Chapter 2

The Imperative Programming Languages, C/C++

KH ALL RIGHTS RESERVED

As we discussed in Chapter 1, the imperative paradigm manipulates named data in a fully specified, fully controlled, and stepwise fashion. It focuses on how to do the job instead of what needs to be done. Imperative programs are algorithmic in nature: do this, do that, and then repeat the operation n times, etc., similar to the instruction manuals of our home appliances. The coincidence between the imperative paradigm and the algorithmic nature makes the imperative paradigm the most popular paradigm among all possible different ways of writing programs. Another major strength of the imperative paradigm is its resemblance to the native language of the computer (von Neumann machine), which makes it efficient to translate and execute the high-level language programs in the imperative paradigm.

In this chapter, we study the imperative programming languages C/C++. We will focus more on C and the non-object-oriented part of C++. We will study about the object-oriented part of C++ in the next chapter.

By the end of this chapter, you should

- have a solid understanding of the imperative paradigm;
- be able to apply the flow control structures of the C language to write programs;
- be able to explain the execution process of C programs on a computer;
- be able to write programs with complex control structures including conditional, loop structures, function call, parameter passing, and recursive structures;
- be able to write programs with complex data types, including arrays, pointers, structures, and collection of structures.

The chapter is organized as follows. Section 2.1 gives a quick tutorial on C/C++ so that students can start their laboratory work on C. Section 2.2 introduces the C/C++ control structures. Sections 2.3, 2.4, and 2.5 discuss data declaration; scope and basic data types; constant, array, pointer, and string; and type construction, including enumeration type, union type, structured types, and file types. Section 2.5 also presents several large program examples using array, pointer, and structures. Section 2.6 studies the functions, function calls, and parameter-passing mechanism in the C language. Section 2.7 teaches a unique technique of understanding recursion and writing recursive programs in four easy steps. Finally, Section 2.8 briefly discusses how to construct C programs into modules and how to use modules to form larger programs.

Imperative programming is largely based on computer architectures and assembly language programming. We will briefly discuss the basic computer architectures as the background material in Appendix A.

2.1 Getting started with C/C++

In this section, we first introduce how to write your first C/C++ program and how to perform input and output. You can develop your programs in different programming environments. Two of the most frequently used programming environments—GNU GCC and MS Visual Studio—are introduced in Appendix B.

2.1.1 Write your first C/C++ program

A C program consists of one or more functions. There are two kinds of functions:

- Built-in functions are prewritten and exist in **libraries**, for example, input and output functions (printf, scanf in C and cin, cout in C++), mathematical functions (abs, sin, cos, sqrt);
- User defined functions are written by the programmers.

The `main()` is a function that all C/C++ programs must have, which is the entry point of the programs (i.e., execution of all programs begin at the first statement of the main function). The shortest and simplest C/C++ program is:

```
  main() { }
```

Obviously, this program does nothing. Usually, `main` will have some statements and invoke other user written or library functions to perform some job. For example:

```
/* My first program, file name hello.c
  This program prints "hello world" on the screen */
#include <stdio.h>   // the library functions in stdio will be used
main( ) {
  printf("hello world\n");
}
```

The simple C program will call a library function `printf` to print

```
hello world
```

The first two lines are comments. There are two ways to write comments in C/C++. Multi-line comments can be quoted in a pair of `/*` and `*/`, while single line comments can simply follow double slashes `//`.

The third line of the program specifies what library package will be used. In this program, we use `printf` that is defined in the `stdio` package. In the print statement, "\n" is the "newline" control symbol that puts this output on a line by itself. Another useful control symbol is "\t" for tab.

A C/C++ function may return a value (like a Pascal function) or return no value (like a Pascal procedure). A function may take zero or a number of parameters. The following are several forms of the main function:

```
main () { ... }       // acceptable for C
void main( ) { ... } // in C++, void must be used.
int main() {... return 0;}
void main (int argc, char *argv [ ]) {...}
```

The first and the second forms do not require the function to return a value. The third form requires the function to return an integer value. The fourth form does not require a return value but requires parameter inputs.

KH ALL RIGHTS RESERVED

You may ask how do we or why do we need to pass values to the function that will be called before any other statements or functions are executed? The answer is that the parameters to the `main()` function allow it to take command line inputs, used to specify, for example, the name of a data file.

For example, if we compiled our first program `hello.c` into the executable code `hello.exe`, we can execute the program by simply typing the name `hello` and the `Enter` key. However, if we have a program, say, `letterReader.exe` that reads a text file, say, `letter.txt`, then we need to execute the program by typing

```
letterReader letter.txt
```

where the file name "`letter.txt`" will be passed to the main function of the program `letterReader` as a parameter.

Unlike Java, C/C++ functions and variables may exist outside any class or functions. These functions and variables are **global**. Function `main()` is always a global function.

2.1.2 Basic input and output functions

Generally, input in C/C++ is reading from a file and output is writing to a file. The keyboard is considered the standard input file and the monitor screen is considered the standard output file. The functions `getchar()` and `putchar(x)` are the basic I/O library functions in C. `getchar()` fetches one character from the keyboard every time it is called, and returns that character as the value of the function. After it reaches the last character of a file, it returns `EOF` (end of file), signifying the end of the file. On the other hand, `putchar(x)` prints one character (the character stored in variable `x`) on the screen every time it is called. The following program reads a line of characters from the keyboard and prints it on the screen. Since both standard input and output are in fact files, a similar program can be used to copy the contents of one file to the other.

```
#include <stdio.h>
main( ) {
   char c;              // declare c as a character type variable
   c = getchar( );      // input one character from the keyboard
   while (c != '\n') {     // while c ≠ the newline control symbol
        putchar(c);     // print to screen
        c = getchar( );    // input another character from the keyboard
   }
}
```

KH ALL RIGHTS RESERVED

To input a stream of characters, you can use fget:

char *fgets(char *tempstr, int n, FILE *inputfile)

where `tempstr` will point to the string read from the file pointer `inputfile`. Using this read operation, we can read from any text file. For now, we will consider the input file is the keyboard and the file name is stdin. The int variable `n` is the maximum number of characters (bytes) we want to read. The operation `fgets` will stop when `n-1` characters are read or a newline character is read. It returns `null` if nothing is read into the tempstr. Otherwise, it will return the value of `tempstr`.

The following snippet of code shows an example of using fgets.

```
char tempstr[256];
char name[32]; char breed[32]; char owner[32];
```

```
printf("Please enter the dog's info in the following format:\n");
printf("name:breed:owner\n");
fgets(input, sizeof(tempstr), stdin); // read from keyboard
// change '\n' char attached to tempstr into null terminator
tempstr[strlen(tempstr) - 1] = '\0';
char* name = strtok(tempstr, ":");  // parse to ":"
char* breed = strtok(NULL, ":");    // remove separator
char* breed = strtok(tempstr, ":"); // parse to ":"
char* owner = strtok(NULL, ":");    // remove separator
```

where strtok function is used to parse the string and extract the part of the string separated by a separator. In this example, `":"` is used. You can use other separators, such as `" " or ","`. In the program, function call `strtok(tempstr, ":")` will read `tempstr` upto `":"`, while `strtok(NULL, ":")` will remove `":"` and return the remaining string.

2.1.3 Formatted input and output functions

The basic input/output functions allow us to read and write a character at a time. They cannot be used to read and write other types of variables and cannot control the format of output.

The formatted input/output functions are `printf` and `scanf` that take an argument for formatting information. The following program demonstrates a simple use of the functions.

```
/* The program takes a number from the keyboard, processes the number,
 and then prints the result. */
#include <stdio.h>
main () {
  int i;        // i is an integer type variable
  float n = 5.0;// n is floating-point type and is initialized to 5.0
  printf("Please enter an integer\n");
  scanf("%d", &i);   // An integer is expected from the keyboard
  if (i > n)
       n = n + i;
  else
       n = n - i;
  printf("i = %d\t n = %f\n", i, n); //%d, \t, %f, and \n control formats
}
```

Assume a number 12 is entered when `scanf` is executed; the output of the program is

```
i = 12 n = 17.0
```

Generally, the formats of `scanf` and `printf` are

```
scanf ("control sequence", &variable1, &variable2, ... &variablek);
printf ("control sequence", expressions);
```

In the `scanf` function, the ampersand "`&`" is the address-of operator that returns the address of the variable. Using the address-of operator in the argument of a function (e.g., `&i` in `scanf`) enforces the parameter passing by reference. Parameter-passing mechanisms will be explained in detail later in Section 2.6.

KH ALL RIGHTS RESERVED

In the `printf` function, the "`expressions`" is a list of expressions whose values are to be printed out. Each expression is separated by a comma.

The control sequence includes constant strings to be printed (e.g., "`i =`"), and control symbols to be used to convert the input/output from their numeric values that are stored in the computer to their character format displayed. The control symbol "`%d`" in the `scanf` and `printf` signifies that the next argument in the argument list is to be interpreted as a decimal number and "`%f`" signifies that the next argument is to be interpreted as a floating-point number. The other control characters include "`%c`" for characters and "`%s`" for strings of characters. The symbols "`\n`" and "`\t`" signify the "`newline`" that puts the next output on a new line, and "`tab`" puts the next output after a tab. If there is no "`newline`" or "`tab`" at the end of the first output line, successive calls to `printf` (or `putchar`) will simply append the string or character to the previous output line.

In C++, a different library package and different I/O functions are used. When you use C++ specific features, your program name must have an extension `.cpp` for C++ program. If you use extension `.c`, your program will be considered to be a C program only, and you will obtain compilation errors for C++ specific features.

```
#include <iostream>  // iostream is the C++ library I/O package
using namespace std;
void main() {
   int i, j, sum;        // declaration
   cout << "Enter an integer" << endl;  // prompt for input
   cin >> i;          // read an integer and put in variable i
   cout << "Enter an integer" << endl;
   cin >> j;          // read an integer and put in variable j
   sum = i + j;
   cout << "Sum is " << sum << endl; // print sum
}
```

KH ALL RIGHTS RESERVED

A scenario of execution of the program is

```
Enter an integer
5
Enter an integer
7
Sum is 12
```

In the program, the functions `cin` and `cout` are C++ standard input and output functions. The function `endl` is the C++ `newline` function corresponding to C's "`\n`".

In C-formatted I/O, a programmer must specify the type of variables to be printed. In C++, the types are automatically recognized. This improvement simplifies printing statements in most cases. Unfortunately, we still have situations where we have to tell the program what type of data to print. For example, a character type in C/C++ is the same as an integer type. How do we tell the program that we want to print a character or an integer? The solution is type casting. The following example shows how a character type variable `c`, initialized to 68 and corresponding to the ASCII character "`D`," is printed as integer 68 and as character "`D`" using `printf` and `std::cout`, respectively. The ASCII table is given in Appendix C.

```
#include <iostream>
using namespace std;
```

```
void main(void) {
  char c = 68;
  printf("c = %d", c);
  printf("\tc = %c\n", c);
  cout<<"c = "<<(int) c;
  cout<<"\tc = "<<c<<endl;
}
```

The output of the program is

```
c = 68 c = D
c = 68 c = D
```

Please note that C++ I/O package `<iostream>` contains all C-styled I/O functions and control symbols like `printf`, `scanf`, "`\n`" and "`\t`."

2.2 Control structures in C/C++

In this section, we briefly review the basic control structures in C/C++, which are similar in all imperative programming languages. The topics we will discuss are

- operators and the order of evaluation,
- basic selection structures,
- multiple selection structures, and
- iteration structures.

Recursion structures are much more complex and will be discussed in Section 2.7 in detail. Chapters 4 and 5 will have even more discussion on this topic.

2.2.1 Operators and the order of evaluation

C/C++ provides a set of **operators** to allow programmers to write complex arithmetic and logical expressions. The **precedence** and **associativity** of C/C++ operators affect the grouping and evaluation of operands in expressions. Table 2.1 summarizes the precedence and associativity (the order in which the operands are evaluated) of C operators, listing them in the order of precedence from highest to lowest. Operators with higher precedence are evaluated first. If two operators have equal precedence (they appear at the same level in the table), they are evaluated according to their associativity, either from right to left or from left to right, as defined in the right column of the table.

Table B.4 in Appendix B gives a complete list of the C/C++ operators, their precedence, description, and associativity.

Please note that C/C++ use a **lazy evaluation** policy; that is, an expression will be evaluated only if its value is needed. For example, if we have an expression

```
(i == 0) && j++
```

the second operand, `j++`, will be evaluated only if `i == 0` is true (nonzero). Thus, `j` will not be incremented if `i == 0` is false (0).

KH ALL RIGHTS RESERVED

<table>
<tr><th>Operators</th><th>Type of operation</th><th>Associativity</th></tr>
<tr><td>[ ] ( ) . –> postfix ++ and postfix —</td><td>Expression</td><td>Left to right</td></tr>
<tr><td>prefix ++ and prefix — sizeof & * + – ~ !</td><td>Unary</td><td>Right to left</td></tr>
<tr><td>Typecasts</td><td>Unary</td><td>Right to left</td></tr>
<tr><td>* / %</td><td>Multiplicative</td><td>Left to right</td></tr>
<tr><td>+ –</td><td>Additive</td><td>Left to right</td></tr>
<tr><td><<>></td><td>Logical bitwise shift</td><td>Left to right</td></tr>
<tr><td><><= >=</td><td>Relational</td><td>Left to right</td></tr>
<tr><td>== !=</td><td>Equality</td><td>Left to right</td></tr>
<tr><td>&</td><td>Bitwise-AND</td><td>Left to right</td></tr>
<tr><td>^</td><td>Bitwise-exclusive-OR</td><td>Left to right</td></tr>
<tr><td>|</td><td>Bitwise-inclusive-OR</td><td>Left to right</td></tr>
<tr><td>&&</td><td>Logical-AND</td><td>Left to right</td></tr>
<tr><td>||</td><td>Logical-OR</td><td>Left to right</td></tr>
<tr><td>? :</td><td>Conditional-expression</td><td>Right to left</td></tr>
<tr><td>= *= /= %= += –= <<= >>= &= ^= |=</td><td>Assignment</td><td>Right to left</td></tr>
<tr><td>,</td><td>Sequential evaluation</td><td>Left to right</td></tr>
</table>

KH ALL RIGHTS RESERVED

Table 2.1. C/C++ operators and their precedence.

2.2.2 Basic selection structures (if-then-else and the conditional expression)

The **basic selection structure** in C/C++ is implemented by `if-then` and `if-then-else` statements, which can be defined by the syntax graph in Figure 2.1.

Figure 2.1. Syntax graph for `if-then-else` in C/C++.

In the syntax graph, `<block1>` and `<block2>` contain zero, one, or a block of statements. A block of statements are enclosed within curly braces, which can contain zero, one, or multiple statements. Local declaration can be given in each block. The `<condition>` is any expression that evaluates to integer value. If the expression evaluates to 0 (considered "false"), `<block2>` will be executed, otherwise (any nonzero value will be considered "true"), `<block1>` will be executed. In the syntax graph, we omitted the arrow before the keyword **`if`**. In C, there is no Boolean type, while in C++ a Boolean type is predefined.

The following example illustrates the use of conditional structure and logic and relational operators.

```
if (a == b && c <= d)
   x = 0;
else {
   x = 1;
   y = 2;
```

```
}
```

The character sequence "&&" is a **logical operator** for AND. The character sequences "==" and "<=" are called **relational operators**. A complete set of both arithmetic and logical operators is given in Table 2.1.

C/C++ also provides a ternary **conditional operator** "?:" to form a **conditional expression**. The conditional operator takes three operands and performs a similar selection function as an if-then-else statement. The general form of the conditional operator is given in Figure 2.2.

In the syntax graph below, <operand1> and <operand2> can be any expression that returns a value or a simple assignment statement, in which case, the assigned value will be considered the return value of the operand. When the conditional expression is executed, the <condition> is first tested. If it returns a nonzero or true value, <operand1> will be evaluated; otherwise, <operand2> will be evaluated.

→ **(** → <condition> → **?** → <operand1> → **:** →<operand2> → **)** →; →

Figure 2.2. Syntax graph for the conditional operator in C/C++.

The following example illustrates different ways of using the conditional expression and their effects. A conditional expression can be used as an expression in the right-hand side of an assignment statement or as a stand-alone statement.

```
i = 0; j = 0;             // i = 0 and j = 0 represent false value
(i? i=5 : i=9);           // 9 will be assigned to i;
k = (j? i=5 : j=9);       // 9 will be assigned to j and to k
i = 1; j = 2              // i ≠ 0 and j ≠ 0 represent true value
(i? i=5 : i =9);          // 5 will be assigned to i;
k = (j? j=5 : j=9);       // 5 will be assigned to j and to k
```

KH ALL RIGHTS RESERVED

2.2.3 Multiple selection structure (switch)

The basic selection statements select one out of two cases. If we have multiple choices, we need to use nested if-then-else statements. For example:

```
if (ch == '+')   x = a + b;
  else if (ch == '-')    x = a - b;
  else if (ch == '*')    x = a * b;
  else if (ch == '/')    x = a / b;
  else printf("invalid operator");
```

It is often more convenient in such situations to use a multiple selection statement switch, as defined in Figure 2.3.

The case statements label different actions we want to execute. The loop in the definition signifies that we can have any number of case statements (the number must be greater than or equal to one). The break statements, which exit the switch construct if a case is satisfied, are optional (there is a bypass route). The default case is performed if none of the other cases is satisfied. According to the definition, default is optional (there is a bypass route). If default is not included and none of the cases match, no action will be executed. For example, the following piece of code selects one of the four operations.

```
switch (ch) {
  case '+': x = a + b; break;
```

```
    case '-': x = a - b; break;
    case '*': x = a * b; break;
    case '/': x = a / b; break;
    default: printf("invalid operator");
}
```

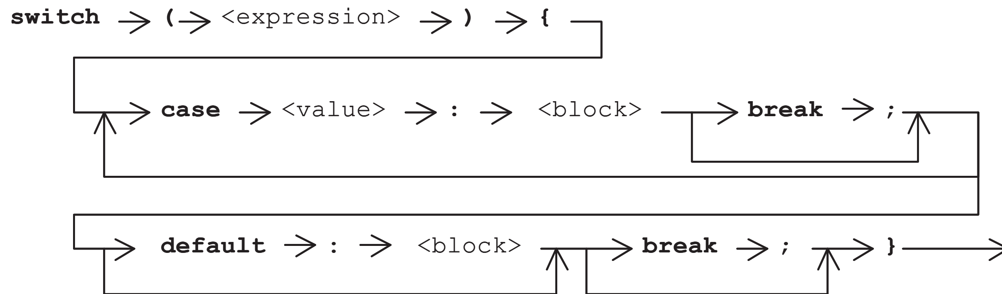


Figure 2.3. Syntax graph for `switch` in C/C++.

Including the `break` statements in the code is not an efficiency issue, as many people believe. What would happen if any one of the four `break` statements is omitted? Examine the following program without the `break` statements.

```
/* This C program demonstrates the switch statement without using breaks.
   The program is tested on MS Visual C++ platform */
#include <stdio.h>
void main() {
   char ch = '+';
   int f, a=10, b=20;
   printf("ch = %c\n", ch);
   switch (ch) {
        case '+': f = a + b; printf("f = %d\n", f);
        case '-': f = a - b; printf("f = %d\n", f);
        case '*': f = a * b; printf("f = %d\n", f);
        case '/': f = a / b; printf("f = %d\n", f);
        default: printf("invalid operator\n");
   }
   ch = '-';
   printf("ch = %c\n", ch);
   switch (ch) {
        case '+': f = a + b; printf("f = %d\n", f);
        case '-': f = a - b; printf("f = %d\n", f);
        case '*': f = a * b; printf("f = %d\n", f);
        case '/': f = a / b; printf("f = %d\n", f);
        default: printf("invalid operator\n");
   }
   ch = '*';
   printf("ch = %c\n", ch);
```

KH ALL RIGHTS RESERVED

```
    switch (ch) {
        case '+': f = a + b; printf("f = %d\n", f);
        case '-': f = a - b; printf("f = %d\n", f);
        case '*': f = a * b; printf("f = %d\n", f);
        case '/': f = a / b; printf("f = %d\n", f);
        default: printf("invalid operator\n");
    }
    ch = '/';
    printf("ch = %c\n", ch);
    switch (ch) {
        case '+': f = a + b; printf("f = %d\n", f);
        case '-': f = a - b; printf("f = %d\n", f);
        case '*': f = a * b; printf("f = %d\n", f);
        case '/': f = a / b; printf("f = %d\n", f);
        default: printf("invalid operator\n");
    }
    ch = '%';
    printf("ch = %c\n", ch);
    switch (ch) {
        case '+': f = a + b; printf("f = %d\n", f);
        case '-': f = a - b; printf("f = %d\n", f);
        case '*': f = a * b; printf("f = %d\n", f);
        case '/': f = a / b; printf("f = %d\n", f);
        default: printf("invalid operator\n");
    }
}
```

KH ALL RIGHTS RESERVED

The `switch` statements in this program are all syntactically correct, but they do not implement the selection at all. The omission of the `break` statements leads to the "fall through" execution of all the following cases of statements. The output of the program is

```
ch = +
f = 30
f = -10
f = 200
f = 0
invalid operator
ch = -
f = -10
f = 200
f = 0
invalid operator
ch = *
f = 200
f = 0
```

```
invalid operator
ch = /
f = 0
invalid operator
ch = %
invalid operator
```

This rather "unexpected" output is due to the "jump-table" implementation of `switch` statements at the assembly language level as shown in Figure 2.4.

The variable `ch` will be compared with the label values stored in the **jump-table**. If a match is found, the control will jump to the right address of the statement-table. Obviously, if no break statement appears at the end of each case, the machine would continue to execute the next statement. In some languages (e.g., Pascal), the compiler automatically adds a break statement at the end of each case. The advantage is the elimination of a possible error source and the drawback is that the programmer loses a bit of writability—in a rare case, a programmer may want to execute all the following cases once a condition is met.

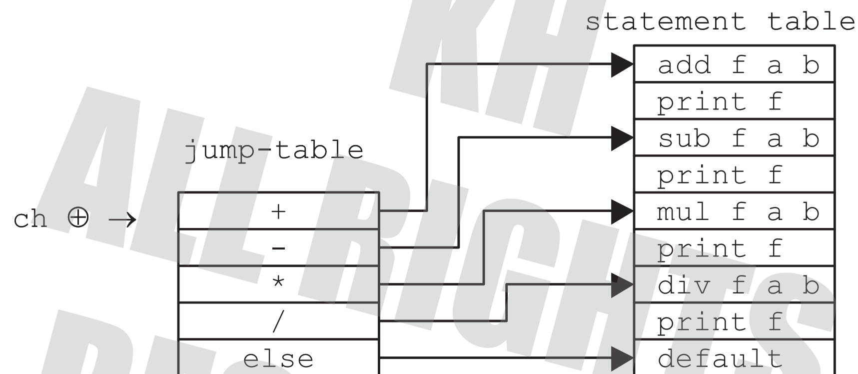


Figure 2.4. The assembly language level implementation of the `switch` statement.

KH ALL RIGHTS RESERVED

2.2.4 Iteration structures (while, do-while, and for)

The basic looping structure in C/C++ is the **while-loop**. The syntax graph of a while-loop is given in Figure 2.5.

while → **(** → <condition> → **)** → <block> →

Figure 2.5 Syntax graph for `while` statement in C/C++.

In a while-loop, the `block` of statements, called **loop body**, will execute repeatedly as long as the `condition` statement produces a `true` (nonzero) value. However, there is no looping in the syntax definition. This is because there is only a semantic level looping for the `while`-statement, but no looping at the syntactic level. This is also true for the `for`-loop. On the other hand, there is a looping structure in the syntax graph of the `switch` statement, but there is no looping for the statement at the semantic level. The following program counts the number of inputs that are greater than 90. The program stops when a negative number is entered.

```
#include <stdio.h>
main () {
   int i, c = 0;
```

```
   scanf("%d", &i);
   while( i >= 0 ) {
        if (i > 90)
              c++;      // counting: same as c = c + 1;
        scanf("%d", &i);
   }
}
```

A variation of the while-loop is the **do-while-loop** that tests the condition after the loop body has been executed once. Using a do-while-loop, we need only one scanf statement for the example above:

```
#include <stdio.h>
main () {
   int i, c = 0;
   do { scanf("%d", &i);
        if (i > 90)
              c++;
   }
   while (i >= 0);
}
```

The for-**loop** can be considered a more compact form of the while-loop. It allows us to put the initialization, condition-test, and increment parts of a loop in a single statement. The syntax graph of the for-loop is given in Figure 2.6.

In the syntax graph, the <initialization> and <increment> can be single statements or multiple statements separated by commas. The function of the for-loop is equivalent to the function of the following code with a while-loop:

```
<initialization>
while(<condition>) {
   <block>
   <increment>
}
```

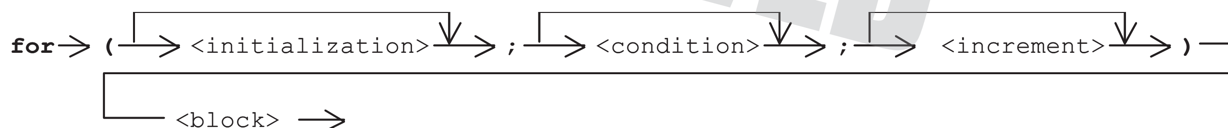


Figure 2.6. Syntax graph for the for statement in C/C++.

For example, the following program does a similar job as the program with a while-loop, except that the program with a while-loop terminates if a negative number is entered, while the following program terminates when exactly n numbers are entered.

```
main () {
   int i, k, n = 10, c=0;
   for(k=0; k<n; k++) {
        scanf("%d", &i);
        if (i > 90)
```

KH
ALL RIGHTS
RESERVED

```
            c++;
    }
}
```

All three components in the parentheses of a `for-loop` are optional. A `for-loop` with all three components absent creates an infinite loop:

```
  for(;;) <block>
```

In some programming languages (e.g., Pascal), the loop iteration variable `k` and the loop boundary variable `n` may not be modified in the loop body, which means that the `for-loop` can only iterate a fixed number of times. In C/C++, both variables can be modified and thus the `for`-loop can iterate a variable number of times. However, it is not a good programming practice to modify any of the two variables even if we are allowed to modify them. Normally, we use a `while-loop` or a `do-while-loop` if the number of iterations is unknown and we use a `for-loop` if the number of iterations is fixed.

2.3 Data and basic data types in C/C++

The key concepts of data in a programming language include:

- Type: What values and operations are allowed on the type of data?
- Location: Where is data stored in memory?
- Address/Reference (of location in memory): How do we find the location where a particular piece of data is stored?
- Name: How do we conveniently access the locations of data?
- Value: What is stored in a memory location?
- Scope (visibility and lifetime): Where and when is a piece of data visible or accessible?

We will look at these concepts while studying basic data types in C/C++.

2.3.1 Declaration of variables and functions

At machine level, all data and instructions are stored in memory locations in sequences of binary bits: 001011. It is up to the programmer to manage and interpret the bit patterns.

A variable **declaration** in a high-level programming language binds a name to a location in memory and describes the attributes of the value in the location, so that the programmer can use the name to access the memory location and the value stored in the location conveniently. A variable declaration describes the following attributes of the value:

- type
- scope
- qualifier (modifiability, e.g., constant)
- variable initialization

Typically, the compiler allocates memory for the variable and binds the name to that location when a variable is declared.

The general form of variable declaration in C/C++ is

```
qualifier typename variable_names separated by a comma.
```

KH ALL RIGHTS RESERVED

For example:

```
int i = 0, j, k;
const double pi = 3.1415926;
float x = 3.0, y, z = 2.5;
```

The general form of function declaration in C/C++ is

```
typename function_name(typename name, ..., typename name){ <body> }
```

The `typename` before the `function_name` specifies the return-value type of the function. The list in the parentheses is the list of parameters with their types. If a function does not return a value, we can either write the type name `void` or write nothing. Similarly, if a function does not have any parameter, we can either write void or nothing in the parentheses.

