

```

49         }
50     }
51 }
52 if (i != 0) {
53     fprintf(fout, "\n");
54 }
55 fclose(fin);
56 fclose(fout);
57
58 return 0;
59 }
```

## 1.24 Structures

Structures, sometimes referred to as aggregates, are collections of related variables under one name. Structures may contain variables of many different data types, in contrast to arrays, which contain only elements of the same data type. Structures are commonly used to define records to be stored in files. Pointers and structures facilitate the formation of more complex data structures such as linked lists, queues, stacks and trees which will be the subject of many chapters to follow. Structures are derived data types, i.e., they are constructed using objects of other types. There are at least three possible ways to define a structure.

Following the first strategy, we firstly create a new structure type:

```
struct <name_struct> {
    <type_1> <name1>;
    <type_2> <name2>;
    ...
    <type_N> <nameN>;
};
```

and then we define variables of that type:

```
struct <struct_name> <var1>, <var2>, ...;
```

Following second strategy, we defines the new type and variables of that type at the same time:

```
struct <struct_name> {
    ...
} <var1>, <var2>, ..., <varN>;
```

Following the third strategy, we use **typedef** to define the new type:

```
typedef struct <struct_name> <type_name>;
typedef struct <struct_name> {
    ...
} <type_name>;
```

and then we define variable of the type:

```
<type_name> <var1>, <var2>, ..., <varN>;
```

### 1.24.1 Reference to fields

Two operators are used to access members of structures: The structure member operator ".", also called the dot operator, and the structure pointer operator "->", also called the arrow operator. The structure member operator accesses a structure member via the structure variable name.

A field can be accessed when the structure is possessed by value:

```
<var_name>.<field_name>
or by reference
(*<ptr_name>).<field_name>
```

In this last case, the structure pointer operator, consisting of a minus “-” sign and a greater than “>” sign with no intervening spaces, is usually preferred to access a structure member via a pointer to the structure:

```
<ptr_name>-><field>
```

### 1.24.2 Advanced Usage

Structures can be directly assigned to each other. For example in the following piece of code:

```
struct element {
    char c;
    int i;
    float f;
};

...
struct element e1, e2;
...
e1 = e2;
```

the assignment performed in the last row directly assigns all fields within structure `e2` to element in structure `e1`. In other words, it is equivalent to the following statements:

```
e1.c = e2.c;
e1.i = e2.i;
e1.f = e2.f;
```

This is true also when the structure includes array or matrices.

Structures may be passed to functions by passing individual structure members, by passing an entire structure, or by passing a pointer to a structure. When structures or individual structure members are passed to a function, they are passed by value. Therefore, the members of a caller’s structure cannot be modified by the called function. To pass a structure by reference, pass the address of the structure variable (we will accurately analyze this issue in Chapters 2 and 3). Arrays of structures, like all other arrays, are automatically passed by reference.

Remind that it is possible to define: Nested structures, array of structures, and structures of arrays.

## 1.25 Space Points

### Specifications

A file stores the Cartesian coordinates of a set of points in the 3D space. Each point is stored in a different line of the file, with the following format:

```
coordinateX coordinateY coordinateZ
```

where each coordinate is a real number. The maximum number of points is 100.

Write a program able to:

1. Read two file names on the command line.

2. Read the first file, containing space points, and store these point in a proper data structure, i.e., and array of C structures.
3. Compute the distance of all points from the origin of the Cartesian space.
4. Order all points in ascending order of the distance.
5. Store the ordered points in the second file with the following format:

coordinateX coordinateY coordinateZ distanceFromOrigin

## Solution

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 #define MAX_POINT 100
6
7 /* structure declaration */
8 struct point {
9     float x, y, z;
10    float distance;
11 };
12
13 /* function prototypes */
14 int read_points(char *, struct point []);
15 void sort_points(struct point [], int);
16 void write_points(char *, struct point [], int);
17
18 /*
19 * main program
20 */
21 int main (int argc, char *argv[]) {
22     struct point space_points[MAX_POINT];
23     int n_points;
24
25     if (argc < 3) {
26         printf("Error: missing parameter.\n");
27         printf("Run as: %s <input_file> <output_file>.\n", argv[0]);
28         return EXIT_FAILURE;
29     }
30
31     n_points = read_points(argv[1], space_points);
32     sort_points(space_points, n_points);
33     write_points(argv[2], space_points, n_points);
34
35     return EXIT_SUCCESS;
36 }
37
38 /*
39 * read the points input file
40 */
41 int read_points(char *name, struct point points[]) {
42     float x, y, z;
43     FILE *fp;
44     int i=0;
45
46     if ((fp=fopen(name, "r")) == NULL) {
47         printf("File open error (file=%s).\n", name);
48         exit(EXIT_FAILURE);

```

```

49     }
50     while (fscanf(fp, "%f%f%f", &x, &y, &z) !=EOF && i<MAX_POINT) {
51         points[i].x = x;
52         points[i].y = y;
53         points[i].z = z;
54         points[i].distance = sqrt(pow(x, 2)+pow(y, 2)+pow(z, 2));
55         i++;
56     }
57     fclose(fp);
58
59     return i;
60 }
61
62 /*
63 *   Insertion sort: sort all points based on distance from origin
64 *
65 */
66 void sort_points(struct point points[], int n_points) {
67     int i, j;
68     struct point current;
69
70     for (i=1; i<n_points; i++) {
71         // Save entire data struct (all fields) in temp variable
72         current = points[i];
73         j = i;
74         while ((--j >= 0) && (current.distance < points[j].distance)) {
75             // Move entire data struct (all fields) one element forward
76             points[j+1] = points[j];
77         }
78         // Copy temp data struct (all fields) back into array
79         points[j+1] = current;
80     }
81 }
82
83 /*
84 *   write the points output file
85 */
86 void write_points(char *name, struct point points[], int n_points) {
87     FILE *fp;
88     int i;
89
90     if ((fp=fopen(name, "w")) == NULL) {
91         printf("File open error (file=%s).\n", name);
92         exit(EXIT_FAILURE);
93     }
94     for (i=0; i<n_points; i++) {
95         fprintf(fp, "%f %f ", points[i].x, points[i].y);
96         fprintf(fp, "%f %f\n", points[i].z, points[i].distance);
97     }
98     fclose(fp);
99 }
```

## 1.26 Flights

### Specifications

A file stores the set of flights of an airline company, with the following format:

flightCode origin destination departureTime arrivalTime

where:

- ▷ `flightCode` is a string of 5 characters at most (e.g., “KLM01”) which uniquely identifies the flight.
- ▷ `origin` and `destination` indicate the departure and arrival airport cities (string of maximum 20 characters), respectively.
- ▷ `departureTime` and `arrivalTime` indicates the arrival and departure times as real numbers starting from midnight (e.g., 7.15 indicates 7h15' a.m.).

Write a program that once read a departure and an arrival city names, and a departure time, is able to find all connections (direct with no stop and with one single stop) from those two cities from the indicated departure time on. For each connection (intermediate stop) the departure time has to (obviously) follow the arrival time.

Notice that:

- ▷ The input file name has to be read on the command line.
- ▷ The maximum number of flights in the file is 150.
- ▷ All times are reported within the same day and the program does not have to consider connections beyond a single day time boundary.

**Example 1.26** Let the following be the input file:

```
AZ0A1 TOR ROM 07.00 08.00
AZ0A2 TOR ROM 11.00 12.00
AZ0A3 TOR ROM 17.00 18.00
AZ0A4 TOR ROM 19.00 20.00
AZ0B1 ROM PAL 07.30 08.30
AZ0B2 ROM PAL 11.30 12.30
AZ0B3 ROM PAL 17.30 18.30
AZ0B4 ROM PAL 19.30 20.30
AZ0C1 TOR PAL 07.45 09.45
AZ0C2 TOR PAL 11.45 13.45
AZ0C3 TOR PAL 17.45 19.45
AZ0C4 TOR PAL 19.45 21.45
```

If all flight from TOR to PAL leaving from 16.00 are requested, the program has to find the following connections:

```
1 stop:
AZ0A3 TOR ROM 17.00 18.00
AZ0B4 ROM PAL 19.30 20.30
No stop:
AZ0C3 TOR PAL 17.45 19.45
No stop:
AZ0C4 TOR PAL 19.45 21.45
```

## Solution

```
1 #include <stdio.h>
2 #include <string.h>
3
4 #define MAX_FLIGHTS 150
5
6 /* structure declaration */
7 typedef struct {
8     char code[6];
9     char depCity[4];
10    char arrCity[4];
11    float depTime;
```

```
12     float arrTime;
13 } flight_t;
14
15 /* function prototypes */
16 int flight_read (flight_t *flights, char *name);
17 void connection_search (flight_t *flights, int nf, flight_t request);
18 void info_write (flight_t flight);
19
20 /*
21 * main program
22 */
23 int main(int argc, char *argv[]) {
24     flight_t flights[MAX_FLIGHTS], request;
25     int nf;
26
27     if (argc < 2) {
28         printf("Error: missing parameter.\n");
29         printf("Run as: %s <input_file>.\n", argv[0]);
30         return 1;
31     }
32
33     /* load all flights */
34     nf = flight_read(flights, argv[1]);
35     if (nf == 0) {
36         return 1;
37     }
38
39     /* input requested connection */
40     printf("Introduce the departure city: ");
41     scanf("%s", request.depCity);
42     printf("Introduce the arrival city: ");
43     scanf("%s", request.arrCity);
44     printf("Introduce the departure time: ");
45     scanf("%f", &request.depTime);
46
47     /* search for the request */
48     connection_search(flights, nf, request);
49     return 0;
50 }
51
52 /*
53 * read the flights list from file
54 */
55 int flight_read (flight_t *flights, char *name) {
56     char line[100];
57     FILE *fp;
58     int i=0;
59
60     fp = fopen(name, "r");
61     if (fp == NULL) {
62         printf("Error opening the input file.\n");
63         return 0;
64     }
65
66     while (fgets(line, 100, fp)!=NULL && i<MAX_FLIGHTS) {
67         sscanf(line, "%s%s%s%f%f", flights[i].code,
68                 flights[i].depCity, flights[i].arrCity,
69                 &flights[i].depTime, &flights[i].arrTime);
70         i++;
71 }
```

```

71     }
72
73     fclose(fp);
74     return i;
75 }
76
77 /* search all flights satisfying the request
78  * 
79  */
80 void connection_search (flight_t *flights, int nf, flight_t request) {
81     int i, j, found;
82
83     for (i=0; i<nf; i++) {
84         if (strcmp(request.depCity, flights[i].depCity)==0 &&
85             request.depTime<=flights[i].depTime) {
86             if (strcmp(request.arrCity, flights[i].arrCity)==0) {
87                 /* found a direct flight: print info */
88                 printf("Direct flight:\n");
89                 info_write(flights[i]);
90             } else {
91                 /* it might exist a non direct flight: search */
92                 found = 0;
93                 for (j=0; j<nf && !found; j++) {
94                     if (strcmp(flights[j].depCity, flights[i].arrCity)==0 &&
95                         flights[j].depTime>=flights[i].arrTime &&
96                         strcmp(request.arrCity, flights[j].arrCity)==0) {
97                         /* found a non direct flight: print info */
98                         printf("Flight with one stop:\n");
99                         info_write(flights[i]);
100                        info_write(flights[j]);
101                        found = 1; /* print only one connection */
102                    }
103                }
104            }
105        }
106    }
107 }
108
109 /*
110  * print flight information
111  */
112 void info_write (flight_t flight) {
113     printf("%s ", flight.code);
114     printf("%s %s ", flight.depCity, flight.arrCity);
115     printf("%2.2f %2.2f\n", flight.depTime, flight.arrTime);
116
117     return;
118 }

```

## 1.27 Sudoku

### Problem definition

A standard Sudoku puzzle consists of a grid of 9 blocks. Each block contains 9 numbers arranged in 3 rows and 3 columns.

The puzzle can be considered solved correctly when all 81 numbers satisfy the following rules:

- ▷ Each column must contain all of the numbers 1 through 9 and no two numbers in

the same column of a Sudoku puzzle can be the same.

- ▷ Each row must contain all of the numbers 1 through 9 and no two numbers in the same row of a Sudoku puzzle can be the same.
- ▷ Each block must contain all of the numbers 1 through 9 and no two numbers in the same block of a Sudoku puzzle can be the same.

**Example 1.27** The following Sudoku scheme is correct.

9	5	2	8	7	6	4	3	1
6	8	1	9	4	3	5	7	2
7	3	4	1	5	2	9	6	8
4	9	6	5	3	1	8	2	7
5	1	7	2	8	4	3	9	6
3	2	8	6	9	7	1	4	5
8	4	3	7	6	5	2	1	9
2	6	5	4	1	9	7	8	3
1	7	9	3	2	8	6	5	4

**Example 1.28** The scheme in the following figure is wrong, 2 is duplicated in the second sub-matrix.

1	2	3	7	5	9	4	8	6
7	5	8	<b>2</b>	4	6	9	1	3
6	9	4	3	1	<b>2</b>	8	5	7
8	1	7	5	2	3	6	9	4
3	6	9	8	7	4	5	2	1
5	4	2	9	6	1	3	7	8
9	3	1	6	8	5	7	4	2
4	7	6	1	9	8	2	3	5
2	8	5	4	3	7	1	6	9

## Specifications

Write a program able to read and verify a Sudoku puzzle of variable size (but limited to 100 rows and 100 columns). The puzzle configuration has to be read from a file whose name is received by the program on the command line.

## Solution

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <math.h>
5
6 #define L      20
7 #define N      10
8 #define DIM   N*N
9

```

```
10 /* function prototypes */
11 int matrix_read (char [], int [] [DIM]);
12 int matrix_check (int [] [DIM], int, int);
13
14 /*
15  * main program
16 */
17 int main (int argc, char *argv[]) {
18     char fileName[L];
19     int matrix[DIM] [DIM];
20     int dim, n, error;
21
22     if (argc < 2) {
23         fprintf (stdout, "Input file name: ");
24         scanf ("%s", fileName);
25     } else {
26         strcpy(fileName, argv[1]);
27     }
28
29     dim = matrix_read (fileName, matrix);
30     n = floor (sqrt(dim));
31     error = matrix_check (matrix, n, dim);
32
33     if (error==0) {
34         fprintf (stdout, "Valid matrix.\n");
35     }
36
37     return 0;
38 }
39
40 /*
41  * read in the matrix
42 */
43 int matrix_read (char name[L], int matrix[DIM] [DIM]) {
44     FILE *fp;
45     int dim, r, c;
46
47     fp = fopen (name, "r");
48     if (fp==NULL) {
49         fprintf (stderr, "File open error.\n");
50         exit(EXIT_FAILURE);
51     }
52
53     if (fscanf (fp, "%d", &dim) != 1) {
54         fprintf (stderr, "File read error.\n");
55         exit(EXIT_FAILURE);
56     }
57
58     if (dim>DIM) {
59         fprintf (stderr, "Matrix on file too large to be stored in static matrix.\n");
60         exit(EXIT_FAILURE);
61     }
62
63     /* read the matrix */
64     for (r=0; r<dim; r++) {
65         for (c=0; c<dim; c++) {
66             if (fscanf (fp, "%d", &matrix[r][c]) != 1) {
67                 fprintf (stderr, "File read error.\n");
68                 exit(EXIT_FAILURE);
69             }
70         }
71     }
72 }
```

```

69     }
70 }
71 }
72 fclose (fp);
73
74     return dim;
75 }
76 }
77
78 /*
79 *   check for an error
80 */
81 int matrix_check (int matrix[][][DIM], int n, int dim) {
82     int r, c, i, j;
83     int v[N];
84
85     // Check Rows
86     for (r=0; r<dim; r++) {
87         for (c=0; c<dim; c++) {
88             v[c] = 0;
89         }
90         for (c=0; c<dim; c++) {
91             if (matrix[r][c]<1 || matrix[r][c]>dim || v[matrix[r][c]-1]!=0) {
92                 fprintf (stderr, "Error on row: Element [%d] [%d].\n", r, c);
93                 return 1;
94             } else {
95                 v[matrix[r][c]-1] = 1;
96             }
97         }
98     }
99
100    // Check Columns
101    for (c=0; c<dim; c++) {
102        for (r=0; r<dim; r++) {
103            v[r] = 0;
104        }
105        for (r=0; r<dim; r++) {
106            if (matrix[r][c]<1 || matrix[r][c]>dim || v[matrix[r][c]-1]!=0) {
107                fprintf (stderr, "Error on column: Element [%d] [%d].\n", r, c);
108                return 1;
109            } else {
110                v[matrix[r][c]-1] = 1;
111            }
112        }
113    }
114
115    // Check Blocks
116    for (r=0; r<dim; r+=n) {
117        for (c=0; c<dim; c+=n) {
118            for (i=0; i<dim; i++) {
119                v[i] = 0;
120            }
121            for (i=0; i<n; i++) {
122                for (j=0; j<n; j++) {
123                    if (matrix[r+i][c+j]<1 || matrix[r+i][c+j]>dim ||
124                        v[matrix[r+i][c+j]-1]!=0) {
125                        fprintf (stderr, "Error on block: Element [%d] [%d].\n", r+i, c+j);
126                        return 1;
127                    } else {

```

```
128         v[matrix[r+i][c+j]-1] = 1;
129     }
130 }
131 }
132 }
133 }
134 return 0;
135 }
```

# Chapter 2

## Memory, Variables, and Pointers

In this chapter, we discuss one of the most powerful features of the C programming language, the pointer. Pointers are among C's most difficult capabilities to master. Pointers enable programs to:

- ▷ Simulate pass-by-reference, to pass variable by reference between functions.
- ▷ Create and manipulate dynamic data structures, i.e., data structures that can grow and shrink at execution time, such as linked lists, queues, stacks and trees.

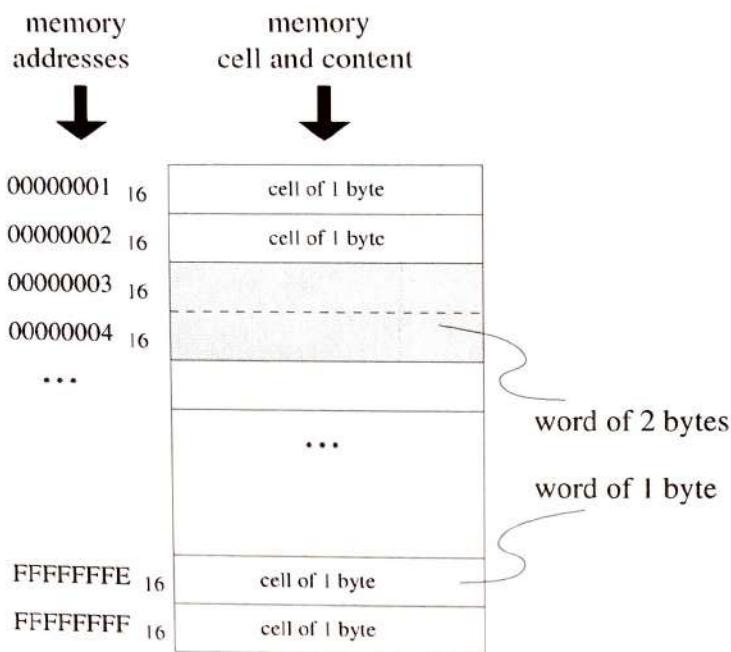
This chapter explains basic pointer concepts.

### 2.1 Memory

A computer memory can be seen as a sequence of cells. In modern computers, cells have usually a length equal to 8 bits, i.e., one byte. A set of cells, usually 2, 4 or 8 cells, usually defines a *word*.

A memory address (or physical memory address) is a reference to a specific memory cell. Memory addresses are fixed-length sequences of digits, conventionally displayed and manipulated as unsigned integers. Such numerical semantic based itself upon features of Central Processing Units (CPUs), as well upon use of the memory like arrays. The memory controllers' bus consists of number of parallel lines, each represented by a binary digits (or bits). The width of the bus, and thus the number of addressable storage units, and the number of bits in each unit, varies among computers. In modern architectures, addresses are usually of 32 or 64 bits. So, 32-bit processors always read 4 bytes at a time, and 64-bit processors always read 8 bytes at a time.

Figure 2.1 shows a memory with address of 32 bits, i.e., 8 hexadecimal digits. Historically memory is byte addressable and arranged sequentially. If the memory is arranged as cells of one byte width, the processor needs to issue 4 memory read cycles to fetch an data type stored on 4 bytes. Thus, it is more economical to read more than 1 bytes in one memory cycle. To take such advantage, the memory will be arranged as group of 4 bytes.



**Figure 2.1** Memory logic structure: A sequence of cells.

## 2.2 Variables and Padding

Variable names correspond to locations in the computer's memory. These locations are not necessarily adjacent in memory, and each one spans a certain number of bytes starting at a particular memory location, or address. The number of bytes essentially depends on the variable type. As the C language is flexible in its storage requirements for the fundamental types, the situation can vary from one architecture to another. Usually:

$$\begin{aligned} \text{sizeof}(char) &= 1 \\ \text{sizeof}(char) \leq \text{sizeof}(short) \leq \text{sizeof}(int) &\leq \text{sizeof}(long) \\ \text{sizeof}(float) \leq \text{sizeof}(double) &\leq \text{sizeof}(longdouble) \end{aligned}$$

that is, character variables are usually 1 bytes long, integer 4 bytes, etc. As variables often need more than one byte and standard memory cells are usually longer than one byte, each C variable can be stored on only a part of a single cell, on an entire cell, or on more than one cell depending on the variable and the cell length. For example, if an integer value is allocated at an address other than multiple of 4, it spans across two cells. Thus, such an integer will require two memory read cycle to be fetched. In other words, any misalignment will force more reading cycles than necessary. Then, data type in C have alignment requirements. *Padding* aligns variables to "natural" address boundaries, i.e., 4 bytes on 32-bit platforms. In other words, in order to align the data in memory, one or more empty bytes are inserted (or left empty) between memory addresses which are allocated for different variables. This concept is called *padding*. Padding in C is activated by default. This is particularly common in C structures, i.e., because of the alignment requirements of various data types, every member of structure is usually naturally aligned.

Pointers are often used in programs to access memory and manipulate addresses

and variables.

**Example 2.1** Consider the following structure with 5 members:

```
struct student1 {
    int id1;
    int id2;
    char a;
    char b;
    float percentage;
};
```

For 32-bit processor, integer and float types occupy 4 bytes each, and characters occupy 1 byte. Thus, only 14 bytes ( $4 + 4 + 1 + 1 + 4$ ) should be allocated for the above structure. To increase the processor speed, the size of this structure is usually of 16 bytes as 2 extra bytes are left empty. The memory configuration is shown in Figure 2.2.

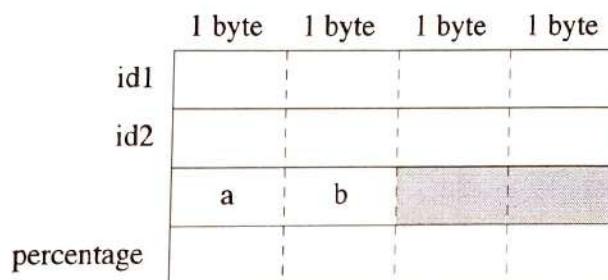


Figure 2.2 Structure student1 stored with padding, 4 cells of 4 bytes, 16 bytes overall.

**Example 2.2** Consider the following structure with the same 5 members present in structure student1 but defined with a different order:

```
struct student2 {
    int id1;
    char a;
    int id2;
    char b;
    float percentage;
};
```

In this case memory occupation should be 20 bytes instead of 14. The memory configuration is shown in Figure 2.3.

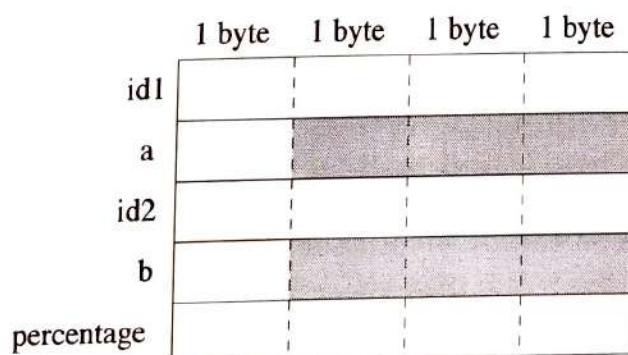


Figure 2.3 Structure student2 stored with padding, 5 cells of 4 bytes, 20 bytes overall.

## 2.3 Pointer Variables

Pointers are variables whose values are memory addresses. They are often used in C to manipulate objects.

Normally, a variable directly contains a specific value. A pointer, on the other hand, contains an address of a variable that contains a specific value. In this sense, a variable name *directly* references a value, and a pointer *indirectly* references a value. Referencing a value through a pointer is called indirection. Pointers, like all variables, must be defined before they can be used.

The definition:

```
<type> *<pointer>;
```

specifies that variable *<pointer>* is of type *<type> \**, i.e., a pointer to *<type>*. It can be rewritten as:

```
<type> * <pointer>;
```

or

```
<type>*<pointer>;
```

but we will use the first one in this book. These definitions are read as (right to left), i.e., “*<pointer>* is a pointer to *<type>*” or “*<pointer>* points to an object of type *<type>*.” Notice that C pointers point to specific types, ad that this is true for all pointers but pointers of type `void *`.

### Example 2.3 The definition:

```
int *countPtr, count;
```

specifies that variable `countPtr` is of type `int *`, i.e., a pointer to an integer. Also, the variable `count` is defined to be an `int`, not a pointer to an `int`. The `*` applies only to `countPtr` in the definition. When `*` is used in this manner in a definition, it indicates that the variable being defined is a pointer.

Figure 2.4(a) shows a possible memory allocation of these two objects, with the pointer allocated before the integer variable. The right-hand side of the memory array indicates a possible value for those memory addresses in hexadecimal form on a 32 bit architecture. Notice that both the integer value and the pointer are not initialized, and the their value its then undefined (i.e., equal to “?”).

Pointers can be defined to point to objects of any type. To prevent the ambiguity of declaring pointer and non-pointer variables in the same declaration as shown above, you should always declare only one variable per declaration.

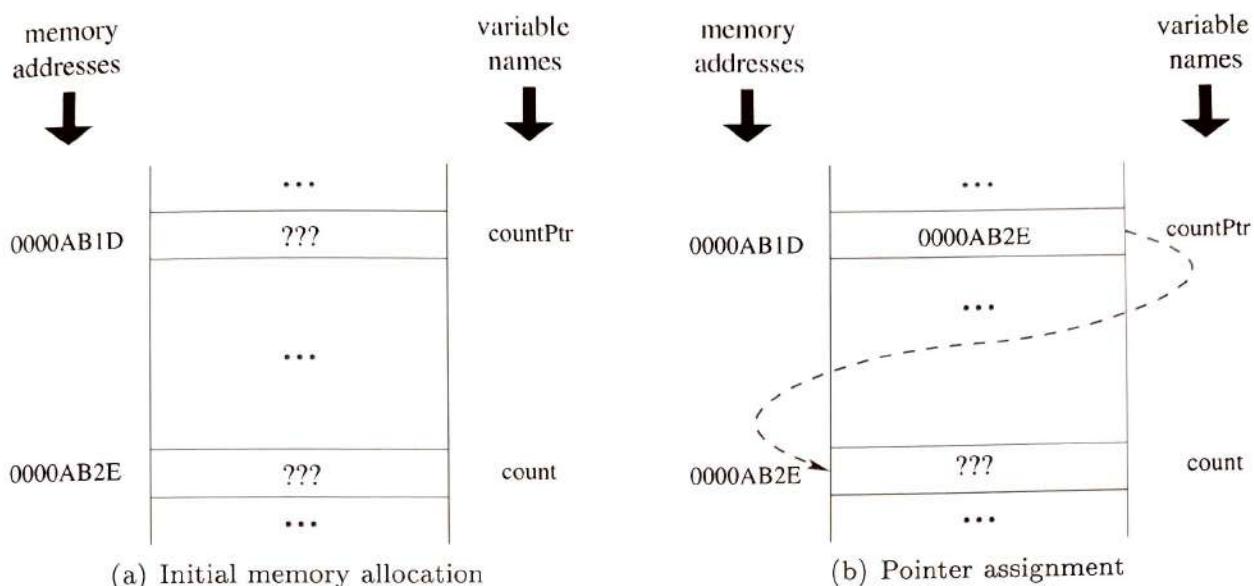
## 2.4 Pointer Operators

### 2.4.1 The Indirection Operator

Pointers have been defined using the `*` operator. The unary `*` operator, commonly referred to as the *indirection operator* or *dereferencing operator*, returns the value of the object to which its operand (i.e., a pointer) points.

### Example 2.4 Extending Example 2.3 we write:

```
int *countPtr, count;
...
countPtr = &count;
```



**Figure 2.4** A pointer to an integer and an integer: Memory allocation and assignment. Those two objects are initially undefined (a). Then, the pointer has a specific value, i.e., it points to variable count (b).

Now the value of `count` is still undefined, but the one of `countPtr` is not, as `countPtr` is now referring `count`. Figure 2.4(b) shows the new memory configuration.

**Example 2.5** If `ptr` is a pointer to a float, then `*ptr` takes the value referred by `ptr`. For example, in the following piece of code:

```
float *ptr;  
float f;
```

```
f = 7.5;
ptr = &f;
fprintf (stdout, "%f %f\n", f, *ptr);
forprintf (stdout, "%lu\n", (long unsigned int) ptr);
```

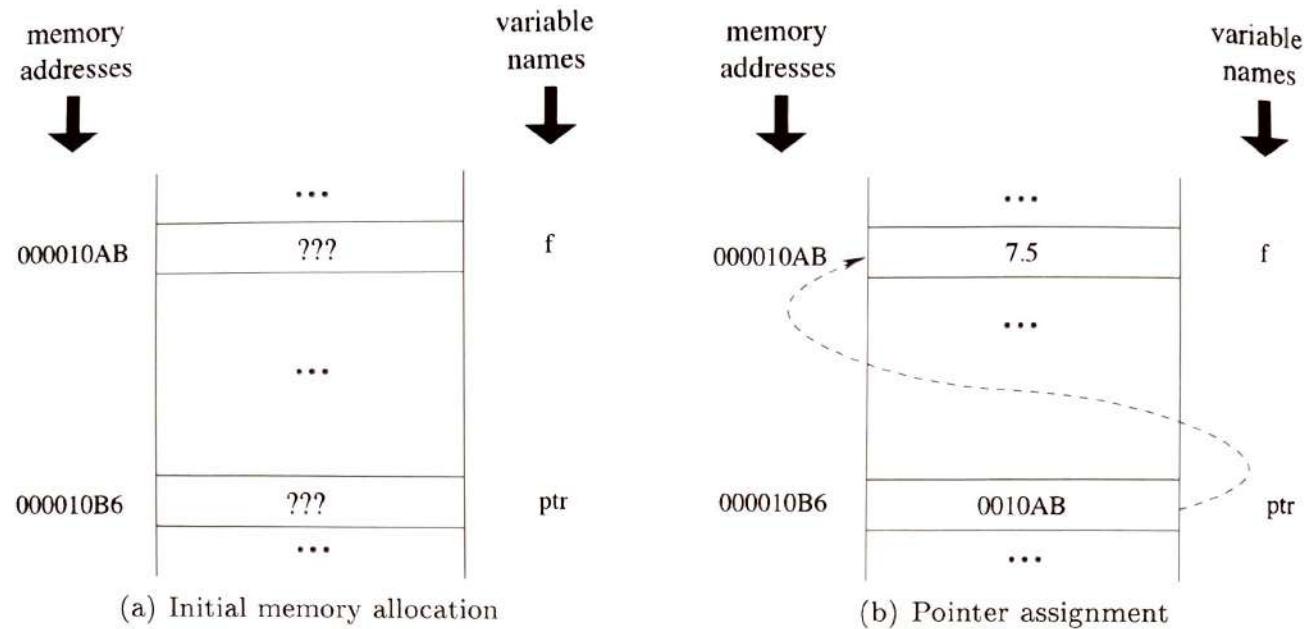
the first `fprintf` prints-out 7.5 twice, whereas the second one prints-out the address of `f` as a long unsigned integer value (`%lu`). This is basically unknown, and depends on where the system has allocated the variable `f` within the available memory. Figure 2.5 shows the memory configuration just after the definition of the two objects and after both assignments. Following this picture the second `fprintf` would print the hexadecimal value `000010B6` into its decimal form, i.e., 4278.

Dereferencing a pointer that has not been properly initialized or that has not been assigned to point to a specific location in memory is an error. This could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion with incorrect results.

### 2.4.2 The Address Operator

The `&`, or *address operator*, is a unary operator that returns the address of its operand. Given the definitions

<type> <var>, \*<ptr>;



**Figure 2.5** A pointer to an integer and an integer: Memory allocation and assignment.

```
<ptr> = &<var>;
```

the pointer `<ptr>` assume the address of variable `<var>`.

**Example 2.6** For example, assuming the definitions

```
int y = 5;
int *yPtr;
```

the statement

```
yPtr = &y;
```

assigns the address of the variable `y` to pointer variable `yPtr`. Variable `yPtr` is then said to “point to” `y`.

### 2.4.3 Relationship between Indirection and Address Operators

The `&` and `*` operators are complements of one another. When they are both applied consecutively to a pointer or to a variable (in either order), the same result is obtained.

**Example 2.7** For example, assuming the definitions

```
int v, *p;
p = &v;
```

`&v` is the address of `v`, and `*(&v)` coincides with `v`. Moreover, `*p` is the content referenced by `p` (i.e., `v`), and `&(*p)` coincides with `p`.

Thus, if `p` is a pointer `&(*p)` (and also `*(&p)`) coincides with the pointer itself. If `v` is a variable `*(&v)` (and also `&(*v)`) coincides with the variable itself.

## 2.5 NULL and void Pointers

Pointers should be initialized (they should be assigned a value) during or after their definition. If they are not initialized, they are undefined, i.e., they can point to anything. If they are initialized, they can be assigned to NULL, to 0, or to an address. A pointer with the value NULL points to nothing.

NULL is a symbolic constant defined in the `<stddef.h>` header (and several other headers, such as `<stdio.h>`). Initializing a pointer to 0 is equivalent to initializing a pointer to NULL, but NULL is greatly preferred. Notice that the value 0 is the only integer value that can be assigned directly to a pointer variable.

When 0 is assigned to a pointer, it's first converted to a pointer of the appropriate type. Anyway, conversions during assignments between different pointer types can be problematic: They are usually allowed by standard C but not by ANSI C. The only safe cast operation between pointer types (beyond the 0 constant) is allowed when one of the type is a pointer to `void`. Thus, we can think of `void *` as a generic pointer type. A pointer of type `void *` is not associated to any type of data, i.e., it is a generic pointer which can be converted to any type. This is an important issue, as many library functions return `void` pointers to be generic. We will present in this book (namely in Chapter 9), how to use a returned object of type `void *`.

## 2.6 Pointer Arithmetic

Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions. However, not all the operators normally used in these expressions are valid in conjunction with pointer variables.

A limited set of arithmetic operations may be performed on pointers. A pointer may be incremented or decremented, an integer may be added to or subtracted from a pointer, and one pointer may be subtracted from another. The last operation is meaningful only when both pointers point to contiguously allocated cells, such as elements belonging to the same array.

When an integer is added to or subtracted from a pointer, the pointer is not incremented or decremented simply by that integer, but by that integer times the size of the object to which the pointer refers. For example, if after defining

```
int *p;
```

we do `p++`, the value of `p` is not incremented by 1, but by a value that is  $1 \cdot \text{sizeof}(int)$ , which is usually 4. The pointer arithmetic is done “modulo” the number of bytes of the object's data type. In other word, incrementing a pointer by 1 means making the pointer referring to the next element of the same type in memory.

When performing pointer arithmetic on a character pointer, the results will be consistent with regular arithmetic, because each character is 1 byte long.

A pointer can be assigned to another pointer if both have the same type. The exception to this rule is the pointer to `void` (i.e., `void *`), which is a generic pointer that can represent any pointer type. All pointer types can be assigned a pointer to `void`, and a pointer to `void` can be assigned a pointer of any type. In both cases, a cast operation is not required but it can be useful to explicitly define the programmer's intention. A pointer to `void` cannot be dereferenced.

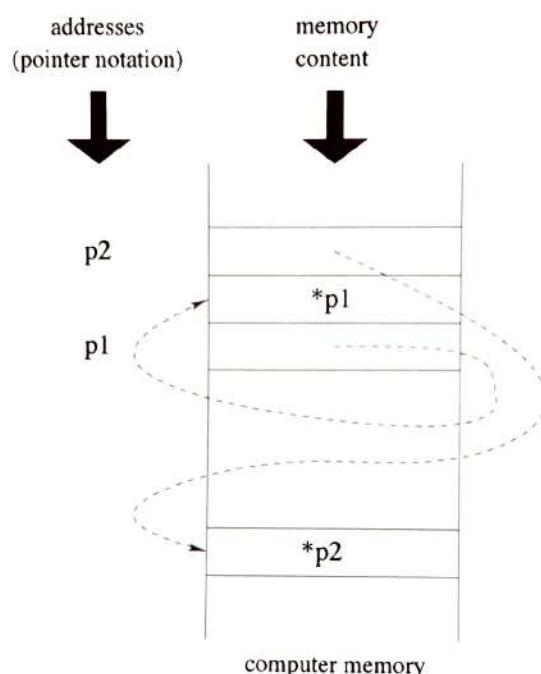
Pointers can be compared using equality and relational operators, but such comparisons are meaningless unless the pointers point to elements of the same array. Pointer

comparisons compare the addresses stored in the pointers. A comparison of two pointers pointing to elements in the same array could show, for example, that one pointer points to a higher-numbered element of the array than the other pointer does. A common use of pointer comparison is determining whether a pointer is NULL.

**Example 2.8** The following lines of code:

```
int *p1, *p2;
...
p1 = ...;
...
p2 = ...;
```

create the situation represented in Figure 2.6 into the main memory.



**Figure 2.6** Comparison of two pointers.

With these premises, checking whether:

- ▷  $(\ast p1) == (\ast p2)$  means checking whether the referenced values are the same even if they are placed in different position within the system memory.
- ▷  $p1 == p2$  means checking whether the two pointers refer to the same object, i.e., they store the same memory address. Obviously, if  $p1 == p2$  also the content of those cells will be the same, but those values can be other than scalar value, i.e., it may be impossible to compare  $\ast p1$  with  $\ast p2$  directly.
- ▷  $p1 >= p2$  means checking whether the address `p1` comes after the address `p2` into the system memory. This may be meaningless, unless the cell referenced by `p1` and `p2` share something in common, i.e., they belong to the same array.

## 2.7 Relationship between Pointers and Arrays

Arrays and pointers are intimately related in C and often may be used interchangeably. An array name can be thought of as a *constant* pointer. Pointers can be used to do

any operation involving array sub-scripting. Moreover, the array name (without a subscript) is a pointer to the first element of the array.

Arrays automatically allocate space, but they can not be relocated or re-sized. Pointers must be explicitly assigned to point to allocated space (using `malloc`, `calloc` or `realloc`, which we will present in the following sections). Moreover, they can be reassigned (i.e., pointed at different objects) at will, and have many other uses besides serving as the base of blocks of memory.

Due to the so-called equivalence of arrays and pointers, arrays and pointers often seem interchangeable, and in particular a pointer to a block of memory assigned by `malloc` (see Section 3.1.1) is frequently treated (and can be referenced using the `[]` notation) exactly as if it were a true array.

Let us analyze the following example to better understand the array-to-pointer relationship.

**Example 2.9** Assume the following definitions

```
int v[10];
int *p;
```

Because the array name is a pointer to the first element of the array, with the instruction  
`p = v;`

we set `p` equal to the address of the first element in array `v`. This statement is equivalent to  
`p = &v[0];`

After that, element `v[5]` can alternatively be referenced with the pointer expression `* (p + 5)`. The 5 in the expression is the offset to the pointer. When the pointer points to the array's first element, the offset indicates which array element should be referenced, and the offset value is identical to the array subscript. This notation is referred to as pointer/offset notation and it is represented in Figure 2.7.

The parentheses in `* (p + 5)` are necessary because the precedence of `*` is higher than the precedence of `+`. Without the parentheses, i.e., `*p + 5`, the expression would add 5 to the value of `*p` (i.e., 5 would be added to `v[0]`, assuming `p` points to the beginning of the array).

Just as the array element can be referenced with a pointer expression, the address `&v[9]` can be written with the pointer expression `p+9`.

The array itself can be treated as a pointer and used in pointer arithmetic. For example, the expression `* (v+3)` also refers to the array element `v[3]`.

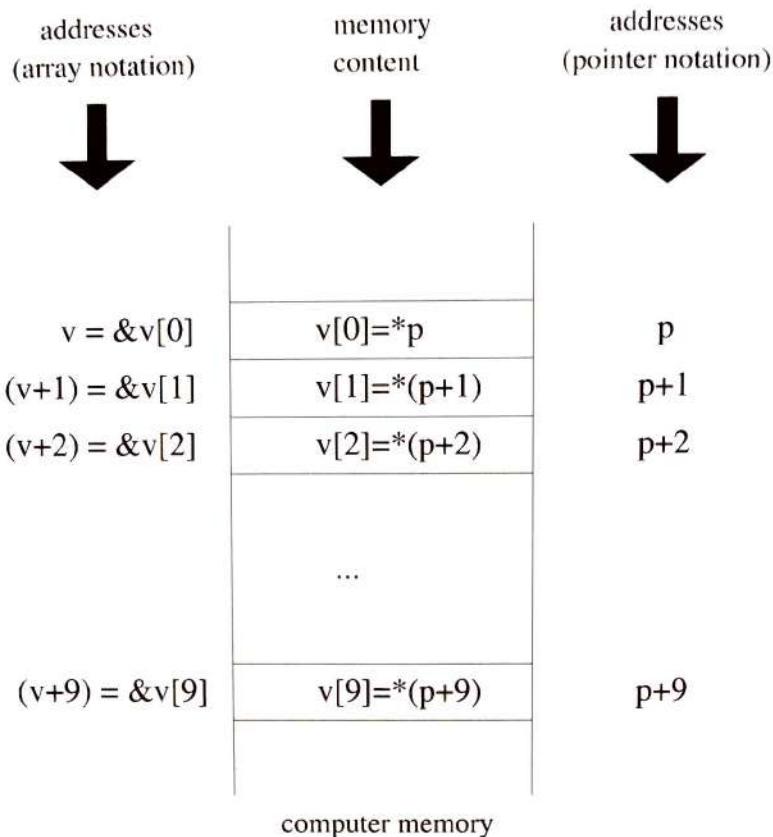
In general, all sub-scripted array expressions can be written with a pointer and an offset. In this case, pointer/offset notation was used with the name of the array as a pointer. The preceding statement does not modify the array name in any way. `v` still points to the first element in the array. Pointers can be sub-scripted like arrays. If `p` has the value `v`, the expression `p[1]` refers to the array element `v[1]`. This is referred to as pointer/subscript notation.

Remember that an array name is essentially a constant pointer. It always points to the beginning of the array. Thus, the expression `v += 3` is invalid, i.e., it is a compilation error, because it attempts to modify the value of the array name with pointer arithmetic.

The following is a further example on how an array can be manipulated using different notations.

**Example 2.10** The following lines of code:

```
int i, v[L];
...
for (i=0; i<L; i++) {
    scanf ("%d", &v[i]);
```



**Figure 2.7** Pointer to array correspondence.

```
    printf ("%d", v[i]);
}
```

reads and array  $v$  of size  $L$  and prints it on standard output. The code can be rewritten using pointer arithmetic as follows:

```
int i, *p, v[L];
...
for (i=0, p=&v; i<L; i++, p++) {
    scanf ("%d", p);
    printf ("%d", *p);
}
```

or as:

```
int i, *p, v[L];
...
p = &v;
for (i=0; i<L; i++) {
    scanf ("%d", (p+i));
    printf ("%d", *(p+i));
}
```

Referring to this last piece of code, note again that  $*(p+5)$  and  $*p + 5$  have a different meaning. The first one takes the pointer  $p$ , adds 5 to it, and it takes the referenced value, i.e., it takes the value 5 positions after  $p$ . The second one takes the cell content pointed by  $p$  and it adds 5 to this value.

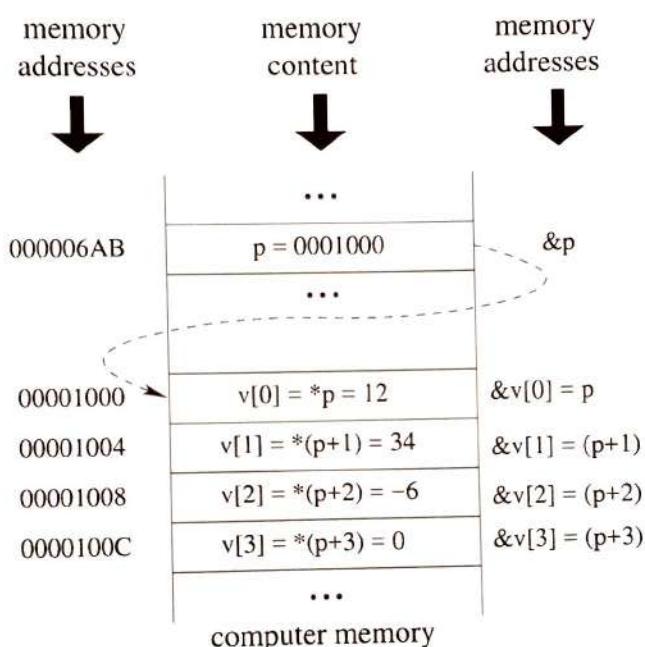
To be a little bit more concrete, the following example illustrate a possible array placement into a main computer memory.

**Example 2.11** Let us suppose to define

```
int v[] = {12, 34, -6, 0};
int *p;
...
p = v;
```

which allocate variables *v* and *p* as illustrated in Figure 2.8. Notice that all memory positions and addresses are selected casually, to be congruent with a 32 bit architecture. Moreover, the example suppose that integers are stored on 4 bytes.

If we perform instruction *p++*, *p* will not be incremented by 1, but by a value that is equal to the number of bytes occupied by an integer, that is,  $1 \cdot \text{sizeof}(int)$ , which we supposed to be equal to 4. In other word, incrementing a pointer by 1 means making the pointer referring to the next element of the same type in memory. As a consequence the first integer is referenced by *\*p*, the second one by *\*(p+1)*, the third one by *\*(p+3)*, etc., as shown in the previous example.



**Figure 2.8** Pointer arithmetic.

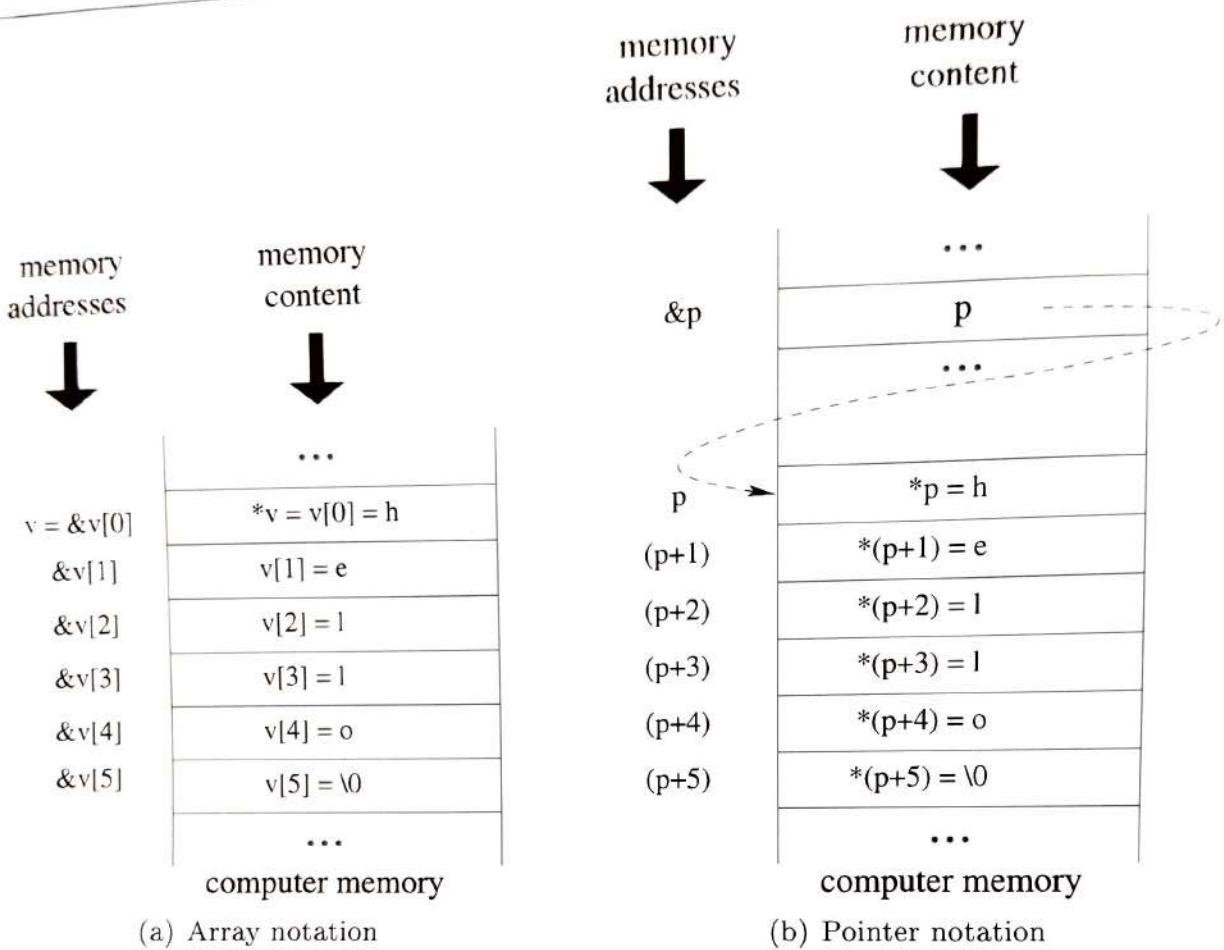
Notice again that parenthesis () are required: *\*(p+1)* takes 34, whereas *\*p+1* refers to 12 and add 1 to this value, returning 13.

It is important to notice that, albeit sharing several “similarity”, arrays and pointers are not identical. Arrays decay into pointers as formal parameters to functions, but arrays are not pointers. This is illustrated by the following example.

**Example 2.12** The following definitions

```
char v[] = "hello";
char *p = "hello";
```

will initialize data structures as represented in Figure 2.9. The array declaration requests that space for six characters be set aside, to be known by the name *v*. That is, there is a location place which holds a pointer, to be known by the name *p*. This pointer can point almost anywhere, i.e., to any character, or to any contiguous array of characters, or nowhere. In this case, it will point to the constant string "hello".



**Figure 2.9** Array, pointer and string correspondence.

It is important to realize that a reference like  $x[3]$  generates different code depending on whether  $x$  is the array  $v$  or the pointer  $p$ . Given the declarations above, when the compiler sees the expression  $v[3]$ , it emits code to start at the location  $v$ , move three past it, and fetch the character there. When it sees the expression  $p[3]$ , it emits code to start at the location  $p$ , fetch the pointer value there, add three to the pointer, and finally fetch the character pointed to. In other words,  $v[3]$  is three places past (the start of) the object “named”  $v$ , while  $p[3]$  is three places past the object “pointed to” by  $p$ . In the example above, both  $v[3]$  and  $p[3]$  happen to be the character ‘l’, but the compiler gets there differently. The values of an array and a pointer are computed differently whenever they appear in expressions.

Given the previous example, saying that arrays and pointers are “equivalent” means neither that they are identical nor even interchangeable. However, array and pointer arithmetic is defined such that a pointer can be conveniently used to access an array or to simulate an array. Specifically, the cornerstone of the equivalence is due to the fact that when array names appear in an expression they decay into a pointer to its first element. That is, whenever an array appears in an expression, the compiler implicitly generates a pointer to the array’s first element, just as if the programmer had written  $\&v[0]$ <sup>1</sup>. As a consequence of this definition, the compiler doesn’t apply the array sub-

<sup>1</sup> Notice that this rule has 3 exceptions: (1) The array is the operand of a `sizeof` operator, (2) array.

scripting operator [ ] that differently to arrays and pointers, after all. In an expression of the form `v[i]`, the array decays into a pointer, following the rule above, and is then sub-scripted just as would be a pointer variable in the expression `p[i]` (although the eventual memory accesses will be different, as previously explained).

Anyway, since arrays decay immediately into pointers, an array is never actually passed to a function. Allowing pointer parameters to be declared as arrays is simply a way of making it look as though an array was being passed, perhaps because the parameter will be used within the function as if it were an array. Specifically, any parameter declarations which “look like” arrays, e.g.,

```
void f(char v[]) { ... }
```

are treated by the compiler as if they were pointers, since that is what the function will receive if an array is passed:

```
void f(char *v) { ... }
```

This conversion holds only within function formal parameter declarations, nowhere else. If the conversion bothers you, avoid it; many programmers have concluded that the confusion it causes outweighs the small advantage of having the declaration “look like” the call or the uses within the function.

The following example analyzes how to use the pointer notation with functions manipulating an array or a part of an array.

**Example 2.13** Let us suppose function `sort` receives an array of integers `v` and its size `n` to sort it in ascending order. Its prototype should look like:

```
void sort (int *v, int n);
```

and its called like:

```
sort (v, n);
```

Now, this function call is equivalent to the following one:

```
sort (&v[0], n);
```

which in turn, enables calls like the following one:

```
sort (&v[i], n-i);
```

in which function `sort` receives the array of integers starting at element `i` and including `n-i` elements. In this case, function `sort` will order the sub-array, within the array `v`, starting at element in position `i` and including `n-i` elements.

## 2.7.1 Relationship between Pointers and Strings

To better understand the relationships between pointers array, this section analyzes how it is possible to implement a couple of string manipulation functions.

For example, function `strlen` (whose prototype is included into library `string.h`) given a string computes and returns its length, i.e., the number of character included in the string before the termination character. A first possible implementation manipulates the string using the array notation.

```
int strlen (char str[]) {
    int cnt;
    cnt = 0;
    while (str[cnt] != '\0')
        cnt++;
}
```

```

    return cnt;
}

```

The same function header can be used with a different function implementation, i.e., instead of manipulating the string with its index, it is obviously possible to manipulate it using a pointer.

```

int strlen (char str[]) {
    int cnt;
    char *p;

    cnt = 0;
    p = &s[0];
    while (*p != '\0') {
        cnt++;
        p++;
    }

    return cnt;
}

```

Now, even the function header can be changed to reflect the pointer nature of its implementation:

```

int strlen (char *str) {
    int cnt;

    cnt = 0;
    while (*str != '\0') {
        cnt++;
        str++;
    }

    return cnt;
}

```

A further step in this process, can be done avoiding the use of the explicit length counter `cnt`, as in the following implementation:

```

int strlen (char *str) {
    char *p;

    p = str;
    while (*p != '\0') {
        p++;
    }

    return (p - str);
}

```

where the string length is given by the difference of two pointers (using the pointer arithmetic logic) `p`, pointing to its last character, and `str` referring to its first character.

A second example is given by function `strcmp` which compares two strings. A first possible array-based implementation, is the following one:

```

int strcmp (char str1[], char str2[]) {
    int i;

    i = 0;
    while ((str1[i]==str2[i]) && (str1[i]!='\0')) {
        i++;
    }
}

```

```

return (str1[i] - str2[i]);
}

```

Notice that the function returns the difference of the ASCII representation of the characters at position *i*. If this difference is larger than (less than) 0 the first (second) string precedes the second one alphabetically. If the difference is 0 the two strings are the same string. A possible pointer-based coding is the following one:

```

int strcmp (char *str1, char *str2) {
    while ((*str1==*str2) && (*str1!='\0')) {
        str1++;
        str2++;
    }

    return (*str1 - *str2);
}

```

## 2.8 Relationship between Pointers and Structures

In C, structures can be passed as arguments to functions and can be returned from them. When a structure is passed as an argument to a function, if it is passed by value a local copy is made for use in the body of the function. If a member of the structure is a variable (or an array), then the variable (or the array) is copied as well. This implies that any modification to the variable (or the array) is made on the local copy and remains local to the function itself. As a consequence an easy question arises: How can we pass a structure to a function such that this function is able to modify the structure fields? First of all, we have to point out that if a member of the structure is a pointer, then the pointer is copied into the destination structure such that both the original structure (in the caller) and the new structure (in the function) point to the same object. Thus, any modification to this object will be visible within the function and from the caller as well. On the contrary, if we want to modify static fields within the structure, we have to pass the structure by reference. Thus, within the function the structure must be manipulated using its pointer not its variable name.

We have already seen the use of the member access operator “.”. C provides the member access operator  $\rightarrow$  to access the members of a structure via a pointer<sup>2</sup>. If a pointer variable is assigned the address of a structure, then a member of the structure can be accessed by a construct of the form

`pointer_to_structure->member_name`

A construct that is equivalent to this is

`(*pointer_to_structure).member_name`

The parentheses are necessary. Along with ( ) and [ ], the operators . and  $\rightarrow$  have the highest precedence and associate from left to right. Thus, the previous construct without parentheses would be equivalent to

`* (pointer_to_structure.member_name)`

This is an error because only a structure can be used with the . operator, not a pointer to a structure. Let us illustrate the use of  $\rightarrow$  by writing a function to manipulate some structure fields.

<sup>2</sup> Notice that the operator  $\rightarrow$  is typed on the keyboard as a minus sign – followed by a greater than sign >.

**Example 2.14** Let us suppose we have the following data structure:

```
struct student {
    char first_name[L], last_name[L];
    int register_number;
    float average;
};
```

and that we want to read an object of this type within function `read`. The caller would do something like:

```
struct student v;
...
read (&v);
...
```

and function `read` would be like the following.

```
void read (struct student *v) {
    char first_name[DIM], last_name[DIM];
    int rn, a;

    fprintf (stdout, "First name, last names, register number, average: ");
    scanf ("%s%s%d%d", first_name, last_name, &rn, &a);
    strcpy (v->first_name, first_name);
    strcpy (v->last_name, last_name);
    v->register_number = rn;
    v->a = a;

    return;
}
```

Note again that `v` is a pointer to structure and function `read` has to manipulate it accordingly. An alternative implementation can use the `*` and `.` operators, as the following.

```
void read (struct student *v) {
    char first_name[DIM], last_name[DIM];
    int rn, a;

    fprintf (stdout, "First name, last names, register number, average: ");
    scanf ("%s%s%d%d", first_name, last_name, &rn, &a);
    strcpy ((*v).first_name, first_name);
    strcpy ((*v).last_name, last_name);
    (*v).register_number = rn;
    (*v).a = a;

    return;
}
```

which is equivalent, but much less used by C programmers.

## 2.9 Array of pointers

Arrays may contain pointers. A common use of an array of pointers is to form an array of strings, referred to simply as a string array. Each entry in the array is a string, but in C a string is essentially a pointer to its first character. So each entry in an array of strings is actually a pointer to the first character of a string.

**Example 2.15** Consider the definition of following array of strings

```
const char *suit[4] = { "Hearts", "Diamonds", "Clubs", "Spades" };
```

which may be useful to represent a deck of cards. The `suit [4]` portion of the definition indicates an array of 4 elements. The `char *` portion of the declaration indicates that each element of array `suit` is of type “pointer to char.” Qualifier `const` indicates that the strings pointed to by each element pointer will not be modified. The four values to be placed in the array are “Hearts”, “Diamonds”, “Clubs” and “Spades”. Each is stored in memory as a null-terminated character string that’s one character longer than the number of characters between quotes. The four strings are 7, 9, 6 and 7 characters long, respectively. Although it appears as though these strings are being placed in the `suit` array, only pointers are actually stored in the array. Each pointer points to the first character of its corresponding string. Thus, even though the `suit` array is fixed in size, it provides access to character strings of any length.

This flexibility is one example of C’s powerful data-structuring capabilities. The suits could have been placed in a two-dimensional array, in which each row would represent a suit and each column would represent a letter from a suit name. Such a data structure would have to have a fixed number of columns per row, and that number would have to be as large as the largest string. Therefore, considerable memory could be wasted when storing a large number of strings of which most were shorter than the longest string. Detailed comments on this issue will be reported in Section 3.8.

## 2.10 Local arrays with variable size

Array dimensions in C traditionally had to be compile-time constants. This means that it was impossible to declare local arrays of a size matching a variable value. In other words, it was impossible to write code such as:

```
scanf ("%d", &n);
...
{
int v[n];
...
}
```

or as in:

```
void f (int n) {
    int v[n];
    ...
}
```

where a formal parameter is used to define the size of a local array.

The C standard ISO/IEC 9899 1999 (C9X) introduced Variable-Length Arrays (VLA’s) which allow this sort of definitions, where local arrays may have sizes set by variables or other expressions, perhaps involving function parameters.

However, we will not use this sort of constructs in this book, for the following reasons:

- ▷ Dynamic memory allocation, with `malloc`, `calloc`, and `realloc`, is a super-set of what VLAs allow the programmer to realize.
- ▷ Run-time allocation is risky, as the object size is defined at run-time, and there is no proper checking strategy.
- ▷ VLAs are local objects, and, as such, they cannot be exported (see Section 3.2.1). In other words, they are automatically deallocated once the environment in which they have been created is abandoned and they cannot be used outside that environment.

# Chapter 3

## Dynamic Memory Allocation

In this chapter, we introduce array-like dynamic data structures that can grow and shrink at execution time. We will heavily used the notation and concepts introduced in Chapter 2.

### 3.1 Dynamic Memory Allocation

When a variable is defined in the source program, the type of the variable determines how much memory the compiler allocates. When the program executes, the variable consumes this amount of memory regardless of whether the program actually uses the memory allocated. This is particularly true for arrays.

However, in many situations, it is not clear how much memory the program will actually need. For example, we may have declared arrays to be large enough to hold the maximum number of elements we expect our application to handle. If too much memory is allocated and then not used, there is a waste of memory. If not enough memory is allocated, the program is not able to fully handle the input data. We can make our program more flexible if, during execution, it could allocate initial and additional memory when needed and free up the memory when it is no more needed.

Allocation of memory during execution is called *dynamic memory allocation*. In other words, creating and maintaining dynamic data structures requires dynamic memory allocation, i.e., the ability for a program to obtain more memory space at execution time to hold new nodes, and to release space no longer needed. C provides library functions to allocate and free up memory dynamically during program execution. Dynamic memory is allocated on the heap by the system (please refer to Chapter 8 for further details on this issue). It is important to realize that dynamic memory allocation also has limits. If memory is repeatedly allocated, eventually the system will run out of memory.

In C, there is a family of four functions which allow programs to manage dynamically allocated memory on the heap: `malloc`, `calloc`, `realloc`, and `free`. In order to use these functions you have to include the `stdlib.h` header file in your program. The operator `sizeof` is also essential to dynamic memory allocation.

### 3.1.1 Function malloc

Function `malloc` takes as an argument the number of bytes to be allocated and returns a pointer of type `void *` (pointer to void) to the allocated memory. This pointer is just a byte address. Thus, it does not point to an object of a specific type. A pointer type that does not point to a specific data type is said to point to `void` type. A `void *` pointer may be assigned to a variable of any pointer type, i.e., we have to type cast the value to the type of the destination pointer. Function `malloc` is normally used with the `sizeof` operator to allocate a proper quantity of memory. All objects stored in the chunk of memory reserved are usually of the same type.

The prototype for this function is

```
void *malloc (size_t number_of_bytes);
```

That is to say:

- ▷ It receives the size `number_of_bytes` of the desired portion of the memory.
- ▷ It returns a pointer of type `void *` to the reserved piece of memory reserved for the user. If the memory cannot be allocated a `NULL` pointer is returned.

Notice that `size_t` is a type used as arguments for functions that require sizes or counts specifications. This represents an unsigned value generally defined in header files as `unsigned int`.

#### Example 3.1 The piece of code

```
char *p;  
...  
p = (char *) malloc (100);
```

assigns to the `p` pointer a usable block of 100 bytes. The same does the following piece of code:

```
char *p;  
...  
p = malloc (100);
```

where the cast (from `void *` to `char *`) is not made explicit.

#### Example 3.2 For example, the statement

```
struct node *newPtr;  
  
newPtr = malloc (1 * sizeof (struct node));
```

evaluates `sizeof(struct node)` to determine the size in bytes of a structure of type `struct node`, allocates a new area in memory of that number of bytes and stores a pointer to the allocated memory in variable `newPtr`. The allocated memory is not initialized. If no memory is available, `malloc` returns `NULL`.

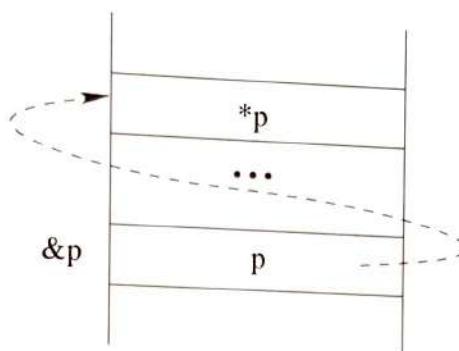
In practice, one must always verify whether the pointer returned is `NULL` or not. If the allocation is successful, objects in dynamically allocated memory can be accessed indirectly by dereferencing the pointer, appropriately cast to the type of required pointer. If it is unsuccessful, the programmer has to figure out how to behave (i.e., exit the program or try to proceed in a different way).

#### Example 3.3 In the following piece of code

```
int *p;
```

```
p = (int *) malloc (1 * sizeof(int));
if (p == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}
fprintf (stdout, "Introduce an integer value: ");
scanf ("%d", p);
...
```

If the allocation fails the program terminates. On the contrary, if the operating system is able to reserve a memory space to store an integer value the program reads that value and stores it in the allocated memory. Figure 3.1 shows a graphical representation of the pointer *p* and its reference



**Figure 3.1** Dynamic allocation of one single integer value.

As previously stated the quantity of memory can be selected at run-time, as shown by the following piece of code.

#### Example 3.4 In the following piece of code

```
int n, *v;

fprintf (stdout, "Introduce n: ");
scanf ("%d", &n);

v = (int *) malloc (n * sizeof (int));
if (v == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}
// use v to refer to (n*sizeof(int)) bytes
...
```

function `malloc` reserves a quantity of memory equal to *n* times the size of an integer value. Note that *n* is known only at run-time and unknown at compilation time.

#### 3.1.2 Function `calloc`

Even if function `malloc` could be sufficient to create all possible C applications, the general utilities library `stdlib.h` provides two other functions for dynamic memory allocation, i.e, `calloc` and `realloc`. These functions can be used to create and modify dynamic arrays.

Function `calloc` (clear alloc) dynamically allocates memory for an array and it initializes this memory. Thus, it corresponds to `malloc` followed by an initialization phase. Initialization is not for free, and it has a linear (in the memory size) cost. Thus, if initialization is not required `malloc` is more efficient. Integer values are initialized with 0, character values with '0' (i.e., `NULL`, ASCII character 48).

The prototype for `calloc` is

```
void *calloc (size_t number_of_objects, size_t size);
```

Its two arguments represent the number of elements (`number_of_objects`) and the size of each element (`size`). Function `calloc` also initializes the elements of the array to zero. The function returns a pointer to the allocated memory, or a `NULL` pointer if the memory is not allocated.

Again, the primary difference between `malloc` and `calloc` is that `calloc` clears the memory it allocates and `malloc` does not.

### Example 3.5 The piece of code

```
int *test;
...
test = (int *) calloc (5, sizeof(int));
if (test==NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}
```

make `test` referring to a usable block containing 5 integer values. It is somehow equivalent to the following one:

```
int *test, i;
...
test = (int *) malloc (5 * sizeof(int));
if (test==NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}
for (i=0, i<5; i++) {
    test[i] = 0;
}
```

Notice that as with `malloc`, it is better to check whether the allocation has been successful or not.

#### 3.1.3 Function `realloc`

Function `realloc` changes the size of an object allocated by a previous call to `malloc`, `calloc` or `realloc`, i.e., it changes the size of a memory block already assigned to a pointer. The original memory block is extended with extra memory space placed at the end of the previous memory block. If this is impossible, function `realloc` reallocate the previous memory space to a different memory area, and it copies all data from the original memory area to the new one. Similarly to `realloc` the cost is potentially linear in the amount of memory copied from the old area to the new one. The original object's contents are not modified provided that the amount of memory allocated is larger than the amount allocated previously. Otherwise, the contents are unchanged up to the size of the new object.

The prototype for `realloc` is

```
void *realloc (void *ptr, size_t size);
```

The two arguments are a pointer to the original object (ptr) and the new size of the object (size). If ptr is NULL, realloc works identically to malloc. If ptr is not NULL and size is greater than zero, realloc tries to allocate a new block of memory for the object. If the new space cannot be allocated, the object pointed to by ptr is unchanged. Function realloc returns either a pointer to the reallocated memory, or a NULL pointer to indicate that the memory was not reallocated.

**Example 3.6** The following piece of code allocates a chunk of memory of 50 bytes, then it enlarges it to 100 bytes and then to 200 bytes.

```
1 int *v1, *v2, *v3;
2
3 v1 = malloc (50 * sizeof (int));
4 if (v1 == NULL) {
5     fprintf (stderr, "Memory allocation error.\n");
6     exit (1);
7 }
8 ...
9 v2 = realloc (v1, 100 * sizeof (int));
10 if (v2 == NULL) {
11     fprintf (stderr, "Memory allocation error.\n");
12     exit (1);
13 }
14 ...
15 v3 = realloc (v2, 200 * sizeof (int));
16 if (v3 == NULL) {
17     fprintf (stderr, "Memory allocation error.\n");
18     exit (1);
19 }
20 ...
```

Notice that the strategy to double the quantity of bytes requested is somehow common to many applications. This because allocation and reallocation are CPU time expensive, and it is better to avoid calling those functions too many times to ask for small size increment.

### 3.1.4 Function free

Function free deallocates memory, i.e., the memory is returned to the system so that it can be reallocated in the future. This function must only be used to release memory assigned with functions malloc, calloc, and realloc. A static array (i.e., int v[100];) cannot be freed.

The prototype for this function is

```
void free (void *pointer);
```

which releases the block of memory referenced by pointer.

The following example shows how to use function free.

**Example 3.7** The following piece of code

- ▷ Reads an integer value n.
- ▷ Allocates n bytes.
- ▷ If the allocation fails ends the entire program.
- ▷ If the allocation succeeds, it goes on, and when it does not require that memory any more it frees it.

```

int n, *p;
fprintf (stdout, "Introduce n: ");
scanf ("%d", &n);

p = (int *) malloc (n * sizeof (int));
if (p == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
} else {
    // use p
    ...
    // when p is no longer needed free it
    free (p);
}

```

As a final comment notice that all dynamically allocated data structure are usually released and returned to the operating system, as soon as the program ends. Anyway, it is good practice to explicitly free dynamically allocated memory as soon as this memory is not required any more. This behavior reduces the possibilities to run out of memory, to insert undesired memory leaks in C programs, and to incur in various types of inefficiencies. As a consequence the number of `free` calls in a program should match the number of `malloc` plus the number of `calloc` as each program should include a `free` operation for each allocation.

Notice, however, that each chunk of memory cannot be used any more in the program after it gets freed. As a consequence if a chunk of memory `p1` includes a pointer to another chunk of memory `p2`, `p2` and the referenced chunk of memory get lost if `p1` is freed before `p2`. This also implies that memory is often freed starting from the last allocated pointer. Anyway, this topic will be clarified in the following sections talking about 2D dynamic arrays.

## 3.2 Dynamically Allocated 1D Arrays

In C, the exact size of array has to be defined at compile time, i.e., the time when a compiler compiles your code into a computer understandable language. As a consequence, sometimes the size of an array can be smaller than necessary creating problems, and sometimes larger than necessary wasting system resources (i.e., memory). As previously analyzed, dynamic memory allocation enables strategies to solve this problem. The following example shows the use of `malloc` to allocate and manipulate 1D arrays.

**Example 3.8** The following piece of code

- ▷ Reads an integer value `n`.
- ▷ Allocates an array of integer values of size `n`.
- ▷ If the allocation fails, the program ends.
- ▷ If the allocation succeeds, the array is read from standard input, it is then printed-out in the inverse order, and then it is freed.

```
int n, *v;
```

```

fprintf (stdout, "Introduce n: ");
scanf ("%d", &n);

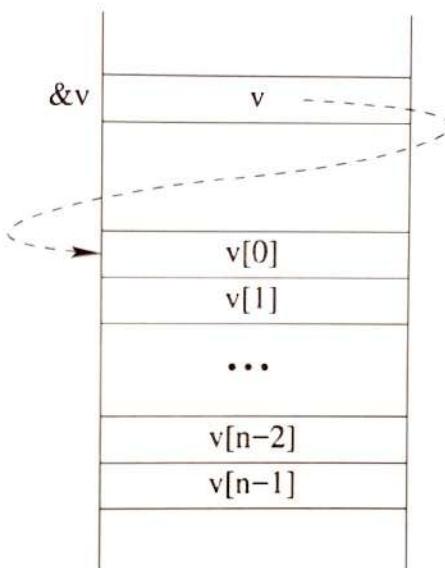
```

```

v = (int *) malloc (n * sizeof (int));
if (v == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
} else {
    for (i=0; i<n; i++) {
        fprintf (stdout, "v[%d]: ", i);
        scanf ("%d", &v[i]);
    }
    for (i=n-1; i>=0; i--) {
        fprintf (stdout, "v[%d]=%d\n", i, v[i]);
    }
    free (v);
}

```

Figure 3.2 shows a graphical representation of the pointer *v* and its referenced memory area.



**Figure 3.2** Dynamic allocation of one array of integers.

Notice that in the previous example, once the array is allocated it is used adopting the standard array notation (with square brackets). Nevertheless, it is also possible to use pointer arithmetic as shown by the following example.

**Example 3.9** The following piece of code performs the same actions performed by the Example 3.8 with two main differences:

- ▷ The `calloc` function is used instead of `malloc`. The explicit cast is not inserted as somehow superfluous.
- ▷ The array is manipulated throughout pointer arithmetic instead of the standard array notation.

```

int n, *v;

fprintf (stdout, "Introduce n: ");
scanf ("%d", &n);

v = calloc (n, sizeof (int));
if (v == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
}

```

```

    exit (1);
} else {
    for (i=0; i<n; i++) {
        fprintf (stdout, "v[%d]: ", i);
        scanf ("%d", v+i);
    }
    for (i=n-1; i>=0; i--) {
        fprintf (stdout, "v[%d]=%d\n", i, *(v+i));
    }
    free (v);
}

```

Notice that the two `for` cycles can be also written as follows:

```

int n, *v, *p;
...
for (i=0, p=v; i<n; i++, p++) {
    fprintf (stdout, "v[%d]: ", i);
    scanf ("%d", p);
}
for (i=0, p--; i>=0; i--, p--) {
    fprintf (stdout, "v[%d]: ", i, *p);
}
...

```

**Example 3.10** The following piece of code allocates an array of integer and reallocates it to a larger size.

```

1 int *v1, *v2;
2
3 v1 = malloc (100 * sizeof (int));
4 if (v1 == NULL) {
5     fprintf (stderr, "Memory allocation error.\n");
6     exit (1);
7 }
8 ...
9 for (i=0; i<100 i++) {
10     v1[i] = i;
11 }
12 ...
13 v2 = realloc (v1, 1000 * sizeof (int));
14 if (v2 == NULL) {
15     fprintf (stderr, "Memory allocation error.\n");
16     free (v1);
17 }
18 // Eventually: exit (1);
19 }
20 ...
21 for (i=0; i<100; i++) {
22     fprintf (stdout, "%d", v2[i]);
23 }
24 ...
25 free (v2);
26 }

```

Notice that:

- ▷ Line 16: It may be necessary to free the original pointer if the reallocation fails.

- ▷ Line 21: It would print values from 0 to 100.

The following is a more complex example with several arrays in use at the same time, and several notations used to manipulate them.

**Example 3.11** The following piece of code includes some examples of declaration and use of 1D dynamic arrays.

```
#define SIZE1 50
#define SIZE2 100

// declare a pointer variable to point to allocated heap space
int *intArray;
double *floatArray;

// define an extra pointer to float
double *dptr;

// call malloc to allocate that appropriate number of bytes for the array
intArray = (int *)malloc(SIZE1 * sizeof(int));
floatArray = (int *)malloc(SIZE2 * sizeof(double));
if (intArray==NULL || floatArray==NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (EXIT_FAILURE);
}

/*
 * use standard ([]) notation to access array buckets
 * this is the preferred (simplest) way to do it
 */
for (i=0; i<SIZE1; i++) {
    intArray[i] = 0;
}

/*
 * use pointer arithmetic (more complex)
 */
*dptr = floatArray;
/*
 * Notice that now the value of floatArray is equivalent
 * to &(floatArray[0])
 */
for (i=0; i<SIZE2; i++) {
    *dptr = 0;
    dptr++;
}
...
free (intArray),
free (floatArray),
```

The following example show once more the difference between static arrays and dynamic one (see also Section 2.7).

**Example 3.12** Assume the following definitions

```
char v[10];
char *p = malloc (10 * sizeof (char))
which value will return the construct sizeof (v) and sizeof (p)?
```

`sizeof (v)` will return the size of the array (in bytes): A set of 10 characters each one of 1 bytes, that is, 10. On the contrary, `sizeof (p)` will return the size of the pointer `p`, i.e., 4 or 8 bytes on modern hardware architectures (at 32 or 64 bits).

### 3.2.1 Dynamic Memory Allocation and Modularity

One of the main issues programmers have to deal with, is how to make dynamically allocated variable exportable objects. Let us analyze the following example.

**Example 3.13** In the following piece of code, function `array_create` is called by the main and it is supposed to allocate an array of `n` integer elements to be used by the main itself.

```
void array_create (int *ptr, int n) {
    ptr = (int *) malloc (n * sizeof (int));
    if (ptr == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
    return;
}

int main (void) {
    int n, *v=NULL;

    scanf ("%d", &n);
    array_create (v, n);
    if (v == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    } else {
        fprintf (stdout, "Memory allocation OK ... continue.\n");
    }
    ...
}
```

Which message is printed-out on standard output by the main program? Notice that function `array_create` allocates an array, and it assigns its reference to `ptr`. Unfortunately `v` is passed by value to `array_create`. As a consequence, even if `ptr` is assigned with a new pointer, `v` does not change and it keeps pointing to `NULL`. Function `array_create` is then completely useless, and the reserved piece of memory is lost after the termination of this function as no one is able to reach it.

To rectify this problem there are at least three possible solutions:

- ▷ It is possible to define all variables, indeed even pointers, as global objects. In this case, it is then possible to define the pointer outside the function which allocates the memory, and make this variable global to the entire program with no visibility problem at all.
- ▷ Any object, i.e, even a pointer, can be returned by the function to the caller using the `return` statement.
- ▷ Any object, even a pointer, can be passed to the function as a parameter. As the pointer has to be returned by the function this implies passing it by *reference*. Anyhow, it has to be noticed that the use of global variables is usually avoided, as they are somehow dangerous and do not fit well into modular programming. For these

reasons, the first technique will never be used in this book (please refer to Chapter 8 for further hints on this issue). The second and third strategies will be the focus of our analysis,

The first one, is the simplest one, and it consists in returning the pointer with the return construct, as shown by the following example.

**Example 3.14** In the following piece of code function `array_create` allocates the array as in the previous example, but it returns the pointer `ptr` to the main, making this pointer visible.

```
int *array_create (int n) {
    int *ptr;

    ptr = (int *) malloc (n * sizeof (int));
    if (ptr == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }

    return ptr;
}

int main (void) {
    int n, *v=NULL;

    scanf ("%d", &n);
    v = array_create (n);
    ...
}
```

The third approach is to pass the pointer by reference to the allocation function. Nevertheless, as the pointer is already a reference, it is necessary to pass the reference of a reference, that is a 2-star object, as shown in the following example.

**Example 3.15** The pointer `v` is passed by reference. As a consequence all modification done by function `array_create` are visible outside the function. Great care has to be taken to deal with proper object types, i.e., the 2-star object. To this respect at least two different approaches are possible. The first one directly uses the 2-star object within the body of the function.

```
void array_create (int **v_ptr, int n) {
    *v_ptr = (int *) malloc (n * sizeof (int));
    if (*v_ptr == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }

    // Manipulate the array using the *v_ptr notation

    return;
}

int main (void) {
    int n, *v=NULL;

    scanf ("%d", &n);
    array_create (&v, n);
    ...
}
```

Notice again that `v_ptr` is a reference of a reference. The reason for this is simple: If to modify a value we need a reference, to modify a reference we need the reference to this reference. A second approach avoids the use of the 2-star object within the body of the function defining a local pointer.

```
void array_create (int **v_ptr, int n) {
    int *ptr;
    ptr = (int *) malloc (n * sizeof (int));
    if (ptr == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
    // Manipulate the array using the ptr notation
    // Then, assign ptr to *v_ptr
    *v_ptr = ptr;
}

int main (void) {
    int n, *v=NULL;

    scanf ("%d", &n);
    array_create (&v, n);
    ...
}
```

In this case, to avoid too many `*` within function `array_create`, the local pointer `ptr` is used to allocate (and eventually manipulate) the array, and it is assigned to `*v_ptr` just before the return statement.

As a final comment notice that this solution is somehow more general than the one presented in the Example 3.14. In that case the use of the `return` statement limits the number of returned pointers to one, whereas there is no limit to the number of pointers which can be passed by reference to a function using this approach.

### 3.2.2 Dynamic String Allocation

Dynamic strings can be allocated as other dynamic arrays. However, it is necessary to remind that a string has a termination character '`\0`' and it is necessary to reserve space for that character too. The following example illustrates this requirements.

**Example 3.16** The following piece of code:

- ▷ Reads a string of 100 characters at most into a static array.
- ▷ Dynamically allocate a second string and it copies the static one into the dynamic one.
- ▷ Print-out the dynamic string.
- ▷ Free all dynamically allocated memory.

```
char str[100+1];
char *v;

scanf ("%s", str);
v = (char *) malloc ((strlen (str) + 1) * sizeof (char));
if (v == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
```

```

    exit (1);
}

strcpy (v, str);
printf ("String = %s\n", v);
free (v);

```

Notice the length of `v` equal to `(strlen (str) + 1)` characters, and not to `(strlen (str))` characters.

The following example shows the use of the `realloc` function with strings.

**Example 3.17** The following piece of code:

```

char *str;

str = (char *) malloc (9 * sizeof (char));
if (str == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}
strcpy (str, "fileName");
...

str = (char *) realloc (str, 13 * sizeof (char));
if (str == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}
strcat (str, ".txt");
...

free (str);

```

uses the `malloc` function to copy the string "fileName" into `str`, and then the `realloc` function to add to this string the extension ".txt". Notice the allocation and reallocation size and remind that `sizeof ("fileName") = 8`, and that `sizeof ("fileName.txt") = 12` (the extra character is reserved for '0').

As a final comment on strings and dynamic allocation, it has to be noticed that the C language makes available an alternative strategy to the one shown in Example 3.16 and 3.17. Function `strdup`, has the following prototypes:

`char *strdup (char *str);`

it receives a string as a parameter, and it:

- ▷ Allocate an array of character whose length is equal to `strlen(str)+1`.
- ▷ If the allocation succeeds, it copies `str` into the newly allocated string.
- ▷ If the allocation fails, it returns a `NULL` value.

Example 3.18 The piece of code of the Example 3.16 can be rewritten as follows.

`char str[100+1];`

`char *v;`

`scanf ("%s", str);`

```

v = strdup (str);
if (v == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}

fprintf (stdout, "String = %s\n", v);
free (v);

```

### 3.2.3 Dynamic Arrays of Structures

Example 3.2 shows the allocation of a dynamic array of C structure. This section elaborate this issue with a greater accuracy.

A first simple case is the one in which the C structure only include scalar types, as shown by the following example.

**Example 3.19** Let us suppose to have the following type definition:

```

struct student {
    int register_number;
    int credit_passed;
    float mark_average;
};

```

It is possible to allocate an array *v* of such a structure of size *n* as follows.

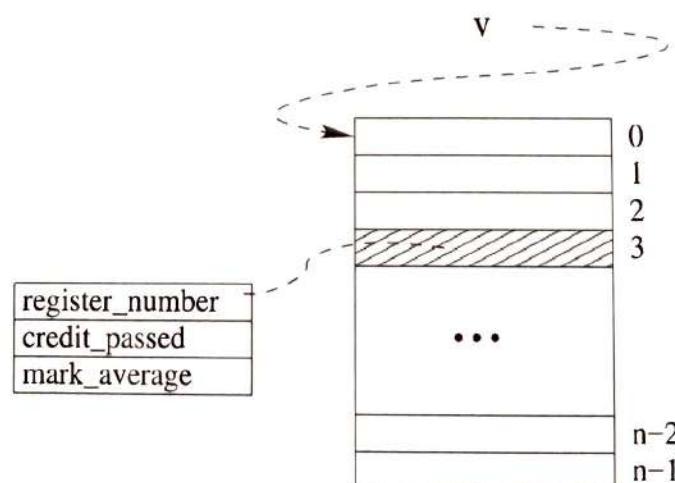
```

int n;
struct student *v;

fprintf (stdout, "Number of students: ");
scanf ("%d", &n);
v = (struct student *) malloc (n * sizeof (struct student));
if (v == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}
...
free (v);

```

Such an array is represented in Figure 3.3.



**Figure 3.3** Dynamically allocated array of structures.

Each element of the array is a struct student, with 3 fields, i.e., register\_number, credit\_passed and mark\_average. For the student in position  $i$  of the array (with  $i \in [0, n-1]$ ) those fields can be accessed as  $v[i].register\_number$ ,  $v[i].credit\_passed$ , and  $v[i].mark\_average$ .

The previous example can be further generalized by considering C structures with static or dynamic arrays as fields. The following is an example allocating arrays of structures with static array fields.

**Example 3.20** Let us consider the following data structure.

```
struct student {
    char last_name[DIM], first_name[DIM];
    int register_number;
    float average;
};
```

where DIM is a predefined constant. It is possible to allocate a dynamic array of such as structure as done in previous cases

```
int n;
struct student *v;

fprintf (stdout, "Number of students: ");
scanf ("%d", &n);
v = (struct student *) malloc (n * sizeof (struct student));
if (v == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}
...
free (v);
```

After the allocation, each element of the array v includes all structure fields exactly like a static array. For example,  $v[2].register\_number$  is the register number of the student stored in position number 3. Moreover, the instruction

```
scanf ("%s %s", v[2].first_name, v[2].last_name);
```

would read it first and last names.

The following example generalizes the previous one defining a data structure with dynamic fields.

**Example 3.21** Let us suppose to modify the data structure analyzed in the Example 3.20 as follows:

```
struct student {
    char *first_name, *last_name;
    int register_number;
    float average;
};
```

The main array v can be allocated as in the previous case. Anyhow, the structure fields first\_name and last\_name are pointers to dynamic arrays. Those need to be allocated for each element of v, as follows:

```
int i, n;
struct student *v;
char first_name[DIM], last_name[DIM];
```

```

fprintf (stdout, "Number of students: ");
scanf ("%d", &n);
v = (struct student *) malloc (n * sizeof (struct student));
if (v == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}
for (i=0; i<n; i++) {
    fprintf (stdout, "First and last names : ");
    scanf ("%s %s", first_name, last_name);
    v[i].first_name = strdup (first_name);
    v[i].last_name = strdup (last_name);
    if (v[i].first_name==NULL || v[i].last_name==NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
    ...
}

```

After this allocation phase all fields can be accessed as they were statically defined. However, all fields have to be deallocated at the end of the process (i.e., when they are no more needed):

```

for (i=0; i<n; i++) {
    free (v[i].first_name);
    free (v[i].last_name);
}
free (v);

```

Notice again, that all fields have to be freed before *v*. Freeing *v* before its an error because all fields would be unreachable, then all pieces of allocated memory lost forever.

It has to be noticed that there is at least one further important difference between the structure analyzed in Example 3.20 and 3.21. Let us suppose to define two variables of the structure defined in Example 3.20:

```
struct student s1, s2;
```

and, then, after a while, when all fields have been properly initialized to assign the first one to the second one as

```
s2 = s1;
```

As in Example 3.20 all fields are statically allocated, the first and last name of *s1* will be copied within the arrays representing the first and last names of *s2*. That is, copying the entire structure would imply a *strcpy* operation, to copy all characters from the source to the destination strings. On the contrary, using the structure defined in Example 3.21, the assignment

```
s2 = s1;
```

will copy pointers without any character or string copy. This is because the structure defined in Example 3.21 includes pointers to dynamic arrays not the array themselves. We will see in Chapter 8 that dynamic memory allocation allocates memory in the heap. Then the arrays do not sit within the memory reserved for *s1* and *s2*, where the pointers are located, but elsewhere in the system memory. In other words, the assignment *s2=s1* will not duplicate the strings, but it will make pointers referring to the same strings, placed in the system heap.

Our last example shows a case in which two structures with dynamic fields are nested.

**Example 3.22** Let us consider the following data structures:

```
struct ingredient_s {
    char name[MAX];
    int quantity;
} ingredient_t;

struct receipt_s {
    int ingredient_number;
    ingredient_t *ingredient;
} receipt_t;
```

where MAX is an integer constant value. The following piece of code creates a dynamic array of receipts, for each receipt it allocates a dynamic array of ingredients, and it stores for each ingredient its name and quantity.

```
int i, j, m, n;
receipt_t *v;

fprintf (stdout, "Number of receipts: ");
scanf ("%d", &n);
v = (receipt_t *) malloc (n * sizeof (receipt_t));
if (v == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}

for (i=0; i<n; i++) {
    fprintf (stdout, "Receipt %d, number of ingredients: ", i);
    scanf ("%d", &m);
    v[i].ingredient_number = m;
    v[i].ingredient = (ingredient_t *) malloc (m * sizeof (ingredient_t));
    if (v[i].ingredient == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
    for (j=0; j<m; j++) {
        printf ("Receipt %d, ingredient %d: ", i, j);
        scanf ("%s %d", v[i].ingredient[j].name, &v[i].ingredient[j].quantity);
    }
}
```

Notice that field name has been defined as a static array, but it could have been defined as a dynamic array as well.

## 3.3 Median Point

### Specifications

A file stores the coordinates of a set of points in the Cartesian coordinate system. The file format is the following one:

```
n
id_1 x_1 y_1 z_1
id_2 x_2 y_2 z_2
id_3 x_3 y_3 z_3
...
id_n x_n y_n z_n
```

where:

- ▷  $n$  is an integer value representing the number of points stored in the file.
- ▷  $id_i$  identifies points number  $i$ , and it is a string of 3 characters.
- ▷  $x_i, y_i$  e  $z_i$  are the coordinates (real values) of point number  $i$ .

Write a program able to:

- ▷ Read a file name from the command line.
- ▷ Store the file content in a proper data structure.
- ▷ Print-out (on standard output) the id and coordinates of the median point, i.e., the point whose sum of all distances from all other points is minimum.

