COMPRO2 Course Notes
(Advanced Computer Programming Course Notes)
URCO Project Number 13MU209

Florante R. Salvador, Dr. Eng.
florante.salvador@dlsu.edu.ph
Software Technology Department
College of Computer Studies
De La Salle University

August 25, 2010

Contents

Chapter 1

Dynamic Memory Allocation

1.1 Motivation

What is **dynamic memory allocation**? Why do we have to learn about it? Is it really necessary for us to learn it?

Instead of giving you (i.e., the learner) the answers to these questions immediately, let us first direct our mental faculty to solving four problems. These problems were designed to lead you to the answers to the last two questions posed above.

Problem #1: Write a program that will allocate memory space for storing an integer value; thereafter, store a value of zero in the said memory space.

Solution: The program is trivial as shown in Listing 1.1.

Listing 1.1: Static memory allocation (example 1)

```
1 int main()
2 {
3     int x;
4
5     x = 0;
6     return 0;
7 }
```

In COMPRO1, we learned that memory space for storing a value can be obtained by declaring a variable that will hold the desired value. We use the term **memory allocation** to refer to the act of requesting and *setting aside contiguous bytes of memory in the RAM* for the use of an object. On the other hand, the term **memory deallocation** is used to refer to the act of releasing or *freeing up memory space* when it is no longer needed (for example, when a function terminates).

Memory allocation and deallocation for variable `x` in Listing 1.1 was handled automatically by the system.[1] This kind of memory allocation scheme is called **static memory allocation**.[2] It is the allocation scheme used for variables and function parameters.

Static memory allocation is simple. It shields the programmer from worrying about certain issues, specifically:

1. How much memory space needs to be allocated to a variable?
2. When will memory be allocated?
3. When will memory be freed?

At this point, an inquisitive mind would ask: "How much memory space was allocated for the use of variable `x`?" The answer to this question can be answered using the operator `sizeof()` which yields the size of a data type in number of bytes as a result. Listing 1.2 shows an example program that prints the size of an `int` data type.

Listing 1.2: Determining the size of a data type

```
1 #include <stdio.h>
2 int main()
3 {
4    printf("Size of int data type is %d bytes.\n", sizeof(int));
5    return 0;
6 }
```

[1]The term "system" here is vague. To simplify the discussion, let us assume that it refers to the memory manager of an operating system. You really need not worry about this for the time being.

[2]Note that the word static in the term "static memory allocation" should not be confused with the usage of the C keyword `static`.

► Self–Directed Learning Activity ◄

In the following self–directed learning activity, you will learn about the sizes of the different basic data types and pointer data types on the platform that you are using.

1. Encode and run the program in Listing 1.2. What is the size of `int`?
2. Edit the program such that it will also print the respective sizes of `char`, `float` and `double` data types. What are the reported values? Fill-up the missing size values in Table 1.1
3. Edit the program such that it will also print the size of the pointer data types `char *`, `int *`, `float *`, `double *` and `void *`. Fill-up the missing size values in Table 1.1.

Table 1.1: Size of basic and pointer data types

`sizeof(char)`		`sizeof(char *)`	
`sizeof(int)`		`sizeof(int *)`	
`sizeof(float)`		`sizeof(float *)`	
`sizeof(double)`		`sizeof(double *)`	
		`sizeof(void *)`	

♦ Some Remarks

Remark 1. The size of a data type dictates the range of values that can be assigned to a variable of that type.

Remark 2. The `sizeof(char)` is always 1 byte in any platform.[3]

Remark 3. The size of all the other data types are not necessarily the same across different platforms.[4] The answer to the question: *"What is the size of int?"* should be *"It is platform dependent."*.

Remark 4. `sizeof()` does matter! Assuming that the size of a particular data type is the same for all platform is one of the primary causes of portability problems.

[3]Platform refers to both the hardware (i.e., CPU) and the software (i.e., compiler) running on top of the hardware.

[4]The `sizeof(int)` on personal computers that were in existence in the mid 1980's yielded only 2 bytes. Current quadcore personal computers with a 64-bit compiler would report `sizeof(int)` equivalent to 8 bytes. Note, however, that the DEV-C/GCC installed in our laboratories is a 32–bit compiler so `sizeof(int)` is equal to 4 bytes only.

Consider the next problem which is a slight modification of Problem #1.

Problem #2: Write a program that will allocate memory space for storing 3 integer values. Thereafter, store a value of zero to the memory space allocated to the first integer, a value of one to the next, and a value of two to the memory space allocated to the third integer.

Solution: The solution is also trivial as shown in Listing 1.3. Memory allocation in this case was achieved through the declaration `int x0, x1, x2`.

Listing 1.3: Static memory allocation (example 2)

```
1 int main()
2 {
3     int x0, x1, x2;
4
5     x0 = 0;
6     x1 = 1;
7     x2 = 2;
8     return 0;
9 }
```

Problem #3 is basically an extension of the first two problems except that it needs a lot more memory space.

Problem #3: Write a program that will allocate memory space for storing 1,000,000 (one million) integer values! Thereafter, store a value of zero to the first integer, a value of one to the second, a value of two to the next, and so on...

▶ Self–Directed Learning Activity ◀

Solve Problem #3 using only the concepts that you learned in COMPRO1. Answer the questions below when you are done.

1. How much time did you spend (or you think you'll spend) to solve the problem?
2. How many variables have to be declared?
3. How many assignment statements are needed?
4. How long (in number of lines) is your source code? :-) ← note smiley here.

Consider next Problem #4. Can it be solved using only the concepts learned in COMPRO1? Think about it. You are encouraged to write/test/run a C program on a computer to see if it can be done. DO NOT turn to the next page without thinking first about a possible solution.

Problem #4: Write a program that will declare an integer variable, say with the name `n`*. The program should then ask the user to input the value of* `n` *using* `scanf()`*. Thereafter, the program should allocate memory space for* `n` *number of integers. For example, if the value of* `n` *is 50, then the program should allocate memory space for 50 integers. Thereafter, initialize the allocated memory spaces by storing a value of zero to the first integer, a value of one to the next, a value of two to the next, and so on. The last integer should be initialized to a value of* `n-1`*.*

♦ Some Remarks

Remark 1. Problem #3 is not really difficult. It represents a scenario where static memory allocation can still be used, but doing so will only result into a cumbersome and unacceptable solution, i.e., a lengthy code with one million variable names and one million assignment statements!

Remark 2. Static memory allocation requires that the size of the memory space to be allocated to objects be known during compile–time. Clearly this is impossible to do in Problem #4 because the size will be have to supplied during run–time, i.e., when the program is executing. The last problem illustrates a limitation of static memory allocation.

There are a lot of scenarios where a program will be required to handle data that may increase or decrease in size dynamically during run–time. Consider, for example, keeping track of the information on your contacts in social networking sites such as Facebook. Initially, you start out with just a few contacts; but as time pass by, your list of contacts may grow by adding new friends or it may decrease as you remove undesirable contacts. Thus, the list of contacts is not static (you're not limited to a maximum of just 5 friends for example) but it can grow or shrink dynamically.

1.2 Dynamic Memory Allocation

Dynamic memory allocation is a scheme that gives programmer direct control in managing the memory needs of a program. Specifically, this means that the programmer can give instructions to do the following during *run–time*:

a. allocate a block of memory space with a size that is not necessarily known during compile time

b. increase or decrease the size of an already allocated block of memory space[5]

c. free-up memory space when it is no longer needed

This scheme is very flexible and allows efficient use of the memory resource. On the other hand, it entails responsibility on the programmer to practice utmost care when writing source codes. Incorrect use of dynamic memory allocation can easily lead to run–time errors resulting to an abnormal program termination.

1.2.1 How do you allocate memory dynamically?

In C language, dynamic memory allocation request is achieved by calling the ANSI C library function `malloc()` which has the following prototype:

```
void *malloc(size_t n)
```

Parameter `n` is a whole number (where `n` > 0) denoting the size of the block of memory space in bytes. `malloc()` returns a value, i.e., a memory address, with `void *` as its data type.[6]

The syntax for using `malloc()` is as follows:

```
<pointer variable name> = malloc(<n>)
```

There are two possible outcomes:

1. Successful case: if the memory request can be satisfied, `malloc()` will return the address of the first byte of the allocated space. This address is assigned as the value of the pointer variable. The allocated space is not initialized, i.e., it contains garbage value.