For example, the following program declares a `max()` function that returns the larger value between two parameter values. The function is called twice in the `main()` function.

```
#include <stdio.h>
int max(int first, int second){      // function declaration
  if (first > second)
      return first;
  else return second;
}
void main (void) {       // main function
  int i = 7, j = 5, k = 12, f;
  f = max(i, j);     // function call
  f = max(k, f);     // call the function again with different parameters
}
```

2.3.2 Scope rule

The **scope rule** of a C/C++ declaration: The scope of a variable is from its declaration to the end of the block defined by a pair of curly braces. The idea of the scope rule is declare-before-use: any variables or functions must be declared before they can be used. For example:

```
{
  int height = 6; int width = 6;
  int area = height*width;
  . . .
} // block ends
```

In this example, the variable `area` is initialized to `height*width`, which are declared just before the `area` is declared. According to the scope rule, the declaration is correct. On the other hand, if we swap the order of the first two lines

```
{
  int area = height*width;
  int height = 6; int width = 6;
  . . .
} // block ends
```

KH ALL RIGHTS RESERVED

we will have a compilation error complaining that `height` and `width` are not declared when their values are used.

There is a subtle difference between the scope rules of an imperative language and a functional language. In a functional language, the scope rule normally says that “the scope of a variable is within the block in which the variable is declared/defined.” Should the scope rule of C/C++ say that “the scope of a variable is within the block in which the variable is declared,” no compilation error would occur if the declaration of variable `area` is placed before `height` and `width`.

The declare-before-use principle is simple to understand and use for variables, but may cause problems for declarations of mutually recursive functions. For example, a function `F` calls function `G` and function `G` calls function `F`. In this case, which function should be declared first?

There are two possible solutions to this dilemma:

Multi-scan compilation: The compiler scans the program multiple times. For example, in the first round of scan, all names (variables and functions) are stored in a name table, and in the second round of scan, binding between names and memory locations is made.

Forward declaration: Each function is declared in two steps: a forward declaration and a genuine declaration. The forward declaration makes a name known in advance (before it is used) and thus needs to specify only the return type, function name, parameter types, and parameter names (parameter names are optional). In the following program segment, for example, function `bar` calls function `foo` and function `foo` calls function `bar`. In such a case, we cannot satisfy the declare-before-use requirement without using forward declaration.

```
void bar(float, char);   // forward declaration to satisfy scope rule
int foo(void);           // forward declare all functions
...
int foo(void) {          // genuine declaration
   . . .
   bar(2.5, '+');        // call function bar()
   . . .
}
void bar(float f, char c) {  // genuine declaration
   . . .
   k = foo();            // call function foo()
   . . .
}
```

Most C/C++ compilers today use the multi-scan technique and thus forward declaration is not necessary for mutually recursive functions. However, forward declaration is still frequently used for two reasons:

- To make the program independent of the compiler
- Better readability: The forward declarations serve as an index to (overview of) all functions

2.3.3 Basic data types

C defines five basic **data types**, sometimes called value types. They are:

- Character (char)
- Integer (int)

KH ALL RIGHTS RESERVED

- Floating-point (float)
- Double precision floating-point (double)
- Valueless (void)

C++ adds two more basic data types:

- Boolean (bool)
- Wide-character (wchar_t)

There is no Boolean type in C. The logic values are represented by integer: 0 for `false` and any other value will be interpreted as `true`. The character type in C is based on the 7-bit ASCII code, which allows 128 characters. C++'s wide-character type is based on the 16-bit Unicode, which allows $2^{16} = 65,536$ characters. Java's character type is also based on the Unicode.

Several of these basic types can be modified using one or more of these modifiers:

```
signed, unsigned, short, long, register
```

The type modifiers `signed` and `unsigned` explicitly specify that the integer type is signed and unsigned, respectively, although by default an integer type without any modifier is signed. For a signed integer, a "1" at the most significant bit indicates a negative number, while for an unsigned integer, no bit is used for the sign and only nonnegative numbers can be represented. Thus a "1" at the most significant bit indicates a large positive number. The type modifiers `short` and `long` indicate the data ranges of the integers of these types. It is more efficient to specify a short integer if you know your integer will not be very large. The type modifier `register` suggests to the compiler that the programmer wants to access the variable as fast as possible. Obviously, a variable can be accessed in the fastest way if it is put in a register. Since a processor has a very limited number of registers, you should use the `register` modifier sparingly. The `register` modifier is normally used for variables that need to be accessed frequently in a short period of time, such as loop variables. Please note that the `register` modifier is only a suggestion to the compiler. The compiler will take it into consideration where it is possible. However, there is no guarantee that the compiler can keep the variable in a register longer than the other variables.

Table 2.2 summarizes the basic data types available in C/C++. Since C/C++ can be implemented on machines of different sizes (e.g., word length = 8, 16, 32, and 64), the number of bits used to implement a particular data type can vary. However, the language requires that a minimum number of bits must be guaranteed for each data type. The larger machines can use more bits to provide extra data range and/or higher precision. The second and third columns of the table list the guaranteed minimum number of bits and the minimum data range for each of the data types.

To find the exact size of each type on a particular machine, you can call the `sizeof` function using the type name as the parameter `sizeof(type_name)`. For example:

```
printf("size of long type = %d\n", sizeof(long));
```

will print the "`size of long type = 4`" if the machine uses 4 bytes to store a `long` integer. If we call

```
printf("bool-size = %d, true = %d, false = %d\n", sizeof(bool), true,
false);
```

The output would be

```
bool-size = 1, true = 1, false = 0
```

which means C++ uses one byte to store a `bool` type variable, the internal value of `true` is 1 and the internal value of `false` is 0.

KH ALL RIGHTS RESERVED

Type	Minimum bits	Minimum range
`bool`(C++ only)	1	true/false
char	8	from −127 to 127
signed char	8	from −127 to 127
unsigned char	8	from 0 to 255
`wchar_t`(C++ only)	16	from 0 to 65 535
int	16	from −32 768 to 32 768
signed int	16	same as `int`
unsigned int	16	from 0 to 65 535
short int	16	from −32 768 to 32 768
signed short int	16	same as `short int`
unsigned short int	16	same as unsigned int
long int	32	±2 147 483 647
signed long int	32	same as `long int`
unsigned long int	32	from 0 to 4 294 967 295
float	32	6 decimal digits of precision
double	64	10 decimal digits of precision

Table 2.2. Basic data types in C/C++.

KH ALL RIGHTS RESERVED

To see the relationship among the types, we can classify the data types into four categories: scalar, function, aggregate, and valueless (`void`) types, as shown in Figure 2.7. The scalar types can be further divided into pointer, arithmetic, and enumeration types. In the diagram, boldfaced words are keywords, and italic words are optional keywords. Other names are generic terms. The basic data types we discuss in this section belong to arithmetic types. Functions are considered a special data type. In the following sections, we will discuss pointer, enumeration, and aggregated types.

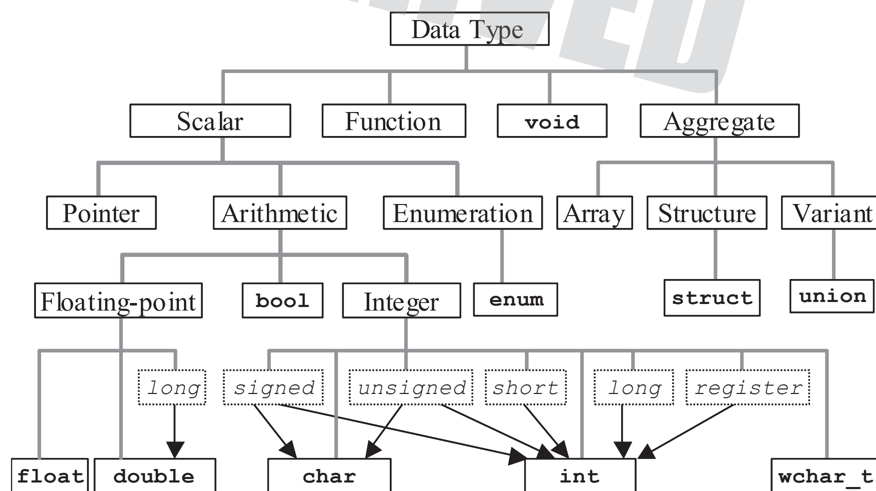


Figure 2.7. Classification of data types.

2.4 Complex types

In the previous section, we discussed basic data types. In this section, we discuss more complex data types including array, string as array of characters, pointer, constant, and enumeration.

2.4.1 Array

Array is a homogeneous collection of data elements that are stored in a consecutive block of memory locations. At the assembly language level, we use the initial address of the block plus the offset (index) of the element to access a particular element. At the high-level language level, we use the array variable name and the index to access an array element. An array is declared by

```
typename variablename[length] = {v_0, v_1, v_2, ..., v_length-1};
```

The `length` and the initialization part, $= \{v_0, v_1, v_2, ..., v_{length-1}\}$, are optional, which produces four possible combinations:

```
1. typename variablename[length];
2. typename variablename[] = {v_0, v_1, v_2, ..., v_length-1};
3. typename variablename[];
4. typename variablename[length] = {v_0, v_1, v_2, ..., v_length-1};
```

The first two array declarations are correct and are most frequently used. In the first declaration, the array variable and its length are declared. However, array elements are not initialized. In the second declaration, the length of the array is implied by the number of elements in the initialization list.

The third array declaration will immediately cause a compilation error because the compiler needs to know the size of the array to allocate the right amount of memory space for the array, and the size of the array is missing in the declaration.

The fourth declaration is syntactically correct, but one can easily make a contextual error in using this declaration! There are three possible cases when we use both explicit and implicit mechanisms to specify the length of the array:

- If length = n (the number of elements given in the initialization list), no problem will occur. However, this case is exactly the same as the second way of declaration.
- If length < n, a compilation (contextual) error will occur: there are not enough places to hold the elements given in the list.
- If length > n, no compilation error will occur. The n elements in the initialization list will be put in the first n places 0, 1, 2, ..., n−1. This is a case that is not covered by the first two ways of array declaration. Maybe this is the only case where we really need to use the fourth way of array declaration.

In the declaration of arrays, the `length` must be an integer value or a simple expression with integer operations like 20+5−1. It cannot contain a variable, even if the variable has been initialized.

The following piece of code illustrates the different ways of array declaration.

```
void main() {
  int a[3], sa, sb, sc, sd;  // a is correctly declared without
initialization
  int b[] = {2, 3, 9, 4};    // b is correctly declared and initialized
  int c[2] = {15, 14};       // c is correct, but the length is unnecessary
  int d[5] = {15, 14, 18};   // the first 3 elements of d are initialized
```

KH
ALL RIGHTS
RESERVED

```
//int d1[2] = {15, 14, 18};  // incorrect: not enough places
//int e[];                   // incorrect: no length indication
  a[0] = 20;                 // array index always starts from 0
  a[1] = 30;
  a[2] = 90;
  sa = sizeof(a);            // number of bytes used by a is 12
  sb = sizeof(b);            // number of bytes used by b is 16
  sc = sizeof(c);            // number of bytes used by c is 8
  sd = sizeof(d);            // number of bytes used by d is 20
  printf("sa = %d\t sb = %d\t sc = %d\t sd = %d\n", sa, sb, sc, sd);
  printf("d0 = %d\t d1 = %d\t d2 = %d\t d3 = %d\t d4 = %d\n",
           d[0], d[1], d[2], d[3], d[4]);}
```

The output of the program is

```
sa = 12     sb = 16  sc = 8   sd = 20
d0 = 15     d1 = 14  d2 = 18 d3 = 0  d4 = 0
```

KH ALL RIGHTS RESERVED

The first four lines of comments explain the different ways of declaration. The two incorrect declarations are commented out so that the program can be compiled and executed.

In the program, the system function `sizeof` returns the number of bytes (a byte = 8 bits) of the variable. The program is compiled on a 32-bit PC, an integer type variable takes 32 bits (4 bytes). If you compile the same program on a different machine (e.g., a 16-bit or 64-bit machine), the `sizeof` function will return different integer sizes. In Table 2.2, the minimum integer size given is 16 bits (2 bytes). When you write C/C++ programs, you need to handle the word length of the machine on which your program runs. You can use the `sizeof` function to find the word length of your computer and use the `sizeof` function to make your program independent of the word length. More uses of the `sizeof` function will be seen later in the text.

In Java, array declaration is different: We can declare an array without indicating its size and later give the size when we create the array object during the execution. This way of memory allocation is called **dynamic memory allocation**. The array declaration we discussed in this section is based on the **static memory allocation** by the compiler. However, C/C++ does provide the dynamic memory allocation mechanism for array and other structured data types. This will be discussed in conjunction with the pointer type.

We can define an array of `int`, `char`, and `float`, etc. Can we have an **array of arrays**? The answer is yes. C and C++ use array of arrays to represent **multidimensional arrays**. Array of arrays are declared and initialized like this:

```
char mac[5][7];
int mai[2][3] = {{4, 2, 3}, {7, 8, 9}};
```

Conceptually, array `mai` is stored in a matrix of 2 by 3, and its elements are accessed using the array name and the two indices `mai[i][j]`. Structurally, array `mai` is stored in a block of consecutive memory locations like this:

4	2	3	7	8	9

The following program illustrates the use of multidimensional arrays. Please note that `maxrow` and `maxcolumn` are defined as macros. The compiler would not accept the declaration of the `maze[maxrow][maxcolumn+1]` if they were defined as constant variables by using "`const`."

```
#define maxrow 50
#define maxcolumn 100
#include <stdio.h>
//const int maxrow = 100, maxcolumn = 100;
char ma[maxrow][maxcolumn+1];
void main(void) {
   int i, j;
   for (i=0; i < maxrow; i++)
        for (j = 0; j < maxcolumn + 1; j++)
             ma[i][j] = 'x';
}
```

2.4.2 Pointer

Pointer type is the most challenging data type in C/C++. This is especially true for Java programmers. Pointers provide programmers flexibility in accessing memory locations and modifying their values. On the other hand, the flexibility can easily create incorrect programs due to lack of understanding of computer organization and the relationships among different data types. This section will explain the principle of pointer type and the correct ways of using pointer variables.

We start by exploring different aspects of a **variable**:

- **Value:** A variable will hold a single value or a set of values. For example, an integer variable holds a single value and an array variable holds a set of values. A value can appear on the right-hand side of an assignment statement only and thus is called an **r-value** (for right-hand-side value).
- **Location:** A variable will be associated with a location or a set of memory locations. The value of a variable is stored in the location.
- **Address:** The address of a variable is a natural number directly associated with a memory location by the hardware. The address provides a direct way for programmers to access (read or write) a memory location or variable. The address refers to the literal number and thus is also an r-value.
- **Name:** The name of a variable is a mnemonic symbol that provides a convenient way for programmers to access a memory location or variable. A name is associated with a memory location (or translated into the address of the location) by the compiler. Some languages only use names to access memory locations (e.g., Java). Some languages allow using both names and addresses to access memory locations (e.g., C/C++). A variable name can appear in the left-hand side and right-hand side of an assignment statement and is called **l-value** (for left-hand-side value). A variable name has two faces: If it is used on the left-hand side of an assignment, it refers to the memory location. If it is used on the right-hand side of an assignment, it refers to the value stored in the memory location.

We can use an analogy to understand these aspects. Consider the soccer teams attending the World Cup. Each team consists of a number of members, corresponding to the set of values stored in a variable. Each team member will stay in a location (i.e., a hotel room). Each location will have a unique address (i.e., street address of the hotel plus room number). The team and each team member can be accessed by the address. In computer memory, the set of values related to a variable is normally stored in a consecutive block of memory locations, and thus, we can use the initial address of the block to access the values starting at that address. The hotel and rooms may also have names. If the context (scope) is clear, we can also use the name of the hotel and the name of a room to access a team member staying in the room. Read Appendix A, Section A.2 for a more detailed example.

KH ALL RIGHTS RESERVED

Why do we need the name if we have the address of a memory location? Humans are better at reading, understanding, and remembering names than long tedious numbers.

Why do we need addresses in high-level language programming if we have names? The reasons are twofold. First, every memory location in a computer has an address, but not every memory location has a name. We can use addresses to access unnamed variables. Second, memory addresses are numbers and can be manipulated. For example, we can easily increment the address of the current location to obtain the address of the next location, or compare the two addresses to determine which address is smaller. As a result, it is more powerful and more flexible to access memory locations using addresses than using names.

What is a pointer? To take advantage of names (easy for humans to remember) and addresses (flexible in programming), we give a name to an address. The name of an address is a **pointer**. In other words, a pointer variable contains the address of another variable. Like any variable, a pointer variable is an l-value and the address stored in the pointer variable is an r-value.

Pointer as a data type is common in most imperative languages. The data range is the address space of the programming language. In C/C++, the data range is the same as an unsigned integer. The operations on pointers include the following:

- **Assignment operation:** An address value can be assigned to a pointer variable.
- **Integer operations:** A pointer variable can be operated like an integer variable.
- **Referencing operation:** Obtain the address of a variable `x` from the variable name: `&x`. The ampersand `&` is called the **address-of** operator that returns the address value of the variable it precedes. For example, if integer `x` is allocated at memory address = 2000, then `&x` will return 2000. Please note that `&x` returns the address value, not a pointer variable containing that address value, and thus `&x` is an r-value and can never appear on the left-hand side of an assignment statement.
- **Dereferencing operation:** To access the variable pointed to by a pointer variable `y`, we can use the **dereferencing** operator`*y`. In other words, the dereferencing operator `*` creates a new name for the variable pointed to by the pointer variable `y`. Please note that `*y` is a new name of the variable pointed to by the pointer variable that `*` proceeds. `*y` is an l-value and can appear on both sides of an assignment statement.

KH ALL RIGHTS RESERVED

Although C/C++ has a pointer type, there is no type name for pointers. A pointer is declared by the type to which it points. For example:

```
int i = 137, *j;
j = &i;
```

Variable `i` is an integer and `j` is a pointer variable pointing to the integer variable `i` or `*j`, which becomes an alias (another name) of the variable. In other words, variable `i` has two names: `i` and `*j`. Assume that the compiler has associated the variable `i` with the address 100, then the statement "`j = &i;`" will assign 100 to `j`.

A pointer variable is a variable too. We can define another pointer variable to point to a pointer variable. For example, we can extend the above example to

```
int i = 137, *j = 0, **k = 1;                                    // 1
j = &i;                                                          // 2
k= &j;                                                           // 3
*j = 0;                                                          // 4
**k = 1;                                                         // 5
```

In the example, `k` is a pointer variable pointing to the pointer variable `j`. Assume the compiler has allocated address 100 to `i`, 160 to `j`, and 120 to `k`. Initially, the three variables are independent, as shown in Figure 2.8. Please note the initializations "`*j = 0, **k = 1;`" at line 1 put the value 0 in variable `j` and put the value 1 in variable `k`. This is different from the assignment statements at lines 4 and 5, where 0 and 1 are put into the variable `*j` and `**k`, respectively! Here, you can see again that static semantics (context) and dynamic semantics are different!

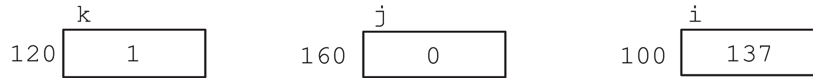


Figure 2.8. Variables `i`, `j`, and `k` are declared as independent variables at line 1 of the code.

After the execution of the statement at line 2, the address of variable `i` is put in `j`, resulting in `j` pointing to `i` (holding `i`'s address); and after the execution of the statement at line 3, the address of variable `j` is put in `k`, resulting in `k` pointing to `j` (holding `j`'s address). The new relationship between the three variables `i`, `j`, and `k` after statements at lines 2 and 3 is illustrated in Figure 2.9.

Variable `i` is initialized to value 137. Using the address-of operator, statement "`j = &i;`" puts the address of `i`, 100, into pointer variable `j`. Statement "`k = &j;`" puts the address of `j`, 160, into pointer variable `k`. On the other hand, we use the dereferencing operator to access the variable pointed to by the pointers. Since `k` holds the address of `j`, we can use `*k` to access `j`, or `*k` becomes an alias of `j`. Similarly, since `j` holds the address of `i`, we can use `*j` to access `i`, or `*j` becomes an alias of `i`. Furthermore, since `*k` is an alias of `j`, `**k` is an alias of `i` too, that is, `i` has two aliases: `*j` and `**k`. However, since `&i` is an r-value (not a variable), we cannot perform an `&&i` operation.

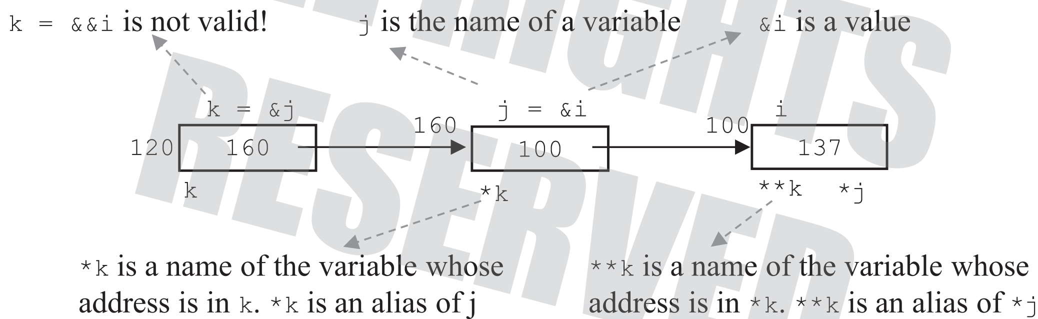


Figure 2.9. Relationship between variables and their pointers after lines 2 and 3.

At lines 4 and 5, both assignment statements modify variable `i` (`*j` and `**k` are aliases of `i`), resulting in the variable `i` being first changed from 137 to 0, and then changed from 0 to 1. If we compare the effect of these two statements with the initialization at line 1, we can see that similar assignment operations in the initialization part (contextual structure) and in the execution part (semantic structure) have different effects.

In this section, we discussed only the concept, the declaration, and the assignment of pointer variables. We will discuss more applications of pointers in the following sections. It will make a lot more sense when we combine pointer type with other complex data types.

2.4.3 String

There is no specific string type in C. Any array of characters can be considered a string, and thus a string variable can be declared as an array of characters, for example:

KH ALL RIGHTS RESERVED

```
char str1[] = {'a', 'l', 'p', 'h', 'a'}; // initialized as an array
char str2[] = "alpha";              // initialized as a string
char str3[5];                       // without initialization
```

As can be seen from the example, there are two different ways to **initialize** an array of characters in the declaration. The effect of these two initializations is slightly different.

The first declaration and initialization uses exactly the same method that declares and initializes any array, and thus `str1` is 100% an array of characters. On the other hand, we can also consider and use `str1` as a string. It has all the features that an array of characters should have. For example, we can modify the string in the following code and print the modified string

```
char str1[] = {'a', 'l', 'p', 'h', 'a'};
for (i = 0; i < sizeof(str1); i++) {
   str1[i] += 1; // same as str1[i] = str1[i]+1;
   printf("%c", str1[i]);
}
printf("\t sizeof(str1) = %d\n", sizeof(str1));
```

As expected, the output of the code is

```
bmqib  sizeof(str1) = 5
```

The second initialization indicates to the compiler that the array of characters is considered a string. In this case, the compiler will append a null terminator (null character) '`\0`' to the end of the string. In the ASCII table, the code for the null character is `0` (seven binary zeros). Please notice that the code for the digit '`0`' is 48. Appending the null character to the end of a string increases the size of the string by one, as shown in the following code.

```
char str2[] = "alpha";
for (i = 0; i < sizeof(str2); i++) {
   str2[i] += 1;
   printf("%c", str2[i]);
}
printf("\t sizeof(str2) = %d\n", sizeof(str2));
```

The output of the code is:

```
bmqib  sizeof(str2) = 6  // '\0'is not a printable character
```

To have the same effect, one can use the following initialization to append the '`\0`' to the end of the string:

```
char str4[] = {'a', 'l', 'p', 'h', 'a', '\0'};
```

As we discussed in the array section, we can specify the size in the declaration. If the size of the array is specified and the size is smaller than "string_length+1," the '`\0`' character and possibly some characters of the string cannot be stored in the variable. This is called **truncated initialization**. For example:

```
char str5[5] = "alpha";
char str6[4] = "alpha";
```

In this example, the null terminator '`\0`' will not be stored in `str5`; furthermore, the last "`a`" and the null terminator are not stored in `str6`. Figure 2.10 shows the memory map and initialization of `str1` through `str6`.

KH ALL RIGHTS RESERVED

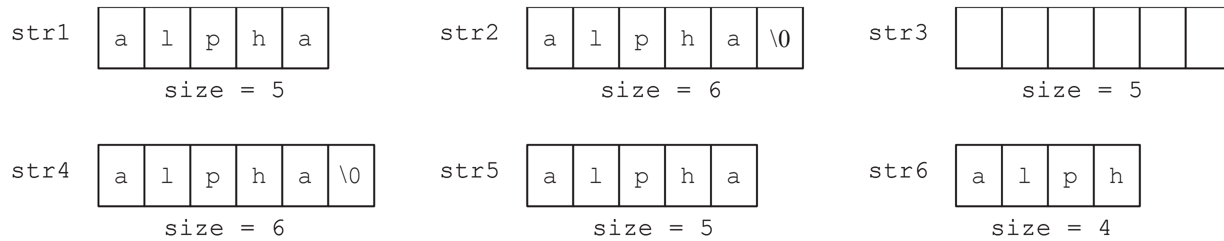


Figure 2.10. Memory allocation and initialization of six strings.

A number of string functions have been defined in the string package <string.h>and related library packages. A list of useful string and character manipulation functions is given in Table 2.3.

Having introduced the string functions, we can use the strlen(str) to replace the sizeof(str) in the previous example. In fact, sizeof(str) worked in that example because each character takes exactly one byte and sizeof(str) returns the number of bytes. The program would not work if we had used the wchar_t (two bytes per character) type instead. However, strlen(str) will work in both cases. Another difference, sizeof will include the byte used to store the null terminator '\0', while strlen does not.

Notice that some environment, such as Visual Studio, implemented a new set of string library functions with an extended name, such as strcpy_s(str1, str2) and strcmp_s(str1, str2). These functions added security features to prevent code attacks at the instruction level, such as the return-oriented programming (ROP) attacks. An ROP attack combines code sequences in library functions to create malicious functions by changing the return address of function calls. This kind of attack does not need to inject malicious code into a system; instead, it modifies the return addresses on system stack.

So far, we have discussed the string as an array of characters. A string can also be defined by a pointer to a character or, more accurately, a pointer to the first character of a string. For example, the declaration

```
char *p = "hello, ", *q = "world", *s;
```

declares three pointer variables p, q, and s, each pointing to a character type variable. Pointer variables p and q are initialized to values that point to a string, while s is not initialized. Now the question is, what is the difference between the array-based strings and the pointer-based strings?

KH ALL RIGHTS RESERVED

Library	Function	Description	Example
stdlib.h	atoi(str)	Convert a numeric string into an integer	`atoi("356")` returns 356 as an integer
	itoa(i, str, base)	Convert an integer `i` to a string using the specified `base` and link the result to pointer *str*	`itoa(356, str, 10)` results in pointer `str` pointing to string "`356`".
stdio.h	getc(stdin)	Read a character from keyboard	ch = getc(stdin);
	gets(str)	Read a string from keyboard	gets(s); strcpy(str, s);
	putc(ch, stdout)	Print a character onto screen	ch = 'a'; putc(ch, stdout);
string.h	strcat(str1, str2)	Concatenate str2 to the end of str1	strcat (str, "hello world");
	strncat(str1, str2, n)	Concatenate the first n characters (substring) of str2 to the end of str1	strcat (str, "hello world", 3);
	strcmp(str1, str2)	Return 0 if str1 == str2, Return <0 if str1 < str2, Return >0 if str1 > str2,	if (strcmp(s1, s2)) y = x+1;
	strncmp(str1, str2, n)	Same as strcmp, except only compare the first n characters	if (strncmp(s1, s2, 3)) y = x+1;
	stricmp(str1, str2)	Same as strcmp, except letter comparisons are case insensitive	if (stricmp(s1, s2)) y = x+1;
	strcpy(str1, str2)	Copy str2 into str1	strcpy(str, "hello world");
	strlen(str)	Return the length of str	L = strlen(str);
ctype.h	tolower(ch)	Return the lowercase equivalent	tolower('D') returns 'd'
	toupper(ch)	Return the uppercase equivalent	toupper('b') returns 'B'

Table 2.3. Useful library functions for string and character manipulation.