**Example 3.23** Given the following input file:

```
6
p01 4.4 3.5 0.9
xyz 10.3 10.2 10.4
dlp -3.2 3.2 0.5
bh3 -2.4 -2.1 -1.3
abc 0.1 0.1 0.1
klm 1.5 -1.0 0.6
```

the program has to print-out:

```
Median point: abc 0.1 0.1 0.1
```

## Solution

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <math.h>
5
6 #define MAX_ID    3+1
7 #define MAX_NAME 10+1
8
9 /* structure declaration */
10 typedef struct {
11     float x;
12     float y;
13     float z;
14     float dist;
15 } point_t;
16
17 int main (void) {
18     char name[MAX_NAME];
19     int n, i, j, min=0;
20     float dx, dy, dz;
21     point_t *points;
22     FILE *fp;
23
24     /* load the input file */
25     fprintf(stdout, "Input file name: ");
26     scanf("%s", name);
27     fp = fopen(name, "r");
28     if (fp == NULL) {
29         fprintf(stderr, "File open error (file=%s).\n", name);
```

```

30     exit(EXIT_FAILURE);
31 }
32
33 if (fscanf(fp, "%d", &n)==EOF || n<=0) {
34     fprintf(stderr, "Read file error (file=%s).\n", name);
35     return EXIT_FAILURE;
36 }
37 points = (point_t *)malloc(n * sizeof(point_t));
38 if (points == NULL) {
39     fprintf(stderr, "Memory allocation error.\n");
40     exit(EXIT_FAILURE);
41 }
42
43 i = 0;
44 while (i<n && fscanf(fp, "%f%f%f", &dx, &dy, &dz)!=EOF) {
45     points[i].x = dx;
46     points[i].y = dy;
47     points[i].z = dz;
48     points[i].dist = 0;
49     i++;
50 }
51 n = i;
52
53 /* find and print the median point */
54 min = 0;
55 for (i=0; i<n; i++) {
56     for (j=0; j<n; j++) {
57         dx = points[i].x - points[j].x;
58         dy = points[i].y - points[j].y;
59         dz = points[i].z - points[j].z;
60         points[i].dist += sqrt(pow(dx, 2) + pow(dy, 2) + pow(dz, 2));
61     }
62     if (points[i].dist < points[min].dist) {
63         min = i;
64     }
65 }
66 fprintf(stdout, "Median point: %.2f ", points[min].x);
67 fprintf(stdout, "% .2f %.2f\n", points[min].y, points[min].z);
68
69 free(points);
70
71 return EXIT_SUCCESS;
72 }

```

## 3.4 Analytic Index

### Specifications

Write a program to generate the analytic index of a book as follows.

A first file stores the original text book. The number of lines is undefined, but each line has at most 100 characters.

A second file stores all words to insert in the analytic index. The first line of the file reports the number of words, and each subsequent row stores one of those word. Each word has a maximum length of 20 characters.

The program has to find all words reported in the second file into the text stored in the first file. For each word the program has to print-out the number of times the words appear in the text and, only for the first 10 appearances, it has to report its position in the text (its numerical position within the list of words appearing in the

text).

Both file names are passed to the program on the command line, and they can be read only once. The program has to consider small and capital letters as equivalent (i.e., “word” and “WorD” are the same string) but it does not have to cope with punctuation (i.e., “word”, “word!” and “word.” are different strings).

**Example 3.24** Let the first files be the following:

David Allen: You can do anything , but not everything

Unknown Author: The richest man is not he who has the most , but he  
                  who needs the least

Wayne Gretzky: You miss 100 percent of the shots you never take

Abraham Maslow: To the man who only has a hammer , everything he  
                  encounters begins to look like a nail

Aristotle: We are what we repeatedly do ;  
                  excellence , then , is not an act but a habit

and the second one the following:

```
4
anything
you
we
then
```

The program has to print-out the following information:

```
anything - 1 occurrence(s) - word(s): 6
you - 3 occurrence(s) - word(s): 3 32 39
we - 2 occurrence(s) - word(s): 63 66
then - 1 occurrence(s) - word(s): 72
```

## Solution

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <ctype.h>
5
6 #define MAX_WORD_LENGTH 20
7 #define MAX_LINE_LENGTH 100
8 #define MAX_INDEX 10
9
10 /* structure declaration */
11 typedef struct {
12     char word[MAX_WORD_LENGTH+1];
13     int occurrences;
14     int positions[MAX_INDEX];
15 } index_t;
16
17 /* function prototypes */
18 index_t *word_read(char *name, int *num_ptr);
19 void text_read(char *name, index_t *index, int n);
20 void index_display(index_t *index, int n);
21 int compare (char *src, char *dst);
22
23 /*
```

```
24 * main program
25 */
26 int main (int argc, char *argv[]) {
27     index_t *index;
28     int n;
29
30     if (argc < 3) {
31         fprintf(stderr, "Error: missing parameter.\n");
32         fprintf(stderr, "Run as: %s <text_file> <word_file>\n", argv[0]);
33         return EXIT_FAILURE;
34     }
35
36     index = word_read(argv[2], &n);
37     text_read(argv[1], index, n);
38     index_display(index, n);
39
40     free(index);
41     return EXIT_SUCCESS;
42 }
43
44 /*
45 *   read the words file; return the index array
46 */
47 index_t *word_read (char *name, int *num_ptr) {
48     index_t *index;
49     FILE *fp;
50     int i, n;
51
52     fp = fopen(name, "r");
53     if (fp == NULL) {
54         fprintf(stderr, "File open error (file=%s).\n", name);
55         exit(EXIT_FAILURE);
56     }
57
58     if (fscanf(fp, "%d", &n) == EOF) {
59         fprintf(stderr, "Read file error (file=%s).\n", name);
60         exit(EXIT_FAILURE);
61     }
62     index = (index_t *)malloc(n * sizeof(index_t));
63     if (index == NULL) {
64         fprintf(stderr, "Memory allocation error.\n");
65         exit(EXIT_FAILURE);
66     }
67
68     i = 0;
69     while (i<n && fscanf(fp, "%s", index[i].word) !=EOF) {
70         index[i].occurrences = 0;
71         i++;
72     }
73     *num_ptr = i;
74
75     fclose(fp);
76     return index;
77 }
78
79 /*
80 *   read the text file; complete the index infos
81 */
82 void text_read (char *name, index_t *index, int n) {
```

```
83  char word[MAX_LINE_LENGTH+1];
84  int i, j, pos;
85  FILE *fp;
86
87  fp = fopen(name, "r");
88  if (fp == NULL) {
89      fprintf(stderr, "File open error (file=%s).\\n", name);
90      exit(EXIT_FAILURE);
91  }
92
93  i = 1;
94  while (fscanf(fp, "%s", word) != EOF) {
95      /* look for the word in the index */
96      for (j=0; j<n; j++) {
97          if (compare(word, index[j].word) == 1) {
98              /* found an occurrence for a word */
99              pos = index[j].occurrences++;
100             if (pos < MAX_INDEX) {
101                 /* store the word position */
102                 index[j].positions[pos] = i;
103             }
104         }
105     }
106     i++;
107 }
108
109 fclose(fp);
110 }
111
112 /*
113  * output the index contents
114 */
115 void index_display (index_t *index, int n) {
116     int i, j;
117
118     for (i=0; i<n; i++) {
119         fprintf(stdout, "%s - %d occurrence(s) ", index[i].word, index[i].occurrences);
120         if (index[i].occurrences > 0) {
121             fprintf(stdout, "- word(s): ");
122             for (j=0; j<index[i].occurrences; j++) {
123                 if (j < MAX_INDEX) {
124                     fprintf(stdout, "%d ", index[i].positions[j]);
125                 }
126             }
127         }
128         fprintf(stdout, "\\n");
129     }
130
131     return;
132 }
133
134 /*
135  * case insensitive comparison between two strings
136  * return 1 if the strings are equal, 0 otherwise
137 */
138 int compare (char *str1, char *str2) {
139     int i;
140
141     if (strlen(str1) != strlen(str2)) {
```

```

142     return 0;
143 }
144
145 for (i=0; i<strlen(str1); i++) {
146     if (tolower(str1[i]) != tolower(str2[i])) {
147         return 0;
148     }
149 }
150
151 return 1;
152 }
```

## 3.5 Blood Donations

### Specifications

Write an application to deal with the Transylvania blood bank, whose data base, for confidentiality reasons, is divided into two separate files as follows.

A “donation” file stores all blood donations with the following format:

id quantity

where id identifies the donors (unique code of 5 characters), and quantity indicates the volume of given blood (integer value). The number of lines in the file and their order are unknown, and the same id, i.e., donor, can appear more than once in the file.

A “voluntary” file stores the real identity of all people enrolled within the transfusion system:

id title surname name

where title, surname and name are strings with at most 20 characters. The length of the file is unknown.

The application has to compute and print statistics for each voluntary, computing the number of donations and the total amount of blood transfused. See the example for further details.

**Example 3.25** Given the two following files for the donations and the voluntary list:

File “donation”

JH-YY 300
VH-ZZ 400
JH-YY 200
BS-00 500
JH-YY 600
VH-ZZ 400

File “voluntary”

BS-00 Writer Bram Stoker
JH-YY Agent Jonathan Harker
VT-XX Cont Vlad Tepes
VH-ZZ Doctor Van Helsing

the application has to print-out the following information:

BS-00 Bram Stoker: 1 donation/s - 500 cc
JH-YY Jonathan Harker: 3 donation/s - 1100 cc
VT-XX Vlad Tepes: 0 donation/s - 0 cc
VH-ZZ Van Helsing: 2 donation/s - 800 cc

## Solution

As the list of voluntaries must be stored whereas we just need a summary for all donations for each individual, the “voluntary” file must be read before the “donation” one. As the number of individuals is unknown, we need dynamic memory allocation to define an array of structures including one element for each individual. Moreover, as the number of individuals is not defined on top of the file, we have two possible strategies to dynamically allocate the array. The first one, would consist in allocating the array of arbitrary size and to reallocate it when we exceed its size. As each reallocation potentially implies a linear cost, a linear number of reallocation (one for each new file line) would imply a quadratic cost. To avoid this cost, we can use the second strategy. We can read the file twice: The first time to count-up the number of lines to allocate an array, the second one to actually store the file content into the data structure.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 #define MAX_ID      5+1
6 #define MAX_NAME   20+1
7
8 /* structure declaration */
9 typedef struct {
10     char id[MAX_ID];
11     char surname[MAX_NAME];
12     char name[MAX_NAME];
13     int total_amount;
14     int num_donations;
15 } donor_t;
16
17 /* function prototypes */
18 donor_t *donor_read (int *num_ptr);
19 void donation_read (donor_t *donors, int n);
20 int id_search (donor_t *donors, char id[MAX_ID+1], int n);
21 void result_display (donor_t *donors, int n);
22
23 /*
24  * main program
25  */
26 int main (void) {
27     donor_t *donors;
28     int n;
29
30     donors = donor_read(&n);
31     donation_read(donors, n);
32     result_display(donors, n);
33
34     free(donors);
35     return EXIT_SUCCESS;
36 }
37
38 /*
39  * load the donor list into a dynamic array
40  */
41 donor_t *donor_read (int *num_ptr) {
42     char id[MAX_ID], name[MAX_NAME], surname[MAX_NAME];
43     donor_t *donors;
44     int i, n;

```

```
45     FILE *fp;
46
47     fprintf(stdout, "Donors file name: ");
48     scanf("%s", name);
49
50     /* compute the number of rows */
51     fp = fopen(name, "r");
52     if (fp == NULL) {
53         fprintf(stderr, "File open error (file=%s).\n", name);
54         exit(EXIT_FAILURE);
55     }
56     n = 0;
57     while (fscanf(fp, "%s %*s %s %s", id, surname, name) != EOF) {
58         n++;
59     }
60     fclose (fp);
61
62     /* allocate the dynamic array */
63     donors = (donor_t *)malloc(n * sizeof(donor_t));
64     if (donors == NULL) {
65         fprintf(stderr, "Memory allocation error.\n");
66         exit(EXIT_FAILURE);
67     }
68
69     /* parse the file contents */
70     fp = fopen(name, "r");
71     if (fp == NULL) {
72         fprintf(stderr, "File open error.\n");
73         exit(EXIT_FAILURE);
74     }
75     for (i=0; i<n; i++) {
76         fscanf(fp, "%s %*s %s %s", id, surname, name);
77         strcpy(donors[i].id, id);
78         strcpy(donors[i].name, name);
79         strcpy(donors[i].surname, surname);
80         donors[i].total_amount = 0;
81         donors[i].num_donations = 0;
82     }
83     fclose(fp);
84
85     *num_ptr = n;
86
87     return donors;
88 }
89
90 /*
91 * parse the donation file; update the data structure
92 */
93 void donation_read (donor_t *donors, int n) {
94     char name[MAX_NAME], id[MAX_ID];
95     int i, amount;
96     FILE *fp;
97
98     fprintf(stdout, "Donations file name: ");
99     scanf("%s", name);
100    fp = fopen(name, "r");
101    if (fp == NULL) {
102        fprintf(stderr, "File open error.\n");
103        exit(EXIT_FAILURE);
```

```

104      }
105
106      while (fscanf(fp, "%s %d", id, &amount) != EOF) {
107          /* update the "donation" amount */
108          i = id_search(donors, id, n);
109          if (i >= 0) {
110              donors[i].total_amount += amount;
111              donors[i].num_donations++;
112          }
113      }
114
115      fclose (fp);
116
117      return;
118  }
119
120  /*
121   *   search an "id" into the donor array; return its index
122   */
123  int id_search (donor_t *donors, char id[MAX_ID+1], int n) {
124      int i;
125
126      for (i=0; i<n; i++) {
127          if (strcmp(donors[i].id, id) == 0) {
128              return i;
129          }
130      }
131
132      return -1;
133  }
134
135  /*
136   *   output the required results
137   */
138  void result_display (donor_t *donors, int n) {
139      int i;
140
141      for (i=0; i<n; i++) {
142          fprintf(stdout, "%s %s ", donors[i].id, donors[i].name);
143          fprintf(stdout, "%s: %d", donors[i].surname, donors[i].num_donations);
144          fprintf(stdout, " donation(s) - %d cc\n", donors[i].total_amount);
145      }
146
147      return;
148  }

```

## 3.6 String Evolution

### Specifications

A colony of biological organism are represented by strings of at most 19 small characters (e.g.,aaaaaaaaaaaaaaaaaa, ababababababababa, etc.). The colony evolves when an organism mate with the most “similar” organism. The “similarity” between two organisms is expressed by an integer number counting the number of times the two strings have the same character in the same position within the string. Organisms with no common characters cannot mate.

For example, organisms abbbbbbbbbb and baaaaaaaaaaaaaa have a similarity equal to 0, whereas aaaaaaaaaaaaaaaa and aaaaaaaaaa-bcdefghij have a similarity equal to 10.

Each time two organisms mate they die and a new organism is born. The newly born organism has again 19 characters, such that the character in position  $i$  is the character alphabetically smaller between the ones in position  $i$  of the two parents.

For example, when organisms aaaaazzzzzzzzzzz and abcdeeeeeeeeeeeez mate, they generate the organism aaaaeeeeeeeeeeezez.

Write a program which is able to read the initial configuration of the colony from an input file, to make it evolve, and to store the final colony on an output file. The input file stores all organisms on subsequent rows. The first row indicates the number of organism in the colony. The output file must have the same format but it has to store all colony generations from the initial one to the one from which the colony cannot evolve any more. Both file names have to be read from the command line.

**Example 3.26** Given the following input file:

```
5
aaaaaaaaaaaaaaaaaa
xxxzzzzzzzzzzzzz
ababababababababa
xyxyxyxyxyxyxyxyx
azzzzzzzzzzzzzzzz
```

the following organisms have similarity larger than 0: 1 and 3 similarity 10; 1 and 5 similarity 1; 2 and 4 similarity 2; 2 and 5 similarity 16. As a consequence, organisms 2 and 5 and 1 ad 3 mate, giving the following first generation:

```
aaaaaaaaaaaaaaaaaa
axxzzzzzzzzzzzzz
xyxyxyxyxyxyxyxyx
```

At this point, organism 1 and 2 have similarity 1 and 2 and 3 have similarity 1. Mating is then randomly selected, and supposing 1 and 2 mate, we obtain the following (second) generation:

```
aaaaaaaaaaaaaaaaaa
xyxyxyxyxyxyxyxyx
```

This community cannot evolve any more, than the output file will be:

```
Generation 1:
aaaaaaaaaaaaaaaaaa
axxzzzzzzzzzzzzz
xyxyxyxyxyxyxyxyx
Generation 2:
aaaaaaaaaaaaaaaaaa
xyxyxyxyxyxyxyxyx
```

## Solution

The following implementation adopts a bottom-up style, i.e., there are no prototypes, and functions are inserted before the main program. The order in which functions are inserted is such that each function definition appears before any of its calls.

<sup>1</sup> `#include <stdio.h>`  
<sup>2</sup> `#include <stdlib.h>`

```
3  
4 #define DEAD      0  
5 #define ALIVE     1  
6 #define CHILD    2  
7  
8 #define DIM      19  
9 #define MAX      50  
10  
11 /* structure declaration */  
12 typedef struct {  
13     char DNA[DIM+1];  
14     int state;  
15 } organism_t;  
16  
17 /*  
18 * parse the input file, return the organism population array  
19 */  
20 organism_t *input_read (int *n, char *name) {  
21     organism_t *population;  
22     FILE* fp;  
23     int i;  
24  
25     fp = fopen(name, "r");  
26     if (fp == NULL) {  
27         fprintf(stderr, "File open error.\n");  
28         exit(EXIT_FAILURE);  
29     }  
30  
31     fscanf(fp, "%d", n);  
32     population = (organism_t *)malloc((*n) * sizeof(organism_t));  
33     if (population == NULL) {  
34         fprintf(stderr, "Memory allocation error.\n");  
35         exit(EXIT_FAILURE);  
36     }  
37  
38     for (i=0; i<*n; i++) {  
39         fscanf(fp, "%s", population[i].DNA);  
40         population[i].state = ALIVE;  
41     }  
42     fclose(fp);  
43  
44     return population;  
45 }  
46  
47 /*  
48 * generate a new organism from two parents  
49 */  
50 void merge (organism_t *population, int i, int j) {  
51     int k;  
52  
53     for (k=0; k<DIM; k++) {  
54         if (population[i].DNA[k] > population[j].DNA[k]) {  
55             population[i].DNA[k] = population[j].DNA[k];  
56         }  
57     }  
58     population[i].state = CHILD;  
59     population[j].state = DEAD;  
60 }  
61
```

```

62  /*
63   * compare two organisms "char by char"
64  */
65  int likeness (organism_t *population, int i, int j) {
66      int k, count=0;
67
68      for (k=0; k<DIM; k++) {
69          if (population[i].DNA[k] == population[j].DNA[k]) {
70              count++;
71          }
72      }
73
74      return count;
75  }
76
77  /*
78   * find the two organisms with maximum likeness
79  */
80  int likeness_max (organism_t *population, int n, int *p1, int *p2) {
81      int i, j, like, max=0;
82
83      for (i=0; i<n-1; i++) {
84          for (j=i+1; j<n; j++) {
85              if (population[i].state==ALIVE && population[j].state==ALIVE) {
86                  like = likeness(population, i, j);
87                  if (like > max) {
88                      *p1 = i;
89                      *p2 = j;
90                      max = like;
91                  }
92              }
93          }
94      }
95      return max;
96  }
97
98  /*
99   * evolve the current organism generation;
100  * return 1 if no new organism can be generated
101  */
102 int evolve (organism_t *population, int n) {
103     int i, j, endFlag=1;
104
105     while (likeness_max (population, n, &i, &j) != 0) {
106         merge(population, i, j);
107         endFlag = 0;
108     }
109
110     return endFlag;
111 }
112
113 /*
114  * main program
115 */
116 int main (int argc, char *argv[]) {
117     organism_t *population;
118     int i, n, generation=1;
119     FILE* fp;
120

```

```

121     if (argc < 3) {
122         fprintf(stderr, "Error: missing parameter.\n");
123         fprintf(stderr, "Run as: %s <organism_file> <evolution_file>\n", argv[0]);
124         return EXIT_FAILURE;
125     }
126
127     population = input_read (&n, argv[1]);
128
129     fp = fopen(argv[2], "w");
130     if (fp == NULL) {
131         fprintf(stderr, "File open error.\n");
132         return EXIT_FAILURE;
133     }
134
135     while (evolve(population, n) != 1) {
136         /* output a generation */
137         fprintf(fp, "Generation %d:\n", generation++);
138         for (i=0; i<n; i++) {
139             if (population[i].state != DEAD) {
140                 fprintf(fp, "%s\n", population[i].DNA);
141                 population[i].state = ALIVE;
142             }
143         }
144     }
145     fclose(fp);
146
147     free(population);
148     return EXIT_SUCCESS;
149 }
```

## 3.7 Broken lines

### Specifications

A file stores information about a set of segments in the Cartesian plane. Each row of the file stores 4 integer values, space separated:

$x_1 \quad y_1 \quad x_2 \quad y_2$

which define the two end-points of the segment  $(x_1, y_1)$  and  $(x_2, y_2)$ . The number of segments reported in the file is stored in the first line of the file.

Write a program to check whether all segments form a single broken line (set of segments with one end-point in common). In this condition is true, the program must print (on standard output) the total length of the broken line. In this condition is false, the program must state which segment does not belong to the broken line.

Notice that each segment can be connected to another one using both of its endpoints, i.e., as segment  $(x_1, y_1) - (x_2, y_2)$  as well as segment  $(x_2, y_2) - (x_1, y_1)$ . All duplicated segments and all segments with length equal to 0 have to be ruled-out. The file name has to be read on the command line.

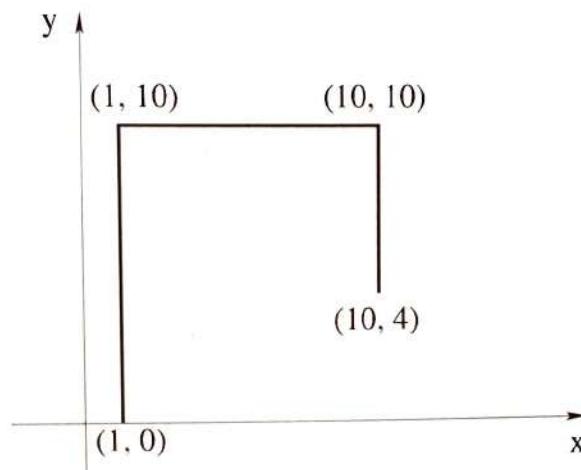
**Example 3.27** Let us suppose that the input file is the following:

```

5 15 15 5
10 10 10 4
25 20 13 12
10 10 1 10

```

Three of the above segments belong to the broken line of Figure 3.4.



**Figure 3.4** Broken line on the Cartesian plane.

The program must print something like:

```

segment 1 0 1 10 - left: disconnected ; right: OK
segment 13 12 25 20 - left: OK ; right: OK
segment 5 15 15 5 - left: disconnected ; right: disconnected
segment 10 10 10 4 - left: OK ; right: disconnected
segment 25 20 13 12 - left: OK ; right: OK
segment 10 10 1 10 - left: OK ; right: OK
Broken line with disconnected segments

```

## Solution

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <math.h>
5
6 /* structure declaration */
7 typedef struct {
8     int x1, y1, x2, y2;
9     int used1, used2;
10 } segment_t;
11
12 /* function prototypes */
13 segment_t *load(char *, int *);
14 void connect(segment_t *, int);
15 void check(segment_t *, int);
16
17 /*
18  * main program
19 */
20 int main (int argc, char *argv[]) {

```

```
21     segment_t *segments;
22     int n;
23
24     segments = load(argv[1], &n);
25     connect(segments, n);
26     check(segments, n);
27
28     free(segments);
29
30     return EXIT_SUCCESS;
31 }
32
33 /*
34  *  load the input file contents
35  */
36 segment_t *load(char *name, int *n) {
37     int i, x1, y1, x2, y2;
38     segment_t *segments;
39     FILE* fp;
40
41     fp = fopen(name, "r");
42     if (fp == NULL) {
43         fprintf(stderr, "File open error.\n");
44         exit(EXIT_FAILURE);
45     }
46
47     fscanf(fp, "%d", n);
48     segments = (segment_t *)malloc(*n * sizeof(segment_t));
49     if (segments == NULL) {
50         fprintf(stderr, "Memory allocation error.\n");
51         exit(EXIT_FAILURE);
52     }
53
54     i = 0;
55     while (i<*n && fscanf(fp, "%d %d %d %d", &x1, &y1, &x2, &y2) !=EOF) {
56         segments[i].x1 = x1;
57         segments[i].y1 = y1;
58         segments[i].x2 = x2;
59         segments[i].y2 = y2;
60         segments[i].used1 = 0;
61         segments[i].used2 = 0;
62         i++;
63     }
64
65     *n = i;
66
67     fclose(fp);
68
69     return segments;
70 }
71
72 /*
73  *  search for connected segments
74  */
75 void connect(segment_t *segments, int n) {
76     int i, j;
77
78     /* for each segment ... */
79     for (i=0; i<n-1; i++) {
```

```
80  /* ... check if one of the next can be connected ... */
81  for (j=i+1; j<n; j++) {
82      if (segments[i].x1==segments[j].x1 && segments[i].y1==segments[j].y1) {
83          segments[i].used1++;
84          segments[j].used1++;
85      }
86      if (segments[i].x1==segments[j].x2 && segments[i].y1==segments[j].y2) {
87          segments[i].used1++;
88          segments[j].used2++;
89      }
90      if (segments[i].x2==segments[j].x1 && segments[i].y2==segments[j].y1) {
91          segments[i].used2++;
92          segments[j].used1++;
93      }
94      if (segments[i].x2==segments[j].x2 && segments[i].y2==segments[j].y2) {
95          segments[i].used2++;
96          segments[j].used2++;
97      }
98  }
99 }
100
101 return;
102 }
103
104 /*
105 * check connection status of all segments
106 */
107 void check(segment_t *segments, int n) {
108     int i, disconnected, overconnected;
109
110     disconnected = 0;
111     overconnected = 0;
112
113     for (i=0; i<n; i++) {
114         fprintf(stdout, "segment %2d %2d %2d %2d - ",
115             segments[i].x1, segments[i].y1, segments[i].x2, segments[i].y2);
116         fprintf(stdout, "left extreme: ");
117         if (segments[i].used1 == 0) {
118             fprintf(stdout, "disconnected ");
119             disconnected++;
120         } else if (segments[i].used1 == 1) {
121             fprintf(stdout, "OK           ");
122         } else {
123             fprintf(stdout, "overconnected");
124             overconnected++;
125         }
126         fprintf(stdout, "; right extreme: ");
127         if (segments[i].used2 == 0) {
128             fprintf(stdout, "disconnected\n");
129             disconnected++;
130         } else if (segments[i].used2 == 1) {
131             fprintf(stdout, "OK\n");
132         } else {
133             fprintf(stdout, "overconnected\n");
134             overconnected++;
135         }
136     }
137
138     if (disconnected > 2) {
```

```

139     fprintf(stdout, "Broken line with disconnected segments\n");
140 }
141 if (overconnected > 0) {
142     fprintf(stdout, "Broken line with over-connected segments\n");
143 }
144 if (disconnected<=2 && overconnected==0) {
145     fprintf(stdout, "Single broken line\n");
146 }
147
148 return;
149 }
```

## 3.8 Dynamically Allocated 2D Arrays

Dynamically declared 2D arrays can be allocated in one of two ways. Let us suppose we need a matrix of  $R$  rows and  $C$  columns. We can:

- ▷ Allocate a single chunk of  $(R \cdot C)$  contiguous elements. This means that the 2D array is seen as a linear 1D structure (a simple 1D array) through “on-the-fly” 2D-to-1D and 1D-to-2D mappings.
- ▷ Allocate an array of arrays. In this case, the standard scheme is to proceed in two steps. Firstly, we allocate one array of  $R$  pointers, each one referencing one array of basic elements. Then, we allocate  $R$  arrays of size  $C$  of basic elements, i.e., one for each row, and we make sure that the previous pointers reference them.

Depending on the selected method, the declaration and the access strategies differ. Notice that, following the second scheme, it is also possible to allocate each column independently and have an array of column arrays. This is somehow less common. We will analyze the previous two strategies in the following subsections.

### 3.8.1 2D Arrays Allocated as 1D Arrays

Following the first strategy a 2D matrix of  $R$  rows and  $C$  columns can be mapped on a 1D array of size  $(R \cdot C)$ . Actually, this is automatically done by a C compiler every time a 2D matrix is defined, as 2D matrices, as all other data structures, are stored in the computer memory, which of course, is just a linear sequence of cells (that is, it is just a 1D data structure).

To perform this linearization two schemes are possible.

In the *row-major-order* scheme rows are allocated one after the other, with their elements in contiguous cells.

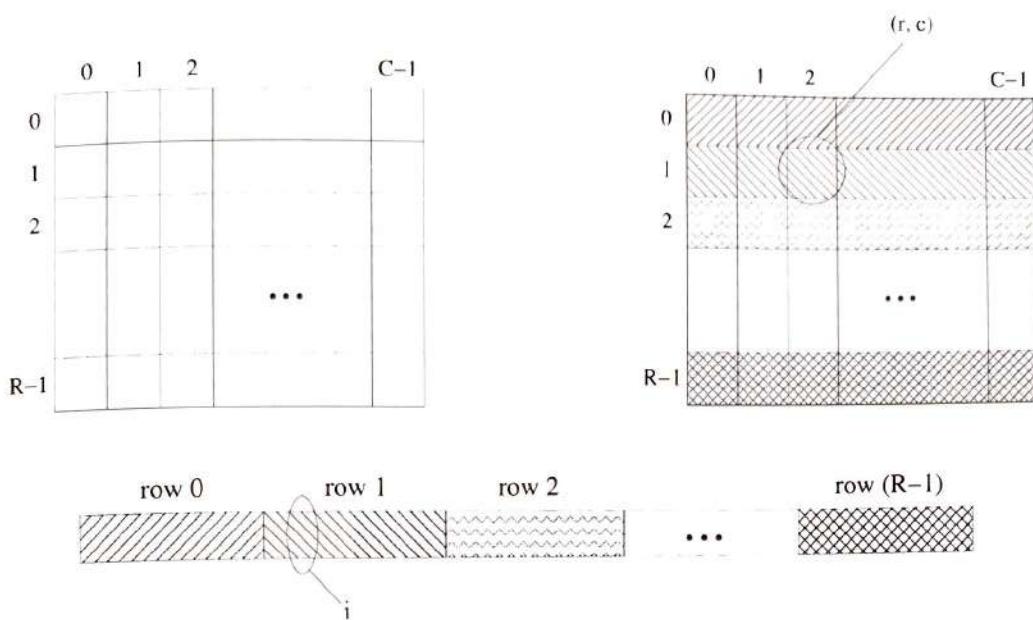
In the *column-major-order* scheme columns are allocated one after the other, with their elements in contiguous cells.

Both schemes arrange multi-dimensional arrays in linear storage such as memory. The difference between the two is simply that the order of the dimensions is reversed. In row-major order the rightmost indices vary faster as one steps through consecutive memory locations, while in column-major order the leftmost indices vary faster. Row-major order is used in Pascal, C, C++, Python, and others. Column-major order is used in FORTRAN, OpenGL, Open CL ES, MATLAB, ad others.

Figure 3.5 illustrates the row-major-order scheme.

Once this scheme is selected, it is possible to implement it using an “on-the-fly” 2D-to-1D conversion and vice-versa.

To map a matrix element  $(r, c)$  onto the corresponding array element  $i$ , it is possible



**Figure 3.5** Matrix linearization: From 2D to 1D storing rows one after the other starting with row 0 and ending with row  $R - 1$ .

to use the following equations:

$$i = r \cdot C + c$$

The opposite mapping, from an element in position  $i$  within the array to the corresponding  $(r, c)$  element within the matrix is allowed by the equation system:

$$\begin{cases} r = i / C \\ c = i \% C \end{cases}$$

Notice that these concepts can be trivially generalized to arrays with more than two dimensions, but we consider this issue outside the scope of the present text.

### 3.8.2 2D Arrays Allocated as 2D Arrays

The following code lines create an array of integer pointers (of size  $r$ ) pointing to array of integers (of size  $c$ ).

```

int r, c, i;
int **mat;

printf ("Number of rows: ");
scanf ("%d", &r);

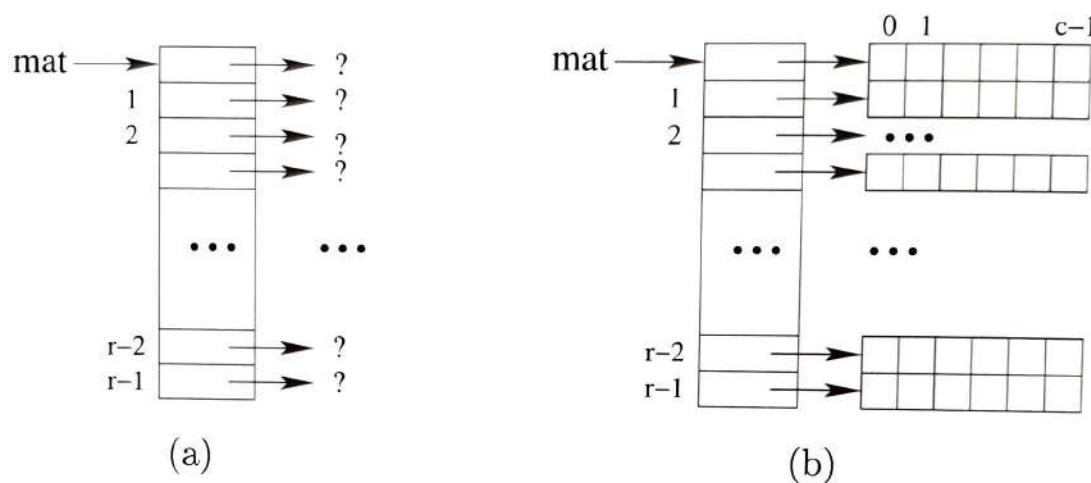
mat = (int **) malloc (r * sizeof (int *));
if (mat == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}

printf ("Number of columns: ");
scanf ("%d", &c);

```

```
for (i=0; i<r; i++) {
    mat[i] = (int *) malloc (c * sizeof (int));
    if (mat[i] == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
}
```

Figure 3.6(a) shows the data structure created after the first allocation, the one assigning to `mat` the array of pointers. This allocation targets `int *` objects, and returns an `int **` pointer. Figure 3.6(b) shows the data structure after all rows have been allocated. Those allocations target `int` objects, and return `int *` pointers.



**Figure 3.6** Dynamic allocation of a 2D array.

The easiest way to reach each matrix element is to use the standard matrix notation, i.e., `mat[i][j]`. Using this notation `mat[i]` indicates an entire row, and `mat` the pointer to the array of pointers.

To free the data structure, we have to use the following code:

```
for (i=0; i<r; i++) {  
    free (mat[i]);  
}  
free (mat);
```

freeing the rows first, and array of pointers after all rows.

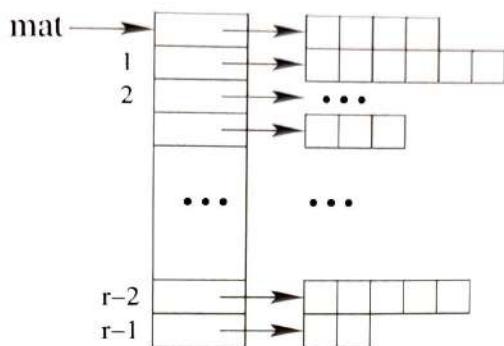
Notice that in the previous case all rows had size equal to  $c$ , but it is possible to create a matrix with rows of different sizes. Figure 3.7 shows an example.

Moreover, the previous examples can be extended to whatever data type we may want to use, such as `char`, `float`, or composite C structures.

To further clarify the relationship between pointers and arrays let's analyze the following example.

**Example 3.28** Let us assume the following piece of code:

```
#define R 2  
#define C 3  
...  
char str[R]  
int **mat;  
  
mat = (char
```



**Figure 3.7** Dynamic allocation of a 2D array with variable size rows.

```

if (mat == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}

for (i=0; i<R; i++) {
    mat[i] = (char *) malloc ((strlen(str)+1) * sizeof (char));
    if (mat[i] == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
    strcpy (mat[i], str[i]);
}

fprintf (stdout, "- &mat      %lx\n", (long unsigned int) (&mat));
fprintf (stdout, "- mat       %lx\n", (long unsigned int) (mat));
fprintf (stdout, "- *mat     %lx\n", (long unsigned int) (*mat));
for (i=0; i<R; i++) {
    fprintf (stdout, "+ &(mat[%d]) %lx\n", i, (long unsigned int) (&(mat[i])));
    fprintf (stdout, "+ mat+%d   %lx\n", i, (long unsigned int) (mat+i));
}
for (i=0; i<R; i++) {
    fprintf (stdout, "> mat[%d]   %lx\n", i, (long unsigned int) (mat[i]));
    fprintf (stdout, "> * (mat+%d) %lx\n", i, (long unsigned int) *(mat+i));
}

```

and let us suppose we run it (notice no matrix value is initialized). One run, the code will generate an output similar to the following one:

```

- &mat      7fff4cb739b0
- mat       bce010
- *mat     bce030
+ &(mat[0]) bce010
+ mat+0    bce010
+ &(mat[1]) bce018
+ mat+1    bce018
> mat[0]   bce030
> * (mat+0) bce030
> mat[1]   bce050
> * (mat+1) bce050

```

Please, compare the values of the printed addresses to better understand the correspondence between different notations.

### 3.8.3 Dynamic Memory Allocation and Modularity

As analyzed in Section 3.2.1 for 1D arrays, “passing” 2D arrays outside the function which has performed the memory allocation requires some care. The available strategies are the same analyzed for 1D arrays.

The following example shows how to return a 2D matrix to the caller using the return statement.

**Example 3.29** The following piece of code allocated a 2D matrix of  $R$  rows, and  $C$  columns of character values, and it returns it to the caller. The code show the prototype, the function definition, and the function call.

```
/* function prototype */
char **malloc2d (int, int);
...

int main (void) {
    char **mat;
    ...
    /* function call */
    mat = malloc2d (R, C);
    ...
}

/* function definition */
char **malloc2d(int r, int c) {
    int i;
    char **mat;

    mat = (char **) malloc (r * sizeof(char *));
    if (mat == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
    for (i=0; i<r; i++) {
        mat[i] = (char *) malloc(c * sizeof (char));
        if (mat[i]==NULL) {
            fprintf (stderr, "Memory allocation error.\n");
            exit (1);
        }
    }
    return (mat);
}
```

The following two examples show how to pass a 2D matrix to a function. As a 2D matrix is already a 2-star object, passing it by reference makes it a 3-star object.

**Example 3.30** As in Example 3.29, the following piece of code allocated a 2D matrix of  $R$  rows, and  $C$  columns of character values. In this case, we use a local variable `mat` to avoid dealing with `m` within the function. This because `m` is a 3-star object and the C syntax is a little awkward.

```
/* function prototype */
void malloc2d (char ***, int, int);
...

int main (void) {
    char **mat;
```

```

...
/* function call */
malloc2d (&mat, R, C);
...
}

/* function definition */
void malloc2d(char ***m, int r, int c) {
    int i;
    char **mat;

    mat = (char **) malloc (r * sizeof(char *));
    if (mat == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
    for (i=0; i<r; i++) {
        mat[i] = (char *) malloc(c * sizeof (char));
        if (mat[i]==NULL) {
            fprintf (stderr, "Memory allocation error.\n");
            exit (1);
        }
    }
    *m = mat;
    return;
}

```

**Example 3.31** In this example, we proceed as in the previous one (Example 3.30) but we do not make use of the local variable mat. Notice the use of the parenthesis to bypass the precedence of the operators ([] has a higher precedence than \*).

```

/* function prototype */
void malloc2d (char ***, int, int);
...

int main (void) {
    char ***mat;
    ...
    /* function call */
    malloc2d (&mat, R, C);
    ...

    /* function definition */
    void malloc2d(char ***m, int r, int c) {
        int i;
        (*m) = (char **) malloc (r * sizeof(char *));
        if (m == NULL) {
            fprintf (stderr, "Memory allocation error.\n");
            exit (1);
        }
        for (i=0; i<r; i++) {
            (*m)[i] = (char *) malloc(c * sizeof (char));
            if ((*m)[i]==NULL) {
                fprintf (stderr, "Memory allocation error.\n");
                exit (1);
            }
        }
    }
}

```

```

    }
}

return;
}

```

### 3.8.4 2D Arrays Summary

In this section we will try to revise all main strategy that can be used to define a 2D matrix.

#### **2D Arrays of Scalar Types**

Using scalar types, a 2D matrix can be defined using the following schemes:

1. Number of rows predefined and number of columns predefined as well.

```
#define R 10
#define C 5
```

```
int mat[R][C];
```

```
...
```

data structure represented in Figure 3.8(a).

2. Number of rows predefined but variable number of columns (Figure 3.8(b)).

```
#define R 10
```

```
int i, c;
int *mat[R];
```

```
for (i=0; i<R; i++) {
    scanf ("%d", &c); // Reading number of columns
    mat[i] = (int *) malloc (c * sizeof (int));
    if (mat[i] == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
}
```

```
...
```

```
for (i=0; i<R; i++) {
    free (mat[i]);
}
```

3. Variable number of rows but predefined number of columns (Figure 3.8(c)).

```
#define C 5
```

```
int r;
int (*mat)[C];
scanf ("%d", &r); // Reading number of rows
mat = (int (*)[C]) malloc (r * sizeof (int [C]));
if (mat == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}
```

```
...
```

```
free (mat);
```

Note that an example of such a data structure is the parameter of the main function argv. The main function can be defined as:

```
int main (int argc, char *argv[]) { ...
```

or

```
int main (int argc, char **argv) { ...
```

which makes explicit the definition of argv as an array of pointers to strings.

#### 4. Variable number of rows and columns (Figure 3.8(d)).

```
int i, r, c;
int **mat;

scanf ("%d", &r); // Reading number of rows
mat = (int **) malloc (r * sizeof (int *));
if (mat == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}

for (i=0; i<r, i++) {
    scanf ("%d", &c); // Reading number of columns
    mat[i] = (int *) malloc (c * sizeof (int));
    if (mat[i] == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
}

...

for (i=0; i<r; i++) {
    free (mat[i]);
}
free (mat)
```

## 2D Arrays of Structures

Obviously, the previous 4 cases can be mapped on composite structures as follows. Let us suppose to define the following C structures

```
typedef struct {
    int v[C];
} static_row;
```

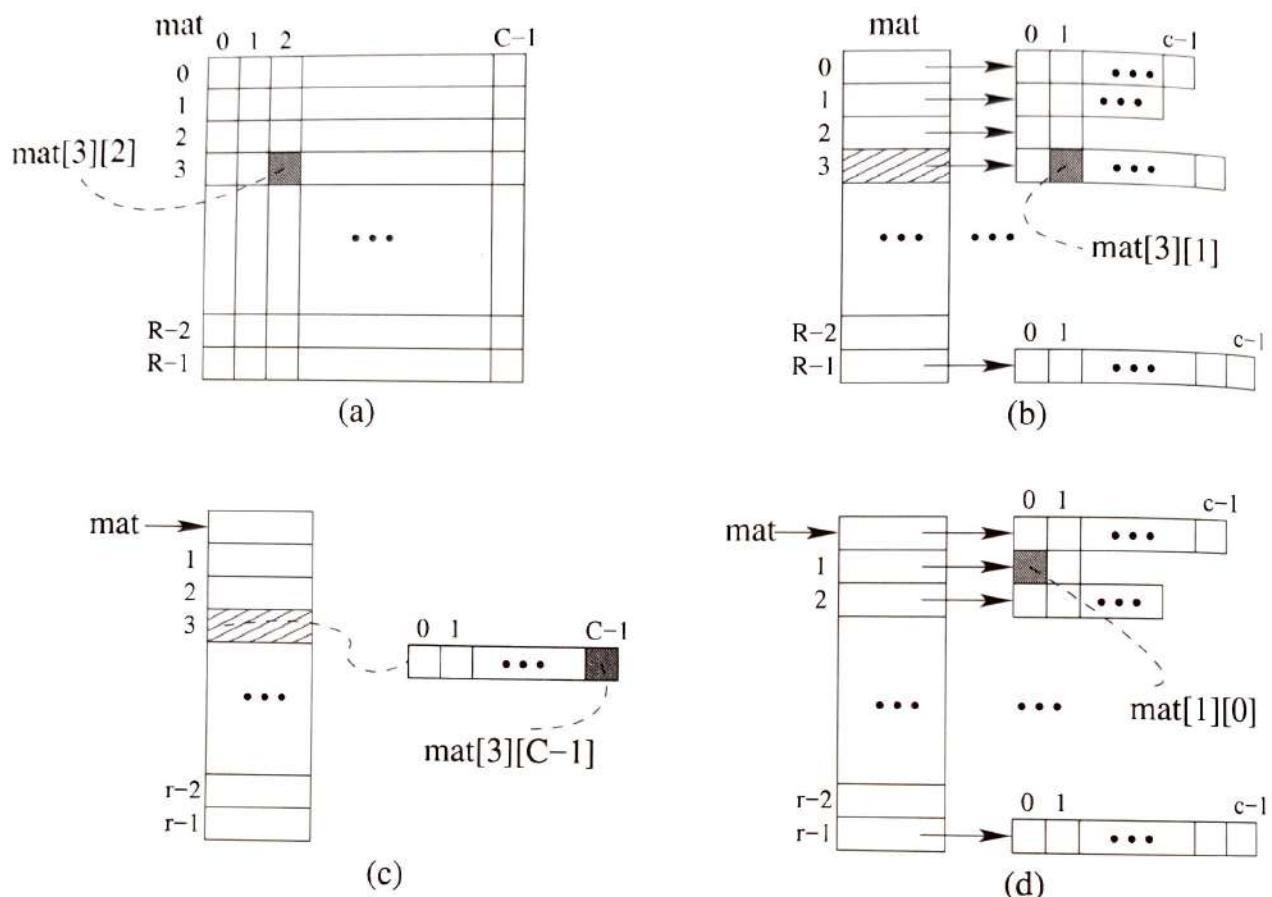
```
typedef struct {
    int *v;
} dynamic_row;
```

where C is a pre-defined constant Then, all previous schemes can be obtained as follows

#### 1. Number of rows and columns predefined:

```
#define R 10
#define C 5
```

```
static_row mat[R];
```



**Figure 3.8** Static and Dynamic 2D Array Definition.

2. Number of rows predefined and number of columns variable:

```
#define R 10
```

```
int i, c;
dynamic_row mat[R];
for (i=0; i<R; i++) {
    scanf ("%d", &c);
    mat[i].v = (int *) malloc (c * sizeof (int));
    if (mat[i].v == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
}
```

3. Number of rows variable ans number columns predefined.

```
#define C 5
```

```
int r;
static_row *mat;

scanf ("%d", &r);
mat = (static_row *) malloc (r * sizeof (static_row));
if (mat == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
```

- }
4. Number of rows and number of columns variable.

```

int i, r, c;
dynamic_row *mat;

scanf ("%d", &r);
mat = (dynamic_row *) malloc (r * sizeof (dynamic_row));
if (mat == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
}

for (i=0; i<r, i++) {
    scanf ("%d", &c);
    mat[i].v = (int *) malloc (c * sizeof (int));
    if (mat[i] == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
}

```

Notice that `free` operations are not indicated in the previous code segments. Moreover, in all cases element `[i][j]` can be referred as `mat[i].v[j]`. The notation is then somehow a little bit more awkward, but the matrix basic type more modular, as it is quite simple to add new fields to the basic data structure. For example, with a matrix with variable number of element for each row, it could be useful to store this number for each row. With the former schemes that would require an extra data structure, whereas in this case it would be easy using the following data structure:

```

typedef struct {
    int *v;
    int size;
} dynamic_row;

```

## **2D Arrays and Function Parameters**

As a final remark notice that compilers usually complain when the programmer try to pass a two-dimensional array to a function expecting a pointer to a pointer. This is because the rule by which arrays decay into pointers is not applied recursively. An array of arrays (i.e., a two-dimensional array in C) decays into a pointer to an array, not a pointer to a pointer. Pointers to arrays can be confusing, and must be treated carefully.

the function's declaration must match. That is, it must be of type:

```

int array[N_ROWS][N_COLUMNS];
...
f(array);
void f(int a[N_ROWS][N_COLUMNS]) {
...}
or:
void f(int a[] [N_COLUMNS]) {

```

```
...
}
or
void f (int (*ap) [N_COLUMNS]) {  
...  
}
```

In the first two declarations, the compiler performs the usual implicit parameter rewriting of “array of array” to “pointer to array”. In the third form the pointer declaration is explicit. Since the called function does not allocate space for the array, it does not need to know the overall size, so the number of rows, N\_ROWS, can be omitted. The width of the array is still important, so the column dimension N\_COLUMNS (and, for three- or more dimensional arrays, the intervening ones) must be retained. If a function is already declared as accepting a pointer to a pointer, it is almost certainly meaningless to pass a two-dimensional array directly to it.

The following example summarizes several ways in which static and dynamic 2D arrays can be defined and passed to functions.

**Example 3.33** Let us suppose we have defined statically- and dynamically-allocated multidimensional arrays as:

```
int array[N_ROWS] [N_COLUMNS];
int **array1;
int **array2;
int *array3;
int (*array4) [N_COLUMNS];
```

Given the declarations

```
void fla (int a[] [N_COLUMNS], int nrows, int ncolumns);
void f1b (int (*a) [N_COLUMNS], int nrows, int ncolumns);
void f2 (int *aryp, int nrows, int ncolumns);
void f3 (int **pp, int nrows, int ncolumns);
```

The following calls should work as expected:

```
fla (array, N_ROWS, N_COLUMNS);
f1b (array, N_ROWS, N_COLUMNS);
fla (array4, nrows, N_COLUMNS);
f1b (array4, nrows, N_COLUMNS);
f2 (&array[0][0], N_ROWS, N_COLUMNS);
f2 (*array, N_ROWS, N_COLUMNS);
f2 (*array2, nrows, ncolumns);
f2 (array3, nrows, ncolumns);
f2 (*array4, nrows, N_COLUMNS);
f3 (array1, nrows, ncolumns);
f3 (array2, nrows, ncolumns);
```

## 3.9 Matrix Multiplication

### Specifications

Two matrices of integer values ( $M_1$  and  $M_2$ ) are stored in two files. Each file reports the number of rows and columns of the matrix (on the first line), and then the matrix values themselves (see the example for further details).

Write a program able to compute the matrix product

$$M_3 = M_1 \cdot M_2$$

using the standard matrix multiplication algorithm. The resulting matrix  $M_3$  must be stored into an output file with the same format of the two input files. The program has to verify whether the product can be computed depending on the matrices size. All file names are received on the command line.

**Example 3.34** Let the two input files, storing  $M_1$  and  $M_2$ , be the ones represented on the left-hand side and at the center of the following picture:

$$\begin{array}{ccc} \begin{matrix} 2 & 3 \\ -1 & 5 & 0 \\ 4 & 0 & 2 \end{matrix} & \begin{matrix} 3 & 4 \\ 0 & -3 & -1 & -1 \\ 1 & 1 & 0 & 2 \\ -1 & 2 & 4 & 3 \end{matrix} & \begin{matrix} 2 & 4 \\ 5 & 8 & 1 & 11 \\ -2 & -8 & 4 & 2 \end{matrix} \end{array}$$

then  $M_3$  will have size (2·4), and the output file will have the content represented on the right-hand side of the previous picture.

## Solution

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
5 int **matrix_read(char *, int *, int *);
6 int **product_compute(int **, int, int, int **, int, int);
7 void matrix_write(char *, int **, int, int);
8 void matrix_quit(int **, int);
9
10 /*
11  * main program
12  */
13 int main (int argc, char *argv[]) {
14     int **m1, **m2, **m3;
15     int r1, c1, r2, c2;
16
17     if (argc < 4) {
18         fprintf(stderr, "Error: missing parameter.\n");
19         fprintf(stderr, "Run as: %s <m1_file> <m2_file> <m3_file>\n", argv[0]);
20         return EXIT_FAILURE;
21     }
22
23     m1 = matrix_read(argv[1], &r1, &c1);
24     m2 = matrix_read(argv[2], &r2, &c2);
25     if (c1 != r2) {
26         fprintf(stderr, "Error: incompatible dimensions.\n");
27         return EXIT_FAILURE;
28     }
29     m3 = product_compute(m1, r1, c1, m2, r2, c2);
30     matrix_write(argv[3], m3, r1, c2);
31
32     matrix_quit(m1, r1);
33     matrix_quit(m2, r2);
34     matrix_quit(m3, r1);
35     return EXIT_SUCCESS;
36 }
37
38 */

```

```
39     * matrix input
40     */
41 int **matrix_read (char *name, int *r, int *c) {
42     int i, j, **m;
43     FILE *fp;
44
45     fp = fopen(name, "r");
46     if (fp == NULL) {
47         fprintf(stderr, "File open error (file=%s).\n", name);
48         exit(EXIT_FAILURE);
49     }
50
51     fscanf(fp, "%d %d", r, c);
52     m = (int **)malloc((*r) * sizeof(int *));
53     if (m == NULL) {
54         fprintf(stderr, "Memory allocation error.\n");
55         exit(EXIT_FAILURE);
56     }
57     for (i=0; i<*r; i++) {
58         m[i] = (int *)malloc((*c) * sizeof(int));
59         if (m[i] == NULL) {
60             fprintf(stderr, "Memory allocation error.\n");
61             exit(EXIT_FAILURE);
62         }
63         for (j=0; j<*c; j++) {
64             fscanf(fp, "%d", &m[i][j]);
65         }
66     }
67
68     fclose(fp);
69     return m;
70 }
71 */
72 /*
73     * compute the matrix product
74 */
75 int **product_compute (int **m1, int r1, int c1, int **m2, int r2, int c2) {
76     int i, j, k, **m3;
77
78     m3 = (int **)malloc(r1 * sizeof(int *));
79     if (m3 == NULL) {
80         fprintf(stderr, "Memory allocation error.\n");
81         exit(EXIT_FAILURE);
82     }
83
84     for (i=0; i<r1; i++) {
85         m3[i] = (int *)malloc(c2 * sizeof(int));
86         if (m3[i] == NULL) {
87             fprintf(stderr, "Memory allocation error.\n");
88             exit(EXIT_FAILURE);
89         }
90         for (j=0; j<c2; j++) {
91             m3[i][j] = 0;
92             for (k=0; k<c1; k++) {
93                 m3[i][j] += m1[i][k] * m2[k][j];
94             }
95         }
96     }
97     return m3;
```

```

98 }
99 /*
100 * matrix output
101 */
102 void matrix_write (char *name, int **m, int r, int c) {
103     int i, j;
104     FILE *fp;
105
106     fp = fopen(name, "w");
107     if (fp == NULL) {
108         fprintf(stderr, "File open error (file=%s).\n", name);
109         exit(EXIT_FAILURE);
110     }
111
112     fprintf(fp, "%d %d\n", r, c);
113     for (i=0; i<r; i++) {
114         for (j=0; j<c; j++) {
115             fprintf(fp, "%d ", m[i][j]);
116         }
117         fprintf(fp, "\n");
118     }
119     fclose(fp);
120 }
121
122 /*
123 * matrix deallocation
124 */
125
126 void matrix_quit (int **m, int r) {
127     int i;
128
129     for (i=0; i<r; i++) {
130         free(m[i]);
131     }
132     free(m);
133
134     return;
135 }
```

## 3.10 String Sort

### Specifications

A file stores a set of strings (one string on each row of the file) of maximum length equal to 60 characters. Write a program able to read those strings, to store them in a dynamic matrix, to order them, and, finally, to store them in an output file. The input and output file names have to be read from the command line.

**Example 3.35** If the input file is the following:

```

here
there
is
a
set
of
strings
the output file will be the following:
```

a  
here  
is  
of  
set  
strings  
there

## Solution

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 #define MAX_LINE 60+1
6
7 /* function prototypes */
8 char **input_read (char *name, int *num_ptr);
9 void string_sort (char **strings, int num);
10 void output_write (char *name, char **strings, int num);
11
12 /*
13 * main program
14 */
15 int main (int argc, char *argv[]) {
16     char **strings;
17     int i, num;
18
19     if (argc < 3) {
20         fprintf(stderr, "Error: missing parameter.\n");
21         fprintf(stderr, "Run as: %s <input_file> <output_file>\n", argv[0]);
22         return EXIT_FAILURE;
23     }
24
25     strings = input_read(argv[1], &num);
26     string_sort(strings, num);
27     output_write(argv[2], strings, num);
28
29     for (i=0; i<num; i++) {
30         free(strings[i]);
31     }
32     free(strings);
33
34     return EXIT_SUCCESS;
35 }
36
37 /*
38 * load the input file contents
39 */
40 char **input_read (char *name, int *num_ptr) {
41     char word[MAX_LINE], **strings;
42     int n, i;
43     FILE *fp;
44
45     /* count the words */
46     fp = fopen(name, "r");
47     if (fp == NULL) {
48         fprintf(stderr, "File open error %s.\n", name);
```

```
49     exit(EXIT_FAILURE);
50 }
51 i = 0;
52 while (fscanf(fp, "%s", word) != EOF) {
53     i++;
54 }
55 fclose(fp);
56 n = *num_ptr = i;
57
58 /* allocate data structure */
59 strings = (char **)malloc(n * sizeof(char *));
60 if (strings == NULL) {
61     fprintf(stderr, "Memory allocation error.\n");
62     exit(EXIT_FAILURE);
63 }
64
65 /* save the words in the array */
66 fp = fopen(name, "r");
67 if (fp == NULL) {
68     fprintf(stderr, "File open error (file=%s).\n", name);
69     exit(EXIT_FAILURE);
70 }
71
72 for (i=0; i<n; i++) {
73     fscanf(fp, "%s", word);
74     strings[i] = (char *)malloc((strlen(word)+1) * sizeof(char));
75     if (strings[i] == NULL) {
76         fprintf(stderr, "Memory allocation error.\n");
77         exit(EXIT_FAILURE);
78     }
79     strcpy(strings[i], word);
80 }
81 fclose(fp);
82
83 return strings;
84 }
85
86 /*
87 * insertion sort on (dynamic) strings
88 */
89 void string_sort (char **strings, int num) {
90     int i, j;
91     char *ptr;
92
93     for (i=1; i<num; i++) {
94         ptr = strings[i];
95         j = i;
96         while (--j>=0 && strcmp(ptr, strings[j])<0) {
97             strings[j+1] = strings[j];
98         }
99         strings[j+1] = ptr;
100    }
101 }
102
103 /*
104 * write the output file
105 */
106 void output_write (char *name, char **strings, int num) {
```

```

108     FILE *fp;
109     int i;
110
111     fp = fopen(name, "w");
112     if (fp == NULL) {
113         fprintf(stderr, "File open error (file=%s).\n", name);
114         exit(EXIT_FAILURE);
115     }
116
117     for (i=0; i<num; i++) {
118         fprintf(fp, "%s\n", strings[i]);
119     }
120     fclose(fp);
121 }
```

## 3.11 Cyclist Training

### Specifications

During a training session each athlete of a group of professional cyclists is checked during each lap. For each athlete all lap times are stored in a file with the following format. The first line of the file stores the number of cyclist in the group. Then, for each cyclist, the file stores:

- ▷ On the first line, his/her name (string of 30 characters at most), identifier (integer value), and number of laps performed.
- ▷ On the second line, all lap times,  $time_1 \ time_2 \ \dots \ time_N$ , stored as real values.

Write a program that, after reading the file and storing its content in a proper data structure, is able to reply to the following menu inquiry:

- ▷ **list**: the program prints-out the number of athletes, their names, identifiers, and number of laps performed.
- ▷ **detail name**: given an athlete name, the program prints-out his/her identifier, and all lap times.
- ▷ **best**: the program prints-out the name, identifier, all lap times and the average lap time for the athlete whose average lap time is smaller.
- ▷ **stop**: end the program.

Notice that all operations can be performed more than once till the **stop** command is issued.

**Example 3.36** Let the following be the input file:

```

4
Rossi 100 3
1.30 1.38 1.29
Bianchi 101 5
1.46 1.43 1.42 1.51 1.28
Neri 117 2
1.26 1.34
Verdi 89 4
2.01 1.45 1.43 1.38
```

The following is a run example of the program (underlined text is inserted by the user):

```

Input file name: cyclist.txt
Command? list
Number of athletes : 4
Name:Rossi      #Id:100 #Laps:3
Name:Bianchi    #Id:101 #Laps:5
Name:Neri       #Id:117 #Laps:2
Name:Verdi      #Id:89  #Laps:4
Command? best
Name:Neri       #Id number:117 #Laps:2 Times: 1.26 1.34 (Average:1.30)
Command? details Bianchi
#Id:101 #Laps:5 Times: 1.46 1.43 1.42 1.51 1.28
Command? stop
Program ended.

```

## Solution

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 #define MAX 31
6
7 /* structure declaration */
8 typedef struct {
9     char *name;
10    int id;
11    int laps;
12    float *times;
13    float avg;
14 } cyclist_t;
15
16 /* function prototypes */
17 cyclist_t *file_read (int *, int *);
18 void list_write (cyclist_t *, int);
19 void detail_write (cyclist_t *, int);
20 void best_write (cyclist_t *, int);
21
22 /*
23  * main program
24  */
25 int main (void) {
26     char cmd[MAX];
27     int i, num, best;
28     cyclist_t *team = file_read(&num, &best);
29
30     do {
31         printf(stdout, "\nAvailable commands:\n");
32         printf(stdout, "list           - list all the cyclists' data\n");
33         printf(stdout, "details <name> - show the details of a single cyclist\n");
34         printf(stdout, "best            - show the details of the best cyclist\n");
35         printf(stdout, "stop            - quit the program\n\n");
36
37         printf(stdout, "Command? ");
38         scanf("%s", cmd);
39         if (strcmp(cmd, "list") == 0) {
40             list_write(team, num);
41         } else if (strcmp(cmd, "details") == 0) {

```

```
42     detail_write(team, num);
43 } else if (strcmp(cmd, "best") == 0) {
44     best_write(team, best);
45 } else if (strcmp(cmd, "stop") != 0) {
46     fprintf(stderr, "Invalid command\n");
47 }
48 } while (strcmp(cmd, "stop") != 0);
49
50 for(i=0; i<num; i++) {
51     free(team[i].name);
52     free(team[i].times);
53 }
54 free(team);
55
56 fprintf(stdout, "End of program.\n");
57
58 return EXIT_SUCCESS;
59 }
60
61 /*
62 * load the input file
63 */
64 cyclist_t *file_read (int *num_ptr, int *best) {
65     cyclist_t *team;
66     char name[MAX];
67     float best_avg=-1;
68     int i, j;
69     FILE* fp;
70
71     fprintf(stdout, "Input file name: ");
72     scanf("%s", name);
73     fp = fopen(name, "r");
74     if (fp == NULL) {
75         fprintf(stderr, "File open error (file=%s).\n", name);
76         exit(EXIT_FAILURE);
77     }
78
79     fscanf(fp, "%d", num_ptr);
80     team = (cyclist_t *)malloc((*num_ptr) * sizeof(cyclist_t));
81     if (team == NULL) {
82         fprintf(stderr, "Memory allocation error.\n");
83         exit(EXIT_FAILURE);
84     }
85
86     for (i=0; i<*num_ptr; i++) {
87         fscanf(fp, "%s %d %d", name, &team[i].id, &team[i].laps);
88         team[i].name = strdup(name);
89         team[i].times = (float *)malloc(team[i].laps * sizeof(float));
90         if (team[i].name==NULL || team[i].times==NULL) {
91             fprintf(stderr, "Memory allocation error.\n");
92             exit(EXIT_FAILURE);
93         }
94         team[i].avg = 0;
95
96         for (j=0; j<team[i].laps; j++) {
97             fscanf(fp, "%f", &team[i].times[j]);
98             team[i].avg += team[i].times[j];
99         }
100 }
```

```
101     team[i].avg /= team[i].laps;
102     if (best_avg<0 || team[i].avg <= best_avg) {
103         best_avg = team[i].avg;
104         *best = i;
105     }
106 }
107 }
108 fclose(fp);
109 return team;
110 }
111 }
112 /*
113 * list all the team components
114 */
115 void list_write (cyclist_t *team, int num) {
116     int i;
117
118     fprintf (stdout, "Number of cyclists: %d\n", num);
119     for (i=0; i<num; i++) {
120         fprintf(stdout, "- Name: %s\n", team[i].name);
121         fprintf(stdout, " Id : %d\n", team[i].id);
122         fprintf(stdout, " Laps: %d\n", team[i].laps);
123     }
124 }
125 }
126 /*
127 * list the details of a single cyclist
128 */
129 void detail_write (cyclist_t *team, int num) {
130     int i, j, found=0;
131     char name[MAX];
132
133     scanf("%s", name);
134     for (i=0; i<num && !found; i++) {
135         if (strcmp(name, team[i].name) == 0) {
136             fprintf(stdout, " Id : %d\n", team[i].id);
137             fprintf(stdout, " Laps : %d\n", team[i].laps);
138             fprintf(stdout, " Times: ");
139             for (j=0; j<team[i].laps; j++) {
140                 fprintf(stdout, "% .2f ", team[i].times[j]);
141             }
142             fprintf(stdout, "\n");
143             found = 1;
144         }
145     }
146     if (!found) {
147         fprintf(stdout, "Cyclist not found.\n");
148     }
149 }
150 }
151 /*
152 * list the details of a best cyclist
153 */
154 void best_write (cyclist_t *team, int best) {
155     int j;
156
157     fprintf(stdout, " Best cyclist: %s\n", team[best].name);
158     fprintf(stdout, " Id : %d\n", team[best].id);
```

```

160     fprintf(stdout, " Laps : %d\n", team[best].laps);
161     fprintf(stdout, " Times: ");
162     for (j=0; j<team[best].laps; j++) {
163         fprintf(stdout, "% .2f ", team[best].times[j]);
164     }
165     fprintf(stdout, "\n Average time: %.2f\n", team[best].avg);
166 }
```

## 3.12 Hotel Reservations

### Specifications

A hotel manager would like to restore his hotel such that it will have a proper room for all tourists who have issued a reservation for the next holiday season. A file stores all reservations done before the restoration activity. For each reservation the file stores the following information:

lastName firstName arrivalDay departureDay

where the first two fields are string of 60 characters at most, and the last two indicate the arrival and departure days numbering the day through the year from 1 to 365 (e.g., 32 indicates the first of February). Each room will be occupied at the arrival day and free at the departure one.

Write a program to:

- ▷ Compute the minimum number of rooms able to satisfy all reservation. Assume all rooms are equivalent and can satisfy all reservations.
- ▷ Evaluate a possible tourist disposition. Suppose, to simplify the problem, that a tourist can be moved from one room to another one during his/her staying.

Write down this information on a file with the following format:

- ▷ The first line stores the number of necessary rooms.
- ▷ Each block of subsequent lines store the room number, and for each day ion which the room is full the last and first names of the tourist occupying the room.

Read the file names from standard input.

**Example 3.37** Let the following be the input file:

```

3
Bond James 2 4
Xavier Charles 3 5
Fantozzi Ugo 1 3
```

The output file will be the following:

```

Number of required rooms = 2
Room number 1
day=1 Fantozzi Ugo
day=2 Bond James
day=3 Bond James
day=4 Xavier Charles
Room number 2
day=2 Fantozzi Ugo
day=3 Xavier Charles
```

## Solution

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 #define MAX      61
6 #define DAY      365
7
8 /* structure declaration */
9 typedef struct {
10     char *surname;
11     char *name;
12     int in;
13     int out;
14 } request_t;
15
16 /* function prototypes */
17 request_t *file_read(int *);
18 int roomNumber(request_t *, int);
19 void reservation_make(request_t *, int, int *[], int);
20 void file_write(int *[], int, request_t *);
21
22 /*
23 * main program
24 */
25 int main (void) {
26     int *reservations[DAY];
27     request_t *requests;
28     int i, n, roomNum;
29
30     requests = file_read(&n);
31     roomNum = roomNumber(requests, n);
32     reservation_make(requests, n, reservations, roomNum);
33     file_write(reservations, roomNum, requests);
34
35     for (i=0; i<n; i++) {
36         free(requests[i].name);
37         free(requests[i].surname);
38     }
39     free(requests);
40     for (i=0; i<DAY; i++) {
41         free(reservations[i]);
42     }
43
44     return EXIT_SUCCESS;
45 }
46
47 /*
48 * load the request file
49 */
50 request_t *file_read (int *n_ptr) {
51     request_t *requests;
52     char word[MAX];
53     FILE *fp;
54     int i;
55
56     fprintf(stdout, "Input file name: ");
57     scanf("%s", word);
```

```
58     fp = fopen(word, "r");
59     if (fp == NULL) {
60         fprintf(stderr, "File open error (file=%s).\n", word);
61         exit(EXIT_FAILURE);
62     }
63
64     fscanf(fp, "%d", n_ptr);
65     requests = (request_t *)malloc((*n_ptr) * sizeof(request_t));
66     if (requests == NULL) {
67         fprintf(stderr, "Memory allocation error.\n");
68         exit(EXIT_FAILURE);
69     }
70
71     for (i=0; i<*n_ptr; i++) {
72         fscanf(fp, "%s", word);
73         requests[i].surname = strdup(word);
74         fscanf(fp, "%s", word);
75         requests[i].name = strdup(word);
76         if (requests[i].surname==NULL || requests[i].name==NULL) {
77             fprintf(stderr, "Memory allocation error.\n");
78             exit(EXIT_FAILURE);
79         }
80         fscanf(fp, "%d %d", &requests[i].in, &requests[i].out);
81     }
82     fclose (fp);
83
84     return requests;
85 }
86
87 /*
88  * compute the room number
89 */
90 int roomNumber(request_t *requests, int n)
91 {
92     int i, j, roomNum=0, tmp;
93
94     /* for each day ... */
95     for (i=1; i<=DAY; i++) {
96         tmp = 0;
97
98         /* ... and for each reservation ... */
99         for (j=0; j<n; j++) {
100             /* ... check the day */
101             if (i>=requests[j].in && i<requests[j].out) {
102                 tmp++;
103             }
104         }
105
106         if (tmp > roomNum) {
107             roomNum = tmp;
108         }
109     }
110
111     return roomNum;
112 }
113
114 /*
115  * assign rooms to reservations
116 */
```

```

117 void reservation_make (
118     request_t *requests, int n, int *reservations[], int roomNum
119 ) {
120     int i, j, k, in, out, stop;
121
122     /* allocate and init structure to assign rooms */
123     for (i=0; i<DAY; i++) {
124         reservations[i] = (int *)malloc(roomNum * sizeof(int));
125         if (reservations[i] == NULL) {
126             fprintf(stderr, "Memory allocation error.\n");
127             exit(EXIT_FAILURE);
128         }
129         for (j=0; j<roomNum; j++) {
130             reservations[i][j] = -1;
131         }
132     }
133
134     /* for each request ... */
135     for (k=0; k<n; k++) {
136         in = requests[k].in - 1;
137         out = requests[k].out - 1;
138
139         /* ... reserve all days ... */
140         for (i=in; i<out; i++) {
141             /* ... in the first available room */
142             for (stop=0, j=0; j<roomNum && stop==0; j++) {
143                 if (reservations[i][j] == -1) {
144                     reservations[i][j] = k;
145                     stop = 1;
146                 }
147             }
148         }
149     }
150 }
151
152 /*
153 * write the output file
154 */
155 void file_write (int *reservations[], int roomNum, request_t *requests) {
156     char name[MAX];
157     int i, j, k;
158     FILE *fp;
159
160     fprintf(stdout, "Output file name: ");
161     scanf("%s", name);
162     fp = fopen (name, "w");
163     if (fp == NULL) {
164         fprintf(stderr, "File open error (file=%s).\n", name);
165         exit(EXIT_FAILURE);
166     }
167
168     fprintf(fp, "Rooms needed = %d\n", roomNum);
169     for (j=0; j<roomNum; j++) {
170         fprintf(fp, "Room n. %d\n", j+1);
171         for (i=0; i<DAY; i++) {
172             if (reservations[i][j] != -1) {
173                 /* Get data from the original structure */
174                 k = reservations[i][j];
175                 fprintf(fp, "day=%d %s %s\n", i+1, requests[k].surname, requests[k].name);
176             }
177         }
178     }
179 }
```

```

176      }
177      }
178  }
179 }
```

### 3.13 Multi Merge

#### Specifications

A matrix of strings is stored in a file with the following format:

```
R C
string11 string12 string13 ... string1C
...
stringR1 stringR2 stringR3 ... stringRC
```

where R and C are integer numbers which indicate the matrix size, and  $\text{string}_{ij}$  are the strings within the matrix. Each string has a maximum size of 20 characters and it does not include space characters. Each row of the matrix is already alphabetically ordered.

Write a program able to merge the R matrix rows in an unique array of strings, where strings are again ordered in alphabetic order. Store this array in an output file.

Check whether the original matrix rows are already ordered. Notice that it is forbidden to use a unique array to store all strings and to order this array with an ordering algorithm starting from scratch.

**Example 3.38** Let the input file be the following:

```
4 3
milano torino venezia
bari genova taranto
firenze napoli roma
bologna cagliari palermo
```

the program has to generate the following output file:

```
12
bari
bologna
cagliari
firenze
genova
milano
napoli
palermo
roma
taranto
torino
venezia
```

#### Solution

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
```

```

4 #define MAX_LEN 21
5
6 /* function prototypes */
7 char ***file_read(int *Rptr, int *Cptr);
8 char **matrix_merge(char ***matrix, int R, int C);
9 void result_write(char **array, int dim);
10 void memory_dispose(char ***matrix, char **array, int R, int C);
11
12 /*
13  * main program
14  */
15 int main (void) {
16     char ***matrix, **array;
17     int R, C;
18
19     matrix = file_read(&R, &C);
20     array = matrix_merge(matrix, R, C);
21     result_write(array, R*C);
22     memory_dispose(matrix, array, R, C);
23
24     return EXIT_SUCCESS;
25 }
26
27 /*
28  * load the string matrix
29  */
30
31 char ***file_read(int *Rptr, int *Cptr) {
32     char word[MAX_LEN], ***mx;
33     int r, c, i, j;
34     FILE *fp;
35
36     fprintf(stdout, "Input file name: ");
37     scanf("%s", word);
38     fp = fopen(word, "r");
39     if (fp == NULL) {
40         fprintf(stderr, "File open error (file=%s).\n", word);
41         exit(EXIT_FAILURE);
42     }
43
44     /* read dimensions and allocate the matrix */
45     fscanf(fp, "%d %d", &r, &c);
46     mx = (char ***)malloc(r * sizeof(char **));
47     if (mx == NULL) {
48         fprintf(stderr, "Memory allocation error.\n");
49         exit(EXIT_FAILURE);
50     }
51     for (i=0; i<r; i++) {
52         mx[i] = (char **)malloc(c * sizeof(char *));
53         if (mx[i] == NULL) {
54             fprintf(stderr, "Memory allocation error.\n");
55             exit(EXIT_FAILURE);
56         }
57     }
58
59     /* parse the file contents */
60     for (i=0; i<r; i++) {
61         for (j=0; j<c; j++) {
62             fscanf(fp, "%s", word);

```

```

63     /* check that the i-th row is sorted */
64     if ((j>0) && (strcmp(mx[i][j-1], word)>0)) {
65         fprintf(stderr, "Error: row %d NOT sorted.\n", i+1);
66         exit(EXIT_FAILURE);
67     }
68     mx[i][j] = strdup(word);
69     if (mx[i][j] == NULL) {
70         fprintf(stderr, "Memory allocation error.\n");
71         exit(EXIT_FAILURE);
72     }
73 }
74 }

75 fclose(fp);
76 *Rptr = r;
77 *Cptr = c;
78 return mx;
79 }

80 }

81 */

82 /*
83  * merge the matrix rows into a sorted array
84 */
85 char **matrix_merge (char ***matrix, int R, int C) {
86     int i, j, min_idx, *idx;
87     char **array, *min_word;
88

89     /* allocate the final array, plus an auxiliary one */
90     array = (char **)malloc(R * C * sizeof(char *));
91     idx = (int *)calloc(R, sizeof(int));
92     if ((array == NULL) || (idx == NULL)) {
93         fprintf(stderr, "Memory allocation error.\n");
94         exit(EXIT_FAILURE);
95     }

96     /* merge the matrix rows */
97     i = 0;
98     while (i < R*C) {
99         min_idx = -1;
100        for (j=0; j<R; j++) {
101            if (idx[j] < C) {
102                if ((min_idx == -1) || (strcmp(matrix[j][idx[j]], min_word) < 0)) {
103                    min_idx = j;
104                    min_word = matrix[min_idx][idx[min_idx]];
105                }
106            }
107        }
108        array[i++] = matrix[min_idx][idx[min_idx]++];
109    }
110    free(idx);
111    return array;
112 }

113 */

114 */

115 */

116 /*
117  * print the output file
118 */
119 void result_write (char **array, int dim) {
120     char name[MAX_LEN];
121     FILE *fp;

```

```

122 int i;
123
124 fprintf(stdout, "Output file name: ");
125 scanf("%s", name);
126 fp = fopen(name, "w");
127 if (fp == NULL) {
128     fprintf(stderr, "File open error (file=%s).\n", name);
129     exit(EXIT_FAILURE);
130 }
131
132 fprintf(fp, "%d\n", dim);
133 for (i=0; i<dim; i++) {
134     fprintf(fp, "%s\n", array[i]);
135 }
136
137 fclose(fp);
138 }
139
140 /*
141 * quit all the allocated memory
142 */
143 void memory_dispose (char ***matrix, char **array, int R, int C) {
144     int i, j;
145
146     for (i=0; i<R; i++) {
147         for (j=0; j<C; j++) {
148             free(matrix[i][j]);
149         }
150         free(matrix[i]);
151     }
152     free(matrix);
153     free(array);
154
155     return;
156 }
```

## 3.14 Cross-puzzle

### Specifications

Write a program able to solve the following puzzle.

A first file stores a matrix of characters whose size is reported on the first row to the file. A second file stores an unknown number of C strings of maximum size equal to 20 characters. The program has to look for those strings within the matrix in all possible directions (8 overall: Left-to-right, right-to-left, up-left-to-down-right, etc.). Notice that each character may belong to more than one string. The words that have been found have to be stored in a third file.

File names must be read on the command line. See the example for further details.

**Example 3.39** The following table shows the two input files and the output (generated by the program) file. Notice that in the first input file the character 'x' is used only to make the example more understandable, but in its place any other character can be used as well. Moreover, in the output file only "recognized" characters are retained, whereas all other are substituted by white spaces.

First File	Second File	Output File
<pre> 12 10 xxxxxxxtxxx xxpippoxxx xxxxxxpxxx xxxxxxoxxx xxxxxxlxxx pxxxxxxiuqx xlxxxxnxxx xxuxxxxoxxx xxxtxxxxxx xxxxoxxxxx xxxxxxxxxx xquoxxxxxx </pre>	<pre> pippo topolino quo pluto qui </pre>	<pre> t pippo p o l p   iuq l   n u   o t   o quo </pre>

## Solution

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define max(R,C) ((R>C)?R:C)
6
7 char **malloc2d (int, int);
8 char **free2d (char **, int);
9 void find (char *, char **, char **, int, int);
10 int find_all (int, int, char *, char **, char **, int, int);
11
12 int main (int argc, char *argv[]) {
13     char **matrixIn, **matrixOut;
14     char *word;
15     int i, j, R, C;
16     FILE *fp;
17
18     if (argc < 4) {
19         fprintf(stderr, "Error: missing parameter.\n");
20         fprintf(stderr, "Run as: %s <matrixFile> <wordFile> <outputFile>\n", argv[0]);
21         return 1;
22     }
23
24     fp = fopen(argv[1], "r");
25     if (fp == NULL) {
26         fprintf(stderr, "File open error (file=%s).\n", argv[1]);
27         return 1;
28     }
29     if (fscanf(fp, "%d%d%c", &R, &C) == EOF) {
30         fprintf(stderr, "Read file error.\n");
31         return 1;
32     }
33
34     matrixIn = malloc2d (R, C);
35     matrixOut = malloc2d (R, C);
36     word = (char *) malloc ((max(R,C)+1) * sizeof(char));
37     if (word==NULL) {
38         fprintf (stderr, "Memory allocation error.\n");
39         exit(EXIT_FAILURE);
40     }
41

```

```

42     for (i=0; i<R; i++) {
43         for (j=0; j<C; j++) {
44             fscanf(fp, "%c", &matrixIn[i][j]);
45         }
46         fscanf(fp, "%*c"); // Skip \n
47     }
48     fclose(fp);
49
50     fp = fopen(argv[2], "r");
51     if (fp == NULL) {
52         fprintf(stderr, "File open error (file=%s).\n", argv[2]);
53         return 1;
54     }
55     while (fscanf(fp, "%s", word) != EOF) {
56         find (word, matrixIn, matrixOut, R, C);
57     }
58     fclose(fp);
59
60     fp = fopen(argv[3], "w");
61     if (fp == NULL) {
62         fprintf(stderr, "File open error (file=%s).\n", argv[3]);
63         return 1;
64     }
65     for (i=0; i<R; i++) {
66         for (j=0; j<C; j++) {
67             fprintf(fp, "%c", matrixOut[i][j]);
68         }
69         fprintf(fp, "\n");
70     }
71     fclose(fp);
72
73     free (word);
74     matrixIn = free2d (matrixIn, R);
75     matrixOut = free2d (matrixOut, R);
76
77     return 0;
78 }
79
80
81 char **malloc2d(int r, int c) {
82     int i, j;
83     char **mat;
84
85     mat = (char **) malloc (r * sizeof(char *));
86     if (mat == NULL) {
87         fprintf (stderr, "Memory allocation error.\n");
88         exit(EXIT_FAILURE);
89     }
90     for (i=0; i<r; i++) {
91         mat[i] = (char *) malloc(c * sizeof (char));
92         if (mat[i]==NULL) {
93             fprintf (stderr, "Memory allocation error.\n");
94             exit(EXIT_FAILURE);
95         }
96     }
97
98     for (i=0; i<r; i++) {
99         for (j=0; j<c; j++) {
100            mat[i][j] = ' ';
101        }
102    }
103 }
```

```
101      }
102  }
103
104  return mat;
105 }
106
107 char **free2d(char **mat, int r) {
108     int i;
109
110     for (i=0; i<r; i++) {
111         free (mat[i]);
112     }
113     free (mat);
114
115     return mat;
116 }
117
118 void find (char *word, char **matrixIn, char **matrixOut, int R, int C) {
119     int i, j;
120
121     for (i=0; i<R; i++) {
122         for (j=0; j<C; j++) {
123             if (find_all(i, j, word, matrixIn, matrixOut, R, C)) {
124                 return;
125             }
126         }
127     }
128
129     return;
130 }
131
132 int find_all (
133     int row, int col, char *word, char **mIn, char **mOut, int R, int C
134 ) {
135     char flag;
136     int r, c, i, j;
137     int offset[2][8] = { {0, -1, -1, -1, 0, 1, 1, 1},
138                          {1, 1, 0, -1, -1, -1, 0, 1} };
139
140     for (i=0; i<8; i++) {
141         flag = 1;
142         for (j=0; j<strlen(word) && flag; j++) {
143             r = row + j * offset[0][i];
144             c = col + j * offset[1][i];
145
146             if (r<0 || r>=R || c<0 || c>=C || mIn[r][c]!=word[j]) {
147                 flag = 0;
148             }
149         }
150
151         if (flag == 1) {
152             for (j=0; j<strlen(word); j++) {
153                 mOut[row+j*offset[0][i]][col+j*offset[1][i]] = word[j];
154             }
155             return 1;
156         }
157     }
158
159     return 0;
```

160 }

## 3.15 Snakes

### Specifications

Write a program able to search snakes on a steady background as follows.

A ground area is represented by a matrix of characters stored in a file. The first row of the file reports the number of rows and columns of the matrix. The matrix uses:

- ▷ Points ‘.’ to indicate the background.
- ▷ The ‘+’ symbol to indicate the head of a snake.
- ▷ The star ‘\*’ to indicate the body of the snake.

Write a program which, once received the file name on the command line, is able to count the number of snakes in the area and print-out the minimum and maximum snake length.

Notice that snakes always lie in a “nice” way on the ground, i.e., they lie straight, they are not twisted on themselves, and they do not overlap to each other.

**Example 3.40** The following is a correct input file:

```
5 20
.*.*****+.*****
...*****+.*****...
.....*....*...
.....+.*****.
.....***+.*****
```

and the program has to print-out the following information:

```
Snake number = 4
Min length = 2
Max length = 7
```

### Solution

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <limits.h>
5
6 typedef struct {
7     char value;
8     int visited;
9 } element_t;
10
11 element_t **malloc2d (int, int);
12 element_t **free2d (element_t **, int);
13 int map_read(char *, element_t ***, int *, int *);
14 int length_evaluate(element_t **, int, int, int, int);
15
16 int main (int argc, char *argv[]) {
17     element_t **map;
18     int nSnake=0, minLen=INT_MAX, maxLen=0;
19     int nr, nc, i, j, length;
```

```
21     if (!map_read(argv[1], &map, &nr, &nc)) {
22         return 1;
23     }
24
25     for (i=0; i<nr; i++) {
26         for (j=0; j<nc; j++) {
27             if (map[i][j].value == '+') {
28                 nSnake++;
29                 length = length_evaluate(map, nr, nc, i, j);
30                 if (length < minLen) {
31                     minLen = length;
32                 }
33                 if (length > maxLen) {
34                     maxLen = length;
35                 }
36             }
37         }
38     }
39
40     free2d (map, nr);
41
42     fprintf(stdout, "Snake number = %d\n", nSnake);
43     fprintf(stdout, "Min length = %d\n", minLen);
44     fprintf(stdout, "Max length = %d\n", maxLen);
45     return 0;
46 }
47
48 element_t **malloc2d(int r, int c)
49 {
50     int i;
51     element_t **mat;
52
53     mat = (element_t **) malloc (r * sizeof(element_t *));
54     if (mat == NULL) {
55         fprintf (stderr, "Memory allocation error.\n");
56         exit(EXIT_FAILURE);
57     }
58     for (i=0; i<r; i++) {
59         mat[i] = (element_t *) malloc(c * sizeof (element_t));
60         if (mat[i]==NULL) {
61             fprintf (stderr, "Memory allocation error.\n");
62             exit(EXIT_FAILURE);
63         }
64     }
65
66     return mat;
67 }
68
69 element_t **free2d (element_t **mat, int r) {
70     int i;
71
72     for (i=0; i<r; i++) {
73         free (mat[i]);
74     }
75     free (mat);
76
77     return mat;
78 }
79
```

```

80 int map_read (char *fname, element_t ***mapP, int *nr, int *nc) {
81     element_t **map;
82     char c;
83     FILE *fp;
84     int i, j;
85
86     fp = fopen(fname, "r");
87     if (fp == NULL) {
88         fprintf(stderr, "File open error (file=%s).\n", fname);
89         return 0;
90     }
91
92     if (fscanf(fp, "%d %d", nr, nc) != 2) {
93         fprintf(stderr, "Error while parsing the input file.\n");
94         fclose(fp);
95         return 0;
96     }
97     fscanf(fp, "%c", &c);
98
99     map = malloc2d (*nr, *nc);
100    for (i=0; i<*nr; i++) {
101        for (j=0; j<*nc; j++) {
102            if (fscanf(fp, "%c", &map[i][j].value) != EOF) {
103                map[i][j].visited = 0;
104            } else {
105                fprintf(stderr, "Error while parsing the input file.\n");
106                fclose(fp);
107                return 0;
108            }
109        }
110        fscanf(fp, "%c", &c); // Skip newline '\n'
111    }
112
113    fclose(fp);
114
115    *mapP = map;
116
117    return 1;
118 }
119
120 int length_evaluate (element_t **map, int nr, int nc, int r, int c) {
121     int end=0, length=1, found, r_new, c_new, i, j;
122
123     map[r][c].visited = 1;
124
125     while (!end) {
126         found = 0;
127         for (i=r-1; i<=r+1 && found==0; i++) {
128             for (j=c-1; j<=c+1 && found==0; j++) {
129                 if (i>=0 && i<nr && j>=0 && j<nc) {
130                     if (map[i][j].value=='*' && !map[i][j].visited) {
131                         found = 1;
132                         r_new = i;
133                         c_new = j;
134                     }
135                 }
136             }
137         }
138     }

```

```

139     if (found) {
140         length++;
141         r = r_new;
142         c = c_new;
143         map[r][c].visited = 1;
144     } else {
145         end = 1;
146     }
147 }
148
149 return length;
150 }
```

## 3.16 Hospitals

### Specifications

Write a C program able to deal with all hospital reservations in a large city for an entire year. Specifications are the following.

A file stores all necessary pieces of information for each hospital, each unit (department) within the hospital, and all places (beds) available in those units. The first row of the file indicates the number of subsequent rows in the entire file. Each subsequent row has the following format:

hospitalName unitName numberOfBeds

where:

- ▷ hospitalName and unitName are C strings of maximum 25 characters.
- ▷ numberOfBeds indicates the number of beds (places) available during the entire year, i.e., for 365 days per year.

It is possible to suppose that all years have 365 days, and that days are numbered starting from 1 on (i.e., 32 is the first day of February).

The program, once read the file name from standard input, has to be able to implement the following commands:

- ▷ **reserve unitName n**  
reserves a free bed for n (integer value) days as soon as this reservation is possible in the indicated unit in one of the hospital. In other words, the command selects the hospital in which the waiting time for the desired unit is smaller, and make the reservation on one available bed. The program has to print out "Reservation impossible!" when there is no availability (for the requested period in any hospital). "Reservation done hospitalName, from=dayIn, to=dayOut, bedNumber=n" where beds are numbered starting from 1 in each hospital. If more than one unit or more than one bed satisfy the request the program can make random choices. Each bed is considered full during both the arrival and the departure days. Notice that it is not possible to change the bed place to a patient during his/her hospitalization.
- ▷ **remove hospitalName unitName from to**  
deletes a reservation (if it exists), i.e., it frees a bed reserved in that hospital and that unit for the specified period of time. The program prints out "Reservation removed." or "Reservation NOT removed!" depending on the outcome of

the operation.  
 ▷ end  
 terminate the program.

**Example 3.41** Let us suppose to have the following file:

```
121
cto cardiology 1
molinette cardiology 1
cto surgery 31
... more file lines ...
```

Here there is an example of execution (the underlined text is introduced by the user):

```
File Name: Piemonte.txt
Command : reserve cardiology 8
Reservation done: molinette 1 8 bed 1.
Command : reserve cardiology 22
Reservation done: cto 1 22 bed 1.
Command : reserve cardiology 3
Reservation done: molinette 9 11 bed 1.
Command : remove molinette cardiology 1 12
Reservation NOT removed!
...
Command : end
```

## Solution

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 #define MAX    100
6 #define WORD   25+1
7 #define DAY    365
8
9 typedef struct reservation_s {
10     char hospital[WORD];
11     char unit[WORD];
12     int n;
13     int **beds;
14 } reservation_t;
15
16 void file_read (int *, reservation_t **);
17 void menu (int, reservation_t *);
18 void delete (int, reservation_t *);
19 void reserve (int, reservation_t *);
20
21 int main () {
22     reservation_t *reservation;
23     int i, j, n;
24
25     file_read (&n, &reservation);
26     menu (n, reservation);
27
28     for (i=0; i<n; i++) {
```

```
29     for (j=0; j<reservation[i].n; j++) {
30         free (reservation[i].beds[j]);
31     }
32     free (reservation[i].beds);
33 }
34 free (reservation);
35
36 return (0);
37 }
38
39 void file_read (int *n, reservation_t **reservationP) {
40     FILE *fp;
41     reservation_t *reservation;
42     char fileName[MAX];
43     int i, j, k;
44
45     fprintf (stdout, "Input file name: ");
46     scanf ("%s", fileName);
47
48     fp = fopen (fileName, "r");
49     if (fp==NULL) {
50         fprintf (stderr, "Error opening file!\n");
51         exit(EXIT_FAILURE);
52     }
53
54     if (fscanf (fp, "%d", n) != 1) {
55         fprintf (stderr, "Error reading file!\n");
56         exit(EXIT_FAILURE);
57     }
58
59     reservation = (reservation_t *) malloc (*n * sizeof (reservation_t));
60     if (reservation == NULL) {
61         fprintf (stderr, "Memory allocation error.\n");
62         exit(EXIT_FAILURE);
63     }
64
65     i = 0;
66     while (i<*n &&
67             fscanf (fp, "%s%s%d", reservation[i].hospital,
68                     &reservation[i].unit,
69                     &reservation[i].n) != EOF) {
70
71         reservation[i].beds = (int **) malloc (reservation[i].n * sizeof (int *));
72         if (reservation[i].beds == NULL) {
73             fprintf (stderr, "Memory allocation error.\n");
74             exit(EXIT_FAILURE);
75         }
76         for (j=0; j<reservation[i].n; j++) {
77             reservation[i].beds[j] = (int *) malloc (DAY * sizeof (int));
78             if (reservation[i].beds[j] == NULL) {
79                 fprintf (stderr, "Memory allocation error.\n");
80                 exit(EXIT_FAILURE);
81             }
82
83             /* Init structure */
84             for (j=0; j<reservation[i].n; j++) {
85                 for (k=0; k<DAY; k++) {
86                     reservation[i].beds[j][k]= 0;
87                 }
88             }
89         }
90     }
91 }
```

```

88     }
89     i++;
90 }
91 }
92 if (*n != i) {
93     fprintf (stderr, "Wrong file format!\n");
94     *n = i;
95 }
96 }
97 *reservationP = reservation;
98 fclose (fp);
99
100 return;
101 }
102 }
103 void menu (int n, reservation_t *reservation) {
104     char command[MAX];
105
106     do {
107         printf (stdout, "Comand> ");
108         scanf ("%s", command);
109
110         if (strcmp (command, "reserve") == 0) {
111             reserve (n, reservation);
112         } else
113             if (strcmp (command, "remove") == 0) {
114                 delete (n, reservation);
115             } else
116                 if (strcmp (command, "end") != 0) {
117                     fprintf (stderr, "Command error!\n");
118                 }
119     } while (strcmp (command, "end") != 0);
120
121     return;
122 }
123 }
124
125 void reserve (int n, reservation_t *reservation) {
126     char unit[MAX];
127     int i, j, k, length, from, to;
128     int stop1, stop2;
129     int minH, minB, minData;
130
131     scanf ("%s%d", unit, &length);
132
133     minH = minB = minData = (-1);
134     /* For each hospital i */
135     for (i=0; i<n; i++) {
136         /* IFF the unit is the right one */
137         if (strcmp (reservation[i].unit, unit) == 0) {
138             /* Look for bed available sooner */
139             for (j=0; j<reservation[i].n; j++) {
140                 for (stop1=0, from=0; from<=DAY-length && stop1==0; from++) {
141                     to = from + length;
142                     for (stop2=0, k=from; k<to && stop2==0; k++) {
143                         if (reservation[i].beds[j][k] != 0) {
144                             stop2 = 1;
145                         }
146                     }
147                 }
148             }
149         }
150     }
151 }
```

```

147     if (stop2==0) {
148         stop1 = 1;
149         if ((minData==(-1) || from<minData)) {
150             minH = i;
151             minB = j;
152             minData = from;
153         }
154     }
155 }
156 }
157 }
158 }
159
160 if (minData == (-1)) {
161     fprintf (stdout, "Reserve NOT done.\n");
162 } else {
163     /* Make the reservation */
164     for (k=minData; k<(minData+length); k++) {
165         reservation[minH].beds[minB][k] = 1;
166     }
167     fprintf (stdout, "Reserve done: %s %d %d letto %d.\n",
168             reservation[minH].hospital, minData+1, minData+length, minB+1);
169 }
170
171 return;
172 }
173
174 void delete (int n, reservation_t *reservation) {
175     char hospital[MAX], unit[MAX];
176     int i, j, k, from, to;
177     int stop1, stop2, stop3;
178
179 //fprintf (stdout, "Hospital Unit From To: ");
180 scanf ("%s%s%d%d", hospital, unit, &from, &to);
181
182 /* For each hospital */
183 for (stop1=0, i=0; i<n && stop1==0; i++) {
184     /* IFF the hospital and the unit are the right ones */
185     if (strcmp (reservation[i].hospital, hospital) == 0 &&
186         strcmp (reservation[i].unit, unit) == 0) {
187         /* Look for a bed */
188         for (stop2=0, j=0; j<reservation[i].n && stop2==0; j++) {
189             for (stop3=0, k=from-1; k<to && stop3==0; k++) {
190                 if (reservation[i].beds[j][k] != 1) {
191                     stop3 = 1;
192                 }
193             }
194             if (stop3 == 0) {
195                 stop2 = 1;
196                 /* Clean the reservation */
197                 for (k=from-1; k<to; k++) {
198                     reservation[i].beds[j][k] = 0;
199                 }
200             }
201         }
202         stop1 = 1;
203     }
204 }

```

```

206
207     if (stop3 == 0) {
208         fprintf (stdout, "Delete done.\n");
209     } else {
210         fprintf (stdout, "Delete NOT done.\n");
211     }
212
213     return;
214 }
```

## 3.17 Counter-intelligence Agency

### Specifications

A counter-intelligence agency decides to supervise the web, looking for encoded messages. Encoded messages are stored vertically on text files, i.e., they are written vertically on the same character on subsequent rows in standard text.