[5]For example, 5KB bytes of previously dynamically allocated memory can be increased to say 8KB or decreased to 2KB.

[6]In C language, `void *` is a generic pointer data type.

2. Unsuccessful case: if memory is not enough, then the request cannot be satisfied. In such a case, `malloc()` returns `NULL` which is assigned as the value of the pointer variable.

It is the responsibility of the programmer to check first if the memory request was granted or not before dereferencing the pointer variable. Note that dereferencing a pointer with a `NULL` value will result into a semantic error.

An example program showing how to use `malloc()` is shown in Listing 1.4.

1.2.2 How do you deallocate memory dynamically?

Memory space which has been allocated dynamically should be relinquished explicitly so that it can be re-used for future memory requests. In C language, this is achieved by calling the ANSI C library function `free()`. Its function prototype is

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

Listing 1.4 shows an example of how to use the `free()` function.

Listing 1.4: Dynamic memory allocation (example 1)

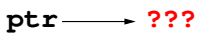
```
1  #include <stdio.h>
2  #include <stdlib.h>  /* DON'T forget to include this file */
3  int main()
4  {
5      int *ptr;
6
7      /* allocate memory for one integer*/
8      ptr = malloc(sizeof(int));
9
10     /* display the address of the 1st byte */
11     printf("The allocated space has an address of %p\n", ptr);
12
13     /* initialize the value of allocated space */
14     *ptr = 123;
15     printf("Value is %d\n", *ptr);
16
17     /* release memory */
18     free(ptr);
19     return 0;
20 }
```

1.2.3 Graphical Representation

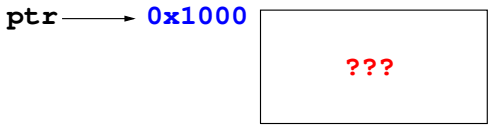
Figure 1.1 shows a graphical representation for dynamic memory allocation. The allocated memory block is represented as a rectangle. The number of bytes making up the memory block corresponds to the parameter passed to the `malloc()` function call. Function `malloc()` returns the **address** of the first byte which is stored by the recipient **pointer** variable. The value stored in the memory block corresponds to **value** shown inside the rectangular area.

Figure 1.2 shows the sequence of graphical representations corresponding to the codes in Listing 1.4. Note that address `0x1000` is hypothetical.

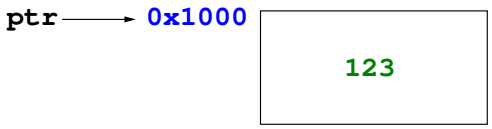
Figure 1.1: Graphical representation of a pointer and memory block



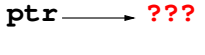
(a) Right after declaration of `int *ptr`



(b) Right after executing `ptr = malloc(sizeof(int))`



(c) Right after executing `*ptr = 123`



(d) Right after executing `free(ptr)`

Figure 1.2: Graphical representation for Listing 1.4, ??? denotes garbage.

► Self–Directed Learning Activity ◄

In this activity, you will learn the proper usage of `malloc()` and `free()`. You will also experience what happens to a buggy program due to mistakes such as (deliberately) omitting the call to `malloc()`.

1. Determine in which ANSI C header file we can find the function prototypes for `malloc()` and `free()`. Don't simply use Google to search for this information. Make sure that you open the header file (in your computer or in the lab) so that you can see for yourself how the functions were actually declared.

2. Assuming that you were able to find and open the header file, try to check the other functions related to dynamic memory allocation. Hint: the word "alloc" is part of the function name. Write down the functions prototypes, and determine by yourself the purpose of these functions.

3. Encode and run the program in Listing 1.4. What are the printed values? What is the meaning of the format symbol "`%p`" in the first `printf()` statement? Replace "`%p`" with "`%u`", compile and run the program. What is the output? What is the meaning of the format symbol "`%u`"?

4. What will happen if the line containing `malloc()` was removed? Will it cause a run-time error?

5. Using the original program in Listing 1.4, i.e., with `malloc()` present, what will happen if the line containing `free()` was removed?

6. Using the original program again, what will happen if we insert an assignment statement `*ptr = 456;` after calling `free()`? In other words, can we dereference a pointer after freeing the associated memory space? The code would appear as:

```
/* some codes here */
free();
*ptr = 456; /* can we dereference ptr like this? */
```

7. Write your own program/s for dynamically allocating memory space for storing one (i) `char`, (ii) `float`, (iii) `double`. Print the starting address of the allocated memory space, and initialize the values. Free the memory space before the program terminates.

8. Watch the "Binky Pointer Fun" video at `http://cslibrary.stanford.edu/104/`. It is a 3–minute clay animation video that can help you review the basics of pointers and memory allocation.

Some Important Rules/Reminders

1. It is possible to call `malloc()` and `free()` more than once as shown in the following code snippet.

```
/* allocate memory: 1st time */
ptr = malloc(sizeof(int));
*ptr = 5;

/*-- some other codes here --*/

free(ptr);  /* free memory if it's not needed anymore */

/*-- some other codes here --*/

/* allocate memory: 2nd time */
ptr = malloc(sizeof(int));
*ptr = 10;

/*-- some other codes here --*/

free(ptr);
```

   In the example above, `malloc()` and `free()` were called two times each. Notice also that we used the same pointer variable in both occasions.

2. Make sure that you free up a dynamically allocated space. Forgetting to do so will cause said space to become unusable, resulting into poor memory utilization.[7]

   For example, running the codes in Listing 1.4 inside a loop without `free()` for 1000 times will result into a total of 1000 * `sizeof(int)` bytes or approximately 4KB of unusable space (assuming `sizeof(int)` is 4).

   The following code snippet shows another common erroneous practice resulting into unusable space. The problem was caused by calling `malloc()` the second time, without first freeing up the memory space pointed to by `ptr`.

```
/* allocate memory: 1st time */
ptr = malloc(sizeof(int));
*ptr = 123;

/* allocate memory: 2nd time */
ptr = malloc(sizeof(int));
*ptr = 456;
```

[7] Recall item 5 from the previous Self–Directed Learning Activity.

3. It is a logical error to dereference a pointer variable after the memory space it pointed to was freed up.[8] Note that after freeing the memory space, it can no longer be accessed indirectly via the pointer variable. A pointer that does not point to a valid memory address is referred to as a *dangling pointer*. The following code snippet shows this problem.

```
/*-- some prior codes here --*/
free(ptr);  /* freed memory space is no longer accessible */
*ptr = 456; /* invalid dereference: ptr is a dangling pointer */
```

1.2.4 Allocating a block of memory

`malloc()` is normally invoked to allocate a *block* of memory which is made up of contiguous bytes much bigger than the size of a single basic data type. An example is shown in Figure 1.3 which indicates a dynamically allocated block that stores 3 integers indicated by the values 5, 10 and 15. For discussion purposes, we use hypothetical memory addresses with `0x1000` as the address of the 1st integer (and also of the first byte). Assuming 4 bytes as `sizeof(int)`, the address of the 2nd and 3rd integers are `0x1004` and `0x1008` respectively.

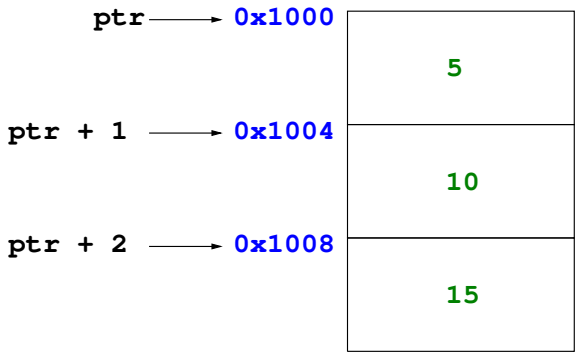


Figure 1.3: Graphical representation of a pointer and memory block

[8] Recall item 6 from the previous Self–Directed Learning Activity.

Listing 1.5 shows how to allocate a memory block that can store 3 integers. Notice that the parameter to `malloc()` is given as `sizeof(int) * 3`. If we want to allocate memory space for storing 1000 integers, we simply change the `malloc()` parameter to `sizeof(int) * 1000`.