KH ALL RIGHTS RESERVED

We can examine the following example to see in detail the differences and similarities between array-based strings and pointer-based strings.

```
#include <stdio.h>
#include <string.h>
void main (void) {
  char p1[] = "hello", q1[] = "this is an array-string", s1[6];          //1
  char *p2 = "Hi", *q2 = "this is a pointer-string", *s2=0;             //2
  char *temp;                                                           //3
// s1 = p1;          // Array name cannot be a L-value                  //4
// s1 = "hi";        // Array name cannot be a L-value                  //5
  strcpy(s1, p1);    // We must use string-copy function                //6
  printf("s1 = %s\t len-s1 = %d\n", s1, strlen(s1));                    //7
  strcpy(s1, q1);                                                       //8
  printf("s1 = %s\t len-s1 = %d\n", s1, strlen(s1));                    //9
```

```
  printf("s1 = %s\t size-s1 = %d\n", s1, sizeof(s1));                  //10
  for (temp = s1; temp < s1+strlen(s1); temp++)                         //11
      *temp += 1;                                                       //12
  printf("s1 = %s\n", s1);                                              //13
  for (temp = &s1[0]; temp <&s1[0] + strlen(s1); temp++)                //14
      *temp -= 1;                                                       //15
  printf("s1 = %s\n", s1);                                              //16
//strcpy(s2, p2);                                                       //17
  s2 = q2;                                                              //18
  printf("s2 = %s\t len-s2 = %d\n", s2, strlen(s2));                    //19
  printf("s2 = %s\t size-s2 = %d\n", s2, sizeof(s2));                  //20
  for (temp = s2; temp < s2+strlen(s2); temp++){                        //21
//    *temp += 1;                                                       //22
  }
  strcpy(s1, q2);                                                       //23
  for (temp = s1; temp < s1+strlen(s1); temp++)                         //24
      *temp += 1;                                                       //25
  printf("s1 = %s\t len-s1 = %d\n", s1, strlen(s1));                    //26
}
```

All incorrect statements are commented out so that the program can be compiled and executed. The output of the program is

```
s1 = hello                        len-s1 = 5
s1 = this is an array-string      len-s1 = 23
s1 = this is an array-string      size-s1 = 6
s1 = uijt!jt!bo!bssbz.tusjoh
s1 = this is an array-string
s2 = this is a pointer-string     len-s2 = 24
s2 = this is a pointer-string     size-s2 = 4
s1 = uijt!jt!b!qpjoufs.tusjohlen-s1 = 24
```

KH ALL RIGHTS RESERVED

Now we explain each statement in the program.

Statement 1 declares three array-based strings `p1`, `q1`, and `s1`. Variables `p1` and `q1` are initialized to a string while `s1` is not initialized.

Statement 2 declares three pointer variables `p2`, `q2`, and `s2`, each pointing to a character type variable. Variables `p2` and `q2` are initialized to values pointing to a string while `s2` is not initialized.

Statement 3 declares a pointer variable "`temp,`" to be used as a temporary pointer variable.

Statement 4 tries to assign the string variable `p1` to string variable `s1`. A compilation error occurs, because `s1` is in fact an array name. We cannot assign anything to an array name. We can assign a value only to an element of an array (e.g., "`s1[0] = 'a';`" is a valid assignment).

Statement 5 tries to assign a string literal (value) to `s1`. For the same reason stated above, the statement causes a compilation error.

The correct way to assign a string variable or a string literal to an array-based string is to use the library function strcpy(s1, p1). In statement 6, string p1 is copied into string s1 correctly and printed correctly in statement 7.

Statement 8 copies q1 into s1. String s1 and its length strlen(s1) are correctly printed in statement 9 (see output of the program). Please note that the length of s1 is declared to be 6. How can the program put 24 characters in 6 places? This is in fact a semantic error that the compiler does not check. The runtime system could handle the error by checking the sizes and lengths of the two arrays in strcpy and prevent a longer array to be string-copied into a shorter array. However, these kinds of checks will slow down the execution of the program, and the designers of C decided to leave the responsibility to the programmers! As a programmer, you should know the lengths of the two arrays. If you really do not, you can always use the strlen function to find the lengths; for example, you can use the following statement to replace statement 6:

```
if (sizeof(s1) >= strlen(p1)) strcpy(s1, p1); else printf("error\n");
```

We still have not answered the question of how to put 24 characters in 6 places. What has happened is that the 18 extra characters are appended to the 6 declared memory locations, as shown in Figure 2.11.

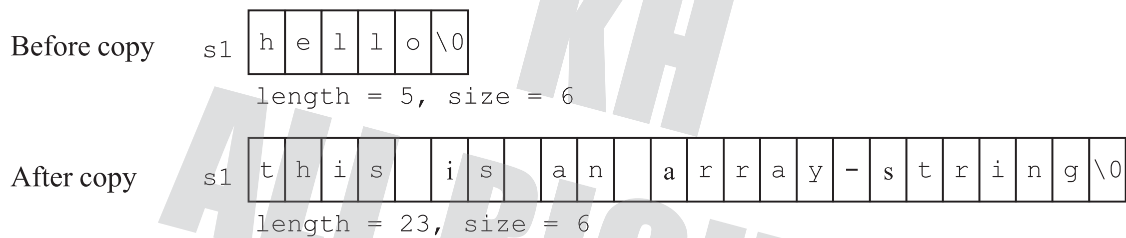


Figure 2.11. A strcpy operation may illegally use more space than what is declared.

Before we perform the strcpy, s1 contains 5 characters and the size of s1 is 6, as specified in the declaration. After we have performed the strcpy, a 23-character string is copied to the memory location starting from address s1. Obviously, the string goes beyond the limit of the size of s1. That is why we can still print the string s1 with all characters, because the printf function starts from s1 and stops when the character '\0' is detected.

The problem is that the use of memory beyond the declared boundary is unknown to the compiler and the runtime system. Please note that in the output of print-statement 10, the size of s1 is still 6, even if a 23-character long string has been copied into s1. There are three possibilities:

(1) The locations are not allocated to any variable and you are lucky;

(2) The locations are allocated to other variables and you have overwritten the values of those variables;

(3) The locations are allocated to other variables and your values will be overwritten later.

In cases (2) and (3), your program may crash or, even worse, still behave normally but produce incorrect results that go undetected and cause much more damaging consequences. We explained in Chapter 1 that C/C++ use weak type checking. As you can see here, C/C++ are also weak in runtime checking, which leaves a huge responsibility entrusted to the programmers.

Now we continue to discuss the example. In statement 11, "temp = s1;" means to assign the address of s1 (or the address of the first element s1[0]) to a pointer variable temp. Note that we use &x to obtain the address of a simple variable x, and we simply use the array name s1 to obtain the address of an array s1

KH ALL RIGHTS RESERVED

(or the address of the first element of the array). In some C compilers, you can use either &s1 or s1 to obtain the address of array s1. However, in C++, you can use the array name only to obtain the address of the array: Please also note that "temp = s1;" is same as "temp = &s1[0];" because s1[0] is a simple variable of character type, not an array (see the use in statement 14). In the next part of the for-loop, we use "temp < s1+strlen(s1);" to test if the value (address) of temp is less than the initial address of s1 plus the length of s1. And then we increment temp in the next part.

In statement 12, we do "*temp += 1;", which means we increment the value pointed to by the pointer variable temp. The statement is the same as "(*temp)++;", but not the same as "*(temp++);", which increments the pointer value, instead of the pointed value. Please note that "*temp++;" is the same as "*(temp++);", because the unary operators * and ++ operate at the same precedence level. However, they associate from RIGHT to LEFT! Therefore, in "*temp++;", temp associates with ++ before *, and hence "*(temp++);" gets evaluated as "*temp++;".

Statement 13 will print the string in which every character is changed to the next character in the ASCII code. For example, s is changed to t, h is changed to i, i is changed to j, etc.

Statement 14 is equivalent to statement 11, and statement 12 is equivalent to statement 13 in structure, and it reverses (decrypts) the encryption in statement 12. Statement 16 prints the decrypted string that is same as the string before encryption.

We have discussed array-based strings so far, and now we turn to discuss pointer-based strings.

In statement 17, we try to do what we did in statement 6: string-copy p2 to s2. However, the attempt will cause a compilation error. Thus, we commented the statement out so that we can continue with the other statements. The reason for this compilation error is that s2 is a pointer variable and there is no memory allocated for a string. In the declaration in line 2, "char ... *s2=0;" means that s2 is declared as a pointer variable to char and the pointer is initialized to 0. It does not mean that the pointer is initialized to the address of the string "0." However, should we use "char ... *s2="0";", it does mean that the pointer is initialized to the address of string "0."

In statement 18, we assign q2 to s2. We assign the value of q2 (a pointer value) to pointer s2. Both pointers point to the same string. Here only pointer manipulation is involved. No string duplication is performed, as shown in Figure 2.12.

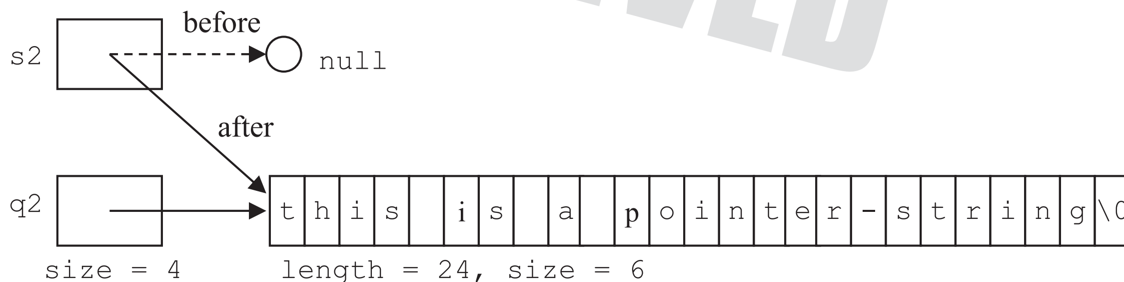


Figure 2.12. Both pointers q2 and s2 point to the same string.

Statement 19 prints the string pointed to by s2 and its length. Statement 20 does not print the size of the string; instead, it prints the size of the pointer variable, which is 4 bytes (same as the size of an integer).

Statement 21 is similar to statements 11 and 14. Statement 22 tries to modify the character pointed to by temp, as we did in statements 12 and 15. However, we will have a runtime error. In C/C++, if a string is

KH ALL RIGHTS RESERVED

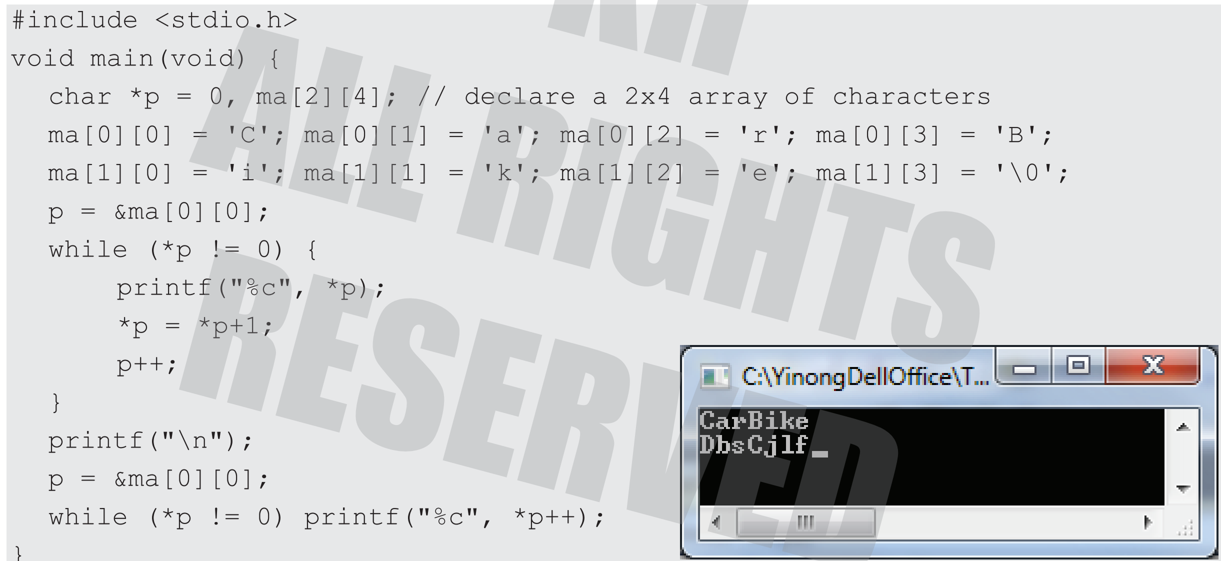
assigned to a pointer-based string variable, the string is a string literal and cannot be modified. If we try to modify it, we will encounter a runtime error. Thus, we commented out this statement so that we can continue to compile other statements.

Although we cannot modify a string initialized as a pointer-based string, we can modify the string if it is copied into an array. Statement 23 copies pointer-based string q2 into s1 and then we can modify the string in s1 in statements 24 and 25. Statement 26 prints the modified string.

Through this example, we explained the following aspects of a string in C/C++.

- We can use the array of characters to declare a string variable and initialize the string variable to a string literal. We can access (read and write) the characters in the string as array elements. We can assign the initial address of the array (string) to a pointer variable and use this pointer to access (read and write) the characters in the string.
- We can declare a string variable using a pointer to character type and initialize the string to a string literal. We can read the characters in the string, but we cannot modify the characters.
- We can copy a pointer-based string into an array and we can modify the characters in the array.

A multidimensional array is stored in memory as a sequence of its elements or a one-dimensional array and can be processed easily using a pointer. We start with an example of 2-D array of characters.




```
#include <stdio.h>
void main(void) {
  char *p = 0, ma[2][4]; // declare a 2x4 array of characters
  ma[0][0] = 'C'; ma[0][1] = 'a'; ma[0][2] = 'r'; ma[0][3] = 'B';
  ma[1][0] = 'i'; ma[1][1] = 'k'; ma[1][2] = 'e'; ma[1][3] = '\0';
  p = &ma[0][0];
  while (*p != 0) {
       printf("%c", *p);
       *p = *p+1;
       p++;
  }
  printf("\n");
  p = &ma[0][0];
  while (*p != 0) printf("%c", *p++);
}
```

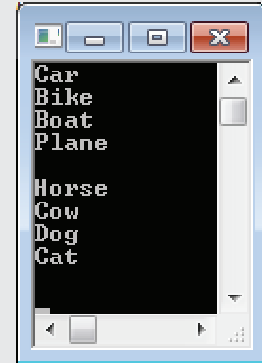
KH ALL RIGHTS RESERVED

In the example above, a 2-D array is declared and initialized. We use pointer variable `p` to parse through each element, adding 1 to each element. The characters before the addition and after the addition are printed and the output of the program is shown in the screenshot.

We can further define 3-D arrays. The code below defines a 2-D array of strings. As a string is an array of characters, the array is in fact a 3-D array of characters.

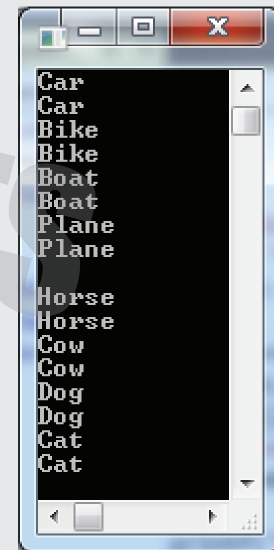
As we initialize the 2-D array using string values, the null terminator is appended to the end of each string. To print these strings, we use character print, and thus use the 3-D array element `ma[i][j][k]` to print each character. We use `ma[i][j][k]!='\0'` as the termination condition of the inner-most for-loop. Another option is to use the string length operation `strlen()` in the condition for the inner-most for-loop: `for (k=0; k < strlen(ma[i][j]); k++)`.

```
#include <stdio.h>
void main() {
  char *ma[2][4] = {{"Car", "Bike", "Boat", "Plane"},
                    {"Horse", "Cow", "Dog", "Cat"}};
  int i=0, j=0, k=0;
  for (i=0; i<2; i++) {
      for (j=0; j<4; j++) {
           for (k=0; ma[i][j][k]!='\0'; k++)
               printf("%c", ma[i][j][k]);
           printf("\n");
      }
      printf("\n");
  }
}
```



We can also use pointer operations to access the 3-D array, in which we obtain the address of each string through the operation char `*p = ma[i][j]`, and then print each string, as shown in the following code.

```
#include <stdio.h>
void main() {
  char *ma[2][4] = {{"Car", "Bike", "Boat", "Plane"},
                    {"Horse", "Cow", "Dog", "Cat"}};
  int i = 0, j = 0, k = 0;
  char *p = 0;
  for (i = 0; i<2; i++) {
      for (j = 0; j<4; j++) {
           p = ma[i][j];        // Do not use &ma[i][j]
           printf("%s\n", p);   // print string
           while (*p!=0)
               printf("%c", *p++); // print char
           printf("\n");
      }
      printf("\n");
  }
}
```



KH ALL RIGHTS RESERVED

Each word is printed twice, because we used two different ways to print the words. First, we printed the entire word as a string, and then we printed each character of a string in a while loop. Notice that we use `p = ma[i][j];` instead of `p = &ma[i][j];` because the element of the 2-D array is an array of character, and the array name is the initial address of the array.

In C++, you can use all the C functions for string processing. However, a C++ library <string> is added to allow string declaration and processing without explicitly declaring an array of characters. The following code shows an example of using the string library.

```
#include <iostream>
#include <string>  // This is the library for C++ string operations
using namespace std;
```

```
void main() {
  string cat1 = "Max", cat2, temp; int  length;
  cout << "please enter a name for a cat" << endl;
  cin >> cat2;
  temp = cat1; cat1 = cat2; cat2 = temp; // swap the names of cat1 & cat2
  // One can also treat the string as an array of characters
  length = cat1.size();
  cout << "The length of cat1 is: " << length << endl;
  for (int i = 0; i < length; i++)
       cout << cat1[i];
  cout << endl;
}
```

Notice that we do not need to use strcpy; instead, we can simply use assignment cat1 = cat2 to copy the name in cat2 into cat1. Instead of using strlen(cat1) or sizeof cat1, we use cat1.size(), where cat1 is an object, and .size() is a member function of the object. We will discuss more details about class and object in the following chapter. The console output of the program is shown as follows:

```
please enter a name for a cat
Molly
The length of cat1 is: 5
Molly
```

KH ALL RIGHTS RESERVED

2.4.4 Constants

Most programming languages allow constants to be declared. However, their implementations depend on the language definition and the compiler technologies.

C/C++ provides three different ways to introduce constants:

- **Macro**: As discussed in Section 1.4.2, we can use a macro definition to introduce a constant. The constant will substitute for the name at the preprocessing time. The advantage of a macro constant is its efficiency. A small constant may fit in an immediate-type of a machine instruction and thus save a memory access. The disadvantage is that the way a macro is defined is different from the way a variable is declared and initialized (nonorthogonal).
- **const qualifier**: A constant is a "variable" that a program cannot modify. The advantage is that the constant is declared and initialized in the same way as a variable is declared and initialized (orthogonal). However, a memory access is needed in order to access the constant (slower). We will discuss this kind of constant in this section.
- **Enumeration constant**: We can introduce constants by defining an enumeration type variable. This topic will be discussed in the following section.

The simplest way to introduce constants in C/C++ is to use the qualifier `const` before the variable declaration. For example:

```
const int min = 5, max = 100, pi = 3.14159265358979;
const char x = 'a', y = 's';
```

Constants increase the readability of programs. For example, the statement

```
if (x >= min and x <= max) x = x*x*pi;
```

is easier to understand than the statement

```
if (x >= 5 and x <= 100) x = x*x*3.14159265358979;
```

Constants also prevent us from making semantic errors. For example, if we try to modify a constant in a statement like

```
max = max + 10;
```

the compiler will raise a compilation error because `max` is a constant.

A constant defined by qualifier `const` is actually a "constant variable" and thus it has a memory address. We can apply the dereferencing operator on the constant. For example, the statement

```
temp = &max;
```

will put the memory address of constant `max` into the pointer variable `temp`.

In the following example, we demonstrate that we can even modify a constant variable if we can get around the compiler's check.

```
void main() {
   const int max = 100;
   int *temp;                  // temp is a pointer to an integer
// max = max + 10;             // Compilation error would occur
   temp = &max;                // assign the address of max to temp
   *temp = *temp + 10;         // max is modified through pointer temp
   printf("max = %d\n", max); // The output is: max = 110
}
```

KH ALL RIGHTS RESERVED

Self-checking question: What would happen if we used "`#define max 100`" to define the constant? Will the statement "`temp = &max;`" work?

Through this example, we can see that a constant defined by `const` qualifier is in fact a variable.

It has a memory location and memory address and we can use the `&` operator to obtain its address.

Compiler protection is used. A compilation error will occur if you try to modify a `const` variable. In some versions of the compiler, a warning, instead of an error, will be given.

It can be modified if you can get around the compiler; for example, using an alias, you can modify a `const` variable.

2.4.5 Enumeration type

We have discussed data types predefined in C/C++. Most modern programming languages also provide mechanisms (type constructors) to allow programmers to define more complex data types. We will discuss the enumeration type in this section and other complex data types in the following sections.

Enumeration type is usually used for variables that can take an enumerable ordered set of values. Each of these values is given a name and we use the name to access the corresponding value. These names are associated with integer values starting from 0. Each enumeration type is a distinct data type.

Enumeration types in C/C++ are defined using the keyword **enum**. For example:

```
#include <stdio.h>
typedef enum {
      Sun, Mon, Tue, Wed, Thu, Fri, Sat
```

```
   } Days;
Days x = Sun, y = Sat;
void main (void) {
   while (x <= y) {
        printf("x = %d\t", x);
        x++;
   }
   printf("\n");
}
```

The names (constants) in the `Days` are not initialized and integers starting from 0 will be associated with each name in the given order. Thus, the type definition above defines seven constants equivalent to:

```
const int Sun = 0;
const int Mon = 1;
const int Tue = 2;
const int Wed = 3;
const int Thu = 4;
const int Fri = 5;
const int Sat = 6;
```

The output of the program is

```
x = 0  x = 1  x = 2  x = 3   x = 4  x = 5  x = 6
```

We can also initialize the names in the definition. For example, if we define `Days` as follows

```
typedef enum {
        Sun = 1, Mon = 2, Tue = 3, Wed = 4, Thu = 5, Fri = 6, Sat = 7
   } Days;
```

then the output of the program will be

```
x = 1  x = 2  x = 3  x = 4   x = 5  x = 6  x = 7
```

We now show a longer example demonstrating the use of enumeration types.

```
#include <stdio.h>
#include <time.h>
typedef enum {
        red, amber, green
   } traffic_light;
void sleep(int wait);    // forward declaration
main() {
traffic_light x = red;
printf("Red:\tStop!\n");
while (1)
   switch (x) {
        case amber:
             sleep(1);         //sleep 1 second
             x = red;
```

KH ALL RIGHTS RESERVED

```
            printf("Red:\tStop!\n"); break;
       case red:
            sleep(6);        //sleep 6 second
            x = green;
            printf("Green:\tGo>>>\n"); break;
       case green:
            sleep(12);       //sleep 12 second
            x = amber; printf("Amber:\tBrake...\n");
   }
}
void sleep(int wait) {   // Sleep for a specified number of seconds.
   clock_t goal;          // clock_t defined in <time.h>
   goal = wait * CLOCKS_PER_SEC + clock();
   while( goal > clock() )
       ;
}
```

In this program, we defined an enumeration type called traffic_light with three possible values: red, amber, and green. We could use an int type instead. However, the program would be less readable and prone to error. A snapshot of the output is given as follows.

```
Red:    Stop!
Green:  Go>>>
Amber:  Brake...
Red:    Stop!
Green:  Go>>>
Amber:  Brake...
...
```

In this example, the time function clock() in <time.h> is used to obtain the number of clock cycles from a given point. This function can be used to measure the time between any two points. For example, the following piece of code can measure the time used by a function foo().

```
c1 = clock();                        // time stamp 1
foo();
c2 = clock();                        // time stamp 2
interval = (double) (c2 – c1) / CLOCKS_PER_SEC; // time difference
```

The time difference computed in this example is in seconds. The precision of this method is 0.001 second.

There is another time function time() that can be used to measure the time in seconds. The following code shows the use of the time function and other related functions:

```
#include <stdio.h>
#include <time.h>
main() {
   int n; time_t start, finish;  double result, duration;
   time( &start );    // get the initial time
   for( n = 0; n < 900000000; n++ )
```

KH ALL RIGHTS RESERVED

```
        result = 3.1415 * 2.23;
   time( &finish );   // get the end time
   duration = difftime( finish, start);  // compute difference
   printf( "\nThe program takes %2.4f seconds\n", duration );
}
```

2.5 Compound data types

In this section, we discuss compound data types that are composed of several data types, including structure, union, array of structures, linked list of structures connected by pointers, and file types.

2.5.1 Structure types and paddings

A structure is created using the keyword `struct`. The general way to define a structure type is

```
struct type_name {
       type  field1;
       type  field2;
       . . .
       type  fieldn;
   } struct_variable_name;
```

For example:

```
struct stype {
   char ch;
   int  x;
} u, v;         // We can declare variables of the type here.
void main() {
   struct stype s, t; // We can use the type to declare variables here too.
   ...          // The keyword struct must be used before type name.
}
```

Now we will study an example with a structure type.

```
struct Contact { // define a type that can hold a person's detail
   char  name[30];
   long  phone;
   char  email[30];
};
void main() {
   struct Contact x, y, z;
   strcpy(x.name, "Mike Smith");
   x.phone = 9650022;
   strcpy(x.email, "mike.smith@asu.edu");
   strcpy(y.name, "Jane Miller");
   y.phone = 9650055;
   strcpy(y.email, "jane.miller@asu.edu");
}
```

KH ALL RIGHTS RESERVED

As you can see from the example, we use `x.name` notation to access the `name` field of the variable `x`. We will see more examples of structures in the following sections, where we will combine the structure types with the array and pointer types.

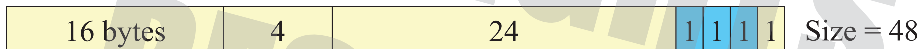
The processor reads integers and floats in words. If a structure contains an integer, a pointer, or a float, the structure size must be aligned into multiples of 4 in a 32-bit computer and multiples of 8 in a 64-bit computer. Consider a 32-bit computer. If a structure has a member of integer, pointer, or float, and a member whole size is not a multiple of 4, padding is needed to align the data in memory. It may need padding of one, two, or three bytes to pad one part of a structure. Consider the following snippet of code:

```
struct personnel {
  char name[16];
  int phone;
  char address[24];
  char gender;       // F or M
  // char CSmajor;   // Y or N
} person;
printf("struct size = %d", sizeof person);
```

If you count the bytes, the `struct` variable person will need 45 bytes. However, the `printf` will print the size of person = 48.

KH ALL RIGHTS RESERVED

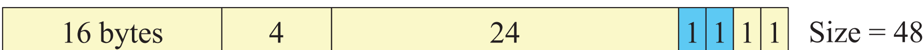
The reason is as follows. The structure contains an integer variable phone. An integer will be read at machine language level by a “Load Word” instruction, and thus the entire structure will be read using the Load Word instructions. As shown in the following memory map, the compiler will add three bytes before the character type variable to make the byte into a four-byte word. As the result, the total size of the person variable is 48.



In the code, uncommenting the two lines of code adds one more character variable into the structure, as shown below:

```
struct personnel {
  char name[16];
  int phone;
  char address[24];
  char gender;    // F or M
  char CSmajor;  // Y or N
} person;
printf("struct size = %d", sizeof person);
```

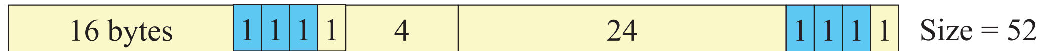
The total size of the structure will remain unchanged as 48. In this case, the three bytes of padding will reduce to two bytes only, as shown in the following map:



The order of the variables in a structure will be preserved by the compiler when it allocates the memory. If the character types are separated by a Word-type variable, multiple paddings are required. Consider the following snippet of code, where the variable gender is moved before the variable phone:

```
struct personnel {
  char name[16];
  char gender;    // F or M
  int phone;
  char address[24];
  char CSmajor;  // Y or N
} person;
printf("struct size = %d", sizeof person);
```

The memory map is given as follows:



Two paddings of three bytes each are added by the compiler, resulting in a structure of 52 bytes. These examples show that it will result in a more efficient code if the character type variables are kept together in the structure definition.