Write a program able to receive two parameters on the command line: A string (of undefined length) and a file name. The program has to look-for the string within the file, written vertically from top-to-bottom.

As text files may be really long and in any case they have an undefined length, the program has to optimize the memory used, i.e., the program can store into the main memory only a small portion of the entire file in each moment of time. The user can anyhow suppose that all rows within a file have a maximum length of 100 characters. Each time the word is located within the file, the program has to print-out the starting position for the word, in terms of the row and the column within the entire file.

**Example 3.42** Let be “terror” and “message.txt” the two strings received by the program on the command line. Let the following be the file content:

```

xxtxxxx xxx xx  xxxxxxxxxxxx
xxe xxx  xxxxxxxxxx xtx  xxxxxxxx xxxx
xxr xxxxxxxx xxr xxxxexx xbx x xxxx
xxrxxxxxx xxxxrxrxx xxrxxxxo x x xxxx
xxox xxx  xxxxxxxo x xxrx x mxxxxx
xxrxxxx  xxxxxxxrxxxxxx o  xxbx
xxxxxxxxxx xxxx  xxrxx
```

The program has to print-out the following information:

Word “terror” found on row 1, character 3  
 Word “terror” found on row 2, character 22

Word “terror” found on row 3, character 27

With the word “bomb”), the output would have been:

Word “bomb” fund on row 3, character 27

### Solution 1

In this solution we store segments of the input file in a dynamically allocated matrix. The search is performed along rows and columns of the matrix. Every time a new row is read from file, this row is read into the last row of the matrix, and all matrix rows are shifted one position up, such that the first row is discarded.

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
```

```
4  /* 100 + '\n' + '\0' */
5  #define MAX 102
6
7  /* function prototypes */
8  void search1(char [], char []);
9  void check1(char (*) [MAX], char [], int);
10 void search2(char [], char []);
11 void check2(char **, char [], int);
12
13 /*
14  * main program
15  */
16
17 int main (int argc, char *argv[]) {
18     if (argc < 3) {
19         fprintf(stderr, "Error: missing parameter.\n");
20         fprintf(stderr, "Run as: %s <keyword> <text_file>\n", argv[0]);
21         return EXIT_FAILURE;
22     }
23
24     fprintf(stdout, "Solution 1:\n");
25     search1(argv[2], argv[1]);
26     fprintf(stdout, "Solution 1:\n");
27     search2(argv[2], argv[1]);
28
29     return EXIT_SUCCESS;
30 }
31
32 /*
33  * search the keyword in the whole text file
34  */
35 void search1(char name[], char word[]) {
36     char (*monitor) [MAX];
37     int row, length;
38     FILE *fp;
39
40     fp = fopen(name, "r");
41     if (fp == NULL) {
42         fprintf(stderr, "File open error (file=%s).\n", name);
43         exit(EXIT_FAILURE);
44     }
45
46     length = strlen(word);
47     monitor = (char (*) [MAX])malloc(length * sizeof(char [MAX]));
48     if (monitor == NULL) {
49         fprintf(stderr, "Memory allocation error.\n");
50         exit(EXIT_FAILURE);
51     }
52
53     row = 0;
54     while (fgets(monitor[row%length], MAX, fp) != NULL) {
55         if (++row >= length) {
56             check1(monitor, word, row);
57         }
58     }
59
60     fclose(fp);
61     free(monitor);
62 }
```

```
63 /* check for the keyword in a text piece
64 */
65 void check1(char (*monitor) [MAX], char word[], int row) {
66     int i, j, equal, min=0, length=strlen(word);
67
68     /* search the shortest line */
69     for (i=1; i<length; i++) {
70         if (strlen(monitor[i]) < strlen(monitor[min])) {
71             min = i;
72         }
73     }
74
75     /* search on all columns */
76     for (j=0; j<strlen(monitor[min]); j++) {
77         equal = 1;
78         for (i=0; i<length && equal==1; i++) {
79             if (word[i] != monitor[(row+i)%length][j]) {
80                 equal = 0;
81             }
82         }
83
84         if (equal == 1) {
85             fprintf(stdout, "Word \"%s\" found in ", word);
86             fprintf(stdout, "row %d, char %d.\n", row-length+1, j+1);
87         }
88     }
89 }
90 }
91 }
92
93 /*
94 * search the keyword in the whole text file
95 */
96 void search2(char name[], char word[]) {
97     char **monitor, line[MAX];
98     int i, row, length;
99     FILE *fp;
100
101    fp = fopen(name, "r");
102    if (fp == NULL) {
103        fprintf(stderr, "File open error (file=%s).\n", name);
104        exit(EXIT_FAILURE);
105    }
106
107    length = strlen(word);
108    monitor = (char **)malloc(length * sizeof(char *));
109    if (monitor == NULL) {
110        fprintf(stderr, "Memory allocation error.\n");
111        exit(EXIT_FAILURE);
112    }
113    for (i=0; i<length; i++) {
114        monitor[i] = NULL;
115    }
116
117    row = 0;
118    while (fgets(line, MAX, fp) != NULL) {
119        /* shift previously read lines */
120        free(monitor[0]);
121        for (i=0; i<length-1; i++) {
```

```

122     monitor[i] = monitor[i+1];
123 }
124 monitor[i] = strdup(line);
125 if (monitor[i] == NULL) {
126     fprintf(stderr, "Memory allocation error.\n");
127     exit(EXIT_FAILURE);
128 }
129 if (++row >= length) {
130     check2(monitor, word, row);
131 }
132 }
133
134 fclose(fp);
135 for (i=0; i<length; i++) {
136     free(monitor[i]);
137 }
138 free(monitor);
139 }
140
141 /*
142 *   check for the keyword in a text piece
143 */
144 void check2(char **monitor, char word[], int row) {
145     int i, j, equal, min=0, length=strlen(word);
146
147     /* search the shortest line */
148     for (i=1; i<length; i++) {
149         if (strlen(monitor[i]) < strlen(monitor[min])) {
150             min = i;
151         }
152     }
153
154     /* search on all colums */
155     for (j=0; j<strlen(monitor[min]); j++) {
156         equal = 1;
157         for (i=0; i<length && equal==1; i++) {
158             if (word[i] != monitor[i][j]) {
159                 equal = 0;
160             }
161         }
162
163         if (equal == 1) {
164             fprintf(stdout, "Word \"%s\" found in ", word);
165             fprintf(stdout, "row %d, char %d.\n", row-length+1, j+1);
166         }
167     }
168 }
```

## Solution 2

In this solution we store single rows of the matrix in a dynamically allocated array. This array is visited to search for strings horizontally. For the vertical searches an array of counters stores the number of matches found along each columns.

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 /* 100 + '\n' + '\0' */
```

```
6 #define MAX 102
7 /* function prototypes */
8 void search(char *, char *);
9 void check(char *, int *, char *, int);
10
11 /*
12 * main program
13 */
14 int main(int argc, char *argv[]) {
15     if (argc < 3) {
16         fprintf(stderr, "Error: missing parameter.\n");
17         fprintf(stderr, "Run as: %s <keyword> <text_file>\n", argv[0]);
18         return EXIT_FAILURE;
19     }
20
21     search(argv[2], argv[1]);
22     return EXIT_SUCCESS;
23 }
24
25
26 /*
27 * search the keyword in the whole text file
28 */
29 void search(char *name, char *word) {
30     char *monitor;
31     int *counter, i, row;
32     FILE *fp;
33
34     fp = fopen(name, "r");
35     if (fp == NULL) {
36         fprintf(stderr, "File open error (file=%s).\n", name);
37         exit(EXIT_FAILURE);
38     }
39
40     monitor = (char *)malloc(MAX * sizeof(char));
41     counter = (int *)malloc(MAX * sizeof(int));
42     if (monitor == NULL || counter == NULL) {
43         fprintf(stderr, "Memory allocation error.\n");
44         exit(EXIT_FAILURE);
45     }
46     for (i=0; i<MAX; i++) {
47         counter[i] = 0;
48     }
49
50     row = 0;
51
52     while (fgets(monitor, MAX, fp) != NULL) {
53         row++;
54         check(monitor, counter, word, row);
55     }
56
57     free(monitor);
58     free(counter);
59     fclose(fp);
60 }
61
62 /*
63 * check for the keyword in a text piece
64 */
```

```

65 void check(char *monitor, int *counter, char *word, int row) {
66     int i, length = strlen(word);
67
68     /* search on all columns */
69     for (i=0; i<strlen(monitor); i++) {
70         if (word[counter[i]] != monitor[i]) {
71             counter[i] = 0;
72         } else {
73             counter[i]++;
74             if (counter[i] == length) {
75                 fprintf(stdout, "Word \"%s\" found in ", word);
76                 fprintf(stdout, "row %d, char %d.\n", row-length+1, i+1);
77                 counter[i] = 0;
78             }
79         }
80     }
81 }
```

## 3.18 Political Elections

### Specifications

Write a C program able to supervise the political elections as follows.

Each candidate can run for a party in different districts, and a file stores all candidate information:

- ▷ The first row specifies the number of candidates.
- ▷ All subsequent rows are divided into groups, one group for each candidate.
  - ◊ The first row of each group reports the following pieces of information:  
 partyName candidateName districtNumber  
 where the first two fields are string of maximum 20 characters, and the last one is an integer value.
  - ◊ The following districtNumber lines indicates the districts name (maximum 5 characters) where the candidate runs, e.g., “TO123”, “MI001”, etc.

Notice that, there is no pre-defined order within the file.

Once it has been run, the program must be able to receive the following commands:

- ▷ scrutinizing fileName: the file fileName stores the votes in a specific district, The first line of the file stores the district name. All subsequent lines stores the vote obtained by each candidates:

candidateName numberOfVotes

The program has to read such a file and update the results for each candidate.

- ▷ candidate candidateName: print-out the total number of votes (on all districts) for that candidate.
- ▷ party partyName: print-out the total number of votes (on all districts and for all candidates) for that party.
- ▷ district districtName: print-out the total number of votes in that district, summing up all candidates running in that district.
- ▷ stop: end the program.

**Example 3.43** Let the files "candidates.txt", "mi001.txt" and "na005.txt" have the following content:

candidates.txt

```
3
LooserParty Verdi 2
TO123
NA005
UndecidedParty Rossi 1
RM111
WinnerParty Bianchi 3
NA005
MI001
RM111
```

mi001.txt

MI001
Bianchi 100

na005.txt

NA005
Verdi 200
Bianchi 50

The following is a correct example of execution (the underlined text is introduced by the user):

```
Candidates file: candidates.txt
Command? scrutinize na005.txt
Command? candidate Bianchi
Bianchi obtained 50 votes.
Command? party LooserParty
The LooserParty obtained 200 votes.
Command? scrutinize mi001.txt
Command? candidate Bianchi
Bianchi obtained 150 votes.
Command? district NA005
District NA005 registered 250 votes.
Command? party UndecidedParty
The UndecidedParty obtained 0 votes.
Command? stop
Program ended.
```

## Solution

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 #define L1 5+1
6 #define L2 20+1
7
8 /* structure declaration */
9 typedef struct {
10     char code[L1];
11     int num_vote;
12 } district_t;
13
14 typedef struct {
15     char party[L2];
16     char name[L2];
17     int num_district;
18     district_t *districts;
19 } candidate_t;
```

```

21 /* function prototypes */
22 candidate_t *candidate_read(int *);
23 void district_read(candidate_t *, int);
24 void candidate_write(candidate_t *, int);
25 void party_write(candidate_t *, int);
26 void district_write(candidate_t *, int);
27
28 /*
29  * main program
30 */
31 int main (void) {
32     candidate_t *candidates = NULL;
33     char cmd[L2];
34     int n;
35
36     candidates = candidate_read(&n);
37     do {
38         fprintf(stdout, "Command: ");
39         scanf("%s", cmd);
40         if (strcmp(cmd, "poll") == 0) {
41             district_read(candidates, n);
42         } else if (strcmp(cmd, "candidate") == 0) {
43             candidate_write(candidates, n);
44         } else if (strcmp(cmd, "party") == 0) {
45             party_write(candidates, n);
46         } else if (strcmp(cmd, "district") == 0) {
47             district_write(candidates, n);
48         } else if (strcmp(cmd, "stop") != 0) {
49             fprintf(stdout, "Unknown command.\n");
50         }
51     } while (strcmp(cmd, "stop") != 0);
52
53     for (n--; n>=0; n--) {
54         free(candidates[n].districts);
55     }
56     free(candidates);
57
58     fprintf(stdout, "End of program.\n");
59
60     return EXIT_SUCCESS;
61 }
62
63 /*
64  * load the candidates file contents
65 */
66 candidate_t *candidate_read (int *num_ptr) {
67     candidate_t *candidates;
68     char name[L2];
69     int i, j, nd;
70     FILE *fp;
71
72     fprintf(stdout, "Candidates file: ");
73     scanf("%s", name);
74     fp = fopen(name,"r");
75     if (fp == NULL) {
76         fprintf(stderr, "File open error (file=%s).n", name);
77         exit(EXIT_FAILURE);
78     }
79

```

```

80     fscanf(fp, "%d", num_ptr);
81     candidates = (candidate_t *)malloc(*num_ptr * sizeof(candidate_t));
82     if (candidates == NULL) {
83         fprintf(stderr, "Memory allocation error.\n");
84         exit(EXIT_FAILURE);
85     }
86     for (i=0; i<*num_ptr; i++) {
87         fscanf(fp, "%s %s %d", candidates[i].party, candidates[i].name, &nd);
88         candidates[i].num_district = nd;
89         candidates[i].districts = (district_t *)malloc(nd * sizeof(district_t));
90         if (candidates[i].districts == NULL) {
91             fprintf(stderr, "Memory allocation error.\n");
92             exit(EXIT_FAILURE);
93         }
94         for (j=0; j<nd; j++) {
95             fscanf(fp, "%s", candidates[i].districts[j].code);
96             candidates[i].districts[j].num_vote = 0;
97         }
98     }
99 }
100 fclose(fp);
101
102 return candidates;
103 }
104
105 /*
106 * load a district file contents
107 */
108 void district_read (candidate_t *candidates, int n) {
109     char name[L2], district[L2], candidate[L2];
110     int i, j, stop, num_vote;
111     FILE *fp;
112
113     scanf("%s", name);
114     fp = fopen(name, "r");
115     if (fp == NULL) {
116         fprintf(stderr, "File open error (file=%s).\n", name);
117         exit(EXIT_FAILURE);
118     }
119
120     fscanf(fp, "%s", district);
121     while (fscanf(fp, "%s%d", candidate, &num_vote) != EOF) {
122         for (stop=i=0; i<n && stop==0; i++) {
123             if (strcmp(candidates[i].name, candidate) == 0) {
124                 for (stop=j=0; j<candidates[i].num_district; j++) {
125                     if (strcmp(candidates[i].districts[j].code, district) == 0) {
126                         candidates[i].districts[j].num_vote = num_vote;
127                     }
128                 }
129             }
130         }
131     }
132     fclose(fp);
133 }
134
135 /*
136 * print a candidate result
137 */
138 void candidate_write (candidate_t *candidates, int n) {

```

```

139     int i, j, found=0, num_vote=0;
140     char candidate[L2];
141
142     scanf("%s", candidate);
143     for (i=0; i<n && found==0; i++) {
144         if (strcmp(candidates[i].name, candidate) == 0) {
145             for (j=0; j<candidates[i].num_district; j++) {
146                 num_vote += candidates[i].districts[j].num_vote;
147             }
148             found = 1;
149         }
150     }
151     fprintf(stdout, "Candidate %s obtained %d votes.\n", candidate, num_vote);
152 }
153
154 /*
155 * print a party result
156 */
157 void party_write (candidate_t *candidates, int n) {
158     int i, j, num_vote=0;
159     char party[L2];
160
161     scanf("%s", party);
162     for (i=0; i<n; i++) {
163         if (strcmp(candidates[i].party, party) == 0) {
164             for (j=0; j<candidates[i].num_district; j++) {
165                 num_vote += candidates[i].districts[j].num_vote;
166             }
167         }
168     }
169     fprintf(stdout, "Party %s obtained %d votes.\n", party, num_vote);
170 }
171
172 /*
173 * print a district result
174 */
175 void district_write (candidate_t *candidates, int n) {
176     int i, j, num_vote=0;
177     char district[L1];
178
179     scanf("%s", district);
180     for (i=0; i<n; i++) {
181         for (j=0; j<candidates[i].num_district; j++) {
182             if (strcmp(candidates[i].districts[j].code, district) == 0) {
183                 num_vote += candidates[i].districts[j].num_vote;
184             }
185         }
186     }
187     fprintf(stdout, "District %s has %d registered votes.\n", district, num_vote);
188
189     return;
190 }
```

## 3.19 Library

### Specifications

A small library uses three files to supervise its activity:

- ▷ “users.txt” stores information on all users. Each row of this file stores the user

name, his/her phone number, and a unique library identifier.

- ▷ “catalogue.txt” stores the list of available books. Each row of this file stores the writer name, the title name, the book position within the library, a book unique identifier, and the number of copies available.
- ▷ “borrow.txt” stores a list of all books borrowed from the library. Each row of this file stores the identifier of the user who borrowed the book, and the date of the operation (with format dd/mm/yyyy).

All data, but the number of available copies, are strings (without spaces) of maximum 100 characters.

Write a program which, once read the three files, is able to generate the following two files:

- ▷ “user\_borrows.txt”: It stores the same information included in file “users.txt” but with the list of borrowed books for each users. For each user line, the last field reports the total number  $n$  of books in possession of that user. After that, the following  $n$  lines store the identifier of those books, and the respective date.
- ▷ “book\_borrow.txt”: It stores the same information included in file “catalogue.txt” but with the list of borrowed copies for each book. For each book line, the last field reports the total number  $n$  of books borrowed. After that, the following  $n$  lines store the user identifiers who have a copy of the book.

Assume all file contents as correct.

**Example 3.44** Let the following be the three input files:

users.txt

Potter 011001 Wizard HP
Granger 011002 Witch HG
Weasley 011003 Wizard RW

borrow.txt

HG JKR5 10/09/2009
HG JKR6 05/09/2009
RW JKR1 15/08/2009
HP JKR4 01/09/2009
HG JKR7 15/09/2009
HP JKR5 20/09/2009

catalogue.txt

J.K.Rowling ThePhilosopher'sStone R1S2B1 JKR1 3
J.K.Rowling TheChamberOfSecrets R1S2B2 JKR2 3
J.K.Rowling ThePrisonerOfAzkaban R1S2B3 JKR3 2
J.K.Rowling TheGobletOfFire R1S2B4 JKR4 3
J.K.Rowling TheOrderOfThePhoenix R1S2B5 JKR5 5
J.K.Rowling TheHalf-BloodPrince R1S2B6 JKR6 2
J.K.Rowling TheDeathlyHallows R1S2B7 JKR7 5

The program has to generate the following two files.

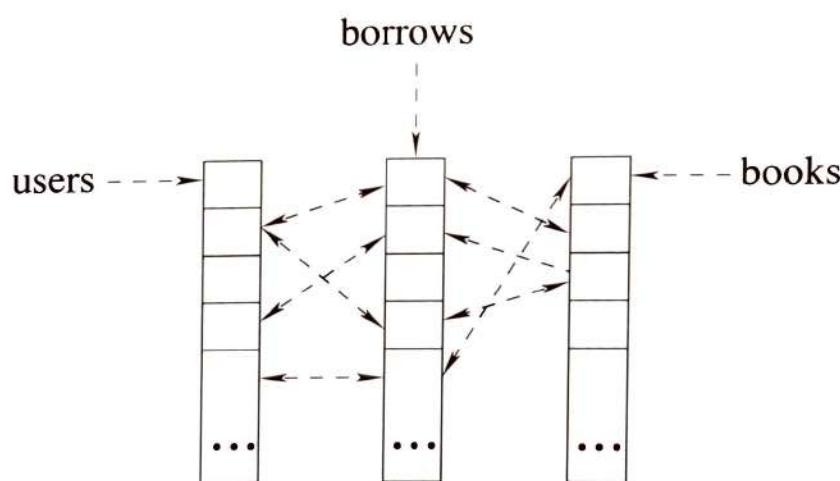
```
users_borrows.txt
Potter 011001 Wizard HP 2
JKR4 01/09/2009
JKR5 20/09/2009
Granger 011002 Witch HG 3
JKR5 10/09/2009
JKR6 05/09/2009
JKR7 15/09/2009
Weasley 011003 Wizard RW 1
JKR1 15/08/2009
```

```
book_borrows.txt
J.K.Rowling ThePhilosopher'sStone R1S2B1 JKR1 3 1
RW 15/08/2009
J.K.Rowling TheChamberOfSecrets R1S2B2 JKR2 3 0
J.K.Rowling ThePrisonerOfAzkaban R1S2B3 JKR3 2 0
J.K.Rowling TheGobletOfFire R1S2B4 JKR4 3 1
HP 01/09/2009
J.K.Rowling TheOrderOfThePhoenix R1S2B5 JKR5 5 2
HG 10/09/2009
HP 20/09/2009
J.K.Rowling TheHalf-BloodPrince R1S2B6 JKR6 2 1
HG 05/09/2009
J.K.Rowling TheDeathlyHallows R1S2B7 JKR7 5 1
HG 15/09/2009
```

## Solution 1

The problem is somehow symmetric between books and users: All operations done on users have to be performed also on books, with borrowed books connecting those two data structures.

In this solution each file is stored in a dynamic array. Figure 3.9 sketches the overall data structure:



**Figure 3.9** Main arrays and their relationship. Arrows indicate all loan relationships.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define MAX_LINE 1000
6 #define MAX_WORD 100+1
7
8 /* structure declaration */
9 typedef struct user_s {
10     char *name;
11     char *telephone;
12     char *job;
```

```

13  char *code;
14  int *borrows;
15  int num_borrow;
16 } user_t;
17
18 typedef struct book_s {
19   char *author;
20   char *title;
21   char *position;
22   char *code;
23   int num_copies;
24   int *borrows;
25   int num_borrow;
26 } book_t;
27
28 typedef struct borrow_s {
29   int user;
30   int book;
31   char *date;
32 } borrow_t;
33
34 /* function prototypes */
35 user_t *load_users(int *);
36 book_t *load_books(int *);
37 borrow_t *load_borrows(user_t *, int, book_t *, int, int *);
38 int find_user(user_t *, int, char *);
39 int find_book(book_t *, int, char *);
40 void print_users(user_t *, int, borrow_t *, book_t *);
41 void print_books(book_t *, int, borrow_t *, user_t *);
42 void quit_memory(user_t *, int, book_t *, int, borrow_t *, int);
43
44 /*
45  * main program
46  */
47 int main(void) {
48   int num_user, num_book, num_borrow;
49   user_t *users;
50   book_t *books;
51   borrow_t *borrows;
52
53   /* load the three input files */
54   users = load_users(&num_user);
55   books = load_books(&num_book);
56   borrows = load_borrows(users, num_user, books, num_book, &num_borrow);
57
58   /* print the two output files */
59   print_users(users, num_user, borrows, books);
60   print_books(books, num_book, borrows, users);
61   quit_memory(users, num_user, books, num_book, borrows, num_borrow);
62
63   return EXIT_SUCCESS;
64 }
65
66 /*
67  * load the user file contents
68  */
69 user_t *load_users(int *num_ptr) {
70   char line[MAX_LINE], name[MAX_WORD], phone[MAX_WORD],
71   job[MAX_WORD], code[MAX_WORD];

```

```
72 int i, count=0;
73 user_t *users;
74 FILE *fp;
75
76 /* count the file rows */
77 fp = fopen("users.txt", "r");
78 while (fgets(line, MAX_LINE, fp) != NULL) {
79     count++;
80 }
81 fclose(fp);
82
83 /* allocate the user array and save the file contents */
84 users = (user_t *)malloc(count*sizeof(user_t));
85 fp = fopen("users.txt", "r");
86 for (i=0; i<count; i++) {
87     fscanf(fp, "%s %s %s %s", name, phone, job, code);
88     users[i].name = strdup(name);
89     users[i].telephone = strdup(phone);
90     users[i].job = strdup(job);
91     users[i].code = strdup(code);
92     users[i].borrows = NULL;
93     users[i].num_borrow = 0;
94 }
95 fclose(fp);
96
97 *num_ptr = count;
98 return users;
99 }
100
101 /*
102 * load the book file contents
103 */
104 book_t *load_books(int *num_ptr) {
105     char line[MAX_LINE], author[MAX_WORD], title[MAX_WORD];
106     char position[MAX_WORD], code[MAX_WORD];
107     int i, num, count=0;
108     book_t *books;
109     FILE *fp;
110
111 /* count the file rows */
112 fp = fopen("books.txt", "r");
113 while (fgets(line, MAX_LINE, fp) != NULL) {
114     count++;
115 }
116 fclose(fp);
117
118 /* allocate the book array and save the file contents */
119 books = (book_t *)malloc(count*sizeof(book_t));
120 fp = fopen("books.txt", "r");
121 for (i=0; i<count; i++) {
122     fscanf(fp, "%s %s %s %s %d", author, title, position, code, &num);
123     books[i].author = strdup(author);
124     books[i].title = strdup(title);
125     books[i].position = strdup(position);
126     books[i].code = strdup(code);
127     books[i].num_copies = num;
128     books[i].borrows = NULL;
129     books[i].num_borrow = 0;
130 }
```

```

131     fclose(fp);
132     *num_ptr = count;
133
134     return books;
135 }
136
137 /*
138  * load the borrow file
139 */
140 borrow_t *load_borrows (
141     user_t *users, int num_user,
142     book_t *books, int num_book, int *num_ptr
143 ) {
144     char line[MAX_LINE], user_code[MAX_WORD], book_code[MAX_WORD], date[MAX_WORD];
145     int i, j, count=0;
146     borrow_t *borrows;
147     FILE *fp;
148
149     /* count the file rows */
150     fp = fopen("borrows.txt", "r");
151     while (fgets(line, MAX_LINE, fp) != NULL) {
152         count++;
153     }
154     fclose(fp);
155
156     /* allocate the borrow array and save the file contents */
157     borrows = (borrow_t *)malloc(count*sizeof(borrow_t));
158     fp = fopen("borrows.txt", "r");
159     for (i=0; i<count; i++) {
160         fscanf(fp, "%s %s %s", user_code, book_code, date);
161         borrows[i].date = strdup(date);
162
163         borrows[i].user = find_user(users, num_user, user_code);
164         users[borrows[i].user].num_borrow++;
165
166         borrows[i].book = find_book(books, num_book, book_code);
167         books[borrows[i].book].num_borrow++;
168     }
169     fclose(fp);
170
171     /* arrange the "pointers" from books and users to borrows */
172     for (i=0; i<num_user; i++) {
173         if (users[i].num_borrow > 0) {
174             users[i].borrows = (int *)malloc(users[i].num_borrow*sizeof(int));
175             users[i].num_borrow = 0;
176         }
177     }
178
179     for (i=0; i<num_book; i++) {
180         if (books[i].num_borrow > 0) {
181             books[i].borrows = (int *)malloc(books[i].num_borrow*sizeof(int));
182             books[i].num_borrow = 0;
183         }
184     }
185
186     for (i=0; i<count; i++) {
187         j = borrows[i].user;
188         users[j].borrows[users[j].num_borrow++] = i;
189     }

```

```
190     j = borrows[i].book;
191     books[j].borrows[books[j].num_borrow++] = i;
192 }
193
194 *num_ptr = count;
195 return borrows;
196 }
197
198 /*
199  * find a user given his code
200 */
201 int find_user (user_t *users, int dim, char *key) {
202     int i;
203
204     for (i=0; i<dim; i++) {
205         if (strcmp(key, users[i].code) == 0) {
206             return i;
207         }
208     }
209
210     return -1;
211 }
212
213 /*
214  * find a book given its code
215 */
216 int find_book(book_t *books, int dim, char *key) {
217     int i;
218
219     for (i=0; i<dim; i++) {
220         if (strcmp(key, books[i].code) == 0) {
221             return i;
222         }
223     }
224
225     return -1;
226 }
227
228 /*
229  * print the users-borrows file
230 */
231 void print_users (
232     user_t *users, int num_user, borrow_t *borrows, book_t *books
233 ) {
234     int i, j, k, b;
235     FILE *fp;
236
237     fp = fopen("users_borrows.txt", "w");
238     for (i=0; i<num_user; i++) {
239         fprintf(fp, "%s %s ", users[i].name, users[i].telephone);
240         fprintf(fp, "%s %s ", users[i].job, users[i].code);
241         fprintf(fp, "%d\n", users[i].num_borrow);
242         for (j=0; j<users[i].num_borrow; j++) {
243             k = users[i].borrows[j]; /* borrow index */
244             b = borrows[k].book; /* book index */
245             fprintf(fp, "%s %s\n", books[b].code, borrows[k].date);
246         }
247     }
248     fclose(fp);
```

```
249 }
250
251 /* print the books-borrows file
252 */
253 void print_books(
254     book_t *books, int num_book, borrow_t *borrows, user_t *users
255 ) {
256     int i, j, k, u;
257     FILE *fp;
258
259     fp = fopen("books_borrows.txt", "w");
260     for (i=0; i<num_book; i++) {
261         fprintf(fp, "%s %s ", books[i].author, books[i].title);
262         fprintf(fp, "%s %s ", books[i].position, books[i].code);
263         fprintf(fp, "%d %d\n", books[i].num_copies, books[i].num_borrow);
264         for (j=0; j<books[i].num_borrow; j++) {
265             k = books[i].borrows[j]; /* borrow index */
266             u = borrows[k].user; /* user index */
267             fprintf(fp, "%s %s\n", users[u].code, borrows[k].date);
268         }
269     }
270 }
271 fclose(fp);
272 }
273
274 /*
275  * free all allocated memory
276 */
277 void quit_memory(
278     user_t *users, int num_user, book_t *books, int num_book,
279     borrow_t *borrows, int num_borrow
280 ) {
281     int i;
282
283     for (i=0; i<num_user; i++) {
284         free(users[i].name);
285         free(users[i].telephone);
286         free(users[i].job);
287         free(users[i].code);
288         free(users[i].borrows);
289     }
290     free(users);
291
292     for (i=0; i<num_book; i++) {
293         free(books[i].author);
294         free(books[i].title);
295         free(books[i].position);
296         free(books[i].code);
297         free(books[i].borrows);
298     }
299     free(books);
300
301     for (i=0; i<num_borrow; i++) {
302         free(borrows[i].date);
303     }
304     free(borrows);
305 }
```

## Solution 2

This solution solves the problem in a more direct way, as only file `borrow.txt` is stored into a proper data structure. The other two files are read on-the-fly and the output file generated at the same time.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define MAX_LINE 1000
6 #define MAX_WORD 100+1
7
8 /* structure declaration */
9 typedef struct {
10     char *user;
11     char *book;
12     char *date;
13 } borrow_t;
14
15 /* function prototypes */
16 borrow_t *load_borrows(int *);
17 void print_users(borrow_t *, int);
18 void print_books(borrow_t *, int);
19 void quit_memory(borrow_t *, int);
20
21 /*
22  * main program
23  */
24 int main(void) {
25     borrow_t *borrows;
26     int num_borrows;
27
28     borrows = load_borrows(&num_borrows);
29     print_users(borrows, num_borrows);
30     print_books(borrows, num_borrows);
31
32     quit_memory(borrows, num_borrows);
33     return EXIT_SUCCESS;
34 }
35
36 /*
37  * load the borrow file contents
38  */
39 borrow_t *load_borrows(int *num_ptr) {
40     char line[MAX_LINE], user[MAX_WORD], book[MAX_WORD], date[MAX_WORD];
41     borrow_t *borrows;
42     int i, count=0;
43     FILE *fp;
44
45     /* count the file rows */
46     fp = fopen("borrows.txt", "r");
47     if (fp == NULL) {
48         fprintf(stderr, "File open error.\n");
49         exit(EXIT_FAILURE);
50     }
51     while (fgets(line, MAX_LINE, fp) != NULL) {
52         count++;
53     }
54     fclose(fp);

```

```

55  /* allocate the borrow array and save the file contents */
56  borrows = (borrow_t *)malloc(count*sizeof(borrow_t));
57  if (borrows == NULL) {
58      fprintf(stderr, "Memory allocation error.\n");
59      exit(EXIT_FAILURE);
60  }
61  fp = fopen("borrows.txt", "r");
62  if (fp == NULL) {
63      fprintf(stderr, "File open error.\n");
64      exit(EXIT_FAILURE);
65  }
66  for (i=0; i<count; i++) {
67      fscanf(fp, "%s %s %s", user, book, date);
68      borrows[i].user = strdup(user);
69      borrows[i].book = strdup(book);
70      borrows[i].date = strdup(date);
71      if (borrows[i].user==NULL || borrows[i].book==NULL || borrows[i].date==NULL) {
72          fprintf(stderr, "Memory allocation error.\n");
73          exit(EXIT_FAILURE);
74      }
75  }
76  fclose(fp);
77
78  *num_ptr = count;
79  return borrows;
80 }
81 }
82
83 /*
84  * print the users-borrows file
85  */
86 void print_users(borrow_t *borrows, int num_borrow) {
87     char name[MAX_WORD], phone[MAX_WORD], job[MAX_WORD], code[MAX_WORD];
88     FILE *fin, *fout;
89     int i, num_book;
90
91     fin = fopen("users.txt", "r");
92     fout = fopen("users_borrows.txt", "w");
93     if (fin==NULL || fout==NULL) {
94         fprintf(stderr, "File open error.\n");
95         exit(EXIT_FAILURE);
96     }
97
98     /* read a user and immediately output the related info */
99     while (fscanf(fin, "%s %s %s %s", name, phone, job, code) != EOF) {
100         fprintf(fout, "%s %s %s %s ", name, phone, job, code);
101         fprint(fout, "%s %s %s %s ", name, phone, job, code);
102
103         num_book = 0;
104         for (i=0; i<num_borrow; i++) {
105             if (strcmp(code, borrows[i].user) == 0) {
106                 num_book++;
107             }
108         }
109         fprintf(fout, "%d\n", num_book);
110         for (i=0; i<num_borrow; i++) {
111             if (strcmp(code, borrows[i].user) == 0) {
112                 fprintf(fout, "%s %s\n", borrows[i].book, borrows[i].date);
113             }
114         }
115     }
116 }
```

```
114     }
115
116     fclose(fin);
117     fclose(fout);
118 }
119
120 /*
121 * print the books-borrows file
122 */
123 void print_books(borrow_t *borrows, int num_borrow) {
124     char author[MAX_WORD], title[MAX_WORD], position[MAX_WORD];
125     int i, num, num_user;
126     FILE *fin, *fout;
127
128     fin = fopen("books.txt", "r");
129     fout = fopen("books-borrows.txt", "w");
130     if (fin==NULL || fout==NULL) {
131         fprintf(stderr, "File open error.\n");
132         exit(EXIT_FAILURE);
133     }
134
135 /* read a book and immediately output the related info */
136 while (
137     fscanf(fin, "%s %s %s %s %d", author, title, position, code, &num) != EOF
138     ) {
139     fprintf(fout, "%s %s %s %s %d ", author, title, position, code, num);
140
141     num_user = 0;
142     for (i=0; i<num_borrow; i++) {
143         if (strcmp(code, borrows[i].book) == 0) {
144             num_user++;
145         }
146     }
147     fprintf(fout, "%d\n", num_user);
148     for (i=0; i<num_borrow; i++) {
149         if (strcmp(code, borrows[i].book) == 0) {
150             fprintf(fout, "%s %s\n", borrows[i].user, borrows[i].date);
151         }
152     }
153 }
154
155     fclose(fin);
156     fclose(fout);
157 }
158
159 /*
160 * free all allocated memory
161 */
162 void quit_memory(borrow_t *borrows, int num_borrows) {
163     int i;
164
165     for (i=0; i<num_borrows; i++) {
166         free(borrows[i].user);
167         free(borrows[i].book);
168         free(borrows[i].date);
169     }
170     free(borrows);
171 }
```

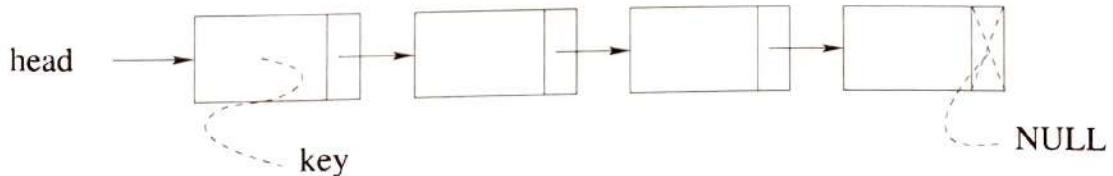
# Chapter 4

## Lists

### 4.1 General concepts

A linked list is a linear collection of self-referential structures, called elements or nodes, connected by pointer links. This structure motivates the term “linked” list.

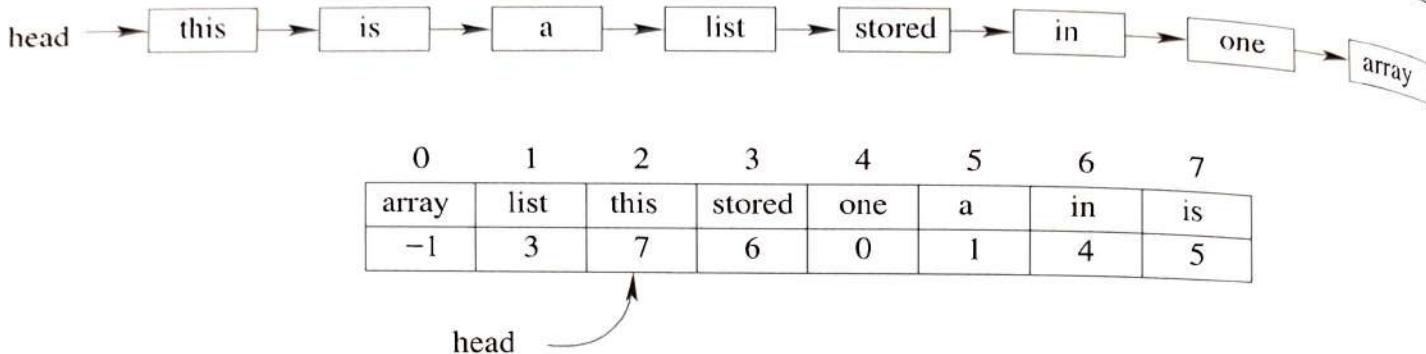
A linked list (see Figure 4.1 for a graphical representation) is accessed via a pointer (head in the picture) to the first node of the list. Subsequent nodes are accessed via the link pointer member stored in each node. By convention, the link pointer in the last node of a list is set to NULL to explicitly mark the end of the list. Nodes are manipulated in a linked list dynamically, i.e., each node is created only when necessary and erased when no longer needed. A node can contain data of any type including other structures. The desired data is stored on fields of the various list nodes. Among all data fields a specific field acts as unique identifier or key (key) of the node. Insertion (and extractions) of new nodes in (from) the list are possible through pointers manipulation.



**Figure 4.1** A simple list data structure.

Lists of data can be stored in arrays as shown in Figure 4.2. Anyhow linked lists provide several advantages.

The size of an array is define at compile (when static) or run (when dynamic) time, however, it cannot be easily altered. Arrays can become full. An array can be declared to contain more elements than the number of data items expected, but this can waste memory. Array realloc may have a linear cost, thus a linear number of realloc (one for each new element inserted into the array) potentially has a quadratic cost. A linked list is appropriate when the number of data elements to be represented in the data structure is unpredictable. Linked lists are dynamic, so the length of a list can increase



**Figure 4.2** The same list of ordered string stored in linked list and in an array acting as a list. Each element of the array (a C structure) includes the string and the index (i.e., the pointer) to the next element in the list.

or decrease at execution time as necessary. Linked lists can provide better memory utilization in these situations even considering the space “wasted” to store the required pointer links. Linked lists become full only when the system has insufficient memory to satisfy dynamic storage allocation requests.

The elements of an array are stored contiguously in memory. This allows immediate access (or *direct* access) to any array element. The address of any element can be computed directly based on its position relative to the beginning of the array. In other words, visiting element in position 100 is as expensive as visiting element in position 0. Linked lists do not afford such immediate access to their elements as linked-list nodes are normally not stored contiguously in memory. To visit element in position 100, we have to visit element in position 0 and from it reach element in position 1, and from it reach element in position 2, till element in position 100 is reached. Then the only possible access scheme is sequential. Logically, however, the nodes of a linked list appear to be contiguous.

Linked lists can be maintained in sorted order by inserting each new element at the proper point in the list or they can respect personalized strategy to insert and extract data. Complex lists, such as circular lists, or lists with more than one pointer per elements, can also be created and they will be analyzed in the sequel.

## 4.2 Simple List

Linked lists are defined as primitive elements in some language but not in C. In this section we will analyze how to implement all main operations on a simple linked list like to one represented in Figure 4.1.

### 4.2.1 C Representation

A list element is usually defined through a C structure including several data fields, and an auto-referencing pointer. This pointer makes the structure *recursive*, i.e., partially defined in terms of itself or of its type<sup>1</sup>. The following lines define a generic recursive C data structure:

<sup>1</sup> Please, notice that recursive structures, and recursion, will be fully analyzed in Chapter 5.

```
typedef struct list_s list_t;
struct list_s {
    /* Data Fields */
    ...
    /* Auto-Referencing Pointer */
    list_t *next;
};
```

Among all possible data fields we will always consider at least an integer value or a string (named key or value) acting as unique identifier for the data fields, such as:

```
typedef struct list_s list_t;
struct list_s {
    int key;
    /* Data Fields */
    ...
    /* Auto-Referencing Pointer */
    list_t *next;
};
```

The same definition can be more compactly written as:

```
typedef struct list_s {
    int key;
    ...
    struct list_s *next;
} list_t;
```

Each element has to be separately allocated with a function such as the following one.

```
list_t *new_element () {
    list_t *e_ptr;

    e_ptr = (list_t *) malloc (sizeof (list_t));

    if (e_ptr==NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (FAILURE);
    }

    return (e_ptr);
}
```

The new\_element function can be called as follows:

```
list_t *p;
...
p = new_element();
```

## 4.2.2 Visit

From this section on, we will consider several pointers to elements of the list. These pointers are supposed to be defined as:

```
list_t *head, *tail, *p, ...;
```

even if we do not recall this definition any more in the text.

Given a pointer p previously defined, the following lines traverse a list, i.e., visit all nodes belonging to the list. Starting from the pointer to the beginning of the list, the code determines whether the list is not empty. If so, it visits the data in the list while p is not NULL. If the link in the last node of the list is not NULL, the code will try to

print past the end of the list, and an error will occur. The visit algorithm is identical for many different types of lists, such as stacks and queues.

```
p = head;
while (p != NULL) {
    /* Visit Element p (p->key field) */
    ...
    p = p->next;
}
```

### 4.2.3 Search

Let us suppose we are looking for a node containing a key equal to `value` within the list. We can proceed as follows.

```
p = head;
while (p!=NULL) {
    if (value==p->key) {
        /* Found Element value */
        ...
    } else {
        p = p->next;
    }
}
```

In this case, once the element has been found and managed, it is necessary to get out from the main `while` iteration. This can be done using a `break` instruction or in a more structured, clean, way, such as:

```
goOn = 1;
p = head;
while (p!=NULL && goOn==1) {
    if (value==p->key) {
        /* Found Element value */
        ...
        goOn = 0;
    } else {
        p = p->next;
    }
}
```

where `goOn` is a variable used as a flag to leave the iteration.

The previous cycle can also be rewritten as:

```
p = head;
while ((p!=NULL) && (p->key!=value)) {
    p = p->next;
}
if (p!=NULL) {
    /* Here the value has been found */
    ...
}
```

Notice that in this case the order of the two conditions

`(p!=NULL) && (p->key!=value)`

is important. In fact, if `(p==NULL)`, we cannot check whether `(p->key!=value)` because `p->key` does not exist. Then, using the reverse order

`(p->key!=value) && (p!=NULL)`

would result in a buggy piece of code.

**Example 4.1** Let us suppose to define the following list element:

```
typedef struct student list_t;
struct student {
    int id;
    char name[N];
    list_t *next;
};
```

The following function `search` search in a simple linked list the student with a specific `id`, and it returns the pointer to its element. A value equal to `NULL` is return in case the student does not exist.

```
list_t *search (list_t *head, int id) {
    list_t *p = head;

    while (p!=NULL && p->id!=id) {
        p = p->next;
    }

    return p;
}
```

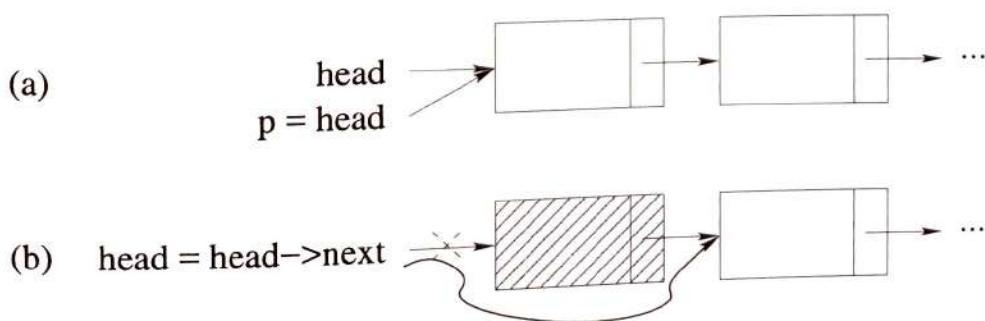
Notice that if the list were ordered the search process would stop once passed the element with an `id` field larger than the one searched, then somehow before the end of the list. Then in an ordered list the search would somehow be more efficient.

#### 4.2.4 Extraction

Extractions can be performed in three different positions within the list: On the head (on the first element), in the middle (on a specific element), and on the tail (as the last element). We will analyze those three cases in the following sub-sections.

##### **Extraction from the List Head**

Figure 4.3 show the case in which the first element is the one to be extracted from the list.



**Figure 4.3** Extraction of the first element.

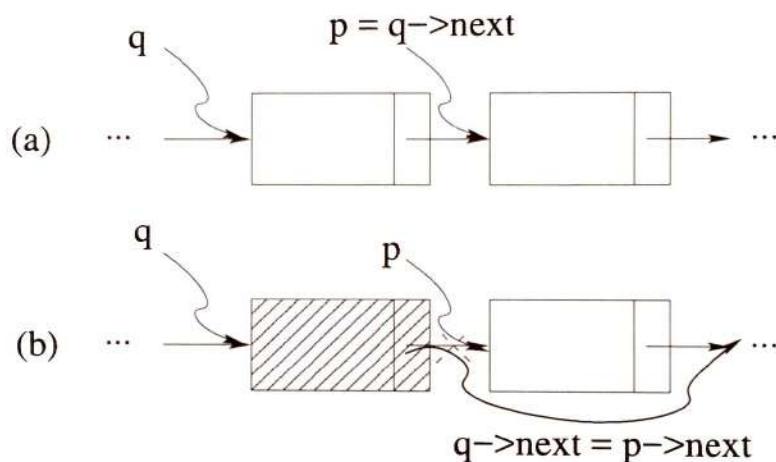
This operation is implemented by the following two lines (corresponding to Figure 4.3(a) and (b), respectively).

```
p = head;
head = head->next;
```

In the code, the head pointer is moved forward one element, and the “old” head pointer (i.e.,  $p$ ) is used to grab the extracted element. This can be used and/or deleted as desired. Obviously the list must contain at least one element, otherwise we try to access pointer  $\text{head} \rightarrow \text{next}$  which does not exist. If there is just one element the final head pointer will be equal to NULL.

### **Extraction from an Intermediate Position**

The extraction of a given element is possible only if we have access to the element placed before the one we want to extract. In fact, to extract an element we need its pointer, that it is stored in the element placed in the list before it. Figure 4.4 shows the case in which it is the element following the one referenced by  $q$  that has to be extracted.



**Figure 4.4** Extraction of the element following the one referenced by  $q$ .

The code is the following one:

```
p = q->next;
q->next = p->next;
```

Element  $q$  is made to point to the element following the one extracted, and  $p$  is used to reference the extracted element. As before element  $p$  can be used, later re-inserted into the same or another list, or deleted (with all its fields) as desired.

### **Extraction from the List Tail**

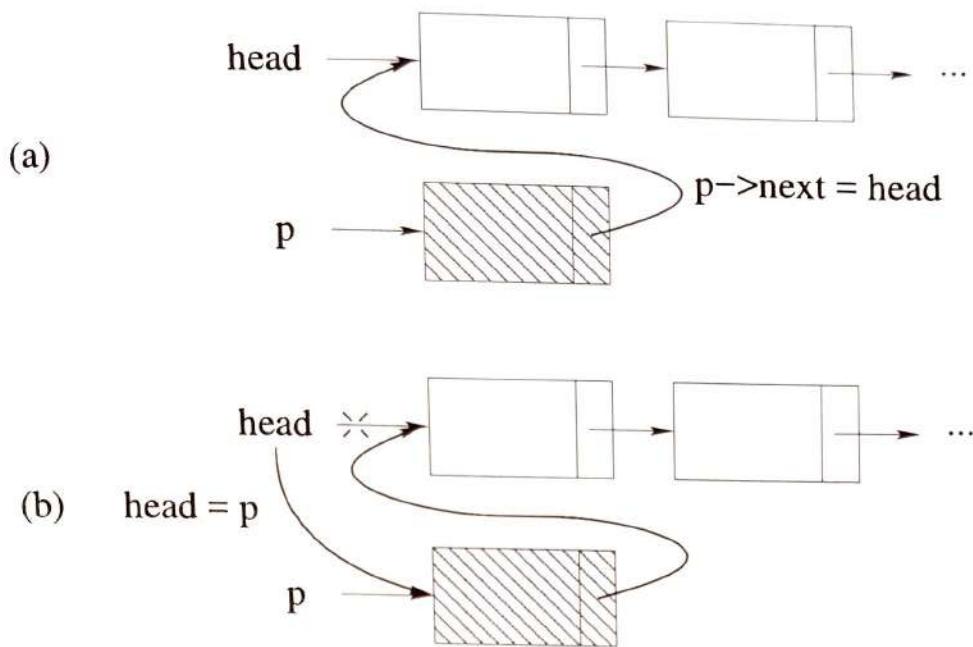
The code presented in the previous sub-section can also be used to extract an element from the list tail, whenever  $q$  points to the element before the last one.

## **4.2.5 Insertion**

As for extractions, also insertions can be performed: On the head (as a “new” first element), in the middle (after a specific pre-existing element), and on the tail (as a “new” last element). We will analyze those three cases in the following sub-sections.

### **Insertion on the List Head**

A first possibility to insert a new element into the list consists in inserting it as a first element of the list. This is also the easiest and more efficient insertion strategy. Figure 4.5 shows this case.



**Figure 4.5** Head insertion.

Once the element referred by *p* has been created (we will cope with this problem in the future) two instructions are necessary to implement this sort of insertion:

```
p->next = head;
head = p;
```

corresponding to Figure 4.5(a) and (b), respectively.

### **Insertion in an Intermediate Position**

Similarly to the extraction case, to insert a new element *before* a given element *q*, it is necessary to access the pointer field within the element coming before *q*. Then, to deal with this case it is necessary some further manipulation, which we will analyze in Section 4.4.

As a consequence, Figure 4.6 shows the standard case in which the new element *p* is placed *after* (not before) an existing element *q*. Notice that the *q* element has to be somehow located within the list. Two lines of code (corresponding to Figure 4.6(a) and (b), respectively) are sufficient to perform the operation:

```
p->next = q->next;
q->next = p;
```

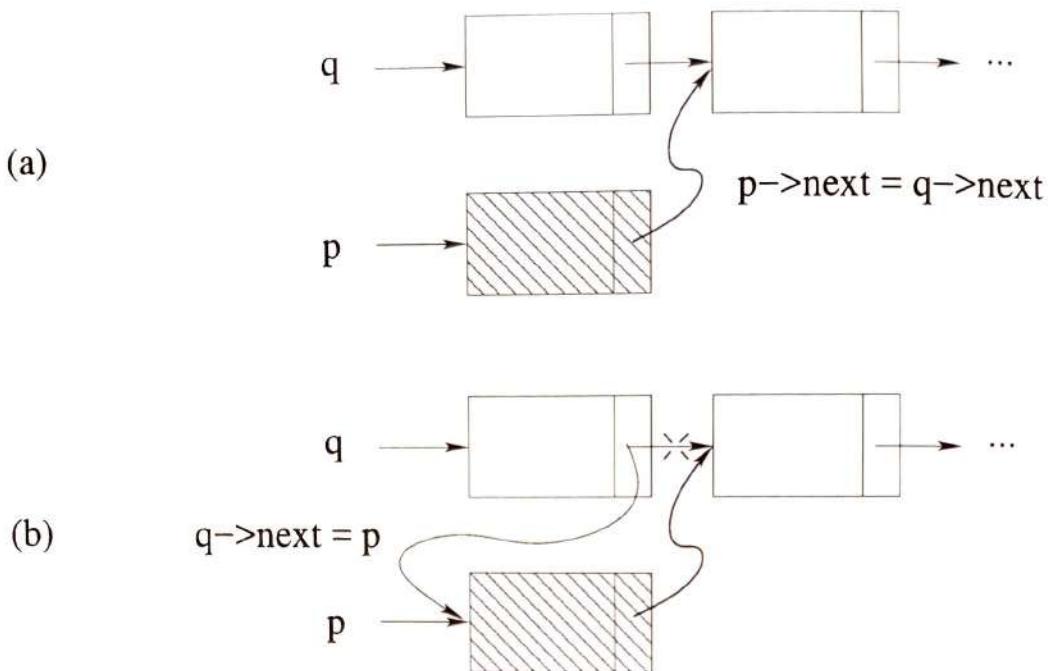
### **Insertion on the List Tail**

If it is necessary to insert an element into the list tail, i.e., as a last element, it is possible to reuse the previous code after making *q* referring to the last element of the list. This requires a visit as illustrated in Section 4.2.2.

### **4.2.6 Free**

Every list has to be freed, sooner or later. The following piece of code:

```
p = head;
```



**Figure 4.6** Insertion after a specific element.

```
while (p != NULL) {
    free(p);
    p = p->next;
}
head = NULL;
```

is actually buggy, as instruction *p* = *p*->next make an access to the next field of a freed element. Moreover, notice that the *p* pointer can be avoided and *head* can be used directly.

The following piece of code fixes the previous problem:

```
while (head != NULL) {
    p = head;
    head = head->next;
    /* ... */
    free (p);
}
```

### 4.3 Lists with Sentinel and Dummy Elements

Several operations on lists can be simplified using the so called *sentinels*. A sentinel (also called signal value, or dummy value, or flag value) is often used to indicate the “end” or the “beginning” of the list. There are at least three type of extensions using sentinels, as sentinels can be used on the head of the list, on the tail, or on both the head and tail. In this book do not introduce these strategies in a complete way, because we consider them a little bit outdated. Anyhow, we give some hints on the topic in the following paragraphs.

In Section 4.2.3 we showed that the piece of code:

```
p = head;
while ((p!=NULL) && (p->key!=value)) {
```

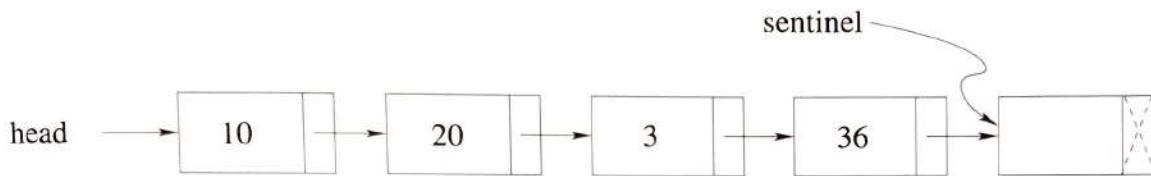
```

    p = p->next;
}
if (p!=NULL) {
/* Here the value has been found */
...
}

```

can be used to search for the element storing the key value. Nevertheless, the while iteration is controlled by two conditions, as p does not have to be NULL, and it does not have to refer to the node storing value. This is somehow inconvenient, and make the code unclean and slower.

Let us suppose to have a list with one final extra element, called `sentinel` as represented in Figure 4.7.



**Figure 4.7** List with a terminal sentinel element.

To perform a search, we can first copy the element we are looking for into the sentinel element. This implies that we will find it, at least into the last element. Then the while condition can be simplified as illustrated by the following code:

```

sentinel->key = value;
p = head;
while (value!=p->key) {
    p = p->next;
}
if (p!=sentinel) {
/* Here the value has been found */
...
} else {
/* Here the value has not been found
(its only in the sentinel element) */
...
}

```

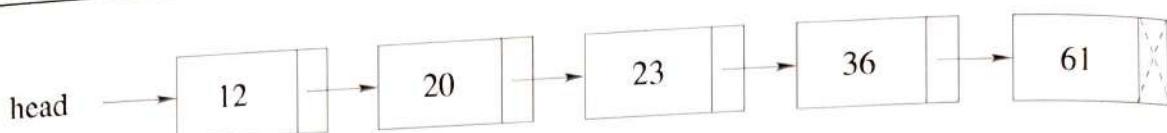
Similar considerations can be done for other operations, but again we do not report them for the sake of space.

## 4.4 Ordered List

Figure 4.8 shows a list ordered by increasing values of its integer key.

With a sorted list, a search can be terminated unsuccessfully when a record with a key larger than the search key is found. Thus only about half the records (not all) need to be examined for an unsuccessful search. The sorted order is easy to maintain because a new record can simply be inserted into the list at the point at which the unsuccessful search terminates.

As usual with linked lists, a dummy header node `head` and a sentinel tail node allow the code to be substantially simpler than without them.



**Figure 4.8** Ordered list of integer keys. Keys are ordered in ascending order.

#### 4.4.1 Search

The algorithm is similar to the one analyzed in Section 4.2.3 but, if the key is not found the search can be stopped as soon as the value becomes larger than the key.

```

p = head;
while (p!=NULL && value>p->key) {
    p = p->next;
}
if (p!=NULL && value==p->key) {
    /* Element value found */
    ...
}
  
```

Notice that the condition `p !=NULL` in the `if` statement can be avoided when one is sure to find the value into the list.

#### 4.4.2 Extraction

To extract (or delete) an element from a list, we have to individuate it, and then we have to remove it from the list. If we suppose to search the element using the same piece of code introduce if Section 4.4.1, we can write:

```

p = head;
while (p!=NULL && value>p->key) {
    p = p->next;
}
if (p!=NULL && value==p->key) {
    /* Element p should be extracted or removed */
    ...
}
  
```

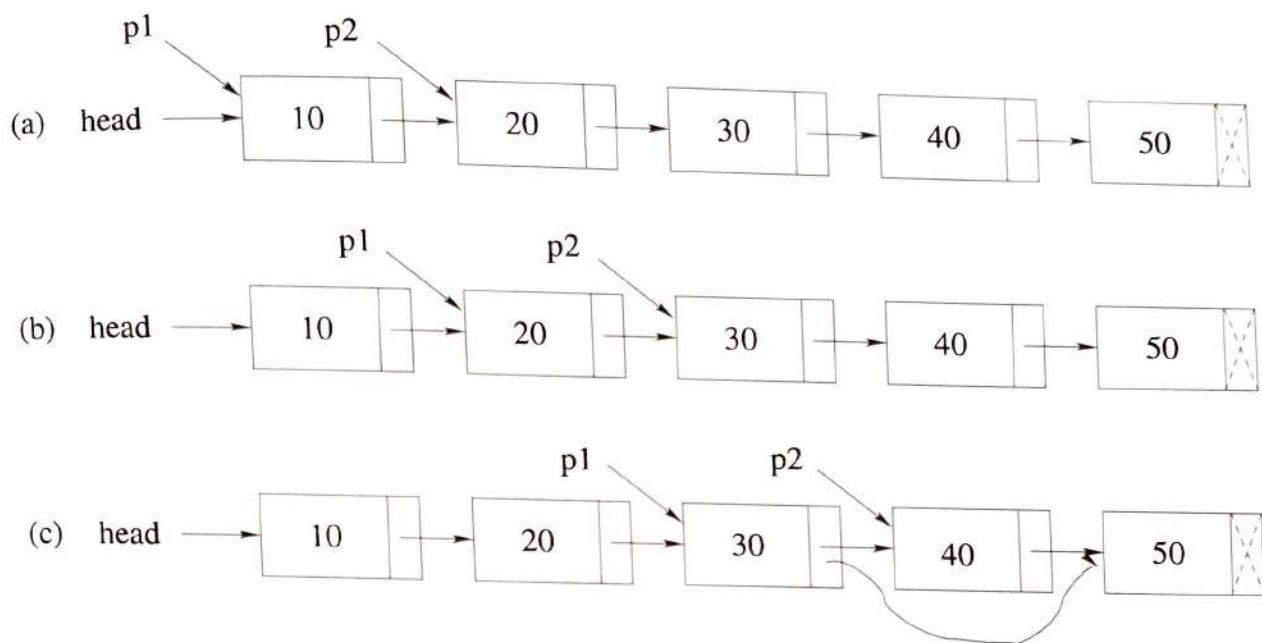
Unfortunately `p` is referencing the element that has to be removed, but we have to observe that an element cannot be deleted if we do not have access to the element before it, i.e., the one pointing to it. The previous code is then unsuited to remove an element from a list. To solve the problem, we must access the element coming before a given element. To do that, there are at least three possible solutions:

- ▷ Use two pointers to individuate two consecutive elements and move them along the list in a synchronized way.
- ▷ Use a pointer to an element just to get its pointer, i.e., using a look-ahead of one element along the list. In this case the idea is to reach the element referenced by the pointer stored into the element referenced by the owned pointer.
- ▷ Use the “direct” pointer to a pointer field, i.e., somehow manipulating the pointers within the list by reference.

We are not going to describe the third technique, as, at this point, it is unnecessarily complex. We will analyze the first two strategy in the following two sub-sections.

### Extraction using two Explicit Pointers

Figure 4.9 illustrates the first technique, somehow the easiest to implement, and the more commonly used.



**Figure 4.9** Extracting the element with key equal to 40 in an ordered list. (a) Initial pointers ( $p_1$  and  $p_2$ ) configuration. (b) Pointers position after one iteration of the main cycle. (c) Final position of  $p_1$  and  $p_2$  before extraction. The new link  $p_1 \rightarrow \text{next} = p_2 \rightarrow \text{next}$  is created and it allows the extraction of element  $p_2$ .

The corresponding code implementation is the following:

```

if (head == NULL) {
    /* Empty list */
    ...
}

if (value == head->key) {
    p = head->next;
    free(head);
    head = p;
}

p1 = head;
p2 = head->next;
while (p2!=NULL && value>p2->key) {
    p1 = p2
    p2 = p2->next;
}

if (p2!=NULL && value==p2->key) {
    p1->next = p2->next;
    free (p2);
} else {
    /* Element NOT found */
    ...
}

```

Notice that the code copes with two corner cases. The first one, is the one in which it considers empty lists. The second one, takes into considerations the case in which the desired element is the first one (is on the list head, or on “top”) of the list.

### **Extraction using one Explicit Pointer**

As previously stated, it is also possible to use a pointer to an element just to get its pointer. In that case, it is necessary to write the same piece of code we saw in the previous sub-section, using only the pointer p1 to visit the list, but referring to nodes p1 and p1->next at the same time (i.e., p1->next acts as p2 in the former case).

```

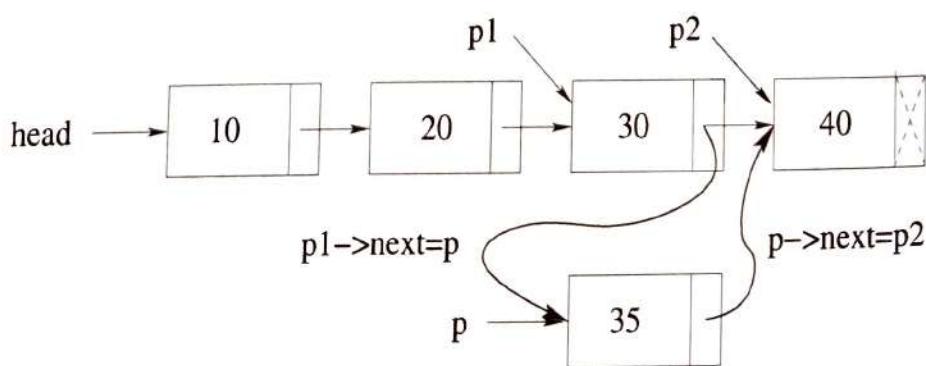
if (head == NULL) {
    /* Empty list */
    ...
}

if (value == head->key) {
    p = head->next;
    free(head);
    head = p;
}
p1 = head;
while (p1->next!=NULL && value>p1->next->key) {
    p1 = p1->next;
}
if (p1->next!=NULL && value==p1->next->key) {
    p2 = p1->next;
    p1->next = p2->next;
    free (p2);
} else {
    /* Element NOT found */
    ...
}

```

### **4.4.3 Insertion**

The insertion of a new element in an ordered list (in the proper position, obviously follows rules similar to the ones analyzed for the extraction in Section 4.4.2. Figure 4.10 illustrates the main steps.



**Figure 4.10** Ordered insertion of an element in an already ordered list.

The required C code is similar to the following:

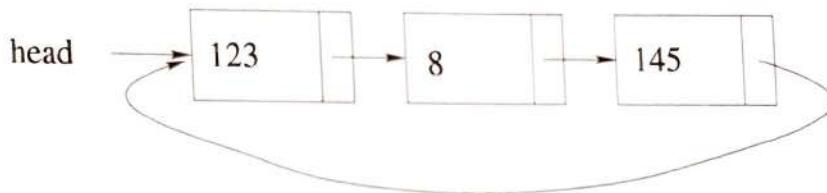
```

p = new_element ();
p->key = value;
p->next = NULL;
if (head==NULL || value<head->key) {
    p->next = head;
    head = p;
} else {
    p1 = head;
    p2 = head->next;
    while (p2!=NULL && value>p2->key) {
        p1 = p2;
        p2 = p2->next;
    }
    p->next = p2;
    p1->next = p;
}

```

## 4.5 Circular Lists

As in all previous examples the last element of a list has its next field equal to NULL, it is somehow free of charge to build a circular list as the one represent in Figure 4.11.

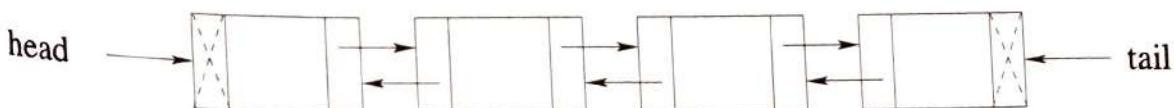


**Figure 4.11** An example of a circular list of integer values.

All standard operations analyzed in previous section on simple lists can be remapped on a circular list with some care. First of all, the last element has to point to the first one (eventually itself for one-element list) and not to NULL. Moreover, the termination condition must be modified, i.e., the list ends not when the NULL pointer is reached but when the initial pointer if found for the second time.

## 4.6 Doubly-Linked Lists

Figure 4.12 shows a doubly-directional linked list. Its element can be accessed from left-to-right (starting from the `headP` pointer) and from right-to-left (starting from the `tailP` pointer).

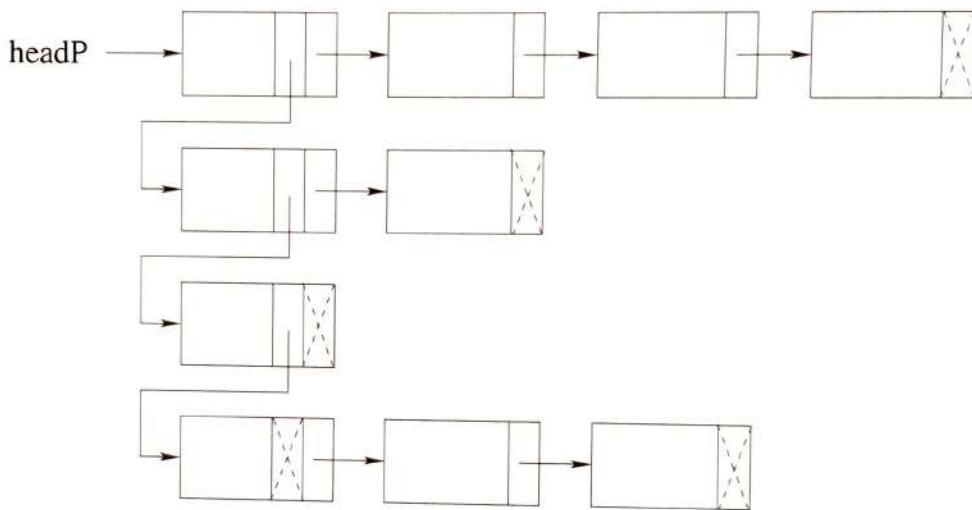


**Figure 4.12** A bi-directional linked list.

The structure is still sequential but the direction visit can be reversed. Notice that each element requires two pointers with an additional memory penalty. Anyhow, a pointer to a node is sufficient information to be able to remove it, as both pointers referring to it can be modified successfully.

## 4.7 List of Lists

Figure 4.13 illustrates a list of lists. For each element in the main list there is a secondary list. Then there should be two C structures: One to define elements of the first list (with two pointers) and one to define elements of the second list (with one single auto-referencing pointer). For each element in the main list, the first pointer is referencing elements of the same main list, and the second one is pointing to elements of its secondary list. For each element in each secondary list, the single pointer refers to the next element of the same type within the same secondary list.



**Figure 4.13** A list of lists: A vertical “main” list where each element refers to a horizontal “secondary” list.

Notice that this structure is somehow a dynamic generalization of 2D matrices where both the number of rows and the number of columns can dynamically change at run-time.

Further generalizations are possible and somehow logically trivial, but less common, and rarely used.

## 4.8 Ordered List of Integers Specifications

Write a program able to manipulate an ordered set of integers using a dynamic list. The program has to include functions to perform the following operations:

- ▷ Read an integer value from standard input and insert it in the right position into the list. The list has to be ordered in ascending order,

- ▷ Read an integer value from standard input and search it in the list printing out a message in case it exists.
- ▷ Delete the first element of the list.
- ▷ Delete the last element of the list.
- ▷ Read an integer value from standard input, search it in the list, and delete it from the list in case it is found.
- ▷ Print-out the ordered set of all integers stored into the list.
- ▷ Stop the program.

Those operations can be selected by the user through a menu.

## Solution

The following program puts together all notions we have presented in this chapter on a simple and ordered list. Then, conceptually there is nothing new, but the program gives a more complete view of how list have to be managed from the very beginning to the very end.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <ctype.h>
4
5 /* structure declaration */
6 typedef struct node_s {
7     int val;
8     struct node_s *next;
9 } node_t;
10
11 /* function prototypes */
12 int read (void);
13 node_t *insert (node_t *head, int val);
14 void search (node_t *head, int val);
15 node_t *delete_first (node_t *head);
16 node_t *delete_last (node_t *head);
17 node_t *delete (node_t *head, int val);
18 void display (node_t *head);
19
20 /*
21  * main program
22  */
23 int main(void) {
24     node_t *head=NULL;
25     int val, stop=0;
26     char choice;
27
28     while (stop == 0) {
29         fprintf(stdout, "\nAvailable commands:\n");
30         fprintf(stdout, " i: insert a value (sorted)\n");
31         fprintf(stdout, " s: search a value\n");
32         fprintf(stdout, " f: delete the first value\n");
33         fprintf(stdout, " l: delete the last value\n");
34         fprintf(stdout, " d: delete a specified value\n");
35         fprintf(stdout, " c: display the list contents\n");
36         fprintf(stdout, " e: end program\n");
37         fprintf(stdout, "Make your choice: ");
38         scanf("%c%c", &choice);
39 }
```

```

40     switch (choice) {
41         case 'i': val = read();
42             head = insert(head, val);
43             break;
44         case 's': val = read();
45             search(head, val);
46             break;
47         case 'f': head = delete_first(head);
48             break;
49         case 'l': head = delete_last(head);
50             break;
51         case 'd': val = read();
52             head = delete(head, val);
53             break;
54         case 'c': display(head);
55             break;
56         case 'e': fprintf(stdout, "End of session.\n");
57             stop = 1;
58             break;
59         default : fprintf(stdout, "Wrong choice!\n");
60             break;
61     }
62 }
63
64 return EXIT_SUCCESS;
65 }
66
67 /*
68 *   read in a value
69 */
70 int read (void) {
71     int val;
72
73     fprintf(stdout, "Value: ");
74     scanf("%d%c", &val);
75
76     return val;
77 }
78
79 /*
80 *   insert a value in the list (sorted)
81 */
82 node_t *insert (node_t *head, int val) {
83     node_t *p, *q=head;
84
85     p = (node_t *)malloc(sizeof(node_t));
86     p->val = val;
87     p->next = NULL;
88
89     /* insert ahead */
90     if (head==NULL || val<head->val) {
91         p->next = head;
92         return p;
93     }
94
95     /*
96     *   scan the list with the q pointer in order to find
97     *   the correct position where to perform the insertion
98     */

```

```
99     while (q->next!=NULL && q->next->val<val) {
100         q = q->next;
101     }
102     p->next = q->next;
103     q->next = p;
104     return head;
105 }
106
107 /* search a value in the list
108  * 
109 void search (node_t *head, int val) {
110     node_t *p;
111     int i;
112
113     for (p=head, i=0; p!=NULL && p->val<val; p=p->next, i++) ;
114
115     if (p!=NULL && p->val==val) {
116         fprintf(stderr, "Element found (index = %d)\n", i);
117     } else {
118         fprintf(stderr, "Element NOT found.\n");
119     }
120 }
121
122
123 /* delete the first element of the list
124  * 
125  */
126 node_t *delete_first (node_t *head) {
127     node_t *p;
128
129     /* empty list */
130     if (head != NULL) {
131         p = head->next;
132         free(head);
133         return p;
134     }
135
136     return head;
137 }
138
139
140 /* delete a list element, keeping it sorted
141  * 
142 node_t *delete_last (node_t *head) {
143     node_t *p, *q=head;
144
145     /* empty list */
146     if (head == NULL) {
147         fprintf(stderr, "Error: empty list\n");
148         return NULL;
149     }
150
151     /* delete ahead */
152     if (head->next == NULL) {
153         free(head);
154         return NULL;
155     }
156
157     /* scan the list with the q pointer */
```

```
158     while (q->next->next!=NULL) {
159         q = q->next;
160     }
161     p = q->next;
162     q->next = NULL;
163     free(p);
164
165     return head;
166 }
167
168 /*
169 * delete a list element, keeping it sorted
170 */
171 node_t *delete (node_t *head, int val) {
172     node_t *p, *q=head;
173
174     /* empty list */
175     if (head == NULL) {
176         fprintf(stderr, "Error: empty list\n");
177         return NULL;
178     }
179
180     /* delete ahead */
181     if (val == head->val) {
182         p = head->next;
183         free(head);
184         return p;
185     }
186
187     /*
188      * scan the list with the q pointer in order to find
189      * the element to remove from the list
190      */
191     while (q->next!=NULL && q->next->val<val) {
192         q = q->next;
193     }
194     if (q->next!=NULL && q->next->val==val) {
195         p = q->next;
196         q->next = p->next;
197         free(p);
198     } else {
199         fprintf(stderr, "Element NOT found.\n");
200     }
201
202     return head;
203 }
204
205 /*
206 * display the list contents
207 */
208 void display (node_t *head) {
209     int i=0;
210
211     while (head != NULL) {
212         fprintf(stderr, "Element %d = %d\n", i++, head->val);
213         head = head->next;
214     }
215 }
```

## 4.9 Words Frequency

### Specifications

A file stores a text of undefined length including strings of maximum length equal to 100 characters. Write a program able to evaluate the absolute frequency of all words present in the text, and to print-out the final statistics on an output file. Small and capital letters have to be considered as equivalent, i.e., “string” and “StRiNg” are the same word.

Input and output file names have to be received on the command line.

**Example 4.2** Let the following be the input file:

```
The A&P course is difficult
The A&P exam is hard
I have to study a lot all A&P topics
```

The output file has to have a content such as:

```
a 1
all 1
course 1
difficult 1
exam 1
hard 1
have 1
i 1
lot 1
study 1
to 1
topics 1
is 2
the 2
a&p 3
```

### Solution

As the number of different strings stored within the input file is unknown, using a dynamic matrix would imply re-allocation as there is no simple method to know its size before allocation. As a consequence, this is first typical example of a problem where lists are really beneficial.

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <ctype.h>
5
6 #define LINE 101
7
8 /* structure declaration */
9 typedef struct word_s {
10     char *word;
11     int count;
12     struct word_s *next;
13 } word_t;
14
15 /* function prototypes */
```

```
16 word_t *input_read (FILE *);  
17 word_t *word_find (word_t *, char *);  
18 word_t *word_sort (word_t *);  
19 void output_write (word_t *, FILE *);  
20 void cleanUp (word_t *);  
21  
22 /*  
23 * main program  
24 */  
25 int main (int argc, char *argv[]) {  
26     word_t *head=NULL;  
27     FILE *fp;  
28  
29     fp = fopen(argv[1], "r");  
30     head = input_read(fp);  
31     fclose(fp);  
32  
33     head = word_sort(head);  
34  
35     fp = fopen(argv[2], "w");  
36     output_write(head, fp);  
37     fclose(fp);  
38  
39     cleanUp(head);  
40     return 0;  
41 }  
42  
43 /*  
44 * count the occurrences of every word in the input file  
45 */  
46 word_t *input_read (FILE *fp) {  
47     word_t *tmp, *head=NULL;  
48     char buffer[LINE];  
49     int i;  
50  
51     while (fscanf(fp, "%s", buffer) != EOF) {  
52         for (i=0; i<strlen(buffer); i++) {  
53             buffer[i] = tolower(buffer[i]);  
54         }  
55         tmp = word_find(head, buffer);  
56         if (tmp != NULL) {  
57             tmp->count++;  
58         } else {  
59             tmp = (word_t *)malloc(sizeof(word_t));  
60             tmp->word = strdup(buffer);  
61             tmp->count = 1;  
62             tmp->next = head;  
63             head = tmp;  
64         }  
65     }  
66  
67     return head;  
68 }  
69  
70 /*  
71 * search for a given word in the list of words  
72 */  
73 word_t *word_find (word_t *head, char *word) {  
74     word_t *tmp=head;
```

```

75     while (tmp != NULL) {
76         if (strcmp(tmp->word, word) == 0) {
77             return tmp;
78         }
79         tmp = tmp->next;
80     }
81     return NULL;
82 }
83
84
85 /* sort the list of words (alphabetical order)
86  */
87 word_t *word_sort (word_t *head) {
88     word_t *tmp, *scan, *ptr, *sort=NULL;
89
90     while (head != NULL) {
91         tmp = head;
92         head = head->next;
93         if (sort==NULL || sort->count>tmp->count ||
94             (sort->count==tmp->count && strcmp(sort->word, tmp->word)>0)) {
95             tmp->next = sort;
96             sort = tmp;
97         } else {
98             scan = sort;
99             ptr = scan->next;
100            while (ptr!=NULL && (ptr->count<tmp->count ||
101                  (ptr->count==tmp->count && strcmp(ptr->word, tmp->word)<0))) {
102                scan = ptr;
103                ptr = scan->next;
104            }
105            scan->next = tmp;
106            tmp->next = ptr;
107        }
108    }
109 }
110
111 return sort;
112 }
113
114 /* write the output file
115  */
116 void output_write (word_t *head, FILE *fp) {
117     word_t *tmp=head;
118
119     while (tmp != NULL) {
120         fprintf(fp, "%s %d\n", tmp->word, tmp->count);
121         tmp = tmp->next;
122     }
123
124     return;
125 }
126
127 /*
128  * quit the dynamically allocated memory for the list of words
129  */
130 void cleanUp (word_t *head) {
131     word_t *tmp;
132
133     while (head != NULL) {

```

```

134     tmp = head;
135     head = head->next;
136     free(tmp->word);
137     free(tmp);
138 }
139
140 return;
141 }
```

## 4.10 Employees

### Specifications

A file stores information regarding all employees of a large firm. For each employee the file includes the following lines:

```

lastName firstName id workingWeeks
week1 workingDays hoursDay1 hoursDay2 ...
week2 workingDays hoursDay1 hoursDay2 ...
...
...
```

where:

- ▷ All fields are strings of user selected lengths or integer values.
- ▷ The first line (for each employee) includes the generality of the employee, and the number of weeks he/she has worked (`workingWeeks`).
- ▷ All following lines report for each week the number of days, and for each day the number of hours he/she has worked. Notice that `week` indicates the week within the year (an integer from 1 to 52), `workingDays` the working days within that week (from 1 to 7), and `hoursDay` the hour worked within that day (from 1 to 24).

The file order is unspecified.

Write a program able to:

- ▷ Receive two file names on the command line.
- ▷ Read the first file, with the previous content, and store its content in a proper data structure.
- ▷ Store the entire data structure in an output file, where employees are ordered by ascending number of total working hours.

All data structures to store file information have to be dynamically allocated.

**Example 4.3** Let the following be the input file:

```

Rossi Giovanna AAABBBCCDEEFFFFG 3
3 5 8 8 8 8 8
5 6 7 5 8 3 8 9
12 7 6 5 7 8 8 8 8
Bianchi Alberto GGGHHHIILMMNNNNO 2
2 5 6 6 6 6 6
3 5 7 5 8 4 3
```

then the following will be the output file:

```

Bianchi Alberto GGGHHHIIILMMNNNNO 2
2 5 6 6 6 6 6
3 5 7 5 8 4 3
Rossi Giovanna AAABBBCCDEEFFFG 3
3 5 8 8 8 8 8
5 6 7 5 8 3 8 9
12 7 6 5 7 8 8 8 8

```

## Solution

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 #define MAX 50+1
6
7 /* structure declaration */
8 typedef struct employee_s {
9     char *nc;
10    char cf[17];
11    int ns;
12    int **mat;
13    int to;
14    struct employee_s *next;
15 } employee_t;
16
17 /* function prototypes */
18 employee_t *employee_read (FILE *);
19 employee_t *list_insert (employee_t *, employee_t *);
20 void file_write (FILE *, employee_t *);
21 void memory_free (employee_t *);
22
23 /*
24  * main program
25 */
26 int main (int argc, char *argv[]) {
27     employee_t *e, *head = NULL;
28     FILE *fin, *fout;
29
30     fin = fopen(argv[1], "r");
31     do {
32         e = employee_read(fin);
33         if (e != NULL) {
34             head = list_insert(head, e);
35         }
36     } while (e != NULL);
37     fclose(fin);
38
39     fout = fopen(argv[2], "w");
40     file_write(fout, head);
41     fclose(fout);
42
43     memory_free(head);
44     return EXIT_SUCCESS;
45 }
46
47 /*
48  * read info for a new employee from file

```

```
49  */
50 employee_t *employee_read (FILE *fin) {
51     char n[MAX], c[MAX], cf[17], tmp[MAX*2];
52     employee_t *e;
53     int i, j, ns, num, day;
54
55     if (fscanf(fin, "%s %s %s %d", c, n, cf, &ns) == EOF) {
56         return NULL;
57     }
58
59     e = (employee_t *)malloc(sizeof(employee_t));
60     sprintf(tmp, "%s %s", c, n);
61     e->nc = strdup(tmp);
62     strcpy(e->cf, cf);
63     e->ns = ns;
64     e->to = 0;
65
66     e->mat = (int **)malloc(ns * sizeof(int *));
67     for (i=0; i<ns; i++) {
68         fscanf(fin, "%d %d", &num, &day);
69         e->mat[i] = (int *)malloc((day+2) * sizeof(int));
70         e->mat[i][0] = num;
71         e->mat[i][1] = day;
72         for (j=0; j<day; j++) {
73             fscanf(fin, "%d", &e->mat[i][j+2]);
74             e->to = e->to + e->mat[i][j+2];
75         }
76     }
77
78     return e;
79 }
80
81 /*
82 * add a new element to the list of employees
83 */
84 employee_t *list_insert (employee_t *head, employee_t *e) {
85     employee_t *p;
86
87     if (head==NULL || e->to<head->to) {
88         e->next = head;
89         return e;
90     }
91
92     p = head;
93     while (p->next!=NULL && p->next->to<e->to) {
94         p = p->next;
95     }
96     e->next = p->next;
97     p->next = e;
98
99     return head;
100}
101
102 /*
103 * write the output file
104 */
105 void file_write (FILE *fp, employee_t *head) {
106     employee_t *e=head;
107     int i, j;
```

```

108 while (e != NULL) {
109     fprintf(fp, "%s %s %d\n", e->nc, e->cf, e->ns);
110     for (i=0; i<e->ns; i++) {
111         for (j=0; j<e->mat[i][1]+2; j++) {
112             fprintf(fp, "%d ", e->mat[i][j]);
113         }
114         fprintf(fp, "\n");
115     }
116     e = e->next;
117 }
118 }
119 */
120 /*
121 * free all the dynamically allocated memory
122 */
123 void memory_free (employee_t *head) {
124     employee_t *e;
125     int i;
126
127     while (head != NULL) {
128         for (i=0; i<head->ns; i++) {
129             free(head->mat[i]);
130         }
131         free(head->mat);
132         free(head->nc);
133         e = head;
134         head = head->next;
135         free(e);
136     }
137 }
138 }
```

## 4.11 Safari

### Specifications

Write an application to organized safari in Tanzania in a more eco-friendly way, avoiding to much pollution and overcrowded park areas.

The program receives two file names on the command line.

The first file stores all Tanzania parking area names with the maximum number of vehicles which can park in those areas. Each line of the file has the following format:

`areaName maxNumberOfVehicles`

where `areaName` is a string of 25 characters at most, and `maxNumberOfVehicles` is the maximum number of vehicles allowed in the area. The following is a correct file:

```

lakeManyara 9
serenghetia 15
serenghetib 20
serenghetic 20
ngorongoroz 10
...
```

The second file stores all travel agency requests. It stores for each vehicle of each travel agency a set of lines indicating the arrival and departure times for each area the vehicle would like to visit:

```

#vehicleName N
areaName1 arrivalTime departureTime
...
```

```
areaNameN arrivalTime departureTime
```

where N is the number of areas that vehicle would like to visit. Times are reported with the format hh:mm, i.e., 06:00, and are discretized within 15 minutes intervals, i.e., 06:00, 06:15, 06:30, etc. The following is a correct file:

```
#RiftSafariA 3
ngorongoroz 06:00 08:00
serenghetia 09:00 09:45
serenghetib 12:30 14:15
#Leopard 2
lakeManyara 06:00 07:30
serenghetic 11:30 14:00
...
```

The program, once read the two files, has to verify, at 15 minutes intervals, whether the number of vehicles parked in each area is smaller than the allowed number of vehicle for that area. The program has to indicate all possible problems, reporting the area name and the time range whenever a problem is detected.