Listing 1.5: Dynamic memory allocation (example 2)

```
1  #include <stdio.h>
2  #include <stdlib.h>  /* DON'T forget to include this file */
3  int main()
4  {
5      int *ptr;
6
7      /* allocate a block of memory for storing three integers */
8      ptr = malloc(sizeof(int) * 3);
9
10     /* display the address of the 1st byte */
11     printf("The address of the 1st byte is = %p\n", ptr);
12
13     /* we will not store any value for now... */
14
15     /* release memory */
16     free(ptr);
17     return 0;
18 }
```

▶ Self–Directed Learning Activity ◀

You will learn in the following activity how to allocate blocks of memory space for storing more than one instance of `char`, `float` and `double` data type values. You will also get to know how to compute the total memory space allocated.

1. Encode and run the program in Listing 1.5. What is the size of the memory block in bytes? What is the address of the first byte in the block? What is the address of the last byte?

2. Write your own program that will dynamically allocate memory space for (i) 10 characters, (ii) 10 floats, and (iii) 10 doubles. Print also the addresses corresponding to the first byte of these memory blocks. How many bytes were allocated for (i), (ii) and (iii) respectively? What is the address of the last byte for each case?

1.2.5 Memory addressing, base address and index

Only the address of the first integer is displayed in Listing 1.5. How can we determine the addresses associated with the second, and third integers? In order to answer this question, we have to introduce the concept of *base address* and *index*.

The *base address* is simply the address of the first byte of the allocated block of memory space. This is the value returned by a successful call to `malloc()`. Think of *index* as a number that uniquely identifies the relative position of an element among a group of elements arranged in a sequence. The first element has an index of 0, the second element has an index of 1, and so on. Thus, if there are m elements, then the indices are numbered as $0, 1, \ldots, m-1$.

Determining the address of a specific element within a block of memory is referred to as **memory addressing**. The address of an element is computed by adding the base address to the element's index. For example, the addresses of the three integers shown in Figure 1.3 and in Listing 1.5 are computed as follows: `ptr + 0` is the address of the first integer, `ptr + 1` is the address of the second integer, and `ptr + 2` is the address of the third integer. Note that we normally omit the addition of index 0 in the case of the first element.

Listing 1.6 shows an example of memory addressing. The address of the `i`'th element is computed as `ptr + i` where `ptr` represents the base address and `i` is the value of the index generated using a `for` loop.

Listing 1.6: Example of memory addressing

```
1  #include <stdio.h>
2  #include <stdlib.h>  /* DON'T forget to include this file */
3  int main()
4  {
5      int i;      /* i is the element's index */
6      int *ptr;   /* ptr is the base address  */
7
8      ptr = malloc(sizeof(int) * 3);
9
10     /* display the address of each element */
11     for (i = 0; i < 3; i++)
12         printf("The address of element %d = %p\n", i, ptr + i);
13
14     /* we will not store any value for now... */
15     free(ptr);
16     return 0;
17 }
```

► Self–Directed Learning Activity ◄

You will be able apply what you learned about memory addressing in this activity. You will also see how to extend the concept to other data types aside from `int`.

1. Encode and run the program in Listing 1.6. What addresses were generated for each integer element?
2. Subtract the address of the 1st element from the address of the 2nd element. What value did you get? Notice that the difference is equivalent to `sizeof(int)`.
3. Edit the program by changing the `malloc()` parameter to request for a block of memory for storing 5 integer elements. Without running the program, compute and write down the address of each element. Assume that the address of the first element is the same as before. Verify your answer by running the edited code.
4. Create new programs to apply the concept to the other basic data types, namely `char`, `float` and `double`. Allocate 5 elements each, and print the addresses of the elements. What is the difference between the addresses of two consecutive elements? Is it the same as the size of the data type?

1.2.6 Accessing the elements of a memory block

So far, we know how to: (i) dynamically allocate memory, (ii) determine the address of an element in a block of memory and (iii) free the memory.

To be able to access a particular element, we simply need to use the dereference (also called indirection) operator with the element's memory address as operand. Specifically, if we want to store a value say `v` to element with index `i`, we write:

```
*(ptr + i) = v;
```

Notice that it is necessary to parenthesize the expression `ptr + i` because the indirection operator `*` has a higher priority than addition. Without the parentheses, the expression will become syntactically incorrect. The following example shows how to initialize the values of the 3 integers elements to 5, 10, and 15 respectively:

```
*(ptr + 0) = 5;
*(ptr + 1) = 10;
*(ptr + 2) = 15;
```

Listing 1.7: Accessing elements

```
1  #include <stdio.h>
2  #include <stdlib.h>  /* DON'T forget to include this file */
3  int main()
4  {
5      int i;      /* i is the element's index */
6      int *ptr;   /* ptr is the base address  */
7
8      ptr = malloc(sizeof(int) * 3);
9
10     /* initialize elements */
11     *(ptr + 0) = 5;
12     *(ptr + 1) = 10;
13     *(ptr + 2) = 15;
14
15     /* display the value stored in each element */
16     for (i = 0; i < 3; i++)
17         printf("Value of element %d = %d.\n", i, *(ptr + i));
18
19     free(ptr);
20     return 0;
21 }
```

Listing 1.7 shows a complete program illustrating all the concepts that we learned so far: (i) dynamic memory allocation, (ii) memory addressing, (iii) accessing individual elements via indirection operator, and (iv) freeing up the memory.

▶ Self–Directed Learning Activity ◀

Apply what you learned in this section, and extend the concept to the other data types.

1. Encode and run the program in Listing 1.7.

2. Remove the pair of parentheses from the statement `*(ptr + 2) = 15;`. Compile the program. What is the error message? Find out the meaning of the error message.

3. Revert back to the original program. Remove the inner pair of parentheses enclosing the expression `ptr + i` in the `printf()` statement. Will this cause

a syntax error? Why not? Compile the edited code, and take note of the output. Compare them with the results you got using the original program.

Based on the results, explain the semantic difference between `*(ptr + i)` and `*ptr + i` (aside from the obvious fact that the other does not have a pair of parentheses).

4. Edit the original program in Listing 1.7 to allocate a memory block for 5 integers. Initialize the integers with any valid value that you want. Modify the `for` loop condition as well. Run and test your program to verify that it works correctly.

5. Edit your code such that the values of the elements will be printed in reversed sequence. That is, print first the value of the 5th element down to the 1st element. Which part of the program do you need to modify?

6. Repeat the last two activities considering `char`, `float` and `double` data types.

We now turn our attention back to Problems #3 and #4 (refer to pages 4 and 5) and see how they can be solved using dynamic memory allocation. Problem #3 can actually be solved using static memory allocation. However, if we use only our COMPRO1 background knowledge, it would require a lengthy solution with a million variable declarations, and a million explicit assignment statements.[9] Listing 1.8 shows a solution using dynamic memory allocation. Note that only two variables are needed, i.e., a pointer variable to the memory block and a variable for the element index. The assignment inside the `for` loop achieves the required initialization.

The program also illustrates the need to check the return value of `malloc()` – neglected deliberately in previous programs. A memory request for a very large block size may not be satisfied. The programmer has to decide what to do when such a case occurs. In the sample program, we chose to halt program execution via pre–defined function `exit(1)` indicating abnormal program termination.

Listing 1.8: Problem #3 solution

```
1  #include <stdio.h>
2  #include <stdlib.h>  /* DON'T forget to include this file */
3  int main()
4  {
5      int i;      /* i is the element's index */
6      int *ptr;   /* ptr is the base address  */
7      /* allocate memory for one million integer elements */
8      ptr = malloc(sizeof(int) * 1000000);
9      if (ptr == NULL) {
10         printf("Not enough memory. Program will now terminate.\n");
11         exit(1); /* 1 means abnormal program termination */
12     }
13
14     /* if ptr != NULL, the following codes will be executed */
15     for (i = 0; i < 1000000; i++)
16         *(ptr + i) = i;  /* initialize the elements */
17
18     /* display the value stored in each element */
19     for (i = 0; i < 1000000; i++)
20         printf("Value of element %d = %d.\n", i, *(ptr + i));
21
22     free(ptr);
23     return 0;
24 }
```

[9] Actually, there is a way to allocate space for a million integer elements using only a single variable declaration. This will be explained in the next chapter on Arrays.