Padding is required only for word type of variables, such as int, float, and pointer. If a structure contains character type of variables only, no padding will be added, irrespective of whether the total size is a multiple of four or not. The compiler will use "Load Byte" instructions to read the structures that contain character (byte) type only. Consider the following code, where the int variable phone is removed:

```
struct personnel {
  char name[16];
  char gender;    // F or M
  char address[24];
  char CSmajor;  // Y or N
  char KnowJava;   // Y or N
} person;
printf("struct size = %d", sizeof person);
```

KH
ALL RIGHTS
RESERVED

The program will print the size of person = 43, which is not a multiple of 4.

Consider the following snippet of code in a 32-bit computer:

```
struct contact {
    char    name[30];
    int     phone;
    char    email[30];
} x;
```

What is the size of variable `x` in bytes? It is incorrect to simply add 30+4+30. Because there is an integer type involved, two padding bytes must be added to name and email arrays, and thus, the total number of bytes for variable `x` is 68, instead of 64. However, if we keep the name and email members together in the order, the size will become 64.

2.5.2 Union

A **`union`** type variable is a region of shared memory that, over time, can contain different types of values. At any given moment, a union can contain only one value. Programmers must make sure the proper type is used at the proper time. The general way to define a union type is

```
union union_name {
```

```
    type field1;
    type field2;
    . . .
    type fieldn;
} union_variable_name;
```

For example:

```
union utype {
    char ch;
    int  x;
} v;
void main(){
    union utype s, t;  //We can use the type to declare variables here too.
    ...                // The keyword union must be used before type name.
}
```

In this example, we define a union type called `utype` and declare a variable `v` of `utype`. Similar to a structure type, we can have multiple data fields in the type definition. In this example, there are two data fields. The field variable `x` belongs to the `int` type and takes 32 bits or 4 bytes (in a 32-bit machine) and `ch` takes 8 bits or 1 byte. If the union type is defined as a structure type, the variable `v` will have 4+1 bytes of memory allocated. However, in the union type, all data fields share the same memory. If these data fields require different sizes of memory, the largest size among the data fields will be allocated and the smaller sized fields will occupy a part of the memory. In this example, 4 bytes of memory will be allocated and the smaller field `ch` will share the first byte of `x`.

The way we access a union type variable is similar to that of the structure type variable. For example,

```
v.x = 124000;    // put an integer value into the data field x
v.ch = 'C'; // put a character value into the data field ch.
```

Since the two data fields share a part of the memory, the second assignment will overwrite the first byte of `v.x`, destroying the integer value in `v.x`. Obviously, if we do not use the data fields carefully, we can easily make mistakes in programming.

The question is why do we need such an unsafe data structure? The reason is that it could be useful in certain situations. The following example depicts such a situation where union type variables make the program more elegant.

Assume we want to define a data type to store personnel information for both faculty members and students in a university. The faculty and students have ID numbers with different lengths. A person in a university has either a faculty ID or a student ID. If we use two separate data fields for faculty ID and student ID, we will use only one of the two data fields for every record. If we use only one data field and leave the extra bytes free when an ID number does not have enough characters to fill all bytes, we could lose our view of whether we are dealing with a student record or a faculty record. A union type would solve the problem, as shown in the following program:

```
#include <stdio.h>
#include <string.h>
struct Personnel {   // Define a structure type called Personnel
    char  name[30];
    long  phone;
```

KH ALL RIGHTS RESERVED

```
  union identity {   // Define a union type inside the structure type
      char facultyid[8]; // Two alternative data fields are defined here
      char studentid[12];
  } id;     // We declare a variable of the union type here.
};

main(){
  struct Personnel x, *p;  // Declare a Personnel type variable and a
pointer
  strcpy(x.name, "Mike Lee");     // Copy a name into the name field
  x.phone = 21400000;             // Assign a number to phone field
  strcpy(x.id.studentid, "1999eas1234"); // Copy student ID
  printf("x.id.studentid = %s\n", x.id.studentid);
  strcpy(x.name, "Jane Smid");    // Use the same x for a faculty record
  x.phone = 9659876;
  strcpy(x.id.facultyid, "cse1234");
  printf("x.id.facultyid = %s\n", x.id.facultyid);
  p = &x;
  printf("p->id.studentid = %s\n", p->id.studentid);
  printf("p->id.facultyid = %s\n", p->id.facultyid);
}
```

In this example, the same variable is used for a student record and a faculty record. The different ID field names allow us to differentiate which record we are handling. In the next chapter, we will discuss the generic class in C++, which is a more general way of associating different classes with a class reference.

2.5.3 Array of structures using static memory allocation

Structure types will make more sense if we combine them with array and pointer types to form collections of structures. In the following example, we define an array of structures to form a database.

In the following program, `Contact` is a structure type with three data fields. The declaration

```
struct Contact ContactBook[max];
```

declares an array of structures with 100 entries. Then we use the `tail` variable as the index to access the next unused element of the array: `ContactBook[tail]`. Figure 2.13 shows the structure of the array.

Since the element of the array is of the `structcontact` type with three data fields, we use the **dot-notation** to access the data fields of the ith element:

```
ContactBook[i].name
ContactBook[i].phone
ContactBook[i].email
```

KH ALL RIGHTS RESERVED

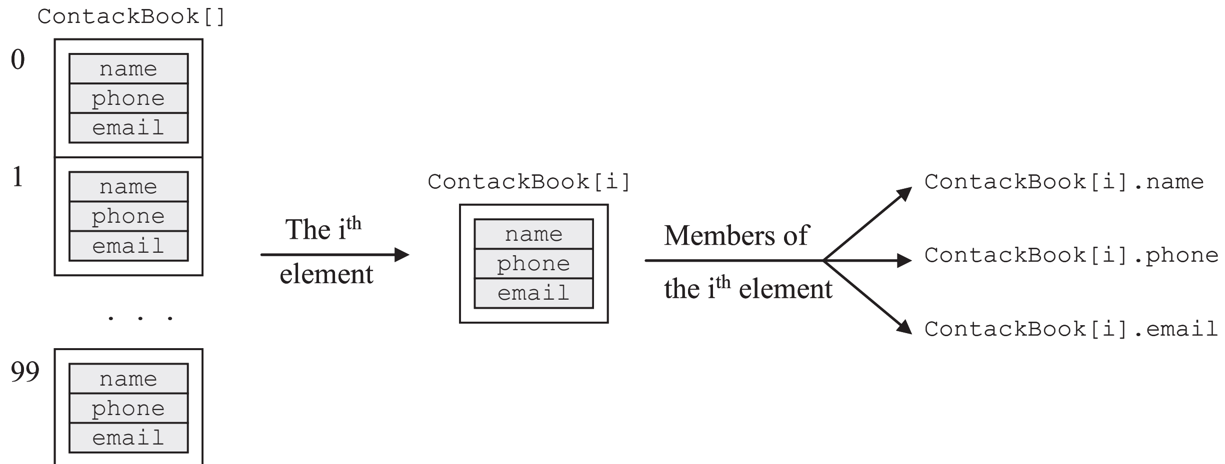


Figure 2.13. Array of structures, its element, and members of the element.

The array variable `ContactBook[max]` is a **global variable**, that is, a variable that is outside all functions. The memory locations for global variables are statically allocated by the compiler during compilation time. We call this kind of memory allocation **static memory allocation**. In this example, the compiler will allocate an array of 100 (`max`) elements before the program starts. Assume a `long` integer takes 4 bytes, and the name and email take 30 bytes each. The total number of bytes needed for one array element is then 64 bytes. The array of 100 elements will take 6400 bytes.

```
/* This program demonstrates how to define an array of structures.
   It statically allocates memory for the variables of structure type */
#include <stdio.h>
#include <string.h>
#define max 100
struct Contact { // define a node that can hold a person's detail
   char name[30];
   long phone;
   char email[30];
};
struct Contact ContactBook[max];  // an array of structures, 100 entries
int tail = 0;             // tail is defined here as a global variable
void branching(char c);  // forward declaration of a function
int insertion();          // forward declaration of a function
int search();             // forward declaration of a function
// void deletion();       // not implemented in this example
// void printall();       // not implemented in this example
void main() {             // main() first prints a menu for selection
   char ch = 'a';
   while (ch != 'q') {
       printf("enter your selection\n");
       printf("      i: insert a new entry\n");
       printf("      s: search an entry\n");
       printf("      d: delete an entry\n");    // not implemented
```

KH ALL RIGHTS RESERVED

```
        printf("       p: print all entries\n");  // not implemented
        printf("       q: quit\n");
        fflush(stdin);                // flush input buffer to make
        ch = getc(stdin);             // sure getc reads correctly
        branching(ch);
    }
}
void branching(char c) { // branch to different tasks
    switch(c) {
        case 'i': insertion(); break;
        case 's': search(); break;
        case 'q': printf("You exit the program\n"); break;
        default:  printf("Invalid input\n");
    }
}
int insertion() {    // insert a new entry
    if (tail == max) {
        printf("There is no more place to insert\n");
        return -1;
    }
    else {
        printf("Enter name, phone, email\n");
        scanf("%s", ContactBook[tail].name);
        scanf("%d", &ContactBook[tail].phone);
        scanf("%s", ContactBook[tail].email);
        tail++;
        printf("The number of entries = %d\n", tail);
        return 0;
    }
}
int search() {   // search and print phone and email via name
    char sname[30];
    int  i;
    printf("please enter the name to be searched\n");
    scanf("%s", sname);
    for (i=0; i<tail; i++)
        if (stricmp(sname, ContactBook[i].name)== 0) {
            printf("phone = %d\n", ContactBook[i].phone);
            printf("email = %s\n", ContactBook[i].email);
            return 0;
        }
    printf("The name does not exist\n");
    return -1;
}
```

KH ALL RIGHTS RESERVED

In the next chapter, we will discuss in detail the three different memory areas: static, stack, and heap, the mechanisms for allocating memory from these three areas, as well as how memory is deallocated (garbage collection).

2.5.4 Linked list using dynamic memory allocation

The advantage with static memory allocation is that the memory for the variables is already available when we want to store data in them. The problem is that we need to know the maximum number of elements in advance, which is possible in some cases and not possible in some other cases. If we overestimate the data amount, we waste memory. If we underestimate the data amount, we have to stop the program, modify the `max` value, and recompile the program. To solve this problem, we can use **dynamic memory allocation** that allocates memory to variables during the execution by a function call.

In C, the function that dynamically allocates memory is

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

The function takes one parameter that is of type `size_t`. The type `size_t` is usually an `unsigned int`. The parameter specifies the number of bytes to be allocated. For example, if you need a memory location for an integer variable, then you can call

```
p = malloc(4);
```

However, this statement will work only on a machine that uses 4 bytes for an integer. If you run your program on another machine with a different word-length, the statement will cause a problem. A better way to allocate memory for a given type of variable is to call `malloc (sizeof(type_name))`. For example, if you need memory for an integer variable, it is better to do

```
p = malloc(sizeof(int));
```

The function `malloc` returns a pointer to the initial address of the memory. If the runtime system runs out of memory, it returns `null`.

Please notice that the notation "`void *`" means here that the `malloc` function returns a generic pointer that can point to a variable of any data type. This is possible because all pointer types are structurally equivalent and C mainly uses structural type equivalence in its type checking. Of course, you can also make an explicit type casting to convert the generic pointer type to the specific type, for the purpose of readability, for example:

```
p = (int *) malloc(sizeof(int));
```

casts the return value to an integer type pointer.

In C++, a new dynamic memory allocation operator has been introduced:

```
class_name p = new class_name;
```

The `new` operator allocates the right amount of memory for a variable (object) of the given class and returns a pointer of that class. Java uses a similar operator to dynamically allocate memory. The `new` operator will be explained in more detail in the next chapter.

Since the `malloc` function returns a generic pointer, we often combine dynamic memory allocation with pointers to define a collection of structures. The following example re-implements the array of structures using dynamic memory allocation.

```
/* This program demonstrates how to define a linked list of structures.
   It dynamically allocates memory for the variables of structure type.
   Only the parts that are different from the array of structure example
```

KH ALL RIGHTS RESERVED

```
  are given here. */
#include <stdio.h>
#include <stdlib.h>      // used for malloc
struct Contact {         // define a node holding a person's detail
  char name[30];
  long phone;
  char email[30];
  struct Contact *next; // pointer to Contact structure
} *head = NULL;          //head is a global pointer to first entry
void branching(char c);  // function forward declaration
int insertion();
int search();
// void deletion();
// void printall();

int insertion() {        // insert a new entry at the beginning
  struct Contact *p;
  p = (struct Contact *) malloc(sizeof(struct Contact));
  if (p == 0) {
      printf("out of memory\n");  return -1;
  }
  printf("Enter name, phone, email \n");
  scanf("%s", p->name);
  scanf("%d", &p->phone);
  scanf("%s", p->email);
  p->next = head;
  head = p;
  return 0;
}
int search() {   // print phone and email via name
  char sname[30];
  struct Contact *p = head;
  printf("please enter the name to be searched\n");
  scanf("%s", sname);
  while (p != 0)
      if (strcmp(sname, p->name)== 0) {
           printf("phone = %d\n", p->phone);
           printf("email = %s\n", p->email);
           return 0;
      }
      else p = p->next;
  printf("The name does not exist\n");
  return -1;
}
```

KH
ALL RIGHTS
RESERVED

In the example, the `Contact` type is redefined with an extra field `next`:

```
struct Contact *next; // pointer to Contact structure
```

The `next` field is a pointer to a `Contact` type variable. We use it to form a linked list. Please note that we need to use the keyword **`struct`** whenever we refer to a structure type.

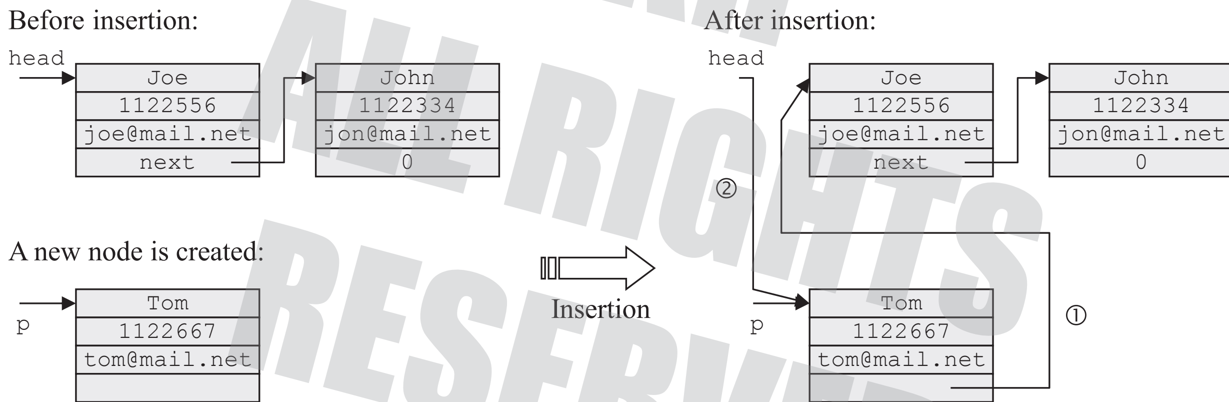
In the insertion function, we use

```
p = (struct Contact *) malloc(sizeof(struct Contact));
```

to allocate the right amount of memory for a variable of `Contact` type, and we link the initial address of this memory chunk to a pointer variable `p`. The type casting makes it clearer that the memory is allocated for a `Contact` type variable. Using malloc(sizeof(struct Contact)), instead of using malloc(68), can avoid calculation error, and particularly, avoid calculating the padding bytes.

Figure 2.14 illustrates the insertion process. Assume that the linked list already has two nodes and a new node is being inserted.

This insertion function inserts the new node at the beginning of the linked list. You can also insert the new node at the end (or at any required position). In this case, you can use a temporary pointer, say `temp`, and move temp to the last node before performing insertion, as shown in the following code.



KH ALL RIGHTS RESERVED

Figure 2.14. Insert a new node at the beginning of a linked list.

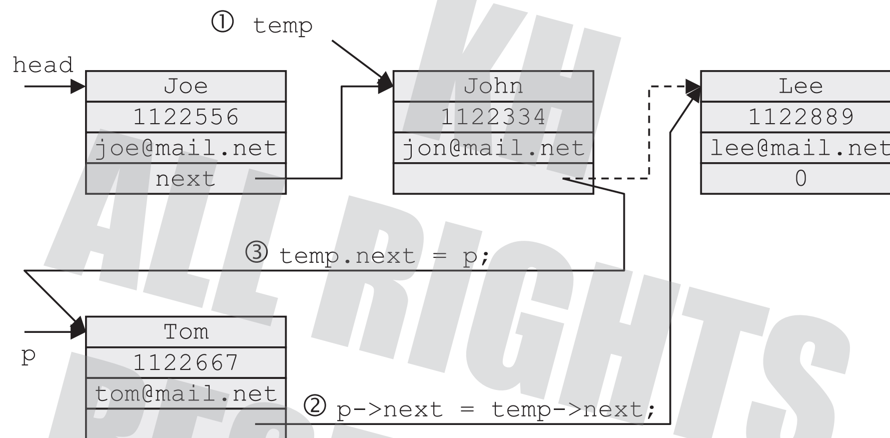
```
int insertion_at_end() {      // insert a new entry at the end
   struct Contact *p, *temp;
   p = (struct Contact *) malloc(sizeof(struct Contact));
   if (p == 0) {
        printf("out of memory\n");  return -1;
   }
   printf("Enter name, phone, email \n");
   scanf("%s", p->name); scanf("%d", &p->phone); scanf("%s", p->email);
   p->next = 0;
   if (head == 0) head = p;
   else {
        while (temp->next != null)
              temp = temp->next;       // Find the last node
        temp->next = p;                // Link the new node
```

```
    }
}
```

Generally, a node can be inserted in any position in a linked list. Figure 2.15 illustrates the insertion process. It consists of three steps: (1) Find the position where the new node is to be inserted. Use a temporary pointer variable `temp` to point to this position. (2) Set the new node's next pointer to the node next to the node pointed to by `temp`. (3) Set the next pointer of the node pointed to by `temp` to the new node.

In the earlier example of the array of structures, we used the dot-notation to access the data field of a structure variable. It is different when referring to a data field of a structure pointed to by a pointer variable. We use the **arrow operator** (sometimes called pointer-to-member operator) instead; that is, we use

```
p->name
p->phone
p->email
p->next
```



KH ALL RIGHTS RESERVED

Figure 2.15. Insert a new node in the middle of a linked list.

to access the four fields of a `Contact` structure variable pointed to by `p`. The differentiation is necessary because their meanings are different. Examine the piece of code:

```
struct Contact *p, q;
p = (struct Contact *) malloc(sizeof(struct Contact));
strcpy(p->name, "smith");     // or strcpy(*p.name, "smith");
strcpy(q.name, "miller");
free(p);    // return the memory allocated by malloc to the memory heap
p = &q;     // p is now pointing to variable q.
```

In the example, `p` is a pointer to a `Contact` structure variable and `p` has only 4 bytes of memory allocated, while `q` is the name of a variable of `Contact` type, as shown in Figure 2.16. The compiler has allocated the entire memory that can hold all four data fields to `q`. Thus, we can directly copy a name "`miller`" into the name field of `q`. Before we can copy anything into the variable pointed to by `p`, we must first use `malloc` to obtain the memory for that variable.

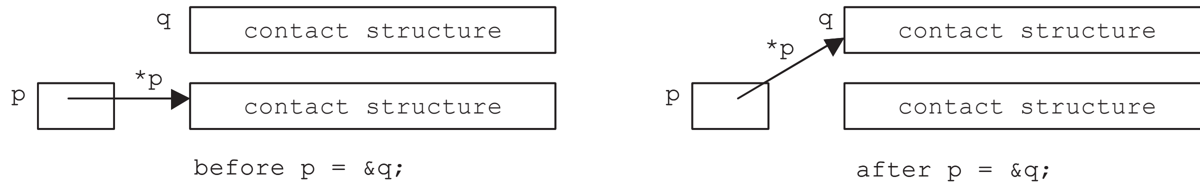


Figure 2.16. Variable `q` is allocated statically, while the variable pointed to by `p` is dynamically allocated.

The last statement in the example assigns the address of `q` to `p`. Now `p` is pointing to the variable `q`. In other words, now `q` has another name which is `*p`. If we do not free (delete) the `Contact` variable pointed to by `p` before we assign the address of `q` to `p`, the variable will be completely inaccessible and becomes a piece of **garbage**. The "`free`" function is, in fact, doing the job of garbage collection.

The function `free`(`p`) is the opposite of the function p = `malloc(size)`, in that it returns the memory linked to `p` to the **heap**, the pool of free memory. If we keep using `malloc` to get memory from the heap, but do not collect the garbage, the heap will eventually be empty and we thus run out of memory. Not collecting garbage is also called a **memory leak**. The following examples show memory leak when deleting a linked list.

KH ALL RIGHTS RESERVED

Assume that you want to delete the entire linked list pointed to by head. If you simply assign `head = null`, the linked list becomes an empty list. However, the memory used by all the nodes in the linked list becomes uncollectable garbage. What is the result of the following operations?

```
free(head);
head = null;
```

This snippet of code will free the memory used by the first node only. The memory used by all the other nodes will become uncollectable garbage. The correct way of deleting the entire linked list is to use a loop to free each and every node, as shown in the following snippet of code.

```
temp = head;
while (temp != null) {
   temp = temp->next;
   free(head);
   head = temp;
}
```

Garbage collection and memory leak will be discussed in the section on memory management in the next chapter.

2.5.5 Doubly linked list

When traversing a linked list, you can easily move a temp pointer forward from the head pointer to the end of the linked list. However, you cannot move the pointer backward along the linked list. If you need a data structure that can move both forward and backward, doubly linked list is a good solution.

A doubly linked list node has two pointers pointing to the previous node and the next node. The following code shows a simple example of a doubly linked list. An insertion function is given, which inserts a new node at the sorted place by name. The id of the Node `struct` is generated automatically, and the names are entered from the keyboard. This program can be combined with the complete program of the singly linked list to allow full functionalities.

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

```
#include <stdlib.h>
#include<string.h>
#pragma warning(disable: 4996) // disable warning in Visual Studio
struct Node {
  int id;                    // int size is 4 bytes
  char *name;                // name is a pointer, not an array
  struct Node* previous;     // pointer to previous node
  struct Node* next;         // pointer to next node in list
};
struct Node *head = NULL, *tail = NULL;
int insertion(int i, char* n) {
  struct Node *temp = (struct Node*)malloc(sizeof(struct Node));
  if (temp == NULL) {
      printf("out of memory\n"); return -1;
  }
  temp->id = i;
  temp->name = n;
  if (head == NULL) {  // Case 0: the linked list is empty
      head = temp;
      head->next = NULL;
      head->previous = NULL;
      tail = temp;
      return 0;
  }
  else {  // Case 1: The list is not empty, insert at the beginning
      if (strcmp(temp->name, head->name) < 0) {
          temp->next = head;
          head->previous = temp;
          head = temp;
          head->previous = NULL;
          return 1;
      }
  };
  struct Node *iterator = head;
  struct Node *follower = iterator;
  while (iterator != NULL) {  // Case 2
      if (strcmp(temp->name, iterator->name) < 0) {
          temp->next = iterator;
          iterator->previous = temp;
          temp->previous = follower;
          follower->next = temp;
          return 2;
      }
      follower = iterator;
```

KH ALL RIGHTS RESERVED

```
      iterator = iterator->next;
  }
  follower->next = temp;   // Case 3
  temp->previous = follower;
  temp->next = NULL;
  tail = temp;
  return 3;
}
int main() {
  int identity = 0;
  char *name1 = malloc(32);
  char *name2 = malloc(32);
  char *name3 = malloc(32);
  struct Node *temp1, *temp2;
  printf("Please enter 3 names:\n");
  scanf("%s", name1);   // enter John
  scanf("%s", name2);   // enter Mary
  scanf("%s", name3);   // enter David
  insertion(identity++, name1);
  insertion(identity++, name2);
  insertion(identity++, name3);
  temp1 = head;
  temp2 = tail;
  printf("ID = %d, name = %s\n", temp1->id, temp1->name);
  printf("ID = %d, name = %s\n", temp1->next->id, temp1->next->name);
  printf("ID = %d, name = %s\n", temp2->id, temp2->name);
  return 0;
}
```

KH
ALL RIGHTS
RESERVED

Notice that dynamic memory is used for `name1`, `name2`, and `name3` in the main program. It is necessary in order to keep the memory in the linked list. If a local variable is used for the names, the memory will go out of scope when function exits. Memory management will be discussed in detail in the next chapter. The output of the program is as follows:

```
Please enter 3 names:
John
Mary
David
ID = 2, name = David
ID = 0, name = John
ID = 1, name = Mary
```

2.5.6 Stack

A **stack** is a data structure that can contain a set of ordered items. The items are ordered in such a way that an item can be inserted into the stack or removed from the stack at the same end. This end is called the top of the stack.

The stack is one of the most important data structures in computer hardware and software design. This section introduces the basic concept of stack through an example. More applications of the stack will be further discussed in Chapter 3 when we study memory management, in Section A.2 in Appendix A when we introduce basic computer architectures, and in A.3 when we discuss the implementation of function calls at the assembly language level.

Like any structured data type or a data structure, a stack is defined on simpler data types and the new operations on the data types.

Typically, a stack is defined on an array type. The basic operations defined on the stack are **push** (add an element onto the stack top) and **pop** (remove the top element from the stack). The code below shows the definition of a stack:

```
elementType stack[stackSize];
int top = 0;
void push(elementType Element) {
  if (top < stackSize) {
       stack[top] = Element;
       top++;
  }
  Printf("Error: stack full\n");
}
elementType pop() {
  if (top > 0) {
       top--;
       return stack[top];
  }
  Printf("Error: stack empty\n");
}
```

Now we use the stack to implement a four-function calculator that supports addition, subtraction, multiplication, and division operations on floating point numbers. The basic part of the implementation is the same as the code above, except that the `elementType` is now `float`, and four extra arithmetic functions are included. To perform operations, data are first pushed onto the stack. Every time an operation is performed, the two data items on the stack top are popped out for operation and the result is pushed back onto the stack.

```
#define stackSize 8  // a sample value
#include <stdio.h>
float stack[stackSize];
int top = 0;
void push(float Element) {
  if (top < stackSize) {
       stack[top] = Element;
       top++;
  } else
  printf("Error: stack full\n");
}
float pop() {
```

KH ALL RIGHTS RESERVED

```
    if (top > 0) {
        top--;
        return stack[top];
    } else
    printf("Error: stack empty\n");
}
float add() {
    float y;
    y = pop() + pop(); push(y);
}
float sub() {
    float y;
    y = pop() - pop(); push(y);
}
float mul() {
    float y;
    y = pop() * pop(); push(y);
}
float div() {
    float y;
    y = pop() / pop(); push(y);
}
void main() {
    float x1 = 1.5, x2 = 2.5, x3 = 3.5, x4 = 4.5, x5 = 5.5, x6 = 6.5;
    push(x1); push(x2); push(x3);
    push(x4); push(x5); push(x6);
    add(); sub(); mul(); div(); add();
    printf("final value = %f\n", pop());
}
```

KH ALL RIGHTS RESERVED

What is computed in the main program by the sequence of operations `add()`, `sub()`, `mul()`, `div()`, and `add()`? Figure 2.17 shows the stack status after each `push` operation and after each arithmetic operation. Initially, stack `top` = 0. It increments after each `push` operation. In each arithmetic operation, two `pop` operations and one `push` operation are performed, resulting in the `top` being decreased by one. The final value computed is 12.0. After the `pop` operation performed in the `printf` statement, the `top` returns to zero.

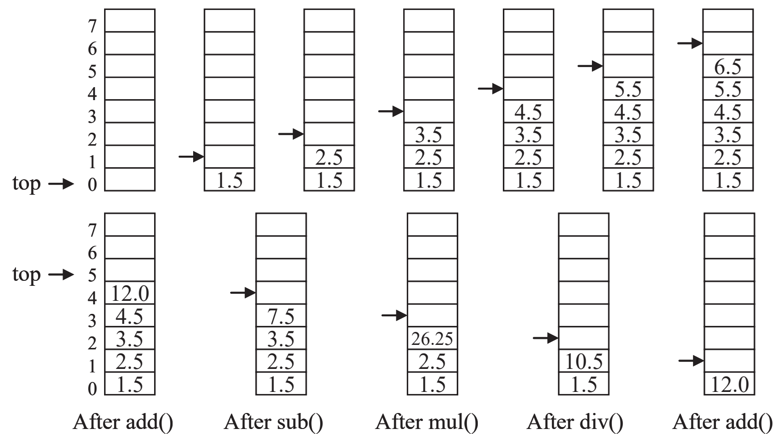


Figure 2.17. Stack status after each operation.