## Solution

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define DIM1 25
6 #define DIM2 48
7
8 /* structure declaration */
9 typedef struct area_s {
10     char *name;
11     int max_jeeps;
12     int hours[DIM2];
13     struct area_s *next;
14 } area_t;
15
16 /* function prototypes */
17 area_t *load(char *filename);
18 void fill(area_t *head, char *filename);
19 area_t *find(area_t *head, char *name);
20 void check(area_t *head);
21 void quit(area_t *head);
22 void display(area_t *head);
23
24 /*
25  * main program
26 */
27 int main(int argc, char *argv[]) {
28     area_t *head=NULL;
29
30     head = load(argv[1]);
31     fill(head, argv[2]);
32
33     check(head);
34     quit(head);
35
36     return EXIT_SUCCESS;
37 }
```

```

38
39  /*
40   * load the set of areas from file
41  */
42 area_t *load(char *filename) {
43     area_t *ptr, *head=NULL;
44     char name[DIM1];
45     int i, num;
46     FILE *fp;
47
48     fp = fopen(filename, "r");
49     if (fp==NULL) {
50         fprintf (stderr, "File open error.\n");
51         exit(EXIT_FAILURE);
52     }
53     while (fscanf(fp, "%s %d", name, &num) != EOF) {
54         ptr = (area_t *)malloc(sizeof(area_t));
55         ptr->name = strdup(name);
56         ptr->max_jeeps = num;
57         for (i=0; i<DIM2; i++) {
58             ptr->hours[i] = 0;
59         }
60         ptr->next = head;
61         head = ptr;
62     }
63     fclose(fp);
64     return head;
65 }
66
67 /*
68  * load the set of requests from file
69  */
70 void fill(area_t *head, char filename[DIM1]) {
71     char name[DIM1], start[DIM1], stop[DIM1];
72     int h1, m1, h2, m2, n, i, j, k, na;
73     area_t *ptr;
74     FILE *fp;
75
76     fp = fopen (filename, "r");
77     if (fp==NULL) {
78         fprintf (stderr, "File open error.\n");
79         exit(EXIT_FAILURE);
80     }
81     while (fscanf (fp, "%s %d", &na) != EOF) {
82         for (n=0; n<na; n++) {
83             fscanf (fp, "%s %s %s", name, start, stop);
84             ptr = find (head, name);
85             if (ptr != NULL) {
86                 sscanf (start, "%d:%d", &h1, &m1);
87                 sscanf (stop, "%d:%d", &h2, &m2);
88                 i = h1*4 - 24 + m1/15;
89                 j = h2*4 - 24 + m2/15;
90
91                 for (k=i; k<j; k++) {
92                     ptr->hours[k]++;
93                 }
94             }
95         }
96     }
}

```

```
97     fclose(fp);
98 }
99
100 /*
101 * find an area given its name
102 */
103 area_t *find(area_t *head, char *name) {
104     area_t *ptr=head;
105
106     printf ("Looking for %s --> ", name);
107     while (ptr != NULL) {
108         printf ("%s ", ptr->name);
109         if (strcmp(ptr->name, name) == 0) {
110             printf ("FOUND\n");
111             return ptr;
112         }
113         ptr = ptr->next;
114     }
115     printf ("NOTFOUND\n");
116
117     return NULL;
118 }
119
120 /*
121 * verify the number of jeeps in every area against the maximum
122 */
123 void check(area_t *head) {
124     int i, h1, m1, h2, m2, min, max;
125     area_t *ptr=head;
126
127     while (ptr != NULL) {
128         for (i=0; i<DIM2; i++) {
129             if (ptr->hours[i] >= ptr->max_jeeps) {
130                 min = i;
131                 while (i<DIM2 && ptr->hours[i]>=ptr->max_jeeps) {
132                     i = i + 1;
133                 }
134                 max = i;
135                 h1 = (min+24)/4;
136                 m1 = ((min+24)%4)*15;
137                 h2 = (max+24)/4;
138                 m2 = ((max+24)%4)*15;
139                 fprintf(stdout, "In area %s, the number of jeeps", ptr->name);
140                 fprintf(stdout, "is not strictly less than the maximum in ");
141                 fprintf(stdout, "time interval %d:%d -- %d:%d\n", h1, m1, h2, m2);
142             }
143         }
144         ptr = ptr->next;
145     }
146 }
147
148 /*
149 * de-allocate the memory
150 */
151 void quit(area_t *head) {
152     area_t *ptr=head;
153
154     while (head != NULL) {
155         ptr = head;
```

```

156     head = ptr->next;
157     free(ptr->name);
158     free(ptr);
159 }
160 }
```

## 4.12 List Sorting

### Problem definition

Sorting a linked list may be straightforward if we do maintain two lists, an input list (to be ordered) and an output list (the ordered one). For example, selection sort can be modified as follow. While is nonempty, we scan the input list to find the maximum remaining element. Then, we remove this element from the list. Finally, we insert the element at the front of the output list.

In the following, we suggest an application in which sorting is applied “in place”, i.e., using a single list (as input and output list at the same time).

### Specifications

A file stores information about all students who passed an exam. For each student a row of the file reports the last name (string of 20 characters at most), and the final mark (real value in the range [0, 30]). Students are stored in random order.

Write a program able to execute the following commands (eventually more than once):

- ▷ **read fileName**  
read a file name, whose structure is the previous one, and stores the content of the file in a list.
- ▷ **writeN fileName**  
store the current list onto file `fileName` with the same format of the input file but ordered by name.
- ▷ **writeM fileName**  
store the current list onto file `fileName` with the same format of the input file but ordered by mark.
- ▷ **stop**  
terminates the program.

Observe that whenever a `writeN` command is followed by a `writeM` command (or vice-versa) all students must be re-ordered before storing them into the output file.

### Solution

A little attention has to be adopted to implement an ordering algorithm on a list: A list is a sequential data structure and the only possibility to traverse it is to follow links. In other words it is not possible to move “backward” on a list or access a specific element in a direct way.

Moreover, the ordering algorithm has to decide whether it wants to physically swap list elements or just swap element content.

The following solution use exchange sort and swap only the content of the elements without changing pointers.

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <ctype.h>
5
6 #define DEBUG_FLAG 0
7
8 #define ROW_LEN 100
9 #define STR_LEN 21
10 #define SUCCESS 1
11 #define FAILURE 2
12
13 typedef struct node_s {
14     char name[STR_LEN];
15     float mark;
16     struct node_s *next;
17 } node_t;
18
19 node_t *file_read (node_t *, char *);
20 void orderByName (node_t *);
21 void orderByMark (node_t *);
22 void file_write (node_t *, char *);
23 node_t *newE (void);
24 void memory_free(node_t *);
25
26 int main(void) {
27     node_t *headPtr;
28     char cmd[ROW_LEN], name[ROW_LEN];
29     int stop=0;
30
31     headPtr = NULL;
32
33     while (stop == 0) {
34         fprintf(stdout, "\nAvailable commands:\n");
35         fprintf(stdout, "    read <filename>\n");
36         fprintf(stdout, "    writeN <filename> (<filename>==stdout --> screen)\n");
37         fprintf(stdout, "    writeM <filename> (<filename>==stdout --> screen)\n");
38         fprintf(stdout, "    stop\n");
39         fprintf(stdout, "Your choice ---> ");
40         scanf ("%s", cmd);
41
42         if (strcmp(cmd, "read") == 0) {
43             scanf("%s", name);
44             fprintf(stdout, "Reading file %s.\n", cmd);
45             headPtr = file_read (headPtr, name);
46         } else if (strcmp(cmd, "writeN") == 0) {
47             scanf("%s", name);
48             fprintf(stdout, "Writing file %s.\n", name);
49             orderByName (headPtr);
50             file_write (headPtr, name);
51         } else if (strcmp(cmd, "writeM") == 0) {
52             scanf("%s", name);
53             fprintf(stdout, "Writing file %s.\n", name);
54             orderByMark (headPtr);
55             file_write (headPtr, name);
56         } else if (strcmp(cmd, "stop") == 0) {
57             fprintf(stdout, "Program terminated.\n");
58             stop = 1;
59         } else {

```

```

60     fprintf(stderr, "Error: unknown command (%s).\n", cmd);
61 }
62 }
63 memory_free (headPtr);
64 return EXIT_SUCCESS;
65 }

66 void orderByName (node_t *headPtr) {
67     char tmpStr[STR_LEN];
68     float tmpFloat;
69     node_t *endPtr, *tmpPtr1, *tmpPtr2, *tmpPtr3;
70
71     /* Empty List */
72     if (headPtr == NULL) {
73         return;
74     }
75
76     /* List with one element */
77     if (headPtr->next == NULL) {
78         return;
79     }
80
81     /* List with more than one element */
82     endPtr = NULL;
83     /* It iterates once for each list element */
84     for (tmpPtr1 = headPtr; tmpPtr1 != NULL; tmpPtr1 = tmpPtr1->next) {
85         /* It swaps all elements in wrong position */
86         for (tmpPtr2=headPtr, tmpPtr3=headPtr->next; tmpPtr3!=endPtr;
87              tmpPtr2=tmpPtr3, tmpPtr3=tmpPtr3->next) {
88
89             if (strcmp (tmpPtr2->name, tmpPtr3->name) > 0) {
90                 strcpy (tmpStr, tmpPtr2->name);
91                 tmpFloat = tmpPtr2->mark;
92                 strcpy (tmpPtr2->name, tmpPtr3->name);
93                 tmpPtr2->mark = tmpPtr3->mark;
94                 strcpy (tmpPtr3->name, tmpStr);
95                 tmpPtr3->mark = tmpFloat;
96             }
97
98         }
99
100    }
101
102 #if DEBUG_FLAG
103     fprintf (stdout, "List:\n");
104     file_write (headPtr, "stdout");
105 #endif
106
107     /* Move backward final pointer for inner cycle */
108     endPtr = tmpPtr2;
109 }
110
111     return;
112 }
113
114 void orderByMark (node_t *headPtr) {
115     char tmpStr[STR_LEN];
116     float tmpFloat;
117     node_t *endPtr, *tmpPtr1, *tmpPtr2, *tmpPtr3;
118

```

```
119  /* Empty List */
120  if (headPtr == NULL) {
121      return;
122  }
123
124  /* List with one element */
125  if (headPtr->next == NULL) {
126      return;
127  }
128
129  /* List with more than one element */
130  endPtr = NULL;
131  for (tmpPtr1 = headPtr; tmpPtr1 != NULL; tmpPtr1 = tmpPtr1->next) {
132      for (tmpPtr2=headPtr, tmpPtr3=headPtr->next; tmpPtr3!=endPtr;
133            tmpPtr2=tmpPtr3, tmpPtr3=tmpPtr3->next) {
134
135          if (tmpPtr2->mark > tmpPtr3->mark) {
136              strcpy (tmpStr, tmpPtr2->name);
137              tmpFloat = tmpPtr2->mark;
138              strcpy (tmpPtr2->name, tmpPtr3->name);
139              tmpPtr2->mark = tmpPtr3->mark;
140              strcpy (tmpPtr3->name, tmpStr);
141              tmpPtr3->mark = tmpFloat;
142          }
143      }
144  }
145
146  /* Move backward final pointer for inner cycle */
147  endPtr = tmpPtr2;
148 }
149
150 #if DEBUG_FLAG
151     fprintf (stdout, "List:\n");
152     file_write (headPtr, "stdout");
153 #endif
154
155     return;
156 }
157
158 void file_write (node_t *headPtr, char *name) {
159     FILE *fp;
160     node_t *tmpPtr;
161
162     if (strcmp (name, "stdout") == 0) {
163         fp = stdout;
164     } else {
165         fp = fopen (name, "w");
166         if (fp==NULL) {
167             fprintf (stderr, "File open error (file=%s).\n", name);
168             return;
169         }
170     }
171
172     tmpPtr = headPtr;
173
174     while (tmpPtr != NULL) {
175         fprintf (fp, "%s %f\n", tmpPtr->name, tmpPtr->mark);
176         tmpPtr = tmpPtr->next;
177     }
```

```
178     if (strcmp (name, "stdout") != 0) {
179         fclose (fp);
180     }
181
182     return;
183 }
184
185 node_t *file_read (node_t *headPtr, char *name) {
186     FILE *fp;
187     char row[ROW_LEN];
188     float mark;
189     node_t *tmpPtr;
190
191     fp = fopen (name, "r");
192     if (fp==NULL) {
193         fprintf (stderr, "File open error (file=%s).\n", name);
194         return (NULL);
195     }
196
197     headPtr = NULL;
198
199     while ( fgets (row, ROW_LEN, fp) != NULL ) {
200
201         sscanf (row, "%s %f", name, &mark);
202
203         tmpPtr = newE ();
204         strcpy (tmpPtr->name, name);
205         tmpPtr->mark = mark;
206
207         /* Head Insertion */
208         tmpPtr->next = headPtr;
209         headPtr = tmpPtr;
210     }
211
212     fclose (fp);
213
214     return (headPtr);
215 }
216
217
218 node_t *newE (void) {
219     node_t *tmpPtr;
220
221     tmpPtr = (node_t *) malloc (sizeof (node_t));
222     if (tmpPtr == NULL) {
223         fprintf (stderr, "Memory allocation error.\n");
224         exit(EXIT_FAILURE);
225     }
226
227     return (tmpPtr);
228 }
229
230 void memory_free (node_t *head) {
231     node_t *tmp;
232
233     while (head != NULL) {
234         tmp = head;
235         head = head->next;
236         free(tmp);
```

```
237     }
238 }
```

## 4.13 Overlapped Lists

### Specifications

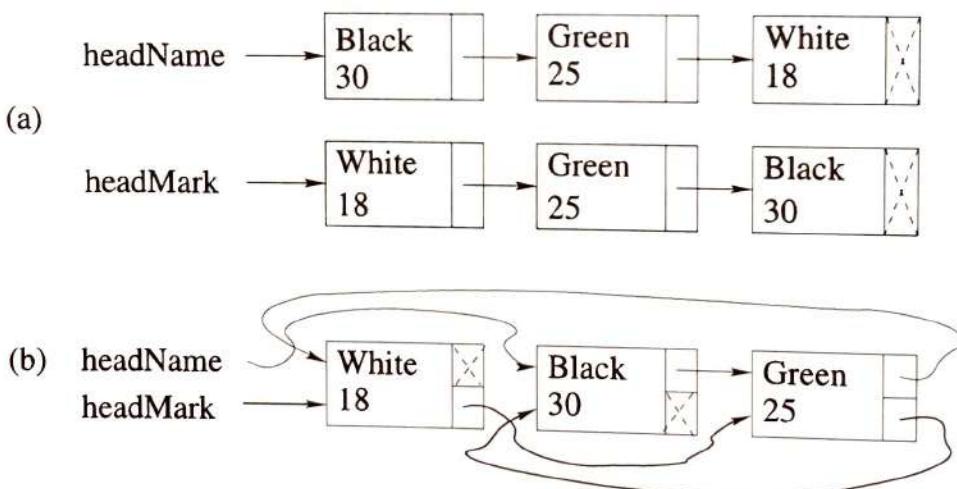
Following specifications of Exercise 4.12, write a program which is able to execute a `writeN` command followed by a `writeM` command (or vice-versa) without re-ordering the data between the two commands.

### Solution

Let the following be the file the program has to manipulate:

```
White 18
Black 30
Green 25
```

Figure 4.14(a) shows a first possible solution in which all data is stored twice in two similar lists, the first one, ordered by name, and the second one by mark.



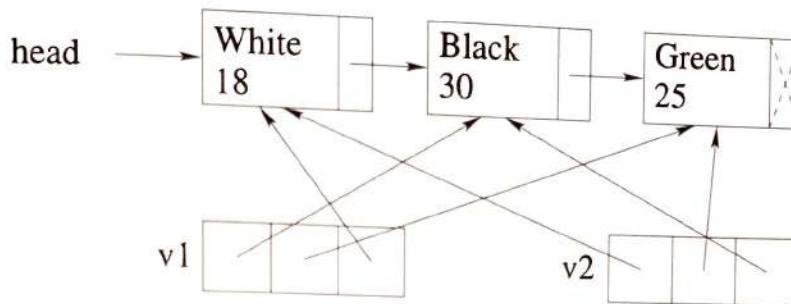
**Figure 4.14** Storing lists: (a) ordered by name and by mark, (b) merged lists.

This solution satisfies the specifications but has several drawbacks. First of all, it requires a memory space which is twice the one required to store all information. Secondly, all operations, i.e., each insertion or extraction of an element, have to be performed twice, once for each list. Thirdly, the structure presents consistency problem, as the program may end up with two lists non including exactly the same set of elements.

A possible fix, for all this problem, is represented by Figure 4.14(b). In this case, the two previous lists are stored as a unique *physical* list representing two *logical* lists. The first logical list is ordered by name, and the second one by mark. To represent the two logical lists the physical list has two pointers for each element, the first one to

deal with the name ordering and the second one to deal with the mark ordering. In practice, this structure is a first example on how to organize a data structure to obtain multiple orders, each one with a different key.

Notice that there may be several other possibilities to this problem. Figure 4.15 shows an example where the list stores the data, and two support arrays specify the two required orders.



**Figure 4.15** Storing list with two support arrays ordered by name and by mark.

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 #define MAX 100
6 #define NAME 1
7 #define MARK 2
8
9 /* structure declaration */
10 typedef struct node_s {
11     char name[MAX];
12     int mark;
13     struct node_s *nextN;
14     struct node_s *nextM;
15 } node_t;
16
17 /* function prototypes */
18 void file_read (node_t **headName, node_t **headMark, char *filename);
19 node_t *insertByName (node_t *head, node_t *ptr);
20 node_t *insertByMark (node_t *head, node_t *ptr);
21 void file_write (node_t *head, char *filename, int nameMark);
22 void memory_free (node_t *headMark);
23
24 /*
25  * main program
26 */
27 int main (void) {
28     node_t *headName=NULL, *headMark=NULL;
29     char cmd[MAX], name[MAX];
30     int stop=0;
31
32     while (stop == 0) {
33         printf(stdout, "\nAvailable commands:\n ");
34         printf(stdout, " read <filename>\n ");
35         printf(stdout, " writeN <filename> (<filename>==stdout --> screen)\n ");
36         printf(stdout, " writeM <filename> (<filename>==stdout --> screen)\n ");
37         printf(stdout, " stop\n ");
  
```

```
38     fprintf(stdout, "Your choice ---> ");
39     scanf ("%s", cmd);
40
41     if (strcmp(cmd, "read") == 0) {
42         scanf("%s", name);
43         fprintf(stdout, "Reading file %s.\n", cmd);
44         memory_free(headMark);
45         file_read(&headName, &headMark, name);
46     } else if (strcmp(cmd, "writeN") == 0) {
47         scanf("%s", name);
48         fprintf(stdout, "Writing file %s.\n", name);
49         file_write(headName, name, NAME);
50     } else if (strcmp(cmd, "writeM") == 0) {
51         scanf("%s", name);
52         fprintf(stdout, "Writing file %s.\n", name);
53         file_write(headMark, name, MARK);
54     } else if (strcmp(cmd, "stop") == 0) {
55         fprintf(stdout, "Program terminated.\n");
56         stop = 1;
57     } else {
58         fprintf(stderr, "Error: unknown command (%s).\n", cmd);
59     }
60 }
61
62 memory_free(headMark);
63 return EXIT_SUCCESS;
64 }
65
66 /*
67 * load the contents of a file into the internal data structure
68 */
69 void file_read (node_t **headName, node_t **headMark, char *filename) {
70     char name[MAX];
71     int mark;
72     node_t *tmp;
73     FILE *fp;
74
75     *headName = NULL;
76     *headMark = NULL;
77
78     fp = fopen(filename, "r");
79     while (fscanf(fp, "%s %d", name, &mark) != EOF) {
80         tmp = (node_t *)malloc(sizeof(node_t));
81         if (tmp==NULL) {
82             fprintf(stderr, "Memory allocation error.\n");
83             exit(EXIT_FAILURE);
84         }
85         strcpy(tmp->name, name);
86         tmp->mark = mark;
87
88         *headMark = insertByMark(*headMark, tmp);
89         *headName = insertByName(*headName, tmp);
90     }
91     fclose (fp);
92 }
93
94 /*
95 * insert a new element to the list according to the mark order
96 */
```

```
97 node_t *insertByMark(node_t *head, node_t *ptr) {
98     node_t *tmp0, *tmp1;
99
100    if (head==NULL || ptr->mark<head->mark) {
101        ptr->nextM = head;
102        return ptr;
103    }
104
105    tmp0 = head;
106    tmp1 = head->nextM;
107    while (tmp1!=NULL && tmp1->mark<ptr->mark) {
108        tmp0 = tmp1;
109        tmp1 = tmp1->nextM;
110    }
111    tmp0->nextM = ptr;
112    ptr->nextM = tmp1;
113
114    return head;
115}
116
117 /*
118  * insert a new element to the list according to the name order
119 */
120 node_t *insertByName(node_t *head, node_t *ptr) {
121     node_t *tmp0, *tmp1;
122
123    if (head==NULL || strcmp(ptr->name, head->name)<0) {
124        ptr->nextN = head;
125        return ptr;
126    }
127
128    tmp0 = head;
129    tmp1 = head->nextN;
130    while (tmp1!=NULL && strcmp(tmp1->name, ptr->name)<0) {
131        tmp0 = tmp1;
132        tmp1 = tmp1->nextN;
133    }
134    tmp0->nextN = ptr;
135    ptr->nextN = tmp1;
136
137    return head;
138}
139
140 /*
141  * write the internal data structure onto a file
142 */
143 void file_write (node_t *head, char *filename, int nameMark) {
144     FILE *fp;
145     node_t *tmp;
146
147     if (strcmp(filename, "stdout") == 0) {
148         fp = stdout;
149     } else {
150         fp = fopen(filename, "w");
151     }
152
153     tmp = head;
154     while (tmp != NULL) {
155         fprintf(fp, "%s %d\n", tmp->name, tmp->mark);
```

```

156     if (nameMark == NAME) {
157         tmp = tmp->nextN;
158     } else {
159         tmp = tmp->nextM;
160     }
161 }
162 if (strcmp(filename, "stdout") != 0) {
163     fclose (fp);
164 }
165 }
166 }
167 */
168 /* free the allocated memory
169 */
170 void memory_free (node_t *headMark) {
171     node_t *tmp;
172
173     while (headMark != NULL) {
174         tmp = headMark;
175         headMark = headMark->nextM;
176         free(tmp);
177     }
178 }
179
180 return;
181 }
```

## 4.14 Formula 1

### Specification

Write a program to handle Formula 1 races. The program receives on the command line a file name storing lines with the following format:

lapNumber lapFileName

where `lapNumber` is an integer and `lapFileName` is a string (with a maximum of 20 characters). Each file `lapFileName` stores statistics on a single lap. For that lap the file stores, for all drivers the following information:

driverNumber driverName team lapTime

where `driverNumber` is an integer, `driverName` and `team` are strings (20 characters at most), and `lapTime` is an integer representing the time required to complete an entire lap (in seconds). The maximum number of teams and the number of drivers for each team is undefined.

The program must find:

- ▷ The driver with the smallest lap time. The driver name, the lap time and the lap number have to printed-out on standard output.
- ▷ The rank of the first 3 drivers with the smallest total time (sum of all lap times for all laps) on the entire race.

Notice that, if two drivers have that same time their relative order is not important.

**Example 4.4** The following is a correct example of all input files:

**First File**

1 fileLap1.dat  
2 fileLap2.dat

fileLap1.dat

1 Raikkonen Ferrari 62  
2 Vettel Ferrari 61  
3 Hamilton Mercedes 74  
4 Bottas Mercedes 78  
11 Ricciardo RedBull 75  
12 Verstappen RedBull 72  
20 Hulkenberg Reault 71  
21 Kvjat ToroRoss 64

fileLap2.dat

1 Raikkonen Ferrari 63  
2 Vettel Ferrari 60  
3 Hamilton Mercedes 78  
4 Bottas Mercedes 72  
11 Ricciardo RedBull 76  
12 Verstappen RedBull 75  
20 Hulkenberg Reault 120  
21 kvjat ToroRosso 62

The program should generate the following output:

Fastest Lap: Vettel, Ferrari, 60 seconds, Lap 2

**Driver Ranking:**

1 Vettel, Ferrari, 121 seconds  
2 Raikkonen, Ferrari, 125 seconds  
3 Kvjat, ToroRosso, 126 seconds

**Solution**

To generate the desired output, the following program does not order the list but it just selects the 3 fastest drivers and it prints them out on standard output.

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 #define MAX 81
6
7 /* structure declaration*/
8 typedef struct driver_s {
9     int id;
10    char *name;
11    char *car;
12    int total_time;
13    int best_time;
14    int best_lap;
15    int marked;
16    struct driver_s *next;
17 } driver_t;
18
19 /* function prototypes */
20 driver_t *load(char *filename);
21 driver_t *read(driver_t *head, int lap, char *filename);
22 driver_t *find(driver_t *head, int id);
23 void display(driver_t *head);
24 void quit(driver_t *head);
25
26 /*
27  * main program
28 */
29 int main (int argc, char *argv[]) {
30     driver_t *head;
31     head = load(argv[1]);
32     display(head);
33     quit(head);
34
35     return EXIT_SUCCESS;

```

```
36  }
37
38 /* 
39  *   read the file containing the lap file names
40  */
41 driver_t *load (char *filename) {
42     driver_t *head=NULL;
43     char name[MAX];
44     int lap;
45     FILE *fp;
46
47     fp = fopen(filename, "r");
48     while (fscanf(fp, "%d %s", &lap, name) != EOF) {
49         head = read(head, lap, name);
50     }
51     fclose(fp);
52     return head;
53 }
54
55 /*
56  *   read a file containing the lap details
57  */
58 driver_t *read (driver_t *head, int lap, char *filename) {
59     char line[MAX], name[MAX], car[MAX];
60     driver_t *driver;
61     int id, time;
62     FILE *fp;
63
64     fp = fopen(filename, "r");
65     if (fp==NULL) {
66         fprintf (stderr, "Error reading file!\n");
67         exit(EXIT_FAILURE);
68     }
69     while (fgets(line, MAX, fp)) {
70         sscanf(line, "%d %s %s %d", &id, name, car, &time);
71         driver = find(head, id);
72         if (driver == NULL) {
73             driver = (driver_t *)malloc(sizeof(driver_t));
74             driver->id = id;
75             driver->name = strdup(name);
76             driver->car = strdup(car);
77             driver->total_time = 0;
78             driver->best_time = time;
79             driver->best_lap = lap;
80             driver->marked = 0;
81             driver->next = head;
82             head = driver;
83         }
84         driver->total_time += time;
85         if (time < driver->best_time) {
86             driver->best_time = time;
87             driver->best_lap = lap;
88         }
89     }
90 }
91
92 fclose(fp);
93
94 return head;
```

```

95 }
96 /* list search
97 */
98 driver_t *find (driver_t *head, int id) {
99     driver_t *tmp = head;
100
101    while (tmp != NULL) {
102        if (tmp->id == id) {
103            return tmp;
104        }
105        tmp = tmp->next;
106    }
107    return NULL;
108 }
109
110 */
111 /* print the requested results
112 */
113 void display(driver_t *head) {
114     driver_t *rank[3] = {NULL, NULL, NULL};
115     driver_t *curr, *tmp, *best = NULL;
116     int i;
117
118     for (i=0; i<3; i++) {
119         curr = NULL;
120         for (tmp=head; tmp!=NULL; tmp=tmp->next) {
121             if (tmp->marked == 0) {
122                 if (curr == NULL || tmp->total_time<curr->total_time) {
123                     curr = tmp;
124                 }
125             }
126             if (best==NULL || tmp->best_time<best->best_time) {
127                 best = tmp;
128             }
129         }
130     }
131     rank[i] = curr;
132     curr->marked = 1;
133 }
134
135     fprintf(stdout, "Best lap: %s, %s, ", best->name, best->car);
136     fprintf(stdout, "%d sec, lap no. %d\n", best->best_time, best->best_lap);
137     fprintf(stdout, "\nDrivers ranking:\n");
138     for (i=0; i<3; i++) {
139         fprintf(stdout, "%d %s, %s, %d s\n", i+1, rank[i]->name,
140             rank[i]->car, rank[i]->total_time);
141     }
142 }
143
144 /*
145 * free the allocated memory
146 */
147 void quit(driver_t *head) {
148     driver_t *tmp;
149
150     while (head != NULL) {
151         tmp = head;
152         head = tmp->next;
153         free(tmp->name);

```

```

154     free(tmp->car);
155     free(tmp);
156 }
157
158 return;
159 }
```

## 4.15 Memory Allocator

### Specifications

Write a program able to simulate dynamic memory manipulation as follows.

The system has 8MB of main memory including blocks of one byte. Initially, all memory blocks (bytes) are free. Each call to the function:

```
void *MyMalloc(int size);
```

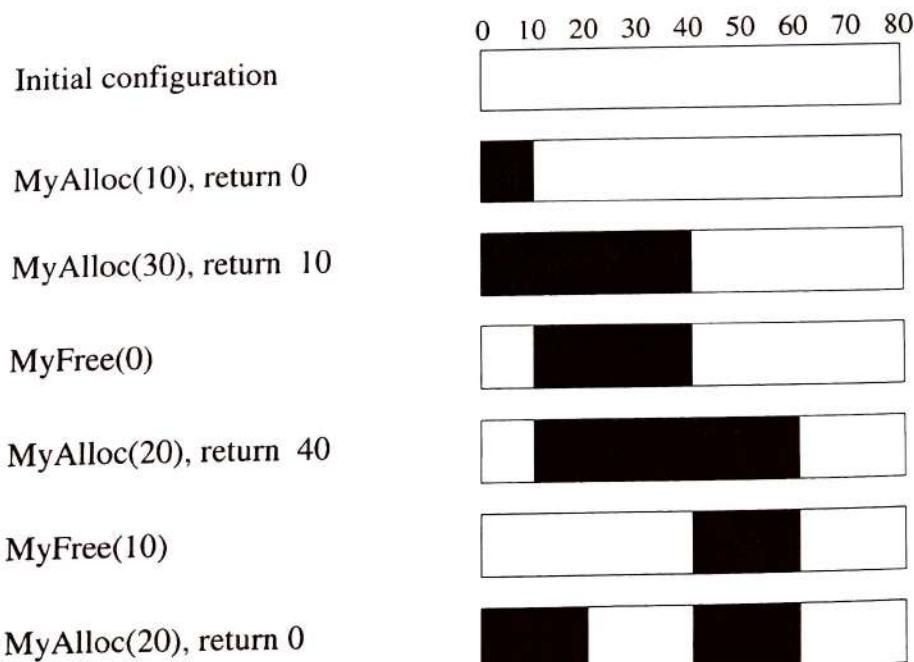
will return the first set of `size` contiguous bytes, and it will mark them as occupied. Each call to function:

```
void MyFree(void *ptr);
```

will mark all bytes referenced by `ptr` as free.

After a certain number of calls to those functions, the memory will include both free and occupied blocks. Those two sets have to be maintained by the program to know how to perform each future `MyMalloc` and `MyFree`. Contiguous free blocks have to be merge together.

**Example 4.5** Figure 4.16 reports a graphical representation of a sequence of legal operation performed by the program on a memory of 80 bytes (instead of 8MBytes).



**Figure 4.16** A personal memory allocator: An example of execution step by step. On the right-hand side the memory configuration is represented after each command reported on the left-hand side of the picture.

**Solution**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define MEM 8000
6
7 /* structure declaration */
8 typedef struct memory_s {
9     int ptr, size;
10    struct memory_s *next;
11 } memory_t;
12
13 /* function prototypes */
14 int myAlloc(memory_t *, int);
15 int myFree(memory_t *, int);
16 void display(memory_t *);
17 void quit(memory_t *);
18
19 /*
20  * main program
21 */
22 int main (void) {
23     memory_t *pTop=NULL, *pNew;
24     int size, ptr, val;
25     char cmd[20];
26
27     pNew = (memory_t *)malloc(sizeof(memory_t));
28     if (pNew==NULL) {
29         fprintf(stderr, "Memory allocation error.\n");
30         exit(EXIT_FAILURE);
31     }
32     pNew->size = 0;
33     pNew->ptr = MEM;
34     pNew->next = NULL;
35     pTop = (memory_t *)malloc(sizeof(memory_t));
36     if (pTop==NULL) {
37         fprintf(stderr, "Memory allocation error.\n");
38         exit(EXIT_FAILURE);
39     }
40     pTop->size = 0;
41     pTop->ptr = 0;
42     pTop->next = pNew;
43
44     do {
45         printf(stdout, "Input command: ");
46         scanf("%s", cmd);
47         if (strcmp(cmd, "malloc")==0 || (strcmp(cmd, "free") == 0)) {
48             scanf("%d", &val);
49             if (strcmp(cmd, "malloc") == 0) {
50                 ptr = myAlloc(pTop, val);
51                 if (ptr >= 0) {
52                     fprintf(stdout, "Allocation (size=%d) returns position %d\n", val, ptr);
53                 } else {
54                     fprintf(stdout, "Memory allocation was not possible!\n");
55                 }
56             } else {
57                 size = myFree(pTop, val);
58             }
59         }
60     }
61 }
```

```
58     if (size > 0) {
59         fprintf(stderr, "Memory block with size %d freed\n", size);
60     } else {
61         fprintf(stderr, "Memory block not found\n");
62     }
63 }
64 display(pTop);
65 } else if (strcmp(cmd, "stop") != 0) {
66     fprintf(stderr, "Error: unknown command\n");
67 }
68 } while (strcmp(cmd, "stop") != 0);
69
70 quit(pTop);
71 return EXIT_SUCCESS;
72 }
73
74 /*
75 * "allocate" a new memory block, if possible
76 */
77 int myAlloc (memory_t *pTop, int size) {
78     memory_t *pNew, *tmp=pTop;
79     int mem;
80
81     while (tmp->next != NULL) {
82         mem = tmp->next->ptr - (tmp->ptr+tmp->size);
83         if (size <= mem) {
84             pNew = (memory_t *)malloc(sizeof(memory_t));
85             if (pNew==NULL) {
86                 fprintf(stderr, "Memory allocation error.\n");
87                 exit(EXIT_FAILURE);
88             }
89             pNew->size = size;
90             pNew->ptr = tmp->ptr + tmp->size;
91             pNew->next = tmp->next;
92             tmp->next = pNew;
93             return pNew->ptr;
94         }
95         tmp = tmp->next;
96     }
97     return -1;
98 }
99
100 /*
101 * "free" a memory block
102 */
103 int myFree (memory_t *pTop, int ptr) {
104     memory_t *pOld, *tmp=pTop;
105     int size;
106
107     while (tmp->next != NULL) {
108         if (tmp->next->ptr == ptr) {
109             pOld = tmp->next;
110             tmp->next = pOld->next;
111             size = pOld->size;
112             free(pOld);
113             return size;
114         }
115         tmp = tmp->next;
116     }
```

```

117     }
118
119     /*
120      * show the memory usage
121     */
122     void display (memory_t *pTop) {
123         memory_t *tmp=pTop->next;
124
125         fprintf(stdout, "Memory usage:\n");
126         while (tmp->ptr != MEM) {
127             fprintf(stdout, "\t%d (%d)\n", tmp->ptr, tmp->size);
128             tmp = tmp->next;
129         }
130         fprintf(stdout, "\n");
131
132         return;
133     }
134
135
136     /*
137      * free the dynamically allocated memory
138     */
139     void quit (memory_t *pTop) {
140         memory_t *tmp;
141
142         while (pTop != NULL) {
143             tmp = pTop;
144             pTop = pTop->next;
145             free(tmp);
146         }
147
148         return;
149     }

```

## 4.16 Bi-linked Lists

### Specifications

A file contains data on a set of employees. For each employee there is a row of the file, including:

- ▷ Last and first name (a single C string, maximum 50 characters, e.g., Smith\_John).
- ▷ Personal identification (exactly 16 characters).
- ▷ Data of hiring (format *dd.mm.yyyy*, e.g, 30.05.2015).
- ▷ Salary (integer value, in euro).

Fields are space-separated. Employees do not appear in any specific order.

A C program receives 3 parameters on the command line:

- ▷ Input file name (the format is the previously defined one).
- ▷ Last and first name (single string, e.g., Clinton\_Bill).
- ▷ A string made of only + and - characters (e.g., +---+-+).

The program has to:

- ▷ Read the file.
- ▷ Store its content in a list where each new insertion is done on the list head. Each element of the list must have two pointers: One pointing right (ahead), and one

pointing left (backward). In other words, the list has to be bi-linked (see Section 4.6 for further details).

- ▷ Find in the list the employee whose name is passed on the command line as a second parameter.
- ▷ Move along the list in the right direction for each ‘+’ character, and to the left direction for each ‘-’ character in the third parameter. For each visited node of the list (included the first one) the program has to print out (on standard output) all data fields of the employee (with the same format this data appears in the original input file). If the end of the list is reached (either side) the program has to print-out the same element data repeatedly.

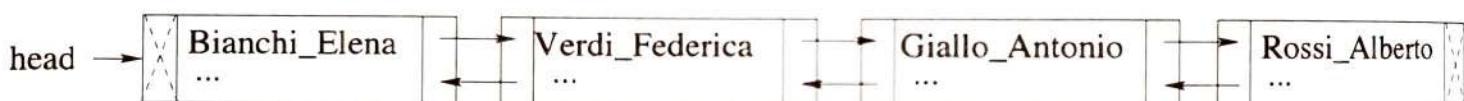
**Example 4.6** Let the command line parameters be the following:

file.txt Giallo\_Antonio ---+

and the file be the following:

```
Rossi_Alberto AAABBBCDDEFGGGH 03.12.1998 1845
Giallo_Antonio AAABBBCDDEFGGGH 13.11.2007 1140
Verdi_Federica AAABBBCDDEFGGGH 25.09.1989 2157
Bianchi_Elena AAABBBCDDEFGGGH 15.02.2004 1345
```

The file has to be stored in the list structure as:



**Figure 4.17** Bi-linked list.

Then, the program has to:

- ▷ Find Giallo\_Antonio in the list and print its data
- ▷ Move on Verdi and print its data (first ‘-’)
- ▷ Move left again and print Bianchi (second ‘-’)
- ▷ Do not move and print Bianchi again (third ‘-’)
- ▷ Move right and print Verdi (first ‘+’):

```
Giallo_Antonio AAABBBCDDEFGGGH 13.11.2007 1140
Verdi_Federica AAABBBCDDEFGGGH 25.09.1989 2157
Bianchi_Elena AAABBBCDDEFGGGH 15.02.2004 1345
Bianchi_Elena AAABBBCDDEFGGGH 15.02.2004 1345
Verdi_Federica AAABBBCDDEFGGGH 25.09.1989 2157
```

## Solution

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define MAX_R 255+1
6 #define MAX_C 16+1
7 #define MAX_D 10+1
8 #define MAX 21
```

```

9  typedef struct employee_s {
10    char *name;
11    char id[MAX_C];
12    char data[MAX_D];
13    int salary;
14    struct employee_s *left;
15    struct employee_s *right;
16 } employee_t;
17
18 employee_t * file_read (employee_t *, char []);
19 employee_t * insert (employee_t *, employee_t *);
20 void write (employee_t *, char *, char *);
21
22 int main (int argc, char * argv[]) {
23     employee_t *head, *tmp1, *tmp2;
24
25     if (argc!=4) {
26         fprintf(stderr,
27             "Run as: <pgrmName> <inFile> <name> <commands>\n");
28         exit(EXIT_FAILURE);
29     }
30
31     head = NULL;
32     head = file_read (head, argv[1]);
33     write (head, argv[2], argv[3]);
34
35     tmp1 = head;
36     while (tmp1!=NULL) {
37         free (tmp1->name);
38         tmp2 = tmp1->right;
39         free (tmp1);
40         tmp1 = tmp2;
41     }
42 }
43
44     return (EXIT_SUCCESS);
45 }
46
47 employee_t * file_read (employee_t *head, char *fileIn) {
48     FILE *input;
49     char riga[MAX_R], name[MAX];
50     employee_t *tmpPtr;
51
52     input=fopen(fileIn, "r");
53     if (input==NULL) {
54         fprintf(stderr, "Error opening file!\n");
55         return (head);
56     }
57
58     while (fgets(riga, MAX_R, input)!=NULL) {
59         tmpPtr = (employee_t *) malloc (sizeof (employee_t));
60         if (tmpPtr==NULL) {
61             fprintf(stderr, "Memory allocation error.\n");
62             exit(EXIT_FAILURE);
63         }
64         sscanf(riga, "%s %s %s %d",
65             name, tmpPtr->id, tmpPtr->data, &tmpPtr->salary);
66         tmpPtr->name = (char *) malloc ((strlen(name)+1)*sizeof(char));
67         if (tmpPtr->name==NULL) {

```

```

68     fprintf(stderr, "Memory allocation error.\n");
69     exit(EXIT_FAILURE);
70 }
71 sprintf (tmpPtr->name, "%s", name);
72 tmpPtr->right = head;
73 tmpPtr->left = NULL;
74 if (head!=NULL) {
75     head->left = tmpPtr;
76 }
77 head = tmpPtr;
78 }
79
80 fclose (input);
81
82 return (head);
83 }
84
85 void write (employee_t *headPtr, char *name, char *command) {
86 employee_t *tmpPtr;
87 int i;
88
89 for (tmpPtr=headPtr; tmpPtr!=NULL; tmpPtr=tmpPtr->right) {
90     if (strcmp(tmpPtr->name, name) == 0 ) {
91         break;
92     }
93 }
94
95 if (tmpPtr==NULL) {
96     return;
97 }
98
99 fprintf (stdout, "%s %s %s %d\n",
100        tmpPtr->name, tmpPtr->id, tmpPtr->data, tmpPtr->salary);
101
102 for (i=0; i<strlen(command); i++) {
103     if (command[i] == '+') {
104         if (tmpPtr->right!=NULL) {
105             tmpPtr = tmpPtr->right;
106         }
107     } else {
108         if (tmpPtr->left!=NULL) {
109             tmpPtr = tmpPtr->left;
110         }
111     }
112     fprintf (stdout, "%s %s %s %d\n",
113        tmpPtr->name, tmpPtr->id, tmpPtr->data, tmpPtr->salary);
114 }
115
116 return;
117 }

```

## 4.17 Polygons

### Specifications

A file stores 4 integer values on each row:

$x_1 \quad y_1 \quad x_2 \quad y_2$

describing a segment of extremes  $(x_1, y_1)$  and  $(x_2, y_2)$ . The length of the file is un-

known.

Write a program able to understand whether all segments form a close curve on the plane or not. The program has to receives two file names on the command line. The first one is the input file. The second one is the output file. If a close curve does exist the program has to store in the output file all segment in the right order to form that curve.

## Solution

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define LENGTH 100
5
6 /* structure declaration */
7 typedef struct segment_s {
8     float x1;
9     float y1;
10    float x2;
11    float y2;
12    struct segment_s *next;
13 } segment_t;
14
15 /* function prototypes */
16 segment_t *read(char *name);
17 int check(segment_t *head);
18 int sticky(segment_t *ptr1, segment_t *ptr2, int check_reverse);
19 void swap(segment_t *ptr1, segment_t *ptr2);
20 void write(char *name, segment_t *head);
21
22 /*
23  * main program
24  */
25 int main (void) {
26     segment_t *head;
27     char name[LENGTH];
28     int found;
29
30     fprintf(stdout, "Input file name: ");
31     scanf("%s", name);
32     head = read(name);
33
34     found = check(head);
35
36     fprintf(stdout, "Output file name: ");
37     scanf("%s", name);
38     if (found) {
39         write(name, head);
40     } else {
41         write(name, NULL);
42     }
43
44     return EXIT_SUCCESS;
45 }
46
47 /*
48  * read the input file contents
49 */

```

```
50 segment_t *read (char *name) {
51     FILE *fp;
52     segment_t *head=NULL, *tmp;
53     float x1, y1, x2, y2;
54
55     fp = fopen(name, "r");
56     head = NULL;
57     while (fscanf(fp, "%f %f %f %f", &x1, &y1, &x2, &y2) != EOF) {
58         tmp = (segment_t *)malloc(sizeof(segment_t));
59         if (tmp==NULL) {
60             fprintf(stderr, "Memory allocation error.\n");
61             exit(EXIT_FAILURE);
62         }
63         tmp->x1 = x1;
64         tmp->y1 = y1;
65         tmp->x2 = x2;
66         tmp->y2 = y2;
67         tmp->next = head;
68         head = tmp;
69     }
70     fclose (fp);
71
72     return head;
73 }
74
75 /*
76 * check whether the set of segments define a polygon
77 */
78 int check (segment_t *head) {
79     segment_t *tmp1, *tmp2, *ptr;
80     int stop, found;
81
82     stop = 0;
83     ptr = NULL;
84     tmp1 = head;
85     while (tmp1!=NULL && stop==0) {
86         stop = 1;
87
88         /* look for the next segment */
89         found = 0;
90         tmp2 = tmp1->next;
91         while (tmp2!=NULL && found==0) {
92             if (sticky(tmp1, tmp2, 1)) {
93                 found = 1;
94
95                 /* tie current segment to the previous one */
96                 swap(tmp1->next, tmp2);
97                 stop = 0;
98             }
99             tmp2 = tmp2->next;
100        }
101
102        ptr = tmp1;
103        tmp1 = tmp1->next;
104    }
105
106    /* check the reason for which the loop stopped ... */
107    if (tmp1 != NULL) {
108        /* not all segments have been tied: polygon NOT found */
```

```

109     return 0;
110 }
111 /* all segments tied: is last segment also sticking to first one? */
112 return sticky(ptr, head, 0);
113 }
114 }
115 /*
116 * check whether two segments have a common extreme
117 */
118 int sticky (segment_t *ptr1, segment_t *ptr2, int check_reverse) {
119     float x, y;
120
121     if (ptr1->x2==ptr2->x1 && ptr1->y2==ptr2->y1) {
122         return 1;
123     }
124
125     if (check_reverse && ptr1->x2==ptr2->x2 && ptr1->y2==ptr2->y2) {
126         /* swap the two extremes */
127         x = ptr2->x1;
128         y = ptr2->y1;
129         ptr2->x1 = ptr2->x2;
130         ptr2->y1 = ptr2->y2;
131         ptr2->x2 = x;
132         ptr2->y2 = y;
133
134         return 1;
135     }
136
137     return 0;
138 }
139 }
140
141 /*
142 * swap two elements of the list
143 */
144 void swap (segment_t *ptr1, segment_t *ptr2) {
145     segment_t tmp;
146
147     tmp = *ptr1;
148     *ptr1 = *ptr2;
149     ptr1->next = tmp.next;
150
151     tmp.next = ptr2->next;
152     *ptr2 = tmp;
153
154     return;
155 }
156
157 /*
158 * write the output file
159 */
160 void write (char *name, segment_t *head) {
161     FILE *fp;
162     segment_t *tmp;
163
164     fp = fopen(name, "w");
165     if (fp==NULL) {
166         fprintf (stderr, "Error opening file!\n");
167         exit(EXIT_FAILURE);

```

```

168     }
169
170     if (head != NULL) {
171         fprintf(fp, "Polygon found\n");
172         for (tmp=head; tmp!=NULL; tmp=tmp->next) {
173             fprintf(fp, "%.1f %.1f %.1f\n", tmp->x1, tmp->y1, tmp->x2, tmp->y2);
174         }
175     } else {
176         fprintf(fp, "Polygon NOT found\n");
177     }
178
179     fclose(fp);
180
181     return;
182 }
```

## 4.18 Lonely Hearts Club Specifications

Write an application to find the perfect match, i.e., the best possible partner, among members of a Lonely Hearts Club.

A text file specifies the characteristics and preferences of all club members, with the format specified on the left-hand side of the following picture. All fields are strings (with at most 100 characters) and M/F is a single character specifying the sex of the person. The one reported on the right-hand side of the following picture is a correct example.

#LastName1 FirstName1 M/F
characteristic1 value1
characteristic2 value2
...
#LastName2 FirstName2 M/F
...

#Alighieri Dante M
eye brown
height tall
hair brown
hair long
hobby poetry
#Mozart Wolfgang M
height short
hair brown
hobby music
hobby play
...

The application has to:

- ▷ Read, from standard input:
  - ◊ A single character M, F or T.
  - ◊ An undefined number of string couples In each couple the first string represent one characteristic, and the second one its value (e.g., “hobby” and “cooking”).

For example, the following is a possible correct input:

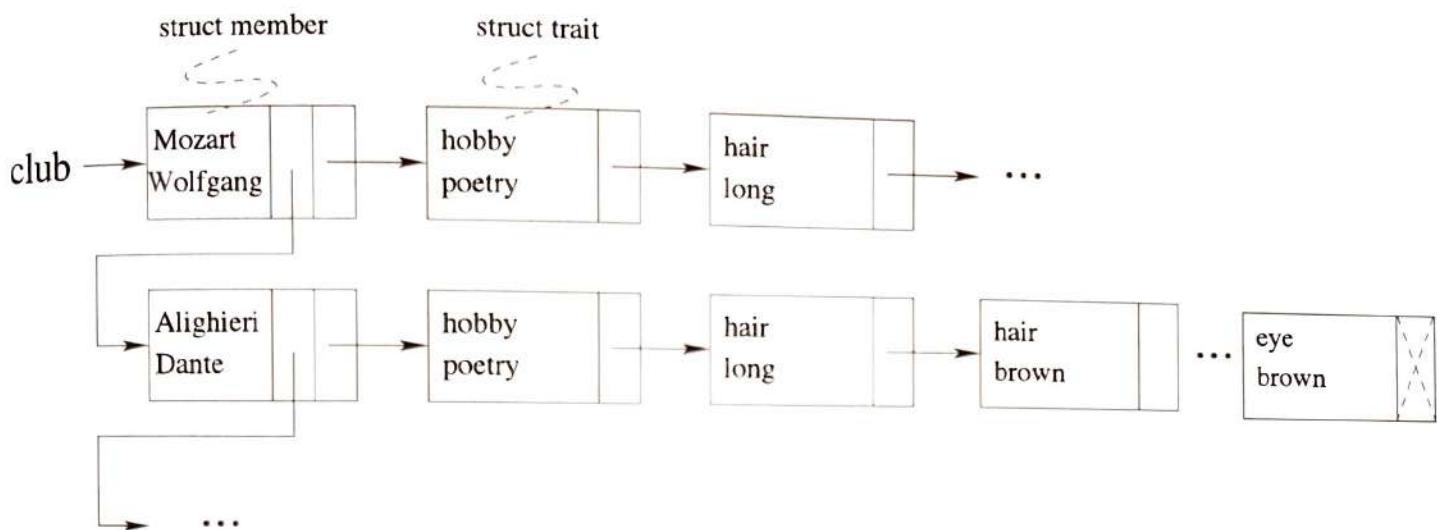
F height average eye brown hobby cooking

meaning that we look-for a mate for a female of average age, with brown eyes, who loves cooking. A single character T terminates the program.

- 221
- ▷ Search the club members to find the best possible match, i.e., the club member of the opposite sex with the highest number of couples characteristic-value equal to the one introduced.
  - ▷ Print-out the name of this person.

### Solution

Referring to the example reported in the text, Figure 4.18 illustrates the data structure created by the following C code. Notice that the program manages all lists (both the main and all secondary ones) with head insertions. This explains the order of the objects within each list, compared to the one in which they are stored within the input file.



**Figure 4.18** A (vertical, main) list of members pointing to (horizontal, secondary) lists of characteristic and value couples.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define MAX 101
6
7 /* structure declaration */
8 typedef struct trait trait_t;
9 typedef struct member member_t;
10
11 struct trait {
12     char *id;
13     trait_t *next;
14 };
15
16 struct member {
17     char *name;
18     char sex;
19     trait_t *features;
20     member_t *next;
21 };
22
```

```
23 /* function prototypes */
24 member_t *load(char *filename);
25 trait_t *read(char *sex_ptr);
26 void process(member_t *club, char sex, trait_t *features);
27 int check(trait_t *head1, trait_t *head2);
28 void destroy(trait_t *head);
29 void quit(member_t *head);
30
31 /*
32  * main program
33 */
34 int main (void) {
35     member_t *club=NULL;
36     trait_t *features=NULL;
37     char name[MAX], sex;
38
39     fprintf(stdout, "Input the club file name: ");
40     scanf("%s", name);
41     club = load(name);
42
43     do {
44         features = read(&sex);
45         if (features) {
46             process(club, sex, features);
47             destroy(features);
48         }
49     } while (features != NULL);
50
51     quit(club);
52
53     return EXIT_SUCCESS;
54 }
55
56 /*
57  * load the info for the club members
58 */
59 member_t *load (char *filename) {
60     char lastName[MAX], firstName[MAX], line[MAX], name[MAX], sex[MAX];
61     member_t *member, *head=NULL;
62     trait_t *feature;
63     FILE *fp;
64
65     fp = fopen(filename, "r");
66     while (fgets(line, MAX, fp) != NULL) {
67         if (line[0] == '#') {
68             member = (member_t *)malloc(sizeof(member_t));
69             if (member==NULL) {
70                 fprintf(stderr, "Memory allocation error.\n");
71                 exit(EXIT_FAILURE);
72             }
73             sscanf(&line[1], "%s %s %s", lastName, firstName, sex);
74             sprintf(name, "%s %s", lastName, firstName);
75             member->name = strdup(name);
76             member->features = NULL;
77             member->sex = sex[0];
78             member->next = head;
79             head = member;
80         } else {
81             feature = (trait_t *)malloc(sizeof(trait_t));
```

```

82     if (feature==NULL) {
83         fprintf(stderr, "Memory allocation error.\n");
84         exit(EXIT_FAILURE);
85     }
86     sscanf(line, "%s %s", lastName, firstName);
87     sprintf(name, "%s %s", lastName, firstName);
88     feature->id = strdup(name);
89     feature->next = member->features;
90     member->features = feature;
91 }
92 }
93 fclose(fp);
94
95 return head;
96 }
97
98 /*
99  * read a user's query (list of features)
100 */
101 trait_t *read (char *sex_ptr) {
102     char line[MAX], name[MAX], value[MAX], id[MAX], sex, *ptr;;
103     trait_t *trait, *features=NULL;
104     int n;
105
106     fprintf(stdout, "Input query: ");
107     fgets(line, MAX, stdin);
108     sscanf(line, "%c", &sex);
109
110    if (sex == 'T') {
111        /* termination */
112        return NULL;
113    }
114
115    ptr = &line[1];
116    // %n writes into variable n the number of characters that have been read
117    while (sscanf(ptr, "%s %s%n", name, value, &n) != EOF) {
118        trait = (trait_t *)malloc(sizeof(trait_t));
119        if (trait==NULL){
120            fprintf(stdout, "Memory allocation error.\n");
121            exit(EXIT_FAILURE);
122        }
123        sprintf(id, "%s %s", name, value);
124        trait->id = strdup(id);
125        trait->next = features;
126        features = trait;
127        ptr += n;
128    }
129    *sex_ptr = sex;
130
131    return features;
132 }
133
134 /*
135  * search the club member having most traits in the given set
136  */
137 void process (member_t *club, char sex, trait_t *features) {
138     member_t *mate=NULL, *ptr=club;
139     int equal, max=0;
140

```

```
141     while (ptr != NULL) {
142         if (ptr->sex == sex) {
143             equal = check(ptr->features, features);
144             if (mate==NULL || equal>max) {
145                 mate = ptr;
146                 max = equal;
147             }
148         }
149         ptr = ptr->next;
150     }
151     fprintf(stdout, "Member found: %s (score %d)\n", mate->name, max);
152
153     return;
154 }
155
156 /**
157 * counts the number of common features
158 */
159 int check (trait_t *head1, trait_t *head2) {
160     trait_t *ptr1, *ptr2;
161     int found, equal=0;
162
163     ptr1 = head1;
164     while (ptr1 != NULL) {
165         ptr2 = head2;
166         found = 0;
167         while (ptr2!=NULL && !found) {
168             if (strcmp(ptr1->id, ptr2->id) == 0) {
169                 found = 1;
170             }
171             ptr2 = ptr2->next;
172         }
173         equal += found;
174         ptr1 = ptr1->next;
175     }
176     return equal;
177 }
178
179 /**
180 * free the memory allocated for a trait list
181 */
182 void destroy (trait_t *head) {
183     trait_t *tmp;
184
185     while (head != NULL) {
186         tmp = head;
187         head = tmp->next;
188         free(tmp->id);
189         free(tmp);
190     }
191 }
192
193 /**
194 * free the memory allocated for the entire club
195 */
196 void quit (member_t *head) {
197     member_t *tmp;
198
199     while (head != NULL) {
```

```

200     tmp = head;
201     head = tmp->next;
202     free(tmp->name);
203     destroy(tmp->features);
204     free(tmp);
205 }
206 }
```

## 4.19 Video club

### Specifications

All members and the entire activity of a video-club are stored into three separate files. Those files store the member list, the film list, and all films rent in a specific period of time, respectively. More in details:

- ▷ The “member” file stores on each row:

name profession age

where name is a string of 30 characters at most, profession is a string of 20 characters at most, and age is an integer with two decimal digits.

- ▷ The “film” file stores on each row:

title length

where title is a string of 40 characters at most, and length is the length of the film in minutes.

- ▷ The “rent” file stores on each row:

name title

which defines that the film title has been rent by the member name. Obviously, the same user may have rent more films, and the same films may have been rent by several members.

The program must read all files and, using a proper data structure, it must list all available films, and indicates for each one: The number of members that has rent it, the average age of the members who rent it, their profession, and for each profession the number of members having that profession.

Notice that it is possible to suppose that all film titles and all member names are distinct.

**Example 4.7** The following are correct input files:

**Member File**

John_D business 42
Jane_D butcher 17
Hunt_E retailer 29
...

**Film File**

The_Avengers 120
Terminator 100
Mission_Impossible 123
The_Accountant 130
...

**Rent File**

Jane_D Terminator
Hunt_E Mission_Impossible
...

The following is a possible output of the program:

The\_Avengers 120 4 42 -> retailer 1 lawyer 2 retailer 1

```

Terminator      100 30 12 -> craftsman 24
Mission_Impossible 123 14 55 -> housewife 12 constructionWorker 8
...

```

where for each row we reported the film title, its length, the number of members who rent it, their age, and their profession list.

## Solution

The following solution uses two data structures to solve the problem:

- ▷ A simple list. This list stores all user's data, and it is based on elements of `user` type.
- ▷ A list of lists. The main list is based on elements `movie`, and it stores all data concerning each film plus the pointer to each secondary list. The secondary list is based on elements `job`, and it stores the professions of all members who rent the film specified by the corresponding element of the primary list.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define MAX 50
6
7 /* structure declaration */
8 typedef struct user user_t;
9 typedef struct movie movie_t;
10 typedef struct job job_t;
11
12 struct user {
13     char *name;
14     char *activity;
15     int age;
16     user_t *next;
17 };
18
19 struct job {
20     char *activity;
21     int n;
22     job_t *next;
23 };
24
25 struct movie {
26     char *title;
27     int duration, age, n;
28     job_t *jobs;
29     movie_t *next;
30 };
31
32 /* function prototypes */
33 user_t *read_users(void);
34 movie_t *read_movies(void);
35 void read_rentals(movie_t *, user_t *);
36 user_t *find_user(user_t *users, char *name);
37 movie_t *find_movie(movie_t *movies, char *title);
38 void display(movie_t *);
39 void quit(movie_t *, user_t *);
40

```

```
41 /*  
42 * main program  
43 */  
44 int main (void) {  
45     user_t *users;  
46     movie_t *movies;  
47  
48     users = read_users();  
49     movies = read_movies();  
50     read_rentals(movies, users);  
51     display(movies);  
52     quit(movies, users);  
53     return 0;  
54 }  
55  
56 /*  
57 * read the users file  
58 */  
59 user_t *read_users (void) {  
60     user_t *tmp, *head=NULL;  
61     char buffer[MAX];  
62     FILE *fp;  
63  
64     fprintf(stdout, "Input the users file name: ");  
65     scanf("%s", buffer);  
66     fp = fopen(buffer, "r");  
67     while (fscanf(fp, "%s", buffer) != EOF) {  
68         tmp = (user_t *)malloc(sizeof(user_t));  
69         if (tmp==NULL) {  
70             fprintf (stderr, "Memory allocation error.\n");  
71             exit(EXIT_FAILURE);  
72         }  
73         tmp->name = strdup(buffer);  
74         fscanf(fp, "%s", buffer);  
75         tmp->activity = strdup(buffer);  
76         fscanf(fp, "%d ", &tmp->age);  
77         tmp->next = head;  
78         head = tmp;  
79     }  
80     fclose(fp);  
81  
82     return head;  
83 }  
84  
85 /*  
86 * read the movies file  
87 */  
88 movie_t *read_movies (void) {  
89     movie_t *tmp, *head=NULL;  
90     char buffer[MAX];  
91     FILE *fp;  
92  
93     fprintf(stdout, "Input the movies file name: ");  
94     scanf("%s", buffer);  
95     fp = fopen(buffer, "r");  
96     while (fscanf(fp, "%s", buffer) != EOF) {  
97         tmp = (movie_t *)malloc(sizeof(movie_t));  
98         if (tmp==NULL){  
99             fprintf (stderr, "Memory allocation error.\n");
```

```

100     exit(EXIT_FAILURE);
101 }
102 tmp->title = strdup(buffer);
103 fscanf(fp, "%d", &tmp->duration);
104 tmp->age = tmp->n = 0;
105 tmp->jobs = NULL;
106 tmp->next = head;
107 head = tmp;
108 }
109 fclose(fp);
110
111 return head;
112 }
113
114 /*
115 *   read the rentals file
116 */
117 void read_rentals (movie_t *movies, user_t *users) {
118     char buffer[MAX];
119     movie_t *movie;
120     user_t *user;
121     job_t *job;
122     int found;
123     FILE *fp;
124
125     fprintf(stdout, "Input the rentals file name: ");
126     scanf("%s", buffer);
127     fp = fopen(buffer, "r");
128     while (fscanf(fp, "%s", buffer) != EOF) {
129         user = find_user(users, buffer);
130         fscanf(fp, "%s", buffer);
131         movie = find_movie(movies, buffer);
132         movie->age += user->age;
133         movie->n++;
134
135         job = movie->jobs;
136         found = 0;
137         while (job != NULL) {
138             if (strcmp(job->activity, user->activity) == 0) {
139                 job->n++;
140                 found = 1;
141             }
142             job = job->next;
143         }
144         if (found == 0) {
145             job = (job_t *)malloc(sizeof(job_t));
146             if (job==NULL){
147                 fprintf (stderr, "Memory allocation error.\n");
148                 exit(EXIT_FAILURE);
149             }
150             job->activity = strdup(user->activity);
151             job->n = 1;
152             job->next = movie->jobs;
153             movie->jobs = job;
154         }
155     }
156     fclose(fp);
157 }
158

```

```
159  /*
160   * search a user given the name
161   */
162 user_t *find_user (user_t *users, char *name) {
163     user_t *user;
164
165     user = users;
166     while (user != NULL) {
167       if (strcmp(user->name, name) == 0) {
168         return user;
169       }
170       user = user->next;
171     }
172     return NULL;
173   }
174
175 /*
176  * search a movie given the title
177 */
178 movie_t *find_movie (movie_t *movies, char *title) {
179   movie_t *movie;
180
181   movie = movies;
182   while (movie != NULL) {
183     if (strcmp(movie->title, title) == 0) {
184       return movie;
185     }
186     movie = movie->next;
187   }
188   return NULL;
189 }
190
191 /*
192  * display the required output
193 */
194 void display (movie_t *movies) {
195   movie_t *movie;
196   job_t *job;
197   int avg;
198
199   for (movie=movies; movie!=NULL; movie=movie->next) {
200     if (movie->n == 0) {
201       avg = 0;
202     } else {
203       avg = movie->age / movie->n;
204     }
205     fprintf(stdout, "%s %d %d %d\n", movie->title, movie->duration,
206             movie->n, avg);
207
208     for (job=movie->jobs; job!=NULL; job=job->next) {
209       fprintf(stdout, "%s %d ", job->activity, job->n);
210     }
211     fprintf(stdout, "\n");
212   }
213 }
214
215 /*
216  * free all the dynamically allocated memory
217 */
```

```
218 void quit (movie_t *movies, user_t *users) {
219     movie_t *movie;
220     user_t *user;
221     job_t *job;
222
223     while (users != NULL) {
224         user = users;
225         users = user->next;
226         free(user->name);
227         free(user->activity);
228         free(user);
229     }
230
231     while (movies != NULL) {
232         movie = movies;
233         movies = movie->next;
234         free(movie->title);
235         while (movie->jobs != NULL) {
236             job = movie->jobs;
237             movie->jobs = job->next;
238             free(job->activity);
239             free(job);
240         }
241         free(movie);
242     }
243
244     return;
245 }
```

# Chapter 5

## Recursion basis and simple recursive problems

Recursion is a fundamental concept used in a variety of disciplines ranging from linguistics to logic. Our primary purpose in this chapter is to examine recursion as a practical tool. As we shall see, many interesting algorithms are quite simply expressed with recursive programs, and many algorithm designers prefer to express methods recursively.

### 5.1 Recursion: Basic Concepts

Informally, recursion is the process a procedure goes through when one of the steps of the procedure involves invoking the procedure itself. A procedure that goes through recursion is said to be “recursive”. In other words, recursion occurs when a thing is defined in terms of itself or of its type.

In mathematics and computer science, a recursive function is one that calls itself and it is defined in term of itself. In C language a recursive function `recur` appears as:

```
<type> recur (<parameters>) {  
    ...  
    recur (<parameters>);  
    ...  
    return (<type>);  
}
```

which is also called *direct* recursion.

To understand recursion, one must recognize the distinction between a procedure and the running of a procedure. A procedure is a set of steps based on a set of rules. The running of a procedure involves actually following the rules and performing the steps. As an analogy, a procedure is like a written recipe, whereas running a procedure is like actually preparing the meal.

A recursive function can't call itself always, or it would never stop. Then, another essential ingredient is that there must be a *termination condition*, such that the function can cease to call itself and no loop or infinite chain of references can occur. The previous C function would then appear as:

```

<type> recur (<parameters>) {
    if (<termination condition is true>) {
        // End recursive process
        return (<type>);
    }
    // Otherwise go-on ...
    ...
    recur (<parameters>);
    ...
    return (<type>);
}

```

Very often the termination condition is a more or less complex function of the number of times the recursive function has already been called. In other words, the termination condition is a function of the recursion depth. In the simplest case, we can then use a sort of counter to check for the termination condition. The following example uses an up-counter:

```

<type> recur (<parameters>, int recursion_depth) {
    if (recursion_depth >= N) {
        // End recursive process
        return (<type>);
    }
    // Otherwise go-on ...
    recursion_depth++;
    ...
    recur (<parameters>, recursion_depth);
    ...
    return (<type>);
}

```

where  $N$  is the depth of the recursion, and `recursion_depth` starts from 0. On the contrary, the following piece of code uses a down-counter:

```

<type> recur (<parameters>, int recursion_depth) {
    if (recursion_depth <= 0) {
        // End recursive process
        return (<type>);
    }
    // Otherwise go-on ...
    recursion_depth--;
    ...
    recur (<parameters>);
    ...
    return (<type>);
}

```

where `recursion_depth` is initialize to the depth of the recursion.

### 5.1.1 Recurrences

Recursive definitions of functions are quite common in mathematics. The simplest type, involving integer argument, are called *recurrence relations*. A recurrence relations can be defined by two properties:

- ▷ A simple base case (or cases), i.e., a terminating scenario that does not use recursion to produce an answer.
- ▷ A set of rules that reduce all other cases toward the base case.

Many mathematical axioms are based upon recursive rules. The following examples show a few of them.

**Example 5.1** For example, the following is a recursive definition of a person's ancestors:

- ▷ One's parents are one's ancestors (base case).
- ▷ The ancestors of one's ancestors are also one's ancestors (recursion step).

**Example 5.2** The formal definition of the natural numbers by the Peano axioms can be described as:

- ▷ 0 is a natural number.
- ▷ Each natural number has a successor, which is also a natural number.

By the base case and recursive rule, one can generate the set of all natural numbers.

**Example 5.3** One of the most familiar recurrences is the factorial function, defined by the formula:

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \quad \text{for } n \geq 1 \end{aligned}$$

Many data structures introduce in programming techniques can be defined as recursive objects. These includes lists as well as trees.

## 5.2 Mutual Recursion

In mathematics and computer science, *mutual* recursion is a form of recursion where two mathematical or computational objects, such as functions or data types, are defined in terms of each other. Mutual recursion is very common in functional programming and in some problem domains, such as recursive descent parsers, where the data types are naturally mutually recursive, but it is uncommon in other domains.

In C two recursive functions `recurA` and `recurB` mutually recursive will act as in Figure 5.1:

### Recursive Function A

```
<type> recurA (<parameters>) {
    ...
    if (<termination condition>) {
        // End recursive process
        return (<type>);
    }
    // Otherwise go-on ...
    ...
    recurB (<parameters>);
    ...
    return (<type>);
}
```

### Recursive Function B

```
<type> recurB (<parameters>) {
    ...
    if (<termination condition>) {
        // End recursive process
        return (<type>);
    }
    // Otherwise go-on ...
    ...
    recurA (<parameters>);
    ...
    return (<type>);
}
```

**Figure 5.1** Mutual recursion between function `recurA` and `recurB`.

Extensions to more than two functions are somehow trivial.

### 5.3 Recursion Tree and Stack

Any function can be recursive, indeed even the main program. Run for example the following program:

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

int main () {
    fprintf (stdout, "The universe is never ending.\n");
    main ();
}

return EXIT_SUCCESS;
```

Theoretically it generates an infinite number of recursive calls, each one printing the message “The universe is never ending.”. Indeed the main does not contain any termination condition, and it is suppose to call itself forever. In practice, how long the recursive process will go on, once the program has been run?

At this point it is important to introduce two important concepts for recursion: The recursion tree and the system stack.

Let's us analyze those two concept on another simple example, this time with a termination condition, before moving back to the previous infinite recursion.

In the following piece of code function `recur` is called with  $n = 3$ . The first instance of the function checks whether  $n > 0$ , and as this is true (as we said  $n = 3$ ) the function calls itself with  $n = 2$ . At this point, a new instance of `recur` is generated with  $n = 2$ . As the previous instance, also this one will check whether  $n > 0$ , and it will call one more time itself. This process goes on until  $n \leq 0$ .

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

void recur (int);

int main () {
    recur (3);

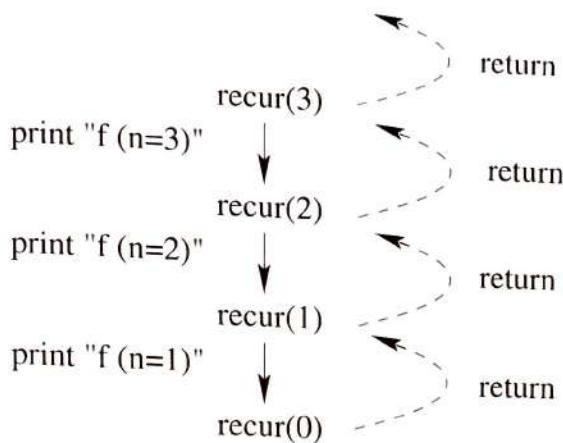
    return EXIT_SUCCESS;
}

void recur (int n) {
    if (n<=0) {
        return;
    }
    fprintf (stdout, "f (n=%d)\n", n);
    recur (n-1);

    return;
}
```

A *recursion tree* is a tree which can be used to represent the recursive process with all its subsequent calls. Figure 5.2 shows the recursion tree for function `recur`.

As far as the system *stack* is concerned, this is a piece of the operating system memory that we will encounter and discussed several times within this book. As a matter of fact, the reader should refer to Sections 8.2.2 and Section 9.1 for further and



**Figure 5.2** Recursion tree for function `recur`. Each recursive call is represented by a new node of the tree, which is evolving downward. Dotted lines represent the `return` statement in action, when the process is terminating one recursive call at a time.

more complete details on the system stack and the LIFO logic. In this section we will just highlight the use of the stack *only* within recursive function calls. The stack is a *Last In First Out (LIFO)* structure, i.e., a sort of pile of objects where each new object is inserted on top of the pile and any old object is extracted from the same top. Indeed, both insertions and extractions are performed on the same side of the structure (the Top of Stack or *TOS*). For our recursive analysis “objects” are program or function “frames”. Each frame usually includes at least:

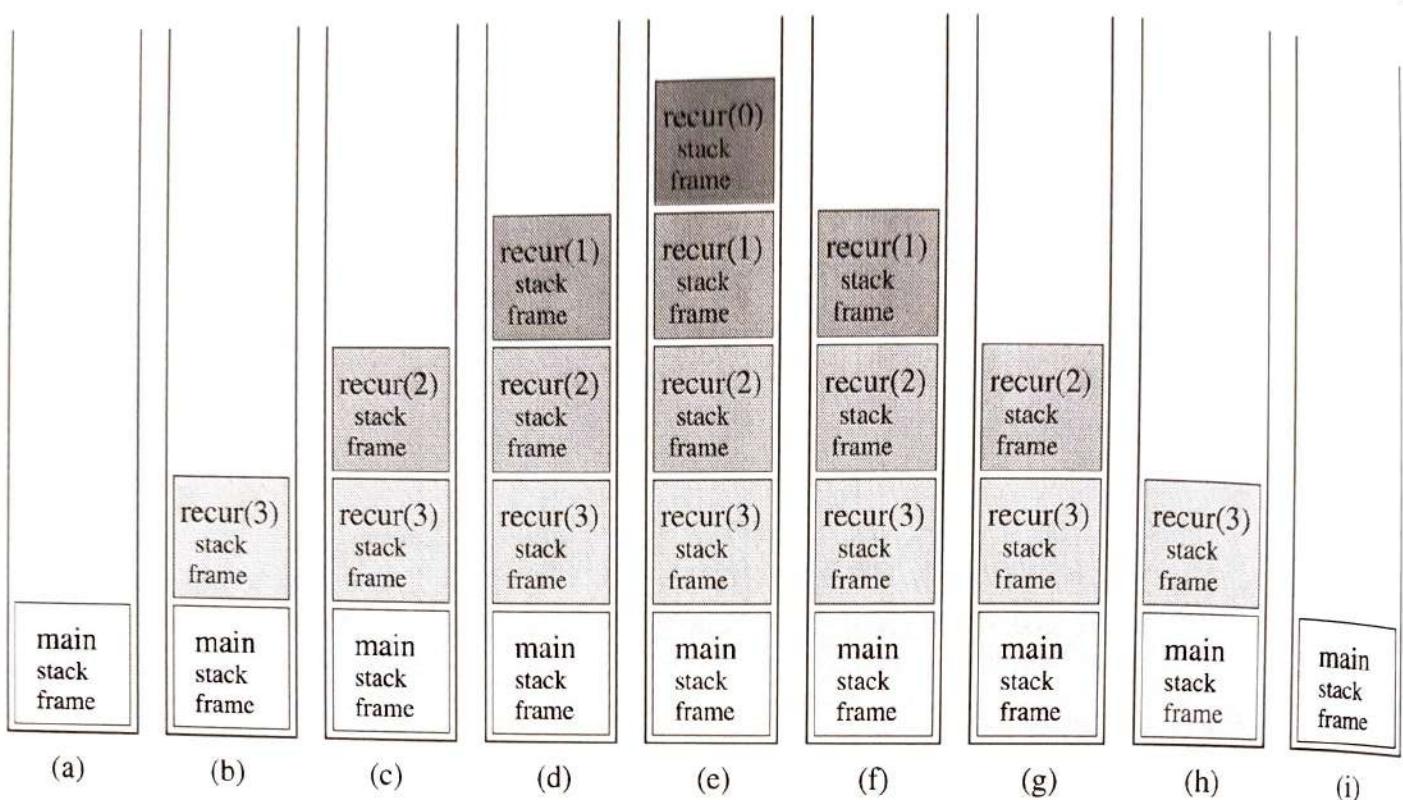
- ▷ All formal parameters.
- ▷ All locally defined variables.
- ▷ The reference (address) to the code of the function, which has to be executed once the function is called.
- ▷ The memory address to which the function is suppose to give control once it has done,

When the program starts the stack includes only the main frame (see Figure 5.3(a)). For each new call a new frame is inserted on the TOS (see Figure 5.3 from (b) to (e)). Whereas for each return the more recently inserted frame is removed from the TOS (see Figure 5.3 from (f) to (i)). In this way, any environment is saved before recurring, and it can be restored once the control is given back from the called function to the caller function.

Now, we can go back to the recursive main program, and to the question “how long the recursive process will go on?”. Usually, it will go on until the stack reserved for the program is full. When this happen (try to run the suggested program) the user usually receives an error such as “Segmentation fault (core dumped)”.

## 5.4 Recursion versus Iteration

Recursion has many negatives. It repeatedly invokes the mechanism, and consequently the overhead, of function calls. This can be expensive in both processor time and memory space. As we have seen in the previous section, each recursive call causes another



**Figure 5.3** Evolution (from (a) to (i)) of the system stack during the execution of function `recur`. Each new call to the function adds a new stack frame on the top of the stack. Lighter elements have been inserted before darker ones. The extraction order will be the opposite (from darker to lighter objects) as the next return address must be within the last caller function.

copy of the function frame to be created on the stack; this can consume considerable memory. Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted.

Any problem that can be solved recursively can also be solved iteratively (non-recursively). This can easily be proved. In fact, recursion is allowed by the system stack (see Section 5.3) which stores a stack frame for each new called function. However, the system stack can always be substituted by a similar user defined stack, on which the user can explicitly perform the same operations the operating system performs on the system stack.

A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem, and results in a program that's easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution may not be apparent.

## 5.5 Divide and Conquer

Most of the recursive programs we consider in this book use two recursive calls, each operating on about half the input. This is the so-called “divide-and-conquer” paradigm for algorithm design, which is often used to achieve significant economies. In general, divide-and-conquer algorithms involve doing some work to split the input into two pieces, or to merge the results of processing two independent “solved” portions of the input, or to help things along after half of the input has been processed. This means,

that there may be code before, after, or in between the two recursive calls. We will see many examples of such algorithms later in this one and the following chapters.

Notice that, we will also encounter algorithms in which it is not possible to follow the divide-and-conquer regimen completely, because the input is divided into unequal pieces or into more than two pieces, or there is some overlap among the pieces.

## 5.6 Factorial computation

### Specifications

Write a program, which calling a recursive function, is able to compute  $n!$  given the integer (positive) value  $n$ .

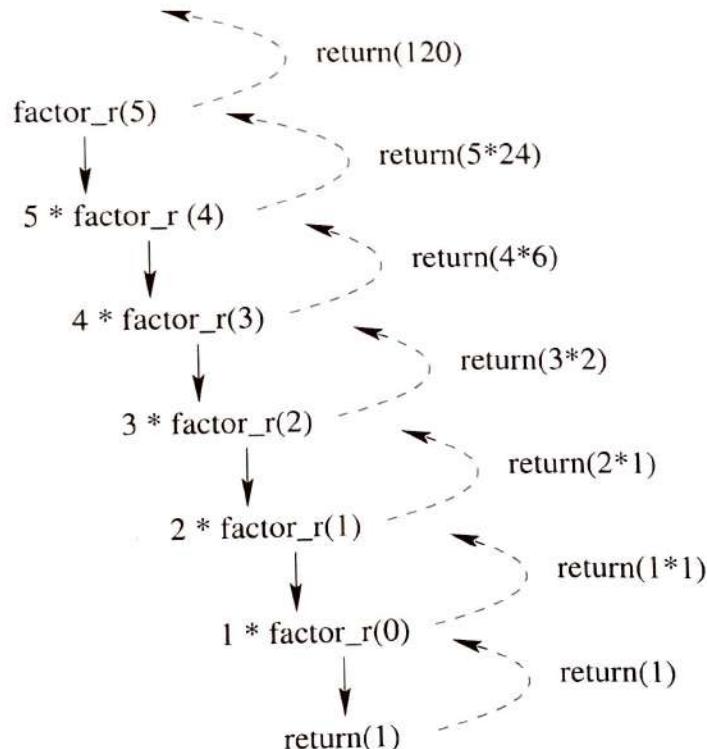
Let us remind the recursive definition of factorial numbers:

$$\begin{aligned} 0! &= 1 \\ n! &= 1 \cdot 2 \cdot 3 \cdots n \end{aligned}$$

### Solution

On the one hand, the following program illustrates the basic features of a recursive function. Function `factor_r` calls itself (with a smaller value of its argument), and it has a termination condition in which it directly computes its result.

The next figure shows the recursion tree generated by a call to function `factor_r` with a parameter equal to  $n = 5$ .



On the other hand, there is no masking the fact that function `factor_r` is nothing more than an iteration, like the following one:

```

int factor (int n) {
    int n;

    fact = 1;
    for (i=1; i<=n; i++) {
        fact = fact * i;
    }
    return (fact);
}

```

so it is hardly a convincing example of the power of recursion.

Finally, it is important to notice that the program does not work for negative value of  $n$  as it results in an infinite recursive loop. This is in fact a common bug that can appear in more subtle forms in more complicated recursive programs. However, in this case the problem can easily be fixed by modifying the termination condition as if  $(n >= 0)$ .

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
5 int factor_r (int n);
6
7 /*
8 * main program
9 */
10 int main (void) {
11     int n;
12
13     fprintf(stdout, "Number (>= 0): ");
14     scanf("%d", &n);
15
16     fprintf(stdout, "Factorial %d! = %d\n", n, factor_r(n));
17     return EXIT_SUCCESS;
18 }
19
20 /*
21 * factorial computation, recursive function
22 */
23 int factor_r (int n) {
24     if (n == 0) {
25         return 1;
26     }
27
28     return n*factor_r(n-1);
29 }

```

## 5.7 Fibonacci computation

### Specifications

Write a program able to:

- ▷ Read an integer number  $n$ .
- ▷ Compute and print-out the Fibonacci number of position  $n$ .

We recall that the Fibonacci numbers are the following:

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad \dots$$

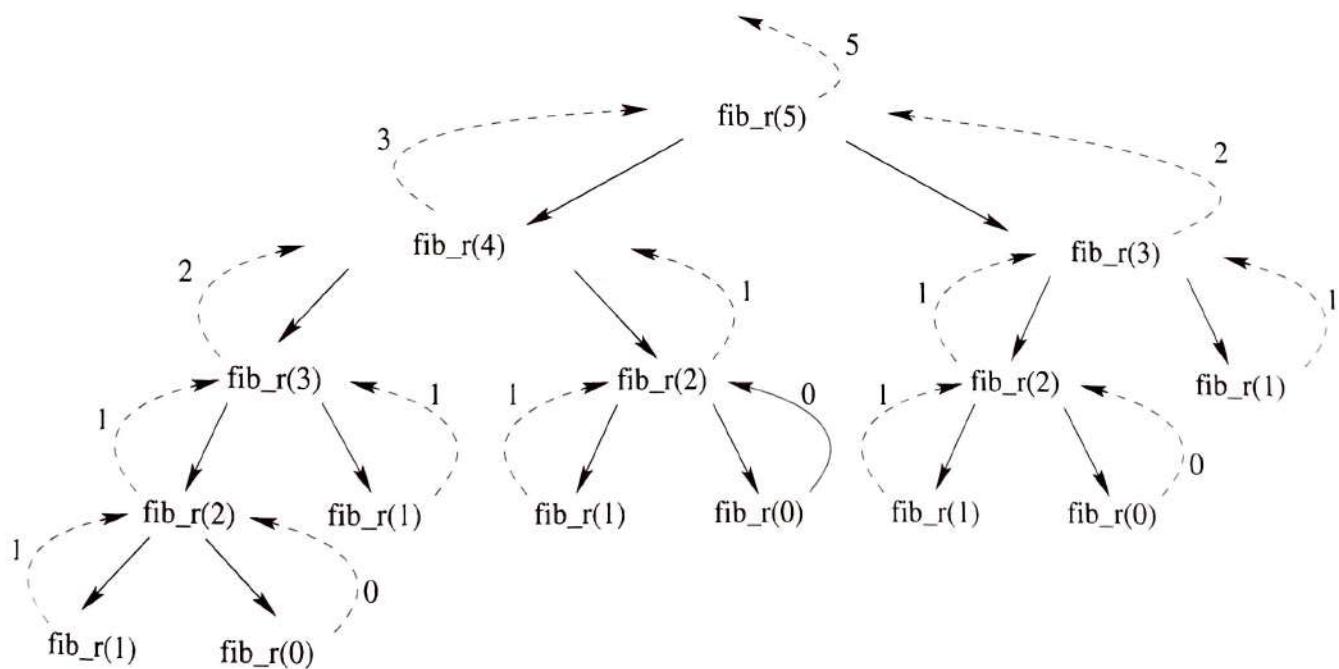
that is

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \quad \text{for } n \geq 2 \end{aligned}$$

### Solution 1

The recurrence corresponds directly to a simple recursive program. This is even less convincing example of the “power” of recursion. Indeed, it is a convincing example that recursion should not be used blindly, or dramatic inefficiencies can result.

The next figure shows the recursion tree generated by a call to function `fibonacci_r` (named `fib_r` in the picture) with a parameter equal to  $n = 5$ . As it can be noticed calling `fibonacci_r(5)` generates 15 subsequent calls to the same functions. Calling `fibonacci_r(25)` will generate 242785 calls to the same functions (to generate value number 75025).



The problem here is that the recursive calls indicate that  $\text{fib}_r(n-1)$  and  $\text{fib}_r(n-2)$  should be computed independently, when, in fact, one certainly would use  $\text{fib}_r(n-2)$  (and  $\text{fib}_r(n-3)$ ) to compute  $\text{fib}_r(n-1)$ . The number of calls needed to compute  $\text{fib}_r(n)$  is the number of calls needed to compute  $\text{fib}_r(n-1)$  plus the number of calls needed to compute  $\text{fib}_r(n-2)$  unless  $n = 0$  or  $n = 1$ , when only one call is needed. This fits the recurrence relation defining the Fibonacci numbers, as the number of calls to compute  $\text{fib}_r(n)$  is exactly  $\text{fib}_r(n)$ , and it can be proved to be about  $1.618^n$ . In other words, the program is an exponential-time algorithm. By contrast, it is very easy to compute  $\text{fib}_r(n)$  in linear time, using an iterative procedure such as:

```

int fibonacci (int n) {
    int i, n1, n2, n3;

    n1 = 0,
    n2 = 1;
    for (i=1; i<=n; i++) {
        n3 = n1 + n2;
        n1 = n2;
        n2 = n3;
    }
    return (n1);
}

```

In any case, the following program includes the recursive computation.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
5 int fibonacci_r (int n);
6
7 /*
8  * main program
9  */
10 int main (void) {
11     int n;
12
13     fprintf(stdout, "Input n: ");
14     scanf("%d", &n);
15     fprintf(stdout, "Fibonacci %d-th term = %d\n", n, fibonacci_r(n));
16
17     return EXIT_SUCCESS;
18 }
19
20 /*
21  * computation of the n-th Fibonacci term, recursive function
22 */
23 int fibonacci_r (int n) {
24     if (n==0 || n==1) {
25         return n;
26     }
27
28     return fibonacci_r(n-1)+fibonacci_r(n-2);
29 }

```

## Solution 2

This second version of the program stores all Fibonacci numbers in a dynamic array named known. This technique of using an array to store previous results is typically the method of choice for evaluating recurrence relations, for it allows rather complex equations to be handled in a uniform and efficient manner. In this case this method avoid re-computations of previously computed values.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
5 int fibonacci_r (int n, int *known);
6

```

```

7  /*
8   * main program
9   */
10 int main (void) {
11     int *known, i, n;
12
13     fprintf(stdout, "Input n: ");
14     scanf("%d", &n);
15     known = (int *) malloc ((n+1)*sizeof(int));
16     if (known==NULL) {
17         fprintf (stderr, "Memory allocation error.\n");
18         exit(EXIT_FAILURE);
19     }
20     for (i=0; i<=n; i++) {
21         known[i] = -1;
22     }
23     fprintf(stdout, "Fibonacci %d-th term = %d\n", n, fibonacci_r(n, known));
24     free(known);
25
26     return EXIT_SUCCESS;
27 }
28
29 /*
30  * computation of the n-th Fibonacci term, recursive function
31 */
32 int fibonacci_r (int n, int *known) {
33     if (known[n] < 0) {
34         if (n==0 || n==1) {
35             known[n] = n;
36         } else {
37             known[n] = fibonacci_r(n-1, known)+fibonacci_r(n-2, known);
38         }
39     }
40     return known[n];
41 }
42 }
```

## 5.8 Power Computation

### Specifications

Write a program able to:

- ▷ Read a real value  $x$ , and an integer positive value  $n$ .
- ▷ Compute and print-out  $x^n$ .

The power has to be computed as:

$$x^n = x \cdot x \cdot x \cdot x \dots x \quad (n \text{ times})$$

without using any iterative construct or mathematical function.

### Solution

The iterative code would compute the value as:

```
int power (float x, int n) {
```

```

float p;
int i;

p = 1.0;
for (i=1; i<=n; i++) {
    p = p * x;
}
return (p);
}

```

The recursive function proceeds as follows.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* function prototypes */
5  float power_r (float x, int n);
6
7  /*
8   * main program
9   */
10 int main (void) {
11     float x, y;
12     int n;
13
14     fprintf(stdout, "Base x: ");
15     scanf("%f", &x);
16     fprintf(stdout, "Exponent n (>= 0): ");
17     scanf("%d", &n);
18
19     y = power_r(x, n);
20     fprintf(stdout, "Power (%.2f^%d) = %.3f\n", x, n, y);
21
22     return EXIT_SUCCESS;
23 }
24
25 /*
26  * power computation, recursive function
27 */
28 float power_r (float x, int n) {
29     if (n <= 0) {
30         return 1.0;
31     }
32
33     return x*power_r(x, n-1);
34 }

```

## 5.9 Array Visit

### Specifications

Write a program which:

- ▷ Defines and initializes an array of integer values of size  $N$ .
- ▷ Calls a recursive function which prints-out all elements of the array (from the element with index 0 to the element with index  $N - 1$ ), and it returns the largest and the smallest values within the array.

## Solution

The following code presents two version of the function.

Function `array_r_ver1` adopts the standard array (`[]`) notation to manipulate the array. Constants `INT_MIN` and `INT_MAX` (defined in `limits.h`) simplify the logic required to discover the minimum and maximum values.

Function `array_r_ver2` uses the pointer `*` notation to manipulate the array. In this second case, the pointer `v` is moved along the array, such that at recursion level  $n$  the first element of the received array is actually the one in position  $n$  on the original array.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <limits.h>
4
5 #define N 10
6
7 /* function prototypes */
8 void array_r_ver1 (int [], int *, int *, int);
9 void array_r_ver2 (int *, int *, int *, int);
10
11 /*
12  * main program
13  */
14 int main (void) {
15     int v[N] = {1, 3, 5, 7, 9, 11, 8, 6, 4, 2};
16     int min, max;
17
18     min = INT_MAX;
19     max = INT_MIN;
20     fprintf(stdout, "Array: ");
21     array_r_ver1 (v, &min, &max, 0);
22     fprintf(stdout, "\n");
23     fprintf(stdout, "Min = %d\n", min);
24     fprintf(stdout, "Max = %d\n", max);
25
26     min = INT_MAX;
27     max = INT_MIN;
28     fprintf(stdout, "Array: ");
29     array_r_ver2 (&v[0], &min, &max, 0);
30     fprintf(stdout, "\n");
31     fprintf(stdout, "Min = %d\n", min);
32     fprintf(stdout, "Max = %d\n", max);
33
34     return EXIT_SUCCESS;
35 }
36
37 /*
38  * array visit using the [] notation
39  */
40 void array_r_ver1 (int v[], int *min, int *max, int n) {
41     if (n >= N) {
42         return;
43     }
44
45     fprintf (stdout, "%d ", v[n]);
46
47     if (v[n] < (*min)) {

```

```

48     *min = v[n];
49 }
50 if (v[n] > (*max)) {
51     *max = v[n];
52 }
53 array_r_ver1 (v, min, max, n+1);
54
55 return;
56 }
57
58 /*
59 * array visit usin the pointer * notation
60 */
61 void array_r_ver2 (int *v, int *min, int *max, int n) {
62     if (n >= N) {
63         return;
64     }
65     fprintf (stdout, "%d ", *v);
66
67     if ((*v) < (*min)) {
68         *min = *v;
69     }
70     if ((*v) > (*max)) {
71         *max = *v;
72     }
73     array_r_ver2 (v+1, min, max, n+1);
74
75     return;
76 }
77 }
```

## 5.10 List Visit

### Specifications

Write a program which:

- ▷ Create a list of integer values.
- ▷ Calls two recursive functions. The first one prints-out all elements of the list from head to tail. The second one prints-out all elements of the list from tail to head.