Problem #4 cannot be solved using static memory allocation because the size is unknown during compile time. The only possible way to solve this problem is via dynamic memory allocation. The solution is shown in Listing 1.9.

Listing 1.9: Problem #4 solution

```
#include <stdio.h>
#include <stdlib.h>  /* DON'T forget to include this file */
int main()
{
    int i;      /* i is the element's index */
    int m;      /* m is the number of elements to be allocated */
    int *ptr;   /* ptr is the base address  */

    printf("Input number of elements to be allocated: ");
    scanf("%d", &m);
    /* notice the use of m in the malloc() parameter */
    ptr = malloc(sizeof(int) * m);

    if (ptr == NULL) {
        printf("Not enough memory. Program will now terminate.\n");
        exit(1); /* 1 means abnormal program termination */
    }

    /* if ptr != NULL, the following codes will be executed */
    for (i = 0; i < m; i++)
        *(ptr + i) = i;   /* initialize the elements */

    /* display the value stored in each element */
    for (i = 0; i < m; i++)
        printf("Value of element %d = %d.\n", i, *(ptr + i));

    free(ptr);
    return 0;
}
```

▶ Self–Directed Learning Activity ◀

Apply what you learned from the previous discussion.

1. Encode and run the programs in Listing 1.8 and 1.9.
2. Do an information search on ANSI C library function `exit()`. What is the use of `exit()`? What is the meaning of the parameter of `exit()`? What is the difference between `exit(0)` and `exit(1)`?

1.3 Memory addressing and scanf()

`scanf()` is used to input the value of a variable via the standard input device, i.e., keyboard. One of the required parameters of this function is the address of a variable. For example, `scanf("%d", &n);` will input the value of an integer variable `n` whose address is `&n`.

The same rule applies to input of values for elements in a dynamically allocated memory block. To input the value of a particular element, we supply its memory address as a parameter to `scanf()`. Listing 1.10 shows how this is done. Take note also of the condition made inside the `if()` statement. This is the way an experienced programmer would allocate and test the result of `malloc()`.

Listing 1.10: Initializing a memory block using `scanf()`

```
1  #include <stdio.h>
2  #include <stdlib.h>  /* DON'T forget to include this file */
3  int main()
4  {
5      int i;      /* i is the element's index */
6      int *ptr;   /* ptr is the base address  */
7      /* allocate memory for 5 integers */
8      if ((ptr = malloc(sizeof(int) * 5)) == NULL)
9         exit(1);
10
11     /* initialize the elements via scanf() */
12     for (i = 0; i < 5; i++) {
13         printf("Input value of element %d: ", i);
14         scanf("%d", ptr + i); /* no need to parenthesize ptr + i */
15     }
16     for (i = 0; i < 5; i++)
17         printf("Value of element %d = %d.\n", i, *(ptr + i));
18
19     free(ptr);
20     return 0;
21 }
```

► Self–Directed Learning Activity ◄

Apply what you learned in this section, and extend the concept to `float` and `double` data types.

1. Encode and run the program in Listing 1.10.
2. Edit the codes to handle (i) `float` and (ii) `double` data types.

1.4 Memory addressing and parameter passing

In the previous section, we showed how elements of a memory block can be accessed by passing the element's memory address to `scanf()`. We generalize this idea to user–defined functions.

It is a recommended practice to organize programs as a collection of functions. We re–implement the program in Listing 1.10 as three user–defined functions. The first function will be in–charge for the input of elements, the second is for the output of the elements, and `main()` as the third function. The first and second functions will have a memory address as formal parameter.

The modified program is shown in Listing 1.11. Functions `InputElements()` and `PrintElements()` have two parameters each. `ptr` is the base address of the memory block, and `n` is the number of elements in the block. Having parameter `n` makes the function generalized, i.e., it will work with any number of elements (not just 5).

Listing 1.11: Passing memory address as parameter

```
1  #include <stdio.h>
2  #include <stdlib.h>  /* DON'T forget to include this file */
3
4  /* Input the value of each element of a memory block */
5  /* ptr is the base address, n is the number of elements */
6  void InputElements(int *ptr, int n)
7  {
8      int i;
9
10     for (i = 0; i < n; i++) {
11         printf("Input the value of element %d: ", i);
12         scanf("%d", ptr+ i);
13     }
14 }
15
16 /* Display the value stored in each element. */
17 void PrintElements(int *ptr, int n)
18 {
19     int i;
20
21     for (i = 0; i < n; i++)
22        printf("Value of element %d = %d.\n", i, *(ptr + i));
23 }
24
```

```
25 int main()
26 {
27     int *ptr;  /* ptr is the base address  */
28
29     /* allocate memory for 5 integers */
30     if ((ptr = malloc(sizeof(int) * 5)) == NULL)
31        exit(1);
32
33     InputElements(ptr, 5);
34     PrintElements(ptr, 5);
35
36     free(ptr);
37     return 0;
38 }
```

▶ Self–Directed Learning Activity ◀

Apply what you learned in this section, and extend the concept to `float` and `double` data types.

1. Encode and run the program in Listing 1.11.
2. Implement a new function `SumElements()` that will compute and return the sum of the elements. For example, if the values of the 5 elements are 10, 20, 30, 40 and 50 respectively, the computed sum should be 150. Test your implementation by inserting the statement

   ```
   printf("The sum of the elements is %d\n", SumElements(ptr, 5));
   ```

   after the line containing `PrintElements(ptr, 5)`.
3. Repeat these activities with (i) `float` and (ii) `double` data types.

1.5 Pointer Arithmetic

The expression `ptr + i` that we have encountered in the discussion of memory addressing in subsection 1.2.5 is very interesting for two reasons:

1. Notice that the operands of `+` have different data types; `ptr` is `int *` while `i` is `int`. The resulting expression is a memory address with `int *` type.

2. It does not behave exactly like the usual addition. Try to recall the results you obtained in Listing 1.6 and review your answers to the items in the Self–Directed Learning activity.

The output of Listing 1.6 on the author's computer produced the following hexadecimal addresses

```
The address of element 0 = 0x804a008
The address of element 1 = 0x804a00c
The address of element 2 = 0x804a010
```

which indicates that the base address `ptr` has a value of `0x804a008`. The addition of 1 to `ptr`, i.e., `ptr + 1` did not produce `0x804a009` but resulted into `0x804a00c`. Moreover, `ptr + 2` is not equal to `0x804a00a` but computed as `0x804a010`. Taking the difference between the addresses of two consecutive elements yields 4 — which is the `sizeof(int)` in the author's platform.

Table 1.2 lists the addresses of memory blocks (from the author's computer) considering the four basic data types. The difference between the addresess of two consecutive elements is equal to the size of the data type in consideration. These results indicate that it is incorrect to interpret `ptr + i` using the usual notion of arithmetic addition.[10]

Table 1.2: Summary of addresses obtained (using author's computer)

	`char`	`int`	`float`	`double`
base address	`0x804a008`	`0x804a008`	`0x804a008`	`0x804a008`
base address + 1	`0x804a009`	`0x804a00c`	`0x804a00c`	`0x804a010`
base address + 2	`0x804a00a`	`0x804a010`	`0x804a010`	`0x804a018`

[10]For example, in simple arithmetic, the expression `x + 1` where `x = 5` will result into a value of 6.

In the C programming language, the addition or subtraction of an integer value to a memory address is called *address arithmetic* or *pointer arithmetic*. Adding 1 to a pointer means to refer to the "address of the next element", while subtracting 1 means to refer to the "address of the previous element". Note, however, that multiplication and division with a memory addresses is undefined.

An illustrative example of pointer arithmetic is shown in Listing 1.12. Two pointers were declared; `ptr` points to the first byte of the memory block, while `ptemp` is a temporary pointer used to access elements in the block. Notice the use of the increment and decrement operators. The `++` moves the temporary pointer to the address of the next element, while `--` moves it to the previous element.