2.6 Standard input and output, files, and file operations

So far, we have discussed using memory (variables) to store data. However, memory is only a temporary place to store data. When we quit a program, all memory allocated to the program is taken back by the operating system for reuse. If our program has data that need to be stored for future use, we need to store the data into the permanent storage of a computer—the disk.

2.6.1 Basic concepts of files and file operations

Data stored on disk are organized in **files**. We consider a file as a structured data type and we access data in a file using a pointer to an object of type FILE, which records whatever information is necessary to control the stream of data.

As we know that disk operations are extremely slow, million times slower than memory operations, as it involves mechanical rotations of the disk and sliding of the read/write heads. The challenge is to make file operations faster. The solution is to use a buffer in the memory to hold a large block (e.g., 1024 bytes) of data. Each disk operation will transfer a block of data, instead of a byte or a word of data. Figure 2.18 shows how read and write operations are implemented.

For the read operations, the process is as follows:

- Declare a pointer f to a FILE type;
- Open a file for read: Create a buffer that can hold a block of bytes (e.g., 1024 bytes);
- Copy the first block of a file into the buffer;
- A program uses the pointer to read the data in the buffer;
- When the pointer moves down to the end of the buffer, copy the next block into the buffer;
- Close the file at the end of use.

For the write operation, the process is as follows:

- Declare a pointer f to a FILE type;
- Open a file for write: Create a buffer that can hold a block of bytes (e.g., 1024 bytes);
- A program uses the pointer to write the data in the buffer;
- When the buffer is full, copy the block into the disk;
- Move the pointer to the beginning for more write-operations;

KH ALL RIGHTS RESERVED

- Close the file at the end of use.

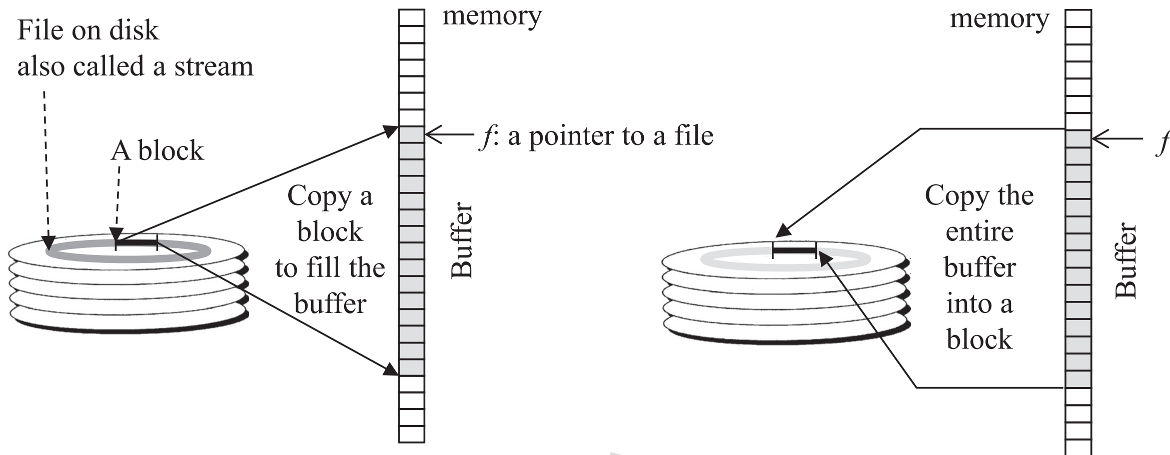


Figure 2.18. File read and write operations.

2.6.2 File operations in C

We focus on C file operations in this section, and we will discuss the C++ file operations in the next chapter. We will use the following example to demonstrate the basic file operations, including opening, reading, writing, and closing a file.

```
// demonstrate the use of fopen, fclose, feof, fgetc and fputc operations
#include <stdio.h>
#include <string.h>
// This function reads all characters in a file and puts them in a string
void file_read(char *filename, char *str) {
  FILE *p;
  int index=0;
  p=fopen(filename, "r");    // Open the file for "read".
  // Other options are "w" (write), "a" (append), and "rw" (read & write).
  while(!feof(p))// while not reaching the end-of-file character
       *(str+index++)=fgetc(p); //read a character from file and put it
                      // in str. p is incremented automatically.
  str[index]='\0';   // add the null terminator
  puts(str);         // print str. You can use printf too.
  fclose(p);         // close the file
}
// This function creates a new file (or opens an existing file), and then
// stores (puts) all characters in the string str into the file.
void file_write(char *filename, char *str) {
  int i, l;
  FILE *p;                         // declare a pointer to file type
  p=fopen(filename, "w");          // open/create a file for "write".
  l = strlen(str);                 // get string-length
  for(i=0;i<l;i++)
```

KH ALL RIGHTS RESERVED

```
        fputc(*(str+i),p);          // write a character to the file pointed
                                    // by p. p is incremented automatically.
    fclose(p);                      // Close the file.
}
// This function cipher-encrypts the string in variable str.
void encrypt(int offset, char *str) {
    int i,l;
    l=strlen(str);
    printf("original str = \n%s\n", str);
    for(i=0;i<l;i++)
        str[i] = str[i]+offset;
    printf("encrypted str = \n%s \nlength = %d\n", str, l);
}
void main() {
    char filename[25];
    char strtext[1024];
    printf("Please enter the name of the file to be read\n");
    // you should enter the name of an existing text file, e.g., letter1.txt
    scanf("%[^\n]s", filename);     //Read a line till the end-of-line "\n"
    file_read(filename, strtext);   //read text from file & put it in strtext
    encrypt(5, strtext);            //manipulate the string strtext
    printf("Please enter the name of the file to be written\n");
    scanf("%[^\n]s", filename);     //Read a line till the end-of-line "\n"
    file_write(filename, strtext); //write the text into the given file
}
```

This program first takes a file name from the keyboard, reads the file (we assume the file exists), and puts the entire contents of the file in a string variable `strtext`. Then we call the encrypt function to encrypt the string. Finally, we write the encrypted string into another text file.

In the program, we use the following basic file open operation:

```
    p = fopen(filename, "r");
```

to open the file in "read" mode. The pointer `p` points to the first character in the text file (buffer). Other mode options are "`w`" for "write" and "`a`" for "append" data at the end of the file. In addition to these modes, the following characters can be included in *mode* to specify the translation mode for newline characters: "t": Open in text (translated) mode. In this mode, CTRL+Z is interpreted as an end-of-file character on input. "b": Open in binary (untranslated) mode; translations involving carriage-return and linefeed characters are suppressed. The letter "b" must be placed at the end of the mode string. If you want to open a file for both read and write operations, you can use an "+." Table 2.4 summarizes the mode definitions:

KH ALL RIGHTS RESERVED

Mode	Description
r	Open an existing file for reading.
w	Open a file for writing. If the file does not exist, a new file is created. If the file exists, its content is cleared, and the file is written as an empty.
a	Open a file for appending. If the file does not exist, a new file is created.
r+	Open an existing file for reading and writing.
w+	Same as w mode, but allow both write and read.
a+	Same as a mode, but allow append and read.
b	Binary mode. The letter b can appear at the end or before +, e.g., both w+b and wb+ are acceptable.

Table 2.4. File operation modes.

Having opened a file, we can use the function

```
ch = fgetc(p);
```

to read the first character from the file. After each `fgetc` call, the pointer is automatically moved to the next position, ready for reading the next character. Another function is

```
fputc(ch, p);
```

that puts the character in parameter `ch` into the file at the position pointed to by `p`.

After we have completed file operations (read or write), we must close a file by using the file close operation

```
fclose(p);
```

If a file is not closed, the file descriptor that is used by the operating system to identify the file will not be freed. The total number of file descriptors that an operating system can issue is usually very limited. For example, in the Unix operating system, the file descriptor must be between 0 and 20. File descriptors 0, 1, and 2 are reserved for three system files: standard input, standard output, and standard error output, leaving only 18 file descriptors for all users concurrently using the operating system. If no file descriptors are available, the operating system will not be able to open any file for any user applications.

In the statement `scanf("%[^\n]s," filename)` in the program example above, the control sequence `[^\n]` ensures that the `scanf` reads until the `newline` symbol "`\n`," including the spaces in the line.

The other file operations include:

- fread(buffer, size, count, fileName); // unformatted read
- fwrite(buffer, size, count, filename); // unformatted write
- scanf(control sequence, parameter list); // formatted input from keyboard
- printf(control sequence, parameter list); // formatted output to screen
- fscanf(filename, control sequence, parameter list);
  // This function is the same as scanf, except it inputs from a file
- fprintf(filename, control sequence, parameter list);
  // This function is the same as printf, except it outputs to a file

The definitions of the functions `fread` and `fwrite` are:

```
size_t fread(void *buffer, size_t size, size_t count, FILE *fileName );
size_t fwrite(const void *buffer, size_t size, size_t count, FILE *fileName
);
```

KH ALL RIGHTS RESERVED

```
// Using these two functions requires including a <stdio.h> header
```

The functions are defined in the standard library `stdio.h`. The function `fread` returns the number of full items actually read, which may be less than `count` if an error occurs or if the end of the file is encountered before reaching `count`. You can use the `feof` or `ferror` function to distinguish a read error from an end-of-file condition. If `size` or `count` is 0, `fread` returns 0, and the `buffer` contents are unchanged.

The function `fwrite` returns the number of full items actually written, which may be less than `count` if an error occurs. Also, if an error occurs, the file-position indicator cannot be determined.

The parameters in the functions are specified as follows:

1. buffer: pointer to the source variable (for fwrite) or to the destination variable (for fread)
2. size: item size in bytes
3. count: maximum number of items to be read or written. Normally, use 1.
4. fileName: pointer to FILE structure

The following segment of code shows the application of `fread` and `fwrite`. It consists of two functions. The `save_file` function saves a linked list of nodes into a file called `fileName`, and the `load_file` function reads the file called `fileName`, and rebuilds the linked list according to the saved data. The segment of code demonstrates how to write and read strings and integers to and from a file.

KH ALL RIGHTS RESERVED

```
// demonstrate the use of fopen, fclose fread and fwrite operations
void save_file() {
   FILE *fileName;
   personnel *node;
   char ch;
   long sid;
   fileName = fopen(file_name, "wb"); // w for write, b for binary mode
   if(fileName != NULL) {
        node = head;
        while(node != NULL) {
              fwrite(node->getName(), 30, 1, fileName);
              fwrite(node->getBirthday(), 11, 1, fileName);
              sid = node->getId();
              fwrite(&sid, sizeof(long), 1, fileName); // binary write
              node = node->getNext();
        }
   }
   else
        printf ("ERROR - Could not open file for saving data !\n");
}
void load_file() {
   FILE *fileName;
   personnel *node, *temp;
   char sname[30];
   char sbirthday[11];
```

```
    long sid;
    fileName = fopen(file_name, "rb");    // r for read, "b" for binary
    if(fileName != NULL) {
        while(fread(sname, 30, 1, fileName) == 1) {
            fread(sbirthday, 11, 1, fileName);
            fread(&sid, sizeof(long), 1, fileName); // read binary
            node = new personnel(sname, sbirthday);
            node->setId(sid);
            if(head != NULL)
                temp->setNext(node);
            else
                head = node;
            temp = node;
        }
        fclose(fileName);
    }
}
```

We have used `scanf` and `printf` to read from the keyboard and print to the screen. In fact, in C/C++, the keyboard is considered to be a read-only file (standard input file) and the screen is considered to be a write-only file (standard output file). Their file names are `stdin` and `stdout`, respectively.

The functions `fscanf` and `fprintf` are more general forms of file operations in which we can specify what file we want to read and write. The standard input and output functions `scanf` and `printf` are special cases of them and are equivalent to

```
fscanf(stdin, control sequence, parameter list);
fprintf(stdout, control sequence, parameter list);
```

The following example shows the application of `fprintf` and `fscanf`. First, an integer number and a float number are written in the file named `PersonData`. Then the file is closed and reopened for read. An integer number and a float number are read into two variables `len` and `hei`, respectively:

```
#include <stdio.h>
void main() {
    FILE *fileID;
    int length = 35429, len;
    float height = 5.8, hei;
    fileID = fopen("PersonData", "wb");   // open for write
    if(fileID != NULL) {
        fprintf(fileID, "%d\n", length);  // write an integer into a file
        fprintf(fileID, "%f\n", height);  // write a float into a file
    }
    else
        printf ("ERROR - Could not open file for saving data !\n");
    fclose(fileID);
    fileID = fopen("PersonData", "rb");   // open for read
    if(fileID != NULL) {
```

KH ALL RIGHTS RESERVED

```
        fscanf(fileID, "%d", &len);        // read an integer from a file
        fscanf(fileID, "%f", &hei);        // read a float from a file
        printf("length = %d, height = %f\n", len, hei);
    }
    fclose(fileID);
}
```

In C++, similar input and output functions are defined:

```
cin >>
cout <<
cin.ignore();
cin.get(strvar, strlength, achar);
cin.getline(strvar, strlength, achar);
```

More details of C++ input, output, and file operations will be discussed in Chapter 3.

2.6.3 Flush operation in C

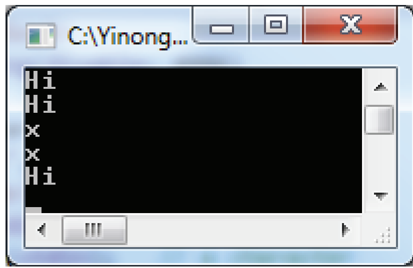
In the aforementioned programs, we used `fflush(stdin)` to remove the delimiter (a space, a newline, etc.) before using `getc(stdin)`. The reason is, the formatted input function `scanf` will read only up to the delimiter and leave the delimiter in the input buffer. If we do not call `fflush(stdin)`, the left delimiter will be read by an unformatted input function such as `getc(stdin)` and `gets(stdin)`. Thus, we must call function `fflush(stdin)` to flush the buffer of the standard input file `stdin`. It is not a problem if two consecutive `scanf` functions are called because a formatted input function can automatically remove the delimiter. The C++ function equivalent to `fflush` is `cin.ignore`, which will be discussed in Chapter 3.

C-styled `fflush(stdin)` function that flushes the input buffer to remove the remaining delimiters in the buffer of the standard input file `stdin` after a `scanf` operation. Consider the following snippet of code:

```
#include <stdio.h>
#pragma warning(disable: 4996) // comment out if not in Visual Studio
int main() {             // Test fflush() function
    char strvar[8], ch;
    scanf("%s", strvar);   // Enter: Hi
    printf("%s\n", strvar);
    //fflush(stdin);       // Try the program with and without fflush()
    ch = getc(stdin);      // enter a character ‘x’
    printf("%c\n", ch);
    printf("%s\n", strvar);
}
```

The inputs and outputs of the program with the fflush() (commented out) and without the fflush, respectively, are shown as follows:

KH
ALL RIGHTS
RESERVED

With fflush: 

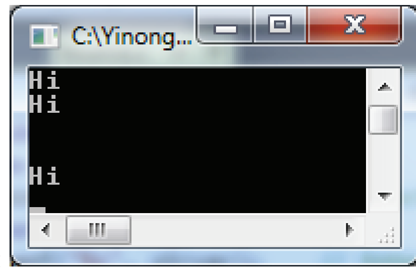
Without fflush: 

Figure 2.19 explains the reason why the character read getc(stdin) does not read the character into the variable `ch`, by illustrating the states of the input variable `strvar` and the input buffer in the execution process of the program above. Notice that the newline character '`\n`' is left in the buffer after the `scanf` operation is completed, and thus the next input character is appended to the character. The `getc` function reads a character from the input buffer, which will read '`\n`', without giving a chance to read the keyboard. As a result, no input is needed, '`\n`' is read into the input variable `ch`, and thus the keyboard operation is skipped.

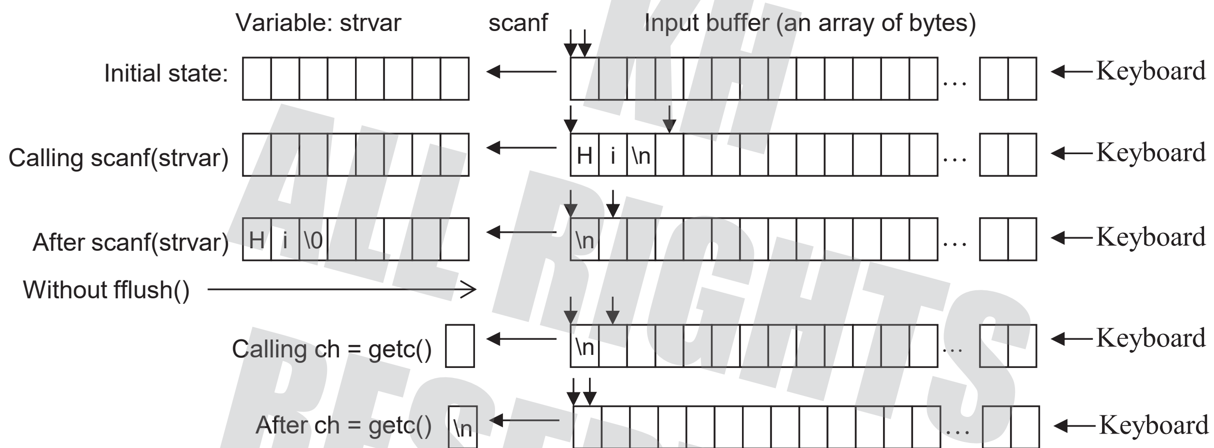


Figure 2.19. The input buffer between the keyboard and the input variable.

KH ALL RIGHTS RESERVED

If the library function `fflush` does not exist in your environment, you can also write your own flush function to flush all the characters in the file buffer. The following snippet of code shows a simple implementation:

```
void myFlush(){ // Manually flush all characters in the stdin buffer
  int c;
  do {
      c = getchar();
  } while (c != '\n' && c != EOF); // EOF: End of File flag
};
```

When you switch back from unformatted input to formatted input, you normally do not need to put the delimiter back. However, it is a good idea to restore the character that is flushed. You can call the library function `ungetc('\n', stdin)` to put the newline character back into the buffer.

2.7 Functions and parameter passing

Functions, also called procedures or subroutines in some other programming languages, are named **blocks** of code that must be explicitly called. The purpose of functions is twofold:

- **Abstraction**: Statements in a function form a conceptual unit.
- **Reuse**: Statements in a function can be executed multiple times in the program.

As a part of a program, a function must communicate with the rest of the program. To pass values into a function (**in-passing**), we usually have two methods: global variable and parameter passing. To pass values out of a function (**out-passing**), we usually have three methods: global variable, parameter passing, and return value. Different programming languages have different value passing policies and mechanisms.

- In imperative and object-oriented programming languages like C/C++ and Java, all combinations of the in-passing and out-passing methods are allowed.
- In functional programming languages like Scheme or Lisp, parameter passing is the only in-passing method and return value is the only out-passing method allowed.
- In logic programming languages like Prolog, parameter passing is the only in-passing and the only out-passing method allowed.

Using a global variable to pass a value in or out of a function causes unwanted side effects and thus it is generally not recommended to use global variables for passing values. It is conceptually simple to use a return value to pass a value out of a function. We will thus focus on the parameter-passing mechanisms that pass values in and out of functions.

When we discuss parameter passing, we need to differentiate two kinds of parameters: formal parameters and actual parameters. **Formal parameters** are the parameters we use when we declare (define) a function. Formal parameters are local variables of the function. **Actual parameters** are the values or variables we use to substitute for the formal parameters when we call a function. Actual parameters are variables/values of the caller before the control enters the function. They become the variables/values of the function after the control enters the function.

Now the question is what would happen if we modify the formal parameters in the function. Will the modification have an impact on the actual parameters? The answer to the question depends on what kind of parameter-passing mechanisms we use. The most frequently used parameter-passing mechanisms are call-by-value, call-by-alias, and call-by-address.

Call-by-value: The formal parameter is a local variable in the function. It is initialized to the value of the actual parameter. It is a copy of the actual parameter. The modification of formal parameters has no impact on the actual parameters. In other words, call-by-value can only pass values into a function, but it cannot pass values outside the function. Functions using call-by-value must use return-value to pass a value to the outside. The advantage of call-by-value is that it has no side effects and it is considered a reliable programming practice. The drawback is that it is not convenient to handle structured data types.

The following piece of code demonstrates value in-passing by global variable and call-by-value mechanisms:

```
#include <stdio.h>
int i = 1;               // i is a global variable outside any function
foo(int m, int n) {      // m and n are formal parameters
  printf("i = %d m = %d n = %d\n", i, m, n);
  i = 5; m = 3; n = 4;   // Modify i, m and n.
```

KH ALL RIGHTS RESERVED

```
  printf("i = %d m = %d n = %d\n", i, m, n);
}
main() {
  int j = 2;            // j is a local variable, local to main() function
  foo(i, j);            // i and j are actual parameters of function foo
  printf("i = %d j = %d\n", i, j);
}
```

The output of the program is

```
i = 1  m = 1n = 2
i = 5  m = 3n = 4
i = 5  j = 2
```

As you can see, the global variable `i` is changed in the function and `i` remains changed after leaving the function. On the other hand, `j` is passed to formal parameter `n` and `n` is modified in the function. The modification to `n` has no impact on `j`.

Call-by-alias: It is also called call-by-reference or call-by-variable. The formal parameter is an *alias* name of the actual parameter. Call-by-alias can pass a value into and out of a function. However, it has a side effect, that is, a variable outside a function can be changed by an action in a function.

For call-by-alias, there is only one variable (memory location) with two names for the formal and actual parameters, respectively. Changing the formal parameter immediately changes the actual parameter. The actual parameter must be a variable. It cannot be a literal value because a value cannot have an alias. This mechanism is supported by C++, but not by C.

To declare a formal parameter `x` in call-by-alias, an ampersand symbol is prefixed to the parameter: `&x`. The following code demonstrates parameter passing by the call-by-alias mechanism, where the second parameter of the `foo` function is an alias to the corresponding actual parameter:

```
#include <iostream>
void foo(int, int &);   // forward declaration
int i = 1;
void main() {
  int j = 2;            // j is a local variable, local to main() function
  foo(i, j);            // i and j are actual parameters of function foo
  printf("i = %d j = %d\n", i, j);
  foo(j, i);            // i and j are swapped
  printf("i = %d j = %d\n", i, j);
}
void foo(int m, int &n) { // call-by-alias is applied to parameter n
  printf("i = %d m = %d n = %d\n", i, m, n);
  i = 5; m = 3; n = 4;    // Modify i, m and n.
  printf("i = %d m = %d n = %d\n", i, m, n);
}
```

The output of the program is

```
i = 1 m = 1 n = 2
```

KH ALL RIGHTS RESERVED

```
i = 5 m = 3 n = 4
i = 5 j = 4          // notice that j is changed
i = 5 m = 4 n = 5
i = 4 m = 3 n = 4    // notice that i is changed immediately
i = 4 j = 4
```

This program is basically the same as the call-by-value example, except that the modification to the variable `n` in the `foo` function is passed to the variable j in the main program, in the first call, and is immediately passed to i in the second call to `foo`.

Call-by-address: It is also called call-by-pointer. The address of the actual parameter is passed into a local variable of the function. The actual parameter can be an address value or a pointer variable. You can use the address to modify directly the actual parameter pointed to by the address. You can also modify the address value stored in the formal parameter. However, this modification will not modify the actual parameter. In fact, for the pointer variable itself, call-by-value is applied. In the following example, we demonstrate parameter passing by call-by-address.

```
#include <stdio.h>
void foo(int *n) {            // declare call-by-address parameter
  printf("n = %d\n", *n);    // print the variable value pointed to by n
  *n = 30;                   // modify the variable value pointed to by n
  printf("n = %d\n", *n);    // print the variable value pointed to by n
again
  n = 0;                     // Modify the pointer itself.
}
void main() {
  int i = 15;
  foo(&i);
  printf("i = %d\n", i);
  i = 10;
  foo(&i);
  printf("i = %d\n", i);
}
```

KH ALL RIGHTS RESERVED

The output of the program is

```
n = 15
n = 30
i = 30
n = 10
n = 30
i = 30
```

In the main program, we have a local variable `i`, initialized to 15. We then call function `foo` using `&i`, the address of variable `i`. The actual parameter is the address value (a pointer) to `i`, not the variable `i` itself. In the function definition body, the formal parameter is a pointer to an integer type. We use the pointer `n` to modify the variable `*n` pointed to by `n`, which is in fact `i`. Thus, we have indirectly modified variable `i` in the `main()` function. When the control exits the function, `i` remains modified. As the last statement in the function, we modify `n` itself. However, this modification will not have an impact on the address value

passed to `n` when the control exits the function. As you can see here, call-by-address and call-by-value are relative. We use call-by-address if our intention is to pass the variable pointed to by the pointer (address value) to the function. If we change the variable through its address, the variable remains changed after exiting the function. On the other hand, if the address value is what we really want to pass into the function, instead of the variable pointed to by the address, we are actually doing call-by-value.

Another application of call-by-address is in the situation where we want to pass a structure variable (an array, a string, or a structure) to a function. It is more convenient to pass the pointer of the structure variable, instead of passing the structure itself.

In Java, parameter passing is limited in such a way that:

- If the parameter is of a primitive type, only call-by-value is allowed;
- If the parameter is of a class-based type, only call-by-address is allowed.

Some books say that Java only supports call-by-value. They are referring to the pointer variable (reference) itself passed to a function. However, since the intention of passing the reference variable is to access the object pointed to by the reference, it is better to say that it supports call-by-address, instead of call-by-value. Notice that the pointer in Java is called "reference." However, Java's parameter passing for objects is not call-by-alias or call-by-reference, according to the definition of call-by-reference here.

In C/C++, all combinations of parameter passing are allowed. You can pass a pointer (call-by-address) or an alias (call-by-alias) to a simple variable (e.g., of integer type), or pass a complex type of variable using call-by-value, call-by-alias, or call-by-address.

The following program shows how to pass a string into and out of a function using call-by-address:

```
// file name: strop.c
#include <stdio.h>
#include <string.h>
char *getString(char *str){  //The function returns a pointer to its
parameter.
  return str;                // returns a pointer.
}
void setString(char *str1, char *str2) {  // Copy str2 into str1
  strcpy(str1, str2);
}
void main() {
  char *p, q[8] = "morning", *s = "hello";
  printf("s = %s\n", s);
  p = getString(s);
  printf("p = %s\n", p);
  setString(q, p);   // q is the address of the array-based string
  printf("q = %s\n", q);
}
```

The output of the program is

```
s = hello
p = hello
q = hello
```

KH ALL RIGHTS RESERVED

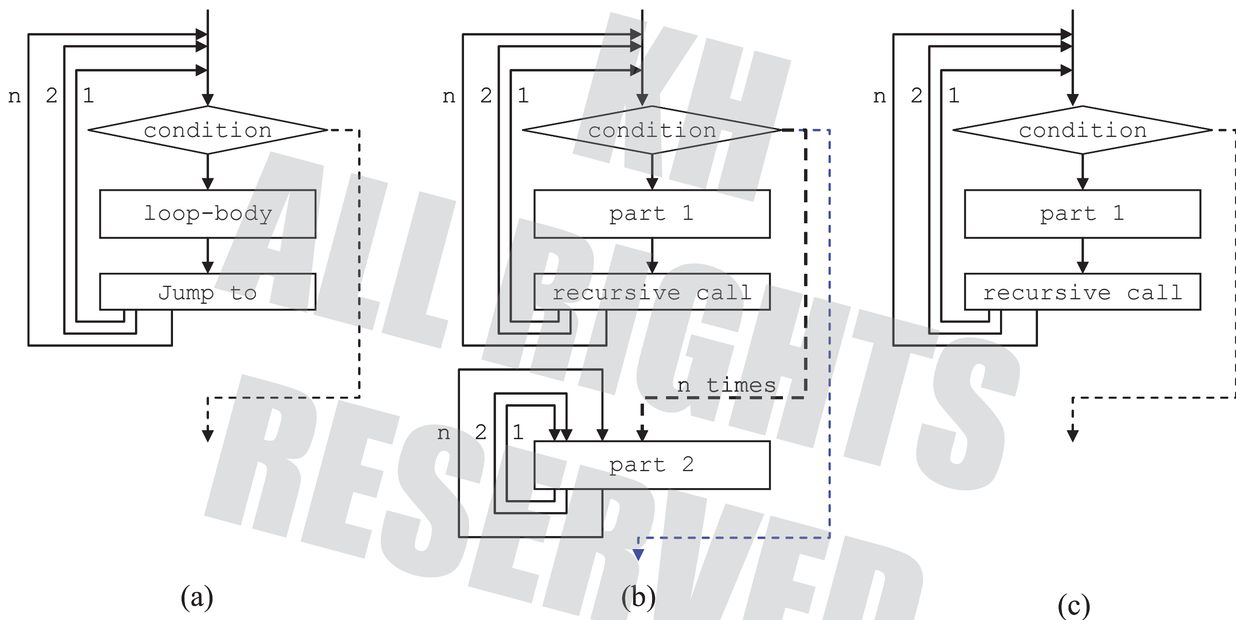
2.8 Recursive structures and applications

Section 2 discussed basic control structures in C/C++. This section studies the complex recursive structures. We first compare recursive structures with the iterative structures. Then we formulate the generic steps of writing recursive functions. Finally, we use a longer example as a case study to go through the design steps. More examples of recursion will be studied in Chapters 4 and 5.