### Solution

The logic is quite close to the one used in Exercise 5.9 to print-out an array. Notice how the two requested function vary very little (just in the instruction order) to perform their job.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <limits.h>
4
5 #define N 10
6
7 typedef struct element_s {
8     int key;
9     struct element_s *next;
10 } element_t;
```

```
11  /* function prototypes */
12  void list_r_ver1 (element_t *);
13  void list_r_ver2 (element_t *);
14
15  /*
16   * main program
17   */
18
19 int main (void) {
20     int i;
21     int v[N] = {1, 3, 5, 7, 9, 11, 8, 6, 4, 2};
22     element_t *p, *head;
23
24     /* list creation (push element into the lsit) */
25     head = NULL;
26     for (i=0; i<N; i++) {
27         p = malloc (1 * sizeof (element_t));
28         if (p == NULL) {
29             fprintf (stderr, "Memory allocation error.\n");
30             exit (EXIT_FAILURE);
31         }
32         p->key = v[i];
33         p->next = head;
34         head = p;
35     }
36
37     fprintf (stdout, "List: ");
38     list_r_ver1 (head);
39     fprintf (stdout, "\n");
40
41     fprintf (stdout, "List: ");
42     list_r_ver2 (head);
43     fprintf (stdout, "\n");
44
45     return EXIT_SUCCESS;
46 }
47
48 /*
49  * list visit from head to tail
50  */
51 void list_r_ver1 (element_t *head) {
52     if (head == NULL) {
53         return;
54     }
55
56     fprintf (stdout, "%d ", head->key);
57     list_r_ver1 (head->next);
58
59     return;
60 }
61
62 /*
63  * list visit from head to tail
64  */
65 void list_r_ver2 (element_t *head) {
66     if (head == NULL) {
67         return;
68     }
69 }
```

```

70     list_r_ver2 (head->next);
71     fprintf (stdout, "%d ", head->key);
72
73     return;
74 }
```

## 5.11 Greatest Common Divisor

### Problem definition

The *greatest common divisor (GCD)* of two integers  $x$  and  $y$  is the greatest integer that divides both  $x$  and  $y$  with no remainder.

The Euclidean algorithm (circa 300 BC) can be thought to work as follow:

1. Given  $x$  and  $y$ , let  $M$  be the greatest value and  $m$  be the smallest one.
2. Let  $r$  be the remainder of the division between  $M$  and  $m$ , i.e.,  $r = M \% m$ .
3. If  $r$  is equal to zero, then  $m$  is the GCD for  $x$  and  $y$ .
4. If  $r$  is not zero, the algorithm restart from step 2 by substituting  $M$  and  $m$  with  $m$  and  $r$ , respectively.

**Example 5.4** Let us suppose to use the Euclidean algorithm to compute  $GCD(20, 12)$ . We obtain the following steps:

$$\begin{aligned}
 GCD(20, 12) &= GCD(12, 20 \% 12) = GCD(12, 8) \\
 &= GCD(8, 12 \% 8) = GCD(8, 4) \\
 &= GCD(4, 8 \% 4) = GCD(8, 0) \\
 &= 4
 \end{aligned}$$

### Example 5.5

$$\begin{aligned}
 GCD(3, 12) &= GCD(12, 3) \\
 &= GCD(3, 12 \% 3) = GCD(3, 0) \\
 &= 3
 \end{aligned}$$

### Example 5.6

$$\begin{aligned}
 GCD(3, 36) &= GCD(36, 3) \\
 &= GCD(3, 36 \% 3) = GCD(3, 0) \\
 &= 3
 \end{aligned}$$

### Specifications

Write a recursive program able to read two positive integers  $x$  and  $y$  and to compute (and print-out) their GCD using the Euclid's algorithm.

### Solution

The following solution includes both the recursive than the iterative computations, i.e., function `gcd_r` and `gcd_i`, respectively.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
```

```
5 int gcd_i (int, int);
6 int gcd_r (int, int);
7
8 /* main program
9 */
10 int main (void) {
11     int x, y, vr, vi;
12
13     fprintf(stdout, "First number: ");
14     scanf("%d", &x);
15     fprintf(stdout, "Second number: ");
16     scanf("%d", &y);
17
18     if (x > y) {
19         vr = gcd_r(x, y);
20         vi = gcd_i(x, y);
21     } else {
22         vr = gcd_r(y, x);
23         vi = gcd_i(y, x);
24     }
25     fprintf(stdout, "GCD(%d, %d): Rec=%d; Iter=%d\n", x, y, vr, vi);
26
27     return EXIT_SUCCESS;
28 }
29
30
31 /*
32 * GCD computation, recursive function
33 */
34 int gcd_r (int M, int m) {
35     if (m == 0) {
36         return M;
37     }
38
39     return gcd_r(m, M%m);
40 }
41 #include <stdio.h>
42
43 /*
44 * GCD computation, iterative function
45 */
46 int gcd_i (int M, int m) {
47     int r;
48
49     while (m != 0) {
50         r = M % m;
51         M = m;
52         m = r;
53     }
54
55     return M;
56 }
```

## 5.12 Map regions

## Specification

A file stores a matrix whose size is unknown but limited to 20 rows and 20 columns. The matrix contains only two characters, points '.' and stars '\*', and it represents a map. The map includes several areas (or "regions") divided fences. Each fence is represented by '\*' characters. Points '.' indicate the area background. Fences are "closed" by considering only vertical and horizontal moves on the map.

Write a program able to count the number of “close” areas (regions) within the map, and to mark each area with a different integer value. The so marked map has to be stored in an output file.

Input and output file names have to be receive as parameters.

**Example 5.7** Let the following picture on the left-hand side be the content of the input file:

A decorative border consisting of a grid of stars. The grid has 10 columns and 10 rows. The stars are arranged in a repeating pattern where each row and column contains 10 stars. The stars are white with black outlines, set against a light gray background.

\*\*\*\*\*  
\*111\*222222222\*  
\*111\*222222222\*  
\*111\*222222222\*\*  
\*111\*222\*\*\*\*3\*  
\*1111\*2\*333333\*  
\*11111\*3333333\*  
\*11111\*3333333\*  
\*11111\*\*\*\*333\*  
\*1111\*44444\*33\*  
\*\*\*\*\*44444444\*3\*  
\*44444444444444\*\*  
\*44444444444444\*  
\*\*\*\*\*

The number of region is 4, and the picture on the right-hand side is a possible content of the output file.

### Solution

**Solution**  
Notice that the output file has the described appearance only when the number of regions is smaller than 10, and the numbers used to count them up can be represented on single digits. The following program does not take any action to improve this aspect in the general case.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define DIM 20
5
6 /* function prototypes */
7 void map_read (char *, int [DIM][DIM], int *, int *);
8 void expand_r (int [DIM][DIM], int, int, int);
9 void map_write (char *, int [DIM][DIM], int, int, int);
10
11 /*
12 * main program
13 */
14 int main (int argc, char *argv[]) {
15     int map[DIM][DIM];
```

```

16   int i, j, nr, nc, n=0;
17   map_read(argv[1], map, &nr, &nc);
18   for (i=0; i<nr; i++) {
19     for (j=0; j<nc; j++) {
20       if (map[i][j] == (-1)) {
21         expand_r(map, i, j, ++n);
22       }
23     }
24   }
25   map_write(argv[2], map, nr, nc, n);
26
27   return EXIT_SUCCESS;
28 }
29
30
31 /* 
32  * load the map from file
33 */
34 void map_read (char *name, int map[DIM] [DIM], int *nr, int *nc) {
35   FILE *fp;
36   char line[DIM+2];
37   int i, j;
38
39   fp = fopen(name, "r");
40
41   i = 0;
42   while (fgets(line, DIM+2, fp) != NULL) {
43     j = 0;
44     while (j<DIM && (line[j]=='.' || line[j]=='*')) {
45       map[i][j] = (line[j]=='*' ? 0 : -1);
46       j++;
47     }
48     i++;
49   }
50
51   fclose(fp);
52
53   *nr = i;
54   *nc = j;
55   return;
56 }
57
58 /*
59  * expand a region numbering it, recursive function
60 */
61 void expand_r (int map[DIM] [DIM], int x, int y, int id) {
62   int i, xx, yy;
63   const int xoff[] = {1, 0, -1, 0};
64   const int yoff[] = {0, 1, 0, -1};
65
66   if (map[x][y] != (-1)) {
67     return;
68   }
69
70   /* mark the position */
71   map[x][y] = id;
72
73   /* look for adjacences */
74   for (i=0; i<4; i++) {

```

```

75     xx = x + xoff[i];
76     yy = y + yoff[i];
77     expand_r(map, xx, yy, id);
78 }
79 }
80
81 /*
82 * write map on output file
83 */
84 void map_write (char *name, int map[DIM][DIM], int nr, int nc, int n) {
85     FILE *fp;
86     int i, j;
87
88     fp = fopen(name, "w");
89
90     for (i=0; i<nr; i++) {
91         for (j=0; j<nc; j++) {
92             if (map[i][j]==0) {
93                 fprintf (fp, "*");
94             } else {
95                 fprintf (fp, "%d", map[i][j]);
96             }
97         }
98         fprintf (fp, "\n");
99     }
100    fprintf (stdout, "Regions number = %d\n", n);
101
102    fclose (fp);
103
104    return;
105 }
```

## 5.13 Prefix (Polish) Notation

### Problem definition

The Prefix or Polish notation (also known as normal Polish notation) is a form of notation for arithmetic, in which operators are placed to the left of their operands. For instance, the expression that would be written in conventional infix notation as

$$(5 + 6) \cdot 7$$

can be written in prefix as

$$\cdot (+ 5 6) 7$$

Since the simple arithmetic operators are all binary (at least, in arithmetic contexts), any prefix representation thereof is unambiguous, and bracketing the prefix expression is unnecessary. As such, the previous expression can be further simplified to

$$\cdot + 5 6 7$$

### Specifications

Write a program able to parse a prefix expression passed as a single string to the program, and to print-out the result on standard output.

There may be 4 types of operators: '+' for sums, '-' for subtractions, '\*' for multiplications, and '/' for divisions.

**Example 5.8** Let us suppose the program is run with the following parameter: "+ 3.5 5.2". It has to print the value 8.7, i.e.,  $3.5 + 5.2$ .

**Example 5.9** If the program receives the string "\* + 1.4 2.1 / - 4.4 0.4 2.0" it has to print-out 7.0, i.e.,  $(1.4 + 2.1) \cdot ((4.4 - 0.4)/2.0)$ .

### Solution

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <ctype.h>
5
6 /* function prototypes */
7 float eval_r (char *expr, int *pos_ptr);
8
9 /*
10  * main program
11 */
12 int main (int argc, char *argv[]) {
13     float result;
14     int pos=0;
15
16     if (argc < 2) {
17         fprintf(stderr, "Error: missing parameter.\n");
18         fprintf(stderr, "Run as: %s \"<prefix_expression>\"\n", argv[0]);
19         return 1;
20     }
21
22     result = eval_r(argv[1], &pos);
23     fprintf(stdout, "Result = %.2f\n", result);
24     return EXIT_SUCCESS;
25 }
26
27 /*
28  * evaluate a prefix expression, recursive function
29 */
30 float eval_r (char *expr, int *pos_ptr) {
31     float left, right, result;
32     char operator;
33     int k = *pos_ptr;
34
35     /* skip spaces */
36     while (isspace(expr[k])) {
37         k++;
38     }
39
40     /* manage sub-expressions and compute the result */
41     if (expr[k]=='+') || expr[k]=='*' || expr[k]=='-' || expr[k]=='/') {
42         operator = expr[k++];
43         left = eval_r(expr, &k);
44         right = eval_r(expr, &k);
45         switch (operator) {
46             case '+': result = left+right; break;
47             case '*': result = left*right; break;

```

```
48     case '-' : result = left-right; break;
49     case '/' : result = left/right; break;
50 }
51 } else {
52     /* terminal case: just a real value */
53     sscanf(&expr[k], "%f", &result);
54     while (isdigit(expr[k]) || expr[k]=='.') {
55         k++;
56     }
57 }
58
59 *pos_ptr = k;
60
61 return result;
62 }
```

# Chapter 6

## Classic recursive problems

In this chapter we analyze some classic recursive problems, going from binary search, to recursive sorting algorithms (such as merge-sort and quick-sort), and to other typical recursive problems.

### 6.1 Binary Search

#### Problem definition

In computer science, binary search, also known as half-interval search or logarithmic search, is a search algorithm that finds the position of a target value within a sorted array.

Binary search begins by comparing the middle element of the array with the target value. If the target value matches the middle element, its position in the array is returned. If the target value is less than or greater than the middle element, the search continues in the lower or upper half of the array, respectively, eliminating the other half from further consideration.

If  $n$  is the number of elements in the array, binary search runs in at worst logarithmic time, making  $O(\log n)$  comparisons. Notice that a linear search requires a linear time  $O(n)$ . Binary search takes  $O(n)$  space, meaning that the space taken by the algorithm is proportional to the number of elements in the array.

Although specialized data structures designed for fast searching (such as hash tables) can be searched more efficiently, binary search applies to a wider range of search problems.

The core idea of the algorithm is simple, but implementing binary search correctly requires attention to several issues such as its exit conditions and midpoint calculation.

#### Specifications

Write a recursive program able to:

- ▷ Read an array  $v$  of  $N$  integer values from standard input ( $N$  being a pre-defined constant).

- ▷ Read an integer value  $d$ .
- ▷ Search  $d$  in  $v$ , printing-out its index position within  $v$  if it belongs to the array.

Assume the array  $v$  is ordered in ascending numeric order.

## Solution

The following solution reports both the recursive than the iterative implementations for the sake of a direct comparison between the two approaches.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N 10
5
6 /* function prototypes */
7 int search_r (int [], int, int, int);
8 int search_i (int [], int, int, int);
9
10 /*
11  * main program
12  */
13 int main (void) {
14     int i, v[N], d, posr, posi;
15
16     /* input data */
17     fprintf(stdout, "Input sorted array (dim=%d):\n", N);
18     for (i=0; i<N; i++) {
19         fprintf(stdout, "v[%d] = ", i);
20         scanf("%d", &v[i]);
21     }
22     fprintf(stdout, "Number to search = ");
23     scanf("%d", &d);
24
25     /* search and output result */
26     posr = search_r(v, 0, N-1, d);
27     posi = search_i(v, 0, N-1, d);
28     if (posr>=0 && posi>=0) {
29         fprintf(stdout, "Number %d found in position: rec=%d ite=%d.\n",
30                 d, posr, posi);
31     } else {
32         fprintf(stdout, "Number %d NOT found.\n", d);
33     }
34
35     return EXIT_SUCCESS;
36 }
37
38 /*
39  * binary search, recursive function
40  */
41 int search_r (int v[], int left, int right, int d) {
42     int c;
43
44     if (left > right) {
45         return -1;
46     }
47
48     c = (left+right)/2;
49     if (d < v[c]) {

```

```

50     return search_r(v, left, c-1, d);
51 }
52 if (d > v[c]) {
53     return search_r(v, c+1, right, d);
54 }
55 /* value found */
56 return c;
57 }
58 }

59 /*
60 * binary search, iterative function
61 */
62 int search_i (int v[], int left, int right, int d) {
63     int i;
64
65     while (left<=right) {
66         i = (left+right)/2;
67         if (d == v[i]) {
68             return i;
69         }
70         if (d < v[i]) {
71             right = i - 1;
72         } else {
73             left = i + 1;
74         }
75     }
76 }
77
78 return -1;
79 }
```

## 6.2 Merge sort

### Problem definition

Merge sort (also commonly spelled merge-sort) is a divide and conquer sorting algorithm. It was presented for the first time by John von Neumann in 1945. It is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output.

Conceptually, a merge sort works as follows:

- ▷ Divide the unsorted list into  $n$  sub-lists, each one containing 1 element. This because a list of 1 element is considered sorted and the division phase may end.
- ▷ Repeatedly merge sub-lists to produce new sorted sub-lists until there is only one sub-list remaining. This will be the sorted list.

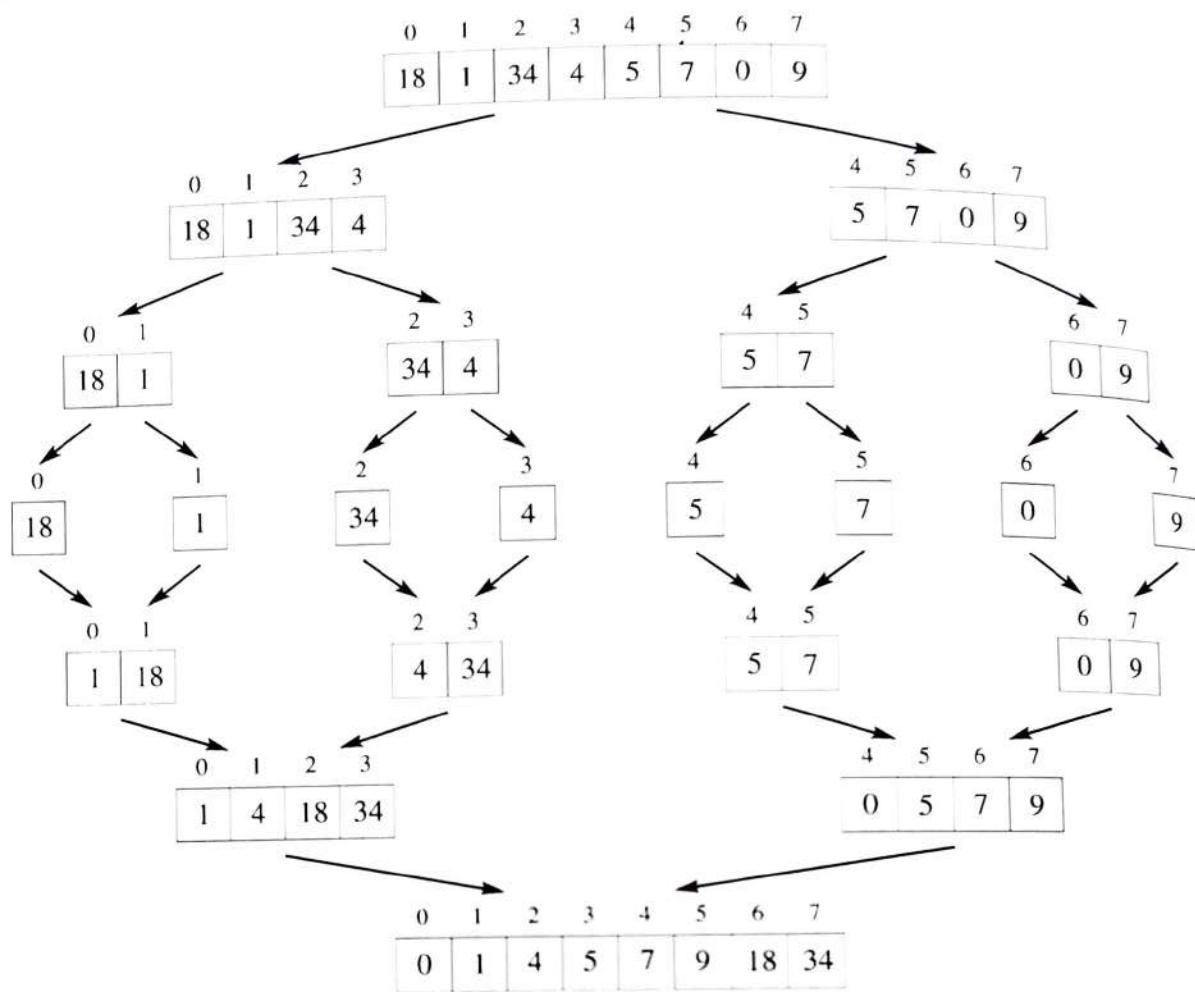
In sorting  $n$  objects, merge sort has an average and worst-case performance of  $O(n \cdot \log n)$ . If the running time of merge sort for a list of length  $n$  is  $T(n)$ , then the recurrence

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$$

follows from the definition of the algorithm, which applies the algorithm to two lists of half the size of the original list, and add the  $n$  steps taken to merge the resulting two

lists.

Figure 6.1 shows how the algorithm works on an array of 8 integers.



**Figure 6.1** Merge sort in action on an array of 8 integer elements. The target is to obtain a numeric ascending order.

## Specifications

Write a program able to read an array of integer values, and to order it in ascending order, using the merge sort algorithm.

## Solution

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
5 int *array_read(int *dim);
6 void merge_sort_r(int *v, int p, int r, int *aux);
7 void merge(int *v, int p, int q, int r, int *aux);
8 void array_write(int v[], int dim);
9
10 /*
11  * main program

```

```
12  /*
13   int main (void) {
14     int n, *v, *aux;
15
16     v = array_read(&n);
17     aux = (int *)malloc(n*sizeof(int));
18     if (aux==NULL) {
19       fprintf (stderr, "Memory allocation error.\n");
20       exit(EXIT_FAILURE);
21     }
22     merge_sort_r(v, 0, n-1, aux);
23     array_write(v, n);
24     free(v);
25     free(aux);
26     return EXIT_SUCCESS;
27   }
28
29  /*
30   * load the initial array
31  */
32 int *array_read (int *dim) {
33   int i, *v;
34
35   fprintf(stdout, "Size: ");
36   scanf("%d", dim);
37   v = (int *)malloc((*dim)*sizeof(int));
38   if (v==NULL) {
39     fprintf (stderr, "Memory allocation error.\n");
40     exit(EXIT_FAILURE);
41   }
42
43   fprintf(stdout, "Initial array:\n");
44   for (i=0; i<*dim; i++) {
45     fprintf(stdout, "v[%d] = ", i);
46     scanf("%d", &v[i]);
47   }
48   return v;
49 }
50
51  /*
52   * sort an array through merge sort, recursive function
53  */
54 void merge_sort_r (int *v, int p, int r, int *aux) {
55   int q;
56
57   if (p < r) {
58     q = (p+r)/2;
59     merge_sort_r(v, p, q, aux);
60     merge_sort_r(v, q+1, r, aux);
61     merge(v, p, q, r, aux);
62   }
63
64   return;
65 }
66
67  /*
68   * merge two sorted sub-arrays
69  */
70 void merge (int *v, int p, int q, int r, int *aux) {
```

```

71     int i, j, k;
72     for (i=p, j=q+1, k=p; i<=q && j<=r; ) {
73         if (v[i] < v[j]) {
74             aux[k++] = v[i++];
75         } else {
76             aux[k++] = v[j++];
77         }
78     }
79 }
80
81     while (i <= q) {
82         aux[k++] = v[i++];
83     }
84     while (j <= r) {
85         aux[k++] = v[j++];
86     }
87
88     for (k=p; k<=r; k++) {
89         v[k] = aux[k];
90     }
91 }
92
93 /*
94  * print the output array
95 */
96 void array_write (int v[], int dim) {
97     int i;
98
99     fprintf(stdout, "Sorted array:\n");
100    for (i=0; i<dim; i++) {
101        fprintf(stdout, "v[%d] = %d\n", i, v[i]);
102    }
103    fprintf(stdout, "\n");
104 }
```

## 6.3 Quick sort

### Problem definition

Quick sort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. Developed by Tony Hoare in 1959, with his work published in 1961, it is still a commonly used algorithm for sorting. When it is implemented in an efficient way, it can be about two or three times faster than its main competitors, merge sort and heap sort.

Quick sort is a comparison sort, meaning that it can sort items of any type for which a “less-than” relation (formally, a total order) is defined. In efficient implementations it is not a stable sort, meaning that the relative order of equal sort items is not preserved. Quick sort can operate in-place on an array, requiring small additional amounts of memory to perform the sorting.

Mathematical analysis of quick sort shows that, on average, the algorithm takes  $O(n \cdot \log n)$  comparisons to sort  $n$  items. In the worst case, it makes  $O(n^2)$  comparisons, though this behavior is rare.

Quick sort is a divide and conquer algorithm. It first divides a large array into two smaller sub-arrays: The low elements and the high elements. Then, it recursively sorts those sub-arrays. The steps are:

- ▷ Selection: Pick an element from the array. This element is usually called *pivot*.
- ▷ Partition: Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it. Notice that values equal to the pivot can go either way. After this partitioning phase, the pivot is in its final position.
- ▷ Recur: Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base case of the recursion is reached when arrays have size zero or one, which means that they do not need to be sorted.

The pivot selection and partitioning steps can be done in several different ways. The choice of specific implementation schemes greatly affects the algorithm's performance.

### Lomuto partition scheme

This partition scheme is attributed to Nico Lomuto and popularized by several publications (see for example [citeCormenEnglish](#)). This scheme chooses a pivot that is typically the last element in the array. The algorithm maintains the index to put the pivot in variable  $i$ , and each time it finds an element less than or equal to pivot, this index is incremented and that element would be placed before the pivot. As this scheme is more compact and easy to understand than other approaches, it is frequently used in introductory material. Anyhow, it is less efficient than Hoare's original scheme. This scheme degrades to  $O(n^2)$  when the array is already sorted as well as when the array has all equal elements. There have been various variants proposed to boost performance including various ways to select pivot, deal with equal elements, use other sorting algorithms such as insertion sort for small arrays and so on.

### Hoare partition scheme

The original partition scheme, described by Hoare, uses two indices that start at the ends of the array being partitioned. They move toward each other, until they detect an inversion: A pair of elements, one greater than or equal to the pivot, one lesser or equal, that are in the wrong order relative to each other. The inverted elements are then swapped. When the indices meet, the algorithm stops and returns the final index. There are many variants of this algorithm. The first version selects the rightmost ( $v[r]$ ) element as pivot. The second version selects the leftmost ( $v[l]$ ) element as pivot. Small differences are implied in the remaining code section.

Hoare's scheme is more efficient than Lomuto's partition scheme because it does three times fewer swaps on average, and it creates efficient partitions even when all values are equal. Like Lomuto's partition scheme, Hoare partitioning also causes quicksort to degrade to  $O(n^2)$  when the input array is already sorted. It also does not produce a stable sort.

### Solution

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /*

```

```
5  * Select (1) One (and only one) Partition Scheme:  
6  * HOARE Version 1 with pivot = rightmost element  
7  * HOARE Version 1 with pivot = leftmost element  
8  * LOMUTO  
9  */  
10 #define HOARE_V1 1  
11 #define HOARE_V2 0  
12 #define LOMUTO 0  
13 /* function prototypes */  
14 int *array_read (int *);  
15 void quick_sort_r (int *, int, int);  
16 int partition (int *, int, int);  
17 void swap (int *, int, int);  
18 void array_write(int *, int, int);  
19  
20 /*  
21  * main program  
22  */  
23 int main (void) {  
24     int n, *v;  
25  
26     v = array_read(&n);  
27     quick_sort_r(v, 0, n-1);  
28     array_write(v, 0, n-1);  
29     free(v);  
30  
31     return EXIT_SUCCESS;  
32 }  
33  
34 /*  
35  * load the initial array  
36  */  
37 int *array_read (int *dim) {  
38     int i, *v;  
39  
40     fprintf(stdout, "Size: ");  
41     scanf("%d", dim);  
42     v = (int *)malloc((*dim)*sizeof(int));  
43     if (v==NULL) {  
44         fprintf (stderr, "Memory allocation error.\n");  
45         exit(EXIT_FAILURE);  
46     }  
47  
48     fprintf(stdout, "Initial array:\n");  
49     for (i=0; i<*dim; i++) {  
50         fprintf(stdout, "v[%d] = ", i);  
51         scanf("%d", &v[i]);  
52     }  
53     return v;  
54 }  
55  
56 /*  
57  * sort an array through quick sort, recursive function  
58  */  
59 void quick_sort_r(int *v, int l, int r) {  
60     int q;  
61  
62     if (l < r) {
```



```

123     int i, j, pivot;
124
125     i = l;
126     j = r + 1;
127     pivot = v[l];
128     while (i < j) {
129         while (i < r && v[++i] <= pivot);
130         while (v[--j] > pivot);
131
132         if (i < j) {
133             swap (v, i, j);
134         }
135     }
136
137     swap (v, l, j);
138
139     return j;
140 }
141 #endif
142
143 /*
144 * swap
145 */
146 void swap (int *v, int i, int j) {
147     int tmp;
148
149     tmp = v[i];
150     v[i] = v[j];
151     v[j] = tmp;
152
153     return;
154 }
155
156 /*
157 * print the output array
158 */
159 void array_write (int v[], int l, int r) {
160     int i;
161
162     fprintf(stdout, "Sorted array:\n");
163     for (i=l; i<=r; i++) {
164         fprintf(stdout, "v[%d] = %d\n", i, v[i]);
165     }
166     fprintf(stdout, "\n");
167 }
```

## 6.4 Combinatorics

This section includes functions to apply to the multiplication methods to sets, to generate all arrangements, permutations, combinations and power-sets.

### 6.4.1 The multiplication method

Given  $n$  sets  $S_i$  (with  $i \in [0, n]$ ) each of cardinality  $|S_i|$ , the number of ordered t-uples  $(s_0, s_1, \dots, s_{n-1})$  with  $s_0 \in S_0, s_1 \in S_1, \dots, s_{n-1} \in S_{n-1}$ , is:

$$\#tuple = \prod_{i=0}^{n-1} |S_i|$$

Alternatively if an object  $x_0$  can be selected in  $p_0$  ways from a group, an object  $x_1$  can be selected in  $p_1$  ways, an object  $x_{n-1}$  can be selected in  $p_n - 1$  ways, the choice of a t-tuple of objects  $(x_0, x_1, \dots, x_{n-1})$  can be done in  $p_0 \cdot p_1 \cdot \dots \cdot p_{n-1}$  ways.

**Example 6.1** In a game it is necessary to select 3 symbols, each one from a set of symbols. The sets are:  $S_0 = (A, B)$ ,  $S_1 = (0, 1)$ ,  $S_2 = (X, Y, Z)$ . If we select one symbol in  $S_0$ , one in  $S_1$  and one in  $S_2$  we can generate a number of sets equal to:

$$\#tuple = \prod_{i=0}^{n-1} |S_i| = 2 \cdot 2 \cdot 3 = 12$$

Those are:

$$\begin{aligned} & \{A, 0, X\}, \{A, 0, Y\}, \{A, 0, Z\}, \{A, 1, Z\}, \{A, 1, Y\}, \{A, 1, Z\}, \\ & \{B, 0, X\}, \{B, 0, Y\}, \{B, 0, Z\}, \{B, 1, Z\}, \{B, 1, Y\}, \{B, 1, Z\}. \end{aligned}$$

## Implementation

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N 10
5
6 typedef struct value_s {
7     int num;
8     char element[N];
9 } value_t;
10
11 /* function prototypes */
12 int multiplication_principle(value_t *, char *, int, int, int);
13
14 /*
15  * main program
16 */
17 int main(int argc, char *argv[]) {
18     value_t value[3];
19     char solution[3];
20     int n = 3, pos = 0, count = 0;
21     int total;
22
23     value[0].num = 2;
24     value[0].element[0] = 'A';
25     value[0].element[1] = 'B';
26     value[1].num = 2;
27     value[1].element[0] = '0';
28     value[1].element[1] = '1';
29     value[2].num = 3;
30     value[2].element[0] = 'X';
31     value[2].element[1] = 'Y';
32     value[2].element[2] = 'Z';
33
34
35     fprintf(stdout, "Multiplication Principle:\n");
36     total = multiplication_principle(&value, &solution, n, pos, count);
37     fprintf(stdout, "\nTotal Number: %d\n", total);
38
39 }
40 return EXIT_SUCCESS;
41

```

```

42  /*
43   * Multiplication Principle
44   */
45  int multiplication_principle (
46    value_t *value, char *solution, int n, int pos, int count
47  )
48  {
49    int i;
50
51    if (pos >= n) {
52      fprintf(stdout, "{ ");
53      for (i=0; i<n; i++) {
54        fprintf(stdout, "%c ", solution[i]);
55      }
56      fprintf(stdout, "}");
57      return count+1;
58    }
59
60    for (i=0; i<value[pos].num; i++) {
61      solution[pos] = value[pos].element[i];
62      count = multiplication_principle (value, solution, n, pos+1, count);
63    }
64  return count;
65 }
```

## 6.4.2 Simple arrangements

A simple arrangement  $D_{n,k}$  of  $n$  distinct objects of class  $k$  ( $k$  by  $k$ ) is an ordered subset composed by  $k$  out of  $n$  objects, with  $0 \leq k \leq n$ .

There are

$$D_{n,k} = \frac{n!}{(n-k)!} = n \cdot (n-1) \cdot \dots \cdot (n-k+1)$$

simple arrangements of  $n$  objects  $k$  by  $k$ .

**Example 6.2** With  $n = 4$ ,  $k = 2$ , and the set  $\{4, 9, 1, 0\}$ , there are  $(4 \cdot 3) = 12$  simple arrangements:

$\{4, 9\}, \{4, 1\}, \{4, 0\}, \{9, 4\}, \{9, 1\}, \{9, 0\}, \{1, 4\}, \{1, 9\}, \{1, 0\}, \{0, 4\}, \{0, 9\}, \{0, 1\}$ .

**Example 6.3** With  $n = 3$ ,  $k = 2$ , and the set  $\{A, B, C\}$ , there are  $(3 \cdot 2) = 6$  simple arrangements:

$\{A, B\}, \{A, C\}, \{B, A\}, \{B, C\}, \{C, A\}, \{C, B\}$ .

## Implementation

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
5 int arrangement_simple(int *, int *, int *, int, int, int, int);
6
7 /*
8  * main program
9  */
10 int main(int argc, char *argv[]) {
11  const int n = 3;
```

```

12 const int k = 2;
13 int value[] = {1, 2, 3};
14 int mark[] = {0, 0, 0};
15 int solution[] = {0, 0};
16 int pos = 0;
17 int count = 0;
18 int total;
19
20 fprintf(stdout, "Simple Arrangements:\n");
21 total = arrangement_simple (value, solution, mark, n, pos, k, count);
22 fprintf(stdout, "\nTotal Number: %d\n", total);
23
24 return EXIT_SUCCESS;
25 }
26
27 /*
28 * Simple Arrangements
29 */
30 int arrangement_simple(
31     int *value, int *solution, int *mark, int n, int pos, int k, int count
32 )
33 {
34     int i;
35
36     if (pos >= k) {
37         fprintf(stdout, "{ ");
38         for (i=0; i<k; i++) {
39             fprintf(stdout, "%d ", solution[i]);
40         }
41         fprintf(stdout, "}");
42         return count+1;
43     }
44
45     for (i=0; i<n; i++) {
46         if (mark[i] == 0) {
47             mark[i] = 1;
48             solution[pos] = value[i];
49             count = arrangement_simple(value, solution, mark, n, pos+1, k, count);
50             mark[i] = 0;
51         }
52     }
53     return count;
54 }
```

### 6.4.3 Arrangements with repetitions

An arrangement with repetitions  $D_{n,k}$  of  $n$  distinct objects of class  $k$  ( $k$  by  $k$ ) is an ordered subset composed of  $k$  out of  $n$  objects, with  $k \geq 0$  each of whom may be taken up to  $k$  times.

There are:

$$D_{n,k} = n^k$$

arrangements with repetitions of  $n$  objects taken  $k$  by  $k$ .

**Example 6.4** With  $n = 2$ ,  $k = 3$ , and the set  $\{0, 1\}$ , there are  $(2 \cdot 2 \cdot 2) = 8$  arrangements with repetition:

$$\{0, 0, 0\}, \{0, 0, 1\}, \{0, 1, 0\}, \{0, 1, 1\}, \{1, 0, 0\}, \{1, 0, 1\}, \{1, 1, 0\}, \{1, 1, 1\}.$$

**Example 6.5** With  $n = 3$ ,  $k = 2$ , and the set  $\{A, B, C\}$ , there are  $(3 \cdot 3) = 9$  arrangements with repetition:

$\{A, A\}, \{A, B\}, \{A, C\}, \{B, A\}, \{B, B\}, \{B, C\}, \{C, A\}, \{C, B\}, \{C, C\}$ .

## Implementation

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
5 int arrangement_repetition(int *, int *, int, int, int);
6
7 /*
8 * main program
9 */
10 int main(int argc, char *argv[]) {
11     const int n = 3;
12     const int k = 2;
13     int value[] = {1, 2, 3};
14     int solution[] = {0, 0};
15     int pos = 0;
16     int count = 0;
17     int total;
18
19     fprintf(stdout, "Arrangements with repetitiona:\n");
20     total = arrangement_repetition (value, solution, n, pos, k, count);
21     fprintf(stdout, "\nTotal Number: %d\n", total);
22
23     return EXIT_SUCCESS;
24 }
25
26 /*
27 * Arrangements with Repetition
28 */
29 int arrangement_repetition(
30     int *value, int *solution, int n, int pos, int k, int count
31 )
32 {
33     int i;
34
35     if (pos >= k) {
36         fprintf(stdout, "{ ");
37         for (i=0; i<k; i++) {
38             fprintf(stdout, "%d ", solution[i]);
39         }
40         fprintf(stdout, "}");
41         return count+1;
42     }
43
44     for (i=0; i<n; i++) {
45         solution[pos] = value[i];
46         count = arrangement_repetition(value, solution, n, pos+1, k, count);
47     }
48     return count;
49 }
```

#### 6.4.4 Simple permutation

A simple arrangement  $D_{n,n}$  of  $n$  distinct objects of class  $n$  ( $n$  by  $n$ ) is a simple permutation  $P_n$ . It is an ordered subset made of  $n$  objects.

There are

$$P_n = D_{n,n} = n!$$

simple permutations of  $n$  objects.

**Example 6.6** With  $n = 4$ , and the set  $\{2, 4, 6, 8\}$ , there are  $4! = 24$  simple permutations:

$\{2, 4, 6, 8\}, \{2, 4, 8, 6\}, \{2, 6, 4, 8\}, \{2, 6, 8, 4\}, \{2, 8, 4, 6\}, \{2, 8, 6, 4\},$   
 $\{4, 2, 6, 8\}, \{4, 2, 8, 6\}, \{4, 6, 2, 8\}, \{4, 6, 8, 2\}, \{4, 8, 2, 6\}, \{4, 8, 6, 2\},$   
 $\{6, 2, 4, 8\}, \{6, 2, 8, 4\}, \{6, 4, 2, 8\}, \{6, 4, 8, 2\}, \{6, 8, 2, 4\}, \{6, 8, 4, 2\},$   
 $\{8, 2, 4, 6\}, \{8, 2, 6, 4\}, \{8, 4, 2, 6\}, \{8, 4, 6, 2\}, \{8, 6, 2, 4\}, \{8, 6, 4, 2\}.$

**Example 6.7** With  $n = 3$ ,  $k = 2$ , and the set  $\{A, B, C\}$ , there are  $3! = 6$  simple permutations:

$\{A, B, C\}, \{A, C, B\}, \{B, A, C\}, \{B, C, A\}, \{C, A, B\}, \{C, B, A\}.$

#### Implementation

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
5 int permutation_simple(int *, int *, int *, int, int, int);
6
7 /*
8 * main program
9 */
10 int main(int argc, char *argv[]) {
11     const int n = 3;
12     int value[] = {1, 2, 3};
13     int solution[] = {0, 0, 0};
14     int mark[] = {0, 0, 0};
15     int pos = 0;
16     int count = 0;
17     int total;
18
19     fprintf(stdout, "Simple Permutations:\n");
20     total = permutation_simple (value, solution, mark, n, pos, count);
21     fprintf(stdout, "\nTotal number: %d\n", total);
22
23     return EXIT_SUCCESS;
24 }
25
26 /*
27 * Simple Permutations
28 */
29 int permutation_simple(
30     int *value, int *solution, int *mark, int n, int pos, int count
31 )
32 {
33     int i;
34
35     if (pos >= n) {

```

```

36     fprintf(stdout, "{ ");
37     for (i=0; i<n; i++) {
38         fprintf(stdout, "%d ", solution[i]);
39     }
40     fprintf(stdout, "}");
41     return count+1;
42 }
43
44 for (i=0; i<n; i++) {
45     if (mark[i] == 0) {
46         mark[i] = 1;
47         solution[pos] = value[i];
48         count = permutation_simple(value, solution, mark, n, pos+1, count);
49         mark[i] = 0;
50     }
51 }
52 return count;
53 }
```

### 6.4.5 Permutations with repetitions

Given a multi-set of  $n$  objects among which  $\alpha$  are identical,  $\beta$  are identical, etc., the number of distinct permutations with repeated objects is:

$$P_n^{\alpha, \beta, \dots} = \frac{n!}{(\alpha! \cdot \beta! \cdot \dots)}$$

**Example 6.8** With  $n = 3$ , the set  $\{1, 2\}$ , and two repetitions of the first value, the number of permutations with repetitions is  $\left(\frac{3!}{2!}\right) = 3$  and they are:

$$\{1, 1, 2\}, \{1, 2, 1\}, \{2, 1, 1\}.$$

**Example 6.9** With  $n = 4$ , 3 distinct values  $\{A, B, C\}$  with the first once repeated twice, there are  $\left(\frac{4!}{2!}\right) = 12$  permutation with repetitions:

$$\begin{aligned} &\{A, A, B, C\}, \{A, A, C, B\}, \{A, B, A, C\}, \{A, C, A, B\}, \{A, B, C, A\}, \{A, C, B, A\}, \\ &\{B, A, A, C\}, \{C, A, A, B\}, \{B, A, C, A\}, \{C, A, B, A\}, \{B, C, A, A\}, \{C, B, A, A\}. \end{aligned}$$

## Implementation

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
5 int permutation_repetition(int *, int *, int *, int, int, int, int);
6
7 /*
8 * main program
9 */
10 int main(int argc, char *argv[]) {
11     const int n = 4;
12     const int n_dist = 3;
13     int value[] = {1, 2, 3};
14     int mark[] = {2, 1, 1};
15     int solution[] = {0, 0, 0, 0};
16     int pos = 0;
17     int count = 0;
```

```

18 int total;
19 fprintf(stdout, "Permutations with repetitions:\n");
20 total = permutation_repetition (value, solution, mark, n, pos, count, n_dist);
21 fprintf(stdout, "\nTotal Number: %d\n", total);
22
23 return EXIT_SUCCESS;
24 }
25
26 /* Permutations with Repetition
27 */
28 int permutation_repetition(
29     int *value, int *solution, int *mark, int n, int pos, int count, int dist
30 )
31 {
32     int i;
33
34     if (pos >= n) {
35         fprintf(stdout, "{ ");
36         for (i=0; i<n; i++) {
37             fprintf(stdout, "%d ", solution[i]);
38         }
39         fprintf(stdout, "}");
40         return count+1;
41     }
42
43     for (i=0; i<dist; i++) {
44         if (mark[i] > 0) {
45             mark[i]--;
46             solution[pos] = value[i];
47             count = permutation_repetition(value, solution, mark, n, pos+1, count, dist);
48             mark[i]++;
49         }
50     }
51     return count;
52 }
53
54 }
```

#### 6.4.6 Simple combinations

A simple combination  $C_{n,k}$  of  $n$  distinct objects of class  $k$  ( $k$  by  $k$ ) is a non ordered subset composed by  $k$  of  $n$  objects with  $0 \leq k \leq n$ .

The number of combinations of  $n$  elements  $k$  by  $k$  equals the number of arrangements of  $n$  elements  $k$  by  $k$  divided by the number of permutations of  $k$  elements.

In other words, there are:

$$C_{n,k} = \binom{n}{k} = \frac{D_{n,k}}{P_k} = \frac{n!}{k! \cdot (n-k)!}$$

simple combination of  $n$  objects  $k$  by  $k$  ( $n$  choose  $k$ ).

**Example 6.10** With  $n = 4$ ,  $k = 2$ , and the set  $\{1, 2, 3, 4\}$ , there are  $(\frac{4!}{2! \cdot 2!}) = 6$  simple combinations:

$$\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}.$$

**Example 6.11** With  $n = 3$ ,  $k = 2$ , and the set  $\{A, B, C\}$ , there are  $(\frac{3!}{2! \cdot 1!}) = 3$  simple

combinations:

$$\{A, B\}, \{A, C\}, \{B, C\}.$$

## Implementation

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
5 int combination_simple(int *, int *, int, int, int, int);
6
7 /*
8 * main program
9 */
10 int main(int argc, char *argv[]) {
11     const int n = 3;
12     const int k = 2;
13     int value[] = {1, 2, 3};
14     int solution[] = {0, 0};
15     int pos = 0;
16     int start = 0;
17     int count = 0;
18     int total;
19
20     fprintf(stdout, "Simple Combinations:\n");
21     total = combination_simple (value, solution, n, k, start, pos, count);
22     fprintf(stdout, "\nTotal Number: %d\n", total);
23
24     return EXIT_SUCCESS;
25 }
26
27 /*
28 * Simple Combinations
29 */
30 int combination_simple(
31     int *value, int *solution, int n, int m, int start, int pos, int count
32 )
33 {
34     int i;
35
36     if (pos >= m) {
37         fprintf(stdout, "{ ");
38         for (i=0; i<m; i++) {
39             fprintf(stdout, "%d ", solution[i]);
40         }
41         fprintf(stdout, "}");
42         return count+1;
43     }
44
45     for (i=start; i<n; i++) {
46         solution[pos] = value[i];
47         count = combination_simple (value, solution, n, m, i+1, pos+1, count);
48     }
49     return count;
50 }
```

### 6.4.7 Combinations with repetitions

A combination with repetitions  $C_{n,k}$  of  $n$  distinct objects of class  $k$  ( $k$  by  $k$ ) is a non ordered subset made of  $k$  of the  $n$  objects with  $k \geq 0$ . Each of them may be taken at

most  $k$  times.  
There are:

$$C'_{n,k} = \frac{(n+k-1)!}{k! \cdot (n-1)!}$$

combinations with repetitions of  $n$  objects  $k$  by  $k$ .

**Example 6.12** With  $n = 6$ ,  $k = 2$ , and the set  $\{1, 2, 3, 4, 5, 6\}$ , there are  $\binom{7!}{2! \cdot 5!} = 21$  combinations with repetitions:

$$\begin{aligned} & \{1, 1\}, \{1, 2\}, \{1, 3\}, \{1, 4\}, \{1, 5\}, \{1, 6\}, \{2, 2\}, \\ & \{2, 3\}, \{2, 4\}, \{2, 5\}, \{2, 6\}, \{3, 3\}, \{3, 4\}, \{3, 5\}, \\ & \{3, 6\}, \{4, 4\}, \{4, 5\}, \{4, 6\}, \{5, 5\}, \{5, 6\}, \{6, 6\}. \end{aligned}$$

**Example 6.13** With  $n = 3$ ,  $k = 2$ , and the set  $\{A, B, C\}$ , there are  $\binom{4!}{2! \cdot 2!} = 6$  combinations with repetitions:

$$\{A, A\}, \{A, B\}, \{A, C\}, \{B, B\}, \{B, C\}, \{C, C\}.$$

## Implementation

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
5 int combination_repetition(int *, int *, int, int, int, int, int);
6
7 /*
8 * main program
9 */
10 int main(int argc, char *argv[]) {
11     const int n = 3;
12     const int k = 2;
13     int value[] = {1, 2, 3};
14     int solution[] = {0, 0};
15     int pos = 0;
16     int start = 0;
17     int count = 0;
18     int total;
19
20     fprintf(stdout, "Combinations with repetitions:\n");
21     total = combination_repetition(value, solution, n, k, start, pos, count);
22     fprintf(stdout, "\nTotal Number: %d\n", total);
23
24     return EXIT_SUCCESS;
25 }
26
27 /*
28 * Combinations with Repetitions
29 */
30 int combination_repetition(
31     int *value, int *solution, int n, int m, int start, int pos, int count
32 )
33 {
34     int i;
35
36     if (pos >= m) {

```

```

37     fprintf(stdout, "{ ");
38     for (i=0; i<m; i++) {
39         fprintf(stdout, "%d ", solution[i]);
40     }
41     fprintf(stdout, "}");
42     return count+1;
43 }
44
45 for (i=start; i<n; i++) {
46     solution[pos] = value[i];
47     count = combination_repetition(value, solution, n, m, i, pos+1, count);
48 }
49
50 return count;
51 }
```

## 6.5 Binary Numbers

### Specifications

Write a program to generate binary numbers with  $n$  bits. Let  $n$  be read from standard input.

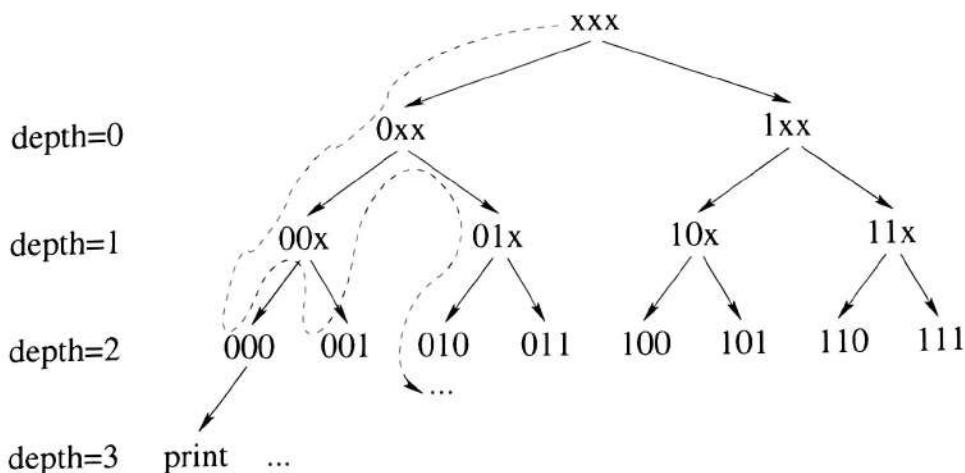
**Example 6.14** For example, with  $n = 3$ , the program has to generate and print-out (in any order): 000, 001, 010, 011, 100, 101, 110, 111.

Notice that the program does not have to perform any sort of numeric conversion (such as decimal to binary conversions) as it must generate binary numbers in a recursive way.

### Solution

All numbers have to be printed-out with all  $n$  bits. Bits have then to be stored in an array of length  $n$ . As  $n$  is defined only at run time, the array must be dynamically allocated.

With  $n = 3$  a possible implementation may work as represented by the following recursion tree:



```

2 #include <stdlib.h>
3
4 #define BASE 2
5 /* function prototypes */
6 void binary_r (int *array, int depth, int n);
7
8 /*
9  * main program
10 */
11 int main (void) {
12     int n, *array;
13
14     fprintf(stdout, "Number of bits: ");
15     scanf("%d", &n);
16
17     array = (int *)malloc(n * sizeof(int));
18     if (array==NULL) {
19         fprintf (stderr, "Memory allocation error.\n");
20         exit(EXIT_FAILURE);
21     }
22     fprintf(stdout, "Binary numbers\n");
23     binary_r(array, 0, n);
24     free(array);
25
26     return EXIT_SUCCESS;
27 }
28
29 /*
30  * binary numbers generation, recursive function
31 */
32
33 void binary_r (int *array, int depth, int n) {
34     int i;
35
36     if (depth >= n) {
37         for (i=0; i<n; i++) {
38             fprintf(stdout, "%d", array[i]);
39         }
40         fprintf(stdout, "\n");
41         return;
42     }
43
44     for (i=0; i<BASE; i++) {
45         array[depth] = i;
46         binary_r(array, depth+1, n);
47     }
48 }
```

## 6.6 Football Pool Specifications

In a football pool three symbols indicate the three possible results:

- ▷ 1: home team win.
- ▷ 2: home team lose.
- ▷ X: there is a draw.

Write a recursive program to manipulate a football system, i.e., to generate all possible patterns given a list of match results stored in a file. For example the following file:

```
1
1
X2
2
12X
```

specify the results of 5 matches (one for each row of the file), in which the home team win for the first 2 matches, there is a draw or the home team lose in match number 3, etc.

The recursive program, once read the file name from standard input, must generate all following patterns:

```
1 1 1 1 1
1 1 1 1 1
X X X 2 2 2
2 2 2 2 2 2
1 2 X 1 2 X
```

and it should print them on standard output with the following format:

```
11X21
11X22
11X2X
11221
11222
1122X
```

## Solution

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 /* function prototypes */
6 char **read_file(int *dim);
7 void play_r(char **scheme, char *column, int n, int row, FILE *fp);
8
9 /*
10 * main program
11 */
12 int main (void) {
13     char *column, **scheme;
14     char filename[20];
15     FILE *fp;
16     int n, i;
17
18     scheme = read_file(&n);
19     column = (char *)malloc((n+1) * sizeof(char));
20     if (column==NULL) {
21         fprintf (stderr, "Memory allocation error.\n");
```

```
22     exit(EXIT_FAILURE);
23 }
24 column[n] = '\0';
25 fprintf(stdout, "Output file name? ");
26 scanf("%s", filename);
27 fp = fopen(filename, "w");
28 play_r(scheme, column, n, 0, fp);
29 fclose(fp);
30
31 for (i=0; i<n; i++) {
32     free(scheme[i]);
33 }
34 free(scheme);
35 free(column);
36
37 return EXIT_SUCCESS;
38 }
39
40 /*
41 * load the input scheme from file
42 */
43 char **read_file (int *dim) {
44     char filename[20], line[10], **scheme;
45     int i, n=0;
46     FILE *fp;
47
48     fprintf(stdout, "Input file name? ");
49     scanf("%s", filename);
50     fp = fopen(filename, "r");
51
52     /* count the number of rows */
53     while (fscanf(fp, "%s", line) != EOF) {
54         n++;
55     }
56     fclose(fp);
57
58     /* read and store the input scheme */
59     scheme = (char **)malloc(n * sizeof(char *));
60     if (scheme==NULL) {
61         fprintf (stderr, "Memory allocation error.\n");
62         exit(EXIT_FAILURE);
63     }
64     fp = fopen(filename, "r");
65     if (fp==NULL) {
66         fprintf (stderr, "Open file allocation error.\n");
67         exit(EXIT_FAILURE);
68     }
69     for (i=0; i<n; i++) {
70         fscanf(fp, "%s", line);
71         scheme[i] = strdup(line);
72     }
73     fclose(fp);
74
75     *dim = n;
76     return scheme;
77 }
78
79 /*
80 */
```

```

81     * column expansion, recursive function
82 */
83 void play_r (char **scheme, char *column, int n, int row, FILE *fp) {
84     int i;
85
86     if (row >= n) {
87         fprintf(fp, "%s\n", column);
88         return;
89     }
90
91     for (i=0; i<strlen(scheme[row]); i++) {
92         column[row] = scheme[row][i];
93         play_r(scheme, column, n, row+1, fp);
94     }
95
96     return;
97 }
```

## 6.7 Anagrams Generation

### Specifications

Given a word or phrase, an *anagram* is the result of rearranging the letters to produce a new word or phrase, using all the original letters exactly once.

Write a program able to read a string (of maximum length equal to 10), to generate and to print-out all possible anagrams of the string.

Write two versions of the program. The first one must print all anagrams (even if they are duplicated), the second one must print all distinct anagrams.

**Example 6.15** If the input string is ABC, both program versions have to produce the following strings:

ABC ACB BAC CAB BCA CBA

**Example 6.16** If the input string is AAC, the first program version has to produce strings

AAC AAC ACA ACA CAA CAA

whereas the second one has to produce strings

AAC ACA CAA

### Solution 1

Following Section 6.4 the program has to generate all simple permutations of the given set of characters.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define MAX 10
6
7 /* function prototypes */
8 void generate_r (char *, char *, int *, int, int);
9
10 /*
11  * main program
12 */
```

```

13 int main (void) {
14     char word[MAX], anagram[MAX];
15     int used[MAX];
16     int length, i;
17     fprintf(stdout, "Input a word: ");
18     scanf("%s", word);
19     length = strlen(word);
20     anagram[length] = '\0';
21     for (i=0; i<length; i++) {
22         used[i] = 0;
23     }
24 }
25 fprintf(stdout, "Anagrams:\n");
26 generate_r(word, anagram, used, length, 0);
27 return EXIT_SUCCESS;
28 }
29
30 /*
31 * add a char in position n of the anagram taking it from
32 * the (unused) available chars in the word
33 */
34 void generate_r (char *word, char *anagram, int *used, int length, int n) {
35     int i;
36
37     if (n == length) {
38         fprintf(stdout, " %s\n", anagram);
39         return;
40     }
41
42     for (i=0; i<length; i++) {
43         if (used[i] == 0) {
44             used[i] = 1;
45             anagram[n] = word[i];
46             generate_r (word, anagram, used, length, n+1);
47             used[i] = 0;
48         }
49     }
50 }
51
52 return;
53 }

```

## Solution 2

Following Section 6.4 the program has to generate all permutations with repetitions of the given set of characters.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MAX 10
5 #define NUM 256
6
7 /* function prototypes */
8 int stat (char *, int *);
9 void generate_r (int *, char *, int, int);
10
11 /*
12 * main program

```

```
13  */
14 int main (void) {
15     int length, frequency[NUM];
16     char word[MAX], anagram[MAX];
17
18     fprintf(stdout, "Input a word: ");
19     scanf("%s", word);
20     length = stat(word, frequency);
21     anagram[length] = '\0';
22
23     fprintf(stdout, "Anagrams:\n");
24     generate_r(frequency, anagram, length, 0);
25     return EXIT_SUCCESS;
26 }
27
28 /*
29 * compute the frequencies of the characters in a word
30 * return the length of the word
31 */
32 int stat (char *s, int *f) {
33     int i;
34
35     for (i=0; i<NUM; i++) {
36         f[i] = 0;
37     }
38
39     for(i=0; s[i]!='\0'; i++) {
40         f[(int)s[i]]++;
41     }
42
43     return i;
44 }
45
46 /*
47 * add a char in position n of the anagram taking it from the pool
48 * of available chars described by the frequency parameter
49 */
50 void generate_r (int *frequency, char *anagram, int length, int n) {
51     int c;
52
53     if (n >= length) {
54         fprintf(stdout, "%s\n", anagram);
55         return;
56     }
57
58     for (c=0; c<NUM; c++) {
59         if (frequency[c] > 0) {
60             /* available char */
61             anagram[n] = c;
62             frequency[c]--;
63             generate_r(frequency, anagram, length, n+1);
64             frequency[c]++;
65         }
66     }
67
68     return;
69 }
```

## 6.8 The power-set

### Problem definition

Given a set  $S$  of  $k$  elements ( $k = \text{card}(S)$ ), its power-set  $\text{Pow}(S)$  is the set of the subsets of  $S$ , including  $S$  itself and the empty set.

**Example 6.17** The power-set of the set  $\{1, 2, 3, 4\}$  includes 16 elements:

$\{\}, \{4\}, \{3\}, \{3, 4\}, \{2\}, \{2, 4\}, \{2, 3\}, \{2, 3, 4\}, \{1\},$   
 $\{1, 4\}, \{1, 3\}, \{1, 3, 4\}, \{1, 2\}, \{1, 2, 4\}, \{1, 2, 3\}, \{1, 2, 3, 4\}.$

### Specifications

Write a program that generates the power-set of a set of objects.

### Solution

We present three different solutions. The first one generates the power-set using a *divide-and-conquer* approach. The second one is based on *arrangements with repetitions*. The last one uses *simple combinations*.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
5 int powerset_ver1 (int *, int, int, int, int, int);
6 int powerset_ver2 (int *, int, int, int, int, int);
7 int powerset_ver3 (int *, int *, int);
8 int powerset_ver3_r (int *, int *, int, int, int, int);
9
10 /*
11 * main program
12 */
13 int main (int argc, char *argv[]) {
14     const int k = 4;
15     int value[] = {1, 2, 3, 4};
16     int solution[4] = {0, 0, 0, 0};
17     int pos = 0;
18     int start = 0;
19     int count = 0;
20     int total;
21
22     fprintf(stdout, "Powerset Version 01:\n");
23     total = powerset_ver1 (value, solution, pos, k, start, count);
24     printf(stdout, "\nTotal Number: %d\n", total);
25
26     fprintf(stdout, "Powerset Version 02:\n");
27     powerset_ver2 (value, solution, pos, k, count);
28     printf(stdout, "\nTotal Number: %d\n", total);
29
30     fprintf(stdout, "Powerset Version 03:\n");
31     total = powerset_ver3 (value, solution, k);
32     printf(stdout, "\nTotal Number: %d\n", total);
33
34 }
35 return EXIT_SUCCESS;
36

```

```
37  /*
38   * Powerset Version 01
39   */
40  int powerset_ver1 (
41      int *value, int *solution, int pos, int k, int start, int count
42  ) {
43      int i;
44
45      if (start >= k) {
46          fprintf(stdout, "{ ");
47          for (i=0; i<pos; i++) {
48              fprintf(stdout, "%d ", solution[i]);
49          }
50          fprintf(stdout, "}");
51          return count+1;
52      }
53
54      for (i=start; i<k; i++) {
55          solution[pos] = value[i];
56          count = powerset_ver1(value, solution, pos+1, k, i+1, count);
57      }
58      count = powerset_ver1(value, solution, pos, k, k, count);
59      return count;
60  }
61
62  /*
63   * Powerset Version 02
64   */
65  int powerset_ver2 (
66      int *value, int *solution, int pos, int k, int count
67  ) {
68      int i;
69
70      if (pos >= k) {
71          printf("{ ");
72          for (i=0; i<k; i++) {
73              if (solution[i]!=0) {
74                  printf("%d ", value[i]);
75              }
76          }
77          printf("}");
78          return count+1;
79      }
80
81      solution[pos] = 0;
82      count = powerset_ver2 (value, solution, pos+1, k, count);
83      solution[pos] = 1;
84      count = powerset_ver2 (value, solution, pos+1, k, count);
85
86      return count;
87  }
88
89  /*
90   * Powerset Version 03
91   */
92  int powerset_ver3 (int *value, int *solution, int k) {
93      int count, i;
94
95      count = 0;
```

```

96     for (i=0; i<=k; i++) {
97         count += powerset_ver3_r (value, solution, k, i, 0, 0);
98     }
99     return count;
100 }
101
102 int powerset_ver3_r (
103     int *value, int *solution, int k, int n, int pos, int start
104 ) {
105     int count = 0, i;
106
107     if (pos == n) {
108         printf("{ ");
109         for (i=0; i<n; i++) {
110             printf(" %d ", solution[i]);
111         }
112         printf(" }");
113         return 1;
114     }
115
116     for (i=start; i<k; i++) {
117         solution[pos] = value[i];
118         count += powerset_ver3_r (value, solution, k, n, pos+1, i+1);
119     }
120
121     return count;
122 }
123 }
```

## 6.9 Partition of a set

### Problem definition

Given a set  $S$  of  $n$  elements, a collection  $S = \{S_i\}$  of non empty blocks forms a partition of  $S$  if and only if both the following conditions hold:

- ▷ Blocks are pairwise disjoint:

$$\forall S_i, S_j \in S \text{ with } i \neq j \quad \text{then} \quad S_i \cap S_j = \emptyset$$

- ▷ Their union is  $S$ :

$$S = \bigcup_i S_i$$

The number of blocks  $k$  ranges from 1 to  $n$ . In the first case, the block coincides with  $S$ . In the second one, each block contains only 1 element of  $S$ .

There are two possibilities to represent partitions:

- ▷ Given the element, identify the unique block it belongs to.
- ▷ Given the block, list the elements that belong to it.

the following examples illustrates the difference between these two strategies.

**Example 6.18** Given 4 elements  $value = \{1, 2, 3, 4\}$  there are 14 possibilities to partition them in 2 sets. For each partition the following table indicates the block to which each element belong

to (left-hand side), and the elements within each block (right-hand side):

Partition 1:	$\{0001\}$	$\rightarrow$	$\{123\}$	$\{4\}$
Partition 2:	$\{0010\}$	$\rightarrow$	$\{124\}$	$\{3\}$
Partition 3:	$\{0011\}$	$\rightarrow$	$\{12\}$	$\{34\}$
Partition 4:	$\{0100\}$	$\rightarrow$	$\{134\}$	$\{2\}$
Partition 5:	$\{0101\}$	$\rightarrow$	$\{13\}$	$\{24\}$
Partition 6:	$\{0110\}$	$\rightarrow$	$\{14\}$	$\{23\}$
Partition 7:	$\{0111\}$	$\rightarrow$	$\{1\}$	$\{234\}$
Partition 8:	$\{1000\}$	$\rightarrow$	$\{234\}$	$\{1\}$
Partition 9:	$\{1001\}$	$\rightarrow$	$\{23\}$	$\{14\}$
Partition 10:	$\{1010\}$	$\rightarrow$	$\{24\}$	$\{13\}$
Partition 11:	$\{1011\}$	$\rightarrow$	$\{2\}$	$\{134\}$
Partition 12:	$\{1100\}$	$\rightarrow$	$\{34\}$	$\{12\}$
Partition 13:	$\{1101\}$	$\rightarrow$	$\{3\}$	$\{124\}$
Partition 14:	$\{1110\}$	$\rightarrow$	$\{4\}$	$\{123\}$

Many of these partitions are symmetric. If we keep the first sample for each symmetric class we obtain:

Partition 1:	$\{0001\}$	$\rightarrow$	$\{123\}$	$\{4\}$
Partition 2:	$\{0010\}$	$\rightarrow$	$\{124\}$	$\{3\}$
Partition 3:	$\{0011\}$	$\rightarrow$	$\{12\}$	$\{34\}$
Partition 4:	$\{0100\}$	$\rightarrow$	$\{134\}$	$\{2\}$
Partition 5:	$\{0101\}$	$\rightarrow$	$\{13\}$	$\{24\}$
Partition 6:	$\{0110\}$	$\rightarrow$	$\{14\}$	$\{23\}$
Partition 7:	$\{0111\}$	$\rightarrow$	$\{1\}$	$\{234\}$

that is, we have 7 different partitions.

## Specifications

Write a program able to partition a set following the previous definitions. Generate all sets, without worrying about symmetric blocks.

## Solution

The core idea is to use the arrangements with repetitions model. Referring to Section 6.8 we can generalize the power-set generation based on arrangements with repetitions making each choice varying within the available blocks.

Among the two possible ways to represent partitions, the following solution identify the block for each element, as this strategy works with a simple array of integers and does not require lists.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
5 int partition (int *, int *, int, int, int);
6
7 /*
8 * main program
9 */
10 int main (int argc, char *argv[]) {
11     int value[4] = {1, 2, 3, 4};
12     int solution[4] = {0, 0, 0, 0};

```

```

13 int n = 4, k = 2, pos = 0, count = 0;
14 int i, total;
15
16 fprintf(stdout, "Partition (%d elements = ", n);
17 for (i=0; i<n; i++) {
18     fprintf(stdout, "%d ", value[i]);
19 }
20 fprintf(stdout, ")\n");
21 total = partition (value, solution, n, k, pos, count);
22 fprintf(stdout, "\nTotal Number: %d\n", total);
23
24 return EXIT_SUCCESS;
25 }
26
27 /*
28 * Partition
29 */
30 int partition(
31     int *value, int *solution, int n, int k, int pos, int count
32 ) {
33     int i, j, end, *occurrences;
34
35     if (pos >= n) {
36         occurrences = calloc (k, sizeof (int));
37         if (occurrences == NULL) {
38             fprintf (stderr, "Memory allocation error.\n");
39             exit(EXIT_FAILURE);
40         }
41         /* I do not need to initialize occurrences as I used calloc */
42         for (j=0; j<n; j++) {
43             occurrences[solution[j]]++;
44         }
45         for (end=j=0; j<k && end==0; j++) {
46             if (occurrences[j]==0) {
47                 end = 1;
48             }
49         }
50         free (occurrences);
51         if (end==1) {
52             return count;
53         }
54         printf (stdout, "Partition %2d: ", count+1);
55         printf (stdout, "{ ");
56         for (i=0; i<n; i++) {
57             printf("%d ", solution[i]);
58         }
59         printf (stdout, "} -> ");
60         for (i=0; i<k; i++) {
61             printf("{ ");
62             for (j=0; j<n; j++) {
63                 if (solution[j]==i)
64                     printf("%d ", value[j]);
65             }
66             printf("}   ");
67         }
68         printf("\n");
69         return count+1;
70     }
71 }
```

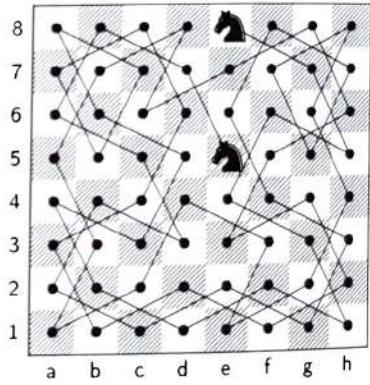
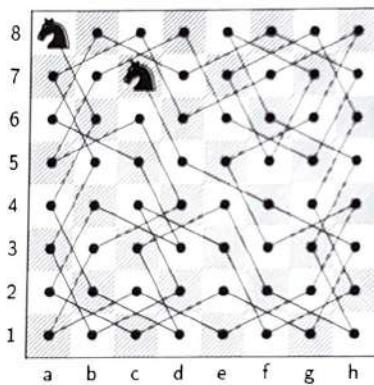
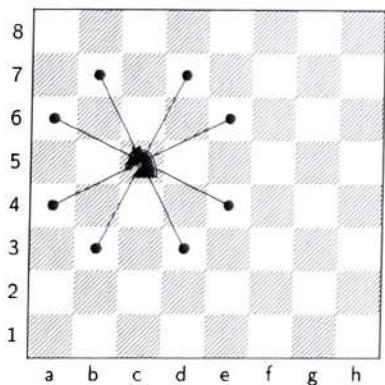
```
72     for (i=0; i<k; i++) {
73         solution[pos] = i;
74         count = partition (value, solution, n, k, pos+1, count);
75     }
76
77     return count;
78 }
```

## 6.10 The Knight Tour

## Specifications

A knight's tour is a sequence of moves of a knight on a chessboard such that the knight visits every square only once. If the knight ends on a square that is one knight's move from the beginning square (so that it could tour the board again immediately, following the same path), the tour is closed, otherwise it is open.

We recall that a chess knight moves on the board “drawing” L letters. The knight’s moves (left-hand side), and two open tours (center and right-hand side) are represented in the following picture.



The knight's tour problem is the mathematical problem of finding a knight's tour. Variations of the knight's tour problem involve chessboards of different sizes than the usual  $(8 \times 8)$ , as well as irregular (non-rectangular) boards.

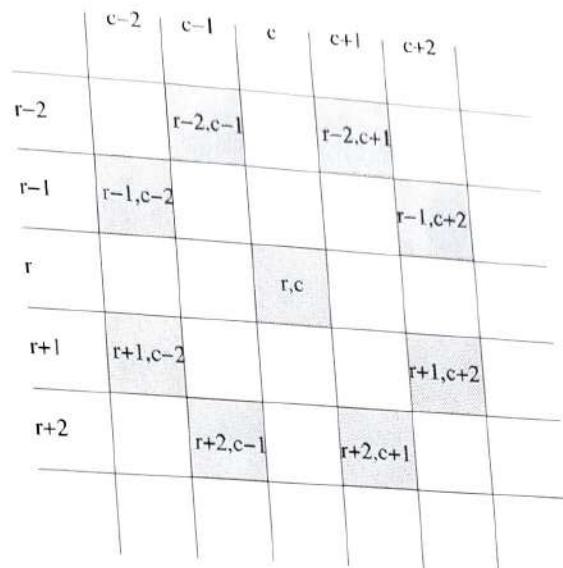
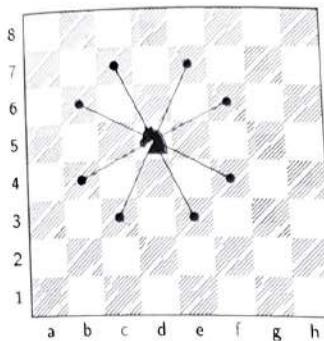
Create a program to find a knight's tour. Adopt a chess board of size  $N \times N$ , where  $N$  is a pre-defined constant value. Read the starting cell position  $[i][j]$  from standard input.

## Solution 1

First off all, the following picture represents how the knight can move on the board. Every time its starting position is cell  $[r][c]$  all other marked cells indicate possible ending position.

Those movements can be implemented using the arrays named `offsetX` and `offsetY` (defined as global objects).

On an  $(8 \cdot 8)$  board there are approximately  $4 \cdot 10^{51}$  possible move sequences, and about  $2.6 \cdot 10^{13}$  directed closed tours (i.e., two tours along the same path that travel in opposite directions are counted separately, as are rotations and reflections).



A brute-force search for a knight's tour is impractical on all but the smallest boards, as it is well beyond the capacity of modern computers (or networks of computers) to perform operations on such a large set. The following is an example of a brute-force solution without any optimization a "deeper human insight" can suggest. Solution on a  $8 \times 8$  board are found in a reasonable time only on lucky cases.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N 6
5
6 const int offsetX[] = {2, 1, -1, -2, -2, -1, 1, 2};
7 const int offsetY[] = {1, 2, 2, 1, -1, -2, -2, -1};
8
9 /* function prototypes */
10 int move_r(int, int, int, int [N][N]);
11
12 /*
13 * main program
14 */
15 int main (void) {
16     int i, j, board[N][N];
17
18     for (i=0; i<N; i++) {
19         for (j=0; j<N; j++) {
20             board[i][j] = 0;
21         }
22     }
23     printf(stdout, "Initial position? ");
24     scanf("%d %d", &i, &j);
25     board[i][j] = 1;
26
27     if (move_r(2, i, j, board)) {
28         printf(stdout, "Solution found:\n");
29         for (i=0; i<N; i++) {
30             for (j=0; j<N; j++) {
31                 printf(stdout, "%2d ", board[i][j]);
32             }
33         }
34     }
35 }
```

```

33         fprintf(stdout, "\n");
34     }
35 } else {
36     fprintf(stdout, "Solution NOT found!\n");
37 }
38
39 return EXIT_SUCCESS;
40 }
41
42 /*
43 * compute a knight tour, recursive function
44 */
45 int move_r (int level, int x, int y, int board[N][N]) {
46     int i, xx, yy;
47
48     if (level == N*N+1) {
49         return 1;
50     }
51
52     for (i=0; i<8; i++) {
53         xx = x + offsetX[i];
54         yy = y + offsetY[i];
55         if (xx<N && xx>=0 && yy<N && yy>=0) {
56             if (board[xx][yy] == 0) {
57                 board[xx][yy] = level;
58                 if (move_r(level+1, xx, yy, board)) {
59                     return 1;
60                 }
61                 board[xx][yy] = 0;
62             }
63         }
64     }
65
66     return 0;
67 }
```

## Solution 2

The following implementation uses an heuristic approach to optimize the solution usually known as the Warnsdorf's rule.

Following the Warnsdorf's rule the knight is moved so that it always proceeds to the square from which the knight will have the fewest onward moves. When calculating the number of onward moves for each candidate square, we do not count moves that revisit any square already visited. It is, of course, possible to have two or more choices for which the number of onward moves is equal; there are various methods for breaking such ties, but we will not enter into those issue.

The following implementation uses the array `count` to order all possible moves accordingly to their onwards moves. The first `for` cycle compute the array, whereas the second one uses it to decide how to recur. The solution is much more efficient than the previous one, and it is able to find solution also for quite large board.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N    12
5 #define MOVE  8
6
```

```

7   const int offsetX[] = {2, 1, -1, -2, -2, -1, 1, 2};
8   const int offsetY[] = {1, 2, 2, 1, -1, -2, -2, -1};
9   /* function prototypes */
10  int move_r(int, int, int, int [N][N]);
11
12  /*
13   * main program
14   */
15  int main (void) {
16      int i, j, board[N][N];
17
18      for (i=0; i<N; i++) {
19          for (j=0; j<N; j++) {
20              board[i][j] = 0;
21          }
22      }
23
24      fprintf(stdout, "Initial position? ");
25      scanf("%d %d", &i, &j);
26      board[i][j] = 1;
27
28      if (move_r(2, i, j, board)) {
29          fprintf(stdout, "Solution found:\n");
30          for (i=0; i<N; i++) {
31              for (j=0; j<N; j++) {
32                  fprintf(stdout, "%2d ", board[i][j]);
33              }
34              fprintf(stdout, "\n");
35          }
36      } else {
37          fprintf(stdout, "Solution NOT found!\n");
38      }
39
40      return EXIT_SUCCESS;
41  }
42
43  /*
44   * compute a knight tour, recursive function
45   */
46  int move_r (int level, int x, int y, int board[N][N]) {
47      int i, j, xx, yy, xxx, yyy;
48      int min, imin, count[MOVE];
49
50      if (level == N*N+1) {
51          return 1;
52      }
53
54      for (i=0; i<MOVE; i++) {
55          xx = x + offsetX[i];
56          yy = y + offsetY[i];
57          if (xx<N && xx>=0 && yy<N && yy>=0 && board[xx][yy]==0) {
58              count[i] = 0;
59              for (j=0; j<MOVE; j++) {
60                  xxx = xx + offsetX[j];
61                  yyy = yy + offsetY[j];
62                  if (xxx<N && xxx>=0 && yyy<N && yyy>=0 && board[xxx][yyy]==0) {
63                      count[i]++;
64                  }
65              }
66          }
67      }
68  }

```

```

66      }
67  }
68  for (i=0; i<MOVE; i++) {
69    min = MOVE; imin = MOVE;
70    for (j=0; j<MOVE; j++) {
71      if (count[j]<min) {
72        min = count[j];
73        imin = j;
74      }
75    }
76    if (imin<MOVE) {
77      xx = x + offsetX[imin];
78      yy = y + offsetY[imin];
79      if (xx<N && xx>=0 && yy<N && yy>=0) {
80        if (board[xx][yy] == 0) {
81          board[xx][yy] = level;
82          if (move_r(level+1, xx, yy, board)) {
83            return 1;
84          }
85          board[xx][yy] = 0;
86        }
87      }
88      count[imin] = MOVE;
89    } else {
90      break;
91    }
92  }
93 }
94
95 return 0;
96 }
```

## 6.11 The Eight Queen Problem

### Problem definition

The eight queens puzzle is the problem of placing eight chess queens on an  $8 \times 8$  chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

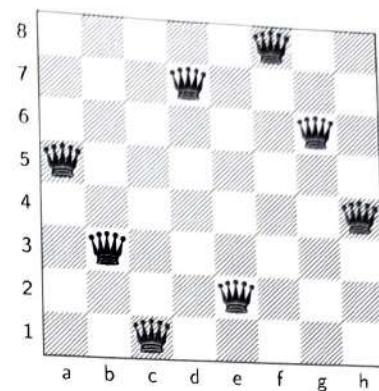
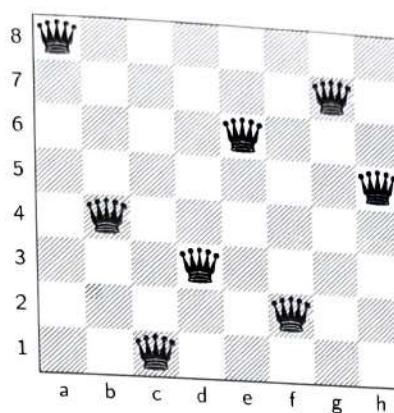
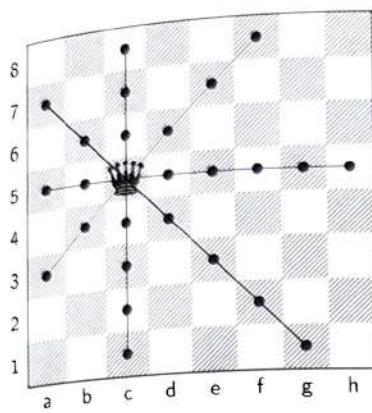
Notice that the eight queens puzzle is an example of the more general  $n$  queens problem of placing  $n$  non-attacking queens on an  $n \times n$  chessboard, for which solutions exist for all natural numbers  $n$  with the exception of  $n = 2$  and  $n = 3$ .

Chess composer Max Bezzel published the eight queens puzzle in 1848. Franz Nauck published the first solutions in 1850. Nauck also extended the puzzle to the  $n$  queens problem. Since then, many mathematicians have worked on both the eight queens puzzle and its generalized  $n$ -queens version.

Two possible solutions for the eight-queen problem are represented in the following picture, together with the threatened cells (left-hand side) given a single queen on the board:

### Considerations on Complexity

The problem can be quite computationally expensive. First of all, it is important to avoid very trivial approaches:



- ▷ Using *arrangements with repetitions* we could set a queen (1) or not (0) in each one of the 64 cells. Thus, we would have  $n = 2$  objects (queen or not) on  $k = 64$  positions (cells), with an enormous space to investigate

$$D'_{n,k} = D'_{2,64} = 2^{64} = 1.84 \cdot 10^{19}$$

Luckily, this approach is highly redundant. In fact, as there is no reason to place on the board more than 8 queens, the majority of the previous arrangements would be useless but discarded only once they have been generated.

- ▷ Using *simple arrangements*, we could consider  $n = 64$  cells among which selecting  $k = 8$  cells. This approach would lead to:

$$D_{n,k} = D'_{64,8} = 64 \cdot 63 \cdots 77 = 1.78 \cdot 10^{14}$$

cases. Again, as there are no motivations to analyze symmetric solutions, i.e., queens are all identical, and there are no reasons to permuting (or swapping) them in any configuration (order does not matter) this approach would be ineffective. Thus, arrangements can be abandoned, and we can move to some approach based on the combinations model.

Once said that, a first attempt would try to place 8 queens on 64 cells meaning that it would find all possible simple combinations of 64 elements in 8 positions. This evaluates to

$$C_{n,k} = C_{64,8} = \frac{n!}{k!(n-k)!} = \frac{64!}{8! \cdot 56!} = 4,426,165,368 \approx 4.42 \cdot 10^9$$

Unfortunately, as it can be proved that there are only 92 solutions, this is still an enormous space to visit. Moreover, many solutions are symmetric under board rotations and reflections. If solutions that differ only by symmetry operations (rotations and reflections) of the board are counted as one, the puzzle has only 12 fundamental solutions. In fact, a fundamental solution usually has eight variants (including its original form) obtained by rotating 90, 180, or 270 degrees and then reflecting each of the four rotational variants in a mirror in a fixed position.

It is possible to use several shortcuts to reduce computational requirements, or rules of thumb to avoid brute-force computational techniques. We will analyze a few possibilities in the following pages.

## Specifications

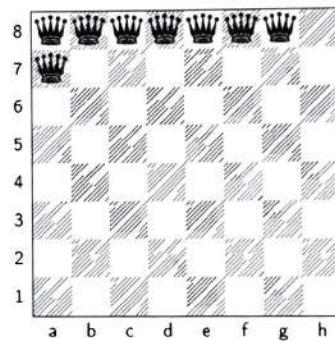
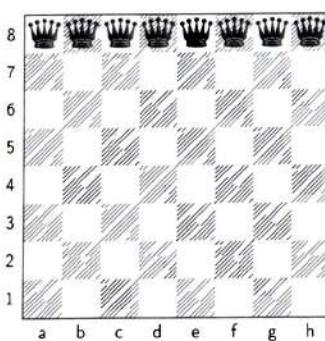
Write a program able to solve the eight-queen problem. Generalize the original problem to boards of size  $(N \cdot N)$ , where  $N$  is a pre-defined constant value.

## Solution 1

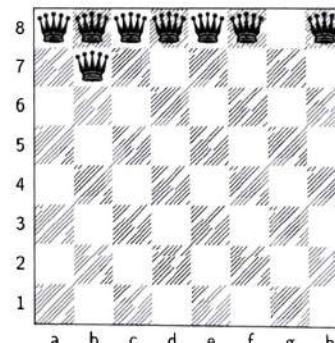
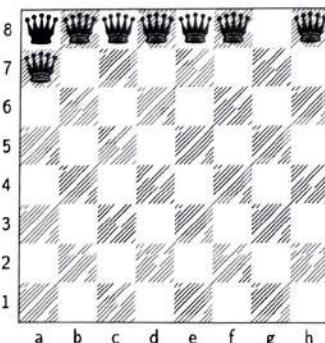
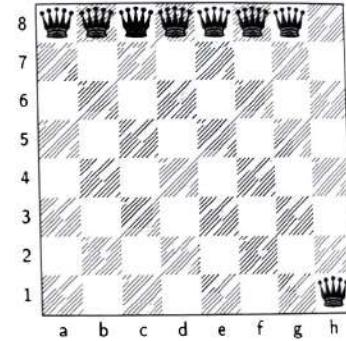
This first implementation:

- ▷ Generates the simple combinations of 8 queens on 64 cells.
- ▷ Checks all conditions (a single queen on each row, column, diagonal, and reverse diagonal) only once placed all queens on the board.

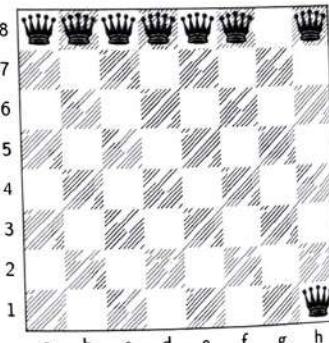
Unfortunately, as previously described, this solution generates many useless configurations. For example, Figure 6.2 shows the configurations generated during the first recursion paths. As it can be seen, all queens are placed on the same rows, and albeit there is no way this configuration satisfies our criteria, the algorithm keeps on going, as it just moves the last queens to look for a proper configuration. In this way, it may take several hours for the program to find a first valuable solution.



• • •



• • •



**Figure 6.2** Eight queen's Puzzle: Useless Moves to follow simple combinations.

A first attempt to improve this solution consists to mark cells already under attack. The next solution will clarify this idea.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N 8
5

```

```

6  /* function prototypes */
7  int place_r(int [] [N], int, int, int);
8  int check_solution(int board[] [N]);
9
10 /* main program
11 */
12 int main(void) {
13     int i, j;
14     int board[N] [N];
15
16     for (i=0; i<N; i++) {
17         for (j=0; j<N; j++) {
18             board[i] [j] = 0;
19         }
20     }
21
22     if (place_r(board, 0, -1, 0)) {
23         fprintf(stdout, "Solution found:\n");
24         for(i=0; i<N; i++) {
25             for(j=0; j<N; j++) {
26                 if (board[i] [j] != 0) {
27                     fprintf(stdout, "Q");
28                 } else {
29                     fprintf(stdout, ".");
30                 }
31             }
32             fprintf(stdout, "\n");
33         }
34     } else {
35         fprintf(stdout, "Solution NOT found!\n");
36     }
37 }
38
39 return EXIT_SUCCESS;
40 }
41
42 /*
43 * place N queens on the NxN board, recursive function
44 */
45 int place_r (int board[N] [N], int r0, int c0, int n) {
46     int r, c;
47
48     if (n == N) {
49         if (check_solution(board) == 1) {
50             return 1;
51         } else {
52             return 0;
53         }
54     }
55
56     for (r=r0; r<N; r++) {
57         for (c=((r==r0)?(c0+1):0); c<N; c++) {
58             // Set queen
59             board[r] [c] = n+1;
60             if (place_r (board, r, c, n+1) == 1) {
61                 return (1);
62             }
63             // Remove queen = backtrack
64             board[r] [c] = 0;

```

```
65      }
66  }
67
68  return 0;
69 }
70
71 /**
72 * check whether it is possible to set a queen in a given position
73 */
74 int check_solution (int board[N][N]) {
75     int r, c, d, n;
76
77     // check rows
78     for (r=0; r<N; r++) {
79         for (n=0, c=0; c<N; c++) {
80             if (board[r][c] != 0) {
81                 n++;
82             }
83         }
84         if (n>1) {
85             return 0;
86         }
87     }
88
89     // check columns
90     for (c=0; c<N; c++) {
91         for (n=0, r=0; r<N; r++) {
92             if (board[r][c] != 0) {
93                 n++;
94             }
95         }
96         if (n>1) {
97             return 0;
98         }
99     }
100
101    // check diagonals
102    for (d=0; d<2*N-1; d++) {
103        n = 0;
104        for (r=0; r<N; r++) {
105            c = d-r;
106            if ((c>=0) && (c<N)) {
107                if (board[r][c] != 0)
108                    n++;
109            }
110        }
111        if (n>1) {
112            return 0;
113        }
114    }
115
116    // check reverse diagonals
117    for (d=0; d<2*N-1; d++) {
118        n = 0;
119        for (r=0; r<N; r++) {
120            c = r-d+N-1;
121            if ((c>=0) && (c<N)) {
122                if (board[r][c] != 0) {
123                    n++;
```

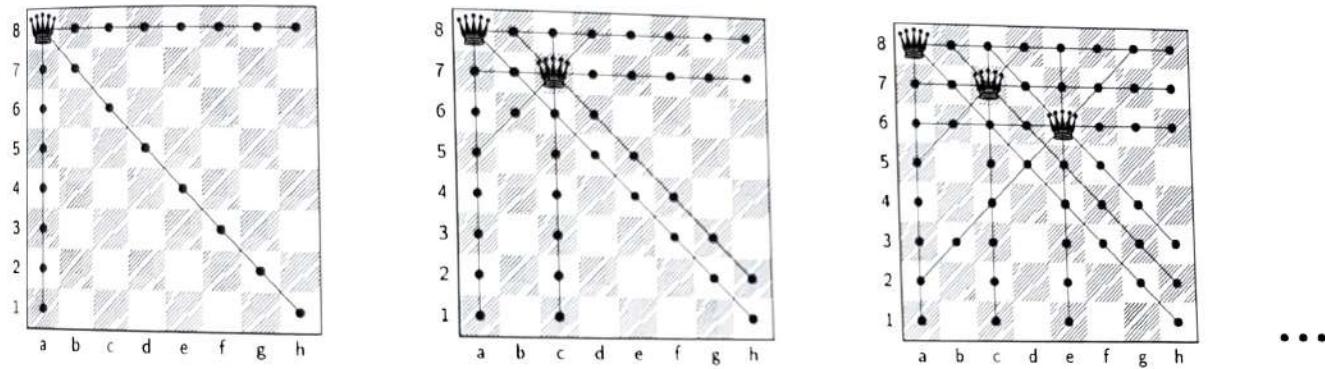
```

124 }
125 }
126 if (n>1) {
127     return 0;
128 }
129 }
130 }
131 return 1;
132 }
133 }
```

**Solution 2**

This second implementation places a queen on an empty cell which is also not under attack (if construct at line 56). To do that, for every queen placed on the board it marks all cells under attack (function `set_queen`).

Figure 6.3 shows this effect.



**Figure 6.3** Eight queen's Puzzle: Marked Cells.

As each cell can be under attack by more than one queen, marking is done by incrementing a counter related to the cell. Then, each cell of matrix `board`, is used as a counter incremented for each new queen placed on the board and decremented for each queen removed from the board. If a solution is not found (lines 58÷60) backtrack is performed on line 61.