Listing 1.12: Pointer arithmetic example

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5      int i;       /* i is the element's index */
6      int *ptr;    /* ptr is the base address  */
7      int *ptemp; /* ptemp is a temporary pointer */
8
9      /* allocate memory for 3 integers */
10     if ((ptr = malloc(sizeof(int) * 3)) == NULL)
11         exit(1);
12
13     ptemp = ptr; /* make ptemp point also to the memory block */
14
15     /* initialize the elements using ptemp only */
16     *ptemp = 5;
17
18     ptemp++;  /* ++ makes ptemp point to NEXT element */
19     *ptemp = 10;
20
21     ptemp++;
22     *ptemp = 15;
23
24     /* display values of elements using ptr */
25     for (i = 0; i < 3; i++)
26         printf("Value of element %d = %d.\n", i, *(ptr + i));
27
28     /* display elements in reversed order using ptemp */
29     printf("Reversed %d\n", *ptemp);
```

```
30
31      ptemp--; /* -- makes ptemp point to PREVIOUS element */
32      printf("Reversed %d\n", *ptemp);
33
34      ptemp--;
35      printf("Reversed %d\n", *ptemp);
36
37      free(ptr);
38      return 0;
39  }
```

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 1.12.
2. It is possible to replace some of the repetitive codes with a loop mechanism. Remove lines 29 to 35 and replace them with the following:

   ```
   while (ptemp >= ptr) {  /* note: comparison of pointers */
        printf("Reversed %d\n", *ptemp);
        ptemp--; /* -- makes ptemp point to PREVIOUS element */
   }
   ```

   This code snippet illustrates that a pointer value can be compared with that of another pointer.
3. Use one of the programs from any of the previous activities. Try the following one at a time; i.e., do not type all the codes in one program together. Take note of the error reported by the compiler.
   (a) Can we use multiplication with a pointer? Find out by inserting the following code:

   ```
   printf("%p\n", ptr * 2);
   ```

   (b) Can we use division with a pointer? Insert the following code:

   ```
   printf("%p\n", ptr / 2);
   ```

   (c) Can we use modulus with a pointer? Insert the following code:

   ```
   printf("%p\n", ptr % 2);
   ```

   (d) Can we compare a pointer value with an integer? Insert the following code:

   ```
   if (ptr > 123456)
      printf("TRUE\n");
   ```

1.6 What is NULL?

`NULL` is a pre–defined macro. It is normally used in the following situations:

1. `NULL` is the value returned by a function (whose return type is a pointer data type) to indicate failure in achieving a certain requirement. Representative examples, in the case of pre–defined functions, are:

   (a) `malloc()` returns `NULL` if it cannot allocate the requested memory

   (b) `fopen()` returns `NULL` if a file cannot be opened[11]

   It is the programmer's responsibility to test if the return value is `NULL` and take appropriate action in such a case. In the previous sample programs (starting from Listing 1.8), we simply stopped program execution via `exit(1)` to indicate abnormal program termination.

2. `NULL` is used to initialize a pointer variable. An example is shown below:

   ```
   int *ptr;

   ptr = NULL;
   /*--  some other codes follow --*/
   ```

   A pointer variable contains a garbage value right after declaration. Garbage value, in this case, means that the pointer contains bit combination corresponding to some memory address. However, it is considered a logical error to dereference a garbage address.

   We set the value of a pointer variable to `NULL` to indicate the fact that it was properly initialized. A pointer variable set to `NULL` means that it does not point (yet) to any allocated memory space. It is also semantically incorrect to dereference a pointer set to `NULL`.

   There is no need to initialize a pointer variable to `NULL` if it will be used immediately as a recipient for the `malloc()` return value. For example:

   ```
   int *ptr;

   ptr = NULL;    /* this is an unnecessary initialization */
   ptr = malloc(sizeof(int));
   ```

3. `NULL` is used in linked lists to indicate that a node is the first node or the last node in the list.[12]

[11] This will be discussed in detail in the chapter on File Processing.

[12] More details about this covered in the chapter on Linked Lists.

► Self–Directed Learning Activity ◄

1. `NULL` as mentioned above is a pre–defined macro. In which header file is it defined? Do not simply search for this information using Google. Open the header file and see for yourself how `NULL` is actually defined. Fill up the missing item indicated by the underline.

   ```
   #define NULL ________
   ```

   DO NOT go to next page without answering this item first.

2. Can `NULL` be assigned to a non–pointer data type variable? Verify your answer by creating a program to test what will happen if `NULL` is assigned to (i) `char` variable, (ii) `int` variable, (iii) `float` variable, and (iv) `double` variable. What is the warning message produced by `gcc`?

3. Create a program that will a declare a pointer variable, initialize it to `NULL` and finally dereference it. Is there a compilation error? Is there a run–time error? What caused the error?

There is a way to see the actual value of `NULL` as it was defined in the specific C distribution installed in your computer. To do this, encode the program in Listing 1.13 and save it into a file that we will call as `test.c`.

Listing 1.13: Test program for NULL value

```
1 #include <stdio.h>
2 int main()
3 {
4     int *ptr;
5
6     ptr = NULL;  /* notice what will happen to NULL */
7     printf("NULL = %p\n", NULL);
8     return 0;
9 }
```

Thereafter, invoke the `gcc` compiler from the command line as follows:

```
gcc -E test.c
```

The `-E` after `gcc` is a command line option that tells the compiler to perform pre–processing only, i.e., it will not perform compilation and linking. In the example above, the macro `NULL` will be replaced with its actual value as defined in the header file.

Look at the results you got on your screen display. What value replaced `NULL`? Also, did you notice what happened to the comment?

In the author's computer, part of the pre–processed code resulted into:

Listing 1.14: Partial code after pre-processing

```
1
2 int main()
3 {
4     int *ptr;
5
6     ptr = ((void *)0);
7     printf("NULL = %p\n", ((void *)0));
8     return 0;
9 }
```

Note that after pre–processing, the two occurences of the name `NULL` were replaced by `((void *)0)`. Based on this result, we can conclude that somewhere in `stdio.h` there is a macro definition that looks like:

```
#define NULL ((void *)0)
```

The `(void *)` that appears before `0` is a *data type cast*. This means that the value `0` was explicitly forced to change its data type from `int` to `void *`. [13]

Technically, the assignment `ptr = NULL` can be written instead as `ptr = 0`. However, this is not recommended by experienced C programmers. If somebody reads the codes, and sees only the assignment and not the declaration of variable `ptr`, the assignment maybe be incorrectly interpreted as assigning zero to a variable with a data type of `int` (or any of the basic data type). The use of the macro `NULL` makes it very clear that the recipient variable is a pointer data type. As a consequence, the value `NULL` should not be used to initialize a non–pointer data type variable.

▶ Self–Directed Learning Activity ◀

1. Search for the meaning of the words *data type cast*. You might also want to check *data type coercion* which is an alternative term used in the literature.

2. The data type of `NULL` based on the discussions above is `void *`. This means that `NULL` can be assigned to any pointer data type variable. Verify this by creating a program that will initialize three pointer variables with the following data types (i) `char *`, (ii) `float *`, and (iii) `double *` to `NULL`.

3. Find out if it is possible to assign a pointer variable initialized to `NULL` to another pointer variable of a different data type. For example:

```
int *pi;    /* int pointer variable */
float *pf;  /* float pointer variable */

pi = NULL;
pf = pi;    /* initialize pf also to NULL; is this OK? */
```

   Compile the program and take note of the compilation result. Was there a warning? If yes, what warning was caused by the integer pointer to float pointer assignment?

[13] It is instructive, at this point in time, to recall that `void *` is C's generic pointer data type. It is also the return type of `malloc()`.

1.7 A program that will run out of memory

The programs that we developed so far are "trivial" and are unlikely to result to an unssuccessful `malloc()`. We show in Listing 1.15 how to create a program that will ensure that memory will become insufficient causing `malloc()` to return `NULL`. The idea here is to allocate 1 million integers without freeing the memory block in each iteration of an endless loop. At some point in time, the memory request can no longer be satisfied and the program will terminate via `exit(1)`. A counter is used to keep track of how many times `malloc()` was called successfully before the program terminated.

Listing 1.15: Program that exhausts memory

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5      int *ptr;
6      int ctr = 0; /* count number of times malloc() was called */
7
8      while (1) { /* infinite! condition is always TRUE! */
9          if ((ptr = malloc(sizeof(int) * 1000000)) == NULL) {
10             printf("Not enough memory.\n");
11             exit(1);
12         }
13
14         ctr++;
15         printf("count = %d, ptr = %u\n", ctr, ptr);
16     }
17     return 0;
18 }
```

► Self–Directed Learning Activity ◄

1. Encode and run the program in Listing 1.15. Observe the value of `ctr` and `ptr` as displayed by the `printf()` function. What was the value of the counter when the program terminated? Approximately how much memory, in total, was allocated before the program terminated?