2.8.1 Loop structures versus recursive structures

A function (or procedure) is said to be **recursive** if the function calls itself. A recursive function can call itself anywhere, except the first statement, and once or multiple times, in the body of the function. If a recursive function calls itself only once and in the last statement, the function is said to be **tail-recursive**. Tail-recursion has the simplest structure.

Figure 2.20 compares three different repetition structures: (a) while-do-loop, (b) nontail-recursion, and (c) tail-recursion.



KH ALL RIGHTS RESERVED

Figure 2.20. Three different repetition structures.

Although other loop structures do exist, like for-loop and do-while-loop, while-do-loop is sufficient to implement other possible loop structures. The nontail-recursion breaks its loop-body down into two parts, separated by the recursive call. Part 1 is first repeatedly executed `n` times, and then part 2 is executed `n` times. It is important to recognize that part 2 is executed the same number of times as part 1. The partially completed computations are stored on the stack. When part 1 is eventually repeated, the sufficient number of times or the stopping condition is satisfied, the control exit part 1 and enters part 2. Then part 2 will be executed the same number of times, and finally exit at the end of part 2.

In the case of tail-recursion, the recursive call is the last statement, and thus, there is no part 2. As we can see, the general recursive structure is very different from the iterative loop structure. However, the tail-recursive structure has exactly the same control structure as the while-do-loop. In other words, the while-do-loop structure is a special case of the recursive structure. We will see in Chapter 4 that functional program languages can use recursion as their only repetition structure, completely removing loop structures from the languages.

Here we are taking a glass-box approach to understand the recursive function; that is, we try to study recursion by trying to understand the structure and the control flow of the function. This is one of the most common approaches taken by many programmers. It works fine for simple recursive functions. However, if a recursive function has multiple recursive calls in its body, the structure will be far too complex to understand. We will take an innovative approach in this book to study the recursive function: the black-box approach, or so-called **abstract approach**. This approach works fine for both simple and complex recursive functions. We will see soon that this approach is far easier to understand and to apply to solve all kinds of recursive problems in all possible programming languages.

2.8.2 The fantastic-four abstract approach of writing recursive functions

The idea of recursion may not be as straightforward as iterative looping. However, writing recursive functions can be as simple as writing iterative functions, as long as you strictly follow the **fantastic-four abstract approach**. The approach was first proposed by one of the authors in the first edition of this book and was called "simple steps for writing recursive procedures." The approach has been rated by all students who learn it to be the most efficient method of teaching and learning recursion and was called by the students "fantastic-four." In the second edition, it is formally named the fantastic-four abstract approach, which consists of the following steps:

1. **Formulate the size-n problem:** Recursion is necessary only if you want to solve a problem that needs to repeat the same operations for a number of times. We assume the number of repetition is `n`. In most cases, `n` is obvious. For example, if we want to compute factorial `n!`, the size `n` is already given. Formulating a size-n problem is merely choosing a function name, using `n` as the parameter, and defining the return type (not the return value) of the function. It is similar to writing the forward declaration of a function in C. Thus, the size-n problem for a factorial problem is

```
int factorial(int n);
```

The return value of the size-n problem is what the function is supposed to compute, or the value we are looking for. In this step, we do not need to design the solution for the size-n problem.

2. **Find the stopping condition and the corresponding return value:** The body of a recursive function should begin with checking the stopping condition. If the stopping condition is true, the function returns the corresponding value and exits. Otherwise, it calls the function itself. In most cases, identifying the stopping condition and corresponding value is trivial or given. For example, the stopping condition of `factorial(n)` is `n = 0`, and the corresponding value is 1.

3. **Select `m` and formulate the size-m problem:** After we have formulated the size-n problem, the size-m problem is easy: We simply replace parameter `n` by `m` in the size-n problem, where `m < n`. Size m is determined by how much we can reduce the size of the problem in one iteration. If we can only reduce the problem size by 1, `m` is `n−1`, and thus our task in this step is formulating a size-(n−1)problem. For example, the size-(n−1) factorial problem is simply `factorial(n−1)`. Sometimes, we may need to find an `m` that is not `n−1`. It is application-specific to find a proper `m`. We will use several examples to illustrate this point in this section and study many more examples in Chapter 4. Most students who have difficulty comprehending recursion misunderstand this step: They try to define a solution, or the return value, of size-m problem here in this step! It is not possible and it is not necessary to produce the return value in this step. All we need to do about the return value here is exactly the same as what we did in step 1. We simply assume the size-m problem will return a value and use this value in step 4. For example, the return value of size-(n−1) factorial problem is `factorial(n−1)`.

4. **Construct the solution of the size-n problem:** In this step, we will use the assumed solution or return value for size-m or size-(n−1) problem to construct the solution of the size-n problem. Again, this is

KH ALL RIGHTS RESERVED

application-specific. In the case of the factorial problem, the solution of the size-n problem is `n*factorial(n-1)`.

Sometimes, we may need to use the return values of multiple size-m problems, where $0 \leq m < n$ (assume size-0 is the stopping condition), to construct the solution of the size-n problem.

Strictly following these steps, we can define the complete factorial function as follows:

```
int factorial(int n) {        // size-n problem
  if  (n == 0)                // Stopping condition
       return 1;              // Return value at the stopping condition
  else
       return n * factorial(n - 1); //use size-(n-1) problem's assumed
       // solution to construct size-n problem's solution
}
```

2.8.3 Hanoi Towers

The Hanoi Towers game is a good example used for explaining recursion. As shown in Figure 2.21, the rules of playing the game are:

- There are three pegs, and `n` successively smaller disks are initially placed on the left peg. In the example in Figure 2.21, `n` = 4. The objective is to move all disks to the right peg. The center peg can be used as an auxiliary holding (spare) peg.
- Disks may be moved from one peg to another. Only one disk may be moved at a time.
- The only disks that may be moved are the top disks on one of the three pegs.
- At no time may a larger disk may be placed on a smaller disk.

Now we follow the fantastic-four abstract approach to define a solution for the Hanoi Towers problem.

1. Formulate the size-n problem

We can simply formulate the size-n problem as `void hanoi(int n)`. However, in the return value (solution), we need to print how to move one disk from one peg to another in each step, and we need to name these three pegs. We could hard code the names as p1, p2, and p3; or left, center, and right. To increase the flexibility of the code, we add three parameters to the function, so that the user can pass different names into the function. Thus, we formulate the problem as

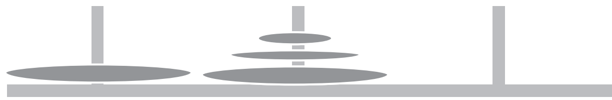
```
void hanoitowers(int n, char *left char *center char *right);
```

Notice that the function does not return a value; instead, it prints instructions (steps) for how to move `n` disk from the `left` peg, using the `center` peg as the auxiliary, to the `right` peg.

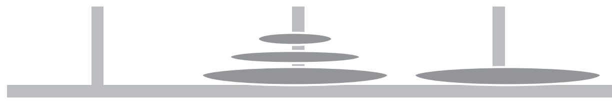
KH ALL RIGHTS RESERVED

Initial state:
n disks on left peg

Step 1:
Move n-1 disks
to the center peg



Step 2:
Move 1 disk
to the right peg



Step 3:
Move n-1 disks
to the right peg

Figure 2.21. Solving Hanoi Towers problem.

KH
ALL RIGHTS
RESERVED

2. Find the stopping condition and the corresponding return value

The stopping condition is `n = 1`. In this case, the size-1 problem is `hanoitowers(1, left, center, right)`, and the solution is to print "move the disk from the `left` peg to the `right` peg."

3. Select m and formulate the size-m problem

Since we can move only one disk at a time, it is obvious that we can only reduce the size by one in one iteration. Furthermore, since we have multiple parameters in the function, we could have multiple size-(n−1) problems. The following are six possible size-(n−1) problems:

(1) move `n-1` disks from `left` to `right`, using `center` as auxiliary:

```
hanoitowers(n-1, left, center, right)
```

(2) move `n-1` disks from `left` to `center`, using `right` as auxiliary:

```
hanoitowers(n-1, left, right, center)
```

(3) move `n-1` disks from `center` to `left`, using `right` as auxiliary:

```
hanoitowers(n-1, center, right, left)
```

(4) move `n-1` disks from `center` to `right`, using `left` as auxiliary:

```
hanoitowers(n-1, center, left, right)
```

(5) move `n-1` disks from `right` to `left`, using `center` as auxiliary:

```
hanoitowers(n-1, right, center, left)
```

(6) move `n-1` disks from `right` to `center`, using `left` as auxiliary:

```
hanoitowers(n-1, right, left, center)
```

4. Construct the solution to the size-n problem

Use the solutions for size-(n−1) problems to construct the solution for the size-n problem.

Figure 2.21 and the text on the left-hand side showed how we construct the solution for the size-n problem based on the solutions for size-(n−1) and size-1 problems, that is,

```
hanoitowers(n-1, left, right, center)    // move n-1 disks left -> center
hanoitowers(1, left, center, right)      // move 1 disk left -> right
hanoitowers(n-1, center, left, right)    // move n-1 disks left -> center
```

In words, the solution for the size-n problem is: (1) Move `n-1` disks from `left` peg to the `center` peg. We simply assume that we can do it, because it is a size-(n−1) problem. (2) Move the remaining disk from `left` to `right`. (3) Move `n-1` disks from `center` to the `right`.

Once we have designed the solution, we can easily obtain the C program that solves the Hanoi Towers problem as follows:

```
#include <stdio.h>
void hanoitowers(int n, char *S, char *M, char *D) {
  if (n == 1) {          // stopping condition
      printf("move top from %s to %s\n", S, D);
      // output at stopping condition
   } else {              // from size-(n-1) to size-n problem
      hanoitowers(n-1, S, D, M);
      hanoitowers(1, S, M, D);
      hanoitowers(n-1, M, S, D);
   }
}
void hanoi(int n) {      // define a simpler human-interface
  hanoitowers(n, "Left", "Center", "Right");
}
void main() {
  hanoitowers(3, "Source", "Spare", "Destination");
  printf("------------------------------------\n");
  hanoi(4);
}
```

KH ALL RIGHTS RESERVED

In the program, we defined a one-parameter function `hanoi(n)` as a simpler user interface, in case the user wants the hard-coded peg names. The function with more parameters is defined as a recursive function. When the `main()` function is executed, the functions `hanoitowers(3, "Source", "Spare","Destination")` and `hanoi(4)` will be called, resulting in the following output describing how to solve the size-3 and size-4 Hanoi Towers problems:

```
move top from Source to Destination
move top from Source to Spare
move top from Destination to Spare
move top from Source to Destination
move top from Spare to Source
move top from Spare to Destination
move top from Source to Destination
------------------------------------
move top from Left to Center
```

```
move top from Left to Right
move top from Center to Right
move top from Left to Center
move top from Right to Left
move top from Right to Center
move top from Left to Center
move top from Left to Right
move top from Center to Right
move top from Center to Left
move top from Right to Left
move top from Center to Right
move top from Left to Center
move top from Left to Right
move top from Center to Right
```

As you can see from the example, the most important idea of recursive functions is that we simply assume that we have the solution for the size-(n−1) problem and we do not need to solve it. Why does it work? Because the recursive mechanism will actually solve the problem from size-1 upward automatically; that is, it will solve the size-1 problem, then it will use the solution to the size-1 problem to construct the solution to the size-2 problem, and so on. Since we have given the solution to the size-1 problem and we have defined how to find the solution to the size-n problem based on the solution to the size-(n−1), we basically have given solutions to the problem of all sizes of problems!

KH ALL RIGHTS RESERVED

2.8.4 Insertion sorting

Now we will follow the fantastic-four abstract approach to solve the **sorting problem** in a simple way (insertion sorting) to demonstrate recursion. Assume that we have an array containing `n` integers: `A[n]`. The task is to sort the `n` numbers in ascending order.

1. Formulate the size-n problem.

We can simply formulate the size-n problem as

```
int* sorting(int *A, int n);
```

where `A` is the initial address of the array to be sorted and `n` is the size of the array. The function will return the initial address of the sorted array.

2. Find the stopping condition and the corresponding return value.

The stopping condition is `n = 1`. In this case, the size-1 problem is `sorting(int *A, 1)`, and the solution or return value is the address of `A`. `A` is not changed because `A` has only one element and is already sorted.

3. Select m and formulate the size-m problem.

Here we take a simple approach by reducing the size of the problem by 1. Thus, `m = n - 1` and the size-(n−1) problem is

```
sorting(B, n-1);
```

where `B` is the address of an array of size-(n−1). We **assume** `B` will be sorted if we call this function. This is very important!

4. Construct the solution of the size-n problem.

Since step 3 can solve the size-(n−1) problem, it is easy to solve the size-n problem:

(1) We split array `A` into two parts: The subarray of the first n−1 elements is `B` and the remaining element is `x`.
(2) We call the function in step 3 to sort the size-(n−1) array `B`.
(3) We find the right position `p` for inserting `x` into `B`.
(4) We make space for `x` by shifting the elements after position `p` one place right.
(5) We insert `x` at the position `p`.

Figure 2.22 graphically illustrates the four steps of implementing the recursive sorting algorithm.

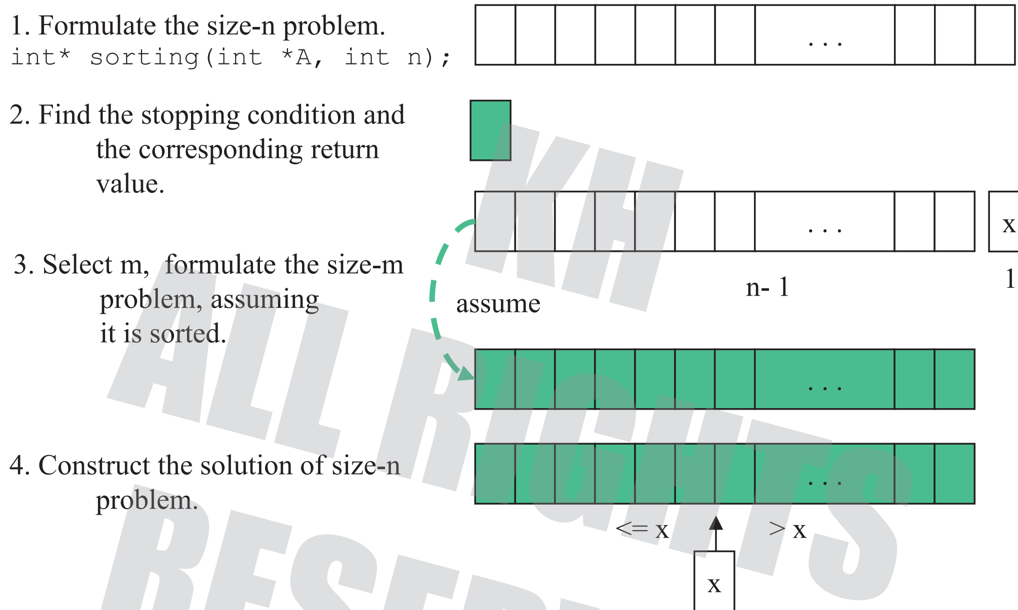


Figure 2.22. The fantastic-four steps sorting an array recursively.

KH ALL RIGHTS RESERVED

The following program implements the four steps in the abstract approach. The comments in the program associate the statements with the four steps described above.

```
#include <stdio.h>
int* sorting(int *A, int n) {
  int *B, i, j, p = n-1, x;
  if (n==1) return A;         // stopping condition and return value
  else {
      x = A[n-1];             // Store the last element in x
      B = sorting(A, n-1);    // size-(n-1) problem
      i = 0;
      while (i < n-1) {       // Start to construct size-n solution
           if (x < B[i]) {
                p = i;  // locate the position p for x
                i = n;  // exit the loop
           }
```

```
            else i++;
       }                  // x should be inserted at position p
       for (j = p; j < n-1; j++)  // make space
            B[n-1-(j-p)] = B[n-1-(j-p)-1];
       B[p] = x;          // put x in the right place
       return B;
   } // end of else branch
}
void main() {
   int *SA, i, k, A[] = {3, 2, 4, 2, 9, 7, 1, 6}; // sample array
   k = (int)sizeof(A)/sizeof(int);    // get the length of the array
   SA = sorting(A, k);
   for (i = 0; i < k; i++)
       printf("%d, ", SA[i]);
}
```

Figure 2.23 illustrates the execution process and the changes of the array. In part 1 of the recursive function (before the recursive call), the array size is reduced by 1 every time the recursive function is called, till n = 0. Since the array index starts from 0, the array has one element when n = 0, as shown on the left-hand side of the figure. On the right-hand side, corresponding to part 2 (after the recursive call), the last element is inserted into the right position to form the size-n problem's solution at each level of recursion.

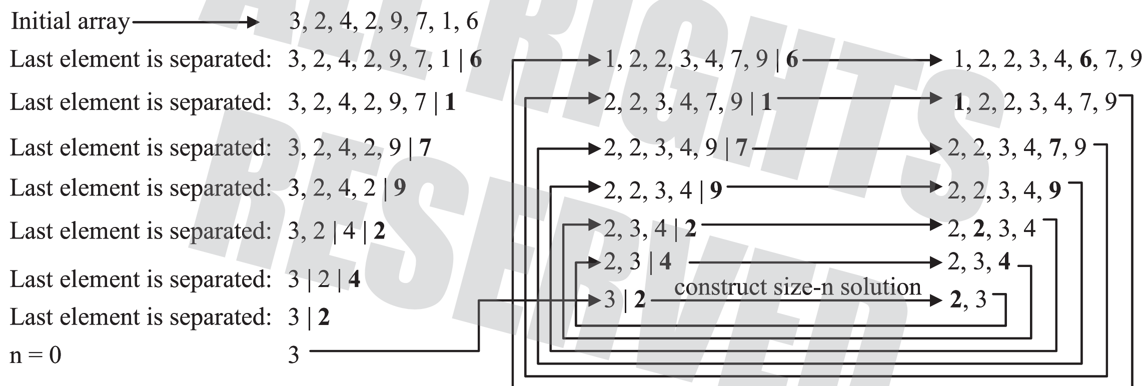


Figure 2.23. Three different repetition structures.

2.8.5 Merge sort algorithm

For some problems, it is possible to reduce the size by more than 1, resulting in a more efficient solution. For example, in step 3 of the above sorting example, we could select `m = ⌊n/2⌋` (floor of `n/2`). In other words, we divide the size-n problem into two approximately equal-sized problems by dividing the array `A` into two half-sized arrays `B1` and `B2`. Then we call

```
sorting(B1, ⌊n/2⌋);   // floor of n/2
sorting(B2, ⌈n/2⌉);   // ceiling of n/2
```

respectively and have both `B1` and `B2` sorted. Then we merge `B1` and `B2` into an array `B` by comparing the elements of the two subarrays sequentially. This sorting algorithm is called **merge sort**, which is one of the

KU ALL RIGHTS RESERVED

most efficient sorting algorithms with a complexity of O($n$log$n$). The pseudo code in Figure 2.24 shows the sorting process through an example, illustrating how each subarray is split, sorted, and merged.

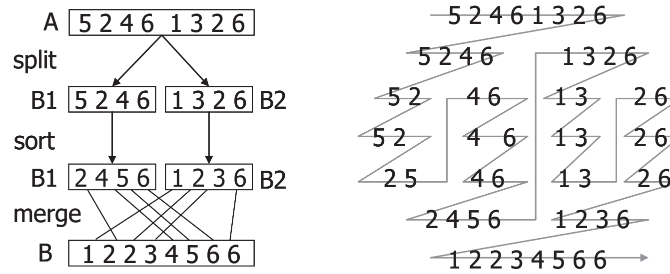


Figure 2.24. Merge sort and its sorting process.

The pseudo code below shows the recursive algorithm that implements merge sort. In the algorithm, the floor(x) function rounds x downward, returning the largest integer value that is not greater than x.

```
mergesort (A, L,R) {
  if R > L then
      { M = floor((R+L)/2);  // rounds down (R+L)/2 to integer
      mergesort (A, L, M);
      mergesort (A, M+1,R);
      merge(A, L, M, R); }
  else
      return A;
}
merge (A, L, M, R) {
  for i = M down to L do B[i] = A[i];
  for j = M+1 to R do  B[R+M+1-j] = A[j];
  i = L;
  j = R;
  for k=L to R do {
     if B[i] < B[j] then
      { A[k]=B[i]; i = i+1; }
     else
      { A[k]=B[j]; j = j-1; }
  }
}
```

L M R

L M M+1 R

KH ALL RIGHTS RESERVED

Please study the pseudo code above and identify the code for defining:

1. size-n problem
2. stopping condition and return value
3. size-m problems
4. the size-n solution from the size-m solutions

2.8.6 Quick sort algorithm

Merge sort has the best complexity O(*n*log*n*) in all the comparison-based sorting algorithms. The big O notation is defined based on the worst case execution time. However, merge sort is not the fast algorithm in terms of the average execution time, which is calculated based on the mean value of all possible input combinations of the input array. Quick sort, on the other hand, the complexity O(n^2). However, the average execution time beats merge sort.

The idea of quick sort is to pick a value (any value) in the array as the pivot value to divide the array into two subarrays, one with all its elements less than the pivot value, and the other with all its value greater than or equal to the pivot value. The pseudo code below shows the recursive algorithm that implements quick sort. In Chapter 5, we will give a full implementation of quick sort in Prolog.

```
void Quicksort (A, L, R) {
  if R = L then return; else {
       k = split(A, L, R);
       Quicksort (A, L, k-1);
       Quicksort (A, k+1, R);
  }
}
int split(A, L, R) {
       int pivot = A[R]; i = L; j = R;
       while i < j do {
             while (A[i] < pivot) do i = i+1;
             while ((j > i) && (A[j] >= pivot)) do j = j - 1;
             if  (i < j) then
                  swap(A[i], A[j]);  // swap the values
       }
       swap(A[i], A[R]);
       return i;
}
```

KH ALL RIGHTS RESERVED

2.8.7 Tree operations

Graphs and trees are widely used data structures. A graph is a mathematical model and a data structure that is widely for representing related objects. A graph consists of a set of nodes and a set of edges between the nodes. A graph is a **directed graph** if a direction is defined for each edge, and a graph is an **undirected graph** if there is no direction defined for any edge. Assuming that the direction of the edge is the driving direction of streets, a directed graph allows the flow following the edge directions. An undirected graph allows the flow in both directions of the edge. A **path** from Node `N1` to Node `N2` is a sequence of edges connecting `N1` to `N2`, for example, `(`**`N1`**`, X1), (X1, X2), (X2, X3), …, (Xk, `**`N2`**`)`.

A directed graph is a tree, if it does not contain a loop and there is no more than one path between any two nodes. A tree is a rooted tree if it has a unique node that does not have an incoming edge. This node is the root of the tree. A node that does not have outgoing edges is a leaf. The height or depths of a tree is the number edges from the root to the deepest (farthest) leaf.

A tree is a binary tree, if any of its nodes can have at most two child (next) nodes. In a full binary tree, each node has either no child or two children. A balanced binary tree is a tree in which the heights (depths) of

the two subtrees of every node never differ by more than 1. Its height is $O(\log_2 n)$. Figure 2.25 shows a rooted tree, a binary tree, a full binary tree, and a balanced binary tree.

A binary search tree stores data in such a way that it keeps keys (used for indexing data) in sorted order, so that data search will be much faster.

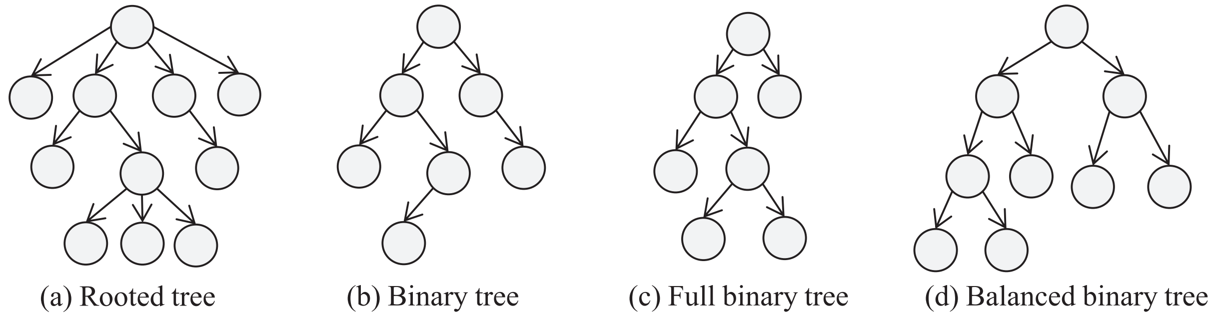


Figure 2.25. Binary trees.

When inserting data into a binary search tree, the simplest algorithm is

1) If the tree is empty, insert the first number as the root;
2) If the tree is not empty:
    a. If the incoming number is small than the key of the current node, insert the number to the left subtree using recursion;
    b. If the incoming number is greater than or equal to the key of the current node, insert the number to the right subtree using recursion.

Using this simple insertion algorithm, the binary search will not be balanced. In the worst-case scenario, when the input numbers are sorted, the tree becomes a linked list. Figure 2.26 shows a not balanced binary search tree and a balanced binary search tree.

KH ALL RIGHTS RESERVED

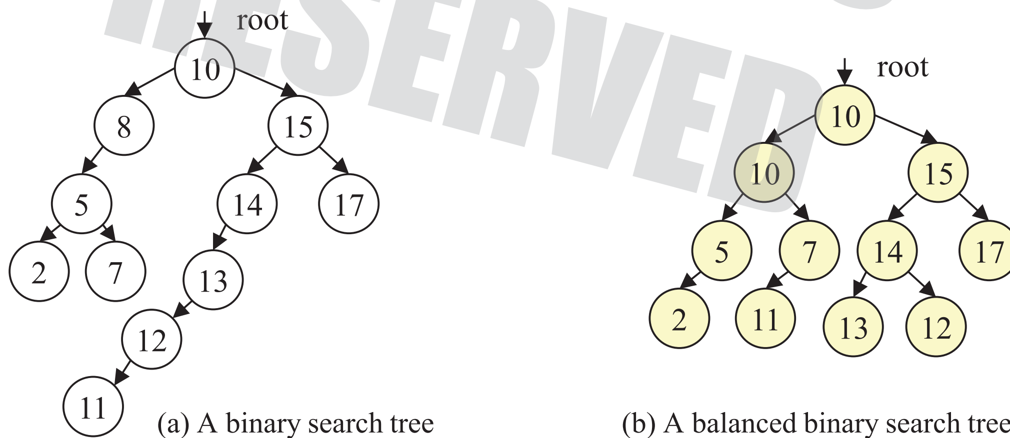


Figure 2.26. A binary search tree and a balanced binary search tree.

To maintain the binary search tree balanced when inserting, a much more complex algorithm must be applied. Red-black tree is a data structure that attempts to have the tree balanced during insertion, with a topic of algorithm class.

As data are searched much often than data are inserted, it is critical to make search faster than insertion. The complexity of search algorithms on different data structures is listed as follows:

- The complexity is O(n) for linear search of data stored in arrays or linked lists.
- The complexity is O(n) for binary search is $O(log_2 n)$, if the binary tree is balanced.
- The complexity is O(n) for binary search is $O(log_2 n)$, if the binary tree is binary.