As it can be argued, this solution is more or less equivalent to check for correctness after each queen has been placed on the board, instead of checking just after all queens have been placed. Moreover, the resulting scheme consists in placing each queen on a different row and on a different column, as no queen can stand on the same row and column of a previous one. Following this idea the next solution is the logic consequence.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N 8
5
6 /* function prototypes */
7 int move_r(int [N][N], int);
8 void set_queen(int [N][N], int, int, int);
```

```
10  /*
11   * main program
12   */
13  int main (void) {
14      int i, j;
15      int board[N][N];
16
17      for (i=0; i<N; i++) {
18          for (j=0; j<N; j++) {
19              board[i][j] = 0;
20          }
21      }
22
23      if (move_r(board, 0)) {
24          fprintf(stdout, "Solution found:\n");
25          for(i=0; i<N; i++) {
26              for(j=0; j<N; j++) {
27                  if (board[i][j] == 6) {
28                      fprintf(stdout, "Q");
29                  } else {
30                      fprintf(stdout, "*");
31                  }
32              }
33              fprintf(stdout, "\n");
34          }
35      } else {
36          fprintf(stdout, "Solution NOT found!\n");
37      }
38
39      return EXIT_SUCCESS;
40  }
41
42  /*
43   * place N queens on the NxN board, recursive function
44   */
45  int move_r (int board[N][N], int n) {
46      int r, c;
47
48      if (n == N) {
49          return 1;
50      }
51
52      for (r=0; r<N; r++) {
53          for (c=0; c<N; c++) {
54              if (board[r][c] == 0) {
55                  set_queen(board, r, c, +1);
56                  if (move_r(board, n+1)) {
57                      return 1;
58                  }
59                  set_queen(board, r, c, -1);
60              }
61          }
62      }
63
64      return 0;
65  }
66
67  /*
68   * mark all the reachable cells when a queen is placed in position r,c
69   */
```

```

69  /*
70   void set_queen (int board[N][N], int r, int c, int val) {
71     int i, j;
72     for (j=0; j<N; j++) {
73       board[r][j] += val;
74     }
75     for (i=0; i<N; i++) {
76       board[i][c] += val;
77     }
78     for (i=r, j=c; i<N && j<N; i++, j++) {
79       board[i][j] += val;
80     }
81     for (i=r, j=c; i>=0 && j>=0; i--, j--) {
82       board[i][j] += val;
83     }
84     for (i=r, j=c; i<N && j>=0; i++, j--) {
85       board[i][j] += val;
86     }
87     for (i=r, j=c; i>=0 && j<N; i--, j++) {
88       board[i][j] += val;
89     }
90   }
91 }
```

### Solution 3

This implementation positions queen number  $n$  on row number  $n$  (line 54). Moreover, as suggested in the previous section, before positioning each new queen (line 58), it checks whether the position is compatible with previously placed queens. Function `check_position` verifies whether there are no previous queens on the same column, diagonal, and reverse diagonal.

Notice that in this case, we apply the model of simple arrangements of 8 queen on the 8 columns. Actually, as each queen has to be on a different column this coincides with computing the permutation of 8 values, the ones which indicates the column. This evaluates to

$$D_{n,k} = D_{8,8} = P_8 = 8! = 40320$$

This solution can be further improved from the implementation point of view as described in the next section.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N 8
5
6 /* function prototypes */
7 int place_r (int [N][N], int);
8 int check_position (int board[N][N], int r, int c);
9
10 /*
11  * main program
12  */
13 int main (void) {
14   int i, j;
15   int board[N][N];
```

```
17     for (i=0; i<N; i++) {
18         for (j=0; j<N; j++) {
19             board[i][j] = 0;
20         }
21     }
22
23     if (place_r(board, 0)) {
24         fprintf(stdout, "Solution found:\n");
25         for(i=0; i<N; i++) {
26             for(j=0; j<N; j++) {
27                 if (board[i][j] != 0) {
28                     fprintf(stdout, "Q");
29                 } else {
30                     fprintf(stdout, ".");
31                 }
32             }
33             fprintf(stdout, "\n");
34         }
35     } else {
36         fprintf(stdout, "Solution NOT found!\n");
37     }
38
39     return EXIT_SUCCESS;
40 }
41
42 /*
43 * place N queens on the NxN board, recursive function
44 */
45 int place_r (int board[N][N], int n) {
46     int r, c;
47
48     if (n == N) {
49         return 1;
50     }
51
52     // Queen n is on row n
53     r = n;
54     // On which column (c) do I place queen n ?
55     for (c=0; c<N; c++) {
56         if (check_position(board, r, c) == 0) {
57             board[r][c] = 1;
58             if (place_r(board, n+1)) {
59                 return 1;
60             }
61             board[r][c] = 0;
62         }
63     }
64
65     return 0;
66 }
67
68 /*
69 * check whether it is possible to set a queen in a given position
70 */
71 int check_position (int board[N][N], int r, int c) {
72     int i, j;
73
74     /* check column */
75     for (i=0; i<r; i++) {
```

```

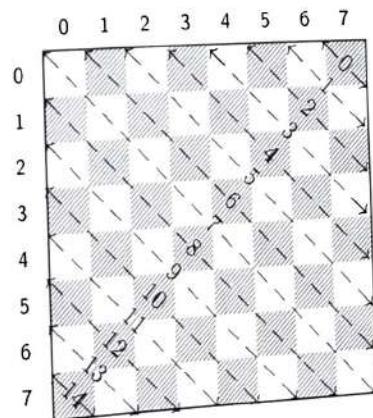
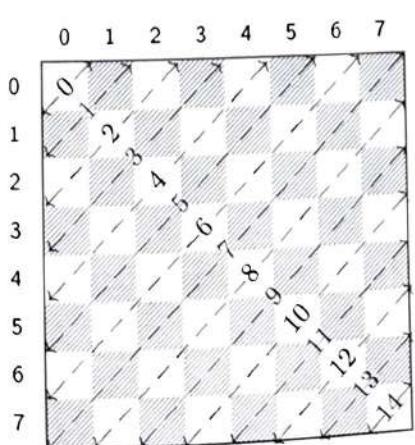
76     if (board[i][c] != 0) {
77         return 1;
78     }
79     /* check diagonal */
80     for (i=r, j=c; i>=0 && j>=0; i--, j--) {
81         if (board[i][j] != 0) {
82             return 1;
83         }
84     }
85     /* check reverse diagonal */
86     for (i=r, j=c; i>=0 && j<N; i--, j++) {
87         if (board[i][j] != 0) {
88             return 1;
89         }
90     }
91 }
92 return 0;
93 }
```

## Solution 4

This solution merges good ideas from the previous two solutions to obtain a more efficient implementation. From the one hand, the queen number  $n$  is placed on row number  $n$  (line 56). On the other one, all cells under attack are marked by function `set_queen`. To do this, instead of using a entire board of size  $n \cdot n$ , this implementation uses 4 arrays, `queen`, `col`, `diag`, and `revdiag`:

- ▷ `queen[i]` indicates the column on which queen  $i$  is placed.
- ▷ `col[j]` is equal to 1 only when the corresponding column is threaten by a queen.
- ▷ `diag[j]` and `revdiag[j]` are equal to 1 only when the corresponding diagonal and reverse diagonal are threaten by a queen.

The next picture shows how rows, columns, diagonals (left-hand side) and reverse diagonals (right-hand side) are enumerated and referenced within the corresponding arrays.



```

1 #include <stdio.h>
2 #include <stdlib.h>
```

```

4 #define N 8
5 #define FULL (+1)
6 #define EMPTY (-1)
7
8 /* function prototypes */
9 int place_r (int, int *, char *, char *, char *);
10 void set_queen (int, int *, char *, char *, char *);
11 void rem_queen (int, int *, char *, char *, char *);
12
13 /*
14 * main program
15 */
16 int main (void) {
17     char col[N], diag[2*N-1], revdiag[2*N-1];
18     int i, j, queen[N];
19
20     // Reset board (no queen at the beginning)
21     for (i=0; i<N; i++) {
22         queen[i] = col[i] = EMPTY;
23     }
24     for (i=0; i<2*N-1; i++) {
25         diag[i] = revdiag[i] = EMPTY;
26     }
27
28     if (place_r(0, queen, col, diag, revdiag)) {
29         fprintf(stdout, "Solution found:\n");
30         for(i=0; i<N; i++) {
31             for(j=0; j<N; j++) {
32                 if (queen[i] == j) {
33                     fprintf(stdout, "Q");
34                 } else {
35                     fprintf(stdout, ".");
36                 }
37             }
38             fprintf(stdout, "\n");
39         }
40     } else {
41         fprintf(stdout, "Solution NOT found!\n");
42     }
43
44     return EXIT_SUCCESS;
45 }
46
47 /*
48 * place N queens on the NxN board, recursive function
49 */
50 int place_r (int n, int *queen, char *col, char *diag, char *revdiag) {
51     int r, c;
52
53     // When n==N all queens have been placed on the board
54     if (n == N) {
55         return 1;
56     }
57
58     // Queen n is on row n
59     r = n;
60     // On which column (c) do I place queen n ?
61     for (c=0; c<N; c++) {
62         if (col[c]==EMPTY && diag[r+c]==EMPTY && revdiag[r-c+N-1]==EMPTY) {

```

```
63     set_queen(n, c, queen, col, diag, revdiag);
64     if (place_r(n+1, queen, col, diag, revdiag)) {
65         return 1;
66     }
67     rem_queen(r, c, queen, col, diag, revdiag);
68 }
69 }
70 return 0;
71 }

73 /*
74 * mark the reachable "cells" when a queen is placed in position i,j
75 */
76 void set_queen (int i, int j, int *queen, char *col, char *diag, char *revdiag) {
77     queen[i] = j;
78     col[j] = FULL;
79     diag[i+j] = FULL;
80     revdiag[i-j+N-1] = FULL;
81

82     return;
83 }

85 /*
86 * remove a queen from position i,j
87 */
88 void rem_queen (int i, int j, int *queen, char *col, char *diag, char *revdiag) {
89     queen[i] = EMPTY;
90     col[j] = EMPTY;
91     diag[i+j] = EMPTY;
92     revdiag[i-j+N-1] = EMPTY;
93
94     return;
95 }
```

# Chapter 7

## Recursion-based advanced problem solving tasks

In this chapter we analyze some more complex problem solving strategies based on recursion.

### 7.1 Towers of Hanoi

#### Specifications

The Towers of Hanoi (also called the Tower of Brahma or Lucas' Tower) is a mathematical game or puzzle. It consists of three rods, and a number of disks of different sizes which can slide onto any rod (see Figure 7.1). The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape. Picture 7.1 shows how the puzzle works.

The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

- ▷ Only one disk can be moved at a time.
- ▷ Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack, i.e. a disk can only be moved if it is the uppermost disk on a stack.
- ▷ No disk may be placed on top of a smaller disk.

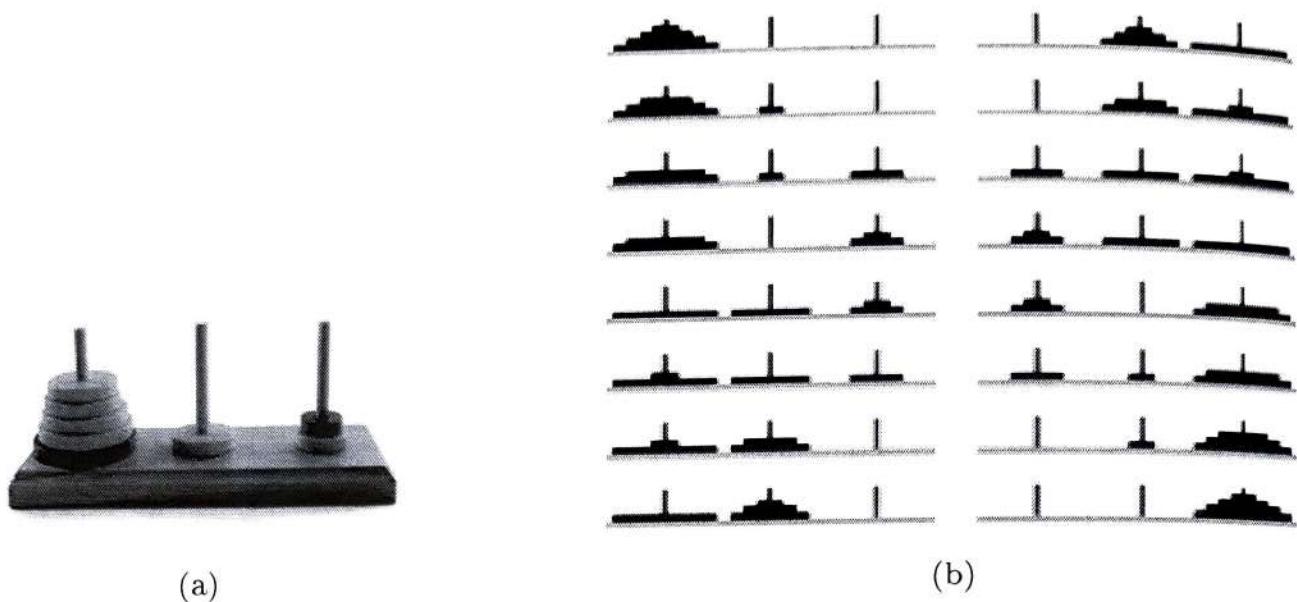
With three disks, the puzzle can be solved in seven moves. The minimum number of moves required to solve a Towers of Hanoi puzzle is  $2^n - 1$ , where  $n$  is the number of disks.

Write a program able to solve the Towers of Hanoi puzzle.

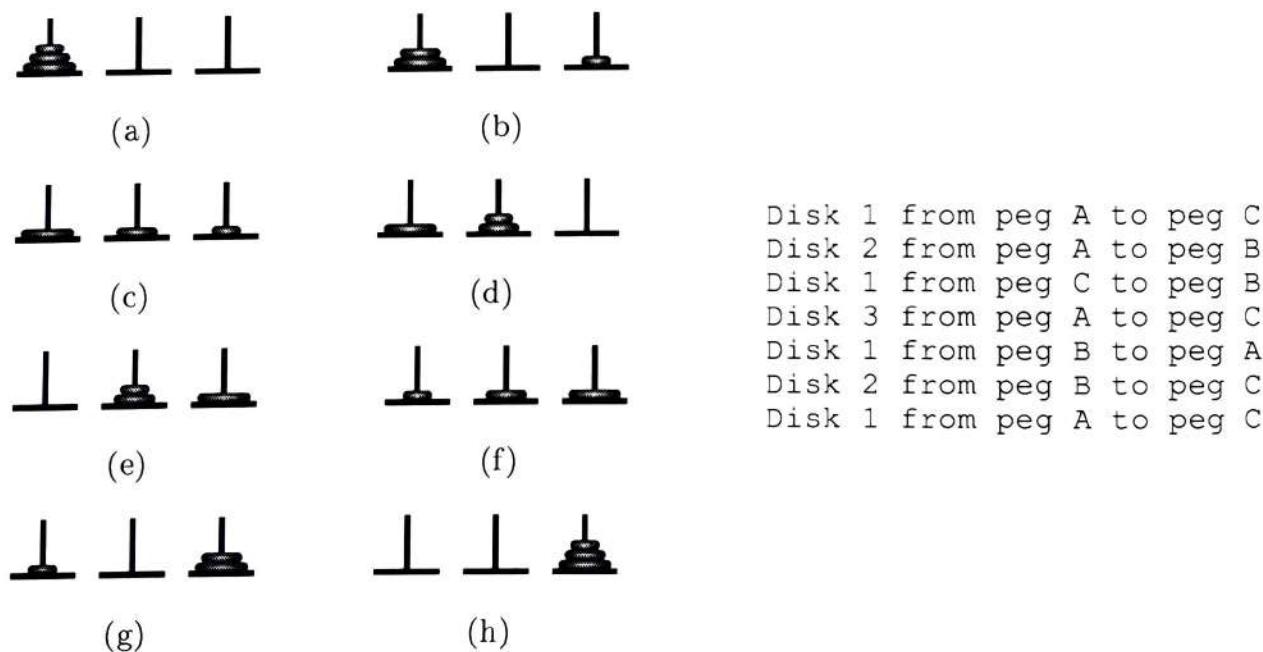
**Example 7.1** Figure 7.2 represents the sequence of moves that have to be performed when  $n = 3$ , and peg are indicated with letters A, B, and C.

**Example 7.2** The following is the sequence of moves that have to be performed when  $n = 4$  (peg are named A, B, and C):

Disk 1 from peg A to peg B  
Disk 2 from peg A to peg C



**Figure 7.1** The Towers of Hanoi Puzzle.



**Figure 7.2** A solution for the Towers of Hanoi Puzzle with 3 disks (1, 2, and 3) and 3 rods (A, B, and C). Diagrammatic representation (left-hand side from (a) to (h)) and sequence of moves (right-hand side).

```

Disk 1 from peg B to peg C
Disk 3 from peg A to peg B
Disk 1 from peg C to peg A
Disk 2 from peg C to peg B
Disk 1 from peg A to peg B
Disk 4 from peg A to peg C
Disk 1 from peg B to peg C
Disk 2 from peg B to peg A

```

```
Disk 1 from peg C to peg A
Disk 3 from peg B to peg C
Disk 1 from peg A to peg B
Disk 2 from peg A to peg C
Disk 1 from peg B to peg C
```

The sequence of moves is represented on the right-hand side of Figure 7.1.

## Solution

The key to solving a problem recursively is to recognize that it can be broken down into a collection of smaller sub-problems to each of which that same general solving procedure that we are seeking applies. The total solution is then found in some simple way from those sub-problems' solutions. Each of thus created sub-problems being "smaller" guarantees the base case(s) will eventually be reached. Hence, for the Towers of Hanoi, we can:

- ▷ Label the pegs *A*, *B*, and *C*.
- ▷ Let *n* be the total number of disks.
- ▷ Number the disks from 1 (smallest, topmost) to *n* (largest, bottom-most).

Assuming all *n* disks are distributed in valid arrangements among the pegs; assuming there are *m* top disks on a source peg, and all the rest of the disks are larger than *m* so can be safely ignored; to move *m* discs from a source peg to a target peg using a spare peg, without violating the rules:

- ▷ Move *m* discs from the source to the spare peg, by the same general solving procedure. Rules are not violated, by assumption. This leaves the disk *m* as a top disk on the source peg
- ▷ Move the disk *m* from the source to the target peg, which is guaranteed to be a valid move, by the assumptions, i.e., a simple step.
- ▷ Move the *m* discs that we have just placed on the spare, from the spare to the target peg by the same general solving procedure, so they are placed on top of the disc *m* without violating the rules the base case being, to move 0 disks, do nothing.

The full Tower of Hanoi solution then consists of moving *n* disks from the source peg *A* to the target peg *C*, using *B* as the spare peg.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
5 void hanoi_r (int, int, int);
6
7 /*
8  * main program
9  */
10 int main (void) {
11     int n;
12
13     fprintf(stdout, "Number of disks: ");
14     scanf("%d", &n);
15     hanoi_r(n, 0, 2);
16
17     return EXIT_SUCCESS;

```

```

18 }
19
20 /* 
21  * generate the hanoi moves, recursive function
22 */
23 void hanoi_r (int n, int src, int dst) {
24     int aux;
25
26     if (n > 0) {
27         aux = 3 - (src + dst);
28         hanoi_r(n-1, src, aux);
29         fprintf(stdout, "Disk %d from peg %c to peg %c\n", n, 'A'+src, 'A'+dst);
30         hanoi_r(n-1, aux, dst);
31     }
32
33     return;
34 }
```

## 7.2 Determinant

### Definition

In linear algebra, the *determinant* is a useful value that can be computed from the elements of a square matrix. The determinant of a matrix  $M$  is denoted  $\Delta$  (or  $|M|$ ). It can be viewed as the scaling factor of the transformation described by the matrix. For a matrix of size  $\text{dim}$ , the determinant can be computing adopting a row expression, on row  $i^1$ :

$$\Delta = \sum_{j=0}^{\text{dim}-1} (-1)^{i+j} \cdot M_{ij} \cdot \Delta_{ij} \quad \text{with } i \in [0, \text{dim} - 1]$$

or adopting a column expression, on column  $J$ :

$$\Delta = \sum_{i=0}^{\text{dim}-1} (-1)^{i+j} \cdot M_{ij} \cdot \Delta_{ij} \quad \text{with } j \in [0, \text{dim} - 1]$$

where  $M_{ij}$  is the element of  $M$  in position  $[i][j]$ , and  $\Delta_{ij}$  is the minor of  $M$  of position  $[i][j]$ , i.e., the determinant of the matrix obtained by ruling-out row  $i$  and column  $j$  from  $M$ .

### Specifications

A file stores a square matrix of real value of size  $N$ . Write a program that, once uploaded the matrix from the file, computes it determinant, and prints-out the result of standard output. The value of  $N$  is limited to 10, but is has to be derived while reading the matrix from file.

### Solution 1

```

1 #include <stdio.h>
2 #include <stdlib.h>
```

---

<sup>1</sup> Please remind that in C all array indices start from 0.

```

3 #define SIZE 10
4 #define MAX 512
5
6 /* function prototypes */
7 int load (float matrix[SIZE][SIZE]);
8 float delta_r (float matrix[SIZE][SIZE], int dim);
9 void reduce (float matrix[SIZE][SIZE], int dim, int col, float rm[SIZE][SIZE]);
10
11 /*
12  * main program
13  */
14
15 int main (void) {
16     float matrix[SIZE][SIZE];
17     int dim;
18
19     dim = load(matrix);
20     fprintf(stdout, "Determinant = %f.\n", delta_r(matrix, dim));
21     return EXIT_SUCCESS;
22 }
23
24 /*
25  * read in the matrix file
26  */
27 int load (float matrix[SIZE][SIZE]) {
28     char line[MAX], name[20];
29     int i, j, n;
30     FILE *fp;
31
32     fprintf(stdout, "Input file name: ");
33     scanf("%s", name);
34     fp = fopen(name, "r");
35
36     /* count the number of rows */
37     while (fgets(line, SIZE, fp) != NULL) {
38         n++;
39     }
40     fclose(fp);
41
42     /* read and store the matrix elements */
43     fp = fopen(name, "r");
44     for (i=0; i<n; i++) {
45         for (j=0; j<n; j++) {
46             fscanf(fp, "%f", &matrix[i][j]);
47         }
48     }
49     fclose(fp);
50
51     return n;
52 }
53
54 /*
55  * compute the determinant of a square matrix, recursive function
56  */
57 float delta_r (float matrix[SIZE][SIZE], int dim) {
58     float d=0, smaller[SIZE][SIZE];
59     int j, sign=1;
60
61     if (dim == 1) {

```

```

62     return matrix[0][0];
63 }
64
65 for (j=0; j<dim; j++) {
66     reduce(matrix, dim, j, smaller);
67     d += sign * matrix[0][j] * delta_r(smaller, dim-1);
68     sign = -sign;
69 }
70
71 return d;
72 }
73
74 /**
75 * reduce a matrix by removing the first row and a specified column
76 */
77 void reduce (
78     float matrix[SIZE][SIZE], int dim, int col, float smaller[SIZE][SIZE]
79 ) {
80     int i, j, k=0;
81
82     for (i=1; i<dim; i++) {
83         for (j=0; j<dim; j++) {
84             if (j != col) {
85                 smaller[k/(dim-1)][k%(dim-1)] = matrix[i][j];
86                 k++;
87             }
88         }
89     }
90
91     return;
92 }
```

## Solution 2

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define SIZE 10
5 #define MAX 512
6
7 /* function prototypes */
8 int load(float matrix[SIZE][SIZE]);
9 float delta_r(float matrix[SIZE][SIZE], int dim, int deleted[SIZE], int i);
10
11 /*
12 * main program
13 */
14 int main (void) {
15     float matrix[SIZE][SIZE];
16     int dim, deleted[SIZE]={0};
17
18     dim = load(matrix);
19     fprintf(stdout, "Determinant = %.3f.\n", delta_r(matrix, dim, deleted, 0));
20     return EXIT_SUCCESS;
21 }
22
23 /*
24 * read in the matrix file
25 */
26 int load (float matrix[SIZE][SIZE]) {
```

```

27     char line[MAX], name[20];
28     int i, j, n;
29     FILE *fp;
30
31     fprintf(stdout, "Input file name: ");
32     scanf("%s", name);
33     fp = fopen(name, "r");
34
35     /* count the number of rows */
36     while (fgets(line, SIZE, fp) != NULL) {
37         n++;
38     }
39     fclose(fp);
40
41     /* read and store the matrix elements */
42     fp = fopen(name, "r");
43     for (i=0; i<n; i++) {
44         for (j=0; j<n; j++) {
45             fscanf(fp, "%f", &matrix[i][j]);
46         }
47     }
48     fclose(fp);
49
50     return n;
51 }
52
53 /*
54 * compute the determinant of a square matrix, recursive function
55 */
56 float delta_r(
57     float matrix[SIZE][SIZE], int dim, int deleted[SIZE], int i
58 ) {
59     int j, sign=1;
60     float d=0;
61
62     if (i >= dim) {
63         return 1;
64     }
65
66     for (j=0; j<dim; j++) {
67         if (deleted[j] == 0) {
68             deleted[j] = 1;
69             d += sign*matrix[i][j]*delta_r(matrix, dim, deleted, i+1);
70             sign = -sign;
71             deleted[j] = 0;
72         }
73     }
74
75     return d;
76 }

```

## 7.3 The Magic Square

### Definition

In recreational mathematics, a *magic square* is a  $(n \cdot n)$  square grid (matrix) filled with distinct positive integers in the range  $1, 2, \dots, n$  such that each cell contains a different integer and the sum of the integers in each row, column and diagonal is equal.

The sum is called the *magic constant* or *magic sum* of the magic square. As the

square includes all numbers from 1 to  $n \cdot n = n^2$  each row, column, and diagonal has to have a sum equal to:

$$(1 + 2 + 3 + \dots + n^2) \cdot \frac{1}{n} = \frac{n^2 \cdot (n^2 + 1)}{2} \cdot \frac{1}{n} = \frac{n \cdot (n^2 + 1)}{2}$$

For example, for a  $(3 \cdot 3)$  square, the magic sum is  $3 \cdot (9 + 1)/2 = 15$ .

## Specifications

Write a program that, given a positive integer value  $n$ , is able to print-out the magic square of size  $(n \cdot n)$ .

**Example 7.3** Let  $n$  be equal to 3. A possible magic square is the following one:

```
2 7 6
9 5 1
4 3 8
```

## Solution 1

Let us suppose that the matrix has a size equal to  $n$  rows and  $n$  columns. First of all, the program generates a permutation of the numbers included in the range  $[1, n \cdot n]$ . After that, it checks whether this permutation represents a solution to the given problem. As the check is run only when each permutation is generated, and as for a square of size  $n \cdot n$  up to  $(n \cdot n)!$  permutations have to be generated, the program is unable to find a solution in reasonable times already for very small squares. For example, for a square of size equal to 4 the program has to generate up to  $16!$  permutations, i.e., about  $2 \cdot 10^{13}$  permutations. If we suppose that one permutation can be generated in  $1 \cdot 10^{-9}$  sec the program may run for about 5.5 hours, whereas if each permutation is generated in  $1 \cdot 10^{-6}$  sec the program may run up to about 230 days.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* function prototypes */
5 int square_r (int **, int, int *, int);
6 int check (int **, int);
7
8 /*
9  * main program
10 */
11 int main (int argc, char *argv[]) {
12     int **matrix, *used, i, j, n;
13
14     n = atoi (argv[1]);
15
16     used = (int *)malloc((n*n+1) * sizeof(int));
17     matrix = (int **)malloc(n * sizeof(int *));
18     if (used==NULL || matrix==NULL) {
19         fprintf (stderr, "Memory allocation error.\n");
20         exit(EXIT_FAILURE);
21     }
22     for (i=0; i<n; i++) {
```

```
23     matrix[i] = (int *)malloc(n*sizeof(int));
24     if (matrix[i]==NULL) {
25         fprintf (stderr, "Memory allocation error.\n");
26         exit(EXIT_FAILURE);
27     }
28     for (i=0; i<=n*n; i++) {
29         used[i] = 0;
30     }
31
32     if (square_r(matrix, n, used, 0)) {
33         fprintf(stdout, "Magic square of size %d:\n", n);
34         for (i=0; i<n; i++) {
35             for (j=0; j<n; j++) {
36                 fprintf(stdout, "%2d ", matrix[i][j]);
37             }
38             fprintf(stdout, "\n");
39         }
40     } else {
41         fprintf(stdout, "Magic Square NOT found!\n");
42     }
43
44     free(used);
45     for (i=0; i<n; i++) {
46         free(matrix[i]);
47     }
48     free(matrix);
49
50     return EXIT_SUCCESS;
51 }
52
53 */
54 * compute a magic square of given size; return 1 upon success
55 */
56 int square_r (int **matrix, int dim, int *used, int k) {
57     int i, j, val;
58
59     if (k == dim*dim) {
60         return check(matrix, dim);
61     }
62
63     i = k/dim;
64     j = k%dim;
65     for (val=1; val<=dim*dim; val++) {
66         if (used[val] == 0) {
67             /* put a new value and recur */
68             matrix[i][j] = val;
69             used[val] = 1;
70             if (square_r(matrix, dim, used, k+1)){
71                 /* solution found: stop the process */
72                 return 1;
73             }
74             /* backtrack */
75             used[val] = 0;
76         }
77     }
78
79     /* solution not found */
80     return 0;
81 }
```

```

82 }
83
84 /* 
85 *   check whether a given matrix is a magic square; return 1 if yes
86 */
87 int check (int **matrix, int dim) {
88     int i, j, sum, target;
89
90     target = dim*(dim*dim+1)/2;
91
92     /* rows */
93     for (i=0; i<dim; i++) {
94         sum = 0;
95         for (j=0; j<dim; j++) {
96             sum += matrix[i][j];
97         }
98         if (sum != target) {
99             return 0;
100        }
101    }
102
103    /* columns */
104    for (j=0; j<dim; j++) {
105        sum = 0;
106        for (i=0; i<dim; i++) {
107            sum += matrix[i][j];
108        }
109        if (sum != target) {
110            return 0;
111        }
112    }
113
114    /* diagonals */
115    sum = 0;
116    for (i=0; i<dim; i++) {
117        sum += matrix[i][i];
118    }
119    if (sum != target) {
120        return 0;
121    }
122
123    sum = 0;
124    for (i=0; i<dim; i++) {
125        sum += matrix[i][dim-1-i];
126    }
127    if (sum != target) {
128        return 0;
129    }
130
131    return 1;
132 }

```

## Solution 2

This second solution check if this condition is verified once each row is generated. Run-times improve a lot and a  $4 \times 4$  magic square can be generated in a fraction of a second.

1 #include <stdio.h>

```

2 #include <stdlib.h>
3 /* function prototypes */
4 int square_r (int **, int, int, int *, int);
5 int check (int **, int, int);
6 int check_runTime (int **, int, int, int, int);
7
8 /*
9 * main program
10 */
11 int main(int argc, char *argv[]) {
12     int **matrix, *used, i, j, n, target;
13
14     n = atoi (argv[1]);
15     target = n*(n*n+1)/2;
16
17     used = (int *)malloc((n*n+1) * sizeof(int));
18     matrix = (int **)malloc(n * sizeof(int *));
19     if (used==NULL || matrix==NULL) {
20         fprintf(stdout, "Memory allocation error.\n");
21         exit(EXIT_FAILURE);
22     }
23     for (i=0; i<n; i++) {
24         matrix[i] = (int *)malloc(n*sizeof(int));
25         if (matrix[i]==NULL) {
26             fprintf(stdout, "Memory allocation error.\n");
27             exit(EXIT_FAILURE);
28         }
29     }
30     for (i=0; i<=n*n; i++) {
31         used[i] = 0;
32     }
33
34     if (square_r(matrix, target, n, used, 0)) {
35         fprintf(stdout, "Magic square of size %d:\n", n);
36         for (i=0; i<n; i++) {
37             for (j=0; j<n; j++) {
38                 fprintf(stdout, "%2d ", matrix[i][j]);
39             }
40             fprintf(stdout, "\n");
41         }
42     } else {
43         fprintf(stdout, "Magic Square NOT found!\n");
44     }
45
46     free(used);
47     for (i=0; i<n; i++) {
48         free(matrix[i]);
49     }
50     free(matrix);
51
52     return EXIT_SUCCESS;
53 }
54
55 /*
56 * compute a magic square of given size; return 1 upon success
57 */
58 int square_r (int **matrix, int target, int dim, int *used, int k) {
59     int i, j, val;
60

```

```
61
62     if (k == dim*dim) {
63         return check(matrix, target, dim);
64     }
65
66     i = k/dim;
67     j = k%dim;
68     for (val=1; val<=dim*dim; val++) {
69         if (used[val] == 0) {
70             /* put a new value and recur */
71             matrix[i][j] = val;
72             if (check_runTime (matrix, target, i, j, dim) == 0) {
73                 continue;
74             }
75             used[val] = 1;
76             if (square_r(matrix, target, dim, used, k+1)) {
77                 /* solution found: stop the process */
78                 return 1;
79             }
80             /* backtrack */
81             used[val] = 0;
82         }
83     }
84
85     /* solution not found */
86     return 0;
87 }
88
89 /*
90 *   check whether a given matrix is a magic square; return 1 if yes
91 */
92 int check (int **matrix, int target, int dim) {
93     int i, j, sum;
94
95     /* rows */
96     for (i=0; i<dim; i++) {
97         sum = 0;
98         for (j=0; j<dim; j++) {
99             sum += matrix[i][j];
100        }
101        if (sum != target) {
102            return 0;
103        }
104    }
105
106    /* columns */
107    for (j=0; j<dim; j++) {
108        sum = 0;
109        for (i=0; i<dim; i++) {
110            sum += matrix[i][j];
111        }
112        if (sum != target) {
113            return 0;
114        }
115    }
116
117    /* diagonals */
118    sum = 0;
119    for (i=0; i<dim; i++) {
```

```
120     sum += matrix[i][i];
121 } if (sum != target) {
122     return 0;
123 }
124 }
125 sum = 0;
126 for (i=0; i<dim; i++) {
127     sum += matrix[i][dim-1-i];
128 }
129 if (sum != target) {
130     return 0;
131 }
132 }
133 return 1;
134 }
135 }
136 /*
137 * check whether a given matrix is a magic square; return 1 if yes
138 */
139 int check_runTime (int **matrix, int target, int r, int c, int dim) {
140     int i, j, sum;
141
142     sum = 0;
143     for (j=0; j<=c; j++) {
144         sum += matrix[r][j];
145     }
146     if (c < (dim-1)) {
147         if (sum > target) {
148             return 0;
149         }
150     } else {
151         if (sum != target) {
152             return 0;
153         }
154     }
155 }
156
157 sum = 0;
158 for (i=0; i<=r; i++) {
159     sum += matrix[i][c];
160 }
161 if (r < (dim-1)) {
162     if (sum > target) {
163         return 0;
164     }
165 } else {
166     if (sum != target) {
167         return 0;
168     }
169 }
170 }
171 return 1;
172 }
```

## 7.4 The maze

## Specifications

A file includes a labyrinth with the following specifications:

- ▷ The first line of the file specifies the number of row R and the number of columns C of a matrix
  - ▷ The following R lines specify the matrix rows (each one with C characters), where each
    - ◊ ' @' indicates the initial position of a human being.
    - ◊ ' ' represents corridors (empty cells).
    - ◊ ' \*' represents walls (full cells).
    - ◊ ' #' represents exit points.

Supposed only one person is present in the maze in the initial configuration.

The following is a correct example of such a maze:

12 10

\* \* \* \* \*

★ ★

\* \* @ \*

\* \* \* \*

\* \* \* \* \*

\* \* \*

\* \* \*

\* \* \* \*

\* \* \* \*

\* \* \* \*

★ ★ ★

\*\*\*\*\*#\*\*\*\*\*

Write three re-

- ▷ One escap

▷ All escape

6 - 1

- ▷ One escape path
  - ▷ All escape paths
  - ▷ The shortest escape path

from the maze. Print out the solution (or all solutions) on standard output.

### Solution

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define MAX 100
6
7 #define EMPTY ' '
8 #define START '@'
9 #define STOP '#'
10 #define PATH '$'
11 #define DONE '/'
```

```

12 const int xOff[4] = { 0, 1, 0, -1};
13 const int yOff[4] = {-1, 0, 1, 0};
14
15 void display(char **, int);
16 int move_r_one(char **, int, int, int, int, int);
17 int move_r_all(char **, int, int, int, int, int, int);
18 int move_r_best(char **, int, char **, int, int, int, int, int, int);
19 FILE *util_fopen(char *, char *);
20 void *util_malloc(int);
21 char *util_strdup(char *);
22
23 int main (int argc, char *argv[]) {
24     int r=-1, c=-1, i, j, nr, nc, res;
25     char **mazeCurr, **mazeBest;
26     char line[MAX];
27     FILE *fp;
28
29     if (argc < 2) {
30         fprintf(stderr, "Error: missing parameter.\n");
31         fprintf(stderr, "Run as: %s <maze_file>\n", argv[0]);
32         exit(EXIT_FAILURE);
33     }
34
35     fp = util_fopen(argv[1], "r");
36     fgets(line, MAX, fp);
37     sscanf(line, "%d %d", &nr, &nc);
38     mazeCurr = (char **)util_malloc(nr * sizeof(char *));
39     mazeBest = (char **)util_malloc(nr * sizeof(char *));
40     for (i=0; i<nr; i++) {
41         fgets(line, MAX, fp);
42         mazeCurr[i] = util_strdup(line);
43         mazeBest[i] = util_strdup(line);
44         for (j=0; j<nc; j++) {
45             if (mazeCurr[i][j] == START) {
46                 r = i;
47                 c = j;
48             }
49         }
50     }
51 }
52
53 if (r<0 || c<0) {
54     fprintf(stderr, "Error: no starting position found!\n");
55     exit(EXIT_FAILURE);
56 }
57
58 // Find one solution
59 fprintf(stderr, "Find one solution:\n");
60 mazeCurr[r][c] = EMPTY;
61 res = move_r_one(mazeCurr, nr, nc, r, c);
62 if (res == 1) {
63     mazeCurr[r][c] = START;
64     display(mazeCurr, nr);
65 } else {
66     fprintf(stderr, "NO solution found!\n");
67 }
68 // Clean matrix
69 for (i=0; i<nr; i++) {
70     for (j=0; j<nc; j++) {

```

```

71     if (mazeCurr[i][j]==PATH || mazeCurr[i][j]==DONE) {
72         mazeCurr[i][j] = EMPTY;
73     }
74 }
75 }

76 // Find all solutions
77 fprintf(stdout, "Find all solutions:\n");
78 mazeCurr[r][c] = EMPTY;
79 res = move_r_all(mazeCurr, nr, nc, r, c, r, c);
80

81 // Find the best solution
82 fprintf(stdout, "Find the best solutions:\n");
83 mazeCurr[r][c] = EMPTY;
84 res = move_r_best(mazeCurr, 0, mazeBest, nr*nc, nr, nc, r, c);
85 if (res > 0) {
86     fprintf(stdout, "Solution:\n");
87     mazeBest[r][c] = START;
88     display(mazeBest, nr);
89 } else {
90     fprintf(stdout, "NO solution found!\n");
91 }
92

93 for (r=0; r<nr; r++) {
94     free(mazeCurr[r]);
95     free(mazeBest[r]);
96 }
97 free(mazeCurr);
98 free(mazeBest);
99 return EXIT_SUCCESS;
100 }

101 }

102 /*

103 * find one (the first) solution
104 */
105
106 int move_r_one (char **mazeCurr, int nr, int nc, int row, int col) {
107     int k, r, c;
108
109     if (mazeCurr[row][col] == STOP) {
110         return 1;
111     }
112     if (mazeCurr[row][col] != EMPTY) {
113         return 0;
114     }
115
116     mazeCurr[row][col] = PATH;
117     for (k=0; k<4; k++) {
118         r = row + xOff[k];
119         c = col + yOff[k];
120         if (r>=0 && r<nr && c>=0 && c<nc) {
121             if (move_r_one(mazeCurr, nr, nc, r, c) == 1)
122                 return 1;
123         }
124     }
125     mazeCurr[row][col] = DONE;
126
127     return 0;
128 }
129

```

```

130 /* find all solutions
131 */
132 int move_r_all (
133     char **mazeCurr, int nr, int nc, int row, int col, int row0, int col0
134 ) {
135     int k, r, c;
136     static int solN = 0;
137
138     if (mazeCurr[row][col] == STOP) {
139         solN++;
140         fprintf(stdout, "Solution # %d:\n", solN);
141         mazeCurr[row0][col0] = START;
142         display(mazeCurr, nr);
143         mazeCurr[row0][col0] = EMPTY;
144         return 1;
145     }
146     if (mazeCurr[row][col] != EMPTY) {
147         return 0;
148     }
149
150     mazeCurr[row][col] = PATH;
151     for (k=0; k<4; k++) {
152         r = row + xOff[k];
153         c = col + yOff[k];
154         if (r>=0 && r<nr && c>=0 && c<nc) {
155             move_r_all(mazeCurr, nr, nc, r, c, row0, col0);
156         }
157     }
158 }
159 mazeCurr[row][col] = EMPTY;
160
161 return 0;
162 }
163
164 /*
165 * find the best solution
166 */
167 int move_r_best (
168     char **mazeCurr, int stepCurr, char **mazeBest,
169     int stepBest, int nr, int nc, int row, int col
170 ) {
171     int k, r, c;
172
173     /* Avoid stopping recursion to find ALL solutions */
174     if (stepCurr >= stepBest) {
175         return stepBest;
176     }
177     if (mazeCurr[row][col] == STOP) {
178         /* STOP recurring to print-out only ONE solution */
179         /* Print all matrices to have ALL solutions */
180         if (stepCurr < stepBest) {
181             stepBest = stepCurr;
182             for (r=0; r<nr; r++) {
183                 for (c=0; c<nc; c++) {
184                     mazeBest[r][c] = mazeCurr[r][c];
185                 }
186             }
187         }
188     }
189     return stepBest;

```

```
189 }
190 if (mazeCurr[row][col] != EMPTY) {
191     return stepBest;
192 }
193 mazeCurr[row][col] = PATH;
194 for (k=0; k<4; k++) {
195     r = row + xOff[k];
196     c = col + yOff[k];
197     if (r>=0 && r<nr && c>=0 && c<nc) {
198         stepBest = move_r_best (
199             mazeCurr, stepCurr+1, mazeBest, stepBest, nr, nc, r, c);
200     }
201 }
202 mazeCurr[row][col] = EMPTY;
203
204 return stepBest;
205 }
206 */
207 *
208 * write the maze scheme on screen
209 */
210 void display (char **maze, int nr) {
211     int i;
212
213     for (i=0; i<nr; i++) {
214         fprintf(stdout, "%s", maze[i]);
215     }
216 }
217
218 /*
219 * fopen (with check) utility function
220 */
221 FILE *util_fopen (char *name, char *mode) {
222     FILE *fp=fopen(name, mode);
223
224     if (fp == NULL) {
225         fprintf(stdout, "File open error (file=%s).\n", name);
226         exit(EXIT_FAILURE);
227     }
228
229     return fp;
230 }
231
232 /*
233 * malloc (with check) utility function
234 */
235 void *util_malloc (int size) {
236     void *ptr=malloc(size);
237
238     if (ptr == NULL) {
239         fprintf(stdout, "Memory allocation error.\n");
240         exit(EXIT_FAILURE);
241     }
242
243     return ptr;
244 }
245
246 /*
247 */
```

```

248     * strdup (with check) utility function
249
250     char *util_strdup (char *src) {
251         char *dst=strdup(src);
252
253         if (dst == NULL) {
254             fprintf(stdout, "Memory allocation error.\n");
255             exit(EXIT_FAILURE);
256         }
257
258         return dst;
259     }

```

## 7.5 The Sudoku Puzzle

### Definition

*Sudoku*, originally called Number Place, is a logic-based, combinatorial number-placement puzzle. The objective is to fill a  $(9 \cdot 9)$  grid with digits so that each column, each row, and each of the nine  $(3 \cdot 3)$  sub-grids that compose the grid (also called “blocks” or “regions”) contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a unique solution.

### Specifications

Write a program able to read an initial Sudoku configuration from a file and to complete it, following the Sudoku rules.

The size of the matrix has to be deduced by the file format.

**Example 7.4** Given the following input file:

```

4 0 0 1
0 1 0 0
3 0 1 0
0 0 0 0

```

the program has to understand that  $N = 2$  (the entire matrix has a size equal to  $4 \times 4$ ), and it has to print-out the following solution:

```

4 3 2 1
2 1 4 3
3 2 1 4
1 4 3 2

```

### Solution

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <math.h>
5
6 #define MAXBUFFER 128
7
8 /* function prototypes */
9 int **load (int *n_ptr);
10 int find_r (int **schema, int n, int step);
11 int check (int **sudoku, int n, int step);

```

```
12
13  /*
14   * main program
15   */
16 int main (void) {
17     int **sudoku, dim, i, j;
18
19     sudoku = load(&dim);
20
21     if (find_r(sudoku, dim, 0)) {
22         fprintf(stdout, "Solution:\n");
23         for (i=0; i<dim; i++) {
24             for (j=0; j<dim; j++) {
25                 fprintf(stdout, "%3d ", sudoku[i][j]);
26             }
27             fprintf(stdout, "\n");
28         }
29     } else {
30         fprintf(stdout, "No solution found.\n");
31     }
32
33     for (i=0; i<dim; i++) {
34         free(sudoku[i]);
35     }
36     free(sudoku);
37
38     return EXIT_SUCCESS;
39 }
40
41 /*
42 * load the initial sudoku scheme from file
43 */
44 int **load (int *dim_ptr) {
45     char buffer[MAXBUFFER], filename[20];
46     int i, j, **sudoku, dim=0;
47     FILE *fp;
48
49     fprintf(stdout, "Input the file name: ");
50     scanf("%s", filename);
51
52     fp = fopen(filename, "r");
53
54     /* count the number of rows to get the sudoku size */
55     while (fgets(buffer, MAXBUFFER, fp) != NULL) {
56         dim++;
57     }
58     fclose(fp);
59
60     /* allocation memoria necessaria */
61     sudoku = (int **)malloc(dim*sizeof(int *));
62     if (sudoku==NULL) {
63         fprintf (stderr, "Memory allocation error.\n");
64         exit(EXIT_FAILURE);
65     }
66     for (i=0; i<dim; i++) {
67         sudoku[i] = (int *)malloc(dim*sizeof(int));
68         if (sudoku[i]==NULL) {
69             fprintf (stderr, "Memory allocation error.\n");
70             exit(EXIT_FAILURE);
```

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
}
/* store the sudoku matrix */
fp = fopen(filename, "r");
for (i=0; i<dim; i++) {
    for (j=0; j<dim; j++) {
        fscanf(fp, "%d", &sudoku[i][j]);
    }
}
fclose(fp);
*dim_ptr = dim;
return sudoku;
}

/*
 * recursively assign values to the empty sudoku cells
 */
int find_r (int **sudoku, int dim, int step) {
    int i, j, k;

    if (step >= dim*dim) {
        /* solution found */
        return 1;
    }

    /* get the current cell position */
    i = step / dim;
    j = step % dim;

    if (sudoku[i][j] != 0) {
        /* originally full cell: no attempt to do */
        return find_r(sudoku, dim, step+1);
    }

    /* try all values from 1 to dim (i.e., n*n) */
    for (k=1; k<=dim; k++) {
        sudoku[i][j] = k;
        if (check(sudoku, dim, step)) {
            if (find_r(sudoku, dim, step+1)) {
                return 1;
            }
        }
        sudoku[i][j] = 0;
    }

    return 0;
}

/*
 * check whether the game rules are violated
 */
int check (int **sudoku, int dim, int step) {
    int i, j, ii, jj, ib, jb, val, n=floor(sqrt(dim));
    /* get the current cell and block position */
    i = step / dim;
    j = step % dim;
    ii = i / n;
    jj = j / n;
    ib = i / n * n;
    jb = j / n * n;
    for (val=1; val<=dim; val++) {
        /* check row */
        if (sudoku[i][j] == val)
            continue;
        for (k=0; k<dim; k++)
            if (sudoku[i][k] == val)
                return 0;
        /* check column */
        if (sudoku[i][j] == val)
            continue;
        for (k=0; k<dim; k++)
            if (sudoku[k][j] == val)
                return 0;
        /* check block */
        if (sudoku[i][j] == val)
            continue;
        for (k=0; k<n; k++)
            for (l=0; l<n; l++)
                if (sudoku[ii+n*k][jj+n*l] == val)
                    return 0;
    }
    return 1;
}

```

```

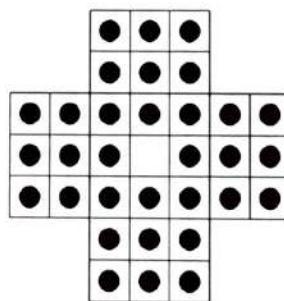
130     ib = (i/n) * n;
131     jb = (j/n) * n;
132     val = sudoku[i][j];
133
134     /* check the row */
135     for (jj=0; jj<dim; jj++) {
136         if (sudoku[i][jj]==val && jj!=j) {
137             return 0;
138         }
139     }
140
141     /* check the column */
142     for (ii=0; ii<dim; ii++) {
143         if (sudoku[ii][j]==val && ii!=i) {
144             return 0;
145         }
146     }
147
148     /* check the block */
149     for (ii=ib; ii<ib+n; ii++) {
150         for (jj=jb; jj<jb+n; jj++) {
151             if (sudoku[ii][jj]==val && (ii!=i || jj!=j)) {
152                 return 0;
153             }
154         }
155     }
156
157     return 1;
158 }
```

## 7.6 Cross Checkers

### Specifications

Find a solution to the following puzzle.

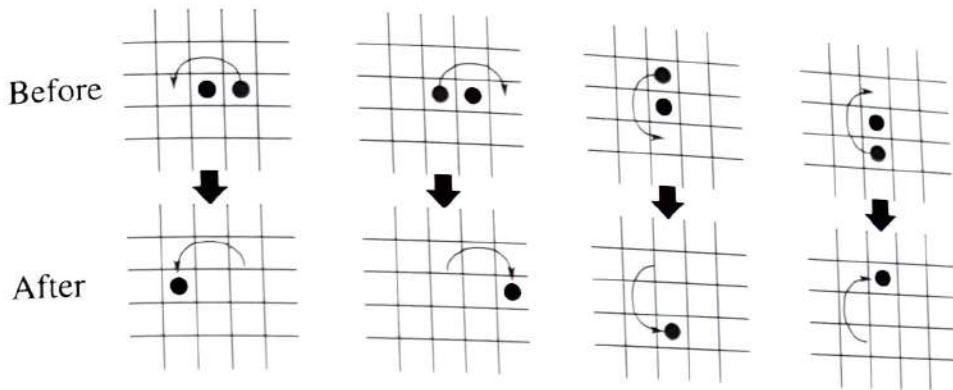
A board has 33 cells, 32 pawn, and the following shape and initial configuration:



Find a sequence of moves such that at the end of the game just one pawn remains on the board. A move is such that a pawn eat an adjacent pawn horizontally or vertically by moving on its other side. The following picture shows all possible moves:

### Solution

The logic is somehow simple: As there are 32 pawns, there are at most 31 moves, after which only one pawn must remain on the board. Then we have to try all possible moves at most 31 times, i.e., on each recursion level.



```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define E    0 /* empty cell */
5 #define F    1 /* full cell */
6 #define O   255 /* out of the board */
7
8 #define SIZE 8 /* board size */
9
10 /* offset for the moves */
11 const int xOff[4] = { 0, 1, 0, -1};
12 const int yOff[4] = {-1, 0, 1, 0};
13
14 /* function prototypes */
15 int move_r (int board[SIZE+1][SIZE+1], int moves[31][4], int n);
16
17 /*
18 * main program
19 */
20 int main (void) {
21     int i, moves[31][4], board[SIZE+1][SIZE+1] =
22     {
23         {O, O, O, O, O, O, O, O, O},
24         {O, O, O, F, F, F, O, O, O},
25         {O, O, O, F, F, F, O, O, O},
26         {O, F, F, F, F, F, F, F, O},
27         {O, F, F, F, E, F, F, F, O},
28         {O, F, F, F, F, F, F, F, O},
29         {O, O, O, F, F, F, O, O, O},
30         {O, O, O, F, F, F, O, O, O},
31         {O, O, O, O, O, O, O, O, O}
32     };
33
34     if (move_r(board, moves, 0) == 0) {
35         fprintf(stdout, "No solution exists!\n");
36     } else {
37         for (i=0; i<31; i++) {
38             fprintf(stdout, "%d %d -> %d %d\n", moves[i][0], moves[i][1], moves[i][2],
39                     moves[i][3]);
40         }
41     }
42
43     return EXIT_SUCCESS;
44 }
```

```

46  /*
47   * make a move (recursive function)
48   */
49 int move_r (int board[SIZE+1][SIZE+1], int moves[31][4], int n) {
50     int x1, y1, x2, y2, x3, y3, k;
51
52     if (n == 31) {
53         return 1;
54     }
55
56     /* for each board cell ... */
57     for (x1=1; x1<SIZE; x1++) {
58         for (y1=1; y1<SIZE; y1++) {
59             /* ... if there is a pawn ... */
60             if (board[x1][y1] == F) {
61                 /* ... look whether it is possible to eat ... */
62                 for (k=0; k<4; k++) {
63                     x2 = x1 + xOff[k];
64                     y2 = y1 + yOff[k];
65                     if (board[x2][y2] == F) {
66                         x3 = x1 + 2*xOff[k];
67                         y3 = y1 + 2*yOff[k];
68                         if (board[x3][y3] == E) {
69                             /* ... then make the move ... */
70                             board[x3][y3] = F;
71                             board[x2][y2] = E;
72                             board[x1][y1] = E;
73                             /* ... and recur ... */
74                             if (move_r(board, moves, n+1) == 1) {
75                                 /* ... storing the move if it was ok ... */
76                                 moves[n][0] = x1;
77                                 moves[n][1] = y1;
78                                 moves[n][2] = x3;
79                                 moves[n][3] = y3;
80                                 return 1;
81                             }
82                             /* ... otherwise backtrack */
83                             board[x3][y3] = E;
84                             board[x2][y2] = F;
85                             board[x1][y1] = F;
86                         }
87                     }
88                 }
89             }
90         }
91     }
92
93     return 0;
94 }
```

## 7.7 Bookshelves

### Specifications

A group of Politecnico students lives in the same apartment. Those students would like to buy a new bookshelf to put all books they have in order. A bookshelf catalog store a set of bookshelf characteristics. The catalog reports for each bookshelf one line with the following pieces of information:

```
id price #shelves width weight height1 ... heightN
```

where:

- ▷ id is a alphanumeric identifier of 10 characters, uniquely identifying the bookshelf.
- ▷ price is the price of the bookshelf (real value).
- ▷ #shelves is the number of shelves present in the bookshelf (integer value).
- ▷ width indicates the width of all shelves (integer value).
- ▷ weight is the maximum weight any shelf can hold (integer value).
- ▷ height<sub>1</sub>, ..., height<sub>N</sub> are the height of all shelves in the bookshelf (#shelves integer values, one for each shelf).

The first line of the file stores the number of different bookshelves present in the catalog.

A second file stores the description of all books that have to be stored in the bookshelf. Each line of the file has the following format:

```
name height thickness weight
```

where:

- ▷ name is a book identifier (a 30 character string).
- ▷ height and thickness indicate the size of the book (real value in centimeters).
- ▷ weight indicates the weight (real values in kilograms).

The first line of the file stores the number of books to store in the bookshelf.

Write a program that, once read the two files, is able to decide which is the cheapest bookshelf it is necessary to buy to store all books. The program has to print-out on which shelf each book can be placed.

Pay attention to the following points:

- ▷ The depth of all books is the same, and each bookshelf is able to contain any of them.
- ▷ All books have to be placed vertically, not horizontally, they have to be placed back-to-back, and they cannot overlap.
- ▷ All placements are considered as equivalent once they respect the constraints specified by the text (such as maximum weight and maximum height for each shelf, etc.).

## Solution

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <float.h>
5
6 #define CODE 10+1
7 #define NAME 30+1
8
9 /* structure declaration */
10 typedef struct {
11     char code[CODE];
12     float price;
13     int shelves;
14     int *heights;
```

```
15     float *widths;
16     float *weights;
17 } bookcase_t;
18
19 typedef struct {
20     char name[NAME];
21     float height;
22     float width;
23     float weight;
24     int shelf;
25 } book_t;
26
27 /* function prototypes */
28 bookcase_t *read_models(int *num_ptr);
29 void sort_models(bookcase_t *models, int left, int right);
30 book_t *read_material(int *num_ptr);
31 int suitable_r(bookcase_t *bookcase, book_t *material, int num_books, int idx);
32
33 /*
34  * main program
35 */
36 int main (void) {
37     int i, j, num_models, num_books, found;
38     bookcase_t *models;
39     book_t *material;
40
41     /* load info */
42     models = read_models(&num_models);
43     material = read_material(&num_books);
44
45     /* solve the problem */
46     for (i=0; i<num_models && !found; i++) {
47         found = suitable_r(&models[i], material, num_books, 0);
48         if (found != 0) {
49             fprintf(stdout, "Bookcase model to buy: %s.\n", models[i].code);
50             fprintf(stdout, "Books disposition:\n");
51             for (j=0; j<num_books; j++) {
52                 fprintf(stdout, "%s --> shelf %d\n",
53                         material[j].name, material[j].shelf+1);
54             }
55         }
56     }
57
58     /* free memory */
59     for (i=0; i<num_models; i++) {
60         free(models[i].heights);
61         free(models[i].widths);
62         free(models[i].weights);
63     }
64     free(models);
65     free(material);
66
67     return EXIT_SUCCESS;
68 }
69
70 /*
71  * read in the bookcases info
72 */
73 bookcase_t *read_models (int *num_ptr) {
```

```

14     int n, i, j, shelves, width, height, weight;
15     bookcase_t *models;
16     char name[100];
17     float price;
18     FILE *fp;
19
20     fprintf(stdout, "Input the bookcases file name: ");
21     scanf("%s", name);
22     fp = fopen(name, "r");
23     fscanf(fp, "%d", &n);
24     models = (bookcase_t *)malloc(n*sizeof(bookcase_t));
25     if (models==NULL) {
26         fprintf (stderr, "Memory allocation error.\n");
27         exit(EXIT_FAILURE);
28     }
29
30     for (i=0; i<n; i++) {
31         fscanf(fp, "%s %f %d %d %d", name, &price, &shelves, &width, &weight);
32         strcpy(models[i].code, name);
33         models[i].price = price;
34         models[i].shelves = shelves;
35         models[i].heights = (int *)malloc(shelves*sizeof(int));
36         models[i].widths = (float *)malloc(shelves*sizeof(float));
37         models[i].weights = (float *)malloc(shelves*sizeof(float));
38         if (models[i].heights==NULL || models[i].widths==NULL ||
39             models[i].weights==NULL) {
40             fprintf (stderr, "Memory allocation error.\n");
41             exit(EXIT_FAILURE);
42         }
43         for (j=0; j<shelves; j++) {
44             fscanf(fp, "%d", &height);
45             models[i].widths[j] = width;
46             models[i].heights[j] = height;
47             models[i].weights[j] = weight;
48         }
49     }
50     fclose(fp);
51
52     sort_models(models, 0, n-1);
53     *num_ptr = n;
54     return models;
55 }
56
57 /*
58  *  sort bookcases by increasing price
59  */
60 void sort_models (bookcase_t *models, int left, int right) {
61     bookcase_t tmp;
62     float price;
63     int i, j;
64
65     price = models[left].price;
66     i = left-1;
67     j = right+1;
68     while (i < j) {
69         while (models[++i].price < price) ;
70         while (models[--j].price > price) ;
71         if (i < j) {
72             tmp = models[i];
73             models[i] = models[j];
74             models[j] = tmp;
75         }
76     }
77 }
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132

```

```

133         models[i] = models[j];
134         models[j] = tmp;
135     }
136 }
137 if (left < right) {
138     sort_models(models, left, j);
139     sort_models(models, j+1, right);
140 }
141 }
142
143 /*
144 *   read in the books info
145 */
146 book_t *read_material (int *num_ptr) {
147     float height, width, weight;
148     book_t *material;
149     char name[100];
150     int n, i;
151     FILE *fp;
152
153     fprintf(stdout, "Input the books file name: ");
154     scanf("%s", name);
155     fp = fopen(name, "r");
156     fscanf(fp, "%d", &n);
157     material = (book_t *)malloc(n*sizeof(book_t));
158     if (material==NULL) {
159         fprintf (stderr, "Memory allocation error.\n");
160         exit(EXIT_FAILURE);
161     }
162
163     for (i=0; i<n; i++) {
164         fscanf(fp, "%s %f %f %f", name, &height, &width, &weight);
165         strcpy(material[i].name, name);
166         material[i].height = height;
167         material[i].width = width;
168         material[i].weight = weight;
169         material[i].shelf = -1;
170     }
171
172     fclose(fp);
173     *num_ptr = n;
174     return material;
175 }
176
177 /*
178 *   recursively check whether a given bookcase can contain all the material
179 */
180 int suitable_r (bookcase_t *bookcase, book_t *material, int num_books, int idx) {
181     int i;
182
183     if (idx == num_books) {
184         return 1;
185     }
186
187     for (i=0; i<bookcase->shelves; i++) {
188         if (bookcase->heights[i]>=material[idx].height &&
189             bookcase->widths[i]>=material[idx].width &&
190             bookcase->weights[i]>=material[idx].weight) {
191

```

```

192 bookcase->widths[i] == material[idx].width;
193 bookcase->weights[i] == material[idx].weight;
194 material[idx].shelf = i;
195 if (suitable_r(bookcase, material, num_books, idx+1)) {
196     return 1;
197 }
198 bookcase->widths[i] += material[idx].width;
199 bookcase->weights[i] += material[idx].weight;
200 material[idx].shelf = -1;
201 }
202 }
203 return 0;
204 }
205 }
```

## 7.8 The knapsack problem

### Problem formulation

The *knapsack problem* or *rucksack problem* is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

### Specifications

Solve the knapsack problem with the following specifications. A file stores the weights of a set of objects. The first line stores the number of objects, and the each one of the following lines reports the object identifier (a string of 20 characters), its weights, and its value. The program receives the maximum weight that the knapsack can hold on the command line.

### Solution

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 /* structure declaration */
6 typedef struct {
7     char *name;
8     int val;
9     int size;
10    int selectedCurr;
11    int selectedBest;
12 } object_t;
13
14 /* function prototypes */
15 object_t *file_read(int *n_ptr);
16 int knap_r(object_t *obj, int n, int cap, int id, int valCurr, int valBest);
17
18 /*
19 * main program
20 */
```

```
21 int main (int argc, char *argv[]) {
22     object_t *objs;
23     int i, n, max;
24
25     objs = file_read(&n);
26     max = knap_r(objs, n, atoi(argv[1]), 0, 0, 0);
27     fprintf(stdout, "The largest possible value is: %d\n", max);
28     fprintf(stdout, "Objects to take:\n");
29     for (i=0; i<n; i++) {
30         if (objs[i].selectedBest) {
31             fprintf(stdout, "- %s\n", objs[i].name);
32         }
33     }
34
35     for (i=0; i<n; i++) {
36         free(objs[i].name);
37     }
38     free(objs);
39     return EXIT_SUCCESS;
40 }
41
42 /*
43 *   read in the object description file
44 */
45 object_t *file_read (int *n_ptr) {
46     object_t *objs;
47     char name[20+1];
48     int i, n;
49     FILE *fp;
50
51     fprintf(stdout, "Input file name: ");
52     scanf("%s", name);
53     fp = fopen(name, "r");
54     fscanf(fp, "%d", &n);
55     objs = (object_t *)malloc(n*sizeof(object_t));
56     if (objs==NULL) {
57         fprintf (stderr, "Memory allocation error.\n");
58         exit(EXIT_FAILURE);
59     }
60     for (i=0; i<n; i++) {
61         fscanf(fp, "%s %d %d", name, &objs[i].size, &objs[i].val);
62         objs[i].name = strdup(name);
63         objs[i].selectedCurr = objs[i].selectedBest = 0;
64     }
65     fclose(fp);
66
67     *n_ptr = n;
68     return objs;
69 }
70
71 /*
72 *   recursively select a set of objs
73 */
74 int knap_r (object_t *objs, int n, int cap, int id, int valCurr, int valBest) {
75     int i;
76
77     if (id >= n) {
78         return valBest;
79     }
```

```

80  /* if possible, select the current object... */
81  if (cap >= objs[id].size) {
82      objs[id].selectedCurr = 1;
83      cap -= objs[id].size;
84      valCurr += objs[id].val;
85      if (valCurr > valBest) {
86          valBest = valCurr;
87          for (i=0; i<n; i++) {
88              objs[i].selectedBest = objs[i].selectedCurr;
89          }
90      }
91      valBest = knap_r(objs, n, cap, id+1, valCurr, valBest);
92      objs[id].selectedCurr = 0;
93      cap += objs[id].size;
94      valCurr -= objs[id].val;
95  }
96 }
97
98 /* ... then, try WITHOUT selecting it */
99 valBest = knap_r(objs, n, cap, id+1, valCurr, valBest);
100
101 return valBest;
102 }
```

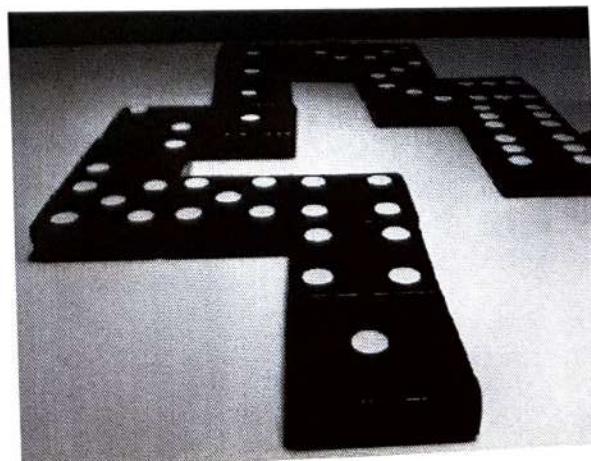
## 7.9 Dominoes

### Puzzle Description

*Dominoes* or *dominos* is a game played with rectangular “domino” tiles. The domino gaming pieces make up a domino set, sometimes called a deck or pack. Each domino is a rectangular tile with a line dividing its face into two square ends. Each end is marked with a number of spots or is blank. The backs of the dominoes in a set are indistinguishable, either blank or having some common design. Figure 7.3 reports some pictures of the game.



(a)



(b)

Figure 7.3 Dominoes pictures.

## Specifications

Given a domino set write a program able to find the longest sequence (or chain) of dominos that can be built such that each back-to-back domino pieces share the same spot configuration.

The program is run with 3 parameters on the command line: an input file name, a domino piece number, and an output file name. The input file stores the domino set, in the format:

```
dominoNumber n1-n2
```

where `dominoNumber` is an integer number uniquely identifying the domino piece, and `n1` and `n2` are the number of spots on the two extreme of that piece. Notice that identical piece are possible. The domino piece number is the domino piece identifier from which the program has to start building the sequence of pieces. Notice that each domino (e.g., 2-3) can be placed in both directions (e.g., as 2-3 or 3-2). The output file name specifies the output file in which the program has to store the longest sequence of dominos.

**Example 7.5** Let us suppose the program is run as:

```
domino a.txt 4 b.txt
```

and let `a.txt` be the following file:

```
2 1-2
5 9-9
3 5-2
4 2-3
1 6-5
```

The longest chain of domino pieces is the following: 3-2, 2-5, 5-6. The program has to store in the output file, the following information:

```
3
4 3-2
3 2-5
1 5-2
```

## Solution

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* structure declaration */
5 typedef struct {
6     int id, n1, n2, taken;
7 } piece_t;
8
9 /* function prototypes */
10 int chain_r(int *currChain, int currLength, int *bestChain, int bestLength,
11             piece_t *pieces, int num);
12
13 /*
14     * main program
15 */
16 int main (int argc, char **argv) {
```

```

17 int *currChain, *bestChain, currLength, bestLength=0;
18 int start, num=0, i, j, k;
19 piece_t *pieces;
20 char line[20];
21 FILE *fp;
22
23 /* count the number of pieces in the input file */
24 fp = fopen(argv[1], "r");
25 while (fgets(line, 20, fp) != NULL) {
26     num++;
27 }
28 fclose(fp);
29
30 /* allocate arrays */
31 pieces = (piece_t *)malloc((num+1)*sizeof(piece_t));
32 currChain = (int *)malloc(num*sizeof(int));
33 bestChain = (int *)malloc(num*sizeof(int));
34 if (pieces==NULL || currChain==NULL || bestChain==NULL) {
35     fprintf (stderr, "Memory allocation error.\n");
36     exit(EXIT_FAILURE);
37 }
38
39 /* store the pieces */
40 fp = fopen(argv[1], "r");
41 for (i=1; i<=num; i++) {
42     fscanf(fp, "%d %d-%d", &pieces[i].id, &pieces[i].n1, &pieces[i].n2);
43     pieces[i].taken = 0;
44 }
45 fclose(fp);
46
47 /* look for the first piece */
48 sscanf(argv[2], "%d", &start);
49 for (i=1; i<=num && pieces[i].id!=start; i++) ;
50 pieces[i].taken = 1;
51 currLength = 1;
52
53 /* search the solution: try the first piece "straight" */
54 currChain[0] = i;
55 bestLength = chain_r(currChain, currLength, bestChain, bestLength,
56                      pieces, num);
57
58 /* search the solution: try the first piece "inverted" */
59 currChain[0] = -i;
60 bestLength = chain_r(currChain, currLength, bestChain, bestLength,
61                      pieces, num);
62
63 /* output result */
64 fp = fopen(argv[3], "w");
65 fprintf(fp, "%d\n", bestLength);
66 for (i=0; i<bestLength; i++) {
67     j = bestChain[i];
68     k = abs(j);
69     if (j > 0) {
70         fprintf(fp, "%d %d-%d\n", pieces[k].id, pieces[k].n1, pieces[k].n2);
71     } else {
72         fprintf(fp, "%d %d-%d\n", pieces[k].id, pieces[k].n2, pieces[k].n1);
73     }
74 }
75 fclose(fp);

```

```
76
77     free(pieces);
78     free(bestChain);
79     free(currChain);
80     return EXIT_SUCCESS;
81 }
82
83 /*
84 *   find the longest chain that can be formed
85 */
86 int chain_r (
87     int *currChain, int currLength, int *bestChain, int bestLength,
88     piece_t *pieces, int num
89 ) {
90     int j, lastNumber, stop;
91
92     if (currChain[currLength-1] > 0) {
93         lastNumber = pieces[currChain[currLength-1]].n2;
94     } else {
95         lastNumber = pieces[-currChain[currLength-1]].n1;
96     }
97
98     stop = 1;
99     for (j=1; j<=num; j++) {
100         if (pieces[j].taken == 0) {
101             if (pieces[j].n1 == lastNumber) {
102                 /* use the new piece "straight" */
103                 currChain[currLength++] = j;
104                 pieces[j].taken = 1;
105                 stop = 0;
106                 bestLength = chain_r(currChain, currLength, bestChain, bestLength,
107                                         pieces, num);
108                 pieces[j].taken = 0;
109                 currLength--;
110             } else if (pieces[j].n2 == lastNumber) {
111                 /* use the new piece "inverted" */
112                 currChain[currLength++] = -j;
113                 pieces[j].taken = 1;
114                 stop = 0;
115                 bestLength = chain_r(currChain, currLength, bestChain, bestLength,
116                                         pieces, num);
117                 pieces[j].taken = 0;
118                 currLength--;
119             }
120         }
121     }
122
123     if (stop && currLength>bestLength) {
124         /* no possible improvement, but better solution found */
125         bestLength = currLength;
126         for (j=0; j<currLength; j++) {
127             bestChain[j] = currChain[j];
128         }
129     }
130     return bestLength;
131 }
```

## 7.10 Shipping Firm

### Specifications

A shipping firm uses truck with a maximum payload of  $10^4$  kilograms.

Write a program able to:

- ▷ Receive from the command line a file name.
- ▷ Read from file all shipping details, that is:
  - ◊ The number of packages to ship (first line of the file).
  - ◊ For each package, its weight (in hundreds of kilograms), on one of the subsequent rows.
- ▷ Find a placement of all packages on the available trucks, such that the number of trucks used is minimum and the maximum payload is not exceeded.
- ▷ Print-out:
  - ◊ The number  $k$  of required trucks.
  - ◊ For each package, the identifier (an integer number from 0 to  $k - 1$ ) of the truck on which it has to be loaded.

**Example 7.6** Let us suppose that the following one

```
10
32
76
20
81
15
25
5
48
59
5
```

is the input file. As a consequence, the program has to produce an output such as the following one:

```
4 trucks needed.
Packages location:
+ Truck 0, weight 91
  - package n. 0, weight 32
  - package n. 8, weight 59
+ Truck 1, weight 91
  - package n. 1, weight 76
  - package n. 4, weight 15
+ Truck 2, weight 93
  - package n. 2, weight 20
  - package n. 5, weight 25
  - package n. 7, weight 48
+ Truck 3, weight 91
  - package n. 3, weight 81
  - package n. 6, weight 5
  - package n. 9, weight 5
```

## Solution

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MAX 100
5
6 /* structure declaration */
7 typedef struct {
8     int weight;
9     int truckBest;
10    int truckCurr;
11 } pack_t;
12
13 /* function prototypes */
14 pack_t *load (char *filename, int *num_ptr);
15 int put_r (pack_t *packages, int num, int *trucks, int nCurr, int nBest, int id);
16 void write (pack_t *packages, int num, int n);
17
18 /*
19  * main program
20 */
21 int main (int argc, char **argv) {
22     int *trucks, num, n=0;
23     pack_t *packages;
24
25     /* store the input data */
26     packages = load(argv[1], &num);
27     trucks = (int *)calloc(num, sizeof(int));
28     if (trucks==NULL) {
29         fprintf (stderr, "Memory allocation error.\n");
30         exit(EXIT_FAILURE);
31     }
32
33     /* look for the solution and display it */
34     n = put_r(packages, num, trucks, 0, num+1, 0);
35     write(packages, num, n);
36
37     free(trucks);
38     free(packages);
39
40     return EXIT_SUCCESS;
41 }
42
43 /*
44  * load the data structure from file
45 */
46 pack_t *load (char *filename, int *num_ptr) {
47     pack_t *packages;
48     FILE *fp;
49     int i, num;
50
51     fp = fopen(filename, "r");
52     fscanf(fp, "%d", &num);
53     packages = (pack_t *)malloc(num * sizeof(pack_t));
54     if (packages==NULL) {
55         fprintf (stderr, "Memory allocation error.\n");
56         exit(EXIT_FAILURE);
57     }

```

```

58 /* save the data for packages */
59 for (i=0; i<num; i++) {
60     fscanf(fp, "%d", &packages[i].weight);
61     packages[i].truckBest = packages[i].truckCurr = -1;
62 }
63
64 fclose(fp);
65 *num_ptr = num;
66
67 return packages;
68 }
69
70
71 /* 
72  * recursively put the packages on trucks
73 */
74 int put_r (pack_t *packages, int num, int *trucks, int nCurr, int nBest, int id) {
75     int i;
76
77     if (id == num) {
78         /* a solution has been found */
79         if (nCurr < nBest) {
80             for (i=0; i<num; i++) {
81                 packages[i].truckBest = packages[i].truckCurr;
82             }
83             return nCurr;
84         }
85         return nBest;
86     }
87
88     /* put the package number "id" to every truck */
89     for (i=0; i<nCurr; i++) {
90         if (trucks[i] + packages[id].weight <= MAX) {
91             packages[id].truckCurr = i;
92             trucks[i] += packages[id].weight;
93             nBest = put_r(packages, num, trucks, nCurr, nBest, id+1);
94             packages[id].truckCurr = -1;
95             trucks[i] -= packages[id].weight;
96         }
97     }
98
99     if (nCurr < nBest) {
100        /* put the package number "id" to a new truck */
101        packages[id].truckCurr = i;
102        trucks[i] += packages[id].weight;
103        nBest = put_r(packages, num, trucks, nCurr+1, nBest, id+1);
104        packages[id].truckCurr = -1;
105        trucks[i] -= packages[id].weight;
106    }
107
108    return nBest;
109 }
110
111 /*
112  * display the solution on screen
113 */
114 void write (pack_t *packages, int num, int n) {
115     int i, j, weight;
116

```

```

117     fprintf(stdout, "%d trucks needed.\n", n);
118     fprintf(stdout, "Packages location:\n");
119     for (i=0; i<n; i++) {
120         weight = 0;
121         for (j=0; j<num; j++) {
122             if (packages[j].truckBest == i) {
123                 weight += packages[j].weight;
124             }
125         }
126         fprintf(stdout, "+ Truck %d, weight %d\n", i, weight);
127         for (j=0; j<num; j++) {
128             if (packages[j].truckBest == i) {
129                 fprintf(stdout, " - Package n. %d, weight %d\n", j,
130                         packages[j].weight);
131             }
132         }
133     }
134 }
135 }
```

## 7.11 Television programs

### Specifications

In a very large house there are  $N$  television sets, where  $N$  is a predefined constant value. To avoid fighting all the time, the members of the family need a program able to solve the following problem.

A file stores the description of all television programs broadcast during the same day. Each television program is specified on a single file line by the following pieces of information:

name appreciation starting\_time ending\_time

where:

- ▷ name is a string (without spaces) specifying the program name.
- ▷ appreciation is a positive integer value which indicates the family approval for that transmission.
- ▷ starting\_time and ending\_time are two integer values indicating the starting and ending times for that transmission. Each time is approximated to the closest half an hour, i.e., 1 corresponds to a time range from 00:00 to 00:30, and 48 a time corresponding a time range from 23:30 to 24:00. As a consequence a program lasting from 21 to 23 will be broadcast from 10:00 to 11:30 in the morning.

The first line of the file stores the number of programs broadcast during the day.

The program has to:

- ▷ Read the input file and store it in a proper data structure.
- ▷ Find the best scheduling for all requested programs. A scheduling is the best one, when the number of requested programs broadcast on one television set is maximized.

The optimal solution has to be stored on a file. The format for this file is free.

Notice that:

- ▷ All input file programs are broadcast during one single day.

- ▷ All programs have to be followed “entirely” on the same television set, i.e., it is impossible to split a program on more television sets.
- ▷ Each television set can be tuned on only one program at a time.
- ▷ All television sets have to be tuned on different programs in each moment of the day.

## Solution

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define N 2
6 #define INTERVALS 49
7
8 /* structure declaration */
9 typedef struct {
10     char *name;
11     int like;
12     int start;
13     int end;
14 } program_t;
15
16 /* function prototypes */
17 program_t *load (int *);
18 int agenda_r (program_t *, int, int, int [][INTERVALS],
19     int, int [][INTERVALS], int);
20 int empty_section (int [], program_t);
21 void mark (int [], int, int, int);
22 void write_agenda (int [][INTERVALS], program_t *);
23 char *timetable (int, char *);
24
25 /*
26  * main program
27 */
28 int main (void) {
29     int tvCurr[N][INTERVALS], tvBest[N][INTERVALS];
30     program_t *programs;
31     int i, num, like;
32
33     /* initialize data structure */
34     programs = load(&num);
35     for (i=0; i<N; i++) {
36         mark(tvCurr[i], 0, INTERVALS, -1);
37     }
38
39     /* solve the problem */
40     like = agenda_r(programs, num, 0, tvCurr, 0, tvBest, 0);
41     fprintf(stdout, "Maximum pleasure = %d\n", like);
42     write_agenda(tvBest, programs);
43
44     /* quit memory */
45     for (i=0; i<num; i++) {
46         free(programs[i].name);
47     }
48     free(programs);
49     return EXIT_SUCCESS;
50 }
```

```

51
52  /*
53   * read in the input file
54  */
55 program_t *load (int *num_ptr) {
56     int i, n, like, start, end;
57     program_t *programs;
58     char buffer[100];
59     FILE *fp;
60
61     fp = fopen("programs.txt", "r");
62     fscanf(fp, "%d", &n);
63     programs = (program_t *)malloc(n*sizeof(program_t));
64     if (programs==NULL) {
65         fprintf (stderr, "Memory allocation error.\n");
66         exit(EXIT_FAILURE);
67     }
68
69     for (i=0; i<n; i++) {
70         fscanf(fp, "%s %d %d %d", buffer, &like, &start, &end);
71         programs[i].name = strdup(buffer);
72         programs[i].start = start;
73         programs[i].end = end;
74         programs[i].like = like*(end-start+1);
75     }
76
77     fclose(fp);
78     *num_ptr = n;
79     return programs;
80 }
81
82 /*
83  * recursively build the optimal agenda
84 */
85 int agenda_r (
86     program_t *programs, int num, int idx, int tvCurr[N][INTERVALS],
87     int likeCurr, int tvBest[N][INTERVALS], int likeBest
88 ) {
89     int i, j;
90
91     /* terminal case: compare solutions */
92     if (idx == num) {
93         if (likeCurr > likeBest) {
94             likeBest = likeCurr;
95             for (i=0; i<N; i++) {
96                 for (j=0; j<INTERVALS; j++) {
97                     tvBest[i][j] = tvCurr[i][j];
98                 }
99             }
100        }
101        return likeBest;
102    }
103
104    /* assign program idx to all the televisions in turn */
105    likeCurr += programs[idx].like;
106    for (i=0; i<N; i++) {
107        if (empty_section(tvCurr[i], programs[idx])) {
108            mark(tvCurr[i], programs[idx].start, programs[idx].end, idx);
109            likeBest = agenda_r(programs, num, idx+1, tvCurr, likeCurr,

```

```

110         tvBest, likeBest);
111     mark(tvCurr[i], programs[idx].start, programs[idx].end, -1);
112   }
113   likeCurr -= programs[idx].like;
114
115   /* now, try NOT to watch the current program: recur on the next one */
116   likeBest = agenda_r(programs, num, idx+1, tvCurr, likeCurr, tvBest, likeBest);
117
118   return likeBest;
119 }
120
121 /*
122  * check whether a given tv is available for a specified time interval
123  */
124 int empty_section (int atv[INTERVALS], program_t program) {
125   int i;
126
127   for (i=program.start; i<=program.end; i++) {
128     if (atv[i] != -1) {
129       return 0;
130     }
131   }
132
133   return 1;
134 }
135
136 /*
137  * mark a tv with a given program for a specified time interval
138  */
139 void mark (int atv[INTERVALS], int start, int end, int val) {
140   int i;
141
142   for (i=start; i<=end; i++) {
143     atv[i] = val;
144   }
145
146
147   return;
148 }
149
150 /*
151  * write the agenda on file
152  */
153 void write_agenda (int tv[N][INTERVALS], program_t *programs) {
154   char buffer[100];
155   int i, j, k;
156   FILE *fp;
157
158   fp = fopen("agenda.txt", "w");
159   for (i=0; i<N; i++) {
160     fprintf(fp, "TV%d\n", i+1);
161     j = 1;
162     while (j < INTERVALS) {
163       if (tv[i][j] == -1) {
164         j++;
165       } else {
166         /* write a program */
167         k = tv[i][j];
168         fprintf(fp, "ore %s ", timetable(j, buffer));

```

```

169         j = programs[k].end+1;
170         fprintf(fp, "- %s ", timetable(j, buffer));
171         fprintf(fp, "%s\n", programs[k].name);
172     }
173 }
175
176 fclose(fp);
177 }
178
179 /*
180 * convert a number to the corresponding (initial) time
181 */
182 char *timetable (int x, char *buffer) {
183     int hour, minute;
184
185     hour = (x-1)/2;
186     minute = (x%2) ? 0 : 30;
187     sprintf(buffer, "%02d:%02d", hour, minute);
188     return buffer;
189 }
```

## 7.12 The Hitori Puzzle

### Specifications

The Japanese *Hitori* game has the following characteristics.

A square grid is initially filled-in with positive integer numbers. The target of the game is to rule-out the minimum number of numbers such that there will be no more than two equal values on the same row and column.

In the following example the initial grid is on the right-hand side and the final one on the left-hand side. To obtain the final configuration it is necessary to delete 7 values.

2	2	1	5	3
2	3	1	4	5
1	1	1	3	5
1	3	5	4	2
5	4	3	2	1

	2		5	3
2	3	1	4	
	1		3	5
1		5		2
5	4	3	2	1

Write a program able to read the initial grid from a first file and to store the final one on a second file. The size of the matrix puzzle is stored on the first line of the input file.

### Solution

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 /* structure declaration */
6 typedef struct {
7     int v;
8     int f;
9 } hitori_t;
```

```

10  /* function prototypes */
11 hitori_t **read_file (char *, int *);
12 int find_solution_r (hitori_t **, int, int, int, int);
13 int check_solution (hitori_t **, int);
14 void print_solution (char *, hitori_t **, int);
15
16 /*
17 * main program
18 */
19 int main (int argc, char *argv[]) {
20     hitori_t **hitori;
21     int n, i, ok=0;
22
23     hitori = read_file(argv[1], &n);
24     fprintf(stdout, "Initial matrix:\n");
25     print_solution("stdout", hitori, n);
26
27     for (i=0; i<n*n && !ok; i++) {
28         if (find_solution_r(hitori, n, i, 0, 0)) {
29             ok = 1;
30             /* output on screen */
31             fprintf(stdout, "Solution (#deleted numbers = %d):\n", i);
32             print_solution("stdout", hitori, n);
33             /* output on file */
34             print_solution(argv[2], hitori, n);
35         }
36     }
37
38
39     for (i=0; i<n; i++) {
40         free(hitori[i]);
41     }
42     free(hitori);
43
44     return EXIT_SUCCESS;
45 }
46
47 /*
48 * store the input file contents
49 */
50 hitori_t **read_file (char *name, int *nptr) {
51     hitori_t **hitori;
52     int n, i, j;
53     FILE *fp;
54
55     fp = fopen(name, "r");
56     fscanf(fp, "%d", &n);
57
58     hitori = (hitori_t **)malloc(n*sizeof(hitori_t *));
59     if (hitori==NULL){
60         fprintf (stderr, "Memory allocation error.\n");
61         exit(EXIT_FAILURE);
62     }
63     for (i=0; i<n; i++) {
64         hitori[i] = (hitori_t *)malloc(n*sizeof(hitori_t));
65         if (hitori[i]==NULL){
66             fprintf (stderr, "Memory allocation error.\n");
67             exit(EXIT_FAILURE);
68         }

```

```

69     for (j=0; j<n; j++) {
70         fscanf(fp, "%d", &hitori[i][j].v);
71         hitori[i][j].f = 1;
72     }
73 }
74 fclose(fp);
75
76 *nptr = n;
77 return hitori;
78 }
79
80 /*
81 * erase n numbers from the hitori matrix, recursive function
82 */
83 int find_solution_r (hitori_t **hitori, int n, int level, int r, int c) {
84     int i, j;
85
86     if (level == 0) {
87         return check_solution(hitori, n);
88     }
89
90     /* erase another number from the previous erased position on */
91     for (i=r; i<n; i++) {
92         for (j=(i==r)?c:0; j<n; j++) {
93             if (hitori[i][j].f != -1) {
94                 /* erase one number */
95                 hitori[i][j].f = -1;
96                 if (find_solution_r(hitori, n, level-1, i, j)) {
97                     return 1;
98                 }
99                 hitori[i][j].f = 1;
100            }
101        }
102    }
103
104    return 0;
105 }
106
107 /*
108 * check whether the current mutrix is a solution of the hitori game
109 */
110 int check_solution (hitori_t **hitori, int n) {
111     int i, j, k;
112
113     for (i=0; i<n; i++) {
114         for (j=0; j<n; j++) {
115             if (hitori[i][j].f != -1) {
116                 /* check row */
117                 for (k=0; k<n; k++) {
118                     if (k!=j && hitori[i][k].f!=-1 && hitori[i][j].v==hitori[i][k].v) {
119                         return 0;
120                     }
121                 }
122                 /* check column */
123                 for (k=0; k<n; k++) {
124                     if (k!=i && hitori[k][j].f!=-1 && hitori[i][j].v==hitori[k][j].v) {
125                         return 0;
126                     }
127                 }

```

```

128     }
129   }
130 }
131 return 1;
132 }
133 }
134 */
135 /* print the hitori solution on file (possibly on screen)
136 */
137 void print_solution (char *name, hitori_t **hitori, int n) {
138   FILE *fp;
139   int r, c;
140
141   if (strcmp(name, "stdout") == 0) {
142     fp = stdout;
143   } else {
144     fp = fopen(name, "w");
145   }
146
147   for (r=0; r<n; r++) {
148     for (c=0; c<n; c++) {
149       if (hitori[r][c].f == -1) {
150         fprintf(fp, "    ");
151       } else {
152         fprintf(fp, "%3d ", hitori[r][c].v);
153       }
154     }
155     fprintf(fp, "\n");
156   }
157
158   if (strcmp(name, "stdout") != 0) {
159     fclose(fp);
160   }
161 }
162 }
```

## 7.13 Soccer competition

### Specifications

Write a program able to organize a soccer competition with the following specifications.

Each team has 5 members, and there are  $5 \cdot n$  players enrolled. Each player is characterized by:

- ▷ A playing role: goalkeeper ('G'), defender ('D'), and striker ('S').
- ▷ An overall ability, i.e., an integer number in the range [1, 100]. Higher the number, better the player.

Those player features are stored in a file with the following format:

```

50
D1234 90 D
D189 20 S
D1231 70 G
D18439 40 D
...
```

which indicates that there are 50 players (and the 10 teams), and that, for example, the first player has an identifier equal to D1234, an ability equal to 90 and he/she is a defender.

The program has to divide all enrolled players into  $n$  teams, such that each team includes 5 players, among which at least one goalkeeper, one defender, and one striker. Moreover, if we define a team force the sum of all players within the team, the program has to minimize the difference between the force of the stronger and the weaker teams.