2. Modify the program by inserting `free(ptr)` say between `ctr++` and `printf()`. Execute the program and observe again the values printed, specifically the value of `ptr`. Explain why the program will never terminate. What do you think will happen to `ctr`?

1.8 Function returning a pointer

Consider the program shown in Listing 1.16. The `main()` function calls `malloc()` to request for a block of memory for storing 5 integer elements. The recipient variable `ptr` stores the base address of the memory block. Thereafter, `main()` calls function `Initialize()` passing the value of `ptr` as parameter. The formal parameter `ptemp` will contain a *copy* of the value of the actual parameter `ptr`. It is very important to note that there are now two separate pointers to the memory block, namely `ptr` and `ptemp`. However, the name `ptr` is not known inside `Initialize()`. The elements of the memory block are then initialized by dereferencing the element's memory address which is given by the expression `ptemp + i`. When `Initialize()` terminates, the temporary pointer `ptemp` will also cease to exist. The `for` loop in `main()` displays the elements of the memory block.

Listing 1.16: Program that passes a memory address as parameter

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void Initialize(int *ptemp)
5  {
6      int i;
7
8      for (i = 0; i < 5; i++)
9          *(ptemp + i) = (i + 1) * 5;
10 }
11
12 int main()
13 {
14     int i;
15     int *ptr;
16
17     if ((ptr = malloc(sizeof(int) * 5)) == NULL) {
18        printf("Not enough memory.\n");
19        exit(1);
20     }
21     Initialize(ptr);
22     for (i = 0; i < 5; i++)
23         printf("Value = %d.\n", *(ptr + i));
24
25     free(ptr);
26     return 0;
27 }
```

Encode and run the program. The program should display the values 5, 10, 15, 20 and 25 one value per line of output.

Let us now introduce a change in the program. Specifically, we move the `malloc()` call from `main()` to `Initialize()`. The modified program is shown in Listing 1.17.

What do you think will be the output if we execute this program? Will the output be the same as in the original code? Encode and run the program to see the actual results.

Listing 1.17: Erroneous program with `malloc()` inside `Initialize()`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void Initialize(int *ptemp)
5  {
6      int i;
7
8      if ((ptemp = malloc(sizeof(int) * 5)) == NULL) {
9         printf("Not enough memory.\n");
10        exit(1);
11     }
12
13     for (i = 0; i < 5; i++)
14         *(ptemp + i) = (i + 1) * 5;
15 }
16
17 int main()
18 {
19     int i;
20     int *ptr;
21
22     Initialize(ptr);
23
24     for (i = 0; i < 5; i++)
25         printf("Value = %d.\n", *(ptr + i));
26
27     free(ptr);
28     return 0;
29 }
```

The output of the modified program will not be the same as the original. In fact, a careful reader should have realized that the program contains several semantic errors.

The program does not work due to incorrect parameter passing.[14] We trace the codes and dissect in detail the nature of the errors in the following exposition:

i. `ptr` contains, by default, garbage value after its declaration in `main()`.

ii. When `main()` calls `Initialize()`, memory space for parameter `ptemp` gets allocated, and it is assigned a *copy* of the value of `ptr` by virtue of parameter passing. In other words, `ptemp` also has garbage as its initial value.

iii. `Initialize()` calls `malloc()` which returns the base address of the allocated memory block. This address is then assigned as the new value of `ptemp`. *Question*: What is the value of `ptr` at this point in time? *Answer*: It is still garbage! That is, it does not point to the allocated memory block.

iv. The body of the `for` loop initializes the elements in the memory block by dereferencing `ptemp + i`.

v. When `Initialize()` terminates, pointer `ptemp` will also cease to exist. Note that the allocated memory block is not properly freed – this is a BIG error! It becomes an unusable memory space since there is no way to access it directly or indirectly via a pointer variable.

vi. Returning back to `main()`, the value of `ptr` remains to be garbage. Thus, the dereference `*(ptr + i)` inside the `printf()` is an incorrect memory access – a BIGGER error!!

The cause of the errors above is incorrect parameter passing. Instead of passing the value of `ptr`, we should have passed the *address* of `ptr`. Thus, the call to `Initialize()`, should be modified as `Initialize(&ptr)`. Consequently, the formal parameter should be corrected as `Initialize(int **ptemp)`, and the references to `ptemp` should also be modified as well. The corrected codes are shown in Listing 1.18.

The formal parameter `ptemp` has a data type of `int **` which is read as *integer pointer pointer* data type. The appearance of the asterisk character or the word pointer twice is not a typographical error. This actually means the `ptemp` contains the address of `ptr` as its value. In other words, `ptemp` points to variable `ptr`. Thus, the dereference `*ptemp` is an indirect access to the contents of `ptr`.

[14] It is instructive to review what you learned in COMPRO1 regarding parameter passing.

Listing 1.18: Corrected version with `malloc()` inside `Initialize()`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void Initialize(int **ptemp)
5  {
6      int i;
7
8      if ((*ptemp = malloc(sizeof(int) * 5)) == NULL) {
9         printf("Not enough memory.\n");
10        exit(1);
11     }
12
13     for (i = 0; i < 5; i++)
14         *(*ptemp + i) = (i + 1) * 5;
15 }
16
17 int main()
18 {
19     int i;
20     int *ptr;
21
22     printf("Corrected!\n");
23     Initialize(&ptr);
24
25     for (i = 0; i < 5; i++)
26         printf("Value = %d.\n", *(ptr + i));
27
28     free(ptr);
29     return 0;
30 }
```

It can be quite challenging, at least initially, for a beginning programmer to understand the corrected codes in full. The lines containing the formal parametrization `int **ptr` and the dereference `*(*ptemp + i) = (i + 1) * 5;` may require some time to comprehend.

There is another way to fix the erroneous program that does not require a difficult approach. We can modify the implementation of `Initialize()` such that instead of `void` return type, it will return a pointer to the allocated memory block. Details are shown in Listing 1.19.

Listing 1.19: Function returning a pointer

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int *Initialize(void)
5  {
6      int i;
7      int *ptemp;
8
9      ptemp = malloc(sizeof(int) * 5);
10
11     if (ptemp != NULL)
12         for (i = 0; i < 5; i++)
13             *(ptemp + i) = (i + 1) * 5;
14
15     return ptemp;
16 }
17
18 int main()
19 {
20     int i;
21     int *ptr;
22
23     ptr = Initialize();
24     if (ptr == NULL) {
25         printf("Not enough memory.\n");
26         exit(1);
27     }
28
29     for (i = 0; i < 5; i++)
30         printf("Value = %d.\n", *(ptr + i));
31
32     free(ptr);
33     return 0;
34 }
```

The function prototype `int *Initialize(void)` indicates that `Initialize()` is a function that returns an integer pointer. It is similar to `malloc()` which also returns a pointer (of type `void *`).

Detailed explanation on how the program works is provided below. It mirrors, step-by-step, the exposition for the erroneous code in Listing 1.17.

i. `ptr` contains, by default, garbage value after its declaration in `main()`.

ii. When `main()` calls `Initialize()`, memory space for local variable `ptemp` gets allocated. Its default value is garbage.

iii. `Initialize()` calls `malloc()` which returns the base address of the allocated memory block or `NULL` if it is unsuccessful. This address is then assigned as the new value of `ptemp`. At this point, `ptr` is still garbage.

iv. If `malloc()` was successful, the body of the `for` loop initializes the elements in the memory block by dereferencing `ptemp + i`.

v. The function returns the value of `ptemp` which is assigned to `ptr` in the `main()` function. When `Initialize()` terminates, pointer `ptemp` will also cease to exist. Note that the allocated memory block will not become unusable. Its base address will be stored in `ptr`. This allows the block to be accessible even outside of function `Initialize()` where it was allocated.

vi. Returning back to `main()`, the assignment statement initializes the value of `ptr` using the value returned by `Initialize()`. Thereafter, the elements are printed by dereferencing `ptr + i`.