The following code shows a simple implementation of search and insertion of binary search tree, where the common functions `main` and `branching` are similar to those discussed in the linked list section.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h> // used by malloc
#include <time.h>
struct treeNode {
   int data;
   struct treeNode *left, *right; // pointers to left and right
} *root = 0;         //root is a global pointer to the root entry
void branching(char);             // function forward declaration
void insertion();
struct treeNode *search(struct treeNode *, int);
void traverse(struct treeNode *);
main( ) {   // print a menu for selection
   char ch = 'i';
   srand( (unsigned)time( 0 ) );  // Use current time as seed
   while (ch != 'q') {
       printf("Enter your selection\n");
       printf("      i: insert a new entry\n");
       printf("      s: search an entry\n");
       printf("      t: traverse the tree and print\n");
       printf("      q: quit \n");
       fflush(stdin);    // flush the input buffer
       ch = tolower(getchar());
       branching(ch);
   }
}
void branching(char c) {      // branch to different tasks
   int key;
   switch(c)  {
   case 'i':
       insertion();           // Not passing root, but use it as global
       break;
   case 's':
       printf("Enter the key to search\n");
       scanf("%d", &key);
       search(root, key);     // root call-by-value
```

KH ALL RIGHTS RESERVED

```
        break;
    case 't':
        traverse(root);         // print all data
        break;
    default:
        printf("Invalid input\n");
    }
}
struct treeNode * search(struct treeNode *top, int key) {
    struct treeNode *p = top;
    if (key == p->data)
        printf("data = %d\n", p->data);
    if (key <= p->data) {
        if (p->left == 0) return p;
        else search(p->left, key);
    }
    else {
        if (p->right == 0) return p;
        else search(p->right, key);
    }
}
```

Deletion is more complex than insertion and search. Three cases need to be considered, as shown in Figure 2.27, where the shaded node is the node to be deleted.

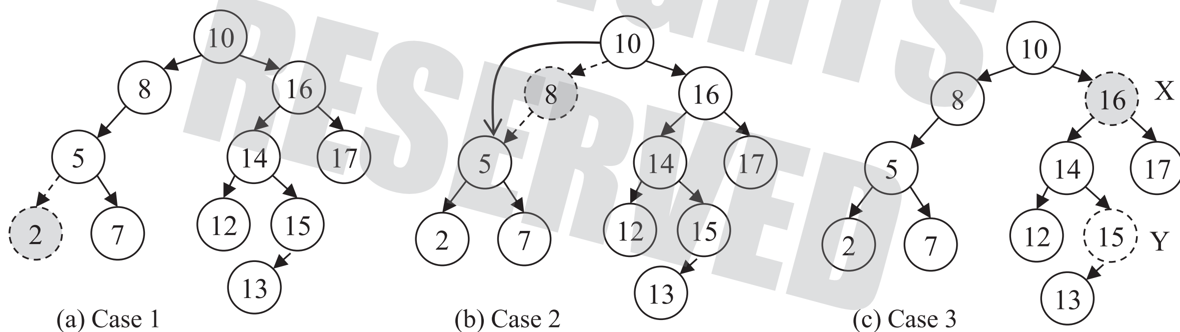


Figure 2.27. Three cases in deletion of a node.

Case 1: If the node to be deleted is a leaf, the node can be simply deleted.

Case 2: If the node to be deleted has only one child node, we can link the parent node to the child node.

Case 3: If the node X to be deleted has two child nodes, we can use a search function to find Y, the successor of X, which is the large node that is smaller than X. We use Y to replace X. Then, the problem becomes deleting Y from the tree. We repeat the same process with three possible cases for deleting Y. In the example in Figure 2.27, Y falls into case 2, and we can use node 14 to 13.

KH ALL RIGHTS RESERVED

2.8.8 Gray code generation

The Gray code, named after Frank Gray, also known as reflected binary code, is a binary coding system where two successive values differ in only one digit. The Gray code was originally designed for preventing spurious output from electromechanical switches. Today, the Gray code is widely used in facilitating error correction in digital communications such as digital terrestrial television and some cable TV systems.

The main feature of the Gray code is that an n-bit Gray code can be constructed from (n-1)-bit code in the process shown in Figure 2.28.

Following the fantastic-four abstract approach, we can formulate the problem and its solution in the following four steps:

1. Formulate the size-n problem.

```
char *gcode(int n);        // will return the array of n-bit gcode
```

2. Find the stopping condition and the corresponding return value.

```
If n = 1, return array {'0', '1'};
```

3. Formulate the size-(n-1) problem, assuming the gcode is found for the size-(n-1) problems.

```
gcode(n-1) will return the (n-1)-bit gcode
```

4. Construct the solution of size-n problem.

```
part1 = gcode(n-1);
part2 = reverse(part1)
left-append '0' to each item in part1
left-append '1' to each item in part2
return: part1 and part2
```

KH ALL RIGHTS RESERVED

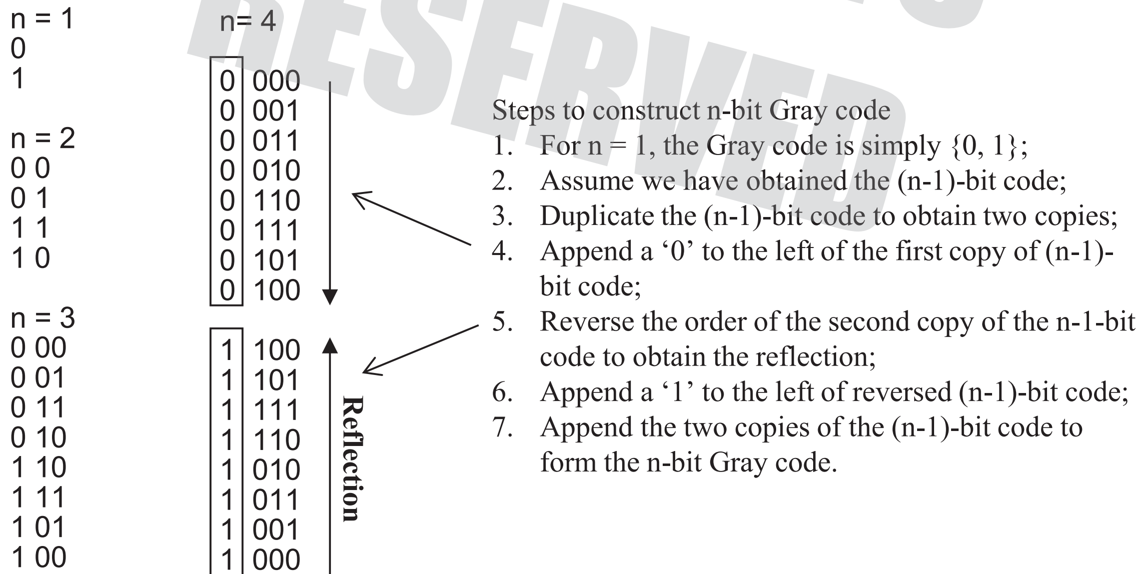


Figure 2.28. Generating n-bit Gray code from (n-1)-bit Gray code.

The complete Gray code generation function and a sample main program are given as follows:

```c
#include <stdio.h>
#include <stdlib.h>  // malloc
#include <math.h>    // double pow(double, double)
#define columns 8    // The example limits the size to 7 columns
#define rows 256     // 256 = pow(2, 8)
char **gcode(int n);
void main() {
   char **g; int i, n, p;
   printf("please enter an integer n for n-bit Gray code\n");
   scanf("%d", &n);   // Note: 0 < n < columns
   g = gcode(n); // Call recursive gcode function
   p = (int) pow(2, n);
   for (i=0; i<p; i++)
        printf("%s\n", g[i]);  // Print each element in array
}
char **gcode(int n) {
   int i, j, p, q;
   char **sizem, **sizen;      // pointers to 2-D arrays
   p = (int) pow(2, n);   // The length of size-n-code
   q = (int) p/2;      // create an array of pointers
   sizen = (char **) malloc(sizeof(char[rows]));
   for (i =0; i<p; i++)
        sizen[i] = (char *) malloc(sizeof(char[columns]));
   if (n<=1) {              // stopping condition
        sizen[0][0] = '0';
        sizen[0][1] = '\0';     // add terminator
        sizen[1][0] = '1';
        sizen[1][1] = '\0';     // add terminator
        return sizen;
   }
   else {
        sizem = gcode(n-1);
        for (i = 0; i < q; i++) {
              sizen[i][0] = '0';
              for (j = 1; j<=n; j++)
                   sizen[i][j] = sizem[i][j-1];
        }
        for (i = 0; i < q; i++) {
              sizen[q+i][0] = '1';
              for (j = 1; j<=n; j++)
                   sizen[q+i][j] = sizem[q-i-1][j-1];
        }
        for (i = 0; i < q; i++)
              free(sizem[i]);  // free each row
```

```
C:\YinongAll\Yin...
please enter an integer n
for n-bit Gray code
4
0000
0001
0011
0010
0110
0111
0101
0100
1100
1101
1111
1110
1010
1011
1001
1000
```

KH ALL RIGHTS RESERVED

```
   free(sizem);                // free the index
        return sizen;
   }
}
```

2.9 Modular design

Functions bring a level of abstraction into our programs. The abstraction makes our program easier to understand and to manage. However, the programming task can still become too large to understand. We need to introduce another level of abstraction, that is, modular design. Other advantages of modular design include:

- **Sharing**: We can group some frequently used functions and data into a module for being shared with other programmers (e.g., library functions).
- **Separate compilation**: This is a maintenance issue. If we find a programming error or we need to make functional modifications in a part of the program, we do not have to recompile the entire program.
- **Expandability**: We can easily add new modules into the system.

So far, we have been focusing on designing a program to solve relatively small problems. This is called **programming-in-the-small**. We need the skill of programming-in-the-small before we can do **programming-in-the-large**, which combines programming modules into a large program.

To design a module in a large system, we need to separate the specification from the implementation. The **specification** part tells what the module does and gives an external view of the module, while the **implementation** part gives code that implements the specification. Variable and function names given in the specification part are available to users inside the module as well as outside the module.

All programming languages provide mechanisms to support modular design. In C, specifications of programs are stored in .h files, while implementations are stored in .c files. In order to use functions defined in another module named, say, `modulename.c`, we need to use

```
#include "modulename.h"  // user-defined header files are quoted by "…"
```

Consider the traffic light example in Section 2.4. Since the `sleep` function may be used by other programs, we want to put this function and possibly other frequently used functions into a module, say, called `mylib.c`, which contains the following code:

```
// file name: mylib.c
#include<time.h>
const float pi = 3.14159265;
// Sleep for a specified number of seconds.
void sleep(int wait) {
   clock_t goal;               // clock_t defined in <time.h>
   goal = wait * CLOCKS_PER_SEC + clock();
   while( goal > clock() )
        ;
}
// This function computes the volume of a cylinder:
double cylinder (int h, int r) {  //h: height, r: radius
```

KH ALL RIGHTS RESERVED

```
  const double pi = 3.14159265;
  return pi*r*r*h;
}
```

Notice that a module does not need to have a `main()` function.

Then, we can put the headers of all functions, the type definitions, as well as the global variables to be shared among different modules, in the header file called `mylib.h`. In this example, we have only two functions and one type definition. Thus, the header file should look like:

```
typedef enum {red, amber, green} traffic_light;
void sleep(int wait);
double cylinder (int h, int r);
```

In the main program of the traffic light example, we do not need the function `sleep`, or the type definition. The code of the main function is as follows:

```
#include <stdio.h>  // system library uses angle brackets to include
#include "mylib.h"  // user library uses quotes to include
main() {
traffic_light x = red;
printf("Red:\tStop!\n");
while (1)
  switch (x) {
      case amber:
            sleep(1);        //sleep 1 second
            x = red;
            printf("Red:\tStop!\n"); break;
      case red:
            sleep(6);        //sleep 6 second
            x = green;
            printf("Green:\tGo>>>\n"); break;
      case green:
            sleep(12);       //sleep 12 second
            x = amber;
            printf("Amber:\tBrake...\n");
  }
}
```

In Visual Studio programming environment, all modules (.c files) should be placed in the folder “Source Files” and all the user-defined header files should be placed in the folder “Header Files,” as shown in Figure 2.29.

KH ALL RIGHTS RESERVED

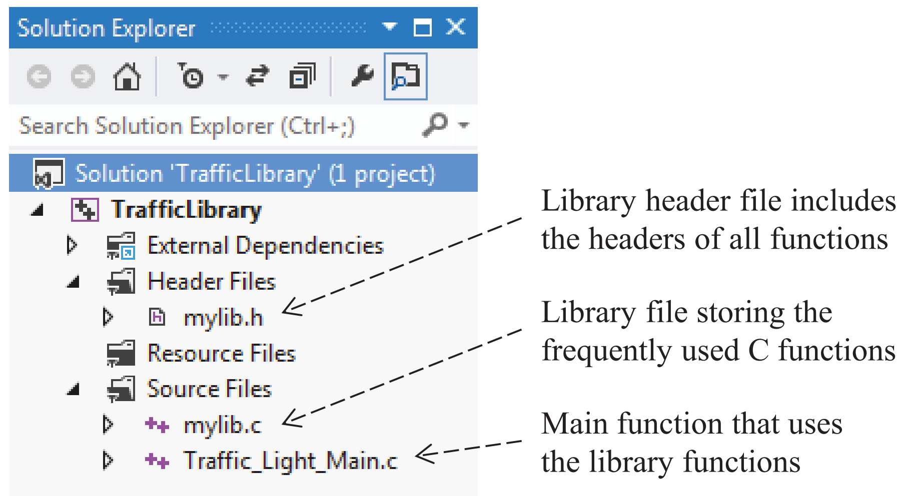


Figure 2.29. Organizing the modules and header files.

In object-oriented computing, better modularity is the main focus. We will discuss program design with multiple classes and multiple modules in more detail in Chapter 3.

2.10 Case Study: Putting All Together

In this section, we give an example that applies many of the data structures and programming techniques learned in this chapter, including array, string, enumeration type, pointer, pointer to pointer, linked list, global variable versus local variable, call-by-value and call-by-address parameter-passing mechanisms, memory management and garbage collection, and recursion. The memory management and garbage collection will be further discussed in the C++ chapter in more detail, where memory leak detection and detection tool will be introduced.

The first part of the example includes the declaration, forward declaration, the main function, and the branching function.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#pragma warning(disable: 4996) // comment out if not in Visual Studio
typedef enum { diploma = 0, bachelor, master, doctor } education;
// A struct for nodes of the linked list.
struct container {
   struct person *plink; // points to a struct person.
   struct container *next;
};
// A struct to hold attributes of a person
struct person {
   char name[32];
   char email[32];
   int phone;
```

KH ALL RIGHTS RESERVED

```
  education degree;
};
void branching(char c);
char* get_name();
void print_list(struct container* root);
int insertion(struct container** ptrToHead); // note: pointer to pointer
struct container* search(struct container* root, char* sname);
void deleteOne(struct container** ptrToHead);
void deleteAll(struct container** ptrToHead);
void print_all(struct container* root);
/*****************************************************************/
int main(){
  // Declare head as a local variable of main function
  struct container* head = NULL;
  char ch = 'i';
  do {          // Print a menu for selection
      printf("Enter your selection\n");
      printf("\ti: insert a new entry\n");
      printf("\td: delete one entry\n");
      printf("\tr: delete all entries\n");
      printf("\ts: search an entry\n");
      printf("\tp: print all entries\n");
      printf("\tq: quit \n");
      fflush(stdin); // Flush the input buffer. Read section 2.6.3
      ch = tolower(getchar());   // Convert uppercase char to lowercase.
      branching(ch, &head);
      printf("\n");
  } while (ch != 113);          // 113 is 'q' in ASCII
  return 0;
};
/*****************************************************************/
// Branch to different tasks: insert a person, search for a person,
// delete a person, and print all added persons.
void branching(char c, struct container** ptrToHead){
  char *p;
  switch (c) {
  case 'i':
      insertion(ptrToHead);break;
  case 's':
      p = get_name();
      search(*ptrToHead, p);break;
  case 'd':
      deleteOne(ptrToHead);break;
  case 'r':
```

KH ALL RIGHTS RESERVED

```
        deleteAll(ptrToHead);break;
    case 'p':
        print_all(*ptrToHead);break;
    case 'q':
        deleteAll(ptrToHead); // free all memory when quit
        break;
    default:
        printf("Invalid input\n");
    }
};
```

The relationship between the container struct and the person struct is illustrated in Figure 2.30. Notice that the head pointer is declared as a local variable in the main function, which is not visible in the other functions, and thus, we need to use parameter passing to access and to modify the head pointer. To read the head pointer, we can use call-by-value parameter passing, which are used in the functions search and printAll. In the functions deleteOne and deleteAll, we need to modify the head pointer, and we need to pass the address of head pointer into these functions, which is thus a pointer to a pointer.

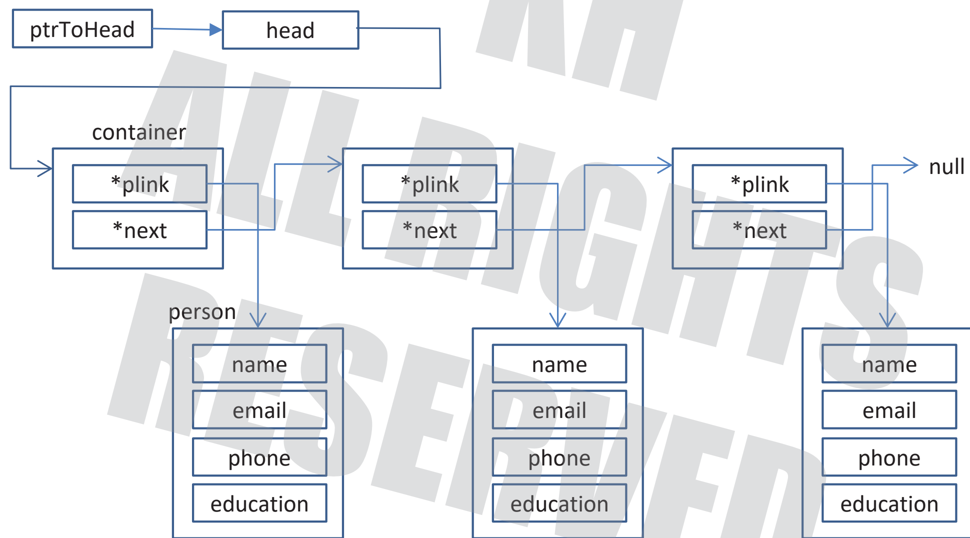


Figure 2.30. A linked list of containers, with pointers to a person node and to the next container.

In the following, we provide the remaining functions listed in the forward declaration part and discuss their implementation.

```
/****************************************************************/
// Delete the first person node in the linked list.
void deleteOne(struct container** ptrToHead) {
    int i = 0;
    struct container *toDelete = NULL;
    if (*ptrToHead == NULL) {
        printf("\nThe list is empty. Nothing was deleted.\n");
    }
    else if ((*ptrToHead)->next == NULL) {
        free((*ptrToHead)->plink);
```

```
        free(*ptrToHead);
        *ptrToHead = NULL;
    }
    else {
        toDelete = *ptrToHead;
        *ptrToHead = (*ptrToHead)->next;
        free(toDelete->plink);
        free(toDelete);
        toDelete = NULL;
    }
    printf("\nA container node was deleted.\n");
};
/****************************************************************/
// Recursively delete the entire list given the head of a linked list.
void deleteAll(struct container** ptrToHead) {
    struct container* pnext;
    if (*ptrToHead == NULL)
        return;
    else {
        deleteOne(ptrToHead);
        deleteAll(ptrToHead);
    }
};
// Read the input from the user.
char * get_name() {
    char *p = (char *) malloc(32); // Use dynamic memory which does not go
out of scope
    printf("Please enter a name for the search: ");
    scanf("%s", p);
    return p;
};
/****************************************************************/
// Inserts the person to the sorted place. Note: A < a, and A will be
ordered first.
int insertion(struct container** ptrToHead) {
    int i = 0;
    struct container* newNode = NULL, *iterator = NULL, *follower = NULL;
    struct person* newPerson = NULL;
    newNode = (struct container*) malloc(sizeof(struct container));
    // Case 1: The program is out of memory.
    if (newNode == NULL) {
        printf("Fatal Error: Out of Memory. Exiting now.");
        return 0;
    }
```

KH ALL RIGHTS RESERVED

```
  // Case 2: The structure still has unfilled slots.
  else {
      newPerson = (struct person*) malloc(sizeof(struct person));
      if (newPerson == NULL) {
           printf("Fatal Error: Out of Memory. Exiting now.");
           return 0;
      }
      else {
           printf("Enter the name:\n");
           scanf("%s", newPerson->name);
           printf("Enter the phone number:\n");
           scanf("%d", &newPerson->phone, sizeof(newPerson->phone));
           printf("Enter the e-mail:.\n");
           scanf("%s", newPerson->email);
           do {
                printf("Enter the degree: select 0 for diploma, select 1
for bachelor, select 2 for master, or select 3 for doctor:\n");
                scanf("%d", &newPerson->degree);
                if (newPerson->degree< diploma || newPerson->degree >
doctor) {
                     printf("Please enter a value from 0 to 3.\n");
                }
           } while (newPerson->degree < diploma || newPerson->degree >
doctor);
           newNode->plink = newPerson;
           if (*ptrToHead == NULL) {
                *ptrToHead = newNode;
                (*ptrToHead)->next = NULL;
                return 0;
           }
           else {
                if (strcmp(newPerson->name,(*ptrToHead)->plink->name)<0){
                     newNode->next = *ptrToHead;
                     *ptrToHead = newNode;
                     return 0;
                }
                iterator = *ptrToHead;
                follower = iterator;
                while (iterator != NULL) {
                     if (strcmp(newPerson->name,iterator->plink->name)<0){
                         newNode->next = iterator;
                         follower->next = newNode;
                         return 0;
                     }
```

KH ALL RIGHTS RESERVED

```
                    follower = iterator;
                    iterator = iterator->next;
                }
                follower->next = newNode;
                newNode->next = NULL;
                return 0;
            }
        }
    }
    return 0;
};
/**************************************************************/
// Print the name, e-mail, phone, and education level of each person.
// It calls the helper printFirst to recursively print the list
void print_all(struct container* root) {
    struct container* iterator = root;
    //Case 1: The structure is empty
    if (iterator == NULL) {
        printf("\nNo entries found.\n");
        return;
    }
    // Case 2: The structure has at least one item in it
    else{
        printFirst(root);
        return;
    }
};
void printFirst(struct container* root) {
    if (root != NULL){
        printf("\n\nname = %s\n", root->plink->name);
        printf("email = %s\n", root->plink->email);
        printf("phone = %d\n", root->plink->phone);
        switch (root->plink->degree) {
        case diploma:
            printf("degree = diploma\n");
            break;
        case bachelor:
            printf("degree = bachelor\n");
            break;
        case master:
            printf("degree = master\n");
            break;
        case doctor:
            printf("degree = doctor\n");
```

KH ALL RIGHTS RESERVED

```
            break;
        default:
            printf("System Error: degree information corruption.\n");
            break;
        }
        printFirst(root->next);
    }
};
/*****************************************************************/
//Find a person by comparing names given the head of the linked list.
struct container* search(struct container* root, char* sname) {
    struct container* iterator = root;
    while (iterator != NULL) {
        if (strcmp(sname, iterator->plink->name) == 0) {
            printf("\n\nname = %s\n", iterator->plink->name);
            printf("email = %s\n", iterator->plink->email);
            printf("phone = %d\n", iterator->plink->phone);
            switch (iterator->plink->degree) {
            case diploma:
                printf("degree = diploma\n");
                break;
            case bachelor:
                printf("degree = bachelor\n");
                break;
            case master:
                printf("degree = master\n");
                break;
            case doctor:
                printf("degree = doctor\n");
                break;
            default:
                printf("System Error: degree information corruption.\n");
                break;
            }
            free(sname); // garbage collection
            return iterator;
        }
        iterator = iterator->next;
    }
    printf("The name does not exist.\n");
    free(sname); // garbage collection
    return iterator;
};
```

KH ALL RIGHTS RESERVED

2.11 Summary

In this chapter, we started from basic issues in writing imperative C/C++ programs and went through important and advanced topics in the languages. The focus is on the topics that are significantly different from Java. The major topics we discussed are:

- Getting started with writing simple C/C++ programs;
- Control structures in C/C++ using syntax graphs;
- Relationships among memory locations, memory addresses, variable names, variable addresses, and the value stored in a variable;
- Pointers and pointer variables, referencing, and dereferencing;
- Array-based and pointer-based string operations;
- Three different ways of introducing constants: macro, const, and enumeration types;
- Structure types and compound data types;
- File type and file operations;
- Three major parameter-passing mechanisms: call-by-value, call by address, and call-by-alias;
- Recursive structures; and
- A brief introduction to modular design. More modular design will be discussed in the C++ chapter.

Table 2.5 summarizes the features supported by C and C++, as well as by Java. As can be seen from the table, C and C++ allow more flexibility, whereas Java is more restricted.

Feature	C and C++	Java
Macro	YES	NO
Inlining	YES	YES
Global variables	YES	NO
Static variables	YES	YES
Pointer	YES	NO
Value semantics for all types	YES	Primitive types
Reference semantics for all types	YES	Reference types
String type	char array	YES
Union type	YES	NO
Parameter passing: call-by-value	YES	Primitive types
Parameter passing: call-by-alias	C++ only	NO
Parameter passing: call-by-address	YES	Reference types
Recursive call and application of the fantastic-four abstract approach	YES	YES

Table 2.5. Feature comparison between C/C++ and Java.

KH ALL RIGHTS RESERVED

KH
ALL RIGHTS
RESERVED

2.12 Homework, programming exercises, and projects

1. Multiple Choice. Choose only one answer for each question. Choose the best answer if more than one answer is acceptable.

1.1 Forward declaration in modern programming practice

❑ provides a level of abstraction. ❑ is never necessary.

❑ is not required if `<iostream>` is included. ❑ is useless.

1.2 C language does not have a Boolean type because

❑ C is not designed to handle logic operations. ❑ C uses strong type checking.

❑ Boolean values can be represented as integers. ❑ C++ has already defined a Boolean type.

1.3 Two functions are said to be mutually recursive if

❑ one function is defined within the other function.

❑ they call each other.

❑ each function calls itself.

❑ they are independent of each other.

1.4 Assume that a string is declared as `char str[] = "alpha"`, what is the return value of `sizeof (str)`?

❑ 1 ❑ 5 ❑ 6 ❑ 7 ❑ 40

1.5 Assume that two pointers are declared as: `char *str1 = "alpha", *str2;`
Which assignment statement below will lead to a semantic error?

❑ `str2 = str1;` ❑ `str2 = 0;` ❑ `str1 = str1+1;` ❑ `*str2 = "Hi";`

1.6 Which of the following declarations will cause a compilation error?

❑ `char s[5];` ❑ `char s[3] = "hello";`

❑ `char s[];` ❑ `char s[] = {'s', 't', 'r'};`

1.7 Given a declaration: int i = 25, *j = &i, **k = &j;
which of the following operations will change the value of variable i?

❑ `j++;` ❑ `k++;` ❑ `(*k)++;` ❑ `(**k)++;`

1.8 Given a declaration: int i = 25, *j = &i, **k = &j;
which of the following operations will cause a compilation error?

❑ `i++;` ❑ `(&i)++;` ❑ `(*j)++;` ❑ `(**k)++;`

1.9 What is the maximum number of padding bytes that a compiler can add to a structure?

❑ 1 ❑ 2 ❑ 3 ❑ more than 3

1.10 The enumeration type of values are stored in the memory as

❑ bool ❑ double ❑ int ❑ string

KH ALL RIGHTS RESERVED

1.11 If we want to store a linked list of structures, with each structure containing different types of data, into a disk file, what file type should we choose?

❑ array file ❑ binary file ❑ text file ❑ structure file

1.12 The reason that we need to call fflush() or cin.ignore() is because the previous

❑ output leaves a character in the file buffer. ❑ output fails to complete its operation.

❑ input leaves a character in the file buffer. ❑ input fails to complete its operation.

1.13 Assume the following structure is defined in a 32-bit programming environment.

```
struct myNode {
    char     name[30];
    char     location[32];
    struct   myNode* next;
} x, *y;
```

what is the size of x?

❑ 4 bytes ❑ 66 bytes ❑ 68 bytes ❑ 72 bytes

what is the size of x?