## Solution

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define NUM      5
6 #define ROLES    3
7 #define MAX     100
8
9 /* structure declaration */
10 typedef struct {
11     char *name;
12     int force;
13     int role;
14     int team;
15 } player_t;
16
17 typedef struct {
18     int players[NUM];
19     int np;
20     int force;
21 } team_t;
22
23 /* function prototypes */
24 player_t *read_players (char *, int *);
25 int assign_r(player_t *, int, team_t *, int, int, int, int);
26 int test_team (player_t *, int, int);
27 void print_teams (team_t *, int, player_t *);
28
29 /*
30  * main program
31 */
32 int main(int argc, char **argv) {
33     int np, unbalance, i;
34     player_t *players;
35     team_t *teams;
36
37     /* init the internal data strucure */
38     players = read_players(argv[1], &np);
39     teams = (team_t *)calloc(np/NUM, sizeof(team_t));
40     if (teams==NULL) {
41         fprintf (stderr, "Memory allocation error.\n");
42         exit(EXIT_FAILURE);
43     }
44
45     /* find and display a solution */
46     unbalance = assign_r(players, np, teams, 0, 0, NUM*MAX, -1);
47     if (unbalance >= 0) {

```

```

48     print_teams(teams, np, players);
49 } else {
50     fprintf(stdout, "No solution found!\n");
51 }
52 /* free memory */
53 for (i=0; i<np; i++) {
54     free(players[i].name);
55 }
56 free(players);
57 free(teams);
58
59 return EXIT_SUCCESS;
60 }
61
62 /*
63 * read in the input file
64 */
65 player_t *read_players (char *fname, int *num_ptr) {
66     char name[20], role;
67     player_t *players;
68     int i, np;
69     FILE *fp;
70
71     fp = fopen(fname, "r");
72     fscanf(fp, "%d", &np);
73     players = (player_t *)malloc(np * sizeof(player_t));
74     if (players==NULL) {
75         fprintf (stderr, "Memory allocation error.\n");
76         exit(EXIT_FAILURE);
77     }
78     for (i=0; i<np; i++) {
79         fscanf(fp, "%s %d %c", name, &players[i].force, &role);
80         players[i].name = strdup(name);
81         switch (role) {
82             case 'G': players[i].role = 0; break;
83             case 'D': players[i].role = 1; break;
84             case 'S': players[i].role = 2; break;
85         }
86     }
87     fclose(fp);
88
89     *num_ptr = np;
90     return players;
91 }
92
93 /*
94 * recursively assign players to teams
95 */
96 int assign_r (
97     player_t *players, int np, team_t *teams, int id,
98     int max_force, int min_force, int unbalance
99 ) {
100     int i, j, k, new_max, new_min, recur;
101
102     if (id == np) {
103         /* better solution found: save it */
104         for (i=0; i<np/NUM; i++) {
105             for (k=0, j=0; j<np; j++) {
106

```

```

107         if (players[j].team == i) {
108             teams[i].players[k++] = j;
109         }
110     }
111 }
112 return max_force - min_force;
113 }
114

115 /* assign current player (id) to all the teams in turn */
116 for (i=0; i<np/NUM; i++) {
117     new_max = max_force;
118     new_min = min_force;
119     recur = 1;
120     if (teams[i].np < NUM) {
121         players[id].team = i;
122         teams[i].force += players[id].force;
123         teams[i].np++;
124

125     /* is the team full? */
126     if (teams[i].np == NUM) {
127
128         /* then, check the role constraint ... */
129         if (!test_team(players, np, i)) {
130             /* the constraint is violated */
131             recur = 0;
132         }
133
134         /* ... and check for the unbalance */
135         if (teams[i].force > max_force) {
136             new_max = teams[i].force;
137         }
138         if (teams[i].force < min_force) {
139             new_min = teams[i].force;
140         }
141         if (unbalance>=0 && new_max-new_min>=unbalance) {
142             /* the current division is not optimal */
143             recur = 0;
144         }
145     }
146
147     if (recur) {
148         unbalance = assign_r(players, np, teams, id+1,
149                               new_max, new_min, unbalance);
150     }
151
152     /* backtrack */
153     teams[i].force -= players[id].force;
154     teams[i].np--;
155 }
156
157 /* optimization: do not generate symmetric solutions */
158 if (teams[i].np == 0) {
159     break;
160 }
161 }
162
163 return unbalance;
164 }
165

```

```

166  /*
167   * test on team composition
168 */
169 int test_team (player_t *players, int np, int team) {
170     int i, role[ROLES]={0};
171     for (i=0; i<np; i++) {
172         if (players[i].team == team) {
173             role[players[i].role]++;
174         }
175     }
176
177     for (i=0; i<ROLES; i++) {
178         if (role[i] == 0) {
179             return 0;
180         }
181     }
182
183     return 1;
184 }
185
186 /*
187  * display the solution
188 */
189 void print_teams (team_t *teams, int np, player_t *players) {
190     int i, j, force;
191
192     fprintf(stdout, "\n");
193     for (i=0; i<np/NUM; i++) {
194         for (force=0, j=0; j<NUM; j++) {
195             force += players[teams[i].players[j]].force;
196         }
197
198         fprintf(stdout, "Team %d - Total strength %d\n", i+1, force);
199         for (j=0; j<NUM; j++) {
200             fprintf(stdout, "%s ", players[teams[i].players[j]].name);
201             switch (players[teams[i].players[j]].role) {
202                 case 0: fprintf(stdout, "(G - "); break;
203                 case 1: fprintf(stdout, "(D - "); break;
204                 case 2: fprintf(stdout, "(S - "); break;
205             }
206             fprintf(stdout, "%d)\n", players[teams[i].players[j]].force);
207         }
208         fprintf(stdout, "\n");
209     }
210 }
211 }
```

## 7.14 String concatenation

### Specifications

A file stores a set of strings. Each string has at most 30 characters, and it is stored on a different file row. The first row of the file specifies the number of strings stored in the file itself.

Write a C program able to print out the longest string that can be generated adopting the following set of rules:

- ▷ Each string within the file can be used at most  $n$  times, where  $n$  is an integer value specified by the user.

- ▷ Two strings  $s_1$  and  $s_2$  can be concatenated if and only if the last two characters of  $s_1$  coincides with the first two characters of  $s_2$ .
- ▷ When the two strings are concatenated those two characters are not repeated. For example, if  $s_1$  is “giorno” and  $s_2$  is “notte”, then concatenating  $s_1$  and  $s_2$  generates “giornotte”.
- ▷ The string order within the file is meaningless to generate the longest string.

**Example 7.7** Let us suppose that  $n = 2$  and that the file content is the following one:

```
9
novara
torino
vercelli
ravenna
napoli
livorno
messina
noviligure
roma
```

the program has to print-out the following string:

Longest string (50 characters):

"torinovaravennapolivornovaravennapolivornoviligure"

## Solution

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 #define NUM    2
6
7 /* structure declaration */
8 typedef struct {
9     char *word;
10    int n;
11 } string;
12
13 /* macro definition */
14 #define strend(word)          ((word) + strlen(word) - NUM)
15 #define strmid(word)          ((word) + NUM)
16 #define chainable(word1, word2) (strncmp(strend(word1), word2, NUM)==0)
17
18 /* function prototypes */
19 string *read_file (int *, int);
20 int chain_r (string *, int, int *, int, int *, int, int);
21 void print_result (string *, int *, int);
22
23 /*
24  * main program
25 */
26 int main (void) {
27     int i, dim, nrip, *chainBest, *chainCurr, length;
28     string *array;
29
30     fprintf(stdout, "Input number of allowed repetitions: ");
```

```
31     scanf("%d", &nrip);
32     array = read_file(&dim, nrip);
33
34     /* allocate temporary arrays (current and chainBest solution) */
35     chainCurr = (int *)malloc((dim*nrip + 1) * sizeof(int));
36     chainBest = (int *)malloc((dim*nrip + 1) * sizeof(int));
37     if (chainCurr==NULL || chainBest==NULL) {
38         fprintf (stderr, "Memory allocation error.\n");
39         exit(EXIT_FAILURE);
40     }
41     for (i=0; i<=dim*nrip; i++) {
42         chainCurr[i] = chainBest[i] = -1;
43     }
44
45     /* search solution */
46     length = chain_r(array, dim, chainCurr, NUM, chainBest, NUM, 0);
47
48     /* output result */
49     print_result(array, chainBest, length);
50
51     /* free memory */
52     for (i=0; i<dim; i++) {
53         free(array[i].word);
54     }
55     free(array);
56     free(chainBest);
57     free(chainCurr);
58
59     return EXIT_SUCCESS;
60 }
61
62 /*
63  * load the input file
64  */
65 string *read_file (int *dim_ptr, int nrip) {
66     string *array;
67     char word[31];
68     int dim, i;
69     FILE *fp;
70
71     fprintf(stdout, "Input file name: ");
72     scanf("%s", word);
73     fp = fopen(word, "r");
74     fscanf(fp, "%d", &dim);
75     array = (string *)malloc(dim * sizeof(string));
76     if (array==NULL) {
77         fprintf (stderr, "Memory allocation error.\n");
78         exit(EXIT_FAILURE);
79     }
80
81     for (i=0; i<dim; i++) {
82         fscanf(fp, "%s", word);
83         array[i].word = strdup(word);
84         array[i].n = nrip;
85     }
86
87     fclose(fp);
88     *dim_ptr = dim;
89     return array;
```

```
90     }
91
92     /*
93      * build the longest string that can be generated, recursive function
94     */
95     int chain_r (
96         string *array, int dim, int *chainCurr, int lengthCurr,
97         int *chainBest, int lengthBest, int level
98     ) {
99         int i, j, last;
100
101        last = (level == 0) ? -1 : chainCurr[level-1];
102        for (i=0; i<dim; i++) {
103            if (last== -1 || (array[i].n>0 && chainable(array[last].word, array[i].word))) {
104                chainCurr[level] = i;
105                array[i].n--;
106                lengthCurr += strlen(strmid(array[i].word));
107                if (lengthCurr > lengthBest) {
108                    lengthBest = lengthCurr;
109                    for (j=0; j<=level; j++) {
110                        chainBest[j] = chainCurr[j];
111                    }
112                    chainBest[level+1] = -1;
113                }
114
115                lengthBest = chain_r(array, dim, chainCurr, lengthCurr,
116                                      chainBest, lengthBest, level+1);
117
118                chainCurr[level] = -1;
119                array[i].n++;
120                lengthCurr -= strlen(strmid(array[i].word));
121            }
122        }
123
124        return lengthBest;
125    }
126
127    /*
128     * display the result
129     */
130    void print_result (string *array, int *chainBest, int lengthBest) {
131        int i, j;
132
133        fprintf(stdout, "Longest string (%d chars):\n", lengthBest);
134        fprintf(stdout, "%s", array[chainBest[0]].word);
135        i = 1;
136        while (chainBest[i] != -1) {
137            j = chainBest[i++];
138            fprintf(stdout, "%s", strmid(array[j].word));
139        }
140        fprintf(stdout, "\n");
141
142        return;
143    }
```

# Chapter 8

## Modularity, Multi-file Applications and Abstract Data Types

### 8.1 Variables and Function Attributes

It is known that a variable has a *name*, a *type*, a *size*, and a *value*. For example

```
int i;
```

defines a variable named *i*, of integer type, of (usually) 4 bytes (i.e., 32 bits), and whose value is undefined. Similarly, function

```
float foo (char c);
```

has name *foo* and it returns an abject of type *float*.

Actually, each identifier, i.e., a variable or a function name, in a C program has other attributes, including *storage class*, *storage duration*, *scope* and *linkage*. We will analyze those attributes in the following sub-sections with particular emphasis to the storage class.

#### 8.1.1 Storage Class and Duration

The storage class determines the part of memory where storage is allocated for an object (particularly variables and functions). The storage class also determines the object storage duration. An identifier's storage duration is the period during which the identifier exists in memory, i.e., how long the storage allocation continues to exist. Some exist briefly, some are repeatedly created and destroyed, and others exist for the program's entire execution.

C provides four storage class specifiers: *automatic* (or *auto*), *register*, *extern* and *static*. Keep in mind that in the hardware terms we have primary storage such as registers, cache, memory (Random Access Memory) and secondary storage such as magnetic and optical disk.

##### **Automatic Objects**

Keyword *auto* is used to declare variables of automatic storage duration. Variables with automatic storage duration are created when the block in which they are defined is entered. They exist while the block is active. Finally, they are destroyed when the block is exited.

**Example 8.1** The following piece of code shows three different situations in which an automatic (`auto`) variable can be defined: On top of the main program, on top of any function, and on top of any block of instructions.

```
int main (...) {
    int i;
    float f;

    ...

}

<type> function (...) {
    int i;
    float f;

    {
        // This is a block of instruction
        int j, k;
        char c;
        ...
    }

    ...
}
```

Please note that variables `i` and `f` are local for both the main program and the function, then they have no relationship (they are completely different variables) even if they share the same names.

Keyword `auto` explicitly declares variables of automatic storage duration. For example, defining local (to the main, a function or a block) variables:

```
auto int i, j;
auto float f;
```

is equivalent to

```
int i, j;
float f;
```

because local variables have automatic storage duration by default. As a consequence the keyword `auto` is rarely used, even if the automatic storage class is the most common one being the default for all local variables.

Automatic variables are defined and considered “local” to the block in which they are defined. They have to be explicitly initialized, as they do not have a predefined default values after the definition. This can be done during the definition itself or before they are actually used:

```
int i = 0, j;
float f = 2.5;

...
```

When the block is exited, the system releases the memory that was reserved for the automatic variable. Thus, the values of these variables is lost. If the block is re-entered, the system once again allocates the memory for the variable, but the previous values remain unknown. In other words, the invocation of a new variable or block sets up a new fresh environment.

Notice that only variables (not functions) can have automatic storage duration.

## Register Objects

The storage class *register* tells the compiler that the associated variables should be stored in high-speed memory registers, provided it is physically and semantically possible. Essentially, the use of this storage class is an attempt to improve execution speed. In other words, when speed is a concern the programmer may choose a few variables that are frequently used and define them as belonging to the storage class *register*. For example

```
register int i;
```

will define variable *i* as a critical variable for efficiency, and it will instruct the compiler to allocate it in a hardware register. Notice, however, that system resource limitations can make this impossible. In other words, this class automatically become *auto* whenever the compiler cannot allocate an appropriate physical register.

Somehow, as compiler have become exceedingly efficient to optimize programmer's code, the keyword *register* is archaic and should be used as least as possible.

## Extern Objects

The use and meaning of the keyword *extern* differs from variables and functions.

As far as variables are concerned, the most common method to transmit information across blocks and functions, is to use global variables. Global variables are created by placing variable definitions outside any function definition. Global variables are of storage class *extern* by default. Storage (the required memory) is permanently assign to those objects, and global variables retain their values throughout the entire execution of the program.

Typically, when the program spans a unique .c file, and variables are defined at the beginning of a source file, the *extern* keyword can be omitted. For example

```
int i, j, k;
```

placed outside any function, defines 3 *extern* (global) variables. Those variables cannot have automatic or *register* storage class. They can have the storage class *static* but its use is special as explain in the next sub-section.

**Example 8.2** The following piece of code shows the definition of both local (automatic) and global (*extern*) variables.

```
// Global
int gi1;

int main (...) {
    // Local
    int li;
    ...
}

// Global
int gi2;
float gf;

<type> function (...) {
    // Local
    int li;
    ...
}
```

```
// Local
float lf;
...
}
...
}
```

The keyword **extern** must be used with more than one .c file or to use a variable within the same .c file but before the original definition within that file. If the program is organized on several source files, and a variable is defined in let say `file1.c` and used in `file2.c` and `file3.c`, then the **extern** keyword must be used in `file2.c` and `file3.c`. In this case the keyword **extern** is used it tells the compiler to “look for the object elsewhere, either in the same or in another file”. That is, in the following definitions

```
int i, j, k;
```

appears in `file1.c`, then `file2.c` and `file3.c` should include:

```
extern int i, j, k;
```

where the use of **extern** tells the compiler that the variables are defined elsewhere in the same or in another file. On the same .c file, if the **int** definition does not appear on top, the the **extern int** declaration can be used before the **int** definition to make the variable visible. In other words, the **int** definition actually defines the variables, whereas the **extern int** declaration actually tells that those variables are defined elsewhere but that they can be used in that “new” environment too.

**Example 8.3** Figure 8.1 shows the usage of the **extern** keyword for variables in a multiple-file scheme.

#### file1.c

```
// Global (definition)
int i;
float f;

int main (...)

<type> function1 (...)

<type> function2 (...)
```

#### file2.c

```
// Global (declaration)
extern int i;
// i can be used from here on

...

// Global (declaration)
extern float f;
// f can be used from here on

...
```

**Figure 8.1** Extern (global) variables with two files.

**Example 8.4** The following piece of code shows the usage of the **extern** keyword in a single-file scheme.

```
// Global (definition)
float f;
// Global (declaration)
extern int i;
```

```
// This makes i "global", i.e., can be used from here on too
int main (...)

 function1 (...)

// Global (definition)
int i;
// i can be used from here on

<type> function2 (...)

{...}
```

External variables may be initialized in declarations just as automatic variables; however, the initializers must be constant expressions. The initialization is done only once at compile time, i.e., when memory is allocated for the variables. In general, it is a good programming practice to avoid using external variables as they destroy the concept of a function as a “black box” or independent module. The black box concept is essential to the development of a modular program with modules. With an external variable, any function in the program can access and alter the variable, thus making debugging more difficult as well. This is not to say that external variables should never be used. There may be occasions when the use of an external variable significantly simplifies the implementation of an algorithm. Suffice it to say that external variables should be used rarely and with caution.

The use of `extern` for functions has a slightly different meaning. All function are of storage class `extern` by default. It is allowed but not required, i.e., the definition

```
extern float f (int);
```

is equivalent to the following one

```
float f (int);
```

as all functions are “global” by default.

Global functions (and variables) can be referenced by any function that follows their declarations or definitions in the file. This is one reason for using function prototypes: The function prototype is placed at the start of our file to make the function known to the rest of the file.

## Static Objects

Static declarations have two important and distinct uses.

The more elementary is to allow a local variable to retain its previous value when a block is re-entered. This is in contrast to ordinary automatic variables, which lose their value upon block exit and must be re-initialized.

For example, let us analyze function

```
void f () {
    static int counter = 0;

    fprintf (stdout, "%d - ", counter);
    counter++;

    return;
}
```

The first time the function is called, the variable `counter` is defined and it is initialized to zero. On function exit, the value of the counter is preserved. Whenever the function is invoked, the counter is not re-initialized. Instead, it retains its previous value, i.e., the one it had last time the function was called. In other words, defining the static variable `counter` (retaining its values) allows the function to print an increasing integer value each time it is called.

All numeric variables of static storage are initialized to zero by default if you do not explicitly initialize them.

The second use is in connection with external declarations. With external construct the keyword `static` provides a “privacy” mechanism. By privacy, in this case, we mean a scope restriction. An identifier's scope is where the identifier can be referenced in a program. Some can be referenced throughout a program, others from only portions of a program. “Static external” variables are scope-restricted external variables. The scope is the remainder of the source file in which they are declared. Thus, they are unavailable to functions defined earlier in the same file or to functions defined in other files, even if those functions attempt to use the keyword `extern`.

For example

```
static void f ();
static int v;
```

defines a function `f` and a variable `v` whose scope (visibility) is restricted to the file including the definition from the definition line on. Somehow, `f` and `v` are local but global object, i.e., local (restricted) to that file but global to the family of functions defined in the file after that line.

**Example 8.5** Figure 8.2 shows the usage of the `extern` and `static` keywords for functions in a multiple-file application.

### file1.c

```
<prototypes>

int main (...)

// Global function definition
<type> function1 (...)

// Local function definition
static <type> function2 (...)
```

### file2.c

```
...

// Prototype for a global function
extern <type> function1 (...);
/*
 * From here on
 * it is possible to use function1
 */
...

/*
 * This is wrong
 * the linker should deliver an error
 * function2 is local to file1.c and
 * it cannot be exported outside
 */
extern <type> function2 (...);
```

**Figure 8.2** Extern (global) and static (local) functions with two files.

**Example 8.6** Figure 8.3 shows another example on the use `extern` and `static` keywords for functions in a multiple-file application.

file1.c

```
<prototypes>

int main (...)

// Global function definition
<type> function1 (...)

// Local function definition
static <type> function2 (...)
```

file2.c

```
...
// Prototype for a global function
extern <type> function1 (...);

/*
 * From here on
 * it is possible to use function1
 */
...

/*
 * This is another (local) definition
 * of function2, different from the one
 * in file1.c
 */
static <type> function2 (...)
```

**Figure 8.3** Extern (global) and static (local) functions with two files.

## Default Initialization

In C, both external variables and static variables that are not explicitly initialized by the programmer are initialized to zero by the system. This includes arrays, strings, pointers, and structures. For arrays and strings, this means that each element is initialized to zero. For structures it means that each member is initialized to zero. In contrast to this, automatic and register variables usually are not initialized by the system. This means they start with “garbage” values.

### 8.1.2 Scope

The scope of an identifier is the portion of the program in which the identifier can be referenced, i.e., the part of the program where a variable name is visible. For example, when we define a local variable in a block, it can be referenced only following its definition in that block or in blocks nested within that block. Notice that storage duration (the period during which the object exists in memory), and scope (where the object can be used) are separate issues.

There are four possible scopes for an identifier: Function scope, file scope, block scope, and function-prototype scope.

Labels (identifiers followed by a colon such as `start:)`) are the only identifiers with function scope. As we will not use labels in this book, we will not consider that issue in more details.

An identifier declared outside any function has *file scope*. Such an identifier is “known” (i.e., accessible) in all functions from the point at which the identifier is declared until the end of the file. Global variables, function definitions, and function prototypes placed outside a function all have file scope.

Identifiers defined inside a block have *block scope*. Block scope ends at the terminating right brace of the block. Local variables defined at the beginning of a function have block scope, as do function parameters, which are considered local variables by the function. Any block may contain variable definitions. When blocks are nested, and an identifier in an outer block has the same name as an identifier in an inner block, the identifier in the outer block is hidden until the inner block terminates. This means that while executing in the inner block, the inner block sees the value of its own local identifier and not the value of the identically named identifier in the enclosing block.

Local variables declared static still have block scope, even though they exist from before program start-up. Thus, storage duration does not affect the scope of an identifier.

The only identifiers with *function-prototype* scope are those used in the parameter list of a function prototype. Function prototypes do not require names in the parameter list-only types are required. If a name is used in the parameter list of a function prototype, the compiler ignores the name. Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity.

### 8.1.3 Linkage

An identifiers linkage determines for a multiple-source-file program whether the identifier is known only in the current source file or in any source file with proper declarations.

### 8.1.4 A complete example

The following example demonstrates scoping issues with global variables, automatic local variables and static local variables.

A global variable *x* is defined and initialized to 1 (line 9). This global variable is hidden in any block (or function) in which a variable named *x* is defined. In main, a local variable *x* is defined and initialized to 5 (line 13). This variable is then printed to show that the global *x* is hidden in main. Next, a new block is defined in main with another local variable *x* initialized to 7 (line 20). This variable is printed to show that it hides *x* in the outer block of main. The variable *x* with value 7 is automatically destroyed when the block is exited, and the local variable *x* in the outer block of main is printed again to show that it's no longer hidden.

```

1 #include <stdio.h>
2
3 // prototypes
4 void function1 ();
5 void function2 ();
6 void function3 ();
7
8 // global variable
9 int x = 1;
10
11 int main (void) {
12     // local variable to main
13     int x = 5;
14
15     printf ("local x in outer scope of main is %d\n", );
16
17     {
18         // start new scope

```

```

19 // local variable to new scope
20 int x = 7;
21 printf( "local x in inner scope of main is %d\n", x );
22 // end new scope
23 }
24 printf( "local x in outer scope of main is %d\n", x );
25
26 function1(); // function1 has automatic local x
27 function2(); // function2 has static local x
28 function3(); // function3 uses global x
29 function1(); // function1 re-initializes automatic local x
30 function2(); // static local x retains its prior value
31 function3(); // global x also retains its value
32
33 printf( "\nlocal x in main is %d\n", x );
34 }
35
36 // Re-initializes local variable x during each call
37 void function1 (void) {
38     int x = 25;
39     printf( "local x in function1 is %d after entering function1\n", x );
40     ++x;
41     printf( "local x in function1 is %d before exiting function1\n", x );
42
43     return;
44 }
45
46
47 // Initializes static local variable x only the first time
48 // value of x is saved between calls
49 void function2 (void) {
50     // initialized once before program start-up
51     static int x = 50;
52     printf ("local static x is %d on entering function2\n", x );
53     ++x;
54     printf( "local static x is %d on exiting function2\n", x );
55
56     return;
57 }
58
59 // Modifies global variable x during each call
60 void function3 (void) {
61     printf ("global x is %d on entering function3\n", x );
62     x *= 10;
63     printf( "global x is %d on exiting function3\n", x );
64
65     return;
66 }

```

The program defines three functions that each take no arguments and return nothing.

Function `function1` defines an automatic variable `x` and initializes it to 25 (line 39). When `function1` is called, the variable is printed, incremented, and printed again before exiting the function. Each time this function is called, automatic variable `x` is reinitialized to 25.

Function `function2` defines a static variable `x` and initializes it to 50 in line 51 (recall that the storage for static variables is allocated and initialized only once, before the program begins execution). Local variables declared as static retain their values

even when they are out of scope. When `function2` is called, `x` is printed, incremented, and printed again before exiting the function. In the next call to this function, static local variable `x` will contain the value 51.

Function `function3` does not define any variables. Therefore, when it refers to variable `x`, the global `x` (line 9) is used. When `function3` is called, the global variable is printed, multiplied by 10, and printed again before exiting the function. The next time function `function3` is called, the global variable still has its modified value, 10. Finally, the program prints the local variable `x` in `main` again (line 34) to show that none of the function calls modified the value of `x` because the functions all referred to variables in other scopes.

## 8.2 Multiple-Source-File Programs

### 8.2.1 Program Development Environment

Starting from the very beginning, developing a program in C typically requires six phases:

- ▷ Editing. During this phase, the source code files are written by the programmer using an editor program.
- ▷ Pre-processing. In a C system, a preprocessor program executes automatically before the compiler. The C preprocessor obeys special commands called preprocessor directives, which indicate that certain manipulations are to be performed on the program before compilation. These manipulations usually consist of including other files in the file to be compiled and performing various text replacements.
- ▷ Compilation. The compiler translates the C program into machine language-code (also referred to as object code). A syntax error occurs when the compiler cannot recognize a statement because it violates the rules of the language. The compiler issues an error message to help you locate and fix the incorrect statement.
- ▷ Linking. C programs typically contain references to functions defined elsewhere, such as in the standard libraries or in the private libraries of groups of programmers working on a particular project. The object code produced by the C compiler typically contains “holes” due to these missing parts. A linker links the object code with the code for the missing functions to produce an executable image (with no missing pieces).
- ▷ Loading. Before a program can be executed, the program must first be placed in memory. This is done by the loader, which takes the executable image from disk and transfers it to memory. Additional components from shared libraries that support the program are also loaded.
- ▷ Executing. The computer, under the control of its CPU, executes the program one instruction at a time. This phase usually reveals logical errors and require debugging.

Small programs are usually included in a unique `.c` file. Each `.c` files include all required system library files `.h`, with the directive `#include <...h>`). Moreover, each application is usually organized through a main program and several functions.

The following example shows how those `.c` files can be organized following two main schemes.

**Example 8.7** Figure 8.4 shows two possible schemes used to write a program on a single file. On the left-hand side the use of prototypes make the order in which functions have been defined indifferent. On the right-hand side prototypes are not inserted, then functions have to be defined such that each definition comes before any calls. In the example we suppose that function2 is called by function1 which in turn is called by the main program.

### scheme 1

```
#include ...
#define ...
typedef ...

<prototypes>

int main (...)
{...}

<type> function1 (...)
{...}

<type> function2 (...)
{...}
```

### scheme 2

```
#include ...
#define ...
typedef ...

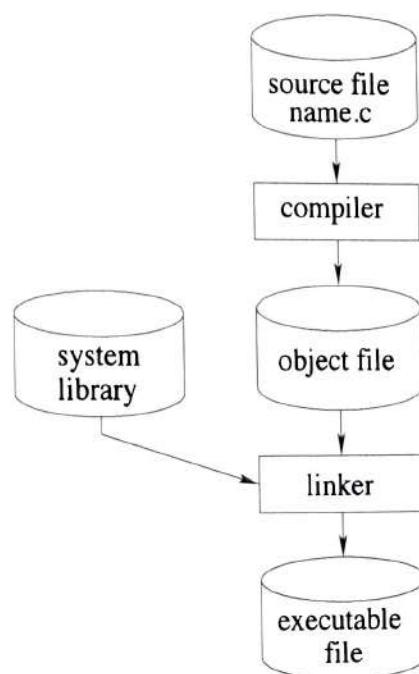
<type> function2 (...)
{...}

<type> function1 (...)
{...}

int main (...)
{...}
```

**Figure 8.4** Compiling and linking a single-file application.

Both the above programs, can be compiled and produce an object file as previously described and represented in Figure 8.5.



**Figure 8.5** Program development for a single-file application.

The object file can then be read by the linker to produce the executable file. In this

last phase all system libraries (such as `stdio.h`, `stdlib.h`, `string.h`, etc.) are used to “complete” the source code written by the programmer with the proper system code.

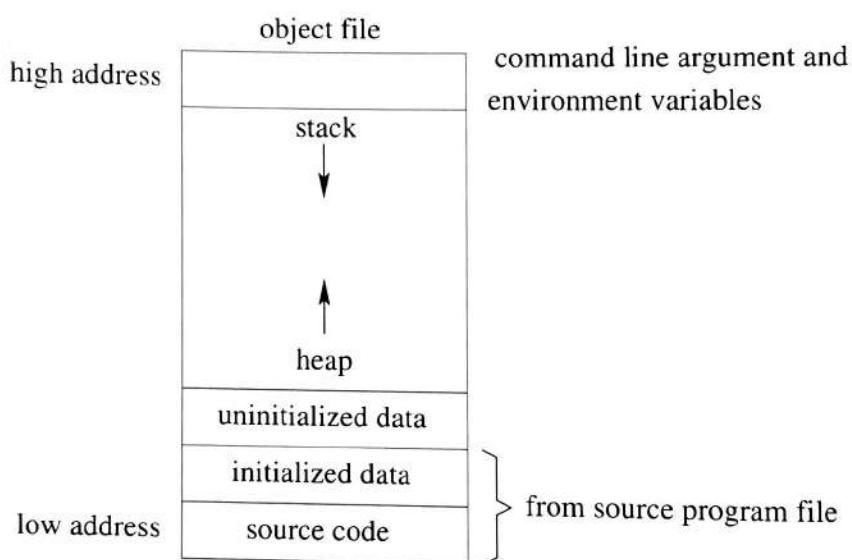
The memory structure of those file will be discussed in the next section.

### 8.2.2 Memory Layout of C Programs

A typical memory representation of C program consists of following sections:

- ▷ Text segment.
- ▷ Initialized data segment.
- ▷ Uninitialized data segment.
- ▷ Stack.
- ▷ Heap.

as represented in Figure 8.6.



**Figure 8.6** Object file structure: system stack, heap, data segment, and source code segment.

### Text Segment

A text segment, also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions. As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, and so on. The text segment is often read-only, to prevent a program from accidentally modifying its instructions.

### Data Segments

A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer. Note that,

data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into initialized read-only area and initialized read-write area.

Variables such as `int i = 10;` will be stored in the initialized data segment.

Data in uninitialized data segment is initialized by the kernel to arithmetic 0 before the program starts executing. Uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code. For instance a variable declared as `static int i;` or a global variable declared as `int j;` would be contained in the uninitialized data segment.

### **Stack Segment**

The stack area traditionally adjoined the heap area and grew the opposite direction. When the stack pointer met the heap pointer, free memory is exhausted. The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On some architectures it grows toward address zero; on some others it grows the opposite direction. A “stack pointer” register tracks the top of the stack. It is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”. A stack frame consists at minimum of a return address. Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

To summarize, the stack has a very fast access as the space is managed efficiently by CPU, and it will not become fragmented. It contains only local variables. Variables on the stack cannot be re-sized, and don’t have to be explicitly de-allocated. Its size is limited by the operating system.

**Example 8.8** The following program allocates all its variables on the stack.

```
int main (void) {
    int i, j, k;
    float f = 12.5;
    char v[10];
    ...
}
```

### **Heap Segment**

The heap is a region of your computer’s memory that is not managed automatically for you, and is not as tightly managed by the CPU. It is a more free-floating region of memory (and is larger). To allocate memory on the heap, you must use `malloc`, `calloc`, and `realloc`, which are built-in C functions. Once you have allocated

memory on the heap, you are responsible for using `free` to de-allocate that memory once you don't need it any more. If you fail to do this, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes). Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer). Heap memory is slightly slower to be read from and written to, because one has to use pointers to access memory on the heap. Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

To summarize, the heap is used to allocate variables which can be accessed globally. There is no limit on memory size, and a relatively slow access, as it is not guaranteed to be efficient in the space use as memory may become fragmented (as blocks of memory are allocated). Users must manage memory, they are in charge of allocating and freeing variables.

**Example 8.9** The following program allocates all its variables on the heap.

```
int main (void) {
    int *p = malloc (sizeof (int));
    float *f = malloc (sizeof (float));
    char *v = malloc (10 * sizeof (char));

    p = 12;
    v[0] = 'A';
    ...

    return 0;
}
```

### 8.3 Multiple-File Applications

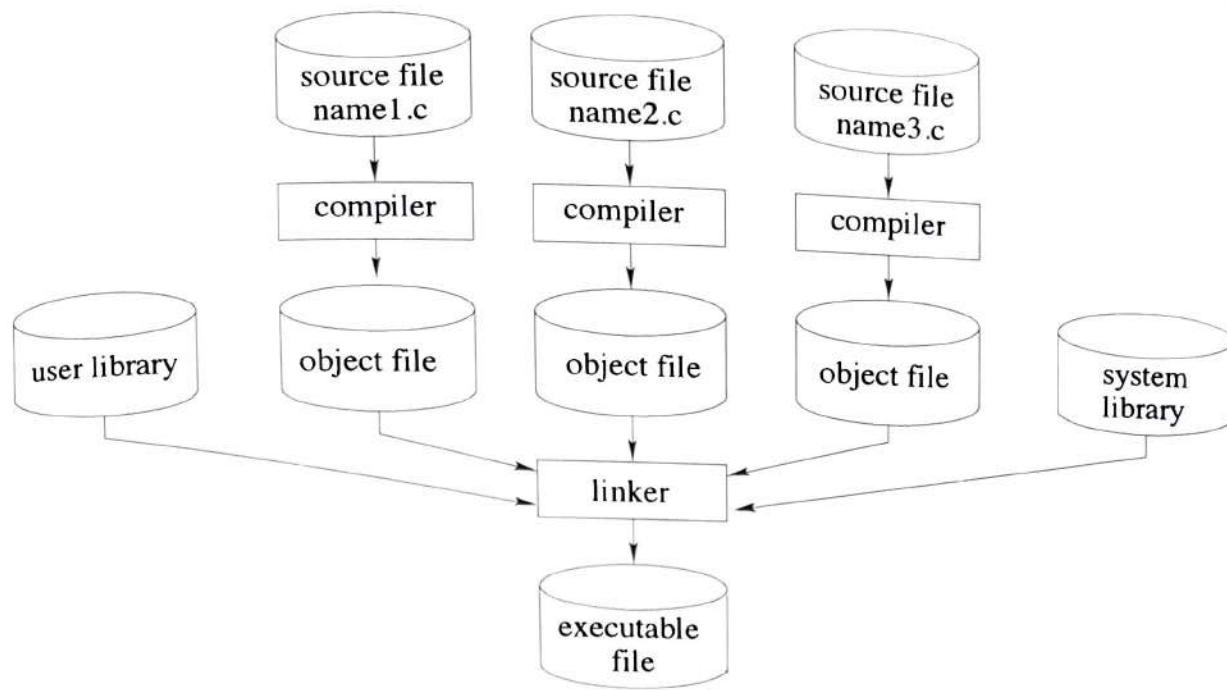
For complex applications, .c files become larger and includes too many functions. Their compilation requires long times, as debugging and maintaining them. Moreover, sharing common pieces of code is practically impossible as everything is included in the same file, and the only option is to duplicate part of the code in another file, with subsequent congruence problems.

As a consequence, typically a large program is written as a collection of source files (both header .h and C .c files) stored in separate directories. Each .c file usually contains one or more function definitions. One specific .c file includes the main program (that has to be unique). Each .h file usually include functions prototypes, structure and constant definitions, etc. User header files are included with the directive `include "name.h"`.

Picture 8.7 represents how to compile and link a multi-file application with 3 source files.

System libraries include `stdio.h`, `stdlib.h`, `string.h`, etc. User libraries include all .h files written by the programmer.

When building large programs in multiple source files, compiling the program becomes tedious if small changes are made to one file and the entire program must be recompiled. Each .c file can be re-compiled as needed, without recompiling all the other (i.e., unchanged) files. This saves time for both the programmer and the hard-



**Figure 8.7** Compiling and linking a multi-file application.

ware platform. Many systems provide special utilities that recompile only the modified program file.

**Example 8.10** Let us suppose we are developing a large program called pgrm. At the top of each of our .c files, we insert the line

```
#include 'pgrm.h'
```

When the preprocessor encounters this directive, it looks first in the current directory for the file pgrm.h. If there is such a file, then this is included in the .c file. If not, then the preprocessor looks in other system-dependent directories for the file. If the file pgrm.h cannot be found, then the preprocessor issues an error message and compilation stops. The pgrm.h file may contain include, defines, enumeration types, structure type definitions, and a list of function prototypes. All these elements are appropriate to be included in every .c file including pgrm.h. At the same time pgrm.h will not contain function definitions, that have to be unique and inserted in a .c file.

### 8.3.1 Functions and Variables Rules

One of the main problem for novice programmer is to understand what to insert in .c and .h files. As previously stated, .c files usually include executable instructions. Files with the .h extension store all global declarations (often starting with the keyword extern even if this is not required for function prototypes). All module (.c files) which wants to use those objects have to include the header file. Please remind that is better to make global only objects that really have to be used outside the module in which they are defined, whereas it is better to keep private all other objects.

Following Section 8.1 functions have to satisfy the following rules:

- ▷ If a module (a .c file) wants to export a function it does not have to do anything, as all C functions are global by default.

- ▷ Each module (another .c file) that wants to use that function has to insert its prototype, eventually (but this is optional) with the keyword `extern`.
- ▷ If a module (a .c file) wants to keep a function private, i.e., it does not want to make this function global, it has to define that function as `static`.

Notice that the linker (not the compiler) will create the required links between each call and its correct function definition. Those have to coincide completely, otherwise the linker will complain.

Following Section 8.1 variables have satisfy the following rules:

- ▷ Local variables (i.e., variables defined within a function or a block) cannot be exported (global).
- ▷ If a module (a .c file) wants to export a variable it does not have to do nothing, as all global C variable can be exported by default.
- ▷ Each module (another .c file) that wants to use that variable has to insert its declaration, i.e., the keyword `extern` followed by its definition.
- ▷ If a module (a .c file or a function) wants to keep a variable private, i.e., it does not want to make this variable exportable, it has to define that function as `static`.

Notice again that the linker (not the compiler) will create the required links between each declaration and its corresponding definition. Declaration and definition have to coincide completely, otherwise the linker will complain.

The following are two examples of how a small application can be modularized.

**Example 8.11** Figure 8.8 shows how to divide a very small program between a source and an header file.

**file.c**

```
#include "file.h"

int main (void) {
    int i;

    for (i=C1; i<C2; i++) {
        fprintf (stdout, "%d ", i);
    }

    return (0);
}
```

**file.h**

```
#include <stdio.h>

#define C1 10
#define C2 100
```

**Figure 8.8** A small program with a single .c and a single .h file.

**Example 8.12** Figure 8.9 shows a small application reading (from standard input) and writing (on standard output) a static array of integers. One .c file includes the main program, one the function reading the array, and one the function writing the array. `main.h` is the programmer's header file including all required prototypes and definitions.

The three .c files can be compiled separately, but they have to be linked together to generate a unique executable code.

**main.c**

```
#include "main.h"

int main (void) {
    int dim;
    int vet[L];

    array_read (vet, &dim);
    array_write (vet, dim);

    return 1;
}
```

**array\_write.c**

```
#include "main.h"

void
array_write (
    int *vet,
    int dim
)
{
    int i;

    fprintf (stdout, "Array:\n");
    for (i=0; i<dim; i++) {
        printf ("vet (%d) = %d\n",
               i, vet[i]);
    }

    return;
}
```

**main.h**

```
#include <stdio.h>

#define L 100

void array_read (int *, int *);
void array_write (int *, int);
```

**array\_read.c**

```
#include "main.h"

void
array_read (
    int *vet,
    int *dim
)
{
    int i;

    printf ("Size (<%d): ", L);
    scanf ("%d", dim);

    printf ("Array:\n");
    for (i=0; i<(*dim); i++) {
        printf ("vet (%d) = ", i);
        scanf ("%d", &vet[i]);
    }

    return;
}
```

**Figure 8.9** A C application reading and writing a static array of integer values with two separate .c files and one .h file.

### 8.3.2 Further Hints

There are several considerations when creating programs in multiple files. For example, the definition of a function must be entirely contained in one file, i.e., it cannot span two or more files. Global variables may be accessible to functions in other files, but they must be declared following the rules described in Section 8.1.

**Example 8.13** To refer to a global integer variable `flag` defined within `file1.c` as:

```
int flag;
within file file2.c, we must add in file2.c the declaration
extern int flag;
```

This declaration uses the storage-class specifier `extern` to indicate that variable `flag` is defined either later in the same file or in a different file. The compiler informs the linker that unresolved

references to variable `flag` appear in the file. If the linker finds a proper global definition, the linker resolves the references by indicating where `flag` is located. If the linker cannot locate a definition of `flag`, it issues an error message and does not produce an executable file. Any identifier that is declared at file scope is `extern` by default.

Just as `extern` declarations can be used to declare global variables to other program files, function prototypes can extend the scope of a function beyond the file in which it's defined (the `extern` specifier is not required in a function prototype). Simply include the function prototype in each file in which the function is invoked and compile the files together. Function prototypes indicate to the compiler that the specified function is defined either later in the same file or in a different file. Again, the compiler does not attempt to resolve references to such a function—that task is left to the linker. If the linker cannot locate a proper function definition, the linker issues an error message.

**Example 8.14** As an example of using function prototypes to extend the scope of a function, consider any program containing the preprocessor directive `#include <stdio.h>`. Other functions in the file can use `printf` and `scanf` to accomplish their tasks. Those functions are defined in other files. We do not need to know where they are defined. We are simply reusing the code in our programs. The linker resolves our references to these functions automatically. This process enables us to use the functions in the standard library.

It is possible to restrict the scope of a global variable or a function to the file in which it is defined. The storage-class specifier `static`, when applied to a global variable or a function, prevents it from being used by any function that's not defined in the same file. This is referred to as internal linkage. Global variables and functions that are not preceded by `static` in their definitions have external linkage, i.e., they can be accessed in other files if those files contain proper declarations and/or function prototypes. The global variable declaration

```
static const double PI = 3.14159;
```

creates constant variable `PI` of type `double`, initializes it to `3.14159` and indicates that `PI` is known only to functions in the file in which it is defined.

The `static` specifier is commonly used with utility functions that are called only by functions in a particular file. If a function is not required outside a particular file, the principle of least privilege should be enforced by using `static`. If a function is defined before it is used in a file, `static` should be applied to the function definition. Otherwise, `static` should be applied to the function prototype.

### 8.3.3 Once-Only Headers

Very often, one header file includes another. It can easily result that a certain header file is included more than once. If a header file happens to be included twice, the compiler will process its contents twice. The following example shows how that is possible.

**Example 8.15** Let us analyze the application frame reported in Figure 8.10. The main program included in `f3.c`, may need to use two libraries, the first one stored in `f2.c`, and the other one in `f1.c`. To do that the programmer writing `f3.c` includes in his/her code (i.e., `f3.h`) both `f1.h` and `f2.h`.

Unfortunately the programmer does not know that library `f2.c` is already using `f1.c` by itself, then `f2.h` includes `f1.h`. The problem here is that `f3.c` ends-up including the same libraries

**f1.h**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
...
#define C1 10
...
```

**f1.c**

```
#include "f1.h"
...
```

**f2.h**

```
#include "f1.h"
...
#define C2 "abc"
...
```

**f2.c**

```
#include "f2.h"
...
```

**f3.h**

```
#include "f1.h"
#include "f2.h"
...
#define C3 12.50
...
```

**f3.c**

```
#include "f3.h"
...
```

**Figure 8.10** The “once-only header” problem. The same libraries (e.g., stdio.h) and objects (e.g., L1) may be included more than once.

(e.g., stdio.h) and other objects (e.g., L1) more than once because it includes them directly but also through f2.h.

The previous effect is very likely to cause errors, if the header file defines structure types or typedefs. Even if it does not, it is certainly wasteful, and it will waste time. Therefore, we often wish to prevent multiple inclusion of a header file. The standard way to do this is to enclose the entire real contents of the file in a conditional, like this:

```
#ifndef HEADER_FILE_SEEN
#define HEADER_FILE_SEEN
...
THE ENTIRE FILE
...
#endif
```

The macro HEADER\_FILE\_SEEN indicates that the file has been included once already, and it is usually called the *controlling macro* or *guard macro*. In any kind of header file, the macro name should contain the name of the file and some additional text, to avoid conflicts with other header files. In a user header file, the macro name should not begin with “\_”, whereas in a system header file, this name should begin with “\_” to avoid conflicts with user programs. Anyway these are just conventions and can be somehow ignored to use other programming schemes. Several C preprocessor are programmed to notice when a header file uses this particular construct and handle it efficiently. If a header file is contained entirely in a #ifndef conditional, then it records that fact. If a subsequent #include specifies the same file, and the macro in the #ifndef is already defined, then the file is entirely skipped, without even reading it.

**Example 8.16** The solution to the previous problem is sketched in Figure 8.11. All header files are protected against multiple-inclusion by the directive ifndef-define-endif.

Notice that here is also an explicit directive to tell the preprocessor that it need not include a file more than once (#pragma once), and there is a variant of #include called #import, but we are not going to discuss those options.

f1.h

```
#ifndef _F1
#define _F1

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
...
#define C1 10
...
#endif
```

f2.h

```
#ifndef _F2
#define _F2

#include "f1.h"
...
#define C2 "abc"
...
#endif
```

f3.h

```
#ifndef _F3
#define _F3

#include "f1.h"
#include "f2.h"
...
#define C3 12.50
...
#endif
```

**Figure 8.11** Solution to the “once-only header” problem. The same libraries cannot be inserted more than once because if any header constant is already defined preclude any further inclusion.

## 8.4 Abstract Data Types (ADTs)

C does not have proper support to Abstract Data Types (ADTs). The target of this section is to learn how to increase the abstraction level of our application, hiding, as long as possible, implementation details. Thus, information hiding is the main target of this chapter. We will specifically follow the style and terminology defined by Sedgewick (see [5] for further details).

**Definition** An Abstract Data Type (ADT) is a data type, i.e., a set of values and a collection of operations on those values, that is accessed only through an *interface*. We refer to a program that uses an ADT as a *client*, and to a program that specifies the data type as an *implementation*.

Following this definition, the client can use the implementation only through the interface. The representation of the data and the functions that implement the operations are in the implementation. Those are completely separated from the client, by the interface. It is said that the interface is opaque, i.e., the client cannot see details of the implementation outside the one reported within the interface. In other words:

- ▷ Client: Is the program using the ADT.
- ▷ Interface: Includes the ADT declarations, i.e., all data type and related function prototypes.
- ▷ Implementation: Includes the ADT implementation, i.e., all functions definitions realizing the ADT, and all other object definitions. We will often call the implementation as “user” *library*.

In this scheme, the user will create his/her own *libraries*, which, as well as standard libraries, will often be reused. While writing a new client, to solve the current task, the programmer will reuse the same libraries, i.e., old implementations throughout their own interfaces. It is possible to construct ADTs, that we can manipulate in the same way we manipulate built-in types in client programs, while achieving the objective of hiding the implementation from the client.

**Example 8.17** We write an application dealing with points in the Cartesian space. The appli-

cation should be able to define points, and to compute the distance between two space points. The interface of such an application is included in file `point.h`. This file stores the type definition together with all exported functions declarations. Figure 8.12 reports all codes divided into the client (file `client.c`), interface (file `point.h`), and implementation (file `point.c`).

**client.c**

```
#include "point.h"

int main (void) {
    point_t p1, p2;
    float d;

    p1.x = p1.y = p1.z = 0.0;
    p2.x = p2.y = p2.z = 10.0;
    d = dist (p1, p2);
    fprintf (stdout, "D = %f\n", d);

    return 1;
}
```

**point.h**

```
#ifndef _POINT
#define _POINT

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

typedef struct point_s {
    float x, y, z;
} point_t;

float dist (point_t, point_t);
#endif
```

**point.c**

```
#include "point.h"

float dist (point_t p1, point_t p2) {
    float d;

    d = 0;
    d = d + (p1.x - p2.x) * (p1.x - p2.x);
    d = d + (p1.y - p2.y) * (p1.y - p2.y);
    d = d + (p1.z - p2.z) * (p1.z - p2.z);
    d = sqrt (d);

    return d;
}
```

**Figure 8.12** Application for space point: ADT version number 1.

In the previous example as the client includes the interface the C structure type is fully visible from the client as the structure definition is visible. As a consequence, this ADT reaches only partial data hiding, even if function prototypes are defined and included correctly. For that reason the following definition follows.

**Definition** A First-Class Data Type is one for which we can have potentially many different instances, and which we can assign to variables which we can declare to hold an instance.

With a first-class ADT, the client programs can manipulate the data without direct access, but rather with indirect access, through operations defined in the ADT. Each operation can have different implementation within the implementation program, even if the interface defines it in the same way (please analyze Section 9.6 for a complete example of this ideas).

Very often, the term *handle* (or *wrapper*) is used to indicate a reference to an abstract object. The goal is to give client programs handles to abstract objects that can be used in assignment statements and as arguments and return value of functions in the same way as built-in data types, while hiding the representation of the objects from the client program.

The following example presents a first version of a first class ADT for space points.

**Example 8.18** Figure 8.13 shows a first implementation for a first class ADT for space points. The interface **point.h** includes only function prototypes and type definitions. Data are defined directly into the implementation **point.c**. As the client **client.c** does not see the data definition, data hiding is full. The price for this result is that the client does not see the data type and it has to use proper functions to allocate, access, and deallocate it (functions `new`, `set`, and `disp`).

In this case the data definition is only inserted into the implementation and it is not visible from the client. The client includes the interface, which includes function prototypes and the type (`typedef`) definition. However, the data type has to be managed only through pointers (handles or wrappers), and it needs dynamic memory allocation. The interface has to provide all functions to get, set (getters and setters), destroy, etc., all data type fields. The client cannot access data fields directly, but it can manage them only through interface functions.

The programming scheme illustrated in Example 8.18 can be further extended in two directions. First of all, instead of defining the new type within the implementation, we can use a “public” header and a “private” header, as follows.

**Example 8.19** Figure 8.14 shows a first class ADT with a public and a private header files. The structure is cleaner, but more complex. It is up to the user to decide which approach is more indicated given the problem at hand.

In this case the client only includes the public interface **complexPublic.h**. Through it, the client will have access to public functions and type pointers as well. On the contrary, the implementation does include the private interface **complexPrivate.h** and through it the public interface too.

The second extension concerns pointers definitions. Instead of defining the new type as a pointer, and hiding the pointer nature of the object within the client, we can define it directly and using it as a pointer within the client.

**Example 8.20** In Figure 8.15 the client includes the public interface **complexPublic.h** as before, but it accesses the **complex\_t** type explicitly as a pointer, i.e., as **complex\_t \***, instead then implicitly.

This definition is more suited for a native C programming style where pointers are made explicit. Notice however, that other programming styles (see for example the Java language or even the Windows API programming environment in C) are used to hide pointer definitions to avoid using the `*` operator explicitly. As a matter of fact, it all boils down to the programmer’s preferences.

#### 8.4.1 ADT Summary Example

In this section we will revise all main concepts of this chapter with one more simple ADT multi-file application. More complex examples will be presented in all following

**client.c**

```
#include "point.h"

int main (void) {
    point_t p1, p2;
    float d;

    p1 = new ();
    p2 = new ();
    set (p1, 0.0, 0.0, 0.0);
    set (p2, 10.0, 10.0, 10.0);
    d = dist (p1, p2);
    sprintf (stdout, "D = %f\n", d);
    disp (p1);
    disp (p2);

    return 1;
}
```

**point.h**

```
#ifndef _POINT
#define _POINT

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

typedef struct point_s *point_t;
point_t new ();
void set (point_t, float, float,
          float);
float dist (point_t, point_t);
void disp (point_t);

#endif
```

**point.c**

```
#include "point.h"

struct point_s {
    float x, y, z;
};

point_t new () {
    point_t p;
    p = malloc (1 * sizeof (struct point_s));
    if (p == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
    return p;
}
void set (point_t p, float x, float y, float z) {
    p->x = x; p->y = y; p->z = z;
}
float dist (point_t p1, point_t p2) {
    float d;
    d = 0;
    d = d + (p1->x - p2->x) * (p1->x - p2->x);
    d = d + (p1->y - p2->y) * (p1->y - p2->y);
    d = d + (p1->z - p2->z) * (p1->z - p2->z);
    d = sqrt (d);
    return d;
}
void disp (point_t p) {
    free (p);
}
```

**Figure 8.13** Application for space point: ADT version number 2.

sections of this chapter.

Our application manipulates complex numbers, i.e., it allows the user to define a

complex number (expressed as  $(real + img \cdot i)$ , where *real* and *img* are two real numbers and *i* is the imaginary unit) and to take the product of two of them.

The first example reports the simplest case in which the implementation is disclosed to the client: The interface of the complex library is inserted in file `complex.h` which is included by the client.

**Example 8.21** Figure 8.16 report files `client.c`, `complex.h`, and `complex.c`.

The second example shows the case in which the structure is directly defined within the C file. This structure allows the programmer not to write a more complex header file structure. Again, notice that the nature of the data structure is hidden to the client. This in turns implies implementing all required access and manipulation functions.

**Example 8.22** Figure 8.17 report files `client.c`, `complex.h`, and `complex.c`.

The third example shows the application with a public and a private interface. The nature of the pointer to the structure is still hidden.

**Example 8.23** Figure 8.18 report files `client.c`, `complexPublic.h`, `complexPrivate.h`, and `complex.c`.

The last example shows the application with a public and a private interface and with explicit pointer definitions.

**Example 8.24** Figure 8.19 report files `client.c`, `complexPublic.h`, `complexPrivate.h`, and `complex.c`.

**client.c**

```
#include "pointPublic.h"
int main (void) {
    point_t p1, p2;
    float d;
    p1 = new ();
    p2 = new ();
    set (p1, 0.0, 0.0, 0.0);
    set (p2, 10.0, 10.0, 10.0);
    d = dist (p1, p2);
    fprintf (stdout, "D = %f\n", d);
    disp (p1);
    disp (p2);
    return 1;
}
```

**pointPublic.h**

```
#ifndef _POINT_PUBLIC
#define _POINT_PUBLIC
#include <stdio.h>
typedef struct point_s *point_t;
point_t new ();
void set (point_t, float, float,
          float);
float dist (point_t, point_t);
void disp ();
#endif
```

**pointPrivate.h**

```
#ifndef _POINT_PRIVATE
#define _POINT_PRIVATE
#include <stdlib.h>
#include <math.h>
#include "pointPublic.h"
struct point_s {
    float x, y, z;
};
#endif
```

**point.c**

```
#include "pointPrivate.h"
point_t new () {
    point_t p;
    p = malloc (1 * sizeof (struct point_s));
    if (p == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
    return p;
}
void set (point_t p, float x, float y, float z) {
    p->x = x; p->y = y; p->z = z;
}
float dist (point_t p1, point_t p2) {
    float d;
    d = 0;
    d = d + (p1->x - p2->x) * (p1->x - p2->x);
    d = d + (p1->y - p2->y) * (p1->y - p2->y);
    d = d + (p1->z - p2->z) * (p1->z - p2->z);
    d = sqrt (d);
    return d;
}
void disp (point_t p) {
    free (p);
}
```

**Figure 8.14** Application for space point: ADT version number 3.

**client.c**

```
#include "pointPublic.h"
int main (void) {
    point_t *p1, *p2;
    float d;
    p1 = new ();
    p2 = new ();
    set (p1, 0.0, 0.0, 0.0);
    set (p2, 10.0, 10.0, 10.0);
    d = dist (p1, p2);
    fprintf (stdout, "D = %f\n", d);
    disp (p1);
    disp (p2);
    return 1;
}
```

**pointPublic.h**

```
#ifndef _POINT_PUBLIC
#define _POINT_PUBLIC
#include <stdio.h>
typedef struct point_s point_t;
point_t *new ();
void set (point_t *, float, float,
          float);
float dist (point_t *, point_t *);
void disp (point_t *);
#endif
```

**pointPrivate.h**

```
#ifndef _POINT_PRIVATE
#define _POINT_PRIVATE
#include <stdlib.h>
#include <math.h>
#include "pointPublic.h"
struct point_s {
    float x, y, z;
};
#endif
```

**point.c**

```
#include "pointPrivate.h"
point_t *new () {
    point_t *p;
    p = malloc (1 * sizeof (struct point_s));
    if (p == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
    return p;
}
void set (point_t *p, float x, float y, float z) {
    p->x = x; p->y = y; p->z = z;
}
float dist (point_t *p1, point_t *p2) {
    float d;
    d = 0;
    d = d + (p1->x - p2->x) * (p1->x - p2->x);
    d = d + (p1->y - p2->y) * (p1->y - p2->y);
    d = d + (p1->z - p2->z) * (p1->z - p2->z);
    d = sqrt (d);
    return d;
}
void disp (point_t *p) {
    free (p);
}
```

**Figure 8.15** Application for space point: ADT version number 4.

## client.c

```
#include "complex.h"

int main (void) {
    complex_t c1, c2, c3;

    c1.real = 1.0;
    c1.img = 1.0;
    c2.real = 2.0;
    c2.img = 2.0;
    c3 = mul (c1, c2);
    fprintf (stdout, "real=%f img=%f\n",
             c3.real, c3.img);

    return 1;
}
```

## complex.c

```
#include "complex.h"

complex_t mul (complex_t c1, complex_t c2) {
    complex_t c;

    c.real = c1.real*c2.real - c1.img*c2.img;
    c.img = c1.real*c2.img + c2.real*c1.img;

    return c;
}
```

## complex.h

```
#ifndef _COMPLEX
#define _COMPLEX

#include <stdio.h>

typedef struct complex_s {
    float real, img;
} complex_t;

complex_t mul (complex_t, complex_t);

#endif
```

Figure 8.16 Application for complex number: ADT version number 1.

**client.c**

```
#include "complex.h"

int main (void) {
    complex_t c1, c2, c3;
    float r, i;

    c1 = new (); c2 = new ();
    set (c1, 1.0, 1.0);
    set (c2, 2.0, 2.0);
    c3 = mul (c1, c2);
    get (c3, &r, &i);
    fprintf (stdout, "real=%f img=%f\n",
             r, i);
    disp (c1); disp (c2);

    return 1;
}
```

**complex.h**

```
#ifndef _COMPLEX
#define _COMPLEX

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

typedef struct complex_s *complex_t;
complex_t new ();
void set (complex_t, float, float);
void get (complex_t, float *,
          float *);
complex_t mul (complex_t, complex_t);
void disp (complex_t);

#endif
```

**complex.c**

```
#include "complex.h"

struct complex_s {
    float real, img;
};

complex_t new () {
    complex_t c;
    c = malloc (1 * sizeof (struct complex_s));
    if (c == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
    return c;
}
void set (complex_t c, float r, float i) {
    c->real = r; c->img = i;
}
void get (complex_t c, float *r, float *i) {
    *r = c->real; *i = c->img;
}
complex_t mul (complex_t c1, complex_t c2) {
    complex_t c;
    c = new ();
    c->real = c1->real * c2->real - c1->img * c2->img;
    c->img = c1->real * c2->img + c2->real * c1->img;
    return c;
}
void disp (complex_t c) {
    free (c);
}
```

**Figure 8.17** Application for complex number: ADT version number 2.

**client.c**

```
#include "complexPublic.h"
int main (void) {
    complex_t c1, c2, c3;
    float r, i;
    c1 = new ();
    set (c1, 1.0, 1.0);
    set (c2, 2.0, 2.0);
    c3 = mul (c1, c2);
    get (c3, &r, &i);
    fprintf (stdout, "real=%f img=%f\n", r, i);
    disp (c1); disp (c2);
    return 1;
}
```

**complexPublic.h**

```
#ifndef _COMPLEX_PUBLIC
#define _COMPLEX_PUBLIC
#include <stdio.h>
typedef struct complex_s *complex_t;
complex_t new ();
void set (complex_t, float, float);
void get (complex_t, float *,
          float *);
complex_t mul (complex_t, complex_t);
void disp (complex_t);
#endif
```

**complex.c**

```
#include "complexPrivate.h"
complex_t new () {
    complex_t c;
    c = malloc (1 * sizeof (struct complex_s));
    if (c == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
    return c;
}
void set (complex_t c, float r, float i) {
    c->real = r; c->img = i;
}
void get (complex_t c, float *r, float *i) {
    *r = c->real; *i = c->img;
}
complex_t mul (complex_t c1, complex_t c2) {
    complex_t c;
    c = new ();
    c->real = c1->real * c2->real - c1->img * c2->img;
    c->img = c1->real * c2->img + c2->real * c1->img;
    return c;
}
void disp (complex_t c) {
    free (c);
}
```

**complexPrivate.h**

```
#ifndef _COMPLEX_PRIVATE
#define _COMPLEX_PRIVATE
#include <stdlib.h>
#include "complexPublic.h"
struct complex_s {
    float real, img;
};
#endif
```

**Figure 8.18** Application for complex number: ADT version number 3.

**client.c**

```
#include "complexPublic.h"
int main (void) {
    complex_t *c1, *c2, *c3;
    float r, i;
    c1 = new ();
    set (c1, 1.0, 1.0);
    set (c2, 2.0, 2.0);
    c3 = mul (c1, c2);
    get (c3, &r, &i);
    fprintf (stdout, "real=%f img=%f\n", r, i);
    disp (c1); disp (c2);
    return 1;
}
```

**complexPublic.h**

```
#ifndef _COMPLEX_PUBLIC
#define _COMPLEX_PUBLIC
#include <stdio.h>
typedef struct complex_s complex_t;
complex_t *new ();
void set (complex_t *, float, float);
void get (complex_t *, float *,
          float *);
complex_t *mul (complex_t *,
                complex_t *);
void disp (complex_t *c);
#endif
```

**complexPrivate.h**

```
#ifndef _COMPLEX_PRIVATE
#define _COMPLEX_PRIVATE
#include <stdlib.h>
#include "complexPublic.h"
struct complex_s {
    float real, img;
};
#endif
```

**complex.c**

```
#include "complexPrivate.h"
complex_t *new () {
    complex_t *c;
    c = malloc (1 * sizeof (struct complex_s));
    if (c == NULL) {
        fprintf (stderr, "Memory allocation error.\n");
        exit (1);
    }
    return c;
}
void set (complex_t *c, float r, float i) {
    c->real = r; c->img = i;
}
void get (complex_t *c, float *r, float *i) {
    *r = c->real; *i = c->img;
}
complex_t *mul (complex_t *c1, complex_t *c2) {
    complex_t *c;
    c = new ();
    c->real = c1->real * c2->real - c1->img * c2->img;
    c->img = c1->real * c2->img + c2->real * c1->img;
    return c;
}
void disp (complex_t *c) {
    free (c);
}
```

**Figure 8.19** Application for complex number: ADT version number 4.

# Chapter 9

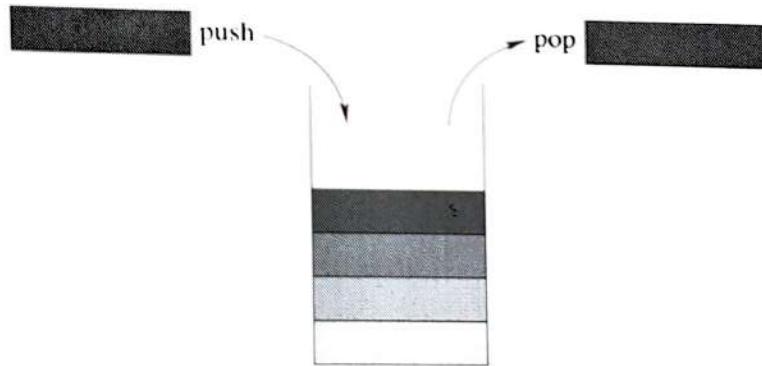
## ADTs and Classic Data Structures

In this chapter we will use notions introduced in Chapters 4 and 8 to describe some very common data structures in computer science.

### 9.1 A LIFO Stack

In computer science, a stack or LIFO (Last In, First Out) or stack is an abstract data type that serves as a collection of elements, with two principal operations, i.e., *push* and *pop*. Each push adds an element to the collection (whenever this is not full). Each *pop* removes the last element that was added (whenever the collection is not empty).

This strategy is typical of a “pile” of objects, where insertions and extractions are done on the same side. Figure 9.1 shows a standard representation for a stack.



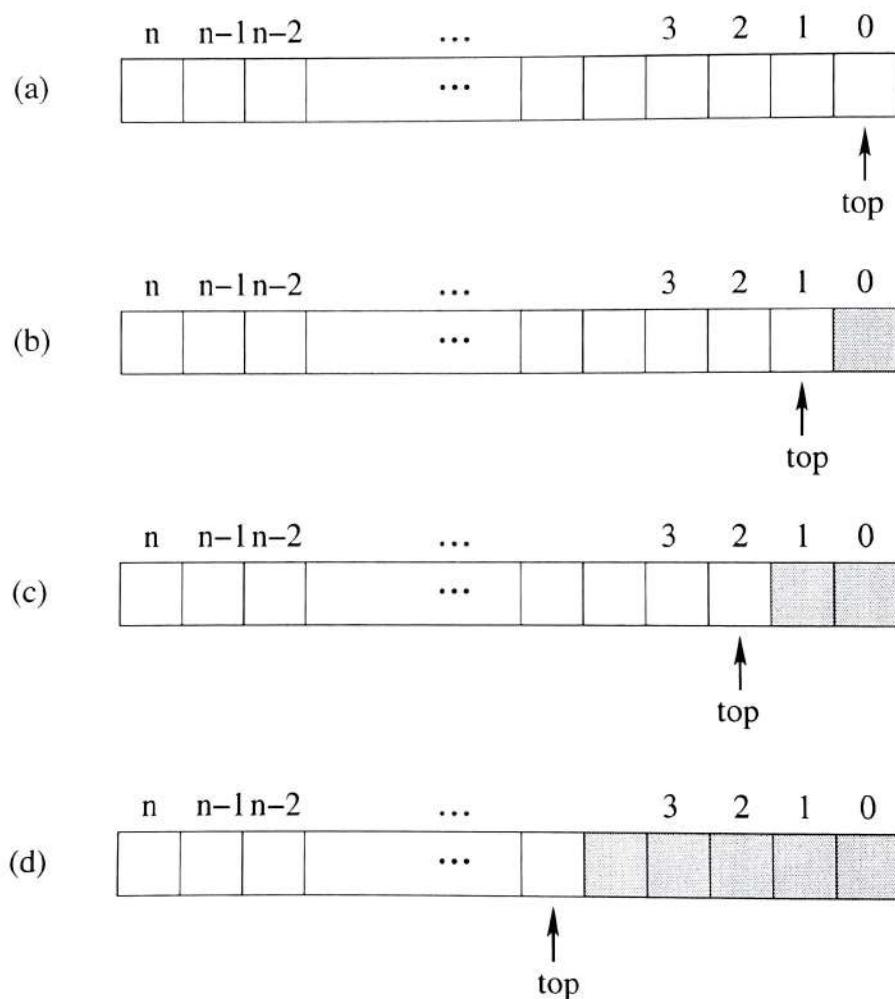
**Figure 9.1** Stack data structure with push and pop operations. Lighter elements have been inserted before darker ones. The extraction order will be the opposite (from darker to lighter objects).

When trying to pop an element from an empty stack, we should receive an *underflow* error. When trying to push an element onto a full stack, we should receive an *overflow* error. We would like to remind that among standard usage of the stack there is the function call mechanism, in which all local variables of the calling functions are freezed on the stack top.

Notice that we have already encountered stacks twice in this book, in Chapters 5 and 8. In those occasion we talked about the “system stack” used by the operating system to manage function calls (and therefore recursion). In this chapter we will concentrate on “user stacks”, i.e., stacks built by the programmers to solve his/her own design problems. There are several possible implementation of a stack. We will analyze a few of them in the following sub-sections.

### 9.1.1 Array implementation

A graphical representation of a stack implemented as a dynamic array is reported in Figure 9.2.



**Figure 9.2** Linear Stack Data Structure. (a) Initial configuration. (b) Configuration after one insertion. (c) Configuration after two insertions. (d) Stack structure after several insertions.

A LIFO structure behaves like a pile of objects, where insertions (pushes) and extraction (pops) are made on the same side of the array. An index (name `top`, or `TOS`, i.e., top of stack) is used to indicate the element for the next insertion. For each push operation, if the stack is not full, the value of `top` has to be increment before inserting the new element into the array:

```
if (top>=(size-1)) {
    /* Stack Full */
    ...
}
```

```

} else {
    top++;
    stack[top] = value;
}

For each pop operation, if the stack is not empty, the value can be extracted from the
stack and then the top value has to be incremented:
if (top==(-1)) {
    /* Stack Full */
    ...
} else {
    value = stack[top];
    top--;
}

```

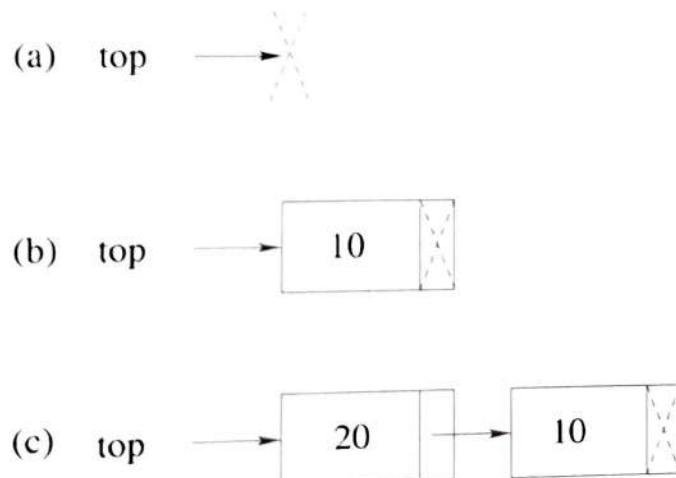
### 9.1.2 List Implementation

The list implementation can use the C data structure introduced in Section 4.2.1. At the very beginning the stack is supposed to be empty, then its entry point pointer `top` has to be initialized as:

```
top = NULL;
```

After that, each push can must create a new element. Notice that push and pop manipulate the same ending point of the data structure. Is then possible to use the methodology analyzed in Sections 4.2.5 and 4.2.4 to insert and extract element on top of a simple list.

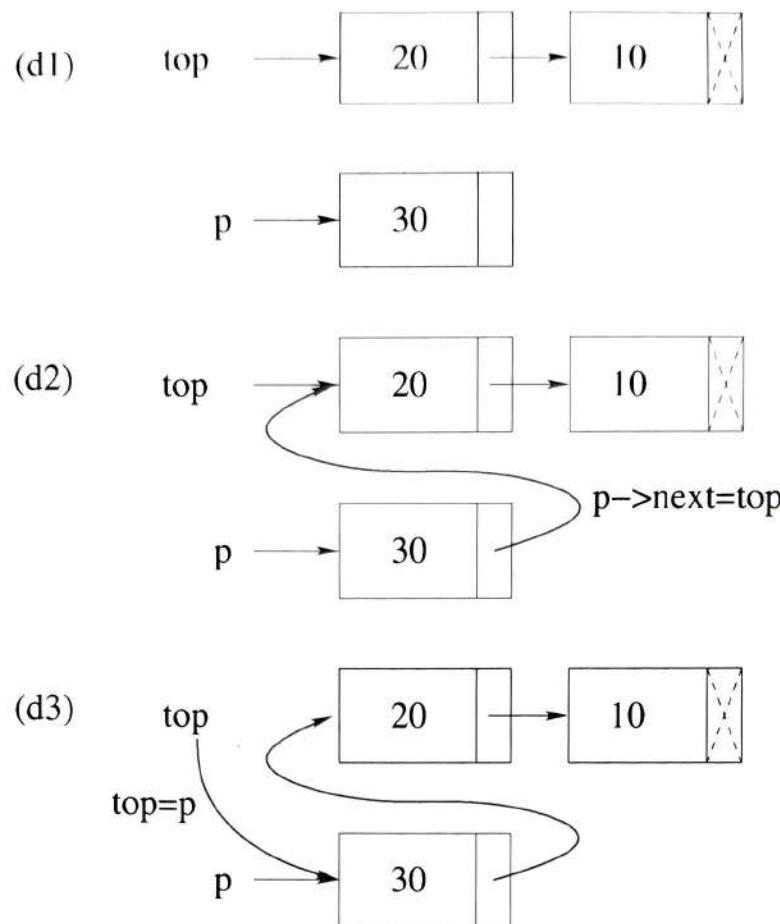
Figure 9.3, 9.4, and 9.5 illustrate all steps.



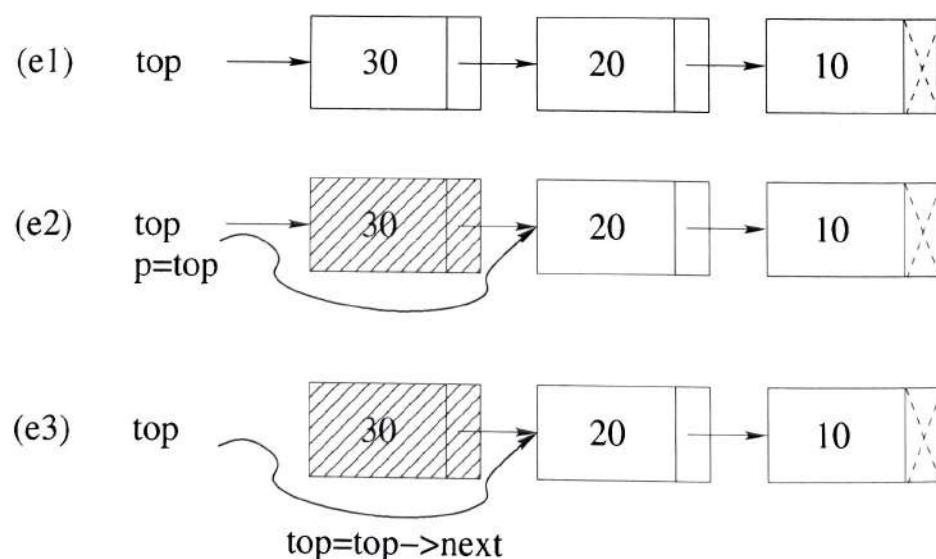
**Figure 9.3** Push operation on a stack, LIFO structure. (a) Initial configuration. (b) Configuration after operation `push("10")`. (c) Configuration after operation `push("20")`.

### Push (insertion)

As represented in Figure 9.3(a) the stack may be supposed initially empty, i.e., the external pointer (named `top`) is equal to `NULL`. Each subsequent push operation is



**Figure 9.4** Detailed phases for operation push ("30") on the stack of Figure 9.3(c).



**Figure 9.5** Detailed phases for operations pop on the stack of Figure 9.4(d2).

performed as represented in Figure 9.4. After the new element `p` is dynamically allocated and all its field properly assigned with data values (Figure 9.4(d1)), each insertion can be implemented with the sequence of the following two steps:

`p->next = top;`

`top = p;`  
 operations represented in Figure 9.4(d2) and Figure 9.4(d3).  
**Pop (extraction)**

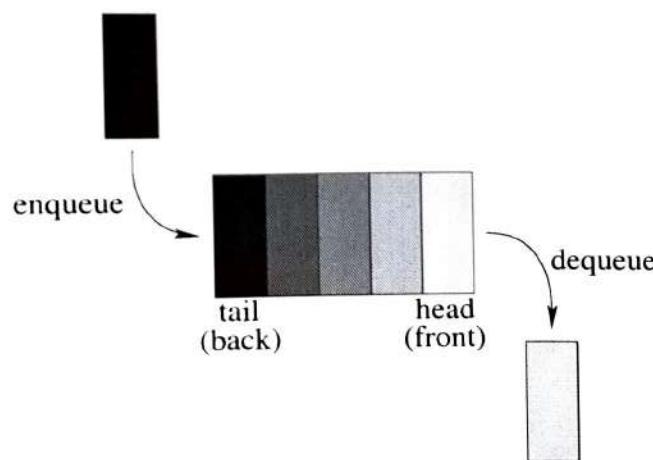
A pop can be performed only if the stack is not empty. In this case, as illustrated in Figure 9.5, it includes two main steps (e1) and (e2)):

```
if (top != NULL) {
    p = top;
    top = top->next;
    /* Use p element (p->value) */
    ...
} else {
    /* Nothing to pop */
    ...
}
```

## 9.2 A FIFO Queue

In computer science, a queue is a particular kind of abstract data type or collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position, known as enqueue, and removal of entities from the front terminal position, known as dequeue.

This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. Figure 9.6 represents a queue with enqueue and dequeue operations.

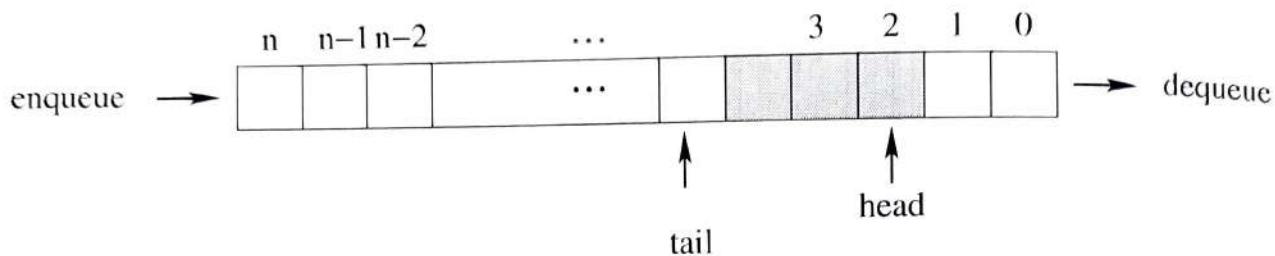


**Figure 9.6** Enqueue and dequeue operations on a FIFO queue. Lighter elements have been inserted before darker ones. The extraction order will be the opposite (from darker to lighter objects). Insertions and extractions are performed on two different extremes of the data structure.

There are several possible implementation of a stack. We will analyze a few of them in the following sub-sections.

### 9.2.1 Array Implementation

Figure 9.7 reports a diagrammatical representation of a queue where object enqueue on the queue tail and dequeue from the queue head.



**Figure 9.7** Linear Queue Data Structure.

Figure 9.8 reports a diagrammatical representation of the enqueue and dequeue operation on a linear queue:

- ▷ The queue is initially empty and the tail and head indexes refer to element 0 (Figure 9.8(a)).
- ▷ Each enqueue is performed into the tail of the structure:

```
array[tail] = value;
tail++;
```

(Figure 9.8(b) and (c)).

- ▷ Each dequeue is performed from the head of the structure:

```
value = array[head];
head++;
```

(Figure 9.8(d)).

Sooner or later the queue becomes full. Obviously, left shifting all stored elements has a linear cost in the number of elements itself, and has to be avoided.

The solution is to see the queue as a circular data structure, where the two end meet as represented in Figure 9.9. Each `tail++` and `head++` operations have to be performed modulo the queue size:

```
array[tail] = value;
tail = (tail+1) % size;
```

to perform each enqueue, and

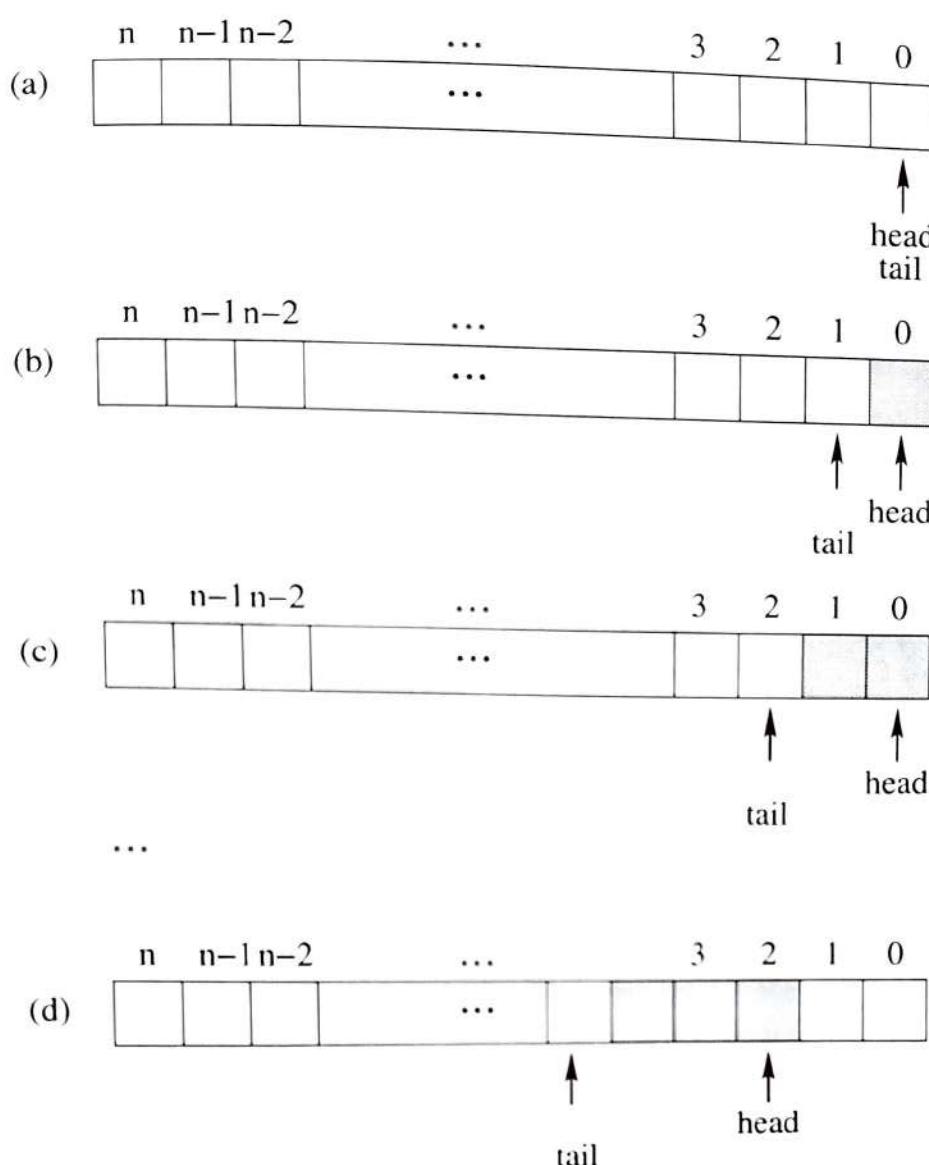
```
value = array[head];
head = (head+1) % size;
```

for each dequeue.

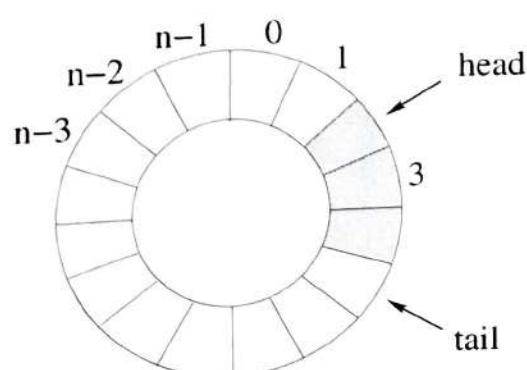
With a circular queue the programmer has to solve one last problem: How to understand whether the queue is full or empty? In fact, if the value of `tail` keeps increasing, `tail` can reach `head` from “behind”, and when `tail==head` the queue is full. This problem can be solved using, at least, two different approaches.

In the first approach, the number of elements in the queue is stored in a variable (named `n`). When the queue is created and every time the queue is empty, the value of `n` is equal to 0. `n` is then incremented for each enqueue operation, and decremented for each dequeue operation. When `n == size` the queue is full.

In the second strategy, the current number of element in the queue is not explicitly stored in the queue data structure. In this scenario it is possible to understand whether



**Figure 9.8** Linear Queue: (a) Empty queue, (b) and (c) insertions (enqueues), (d) dequeue.



**Figure 9.9** Circular Queue.

the queue is empty by checking whether  $\text{tail} == \text{head}$ . As a consequence, this condition has to be avoided when the queue is full. In other words, at least one element of the queue must always remain empty, and the full condition may be verified by checking whether  $\text{tail} + 1 \% \text{size} == \text{head}$  modulo the queue size, i.e.,  $\text{tail} + 1 \% \text{size} == \text{head}$

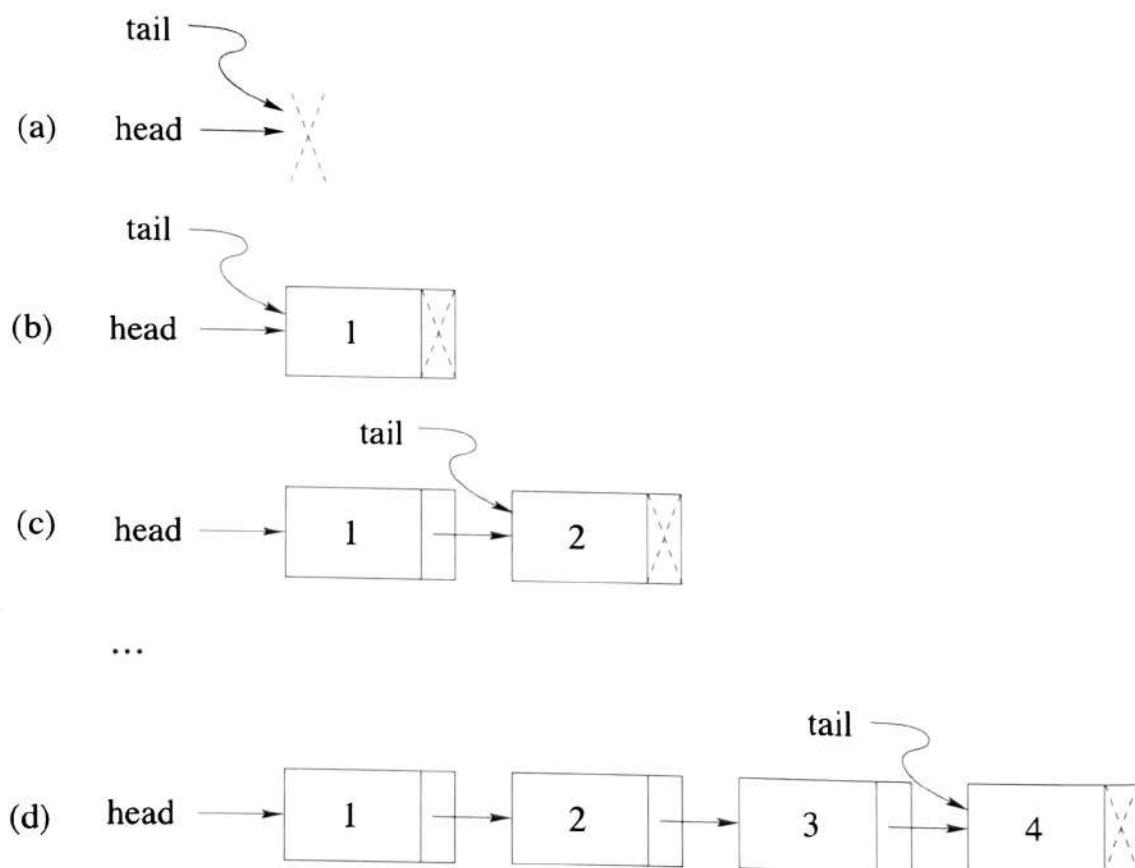
head.

### 9.2.2 List Implementation

As in the stack case, the list implementation can use the C data structure introduced in Section 4.2.1. Insertions (enqueues) and extraction (dequeues) have to be performed on the two different extremes of the data structure. As a consequence, we need two pointers to identify those two extremes. At the beginning, the queue is suppose to be empty. Then those entry points have to be initialized as:

```
head = NULL;
tail = NULL;
```

There are at least two possible data structures. The first one is a simple linear list, in which enqueue are performed on the *tail* of the queue and dequeue from the *head*. Figure 9.10 show a diagrammatical representation of this case. Notice that it would be difficult to make extractions from the tail side (object enqueues on the tail) as that would imply a complete visit of the list.



**Figure 9.10** Simple list implementation of a queue.

#### Enqueue (insertion)

As for the stack structure, each enqueue may first create a new element *p* and properly assign all data to it. Then, if the queue is empty both *head* and *tail* must point to *p* (lines 2 and 6), otherwise *p* is referenced by *p*->*next* (line 4) and *p* adjusted to the new tail element *p* (line 6):

```

if (head == NULL) {
    head = p;
} else {
    tail->next = p;
}
tail = p;

```

### Dequeue (extraction)

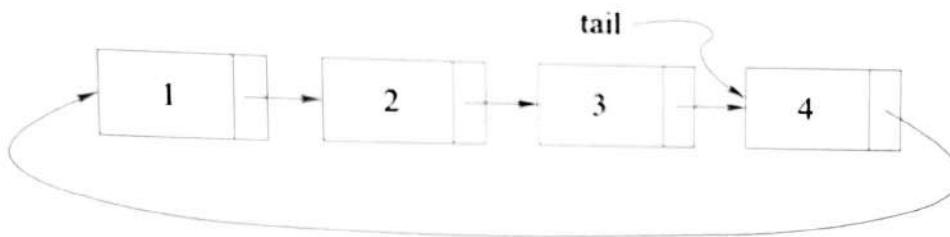
A dequeue operation can be implemented using the following C code:

```

p = head;
... get data fields from p-> ...
head = p->next;
if (head == NULL) {
    tail = NULL;
}

```

When the queue becomes empty both tail and head are made to refer to NULL. As it can be noticed, the pointer of the tail element (`tail->next`) is always equal to NULL. For that reason the structure can be transformed into a circular list, where the head pointer is substituted by `tail->next`. This implies keeping only the tail pointer to access the structure, as represented in Figure 9.11. In this scenario



**Figure 9.11** Circular list implementation of a queue.

Figures 9.12, 9.13, and 9.14 show how to operate subsequent enqueue.

Figures 9.12, 9.13 are realized by the following C code.

```

if (tail==NULL) {
    tail = p;
    tail->next = tail;
} else {
    p->next = tail->next;
    tail->next = p;
    tail = p;
}

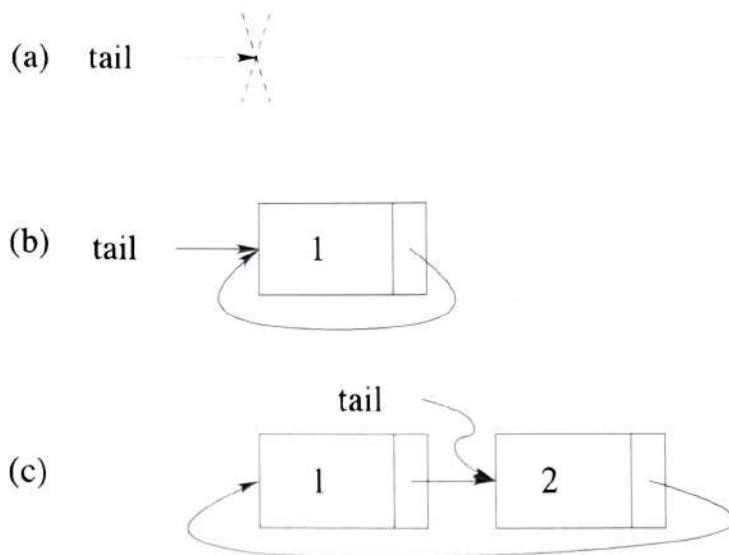
```

Figure 9.14 shows how to operate a dequeue operation. The C code is the following:

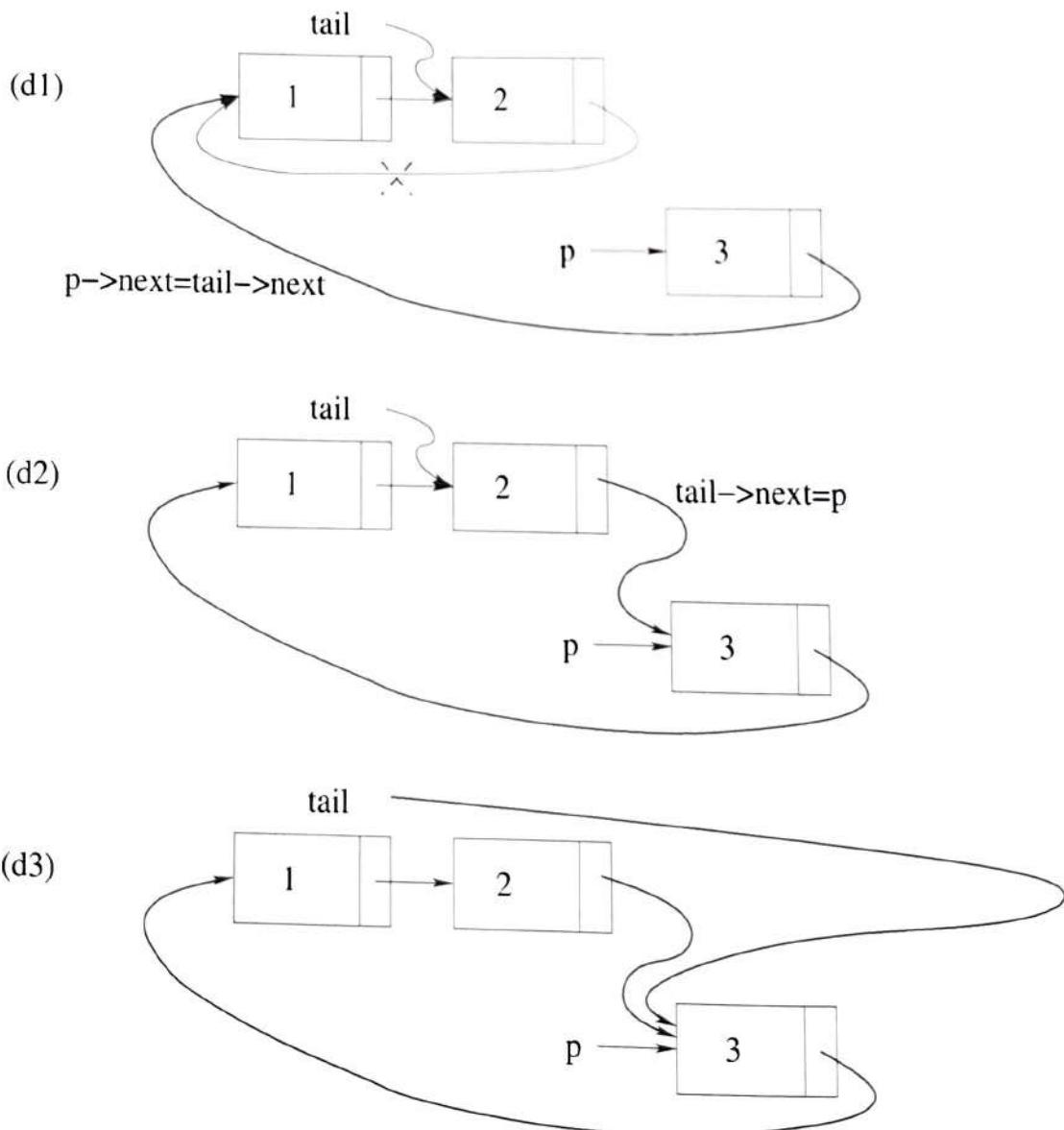
```

if (tail != NULL) {
    if (tail == tail->next) {
        p = tail;
        tail = NULL;
    } else {
        p = tail->next;
        tail->next = p->next;
    }
} else {

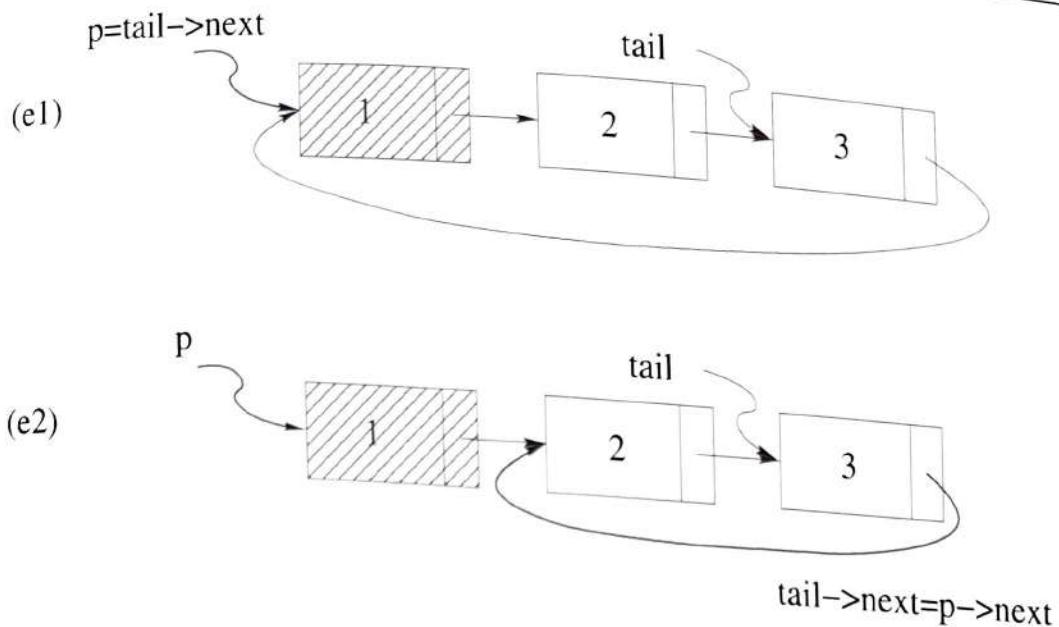
```



**Figure 9.12** Enqueue operation on a circular queue.



**Figure 9.13** Different phases of enqueue (3).



**Figure 9.14** Different phases of a dequeue.