It is very important to become proficient in understanding and writing functions with a pointer return type. Such functions will be needed to implement algorithms for manipulating dynamic data structures such as linked lists and binary trees.[15]

► Self–Directed Learning Activity ◄

1. Encode and run the programs in Listings 1.18 and 1.19.

2. Create your own programs with similar behaviour considering the other basic data types (i) `char`, (ii) `float` and (iii) `double`.

[15]Linked lists will be discussed in another chapter. Binary trees is one of the topics in another subject called DASALGO – Data Structures and Algorithms.

1.9 How do you reallocate memory?

NOTE: This section is optional. Aside from dynamically allocating and deallocating memory, it is possible also to change the size of a memory block associated with a pointer variable during run–time. The pre–defined function `realloc()` can be used to request for a decrease or increase of memory block size. It has the following function prototype:

```
void *realloc(void *ptr, size_t newsize)
```

where `ptr` is the pointer to the "original" memory block and `newsize` is the new size (in bytes) being requested. It is recommended to call `realloc()` as follows:

```
ptemp = realloc(ptr, newsize)
```

where `ptemp` is a temporary pointer which stores the value returned by `realloc()`. The result can be one of the following:

1. `ptemp == NULL`

   This means that `realloc()` returned `NULL` indicating that the request for a change in memory size was unsuccessful because there is not enough memory. In this case, the "original" memory block pointed to by `ptr` remains unchanged. The programmer has to decide what to do when this happens. The usual reaction is to terminate program execution by calling `exit(1)`.

2. `ptr != NULL and ptemp == ptr`

   This means that reallocation was successful, and that the base address of the "new" block is the same as that of the "original" block. The "new" block was obtained by (i) shrinking when `newsize` is smaller than the original size or (ii) growing/extending the "original block" when `newsize` is bigger than the original size.

3. `ptr != NULL and ptemp != ptr`

   This also means that reallocation was successful. However, note that the base address of the "new" block is different from that of the "original" block. This will happen when the "original" block cannot grow/extend because there is not enough free contiguous bytes immediately right after the last byte of the "original" block. The "new" block will be obtained from somewhere else. The contents of the "original block" will be copied onto the

"new" one with the remaining bytes uninitialized. Thereafter, the "original" block will be freed up.

For example: assume the case where the "original" memory block is 12 bytes in size, and that it occupies the addresses from 0022FF00 to 0022FF0B. Assume also that an integer variable occupies four contiguous bytes with addresses from 0022FF0C to 0022FF0F. We then issue a reallocation request with a new size of 36 bytes (three times the original size). It is clear from this scenario that the "original" block cannot simply grow or expand towards the higher memory addresses. This is due to the integer variable occupying the four bytes immediately after the last byte of the original memory block. The "new" block of 36 contiguous bytes will be obtained somewhere else. The values of the "original" 12 bytes will be copied to the first 12 bytes of the "new" memory block; the remaining 24 bytes are uninitialized. The "original" memory block will then be freed. In this scenario, `realloc()` will return an address different from that of the "original" memory block.

4. `ptr == NULL and ptemp != NULL`

   This actually means that there was no "original" block to start with. The function call `ptemp = realloc(NULL, newsize)` in this particular case has the same effect as calling `ptemp = malloc(newsize)`.

Listing 1.20 shows a simple example of how to use `realloc()`. The program first allocates memory for storing 3 integers, then initializes them to 5, 10, and 15 respectively. Thereafter, it requests for an increase in memory size to store 10 integers. The 7 additional elements are initialized to 20, 25, ..., 50. Lastly, we print the values stored in the memory block.

Listing 1.20: Increasing memory block size using `realloc()`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5      int i;
6      int *ptr;
7      int *ptemp;  /* temporary pointer */
8
9      /* first request memory for 3 integer elements */
10     if ((ptr = malloc(sizeof(int) * 3)) == NULL)
11        exit(1);  /* error: not enough memory */
12     printf("The allocated space has an address of %p\n", ptr);
13
14     /* initialize elements as: 5, 10, 15 */
```

```
15      for (i = 0; i < 3; i++)
16         *(ptr + i) = (i + 1) * 5;
17
18      /* reallocate memory to store 10 integer elements */
19      if ((ptemp = realloc(ptr, sizeof(int) * 10)) == NULL)
20         exit(1);  /* error: not enough memory */
21      printf("The re-allocated space has an address of %p\n", ptemp);
22
23      ptr = ptemp; /* make ptr point to the new block */
24      /* initialize additional memory space */
25      for (i = 3; i < 10; i++)  /* note: start with index 3 */
26         *(ptr + i) = (i + 1) * 5;
27
28      for (i = 0; i < 10; i++)
29           printf("Value of element %d = %d.\n", i, *(ptr + i));
30
31    free(ptr);
32    return 0;
33 }
```

Take note of how `realloc()` was called. The recipient pointer variable used was `ptemp`. If the result is `NULL`, the program terminates via `exit(1)` because there is not enough memory. If reallocation was successful, `ptr` is set to the same value as `ptemp`.

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 1.20. What are respective addresses of the (i) original memory block and the (ii) reallocated block?

2. Edit the program by adding new codes that will reallocate memory from a size of 10 integers to 20 integers. Initialize the new integers following the pattern. The program should now have one call to `malloc()` and two calls to `realloc()`.

3. Write new programs to try `realloc()` for storing data values based on the other basic data types (i) `char`, (ii) `float`, and (iii) `double`.

4. Assume that `ptr` is currently `0022FF70`, and that the size of the original block is 1024 bytes. We request a change in size corresponding to 5 million integers! However, the call was made as follows:

   ```
   ptr = realloc(ptr, sizeof(int) * 5000000)
   ```

   For an unsuccessful `realloc()`, what do you think will be the value of `ptr`? What do you think will happen to the original block of 1024 bytes?

Listing 1.21 shows another example program. This time, note that we start with 10 integer size memory block which is later reduced to 5 integer size. After reallocating memory, it is no longer semantically correct to access elements with addresses `ptr + 5` to `ptr + 9`.

Listing 1.21: Decreasing memory block size using `realloc()`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5      int i;
6      int *ptr;
7      int *ptemp;
8
9      /* first request memory for 10 integer elements */
10     if ((ptr = malloc(sizeof(int) * 10)) == NULL)
11        exit(1);  /* error: not enough memory */
12     printf("The allocated space has an address of %p\n", ptr);
13
14     /* initialize elements as: 5, 10, 15, ... */
15     for (i = 0; i < 10; i++)
16        *(ptr + i) = (i + 1) * 5;
17
18     /* reallocate memory to reduce to 5 integer elements */
19     if ((ptemp = realloc(ptr, sizeof(int) * 5)) == NULL)
20        exit(1);
21     printf("The reallocated space has an address of %p\n", ptemp);
22
23     ptr = ptemp;
24     for (i = 0; i < 5; i++)
25          printf("Value of element %d = %d.\n", i, *(ptr + i));
26
27    free(ptr);
28    return 0;
29 }
```

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 1.21. What are respective addresses of the (i) original memory block and the (ii) reallocated block?
2. Find out if it is indeed semantically incorrect to access elements outside of the new (reduced) size due to re–allocation. Do this by changing the condition in the last `for` loop from `i < 5` to `i < 10`.

Listing 1.22 shows an example where a reallocation results to a new block with a different base address from the original block.