❑ 4 bytes ❑ 66 bytes ❑ 68 bytes ❑ 72 bytes

1.14 What parameter-passing mechanism cannot change the variable values in the caller?

❑ call-by-value ❑ call-by-alias ❑ call-by-address ❑ None of them

1.15 What parameter-passing mechanism requires the actual parameter to be a variable?

❑ call-by-value ❑ call-by-alias ❑ call-by-address ❑ None of them

1.16 Given the forward declaration: void foo(char c, int &n); what parameter passing mechanisms are used? Select all that apply.

❑ call-by-value ❑ call-by-alias ❑ call-by-address ❑ None of them

1.17 What type of recursive function is structurally equivalent to a while-loop?

❑ head-recursion ❑ middle-recursion ❑ tail-recursion ❑ mutual recursion

The Ackermann function is defined recursively for two nonnegative integers k and n as follows. Answer the following three questions based on the function and the fantastic-four abstract approach.

$$A(s, t) = \begin{cases} t+1, & if\ s = 0 \\ A(s-1, 1), & if\ s > 0\ and\ t = 0 \\ A(s-1, A(s, t-1)), & if\ s > 0\ and\ t > 0 \end{cases}$$

1.18 What is the size-n problem?

❑ (s, t) ❑ A(s, t) ❑ A(n) ❑ n

1.19 What is the stopping condition and return value at the stopping condition?

❑ s = 0 and t+1 ❑ s = 0 and t = 1 ❑ s > 0 and t = 0 ❑ s > 0 and t > 0

KH ALL RIGHTS RESERVED

1.20 What is the size-m problem that can be used for calculating the size n problem? Select all that apply.

❑ A(s, t) ❑ A(s-1, 1) ❑ A(s, t-1) ❑ A(s-1, A(s, t-1))

1.21 The data stored in a binary search tree is sorted, if the tree is traversed in

❑ preorder ❑ postorder ❑ inorder ❑ in any order

1.22 Consider an array, a linked list, doubly linked list, and a binary search tree. Which data structure requires fewest comparisons in average to search an element stored in the data structure?

❑ binary search tree ❑ array ❑ doubly linked list ❑ linked list

2. What is a byte and what is a word in memory? What is the name of a variable? What is the address of a memory location? What is the content of a memory location?

3. What is the difference between a memory location and a register? How do we access a memory location and a register?

4. A variable has several aspects (name, address, value, location), and different aspects are used in different places.

4.1 If a variable is used on the left-hand side of an assignment statement, which aspect is actually used?

4.2 If a variable is used on the right-hand side of an assignment statement, which aspect is actually used?

4.3 If we apply the address-of operator "`&`" to the variable (i.e., `&v`), which aspect is returned?

5. Given a piece of C code

```
1   #include <stdio.h>
2   void main() {
3       char str[] = "hello", *p;
4       p = str;
5       while (*p != '\0')
6           (*(p++))++;
7       printf("str = %s, p = %s\n", str, p);
8   }
```

KH ALL RIGHTS RESERVED

5.1 What is the exact output of the printf statement?

5.2 At line 3, if we replace `char str[] = "hello"` by `char *str = "hello"`, it will cause

❑ compilation error. ❑ runtime error. ❑ no error at all. ❑ incorrect output.

5.3 At line 3, if we replace `*p;` with `char *p = str;` it will cause

❑ compilation error. ❑ runtime error. ❑ no error at all. ❑ incorrect output.

5.4 In lines 5 and 6, the string is accessed using a pointer and pointer operations. Rewrite the program from line 3 to line 6 so that only array operations are used to access the string.

6. What are the three different methods of defining constants in C/C++? What are the differences of the constants defined in these methods?

6.1 Can a constant defined by `const` ever be modified? If yes, how and why? If no, why?

6.2 Can a constant defined by `#define` ever be modified? If yes, how and why? If no, why?

6.3 What are the advantages of defining an enumeration type instead of using an integer type directly?

7. What is the difference between a structure type and a union type? In what circumstances are union types useful?

8. Parameter passing

8.1 What is a formal parameter and what is an actual parameter?

8.2 What is the difference between call-by-value and call-by-alias?

8.3 Where do you need to use call-by-value and where do you need to use call-by-alias?

8.4 How do you use call-by-value and call-by-alias?

9. Structure type

9.1 How do you define a structure type? How do you declare a variable of a structure type? How do you declare a pointer to a structure type variable?

9.2 How do you obtain memory statically for a structure type variable? How do you create dynamic memory and link it to a pointer?

9.3 How do you use the name of a structure and a pointer to a structure to access the fields in the structure?

10. Programming exercise. This question gives you practice in using declarations, forward declarations and scopes of functions and variables, and type checking in C and C++.

Given the C program below, answer the following questions.

```
// This program shows function and variable declarations and their
scopes.
#include <stdio.h>
int keven = 0, kodd = 0;
long evennumber(short);
long oddnumber(short);
int even(int);
int evennumber(int a) {         // genuine declaration
    if (a == 2) {
         printf("keven = %d, kodd = %d\n", keven, kodd);
         return keven;
    }
    else {
         a = (int)a/2;
         if (even(a)) {    // Is an even?
              keven++;
              return evennumber(a);
         }
```

KH ALL RIGHTS RESERVED

```
        else {
            kodd++;
            return oddnumber(a);
        }
    }
    // return a;
}
int oddnumber(int b) {     // genuine declaration
    if (b == 1) {
        printf("keven = %d, kodd = %d\n", keven, kodd);
        return kodd;
    }
    else {
        b = 3*b+1;
        if (!even(b)) {  // Is b odd?
            kodd++;
            return oddnumber(b);
        }
        else {
            keven++;
            return evennumber(b);
        }
    }
    // return b;
}
int even(int x) {      // % is modulo operator.
    return ( (x%2 == 0) ? 1 : 0);
}
void main() {
    register short r = 0;      // a register type variable is faster,
    int i = r;                 // it is often used for loop variable
    float f;
    for (r = 0; r < 3; r++) {
        printf("Please enter an integer number that is >= 2\n");
        scanf("%d", &i);
        if (even(i))
            f = evennumber(i);
        else
            f = oddnumber(i);
    }
}
```

KH ALL RIGHTS RESERVED

10.1 Save the file as `declaration.c`. (consider the program to be a C program). Choose the commands under the menu “`Build`”:

```
Compile declaration.c
Build declaration.exe
Execute declaration.exe
```

What errors or warning messages are displayed?

10.2 Save the file as `declaration.cpp`. Repeat question 10.1.

10.3 Analyze the type requirement of functions and variables in the given program. Make minimum changes to `declaration.cpp` to remove all compilation errors and warnings.

10.4 Explain global variables and local variables. List the global variables and local variables in the program. The parameters of a function are local variables too.

10.5 Can we swap the order of the two variable declarations: "`register short r = 0;`" and "`int i = r;`"? Explain your answer according to C/C++'s scope rule.

10.6 Explain the forward declaration. If the forward declarations in the program were removed, what would happen?

10.7 Explain type casting and type coercion. List all type castings and type coercions used in the program.

10.8 According to the analysis above and the definition of strong type checking, are C and C++ strongly typed? Which language's typing system is stronger, C or C++?

10.9 Program correctness/reliability issue. A correct program must terminate for all valid inputs. The given program has been tested by many people. It has always terminated for the inputs used. However, nobody so far can prove that this program can terminate for any integer input. Thus, this program is often used as an example of improperly designed loop structure or recursive function. A good programming practice in writing loop or recursive function is to guarantee that the loop variable or the size-related-parameter (they control the number of iterations) is defined on an enumerable set (e.g., integer), has a lower bound (e.g., 0), and decreases strictly (e.g., 9, 6, 5, 3, 2, 1). Add a print-statement in functions `evennumber` and `oddnumber` to print the size-related parameter value and use input values i = 3, 4, and 7, respectively, to test the program. Give the three sequences of values printed by the added print-statements.

10.10 Compare questions 10 with homework question 17 in Chapter 1 and explain why it is difficult to prove that the program can terminate for any integer input.

11. The following program will open and read an existing text file called `file1.txt`, add a number between 1 and 25 to each and every character, and then write the modified text into a new file called `file2.txt`. Read this program carefully and answer the questions following the program.

```
// Filename: encryption0.c
#include <stdio.h>
#include <string.h>

// Read all characters in the file and put them in a string str
void file_read(char *filename, char *str) {
```

KH ALL RIGHTS RESERVED

```
    FILE *p;      // p is declared as a pointer to the FILE type.
    int index=0;
    p=fopen(filename, "r");    // Open the file for "read".
                      // Other options incl. "w" (write) and "a"
(append)
    while(!feof(p))        // while not reaching the end-of-file
character
          *(str+index++)=fgetc(p); //read a character from file and put
it
                         // in str. Then p is increased automatically.
    str[index-1]='\0';            // add the string terminator
    puts(str);                    // print str. You can use printf too.
    fclose(p);                    // close the file
}
void encrypt(int offset, char *str) {
    int i,l;
    l=strlen(str);
    printf("unencrypted str = \n%s\n", str);
    for(i=0;i<l;i++)
          str[i] = str[i]+offset;
    printf("encrypted str = \n%s \nlength = %d\n", str, l);
}
void file_write(char *filename, char *str) {
    int i, l;
    FILE *p;
    p=fopen(filename, "w");       // open the file for "write".
    l = strlen(str);              // string-length
    for(i=0;i<l;i++)
          fputc(*(str+i),p);      // write a character in the file
pointed
                                  // by p. p is increased automatically
    fclose(p);                    // close the file
}
void main(void) {
    char filename[25];
    char string[1024];
    strcpy(filename, "file1.txt");
    file_read(filename, string);
    encrypt(7, string);
    strcpy(filename, "file2.txt");
    file_write(filename, string);
}
```

KH ALL RIGHTS RESERVED

11.1 Enter the following text in a text file named `file1.txt`.

```
Politician A said "Politician B is a liar, because he promised in last
year's election that he would give every homeless person a home".
```

```
Politician B said "Politician A is a liar, because he promised in last
year's election that he would give every jobless person a job". Are these
functions mutually recursive?
```

Enter the following text in a text file named `file4.txt`.

```
Politician B said "Politician A is a liar, because he promised in last
year's election that he would give every homeless person a home".
Politician A said "Politician B is a liar, because he promised in last
year's election that he would give every jobless person a job". These
quotations are not mutually recursive.
```

Use `file1.txt` file and the `key = 7` as the test case for the program `encrytion0.c`. Load the program into Visual C++ and execute the program. The program should generate a file called `file2.txt`.

Hint: `file1.txt` must be in the same directory as the program.

11.2 Rewrite the `void encrypt(int offset, char *str)` function in the given program using pointer operations to replace array operations, for example, replace `str[i]` with `*(str+i)`. Replace the for-loop with a while-loop, where you must use a terminator to detect the end of the string. Comment the code where changes have been made.

11.3 Write a function called `int difference(char *filename1, char *filename2)` that compares two files. The file's names must be passed to the parameters `filename1` and `filename2`, respectively. The program should return the number of characters that are different (mismatches). For example, if the two files are exactly the same, the function returns 0. If the program detects 10 mismatches, it returns 10. If one file is longer than the other, the extra characters count as differences. This function must use string and pointer operations and compare each character one after another. You may not use the library function for string comparison. Write at least two lines of comment at the beginning of the function, describing what this function does and how it is implemented.

11.4 Write a function `void faultinjection(char *filename1, char *filename2, int n)` that injects `n` character faults into the file specified by the parameter `filename1`. Each character fault is a modification to a character by adding a random number between −10 and 10. The positions of the `n` faults are chosen randomly between the first character and the last character in the file. The modified file is stored in the file specified by the parameter `filename2`.

Note: To generate a random number, you can call a library function, for example, `rand()`. For a simple example, the following code prints 20 pseudo random numbers between 0 and 99. The function `srand(unsigned)` seeds the random-number generator with the current time so that the numbers generated by `rand()` will be different every time you call it.

```
#define size 100
#include <stdio.h>
#include <stdlib.h>    // function rand() is defined in this library
#include <time.h>      // function time(NULL)is defined in this library
main() {
    int i, rdm;
    srand((unsigned)time(NULL));  // Use current time as seed
    for (i = 0; i<20; i++) {
        rdm = rand() % size;     // modulo operation
        printf("random = %d\n", rdm);
```

KH ALL RIGHTS RESERVED

```
    }
}
```

11.5 Rewrite the `main()` function to perform the following operations described in the pseudo code.

```
encrypt file1.txt into file2.txt
decrypt file2.txt into file3.txt  // use a negative key
Find the differences between file1.txt and file3.txt
Find the differences between file1.txt and file4.txt
Call faultinjection (file4.txt, file5.txt, n); // choose n = 5
Find the differences between file4.txt and file5.txt
```

12. The following program generates maps representing mazes, where blank (space) characters represent open rooms through which a path may pass, while “X” characters represent closed rooms that cannot be used on any path. The starting position is marked by a character “S” and the goal position is marked by a character “G.” For a given maze, one can write a program to check if there is path from “S” to “G,” and print the paths if they exist. In this exercise, we generate only the mazes and do not attempt to write a program to find the paths. You are given the following C code, try to understand what it does and make the changes given in the following questions.

```
// This program exercises the operations on multidimensional array
#include <stdio.h>
#pragma warning(disable: 4996) // comment out if not in Visual Studio
#define maxrow 50
#define maxcolumn 50
char maze[maxrow][maxcolumn]; // Define a static array of arrays of
characters.
int lastrow = 0;
// Forward Declarations
int triple(int);
void initialization(int, int);
void randommaze(int, int);
void printmaze(int, int);

int triple(int x) {        // % is modulo operator.
    return ((x % 3 == 0) ? 1 : 0);
}
void initialization(int r, int c) {
    int i, j;
    for (i = 0; i < r; i++){
        maze[i][0] = 'X';     // add border
        maze[i][c - 1] = 'X'; // add border
        maze[i][c] = '\0';    // add string terminator
        for (j = 1; j < c - 1; j++)
        {
            if ((i == 0) || (i == r - 1))
                maze[i][j] = 'X'; // add border
            else
```

KH ALL RIGHTS RESERVED

```
                maze[i][j] = ' '; // initialize with space
            }
        }
}
// Add 'X' into the maze at random positions
void randommaze(int r, int c) {
    int i, j, d;
    for (i = 1; i < r - 1; i++) {
        for (j = 1; j < c - 2; j++) {
            d = rand();
            if (triple(d))
            {
                maze[i][j] = 'X';
            }
        }
    }
    i = rand() % (r - 2) + 1;
    j = rand() % (c - 3) + 1;
    maze[i][j] = 'S'; // define Starting point
    do {
        i = rand() % (r - 2) + 1;
        j = rand() % (c - 3) + 1;
    } while (maze[i][j] == 'S');
    maze[i][j] = 'G'; // define Goal point
}

// Print the maze
void printmaze(int r, int c) {
    int i, j;
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++)
            printf("%c", maze[i][j]);
        printf("\n");
    }
}
void main() {
    int row, column;
    printf("Please enter two integers, which must be greater than 3 and
less than maxrow and maxcolomn, respectively\n");
    scanf("%d\n%d", &row, &column);
    while ((row <= 3) || (column <= 3) || (row >= maxrow) || (column >=
maxcolumn)) {
        printf("both integers must be greater than 3. Row must be less
than %d, and column less than %d. Please reenter\n", maxrow, maxcolumn);
```

KH ALL RIGHTS RESERVED

```
        scanf("%d\n%d", &row, &column);
    }
    initialization(row, column);
    randommaze(row, column);
    printmaze(row, column);
    //encryptmaze(row, column);
    //printmaze(row, column);
    //decryptmaze(row, column);
    //printmaze(row, column);
}
```

The program above can be written using pointer operations, instead of using array indices, The code below shows the pointer version of the initialization function.

```
void initialization(int r, int c) {
    int i, j;
    char *p = 0;
    for (i = 0; i <= r; i++){
        p = &maze[i][0];
        // Tt points to initial address of the ith row of the maze
        *p = 'X';          // add left border
        *(p + c - 1) = 'X';    // add right border
        *(p + c) = '\0'; // add string terminator
        for (j = 1; j < c - 1; j++){
            if ((i == 0) || (i == r - 1))
                *(p + j) = 'X'; // add top and bottom borders
            else
                *(p + j) = ' '; // initialize inner maze with space
        }
    }
}
```

KH
ALL RIGHTS
RESERVED

Now, you can follow the example to complete the following exercise questions.

12.1 Rewrite the function int triple(int) using macro definition.

12.2 Rewrite the function randommaze(row, column) by substituting pointer operations for all array operations. You may not use indexed operation like maze[i][j], except getting the initial value of the pointer.

12.3 Rewrite the function printmaze(row, column) by substituting string operations for all character operations.

12.4 Write the function encryptmaze(row, column) based on the pointer operations. The function will encrypt the maze in the following secret rules:

- An integer i will be added to each space character, where i is the row number of the character.
- An integer j will be added to each non-space character (X, S, and G), where j is the column number of the character. Do not encrypt the terminator character ‘\0’.

12.5 Write the function decryptmaze(row, column) to decrypt the maze.

12.6 Call encryptmaze(row, column) and decryptmaze(row, column) functions in the main() function by removing the comment marks, and call the printmaze(row, column) after encryption and after decryption.

13. You are given the following program. Save the given program under the name `contactbook.c`. The program takes a **command line parameter**: the database's name in which the contact records are to be saved, assuming the database name is `person.dbms`.

You can pass command line parameters within the Visual Studio environment as follows:

(1) Use Visual Studio to compile and build the program `contactbook.c`.
(2) Choose menu “`Project`” and “`Properties...`”.
(3) Under the item “`Configuration Properties,`” click on “`Debugging,`” and enter the file name `person.dbms` to the right of the field “`Command Arguments.`”
(4) Click OK to return.
(5) Now you can execute the program.

II. You can also pass command line parameters using the following method:

(1) Compile and build the program.
(2) Choose MS Windows “`Start`” Menu.
(3) Choose “`Run ...`”.
(4) Click on “`Browse ...`”.
(5) Browse to the folder where your `contactbook.c` is stored.
(6) Go into the folder “`Debug.`” (This folder should have been created by the compile and build commands in step 1.)
(7) Choose the executable program called `contactbook` and then click “`open.`”
(8) You should see the path “`...\Debug\contactbook.exe.`”
(9) Append the file name `person.dbms` to the end of the path, with a space in between. The entire command sequence should look like: “`...\Debug\ contactbook.exe`” `person.dbms`.
(10) Click “`OK.`” The program should start to run.

KH ALL RIGHTS RESERVED

The tasks of this assignment are as follows.

13.1 Read the program carefully and make sure you understand the program and each function in the program. Then add at least two lines of comments below each function’s forward declaration to explain what the function does.

13.2 Write a function called `sort()`. The function should sort the existing linked list by the name field in dictionary order. Use the simplest sorting algorithm. For example, the selection sort: find the name with the smallest dictionary value and place it in the first place in the linked list, and then find the name with the next smallest dictionary value and put it in the second place, and so on.

13.3 Add your `sort()` function into the program and modify the program to offer users an extra option “sort” in the menu.

13.4 Test each function of the program: insert, delete, search, and sort. You can quit the program and restart the program. The records stored in the linked list should be saved and reloaded into the list. To copy the output in Visual Studio: Highlight the text you want to copy, click on the small icon

“c:\” at the top-left corner of the output window. Choose Edit-Copy. Then you can paste the output into a text file.

```
// Manipulation of files and singly linked list
// Command line parameter inputs
#include <stdio.h>
#include <stdlib.h>   // malloc is defined in this library
#include <string.h>   // string operations are defined in this library
struct contact {
  char name[30];
  char phone[20];
  char email[30];
  struct contact *next;
}*head = NULL;
char *file_name;
// forward declaration
void menu();
void branching(char c);
struct contact* find_node(char *str, int *position);
void display_node(struct contact *node, int index);
int insert();
int deletion();
int modify();
int search();
void display_all();
void load_file();
void save_file();
int main(int argc, char *argv[]) {
     char ch;
     if(argc != 2) {  // Two command line parameters required
          printf("Command Line Parameters Required !\n");
          printf("Try again......\n");
          getchar();  // enter any character to return
          return -1;
     }
     printf("SINGLY LINKED LIST\n");
     printf("*****************");
     file_name = argv[1];
     load_file();

     do {
          menu();
          fflush(stdin);            // Flush the standard input buffer
          ch = tolower(getchar()); // read a char, convert to lower case
          branching(ch);
```

KH ALL RIGHTS RESERVED

```
     } while (ch != 'q');
     return 0;
}
void menu() {
    printf("\n\nMENU\n");
    printf("----\n");
    printf("i: Insert a new entry.\n");
    printf("d: Delete an entry.\n");
    printf("m: Modify an entry.\n");
    printf("s: Search for an entry.\n");
    printf("p: Print all entries.\n");
    printf("q: Quit the program.\n");
    printf("Please enter your choice (i, d, m, s, p, or q) --> ");
}
void branching(char c) {
     switch(c) {
         case 'i':    if(insert() != 0)
                          printf("INSERTION OPERATION FAILED.\n");
                      else
                          printf("INSERTED NODE IN THE LIST
SUCCESSFULLY.\n");
                      break;
         case 'd':    if(deletion() != 0)
                          printf("DELETION OPERATION FAILED.\n");
                      else
                          printf("DELETED THE ABOVE NODE SUCCESSFULLY.\n");
                      break;
         case 'm':    if(modify() != 0)
                          printf("MODIFY OPERATION FAILED.\n");
                      else
                          printf("MODIFIED THE ABOVE NODE SUCCESSFULLY.\n");
                      break;
         case 's':    if(search() != 0)
                          printf("SEARCH FAILED.\n");
                      else
                          printf("SEARCH FOR THE NODE SUCCESSFUL.\n");
                      break;
         case 'p':    display_all();
                      break;
         case 'q':    save_file();
                      break;
         default: printf("ERROR - Invalid input.\n");
                  printf("Try again.....\n");
                  break;
```

KH ALL RIGHTS RESERVED

```
    }
    return;
}
int insert() {
    struct contact *node;
    char sname[30];
    int index = 1;
    printf("\nInsertion module...............\n");
    printf("Enter the name of the person to be inserted: ");
    scanf("%s", sname);
    node = find_node(sname, &index); // find duplicates
    if(node != NULL) {
        printf("ERROR - Duplicate entry not allowed.\n");
        printf("A entry is found in the list at index %d.\n", index);
        display_node(node, index);
        return -1;
    }
    else {
        node = (struct contact*) malloc(sizeof(struct contact));
        if (node == NULL) {
            printf("ERROR - Could not allocate memory !\n");
            return -1;
        }
        strcpy(node->name, sname);
        printf("Enter his telephone number: ");
        scanf("%s", node->phone);
        printf("Enter his email address: ");
        scanf("%s", node->email);
        node->next = head;
        head = node;
        return 0;
    }
}
int deletion() {
    char sname[30];
    struct contact *temp, *prev;
    int index = 1;
    printf("\nDeletion module...............\n");
    printf("Please enter the name of the person to be deleted: ");
    scanf("%s", sname);
    temp = head;
    while (temp != NULL) // search for the node to be deleted
       if (stricmp(sname, temp->name) != 0) { //case insensitive
strcmp
```

KH ALL RIGHTS RESERVED

```
            prev = temp;
            temp = temp->next;
            index++;
        }
        else {
            printf("Person to be deleted is found at index %d.",
index);
            display_node(temp, index);
            if(temp != head) prev->next = temp->next;
            else head = head->next;
            free(temp);  // Garbage collection
            return 0;
        }
    printf("The person with name '%s' does not exist.\n", sname);
    return -1;
}
int modify() {
    struct contact *node;
    char sname[30];
    int index = 1;
    printf("\nModification module...............\n");
    printf("Enter the name whose record is to be modified in the\n");
    printf("database: ");
    scanf("%s", sname);
    node = find_node(sname, &index);
    if(node != NULL) {
        printf("Person to be modified is found at index %d.", index);
        display_node(node, index);
        printf("\nEnter the new telephone number of this person: ");
        scanf("%s", node->phone);
        printf("Enter the new email address of this person: ");
        scanf("%s", node->email);
        return 0;
    }
    else {
        printf("The person with name '%s' does not exist \n", sname);
        printf("database.\n");
        return -1;
    }
}
int search() {
    struct contact *node;
    char sname[30];
    int index = 1;
```

KH ALL RIGHTS RESERVED

```
    printf("\nSearch module...............\n");
    printf("Please enter the name to be searched in the database: ");
    scanf("%s", sname);
    node = find_node(sname, &index);
    if(node != NULL) {
        printf("Person searched is found at index %d.", index);
        display_node(node, index);
        return 0;
    }
    else {
        printf("The person '%s' does not exist.\n", sname);
        return -1;
    }
}
void display_all() {
    struct contact *node;
    int counter = 0;
    printf("\nDisplay module...............");
    node = head;
    while(node != NULL) {
        display_node(node, ++counter);
        node = node->next;
    }
    printf("\nNo more records.\n");
}
void load_file() {
    FILE *file_descriptor;
    struct contact *node, *temp;
    char str[30];
    file_descriptor = fopen(file_name, "rb"); // "b" for binary mode
    if(file_descriptor != NULL) {
        while(fread(str, 30, 1, file_descriptor) == 1) {
            node = (struct contact*) malloc(sizeof(struct contact));
            strcpy(node->name, str);
            fread(node->phone, 20, 1, file_descriptor);
            fread(node->email, 30, 1, file_descriptor);
            if(head != NULL) temp->next = node;
            else head = node;
            node->next = NULL;
            temp = node;
        }
        fclose(file_descriptor);
    }
}
```

KH ALL RIGHTS RESERVED

```
void save_file() {
    FILE *file_descriptor;
    struct contact *node;
    file_descriptor = fopen(file_name, "w");
    if(file_descriptor != NULL) {
        node = head;
        while(node != NULL) {
            fwrite(node->name, 30, 1, file_descriptor);
            fwrite(node->phone, 20, 1, file_descriptor);
            fwrite(node->email, 30, 1, file_descriptor);
            node = node->next;
        }
    }
    else {
        printf("\nERROR - Could not open file for saving data !\n");
        getchar();
        exit(-1);
    }
}
struct contact* find_node(char *str, int *position) {
    struct contact *temp = head;
    while (temp != NULL) {
        if (stricmp(str, temp->name) != 0) {
            temp = temp->next;
            (*position)++;
        }
        else return temp;
    }
    return NULL;
}
void display_node(struct contact *node, int index) {
    printf("\nRECORD %d:\n", index);
    printf("\t\tName:\t\t%s\n", node->name);
    printf("\t\tTelephone:\t%s\n", node->phone);
    printf("\t\tEmail Address:\t%s\n", node->email);
}
```

KH ALL RIGHTS RESERVED

14. Follow the fantastic-four abstract approach to write a recursive function to find the largest number in a given array of integers.

15. In Section 2.7, an array is sorted by a simple recursive function. In step 3 of the fantastic-four abstract approach, the `m` is selected to be `n - 1`. Rewrite the sorting program, but select `m` to be `n/2` (merge sort). You can assume that the initial size `n` is a power of 2 to simplify the problem.

16. Fibonacci numbers are defined by

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n \geq 2 \end{cases}$$

Follow the fantastic-four abstract approach to implement the function in C.

16.1 Define the size-n problem.

16.2 Define the stopping conditions and the return values.

16.3 Define the size-m problem.

16.4 Explain how you construct the size-n solution from the size-m solutions.

16.5 Implement the function in C that can be used to compute Fibonacci numbers for the integer $n \geq 0$.

16.6 Write a main program that takes the input of n from the keyboard; call the recursive function, and then print the result.

17. The Ackermann function is defined recursively for two nonnegative integers s and t as follows:

$$A(s, t) = \begin{cases} t + 1, & if\ s = 0 \\ A(s-1, 1), & if\ s > 0\ and\ t = 0 \\ A\big(s-1, A(s, t-1)\big), & if\ s > 0\ and\ t > 0 \end{cases}$$

17.1 Follow the fantastic-four abstract approach to implement the function in C. The function should take two integer numbers, m and n, and return the value of A(s, t), which is a long integer. Notice that the Ackerman function is a very rapidly growing function. Even values of 4 for m and n will yield an extremely large number, and thus using a long integer as the return value is necessary.

17.2 Write a main program that takes inputs of m and n from the keyboard; call the recursive function, and then print the result.

KH ALL RIGHTS RESERVED