```
/* No element to extract */
p = NULL;
}
```

Notice that, the extracted element is referenced by the pointer *p* at the end of the dequeue operation.

## 9.3 Functions as Function Arguments

In C, pointers to functions can be passed as arguments to a function and returned (with the `return` statement) by the function itself. In this section, we briefly describe how this facility works.

**Example 9.1** Suppose we want to carry out a specific computation with a variety of different functions. For example, we may want to compute

$$\sum_{i=0}^n f(x)$$

where the procedure performing the sum is unique, whereas function *f* may vary. Thus, the procedure performing the sum should receive *f* as a parameter and call this function within the loop computing the sum.

Given a C function *f* it is possible to think about it in the following ways:

- ▷ *f* is the pointer to the function.
- ▷ *\*f* is the function itself.
- ▷ *((\*f))(n)* is the call to the function.

Thus explicitly referring to *f* or to *(\*f)* is C is equivalent. To illustrates how this work in practice, let us solve the problem stated in Example 9.1.

**Example 9.2** The following program calls function sum to compute the sum indicated in Example 9.1. Function f1, f2, and f3 implement three different versions of function f. Each one of these compute a single term of the sum in a different way.

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

float f1 (float);
float f2 (float);
float f3 (float);
float sum (float f (float), float, int);

int main (int argc, char *argv[]) {
    int x = atof (argv[1]);
    int n = atoi (argv[2]);
    float s;

    s = sum (f1, x, n);
    fprintf (stdout, "x + x + ... = %f\n", s);
    s = sum (f2, x, n);
    fprintf (stdout, "x^2 + x^2 + ... = %f\n", s);
    s = sum (f3, x, n);
    fprintf (stdout, "x^3 + x^3 + ... = %f\n", s);

    return EXIT_SUCCESS;
}

float sum (float f (float x), float x, int n) {
    int i;
    float s;

    s = 0;
    for (i=0; i<n; i++) {
        s = s + f (x);
    }

    return s;
}

float f1 (float x) {
    return (x);
}

float f2 (float x) {
    return (x*x);
}

float f3 (float x) {
    return (x*x*x);
}
```

In this piece of code, we refer to functions using their name only, i.e., f. A different, but equivalent, approach would be using (\*f). In this case we just report the lines which have been modified from the previous piece of code.

```
...
float sum (float (*f) (float), float, int);
...
float sum (float (*f) (float x), float x, int n) {
```

```

int i;
float s;

s = 0;
for (i=0; i<n; i++) {
    s = s + (*f)(x);
}

return s;
}
...

```

Notice that in this case the notation `(*f)` (`float`) means that “`f` is a pointer to function that takes a single argument of type `float` and returns a `float`.” The parentheses are necessary because `( )` binds tighter than `*`. In contrast, consider the declaration `float *f` (`float`) declares `f` to be a function that takes an argument of type `float` and returns a pointer to `float`. More specifically, there are a number of equivalent ways to write a function prototype that has a function as a formal parameter. To illustrate this, let us write a list of equivalent function types

```

float sum (float f (float x), float x, int n);
float sum (float f (float), float, int);
float sum (float (*f) (float x), float x, int n);
float sum (float (*f) (float), float, int);
float sum (float (*) (float), float, int);

```

In the above list, the identifiers `x` and `n` are optional. If present, the compiler disregards them. In a similar fashion, the identifier `f` in the construct `(*f)` is optional. In C, a function can have multiple prototypes, provided that they are all equivalent. Thus, when we put the above list in the header file the program compiles and runs just as before.

## 9.4 Macro Definition

In Section 1.1.4 we introduced the use of the `define` construct to define constant values.

So far, we have considered only simple pre-processing directives. Typical examples we have seen so far are the following:

```

#define EOF -1
#define LENGTH 1000
#define PI 3.14159
#define STRING_LENGTH (80+2)

```

We now want to discuss how we can use the `#define` construct to write macro definition with parenthesis. With macro we mean a sort of *in-line functions*, i.e., functions that are expanded (by the pre-compiler) directly in the source code.

To this respect, the `#define` facility can be used as:

```
#define id(id,id,id,...) string
```

where there is no space between the first `id` and the open parenthesis, and zero or more identifiers can occur in the parameter list. The following is first example.

**Example 9.3** Given the following definition:

```
#define MY_SQUARE(x) ((x)*(x))
```

`MY_SQUARE` is the macro name, `x` is the macro parameter, and `((c)*(x))` is the macro body. Whenever the pre-processor finds `MY_SQUARE(value)` in the text, it expands it with

((value) \* (value)). For example, MY\_SQUARE(23) expands to ((23) \* (23)), whereas MY\_SQUARE(x+y) expands to ((x+y) \* (x+y)). Notice, that in this last case the use of parenthesis becomes clear, as it protects the programmer against the macro expanding an expression so that it lead to an unanticipated order of evaluation.

The following example introduces a macro we will use in several implementations.

**Example 9.4** The following define facility:

```
#define util_check_m(expr, msg) \
    if ( !(expr) ) { \
        fprintf(stderr, "Error: %s\n", msg); \
        exit(EXIT_FAILURE); \
    }
```

defines the macro util\_check\_m which perform a check on an expression exp and if this is FALSE it prints-out (on standard error) the error message msg. Notice the use of the character to go on the next line during the definition of the macro body.

## 9.5 A Utility Library

### Problem definition

In this section we will show how to build a “utility library”, i.e., a set of functions that can be used by client programs to perform “basic” operations. To do that we will organized the application following the rules discussed in Chapter 8.

More specifically the library will make available functions to open a file, allocate and free an array and a 2D matrix, and duplicate a string. All those functions will have to perform proper correctness checks for common error (e.g., file not opened, allocation failure, etc.) issuing error messages in case of problems. Moreover, notice that those functions can be seen as “wrappers” for standard C functions as fopen, malloc, etc.

### Specification

Build the previously described library and a client application able to:

- ▷ Read an input file with an undefined number of lines. Suppose each line is at most 80 character long.
- ▷ Store the entire file in a dynamically allocated 2D matrix. The size of this matrix can be obtained by reading the input file twice.
- ▷ Store the matrix in an output file such that lines are reversed and each line is stored from left-to-right, i.e., the last line becomes the first one, and last characters become first ones.

**Example 9.5** Let us suppose the following one

This is an file example.

In this file there are only 3 rows.

The last row is this one.

is the input file. The application must store it in a dynamic matrix and write the following file:

```
.eno siht si wor tsal ehT
.swor 3 ylno era ereht elif siht nI
.elpmaxe elif na si sihT
```

Notice that the input and output file names have to be read from the command line.

## Solution

### File client.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "util.h"
5
6 #define MAX 82 /* 80 + '\n' + '\0' */
7
8 /* static function prototypes */
9 static char **file_read(char *filename, int *num_ptr);
10 static void file_write(char *filename, char **array, int num);
11
12 /*
13 * main program
14 */
15 int main (int argc, char *argv[]) {
16     char **array;
17     int num;
18
19     util_check_m(argc>=3, "missing parameter.");
20
21     array = file_read(argv[1], &num);
22     file_write(argv[2], array, num);
23     util_array_dispose((void **)array, num, free);
24
25     return EXIT_SUCCESS;
26 }
27
28 /*
29 * load the input file contents
30 */
31 static char **file_read (char *filename, int *num_ptr) {
32     char **array, line[MAX];
33     int i, n=0;
34     FILE *fp;
35
36     fp = util_fopen(filename, "r");
37
38     /* count the number of rows */
39     while (fgets(line, MAX, fp) != NULL) {
40         n++;
41     }
42     fclose(fp);
43
44     /* store the file contents */
45     array = (char **)util_malloc(n * sizeof(char *));
46     fp = util_fopen(filename, "r");
47     for (i=0; i<n; i++) {
48         fgets(line, MAX, fp);
49         array[i] = util_strdup(line);
50     }
51     fclose(fp);
52 }
```

```

53     *num_ptr = n;
54     return array;
55 }
56
57 /*
58 * write the output file
59 */
60 static void file_write (char *filename, char **array, int num) {
61     FILE *fp;
62     int i, j;
63
64     fp = util_fopen(filename, "w");
65     for (i=num-1; i>=0; i--) {
66         for (j=strlen(array[i])-2; j>=0; j--) {
67             fprintf(fp, "%c", array[i][j]);
68         }
69         fprintf(fp, "\n");
70     }
71     fclose(fp);
72 }
```

### File util.h

```

1 #ifndef _UTIL
2 #define _UTIL
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
7
8 /* macro definition */
9 #define util_check_m(expr, msg) \
10    if ( !(expr) ) { \
11        fprintf(stderr, "Error: %s\n", msg); \
12        exit(EXIT_FAILURE); \
13    }
14
15 /* extern function prototypes */
16 extern FILE *util_fopen(char *name, char *mode);
17 extern void *util_malloc(unsigned int size);
18 extern void *util_calloc(unsigned int num, unsigned int size);
19 extern char *util_strdup(char *src);
20 extern void util_array_dispose(void **ptr, unsigned int n, void (*quit)(void *));
21 extern void **util_matrix_alloc(unsigned int n, unsigned int m, unsigned int size);
22 extern void util_matrix_dispose(void ***ptr, unsigned int n, unsigned int m,
23                                void (*quit)(void *));
24 #endif
```

### File util.c

```

1 #include "util.h"
2
3 /*
4 * fopen (with check) utility function
5 */
6 FILE *util_fopen (char *name, char *mode) {
7     FILE *fp=fopen(name, mode);
8     util_check_m(fp!=NULL, "could not open file!");
9     return fp;
10 }
11
```

```
12  /* malloc (with check) utility function
13  */
14  void *util_malloc (unsigned int size) {
15      void *ptr=malloc(size);
16      util_check_m(ptr!=NULL, "memory allocation failed!");
17      return ptr;
18  }
19
20  /*
21  * calloc (with check) utility function
22  */
23  void *util_calloc (unsigned int num, unsigned int size) {
24      void *ptr=calloc(num, size);
25      util_check_m(ptr!=NULL, "memory allocation failed!");
26      return ptr;
27  }
28
29  /*
30  * strdup (with check) utility function
31  */
32  char *util_strdup (char *src) {
33      char *dst=strdup(src);
34      util_check_m(dst!=NULL, "memory allocation failed");
35      return dst;
36  }
37
38  /*
39  * array de-allocation utility function
40  */
41  void util_array_dispose (void **ptr, unsigned int n, void (*quit)(void *)) {
42      int i;
43
44      if (quit != NULL) {
45          for (i=0; i<n; i++) {
46              quit(ptr[i]);
47          }
48      }
49      free(ptr);
50  }
51
52  /*
53  * matrix allocation utility function
54  */
55  void **util_matrix_alloc (unsigned int n, unsigned int m, unsigned int size) {
56      void **ptr;
57      int i;
58
59      ptr = (void **)util_malloc(n*sizeof(void *));
60      for (i=0; i<n; i++) {
61          ptr[i] = util_calloc(m, size);
62      }
63      return ptr;
64  }
65
66  /*
67  * matrix de-allocation utility function
68  */
69  void util_matrix_dispose (
```

```

71     void ***ptr, unsigned int n, unsigned int m, void (*quit)(void *)
72  ) {
73     int i, j;
74
75     for (i=0; i<n; i++) {
76       if (quit != NULL) {
77         for (j=0; j<m; j++) {
78           quit(ptr[i][j]);
79         }
80       }
81       free(ptr[i]);
82     }
83     free(ptr);
84
85   return;
86 }
```

## 9.6 A First Class ADT Library LIFO Stack

### Specifications

Implement an ADT stack library able to manipulate different type of data items within the stack, such as integers, strings and more complex C structures.

Realize two version of the queue based on two different underlying data structure: The first one using a dynamic array, and the second adopting a dynamic list.

### Data Structure Definition

The source files within the package can be seen as belonging to one of three different logical layers. Those layers, from bottom to top, are the following: Data item, stack, and client.

As far as the bottom layer (the data item) is concerned, as requested by the specifications there are three different type of data, each one including two source files:

- ▷ Integer types, implemented within files `item-int.h`, and `item-int.c`.
- ▷ String types, implemented within files `item-string.h`, and `item-string.c`.
- ▷ C structure types, implemented with `item-struct.h`, and `item-struct.c`.

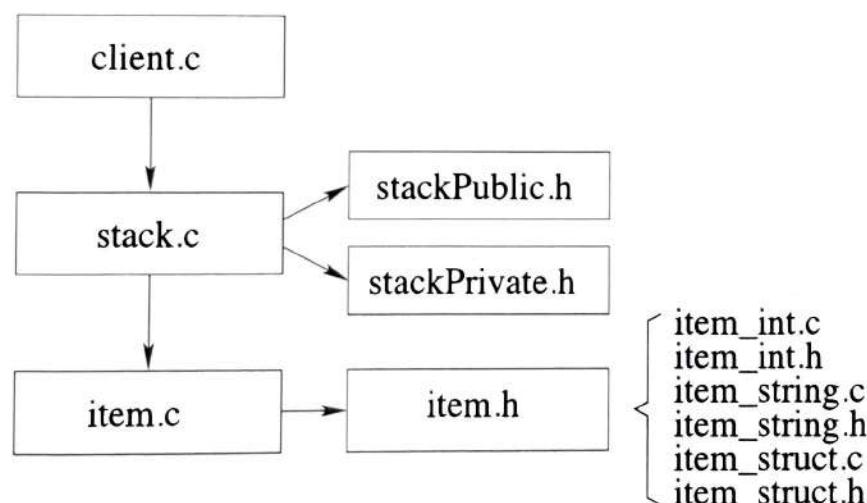
When one of those file set is copied into files `item.h` and `item.c` the package can be compiled with that corresponding data type in place. All other files (i.e., the ones within the stack and the client layers) remain untouched as they are not aware of which type of data structure is actually stored in the stack. Moreover, notice that within this layer there is only one header file (i.e., there is no a public and a private version), as all objects and functions are essentially publicly visible.

As far as the intermediate layer is concerned, the stack is implemented using a complete ADT. Exported (public) functions and types are defined into the `stackPublic.h` file, whereas `stackPrivate.h` includes all internal (i.e., private) types and definition. As previously described, public objects are the ones visible from the client, whereas private objects are the ones visible only within the stack layer. For example, the internal stack structure `struct stack` is defined in the private file, and it is publicly visible only through the definition `typedef struct stack stack_t` included into the public file. Notice that unlike other programming style (see for example [5, 6]) we make the structure visible directly and not through its pointer. This does not change its visibility status (i.e., according to [5] the stack is still a ‘first type ADT’) but we

do not hide the pointer nature of the object. In other words, the client will be able to define only pointers to the `stack_t` structure, not object. In fact, the storage size will remain unknown at the client level. Indeed, we deem a questionable habit to hide the pointer nature of a data types adopting a C programming style, albeit this strategy may be common to other languages and programming environment (see the Java language or the Windows API programming environment). Another major difference from similar programming environment (see for example [5]) is that data item type remains unknown at the stack layer. To this regards, notice that the structure `stack` stores data items into a double star object, i.e., `void **array`. This implies that at the stack level data items are seen as `void *` objects, not as item objects, and those `void *` are managed only through an array of pointers (namely `array`). This sort of implementation can be seen a further form of data hiding, as the exact same stack library implementation can be used with data types (the bottom layer) other than items.

As far as the higher layer is concerned, the client is mainly in charge of orchestrating the stack and the item manipulation. Anyway, all data and stack types and functions are only visible through the corresponding public interfaces. As a final comment, notice that in more complex situations, the client will have to pass to the intermediate layer (the stack in this case) proper pointers to all those functions needed by the intermediate layer to manipulate the adopted data items (such a function needed to compare or copy data items). We will analyze this situation in more complex libraries.

Figure 9.15 graphically shows the application logical structure.



**Figure 9.15** Logical structure of the stack application.

As a final remark notice that this application also uses the utility library defined in Exercise 9.5 (file `util.h` and `util.c`).

### Solution 1

This first solution adopts a dynamic array as underlying data structure to implement the stack library.

For the sake of simplicity, the size of the dynamic array is selected using a `define` construct, even if a user selection (with a dynamic memory allocation) would have been more appropriate.

**File client.c**

```
1 #include <stdio.h>
2 #include <ctype.h>
3 #include "stackPublic.h"
4 #include "item.h"
5
6 #define SIZE 50
7 #define MAX 20
8
9 int main (void) {
10     char command[MAX];
11     stack_t *sp=NULL;
12     item_t data;
13     int stop=0;
14
15     do {
16         fprintf(stdout, "\nAvailable commands:\n");
17         fprintf(stdout, " (N)eW\n (I)nsert\n");
18         fprintf(stdout, " (E)xtract\n (D)isplay\n");
19         fprintf(stdout, " (S)top\nChoice: ");
20         scanf("%s", command);
21
22         switch (tolower(command[0])) {
23             case 'n':
24                 stack_dispose(sp, item_dispose);
25                 sp = stack_init(SIZE);
26                 fprintf(stdout, "New stack correctly created.\n");
27                 break;
28             case 'i':
29                 fprintf(stdout, "Input data: ");
30                 item_read(stdin, (void **)&data);
31                 if (!stack_push(sp, (void *)data)) {
32                     fprintf(stderr, "Error in push!\n");
33                 } else {
34                     fprintf(stdout, "Data correctly inserted.\n");
35                 }
36                 break;
37             case 'e':
38                 if (!stack_pop(sp, (void **)&data)) {
39                     fprintf(stderr, "Error in pop!\n");
40                 } else {
41                     fprintf(stdout, "Extracted data: ");
42                     item_print(stdout, (void *)data);
43                     fprintf(stdout, "\n");
44                 }
45                 break;
46             case 'd':
47                 stack_print(stdout, sp, item_print);
48                 break;
49             case 's':
50                 stop = 1;
51                 break;
52             default:
53                 fprintf(stderr, "Unknown command!\n");
54         }
55     } while (!stop);
56
57     stack_dispose(sp, item_dispose);
```

```

58     return 0;
59 }
60 }
```

### File stackPublic.h

```

1 #ifndef _STACK_PUBLIC
2 #define _STACK_PUBLIC
3
4 #include <stdio.h>
5
6 /* macro definition */
7 #define stack_empty_m(sp) (stack_count(sp) == 0)
8
9 /* type declarations */
10 typedef struct stack stack_t;
11
12 /* extern function prototypes */
13 extern stack_t *stack_init(int size);
14 extern int stack_count(stack_t *sp);
15 extern int stack_push(stack_t *sp, void *data);
16 extern int stack_pop(stack_t *sp, void **data_ptr);
17 extern void stack_print(FILE *fp, stack_t *sp, void (*print)(FILE *, void *));
18 extern void stack_dispose(stack_t *sp, void (*quit)(void *));
19
20#endif
```

### File stackPrivate.h

```

1 #ifndef _STACK_PRIVATE
2 #define _STACK_PRIVATE
3
4 #include <stdio.h>
5 #include "stackPublic.h"
6 #include "util.h"
7
8 /* structure declarations */
9 struct stack {
10     void **array;
11     int index;
12     int size;
13 };
14
15#endif
```

### File stack.c

```

1 #include "stackPrivate.h"
2
3 /*
4  *  create a new empty stack
5  */
6 stack_t *stack_init(int size)
7 {
8     stack_t *sp;
9
10    sp = (stack_t *)util_malloc(sizeof(stack_t));
11    sp->size = size;
12    sp->index = 0;
13    sp->array = (void **)util_malloc(size*sizeof(void *));
14    return sp;
```

```
15 }
16
17 /* 
18  *   return the number of elements stored in the stack
19 */
20 int stack_count(stack_t *sp)
21 {
22     return (sp!=NULL) ? sp->index : 0;
23 }
24
25 /*
26  *   store a new value in the stack (LIFO policy)
27 */
28 int stack_push(stack_t *sp, void *data)
29 {
30     if (sp==NULL || sp->index>=sp->size) {
31         return 0;
32     }
33
34     sp->array[sp->index++] = data;
35     return 1;
36 }
37
38 /*
39  *   extract a value from the stack (LIFO policy)
40 */
41 int stack_pop(stack_t *sp, void **data_ptr)
42 {
43     if (sp==NULL || sp->index<=0) {
44         return 0;
45     }
46
47     *data_ptr = sp->array[--sp->index];
48     return 1;
49 }
50
51 /*
52  *   print all the stack elements (LIFO policy)
53 */
54 void stack_print(FILE *fp, stack_t *sp, void (*print)(FILE *, void *))
55 {
56     int i;
57
58     if (sp != NULL) {
59         for (i=sp->index-1; i>=0; i--) {
60             print(fp, sp->array[i]);
61             fprintf(fp, "\n");
62         }
63     }
64 }
65
66 /*
67  *   deallocate all the memory associated to the stack
68 */
69 void stack_dispose(stack_t *sp, void (*quit)(void *))
70 {
71     int i;
72
73     if (sp != NULL) {
```

```

74     if (quit != NULL) {
75         for (i=0; i<sp->index; i++) {
76             quit(sp->array[i]);
77         }
78     free(sp->array);
79     free(sp);
80 }
81 }
82 }
```

**File item-int.h**

```

1 #ifndef _ITEM
2 #define _ITEM
3
4 #include <stdio.h>
5 #include "util.h"
6
7 /* type declarations */
8 typedef int *item_t;
9
10 /* extern function prototypes */
11 extern int item_read(FILE *fp, void **ptr);
12 extern void item_print(FILE *fp, void *ptr);
13 extern int item_compare(void *data1, void *data2);
14 extern void item_dispose(void *ptr);
15
16 #endif
```

**File item-int.c**

```

1 #include "item.h"
2
3 #define MAX 100
4
5 /*
6  * read an item from file
7  */
8 int item_read (FILE *fp, void **data_ptr) {
9     int *p;
10
11     p = (int *)util_malloc(sizeof(int));
12     if (fscanf(fp, "%d", p) == EOF) {
13         return EOF;
14     }
15     *data_ptr = p;
16
17     return 1;
18 }
19
20 /*
21  * print an item on file
22  */
23 void item_print (FILE *fp, void *ptr) {
24     item_t data = (item_t)ptr;
25     fprintf(fp, "%d ", *data);
26 }
27
28 /*
29  * compare two items
30  */

```

```

31 int item_compare (void *ptr1, void *ptr2) {
32     item_t data1 = (item_t)ptr1;
33     item_t data2 = (item_t)ptr2;
34
35     return (*data1)-(*data2);
36 }
37
38 /*
39 * free an item
40 */
41 void item_dispose (void *ptr) {
42     item_t data = (item_t)ptr;
43     free(data);
44     return;
45 }
```

### **File item-string.h**

```

1 #ifndef _ITEM
2 #define _ITEM
3
4 #include <stdio.h>
5 #include "util.h"
6
7 /* type declarations */
8 typedef char *item_t;
9
10 /* extern function prototypes */
11 extern int item_read(FILE *fp, void **ptr);
12 extern void item_print(FILE *fp, void *ptr);
13 extern int item_compare(void *data1, void *data2);
14 extern void item_dispose(void *ptr);
15
16 #endif
```

### **File item-string.c**

```

1 #include "item.h"
2
3 #define MAX 100
4
5 /*
6 * read an item from file
7 */
8 int item_read(FILE *fp, void **data_ptr)
9 {
10     char word[MAX];
11
12     if (fscanf(fp, "%s", word) == EOF) {
13         return EOF;
14     }
15
16     *data_ptr = util_strdup(word);
17
18     return 1;
19 }
20
21 /*
22 * print an item on file
23 */
24 void item_print(FILE *fp, void *ptr)
```

```

25  {
26      item_t data = (item_t)ptr;
27      fprintf(fp, "%s", data);
28  }
29
30  /*
31   * compare two items
32   */
33  int item_compare(void *ptr1, void *ptr2)
34  {
35      item_t data1 = (item_t)ptr1;
36      item_t data2 = (item_t)ptr2;
37
38      return strcmp(data1, data2);
39  }
40
41  /*
42   * free an item
43   */
44  void item_dispose(void *ptr)
45  {
46      free(ptr);
47  }

```

### File item-struct.h

```

1 #ifndef _ITEM
2 #define _ITEM
3
4 #include <stdio.h>
5 #include "util.h"
6
7 /* type declarations */
8 typedef struct item *item_t;
9
10 /* extern function prototypes */
11 extern int item_read(FILE *fp, void **ptr);
12 extern void item_print(FILE *fp, void *ptr);
13 extern int item_compare(void *data1, void *data2);
14 extern void item_dispose(void *ptr);
15
16 #endif

```

### File item-struct.c

```

1 #include "item.h"
2
3 #define MAX 100
4
5 /* structure declarations */
6 struct item {
7     int id;
8     char *name;
9     int exams;
10    float avg;
11 };
12
13 /*
14  * read an item from file
15  */
16 int item_read (FILE *fp, void **data_ptr) {

```

```

17 int result, id, exams;
18 char name[MAX];
19 float avg;
20 item_t data;
21
22 result = fscanf(fp, "%d %s %d %f", &id, name, &exams, &avg);
23 if (result != EOF) {
24     data = (struct item *)util_malloc(sizeof(struct item));
25     data->id = id;
26     data->name = util_strdup(name);
27     data->exams = exams;
28     data->avg = avg;
29     *data_ptr = data;
30 }
31
32 return result;
33 }
34
35 /*
36 * print an item on file
37 */
38 void item_print (FILE *fp, void *ptr) {
39     item_t data = (item_t)ptr;
40     fprintf(fp, "%7d %-20s %2d %.2f", data->id, data->name, data->exams, data->avg);
41 }
42
43 /*
44 * compare two items
45 */
46 int item_compare (void *ptr1, void *ptr2) {
47     item_t data1 = (item_t)ptr1;
48     item_t data2 = (item_t)ptr2;
49
50     return data1->id - data2->id;
51 }
52
53 /*
54 * free an item
55 */
56 void item_dispose (void *ptr) {
57     item_t data = (item_t)ptr;
58     free(data->name);
59     free(data);
60 }

```

## Solution 2

This second solution adopts a dynamic linked list as underlying data structure. Files which do not differ from the previous implementation (such as the ones belonging to the client layer, i.e., `client.c`, and to the item layer, i.e., all data item files) are not reported. Notice that also the public stack interface (`stackPublic.h`) remains unchanged and it is not reported as well.

### File `stackPrivate.h`

```

1 #ifndef _STACK_PRIVATE
2 #define _STACK_PRIVATE
3
4 #include <stdio.h>

```

```
5 #include "stackPublic.h"
6 #include "util.h"
7 /* structure declarations */
8 typedef struct node {
9     void *data;
10    struct node *next;
11 } node_t;
12
13 struct stack {
14     node_t *head;
15     int num;
16 };
17
18 #endif
```

### File stack.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "stackPrivate.h"
4 #include "util.h"
5
6 /*
7  * create a new empty stack
8  */
9 stack_t *stack_init(int size) {
10    stack_t *sp;
11
12    sp = (stack_t *)util_malloc(sizeof(stack_t));
13    sp->head = NULL;
14    sp->num = 0;
15
16    return sp;
17 }
18
19 /*
20  * return the number of elements stored in the stack
21  */
22 int stack_count(stack_t *sp) {
23    return (sp!=NULL) ? sp->num : 0;
24 }
25
26 /*
27  * store a new value in the stack (LIFO policy)
28  */
29 int stack_push(stack_t *sp, void *data) {
30    node_t *node;
31
32    if (sp == NULL) {
33        return 0;
34    }
35
36    node = (node_t *)util_malloc(sizeof(node_t));
37    node->data = data;
38    node->next = sp->head;
39    sp->head = node;
40    sp->num++;
41
42    return 1;
43 }
```

```
43  }
44
45 /* 
46  * extract a value from the stack (LIFO policy)
47 */
48 int stack_pop(stack_t *sp, void **data_ptr) {
49     node_t *node;
50
51     if (sp==NULL || sp->head==NULL) {
52         return 0;
53     }
54
55     node = sp->head;
56     *data_ptr = node->data;
57     sp->head = node->next;
58     free(node);
59     sp->num--;
60
61     return 1;
62 }
63
64 /*
65  * print all the stack elements (LIFO policy)
66 */
67 void stack_print (FILE *fp, stack_t *sp, void (*print)(FILE *, void *)) {
68     node_t *node;
69
70     if (sp != NULL) {
71         node = sp->head;
72         while (node != NULL) {
73             print(fp, node->data);
74             node = node->next;
75             fprintf(fp, "\n");
76         }
77     }
78 }
79
80 /*
81  * deallocate all the memory associated to the stack
82 */
83 void stack_dispose (stack_t *sp, void (*quit)(void *)) {
84     node_t *node;
85
86     if (sp != NULL) {
87         while (sp->head != NULL) {
88             node = sp->head;
89             sp->head = node->next;
90             if (quit != NULL) {
91                 quit(node->data);
92             }
93             free(node);
94         }
95         free(sp);
96     }
97 }
```

## 9.7 A First Class ADT Library FIFO Queue

### Specifications

Write a library that implement a FIFO queue. As for the stack case (see Exercise 9.6) implement two versions of the queue: The first one on a dynamic array, and the second one on a dynamic list. Moreover, as for the stack case, design the library such that it can work on integer, string, and structure items.

### Solution 1

This first solution uses a dynamic array (seen as a circular array) as data support. As for the stack case, the array is defined as an array of pointers to make the queue library independent from the item type. The client allocates the queue of size equal to the constant value SIZE, and it then orchestrate a main menu in which the user can perform basic operations on the queue, such as enqueue (queue\_put) and dequeue (queue\_get). The empty and full conditions are checked thanks to the field num which stores the number of element actually stored in the queue. The version without this counter can be easily derived from what has been described in the introduction of this chapter.

As a final remark notice that this application also uses the libraries defined in Exercises 9.5 and 9.6 (file util.\* and item.\*).

### File client.c

```

1 #include <stdio.h>
2 #include <ctype.h>
3 #include "queuePublic.h"
4 #include "item.h"
5
6 #define SIZE 50
7 #define MAX 20
8
9 int main (void) {
10     char command[MAX];
11     queue_t *qp=NULL;
12     item_t data;
13     int stop=0;
14
15     do {
16         fprintf(stdout, "\nAvailable commands:\n");
17         fprintf(stdout, " (N)ew\n (I)nsert\n");
18         fprintf(stdout, " (E)xtract\n (D)isplay\n");
19         fprintf(stdout, " (S)top\nChoice: ");
20         scanf("%s", command);
21
22         switch (tolower(command[0])) {
23             case 'n':
24                 queue_dispose(qp, item_dispose);
25                 qp = queue_init(SIZE);
26                 fprintf(stdout, "New queue correctly created.\n");
27                 break;
28             case 'i':
29                 fprintf(stdout, "Input data: ");
30                 item_read(stdin, (void **)&data);
31                 if (!queue_put(qp, (void *)data)) {

```

```

32         fprintf(stdout, "Error in put!\n");
33     } else {
34         fprintf(stdout, "Data correctly inserted.\n");
35     }
36     break;
37 case 'e':
38     if (!queue_get(qp, (void **)&data)) {
39         fprintf(stdout, "Error in get!\n");
40     } else {
41         fprintf(stdout, "Extracted data: ");
42         item_print(stdout, (void *)data);
43         fprintf(stdout, "\n");
44     }
45     break;
46 case 'd':
47     queue_print(stdout, qp, item_print);
48     break;
49 case 's':
50     stop = 1;
51     break;
52 default:
53     fprintf(stderr, "Unknown command!\n");
54 }
55 } while (!stop);

56 queue_dispose(qp, item_dispose);
57
58 return 0;
59 }
```

### **File queuePublic.h**

```

1 #ifndef _QUEUE_PUBLIC
2 #define _QUEUE_PUBLIC
3
4 #include <stdio.h>
5
6 /* macro definition */
7 #define queue_empty_m(qp) (queue_count(qp) == 0)
8
9 /* type declarations */
10 typedef struct queue queue_t;
11
12 /* extern function prototypes */
13 extern queue_t *queue_init(int size);
14 extern int queue_count(queue_t *sp);
15 extern int queue_put(queue_t *sp, void *data);
16 extern int queue_get(queue_t *sp, void **data_ptr);
17 extern void queue_print(FILE *fp, queue_t *sp, void (*print)(FILE *, void *));
18 extern void queue_dispose(queue_t *sp, void (*quit)(void *));
19
20 #endif
```

### **File queuePrivate.h**

```

1 #ifndef _QUEUE_PRIVATE
2 #define _QUEUE_PRIVATE
3
4 #include "util.h"
5 #include "queuePublic.h"
6
```

```

4  /* structure declarations */
5  struct queue {
6      void **array;
7      int head;
8      int tail;
9      int num;
10     int size;
11 };
12
13 #endif
14

```

**File queue.c**

```

1  #include "queuePrivate.h"
2  #include "util.h"
3
4  /*
5   * create a new empty queue
6   */
7  queue_t *queue_init(int size) {
8      queue_t *qp;
9
10     qp = (queue_t *)util_malloc(sizeof(queue_t));
11     qp->size = size;
12     qp->head = qp->tail = qp->num = 0;
13     qp->array = (void **)util_malloc(size*sizeof(void *));
14     return qp;
15 }
16
17 /*
18  * return the number of elements stored in the queue
19  */
20 int queue_count(queue_t *qp) {
21     return (qp!=NULL) ? qp->num : 0;
22 }
23
24 /*
25  * store a new value in the queue (FIFO policy)
26  */
27 int queue_put(queue_t *qp, void *data) {
28     if (qp==NULL || qp->num>=qp->size) {
29         return 0;
30     }
31
32     qp->array[qp->tail] = data;
33     qp->tail = (qp->tail+1) % (qp->size);
34     qp->num++;
35     return 1;
36 }
37
38 /*
39  * extract a value from the queue (FIFO policy)
40  */
41 int queue_get(queue_t *qp, void **data_ptr) {
42     if (qp==NULL || qp->num<=0) {
43         return 0;
44     }
45
46     *data_ptr = qp->array[qp->head];
47     qp->head = (qp->head+1) % (qp->size);

```

```

48     qp->num--;
49     return 1;
50 }
51
52 /* *
53 * print all the queue elements (FIFO policy)
54 */
55 void queue_print(FILE *fp, queue_t *qp, void (*print)(FILE *, void *)) {
56     int i, j;
57
58     if (qp != NULL) {
59         for (i=0; i<qp->num; i++) {
60             j = (qp->head+i) % (qp->size);
61             print(fp, qp->array[j]);
62             fprintf(fp, "\n");
63         }
64     }
65 }
66
67 /* *
68 * deallocate all the memory associated to the queue
69 */
70 void queue_dispose(queue_t *qp, void (*quit)(void *)) {
71     int i;
72
73     if (qp != NULL) {
74         if (quit != NULL) {
75             for (i=qp->head; qp->num>0; i=(i+1)%qp->size) {
76                 quit(qp->array[i]);
77                 qp->num--;
78             }
79         }
80         free(qp->array);
81         free(qp);
82     }
83
84     return; }
```

## Solution 2

This second implementation is based on a dynamic list. More specifically, it uses both a head and a tail pointers to access the list. A wrapper, of type `struct queue` is used to store those two access pointers.

The circular list case can be easily derived from this one, by following the introductory part reported in this chapter.

### *File client.c*

```

1 #include <stdio.h>
2 #include <ctype.h>
3 #include "queuePublic.h"
4 #include "item.h"
5
6 #define SIZE 50
7 #define MAX 20
8
9 int main (void) {
10     char command[MAX];
11     queue_t *qp=NULL;
```

```

12 item_t data;
13 int stop=0;
14
15 do {
16     fprintf(stdout, "\nAvailable commands:\n");
17     fprintf(stdout, " (N)ew\n (I)nsert\n");
18     fprintf(stdout, " (E)xtract\n (D)isplay\n");
19     fprintf(stdout, " (S)top\nChoice: ");
20     scanf("%s", command);
21
22 switch (tolower(command[0])) {
23     case 'n':
24         queue_dispose(qp, item_dispose);
25         qp = queue_init(SIZE);
26         fprintf(stdout, "New queue correctly created.\n");
27         break;
28     case 'i':
29         fprintf(stdout, "Input data: ");
30         item_read(stdin, (void **)&data);
31         if (!queue_put(qp, (void *)data)) {
32             fprintf(stdout, "Error in put!\n");
33         } else {
34             fprintf(stdout, "Data correctly inserted.\n");
35         }
36         break;
37     case 'e':
38         if (!queue_get(qp, (void **)&data)) {
39             fprintf(stdout, "Error in get!\n");
40         } else {
41             fprintf(stdout, "Extracted data: ");
42             item_print(stdout, (void *)data);
43             fprintf(stdout, "\n");
44         }
45         break;
46     case 'd':
47         queue_print(stdout, qp, item_print);
48         break;
49     case 's':
50         stop = 1;
51         break;
52     default:
53         fprintf(stdout, "Unknown command!\n");
54     }
55 } while (!stop);
56
57 queue_dispose(qp, item_dispose);
58 return 0;
59 }
```

## File queuePublic.h

```

1 #ifndef _QUEUE_PUBLIC
2 #define _QUEUE_PUBLIC
3
4 #include <stdio.h>
5
6 /* macro definition */
7 #define queue_empty_m(qp) (queue_count(qp) == 0)
8
9 /* type declarations */
```

```

10 typedef struct queue queue_t;
11
12 /* extern function prototypes */
13 extern queue_t *queue_init(int size);
14 extern int queue_count(queue_t *sp);
15 extern int queue_put(queue_t *sp, void *data);
16 extern int queue_get(queue_t *sp, void **data_ptr);
17 extern void queue_print(FILE *fp, queue_t *sp, void (*print)(FILE *, void *));
18 extern void queue_dispose(queue_t *sp, void (*quit)(void *));
19
20 #endif

```

### **File queuePrivate.h**

```

1 #ifndef _QUEUE_PRIVATE
2 #define _QUEUE_PRIVATE
3
4 #include "util.h"
5 #include "queuePublic.h"
6
7 /* structure declarations */
8 typedef struct node {
9     void *data;
10    struct node *next;
11 } node_t;
12
13 struct queue {
14     node_t *head;
15     node_t *tail;
16     int num;
17 };
18
19 #endif

```

### **File queue.c**

```

1 #include "queuePrivate.h"
2 #include "util.h"
3
4 /*
5  *  create a new empty queue
6  */
7 queue_t *queue_init(int size) {
8     queue_t *qp;
9
10    qp = (queue_t *)util_malloc(sizeof(queue_t));
11    qp->head = qp->tail = NULL;
12    qp->num = 0;
13    return qp;
14 }
15
16 /*
17  *  return the number of elements stored in the queue
18  */
19 int queue_count(queue_t *qp) {
20     return (qp!=NULL) ? qp->num : 0;
21 }
22
23 /*
24  *  store a new value in the queue (FIFO policy)
25  */

```

```
26 int queue_put(queue_t *qp, void *data) {
27     node_t *node;
28
29     if (qp == NULL) {
30         return 0;
31     }
32
33     node = (node_t *)malloc(sizeof(node_t));
34     node->data = data;
35     node->next = NULL;
36
37     if (qp->head == NULL) {
38         qp->head = node;
39     } else {
40         qp->tail->next = node;
41     }
42     qp->tail = node;
43     qp->num++;
44
45     return 1;
46 }
47
48 /*
49 * extract a value from the queue (FIFO policy)
50 */
51 int queue_get(queue_t *qp, void **data_ptr) {
52     node_t *node;
53
54     if (qp==NULL || qp->head==NULL) {
55         return 0;
56     }
57
58     node = qp->head;
59     *data_ptr = node->data;
60     qp->head = node->next;
61     if (qp->head == NULL) {
62         qp->tail = NULL;
63     }
64     qp->num--;
65
66     free(node);
67     return 1;
68 }
69
70 /*
71 * print all the queue elements (FIFO policy)
72 */
73 void queue_print(FILE *fp, queue_t *qp, void (*print)(FILE *, void *)) {
74     node_t *node;
75
76     if (qp != NULL) {
77         node = qp->head;
78         while (node != NULL) {
79             print(fp, node->data);
80             fprintf(fp, "\n");
81             node = node->next;
82         }
83     }
84 }
```

```

85  /*
86   *   deallocate all the memory associated to the queue
87   */
88
89 void queue_dispose(queue_t *qp, void (*quit)(void *)) {
90     node_t *node;
91
92     if (qp != NULL) {
93         while (qp->head != NULL) {
94             node = qp->head;
95             qp->head = node->next;
96             if (quit != NULL) {
97                 quit(node->data);
98             }
99             free(node);
100        }
101        free(qp);
102    }
103 }

```

## 9.8 A List Library

### Specifications

Implement a library to manipulate a linked list to store generic data types. The library has to deliver all main standard linked list operations (such as insertion, search, and extraction).

Use the such a library to solve Exercise 4.13. In that exercise the core idea is to implement two logical lists overlapped into a unique physical list to avoid data duplication. In this case, in a much less efficient way, simply duplicate all data into two physically separated lists.

### Solution

Notice that this application also uses the library defined in Exercises 9.5 (files util.\*).

#### File client.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "util.h"
5 #include "list.h"
6
7 #define MAX 50
8
9 typedef struct {
10     char *name;
11     float mark;
12 } student_t;
13
14 /* static function prototypes */
15 int read_file (list_t **lpp_name, list_t **lpp_mark, char *filename);
16 void print_file (list_t *lp, char *filename);
17 float compare_by_name (void *ptr1, void *ptr2);
18 float compare_by_mark (void *ptr1, void *ptr2);

```

```

19 void student_print (FILE *fp, void *ptr);
20 void student_dispose (void *ptr);
21
22 /* 
23  * main program
24 */
25 int main (void) {
26     list_t *lp_name=NULL, *lp_mark=NULL;
27     char command[MAX];
28
29     do {
30         fprintf(stdout, "\nAvailable commands:\n");
31         fprintf(stdout, "    read <file_name>\n");
32         fprintf(stdout, "    writeMarks <file_name> ");
33         fprintf(stdout, "[<file_name> = \"stdout\" --> on screen]\n");
34         fprintf(stdout, "    writeNames <file_name> ");
35         fprintf(stdout, "[<file_name> = \"stdout\" --> on screen]\n");
36         fprintf(stdout, "    stop\n");
37         fprintf(stdout, "\nChoice --> ");
38         scanf("%s", command);
39
40         if (strcmp(command, "read") == 0) {
41             scanf("%s", command);
42             read_file(&lp_name, &lp_mark, command);
43         } else if (strcmp(command, "writeNames") == 0) {
44             scanf("%s", command);
45             print_file(lp_name, command);
46         } else if (strcmp(command, "writeMarks") == 0) {
47             scanf("%s", command);
48             print_file(lp_mark, command);
49         } else if (strcmp(command, "stop") != 0) {
50             fprintf(stdout, "Error, unknown command: %s", command);
51         }
52     } while (strcmp(command, "stop") != 0);
53
54     list_dispose(lp_name, NULL);
55     list_dispose(lp_mark, student_dispose);
56     return EXIT_SUCCESS;
57 }
58
59 /*
60  * load the input file contents
61 */
62 int read_file (list_t **lpp_name, list_t **lpp_mark, char *filename) {
63     student_t *student;
64     char name[MAX];
65     float mark;
66     int num=0;
67     FILE *fp;
68
69     /* free the old lists and init the new ones */
70     list_dispose(*lpp_name, NULL);
71     list_dispose(*lpp_mark, student_dispose);
72     *lpp_name = list_init();
73     *lpp_mark = list_init();
74
75     /* fill the lists with the data from file */
76     fp = util_fopen(filename, "r");
77     while (fscanf(fp, "%s %f", name, &mark) != EOF) {

```

```
78     num++;
79     student = (student_t *)util_malloc(sizeof(student_t));
80     student->name = strdup(name);
81     student->mark = mark;
82
83     *lpp_name = list_insert(*lpp_name, student, compare_by_name);
84     *lpp_mark = list_insert(*lpp_mark, student, compare_by_mark);
85 }
86 fclose(fp);
87
88 fprintf(stdout, "Loaded %d students\n", num);
89 return num;
90 }
91
92 /*
93 * write an output file
94 */
95 void print_file (list_t *lp, char *filename) {
96     FILE *fp;
97
98     if (strcmp(filename, "stdout") != 0) {
99         fp = util_fopen(filename, "w");
100    } else {
101        fp = stdout;
102    }
103
104    list_print(fp, lp, student_print);
105
106    if (strcmp(filename, "stdout") != 0) {
107        fclose(fp);
108    }
109 }
110
111 /*
112 * write a student on file
113 */
114 void student_print (FILE *fp, void *ptr) {
115     student_t *s = (student_t *)ptr;
116
117     fprintf(fp, "%s %.2f\n", s->name, s->mark);
118 }
119
120 /*
121 * free the memory for a student structure
122 */
123 void student_dispose (void *ptr) {
124     student_t *s = (student_t *)ptr;
125
126     free(s->name);
127     free(s);
128 }
129
130 /*
131 * compare two students (by name)
132 */
133 float compare_by_name (void *ptr1, void *ptr2) {
134     student_t *s1 = (student_t *)ptr1;
135     student_t *s2 = (student_t *)ptr2;
```

```

137     return strcmp(s1->name, s2->name);
138 }
139
140 /* compare two students (by mark)
141 */
142 float compare_by_mark (void *ptr1, void *ptr2) {
143     student_t *s1 = (student_t *)ptr1;
144     student_t *s2 = (student_t *)ptr2;
145
146     return s1->mark - s2->mark;
147 }

```

**File list.h**

```

1 #ifndef _LIST
2 #define _LIST
3
4 #include <stdio.h>
5
6 /* macro definition */
7 #define list_empty_m(lp) (lp == NULL)
8
9 /* type declaration */
10 typedef struct list_s list_t;
11
12 /* extern function prototypes */
13 extern list_t *list_init(void);
14 extern int list_count(list_t *lp);
15 extern int list_search(list_t *lp, void *key, float (*compare)(void *, void *),
16                           void **data_ptr);
17 extern list_t *list_add(list_t *lp, void *data, int pos);
18 extern list_t *list_insert(list_t *lp, void *data, float (*compare)(void *, void *));
19 extern list_t *list_extract(list_t *lp, void *key, float (*compare)(void *, void *),
20                           void **data_ptr);
21 extern void list_print(FILE *fp, list_t *lp, void (*print)(FILE *, void *));
22 extern void list_dispose(list_t *lp, void (*quit)(void *));
23
24 #endif

```

**File list.c**

```

1 #include <stdlib.h>
2 #include "util.h"
3 #include "list.h"
4
5 /* structure declaration */
6 struct list_s {
7     void *data;
8     struct list_s *next;
9 };
10
11 /*
12  * create a new empty list
13 */
14 list_t *list_init (void) {
15     return NULL;
16 }
17
18 /*
19  * return the number of elements of a list

```

```
20  */
21 int list_count (list_t *lp) {
22     int size=0;
23
24     while (lp != NULL) {
25         size++;
26         lp = lp->next;
27     }
28
29     return size;
30 }
31
32 /**
33  *  search an object in a list
34  */
35 int list_search (
36     list_t *lp, void *key, float (*compare)(void *, void *),
37     void **data_ptr
38 ) {
39     for ( ; lp!=NULL; lp=lp->next) {
40         if (compare(lp->data, key) == 0) {
41             *data_ptr = lp->data;
42             return 1;
43         }
44     }
45
46     return 0;
47 }
48
49 /**
50  *  add an object into a list (position specified!)
51  */
52 list_t *list_add (list_t *lp, void *data, int pos) {
53     list_t *node, *ptr;
54     int idx = 1;
55
56     node = (list_t *)util_malloc(sizeof(list_t));
57     node->data = data;
58
59     /* insertion ahead */
60     if (lp==NULL || pos==0) {
61         node->next = lp;
62         return node;
63     }
64
65     /* insertion in the middle (or at the end) */
66     ptr = lp;
67     while (ptr->next!=NULL && idx!=pos) {
68         ptr = ptr->next;
69         idx++;
70     }
71     node->next = ptr->next;
72     ptr->next = node;
73
74     return lp;
75 }
76
77 /**
78  *  add an object into a sorted list
```

```
79  /*
80   list_t *list_insert (list_t *lp, void *data, float (*compare)(void *, void *)) {
81   list_t *ptr, *node;
82   node = (list_t *)util_malloc(sizeof(list_t));
83   node->data = data;
84
85   /* insertion ahead */
86   if (lp==NULL || compare(data, lp->data)<0) {
87     node->next = lp;
88     return node;
89   }
90
91   /* insertion in the middle (or at the end) */
92   ptr = lp;
93   while (ptr->next!=NULL && compare(ptr->next->data, data)<0) {
94     ptr = ptr->next;
95   }
96   node->next = ptr->next;
97   ptr->next = node;
98
99   return lp;
100 }
101
102 /*
103  * extract a specified object from a list
104 */
105 list_t *list_extract (
106   list_t *lp, void *key, float (*compare)(void *, void *), void **data_ptr
107 )
108 {
109   list_t *node, *ptr;
110
111   *data_ptr = NULL;
112
113   if (lp == NULL) {
114     return NULL;
115   }
116   ptr = lp;
117
118   /* extraction ahead */
119   if (compare(key, lp->data) == 0) {
120     *data_ptr = lp->data;
121     lp = lp->next;
122     free(ptr);
123     return lp;
124   }
125
126   /* extraction in the middle (or at the end) */
127   while (ptr->next!=NULL && compare(key, ptr->next->data)!=0) {
128     ptr = ptr->next;
129   }
130   if (ptr->next != NULL) {
131     node = ptr->next;
132     *data_ptr = node->data;
133     ptr->next = node->next;
134     free(node);
135   }
136
137   return lp;
```

```

138 }
139 /*
140 * write a list on file
141 */
142 void list_print (FILE *fp, list_t *lp, void (*print) (FILE *, void *)) {
143     while (lp != NULL) {
144         print(fp, lp->data);
145         lp = lp->next;
146     }
147 }
148 */
149 /*
150 * write a list on file
151 */
152 void list_dispose (list_t *lp, void (*quit) (void *)) {
153     list_t *ptr;
154
155     while (lp != NULL) {
156         ptr = lp;
157         lp = lp->next;
158         if (quit != NULL) {
159             quit(ptr->data);
160         }
161         free(ptr);
162     }
163 }
164 }
```

## 9.9 Date

### Specifications

A file stores an undefined number of date, each one stored on a separate row of the file with the format dd/mm/yyyy, e.g., 24/07/2017.

Write a program able to search and print-out (with the same input format) the first and the last dates stored within the file. Moreover, the program has to compute the time interval (in terms of days, months, and years) between those two dates.

For the same of simplicity, assume that all months have 30 days, i.e., that a year is made-up of 360 days. Write a modular program, stored into more than a single source C file. More in details, write a library to deal with the “data” type, including a public and a private interface, which allow operations such as reading a date, comparing two dates, computing the time distance between two dates, etc.

## Solution

### *File client.c*

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "util.h"
4 #include "date.h"
5
6 int main (void) {
7     date_t prev, next, curr, diff;
8     FILE *fp;
9
10    fp = util_fopen("dates.txt", "r");
```

```

11 date_read(fp, &curr);
12 prev = next = curr;
13 while (date_read(fp, &curr) != EOF) {
14     if (date_compare(prev, curr) > 0) {
15         prev = curr;
16     }
17     if (date_compare(next, curr) < 0) {
18         next = curr;
19     }
20 }
21 fclose(fp);
22
23 fprintf(stdout, "Oldest date: ");
24 date_print(stdout, prev);
25 fprintf(stdout, "Newest date: ");
26 date_print(stdout, next);
27
28 diff = date_difference(prev, next);
29 fprintf(stdout, "Total time spent: ");
30 date_print(stdout, diff);
31
32 return EXIT_SUCCESS;
33 }
```

**File date.h**

```

1 #ifndef _DATE_INCLUDED
2 #define _DATE_INCLUDED
3
4 #include <stdio.h>
5
6 #define DAYS      30
7 #define MONTHS   12
8
9 /* type declarations */
10 typedef struct {
11     int d, m, y;
12 } date_t;
13
14 /* extern function prototypes */
15 extern int date_read(FILE *fp, date_t *dp);
16 extern void date_print(FILE *fp, date_t d);
17 extern int date_compare(date_t d1, date_t d2);
18 extern date_t date_difference(date_t d1, date_t d2);
19
20 #endif
```

**File date.c**

```

1 #include <stdio.h>
2 #include "date.h"
3
4 /*
5  *  read a date from file
6  */
7 int date_read(FILE *fp, date_t *dp) {
8     return fscanf(fp, "%d/%d/%d", &dp->d, &dp->m, &dp->y);
9 }
10
11 /*
12  *  print a date on file
13 */
```

```

13  */
14 void date_print(FILE *fp, date_t d) {
15     fprintf(fp, "%02d/%02d/%04d\n", d.d, d.m, d.y);
16 }
17
18 /**
19 * compare two dates
20 */
21 int date_compare(date_t d1, date_t d2) {
22     if (d1.y != d2.y) {
23         return d1.y-d2.y;
24     }
25     if (d1.m != d2.m) {
26         return d1.m-d2.m;
27     }
28     return d1.d-d2.d;
29 }
30
31 /**
32 * compute the "difference" of two dates
33 */
34 date_t date_difference(date_t d1, date_t d2) {
35     date_t r;
36
37     if (date_compare(d1, d2) > 0) {
38         /* d1 comes after d2: swap the dates */
39         return date_difference(d2, d1);
40     }
41     r.y = d2.y - d1.y;
42     r.m = d2.m - d1.m;
43     r.d = d2.d - d1.d;
44     if (r.d < 0) {
45         r.d += DAYS;
46         r.m--;
47     }
48     if (r.m < 0) {
49         r.m += MONTHS;
50         r.y--;
51     }
52
53     return r;
54 }
```

## 9.10 Plane Points

### Specifications

Write a program able to:

- ▷ Read an undefined number of points on the Cartesian plane from file, and store those data into an array of structures. All Cartesian “x-y” coordinates are stored in the file on subsequent rows. The first line of the file specifies the number of points. Call the array basic item `points`, and dynamically allocate the array to store those points.
- ▷ Compute:
  - ◊ The couple of closest points (printing-out their coordinates).
  - ◊ The couple of farthest points (printing-out their coordinates).

- ◊ The number of segments, obtained joining together any two points of the file, with a length smaller than a threshold  $d$ , specified by the user.
- ▷ Order (using any sorting algorithm) all points of increasing distance from the Cartesian coordinates system.

As for Exercise 9.9 write a modular program including a “point” library, with a public and a private interface, and a client.

## Solution

### File *client.c*

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <float.h>
4 #include "util.h"
5 #include "point.h"
6
7 /* static function prototypes */
8 static point_t *load(int *dim);
9 static void sort(point_t *v, int n);
10 static void write(point_t *v, int n);
11
12 /*
13  * main program
14  */
15 int main (void) {
16     int n, i, j, imax, jmax, imin, jmin, cnt=0;
17     float d, d2, max=0, min=FLT_MAX;
18     point_t *array;
19
20     array = load(&n);
21     fprintf(stdout, "Value for the distance d? ");
22     scanf("%f", &d);
23
24     for (i=0; i<n-1; i++) {
25         for (j=i+1; j<n; j++) {
26             d2 = distance2(array[i], array[j]);
27             if (d2 > max) {
28                 max = d2;
29                 imax = i;
30                 jmax = j;
31             }
32             if (d2 < min) {
33                 min = d2;
34                 imin = i;
35                 jmin = j;
36             }
37             if (d2 < d) {
38                 cnt++;
39             }
40         }
41     }
42
43     fprintf(stdout, "Minimum distance: %.3f\n", min);
44     fprintf(stdout, "Point 1: ");
45     point_print(stdout, array[imin]);
46     fprintf(stdout, "Point 2: ");

```

```
47     point_print(stdout, array[jmin]);
48
49     fprintf(stdout, "Maximum distance: %.3f\n", max);
50     fprintf(stdout, "Point 1: ");
51     point_print(stdout, array[imax]);
52     fprintf(stdout, "Point 2: ");
53     point_print(stdout, array[jmax]);
54
55     fprintf(stdout, "Number of segments with length < %.2f: %d\n", d, cnt);
56
57     sort(array, n);
58     write(array, n);
59
60     free(array);
61
62     return EXIT_SUCCESS;
63 }
64
65 /*
66 *   load the data from file
67 */
68 static point_t *load (int *dim) {
69     char name[21];
70     point_t *v, t;
71     FILE *fp;
72     int i, n=0;
73
74     fprintf(stdout, "Input file name? ");
75     scanf("%s", name);
76     fp = util_fopen(name, "r");
77     while (point_read(fp, &t) != EOF) {
78         n++;
79     }
80     fclose(fp);
81
82     v = (point_t *)util_malloc(n * sizeof(point_t));
83
84     fp = util_fopen(name, "r");
85     for (i=0; i<n; i++) {
86         point_read(fp, &v[i]);
87     }
88     fclose(fp);
89
90     *dim = n;
91     fprintf(stdout, "Number of loaded points: %d\n", n);
92     return v;
93 }
94
95 /*
96 *   sort the array of points according to their distance from origin
97 */
98 static void sort (point_t *v, int n) {
99     int i, j;
100    point_t key;
101
102    /* insertion sort */
103    for (i=1; i<n; i++) {
104        key = v[i];
105        j = i;
```

```

106     while (--j>=0 && distance1(key)<distance1(v[j])) {
107         v[j+1] = v[j];
108     }
109     v[j+1] = key;
110 }
111 }
112 /*
113 * write the array of points on file
114 */
115 static void write (point_t *v, int n) {
116     char name[21];
117     FILE *fp;
118     int i;
119
120     fprintf(stdout, "Output file name? ");
121     scanf("%s", name);
122     fp = fopen(name, "w");
123     for (i=0; i<n; i++) {
124         point_print(fp, v[i]);
125     }
126
127     fclose(fp);
128
129     return;
130 }

```

**File point.h**

```

1 #ifndef _POINT_INCLUDED
2 #define _POINT_INCLUDED
3
4 #include <stdio.h>
5
6 /* type declarations */
7 typedef struct {
8     float x, y, d;
9 } point_t;
10
11 /* extern function prototypes */
12 extern int point_read(FILE *fp, point_t *pp);
13 extern void point_print(FILE *fp, point_t p);
14 extern float distance1(point_t p);
15 extern float distance2(point_t p1, point_t p2);
16
17 #endif

```

**File point.c**

```

1 #include <stdio.h>
2 #include <math.h>
3 #include "point.h"
4
5 /*
6 * read a point from file
7 */
8 int point_read (FILE *fp, point_t *pp) {
9     int result = fscanf(fp, "%f %f", &pp->x, &pp->y);
10
11    if (result != EOF) {
12        pp->d = sqrt(pp->x*pp->x + pp->y*pp->y);

```

```

13     }
14     return result;
15 }
16
17 /**
18 * print a point on file
19 */
20 void point_print (FILE *fp, point_t p) {
21     fprintf(fp, "%+.2f %+.2f\n", p.x, p.y);
22 }
23
24 /**
25 * return the distance of a point from the origin
26 */
27 float distance1 (point_t p) {
28     return p.d;
29 }
30
31 /**
32 * return the distance between two points
33 */
34 float distance2 (point_t p1, point_t p2) {
35     float dx, dy;
36
37     dx = p2.x - p1.x;
38     dy = p2.y - p1.y;
39
40     return sqrt(dx*dx + dy*dy);
41 }
```

## 9.11 Modular Binary Search

### Specifications

A file stores a set of ordered data. Read those data in a proper dynamic data structure and allow the user (on request) to perform binary searches on those data.

As in previous cases, realize a modular program, with a “binary search” library, and a user client. Suppose that the file could store different type of data, such as strings (of maximum length equal to 100 characters), or integer values. For that reason the “binary search” library has to be “data independent” and include a “data library” data dependent.

### Solution

As in other exercises all *item* objects are taken from Exercise 9.6 (i.e., item-int.h, item-int.c, item-struct.h, item-struct.c).

#### **File client.c**

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <ctype.h>
4 #include "util.h"
5 #include "item.h"
6
7 #define MAX 20
8 #define DEBUG 1
```

```
9  /* static function prototypes */
10 static item_t *loadData(int *size_ptr);
11 static int binarySearch(item_t *array, item_t data, int l, int r);
12
13 /*
14  * main program
15 */
16 int main (void) {
17     int stop=0, size, pos;
18     item_t data, *array;
19     char command[MAX];
20
21     /* load the data array */
22     array = loadData(&size);
23
24     /* perform the searches */
25     while (!stop) {
26         printf("New search (Y/N) ? ");
27         scanf("%s", command);
28         if (tolower(command[0]) == 'y') {
29             printf("Item to search: ");
30             item_read(stdin, (void **)&data);
31             pos = binarySearch(array, data, 0, size-1);
32             printf("Item \"%");
33             item_print(stdout, data);
34             if (pos < 0) {
35                 printf("\" NOT found.\n");
36             } else {
37                 printf("\" found in position %d.\n", pos);
38             }
39             item_dispose((void *)data);
40         } else {
41             printf("Execution stopped.\n");
42             stop = 1;
43         }
44     }
45 }
46
47 util_array_dispose((void **)array, size, item_dispose);
48 return EXIT_SUCCESS;
49 }
50
51 /*
52  * load the data from file
53 */
54 static item_t *loadData (int *size_ptr) {
55     char filename[MAX];
56     item_t data, *array;
57     int i, size=0;
58     FILE *fp;
59
60     printf("Input the file name: ");
61     scanf("%s", filename);
62
63     /* count the number of data */
64     fp = util_fopen(filename, "r");
65     while (item_read(fp, (void **)&data) != EOF) {
66         size++;
67         item_dispose((void *)data);
```

```

68      }
69      fclose(fp);
70
71     /* now store the data */
72     array = (item_t *)util_malloc(size*sizeof(item_t));
73     fp = util_fopen(filename, "r");
74     for (i=0; i<size; i++) {
75         item_read(fp, (void **)&array[i]);
76     }
77     fclose(fp);
78
79 #if DEBUG
80     /* debug print out */
81     printf("Number of data loaded: %d\n", size);
82     for (i=0; i<size; i++) {
83         printf("%2d: ", i);
84         item_print(stdout, array[i]);
85         printf("\n");
86     }
87     printf("\n");
88 #endif
89
90     *size_ptr = size;
91     return array;
92 }
93
94 /*
95  * binary search, recursive function
96 */
97 static int binarySearch (item_t *array, item_t d, int l, int r) {
98     int result, c = (l+r)/2;
99
100    if (l > r) {
101        return -1;
102    }
103
104    result = item_compare(d, array[c]);
105    if (result < 0) {
106        return binarySearch(array, d, l, c-1);
107    }
108    if (result > 0) {
109        return binarySearch(array, d, c+1, r);
110    }
111    return c;
112 }

```

## 9.12 Sorting Library

### Specifications

A file stores data for all student enrolled at Politecnico di Torino. The first row of the file specifies the number of students. Each subsequent row specifies a student register number (integer value), his/her surname (string of maximum 20 characters), number of examination passed (integer value), and his/her average mark (real value).

Write a modular program able to sort a set of  $n$  student data stored on file based on their register number.

The program has to include a “sorting library” implementing all main ordering strategies.

## Solution 1

Notice that this application uses the libraries defined in Exercises 9.5 and 9.6 (file util.\* and item.\*).

### File client.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <ctype.h>
4 #include "util.h"
5 #include "item.h"
6 #include "sort.h"

7 #define DEBUG 1

8 /* static function prototypes */
9 static item_t *loadData(int *size_ptr);
10 static void printData(item_t *array, int num);

11 /*
12  * main program
13  */
14 int main (void) {
15     item_t *array;
16     int num;

17     /* load the data array */
18     array = loadData(&num);

19     /* sort the data array */
20     sort_quick(array, num);

21     /* print the resulting array */
22     printData(array, num);

23     util_array_dispose((void **)array, num, item_dispose);
24     return EXIT_SUCCESS;
25 }

26 /*
27  * load the data from file
28  */
29 static item_t *loadData(int *num_ptr) {
30     char filename[51];
31     item_t *array;
32     int i, num=0;
33     FILE *fp;

34     fprintf(stdout, "Input the file name: ");
35     scanf("%s", filename);

36     fp = util_fopen(filename, "r");
37     fscanf(fp, "%d", &num);
38     array = (item_t *)util_malloc(num*sizeof(item_t));
39     for (i=0; i<num; i++) {
40         item_read(fp, (void **)&array[i]);
41     }
42     fclose(fp);
```

```

53
54 #if DEBUG
55     /* debug print out */
56     fprintf(stdout, "Number of data loaded: %d\n", num);
57     printData(array, num);
58     fprintf(stdout, "\n");
59 #endif
60
61     *num_ptr = num;
62     return array;
63 }
64
65 /*
66     * print-out the sorted data array
67     */
68 static void printData(item_t *array, int num) {
69     int i;
70
71     for (i=0; i<num; i++) {
72         item_print(stdout, array[i]);
73         fprintf(stdout, "\n");
74     }
75 }
```

### **File sort.h**

```

1 #ifndef _SORT
2 #define _SORT
3
4 #include "item.h"
5
6 /* extern function prototypes */
7 extern void sort_insertion(item_t *array, int num);
8 extern void sort_selection(item_t *array, int num);
9 extern void sort_bubble(item_t *array, int num);
10 extern void sort_quick(item_t *array, int num);
11 extern void sort_merge(item_t *array, int num);
12
13 #endif
```

### **File sort.c**

```

1 #include "sort.h"
2 #include "util.h"
3
4 /* static function prototypes */
5 static void quick_sort_recur(item_t *array, int p, int r);
6 static void merge_sort_recur(item_t *array, item_t *merged, int p, int r);
7 static void swap(item_t *array, int i, int j);
8
9 /*
10    * insertion sort
11    */
12 void sort_insertion(item_t *array, int num)  {
13     item_t key;
14     int i, j;
15
16     for (i=1; i<num; i++) {
17         key = array[i];
18         j = i;
19         while (--j>=0 && item_compare(array[j], key)>0) {
```

```

20         array[j+1] = array[j];
21     }
22     array[j+1] = key;
23 }
24 }
25 */
26 * selection sort
27 */
28 void sort_selection(item_t *array, int num) {
29     int i, j, min;
30
31     for (i=0; i<num-1; i++) {
32         min = i;
33         for (j=i+1; j<num; j++) {
34             if (item_compare(array[j], array[min]) < 0) {
35                 min = j;
36             }
37         }
38         swap(array, i, min);
39     }
40 }
41 }
42 */
43 * bubble sort
44 */
45 void sort_bubble(item_t *array, int num) {
46     int i, j;
47
48     for (i=num-1; i>=1; i--) {
49         for (j=0; j<i; j++) {
50             if (item_compare(array[j], array[j+1]) > 0) {
51                 swap(array, j, j+1);
52             }
53         }
54     }
55 }
56 }
57 */
58 * quick sort, "wrapper" function
59 */
60 void sort_quick(item_t *array, int num) {
61     quick_sort_recur(array, 0, num-1);
62 }
63 }
64 */
65 */
66 * quick sort, recursive function
67 */
68 static void quick_sort_recur(item_t *array, int p, int r) {
69     int i=p-1, j=r+1;
70     item_t x;
71
72     if (p < r) {
73         x = array[p];
74         while (i < j) {
75             while (item_compare(array[++i], x) < 0) ;
76             while (item_compare(array[--j], x) > 0) ;
77             if (i < j) {
78                 swap(array, i, j);

```

```
79         }
80     }
81     quick_sort_recur(array, p, j);
82     quick_sort_recur(array, j+1, r);
83 }
84 }
85 */
86 /* merge sort, "wrapper" function
87 */
88 void sort_merge(item_t *array, int num) {
89     item_t *merged;
90
91     merged = (item_t *)util_malloc(num*sizeof(item_t));
92     merge_sort_recur(array, merged, 0, num-1);
93     free(merged);
94 }
95
96 /*
97 * merge sort, recursive function
98 */
99 static void merge_sort_recur(item_t *array, item_t *merged, int p, int r) {
100    int i, j, k, q;
101
102    if (p < r) {
103        q = (p + r) / 2;
104        merge_sort_recur(array, merged, p, q);
105        merge_sort_recur(array, merged, q+1, r);
106
107        for (i=k=p, j=q+1; i<=q && j<=r; ) {
108            if (item_compare(array[i], array[j]) < 0) {
109                merged[k++] = array[i++];
110            } else {
111                merged[k++] = array[j++];
112            }
113        }
114    }
115
116    while (i <= q) {
117        merged[k++] = array[i++];
118    }
119    while (j <= r) {
120        merged[k++] = array[j++];
121    }
122    for (k=p; k<=r; k++) {
123        array[k] = merged[k];
124    }
125 }
126 }
127 }
128 */
129 /* swap two elements of an array
130 */
131 static void swap(item_t *array, int i, int j) {
132     item_t tmp;
133
134     if (i != j) {
135         tmp = array[i];
136         array[i] = array[j];
137         array[j] = tmp;
```

```
138     array[j] = tmp;
139 }
140 }
```

## Solution 2

This application uses the libraries defined in Exercises 9.5 and 9.6 (file `util.*` and `item.*`).

### File `client.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <ctype.h>
4 #include "util.h"
5 #include "item.h"
6 #include "sort.h"
7
8 #define DEBUG 1
9
10 /* static function prototypes */
11 static item_t *loadData(int *size_ptr);
12 static void printData(item_t *array, int num);
13
14 /*
15  * main program
16  */
17 int main (void) {
18     item_t *array;
19     int num;
20
21     /* load the data array */
22     array = loadData(&num);
23
24     /* sort the data array */
25     sort_quick((void **)array, num, item_compare);
26
27     /* print the resulting array */
28     printData(array, num);
29
30     util_array_dispose((void **)array, num, item_dispose);
31     return EXIT_SUCCESS;
32 }
33
34 /*
35  * load the data from file
36  */
37 static item_t *loadData(int *num_ptr) {
38     char filename[51];
39     item_t *array;
40     int i, num=0;
41     FILE *fp;
42
43     fprintf(stdout, "Input the file name: ");
44     scanf("%s", filename);
45
46     fp = util_fopen(filename, "r");
47     fscanf(fp, "%d", &num);
48     array = (item_t *)util_malloc(num*sizeof(item_t));
49     for (i=0; i<num; i++) {
```

```

50     item_read(fp, (void **) &array[i]);
51 }
52 fclose(fp);
53
54 #if DEBUG
55 /* debug print out */
56 fprintf(stdout, "Number of data loaded: %d\n", num);
57 printData(array, num);
58 fprintf(stdout, "\n");
59#endif
60
61 *num_ptr = num;
62 return array;
63 }
64
65 /*
66 * print-out the sorted data array
67 */
68 static void printData(item_t *array, int num) {
69     int i;
70
71     for (i=0; i<num; i++) {
72         item_print(stdout, array[i]);
73         fprintf(stdout, "\n");
74     }
75 }
```

### File sort.h

```

1 #ifndef _SORT
2 #define _SORT
3
4 /* extern function prototypes */
5 extern void sort_insertion(void **array, int num, int (*compare)(void *, void *));
6 extern void sort_selection(void **array, int num, int (*compare)(void *, void *));
7 extern void sort_bubble(void **array, int num, int (*compare)(void *, void *));
8 extern void sort_quick(void **array, int num, int (*compare)(void *, void *));
9 extern void sort_merge(void **array, int num, int (*compare)(void *, void *));
10
11#endif
```

### File sort.c

```

1 #include "sort.h"
2 #include "util.h"
3
4 /* static function prototypes */
5 static void quick_sort_recur(void **array, int p, int r,
6     int (*compare)(void *, void *));
7 static void merge_sort_recur(void **array, void **merged, int p, int r,
8     int (*compare)(void *, void *));
9 static void swap(void **array, int i, int j);
10
11 /*
12 * insertion sort
13 */
14 void sort_insertion (void **array, int num, int (*compare)(void *, void *)) {
15     void *key;
16     int i, j;
17
18     for (i=1; i<num; i++) {
```

```

key = array[i];
19   j = i;
20   while (--j>=0 && compare(array[j], key)>0) {
21     array[j+1] = array[j];
22   }
23   array[j+1] = key;
24 }
25 }
26 }
27 */
28 * selection sort
29 */
30 void sort_selection (void **array, int num, int (*compare)(void *, void *)) {
31   int i, j, min;
32   for (i=0; i<num-1; i++) {
33     min = i;
34     for (j=i+1; j<num; j++) {
35       if (compare(array[j], array[min]) < 0) {
36         min = j;
37       }
38     }
39     swap(array, i, min);
40   }
41 }
42 }
43 */
44 */
45 * bubble sort
46 */
47 */
48 void sort_bubble (void **array, int num, int (*compare)(void *, void *)) {
49   int i, j;
50   for (i=num-1; i>=1; i--) {
51     for (j=0; j<i; j++) {
52       if (compare(array[j], array[j+1]) > 0) {
53         swap(array, j, j+1);
54       }
55     }
56   }
57 }
58 }
59 */
60 */
61 * quick sort, "wrapper" function
62 */
63 void sort_quick (void **array, int num, int (*compare)(void *, void *)) {
64   quick_sort_recur(array, 0, num-1, compare);
65 }
66 */
67 */
68 * quick sort, recursive function
69 */
70 static void quick_sort_recur(
71   void **array, int p, int r, int (*compare)(void *, void *)
72 ) {
73   int i=p-1, j=r+1;
74   void * x;
75   if (p < r) {
76     x = array[p];
77   }

```

```

78     while (i < j) {
79         while (compare(array[++i], x) < 0) ;
80         while (compare(array[--j], x) > 0) ;
81         if (i < j) {
82             swap(array, i, j);
83         }
84     }
85
86     quick_sort_recur(array, p, j, compare);
87     quick_sort_recur(array, j+1, r, compare);
88 }
89 }
90
91 /*
92 * merge sort, "wrapper" function
93 */
94 void sort_merge (
95     void **array, int num, int (*compare) (void *, void *)
96 ) {
97     void **merged;
98
99     merged = (void **)util_malloc(num*sizeof(void *));
100    merge_sort_recur(array, merged, 0, num-1, compare);
101    free(merged);
102 }
103
104 /*
105 * merge sort, recursive function
106 */
107 static void merge_sort_recur (
108     void **array, void **merged, int p, int r, int (*compare) (void *, void *)
109 ) {
110     int i, j, k, q;
111
112     if (p < r) {
113         q = (p + r) / 2;
114         merge_sort_recur(array, merged, p, q, compare);
115         merge_sort_recur(array, merged, q+1, r, compare);
116
117         for (i=k=p, j=q+1; i<=q && j<=r; ) {
118             if (compare(array[i], array[j]) < 0) {
119                 merged[k++] = array[i++];
120             } else {
121                 merged[k++] = array[j++];
122             }
123         }
124
125         while (i <= q) {
126             merged[k++] = array[i++];
127         }
128         while (j <= r) {
129             merged[k++] = array[j++];
130         }
131         for (k=p; k<=r; k++) {
132             array[k] = merged[k];
133         }
134     }
135 }
136

```

```

137 /* swap two elements of an array
138 */
139 static void swap (void **array, int i, int j) {
140     void * tmp;
141
142     if (i != j) {
143         tmp = array[i];
144         array[i] = array[j];
145         array[j] = tmp;
146     }
147 }
148 }
```

### Solution 3

#### *File client.c*

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <ctype.h>
4 #include "util.h"
5 #include "sort.h"
6
7 #define DEBUG 1
8
9 /* structure declarations */
10 typedef struct {
11     int id;
12     char *name;
13     int exams;
14     float avg;
15 } student_t;
16
17 /* static function prototypes */
18 static student_t **loadStudents(int *size_ptr);
19 static void printStudents(student_t **array, int num);
20 static int compareStudents(void *ptr1, void *ptr2);
21 static void disposeStudent(void *ptr);
22
23 /*
24  * main program
25 */
26 int main (void) {
27     student_t **array;
28     int num;
29
30     /* load the data array */
31     array = loadStudents(&num);
32
33     /* sort the data array */
34     sort_quick((void **)array, num, compareStudents);
35
36     /* print the resulting array */
37     printStudents(array, num);
38
39     util_array_dispose((void **)array, num, disposeStudent);
40     return EXIT_SUCCESS;
41 }
42
43 /*
```

```
44     * load the data from file
45     */
46 static student_t **loadStudents (int *num_ptr) {
47     char filename[51], name[51];
48     student_t **array, *tmp;
49     int id, exams, num, i;
50     float avg;
51     FILE *fp;
52
53     fprintf(stdout, "Input the file name: ");
54     scanf("%"s, filename);
55
56     fp = util_fopen(filename, "r");
57     fscanf(fp, "%d", &num);
58     array = (student_t **)util_malloc(num*sizeof(student_t *));
59     for (i=0; i<num; i++) {
60         fscanf(fp, "%d %s %d %f", &id, name, &exams, &avg);
61         tmp = (student_t *)util_malloc(sizeof(student_t));
62         tmp->id = id;
63         tmp->name = util_strdup(name);
64         tmp->exams = exams;
65         tmp->avg = avg;
66         array[i] = tmp;
67     }
68     fclose(fp);
69
70 #if DEBUG
71     /* debug print out */
72     fprintf(stdout, "Number of data loaded: %d\n", num);
73     printStudents(array, num);
74     fprintf(stdout, "\n");
75 #endif
76
77     *num_ptr = num;
78     return array;
79 }
80
81 /*
82  * print-out the sorted data array
83 */
84 static void printStudents (student_t **array, int num) {
85     int i;
86
87     for (i=0; i<num; i++) {
88         fprintf(stdout, "%7d %-20s ", array[i]->id, array[i]->name);
89         fprintf(stdout, "%2d %.2f\n", array[i]->exams, array[i]->avg);
90     }
91 }
92
93 /*
94  * compare two students (by id)
95 */
96 static int compareStudents (void *ptr1, void *ptr2) {
97     student_t *s1 = (student_t *)ptr1;
98     student_t *s2 = (student_t *)ptr2;
99     return s1->id - s2->id;
100 }
101
102 /*
```

```

103 * free a student structure
104 */
105 static void disposeStudent (void *ptr) {
106     student_t *s = (student_t *)ptr;
107     free(s->name);
108     free(s);
109 }
```

**File sort.h**

```

1 #ifndef _SORT
2 #define _SORT
3
4 /* extern function prototypes */
5 extern void sort_insertion(void **array, int num, int (*compare)(void *, void *));
6 extern void sort_selection(void **array, int num, int (*compare)(void *, void *));
7 extern void sort_bubble(void **array, int num, int (*compare)(void *, void *));
8 extern void sort_quick(void **array, int num, int (*compare)(void *, void *));
9 extern void sort_merge(void **array, int num, int (*compare)(void *, void *));
10
11 #endif
```

**File sort.c**

```

1 #include "sort.h"
2 #include "util.h"
3
4 /* static function prototypes */
5 static void quick_sort_recur(void **array, int p, int r,
6     int (*compare)(void *, void *));
7 static void merge_sort_recur(void **array, void **merged, int p, int r,
8     int (*compare)(void *, void *));
9 static void swap(void **array, int i, int j);
10
11 /*
12  * insertion sort
13  */
14 void sort_insertion (
15     void **array, int num, int (*compare)(void *, void *)
16 ) {
17     void *key;
18     int i, j;
19
20     for (i=1; i<num; i++) {
21         key = array[i];
22         j = i;
23         while (--j>=0 && compare(array[j], key)>0) {
24             array[j+1] = array[j];
25         }
26         array[j+1] = key;
27     }
28 }
29
30 /*
31  * selection sort
32  */
33 void sort_selection (
34     void **array, int num, int (*compare)(void *, void *)
35 ) {
36     int i, j, min;
```

```
38     for (i=0; i<num-1; i++) {
39         min = i;
40         for (j=i+1; j<num; j++) {
41             if (compare(array[j], array[min]) < 0) {
42                 min = j;
43             }
44         }
45         swap(array, i, min);
46     }
47 }
48
49 /*
50 * bubble sort
51 */
52 void sort_bubble (
53     void **array, int num, int (*compare) (void *, void *)
54 ) {
55     int i, j;
56
57     for (i=num-1; i>=1; i--) {
58         for (j=0; j<i; j++) {
59             if (compare(array[j], array[j+1]) > 0) {
60                 swap(array, j, j+1);
61             }
62         }
63     }
64 }
65
66 /*
67 * quick sort, "wrapper" function
68 */
69 void sort_quick (
70     void **array, int num, int (*compare) (void *, void *)
71 ) {
72     quick_sort_recur(array, 0, num-1, compare);
73 }
74
75 /*
76 * quick sort, recursive function
77 */
78 static void quick_sort_recur (
79     void **array, int p, int r, int (*compare) (void *, void *)
80 ) {
81     int i=p-1, j=r+1;
82     void * x;
83
84     if (p < r) {
85         x = array[p];
86         while (i < j) {
87             while (compare(array[++i], x) < 0) ;
88             while (compare(array[--j], x) > 0) ;
89             if (i < j) {
90                 swap(array, i, j);
91             }
92         }
93
94         quick_sort_recur(array, p, j, compare);
95         quick_sort_recur(array, j+1, r, compare);
96     }
```

```

97 }
98 /*
99 * merge sort, "wrapper" function
100 */
101
102 void sort_merge (
103     void **array, int num, int (*compare) (void *, void *)
104 ) {
105     void **merged;
106
107     merged = (void **)util_malloc(num*sizeof(void *));
108     merge_sort_recur(array, merged, 0, num-1, compare);
109     free(merged);
110 }
111 /*
112 * merge sort, recursive function
113 */
114
115 static void merge_sort_recur(
116     void **array, void **merged, int p, int r, int (*compare) (void *, void *)
117 ) {
118     int i, j, k, q;
119
120     if (p < r) {
121         q = (p + r) / 2;
122         merge_sort_recur(array, merged, p, q, compare);
123         merge_sort_recur(array, merged, q+1, r, compare);
124
125         for (i=k=p, j=q+1; i<=q && j<=r; ) {
126             if (compare(array[i], array[j]) < 0) {
127                 merged[k++] = array[i++];
128             } else {
129                 merged[k++] = array[j++];
130             }
131         }
132
133         while (i <= q) {
134             merged[k++] = array[i++];
135         }
136         while (j <= r) {
137             merged[k++] = array[j++];
138         }
139         for (k=p; k<=r; k++) {
140             array[k] = merged[k];
141         }
142     }
143 }
144
145 /*
146 * swap two elements of an array
147 */
148 static void swap (void **array, int i, int j) {
149     void * tmp;
150
151     if (i != j) {
152         tmp = array[i];
153         array[i] = array[j];
154         array[j] = tmp;
155     }

```

156 }

## 9.13 Post-fix Computations

### Specifications

Write a program able to implement a pocket computer adopting the *post-fix* (or post-order) notation. In the post-fix notation the operation symbol ('+', '−', '∗', and '/') follows the operands (i.e., "5 + 3" becomes "5 3 +").

The program receives on the command line a single string specifying the operation in post-fix notation. Operands and operators are space-separated. The result has to be printed-out on standard output. Suppose all operands are binary.

**Example 9.6** The following are correct parameters for the program:

```
"3 5 +"
"4 5 6 * 9 8 + - 3 + *
```

which has to compute  $5 + 3$  in the first case (printing-out 8), and  $4 \cdot (((5 \cdot 6) - (9 + 8)) + 3)$  (printing-out 80).

### Solution

#### File *client.c*

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <ctype.h>
4
5 #include "util.h"
6 #include "stackPublic.h"
7
8 int main (int argc, char *argv[]) {
9     long int result;
10    int left, right, length, k=0;
11    stack_t *sp=NULL;
12    char *expr;
13
14    util_check_m(argc>=2, "missing parameter.");
15    expr = argv[1];
16    length = strlen(expr);
17    sp = stack_init(length);
18
19    while (k < length) {
20        if (isdigit(expr[k])) {
21            sscanf(&expr[k], "%d", &result);
22            stack_push(sp, (void *)result);
23            while (isdigit(expr[k])) {
24                k++;
25            }
26        } else if (expr[k]=='+') || expr[k]=='*' || expr[k]=='-' || expr[k]=='/') {
27            stack_pop(sp, (void **)&right);
28            stack_pop(sp, (void **)&left);
29            switch (expr[k]) {
30                case '+': result = left+right; break;
31                case '*': result = left*right; break;
32                case '-': result = left-right; break;
33                case '/': result = left/right; break;
34            }
35        }
36    }
37    printf("%d\n", result);
38}
```

```
34     }
35     stack_push(sp, (void *)result);
36   }
37   k++;
38 }
39 stack_pop(sp, (void **)&result);
40 fprintf(stdout, "Result = %ld\n", result);
41 stack_dispose(sp, NULL);
42
43 return EXIT_SUCCESS;
44
45 }
```

# Index

- "?:" construct, 7
- Abstract Data Types, 372
- ADTs, 372
- Anagrams, 26, 276
- Arguments to main, 37, 116
- Arrangements with repetition, 265
- Arrays, 13, 14, 17, 20, 22, 24, 26, 49, 51, 242
  - Arrays of pointers, 74
  - Arrays versus lists, 169
  - Arrays versus pointers, 66
  - Arrays with variable size, 75
- auto, 353
- Automatic objects, 353
- Bi-linked lists, 181, 213
- Binary numbers, 14, 272
- Binary search, 253, 430
- Blocks, 2
- calloc, 79
- Cartesian points, 426
- Cast, 4
- Characters, 25
- Circular lists, 181
- Combinations with repetitions, 270
- Comments, 2
- Compiler, 362
- Conditional compilation, 3
- Constants, 5
- Data segment, 364
- Date, 424
- define, 3
- Dequeue, 391
- Divide and conquer, 236
- do-while repetition statement, 9
- Dummy nodes, 176
- Dynamic 1D versus 2D arrays, 110
- Dynamic arrays of structures, 90
- Dynamic Memory Allocation, 77
- Dynamic objects and modularity, 86, 114
- Dynamic objects and parameters, 119
- Dynamic strings, 88
- Dynamically allocated 1D arrays, 82
- Dynamically allocated 2D arrays, 110, 111
- Eight queen, 288
- Enqueue, 390
- exit, 38
- EXIT\_FAILURE, 12
- EXIT\_SUCCESS, 12
- extern, 355

- Extern objects, 355
- Factorial, 237
- Fibonacci numbers, 238
- FIFO structure, 387, 411
- FIFO structure on array, 388
- FIFO structure on dynamic list, 390
- Files, 43, 45, 46, 49
- for repetition statement, 8, 9
- free, 81
- Functions, 32, 36, 41, 49
- Functions as function arguments, 393
- Greatest Common Divisor (GCD), 246
- Handle, 373
- Heap segment, 365
- Hexadecimal number, 14
- Histogram, 22, 24
- if, 6
- ifndef-endif, 370
- include, 2
- Initialization, 359
- Knight tour, 284
- LIFO structure, 234, 383, 400
- LIFO structure on array, 384
- LIFO structure on dynamic list, 385
- Linkage , 360
- Linker, 362
- List dispose, 175
- List extraction, 173
- List extraction in an ordered list, 178
- List free, 175
- List insertion, 174
- List insertion in an ordered list, 180
- List library, 418
- List of lists, 182
- List search, 172
- List search in an ordered list, 178
- List visit, 171
- Lists, 170, 244
- Lists versus arrays, 169
- Loader, 362
- Macro definition, 395
- malloc, 78
- Matrices, 31, 32, 34, 41, 54
- Memory model, 59
- Merge sort, 255
- Multiple-source-file programs, 362
- Multiplication method, 262
- Mutual recursion, 233
- NULL, 65
- Octal numbers, 14
- Once-only headers, 370
- Ordered lists, 177
- Ordering, 49
- Padding, 60
- Partition of a set, 281
- Permutations with repetitions, 268
- Pointer arithmetic, 65
- Pointer operators, 62
- Pointer variables, 62
- Pointers versus arrays, 66
- Polish notation, 250
- Pop, 387
- Power computation, 241
- Power-set, 279
- Pre-processor, 362
- printf, 6
- Program memory layout, 364
- Program structure, 1
- Push, 385
- Queue, 387, 411
- Queue on array, 388
- Queue on dynamic list, 390
- Quick sort, 258
- realloc, 80
- Recurrences, 232
- Recursion, 231
- Recursion tree, 234
- Recursion versus iteration, 235
- Region count, 248
- register, 355
- Register variables, 355
- scanf, 5
- Scope, 359

Sentinels, 176  
Simple arrangements, 264  
Simple combinations, 269  
Simple permutation, 267  
`sizeof`, 5  
Stack, 234, 383, 400  
Stack on array, 384  
Stack on dynamic list, 385  
Stack segment, 365  
`static`, 357  
Static objects, 357  
Storage classes, 353  
 `strdup`, 89  
Strings, 25, 26, 29, 39  
`struct`, 48  
Structures, 48, 49, 51  
Structures and pointers, 73  
Sudoku, 54, 319

`switch`, 7  
Text segment, 364  
The Address Operator, 63  
The Indirection Operator, 62  
The `->` operator, 73  
Type, 4  
`typedef`, 4  
Utility library, 396  
Variables, 5, 60  
`void`, 65  
`while` repetition statement, 8, 9  
Word, 59  
Wrapper, 373

# Bibliography

- [1] S. NOCCO, S. QUER. *Guida alla Programmazione in Linguaggio C; Volume I: Fondamenti di programmazione.* CLUT (Coperativa Libraria Universitaria Torinese), Torino, Italia (2016). ISBN 978-88-7992-404-4.
- [2] N. WIRTH. *Algorithms + Data Structures = Programs.* Prentice Hall (1976). ISBN 978-0-13-022418-7.
- [3] D. E. KNUTH. *The Art of Computer Programming, Volume 1-4A.* Addison Wesley (1968-2005). ISBN 0-201-03801-3.
- [4] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, C. STEIN. *Introduction to Algorithms, Third Edition.* The Mit Press, Cambridge Massachusetts (2009). ISBN 978-0-262-53305-8.
- [5] R. SEDGEWICK. *Algorithms in C, Parts 1-4, Third Edition.* Addison-Wesley (1998). ISBN 0-201-31452-5.
- [6] R. SEDGEWICK. *Algorithms in C, Part 5, Third Edition.* Addison-Wesley (2002). ISBN 0-201-31663-3.
- [7] R. P. GRIMALDI. *Discrete and Combinatorial Mathematics, 5th Edition.* Pearson Education Limited, New York, NY, USA (2014). ISBN 1-292-02279-5.
- [8] B. W. KERNIGHAN, D. M. RITCHIE. *The C Programming Language, 2nd Edition.* Prentice Hall (1988). ISBN 978-0131103627.
- [9] P. J. DEITEL, H. DEITEL. *C How to Program.* Pearson Education, Prentice Hall (2015). ISBN 978-1292110974.
- [10] A. KELLEY, I. POHL. *A Book on C: Programming in C.* Addison Wesley (1998). ISBN 0-201-18399-4.
- [11] ISO/IEC 9899. 1990, *Information technology - Programming Languages - C.* International Organization for Standardization (ISO) (1990).
- [12] ISO/IEC 9899. 1999, *Information technology - Programming Languages - C.* International Organization for Standardization (ISO) (1999).
- [13] ISO/IEC 9899. 2011, *Information technology - Programming Languages - C.* International Organization for Standardization (ISO) (2011).