Listing 1.22: Differing base address with `realloc()`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5      char *pch;
6      int *ptr;
7      int *ptemp;
8
9      /* allocate space for 5 integers */
10     ptr = malloc(sizeof(int) * 5);
11     *ptr = 123; /* initialize first integer only */
12     /* allocate space for 100 characters */
13     pch = malloc(sizeof(char) * 100);
14     printf("Before realloc():\n");
15     printf("  Value of pch  = %p\n", pch);
16     printf("  Value of ptr  = %p\n", ptr);
17     printf("  Value of *ptr = %d\n", *ptr);
18
19     /* reallocate for 500 integers */
20     ptemp = realloc(ptr, sizeof(int) * 500);
21     printf("\n");
22     printf("After realloc():\n");
23     printf("  Value of ptr    = %p\n", ptr);
24     printf("  Value of *ptr   = %d\n", *ptr);
25     printf("  Value of ptemp  = %p\n", ptemp);
26     printf("  Value of *ptemp = %d\n", *ptemp);
27     return 0;
28 }
```

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 1.22. Take note of the program's output. What is the value of base address of the original memory block allocated to `ptr`? What is the base address of the new memory block? Are they the same or different? Take note also of the values of `*ptr` and `*ptemp` after calling `realloc()`. Are they different? If yes, what do you think is the reason why they are different?

1.10 Experiments for the Curious

Learning what will NOT work (i.e., incorrect) is as important as learning what will work (i.e., correct). Try doing the following experiments. They are designed for the "curious" individual who wants to find out what will happen if a correct or prescribed way of doing things is not followed. In other words, we deliberately break rules and commit mistakes to see what will actually happen as a result.

1. Determine what will happen when a negative integer value is used as `malloc()` parameter. Use the following sample code for this experiment.

```
int *ptr;

if ((ptr = malloc(-1)) == NULL)  /* is negative size allowed? */
     printf("Unsuccessful: no memory space allocated.\n");
else {
        printf("Successful: memory space was allocated.\n");
        free(ptr);
}
```

2. Determine/explain what will happen if zero is used as `malloc()` parameter as shown in the following sample code.

```
int *ptr;

if ((ptr = malloc(0)) == NULL)  /* is zero size allowed? */
     printf("Unsuccessful: no memory space allocated.\n");
else {
         printf("Successful: memory space was allocated.\n");
         free(ptr);
}
```

3. Determine what will happen if we use free on a `NULL` pointer value as shown in the following sample code:

```
int *ptr = NULL;

free(ptr);  /* can we free a NULL pointer? */
```

4. Can we free a memory space associated to a variable allocated using static memory allocation? See the two sample codes shown below:

```
/* Code 1 */
int n;    /* n was allocated using static memory allocation */
int *ptr;

ptr = &n;
free(ptr);  /* can we free space associated to n? */
```

An equivalent scenario is shown in the following:

```
/* Code 2 */
int n;    /* n was allocated using static memory allocation */

free(&n);
```

5. Can we free a portion of a dynamically allocated memory space? Use the following codes to find the answer:

```
int i;
int *ptr;
int *ptemp;

/* allocate space for 5 integers */
if ((ptr = malloc(sizeof(int) * 5)) == NULL)
    exit(1);

for (i = 0; i < 5; i++)
    *(ptr + i) =  i * 5;   /* initialize elements */

ptemp = ptr + 3;   /* ptemp points to the 2nd to the last element */
free(ptemp);  /* can we free space used by the last two elements? */
```

6. Can we perform pointer arithmetic on a pointer initialized to `NULL` and thereafter dereference it? Use the following codes to find the answer:

```
int *ptr;

ptr = NULL;
ptr++;
printf("%p\n", ptr); /* what do you think is the value of ptr? */
printf("%d\n", *ptr); /* is it correct to dereference it? */
```

7. Come up with your own "curious" experiment/s. Email the author about your ideas. Interesting experiments will be included in future versions of this document with the contributor's name and email address included for proper acknowledgement.

1.11 Chapter Summary

The key ideas presented in this chapter are summarized below:

- Dynamic memory allocation, as opposed to static memory allocation, allows memory to be allocated, reallocated and freed during run–time.
- Dynamic memory allocation is programmer–controlled.
- It is a powerful technique requiring the programmer to be very careful in handling pointers.
- `void *malloc(size_t n)` is the function prototype for requesting a memory block of size `n` bytes. It returns the base address of the allocated memory if there is enough memory to satisfy the request; otherwise, it returns `NULL`.
- `void *realloc(void *ptr, size_t newsize)` is the function prototype for reallocating a memory block associated with `ptr`. It returns the base address of the new block of memory (which may or may not be the same as the address of the original block) if there is enough memory to satisfy the request; otherwise it returns `NULL.`
- `void free(void *ptr)` is the function prototype for freeing the block of memory pointed to by `ptr`. Immediately after calling `free()`, the associated `ptr` will be a dangling pointer so it will be semantically incorrect to dereference it.
- `NULL` is a value that can only be assigned to pointer variables. It is the value returned by an unsuccessful call to `malloc()` or `realloc()`.
- Do not forget to include the header file `stdlib.h` which contains the function prototypes for `malloc()`, `realloc()`, `free()` and `exit()`.
- The `NULL` macro is defined in `stdio.h` and also in `stdlib.h`.
- It is the programmer's responsibility to check if `malloc()` or `realloc()` returned a `NULL` value, and to take appropriate action. The usual response is to stop program execution by calling `exit(1)`.
- It is possible to have several calls to `malloc()`, `realloc()`, and `free()` within a user–defined function.
- The address of a specific element in a memory block is equal to the sum of the base address and the element's index.

- An element in a memory block can be accessed by dereferencing (using `*`) its associated memory address. A dereference operation on a pointer should be done only after memory was successfully allocated and before said memory is freed.

Problems for Chapter 1

For the following problems, assume that `plist` *is a pointer to an existing block of memory with* `n` *integer elements. As common representative example, let* n = 5 *with elements containing* 10, -5, 23, -8, *and* 74. *We will use the term "list" to refer to such a group of elements.*

Problem 1.1. Write a function `int CountPositive(int *plist, int n)` that will count and return the value of the number of positive integer elements in the list. For example, the call `CountPositive(plist, 5)` will return a value of 3 since there are 3 positive elements in the list, i.e., 10, 23 and 74.

Problem 1.2. Same as in the previous problem, except that this time, implement a function `int CountNegative(int *plist, int n)` which will return the number of negative elements in the list. For example, the call `CountNegative(plist, 5)` will return a value of 2.

Problem 1.3. Write a function `int CountOdd(int *plist, int n)` which will count and return the number of odd elements in the list. For example, the call `int CountOdd(plist, 5)` will return a value of 2 since there are two numbers in the list namely 5 and 23.

Problem 1.4. Write a function `int Minimum(int *plist, int n)` which will determine and return the smallest element in the list. For example, the call `Minimum(plist, 5)` will return a value of -8.

Problem 1.5. Write a function `int Maximum(int *plist, int n)` which will determine and return the largest element in the list. For example, the call `Maximum(plist, 5)` will return a value of 74.

Problem 1.6. Write a function `int Sum(int *plist, int n)` which will determine and return the sum of all the elements in the list. For example, the call `Sum(plist, 5)` will return a value of 94 (which is the computed as 10 + -5 + 23 + -8 + 74 = 94).

Problem 1.7. Write a function `float Average(int *plist, int n)` which will determine and return the average of the elements in the list. The average is

computed as the sum of the elements in the list divided by the number of elements. For example, the call `Average(plist, 5)` will return a value of 18.8 (which is computed as 94 divided by 5). Caution: it is incorrect to perform integer division in this problem.

Problem 1.8. Write a function `int Search(int *plist, int n, int key)` that will search `key` in the list. If it is in the list, the function should return the index of the element, otherwise it should return a value of -1. We assume that the elements in the list are unique, i.e., no two elements have the same value. For example the call `Search(plist, 5, -8)` will return a value of 3. This means that -8 was found in the list, and that its index is 3. The function call `Search(plist, 5, 62)` on the other hand will return a value of -1 which means that 62 was not found in the list.

Problem 1.9. Write a function `void RotateRight(int *plist, int n, int x)` which will rotate the elements towards the right by `x` number of positions. Assume that `x > 0`. For example, the rotated list after calling `RotateRight(plist, 5, 1)` will contain the elements 74, 10, -5, 23, -8. Using the original list, calling `RotateRight(plist, 5, 3)` will result into 23, -8, 74, 10, -5.

Problem 1.10. Same as the previous problem except that rotation is towards the left. Implement function `void RotateLeft(int *plist, int n, int x)`. Assume `x > 0`. For example, the rotated list after calling `RotateLeft(plist, 5, 1)` will contain the elements -5, 23, -8, 74, 10. Using the original list, calling `RotateLeft(plist, 5, 3)` will result into -8, 74, 10, -5, 23.

For the next two problems, assume that there are two blocks of memory of the same size. Let `psource` *be the pointer to the first block, and* `pdest` *be the pointer to the second block.*

Problem 1.11. Write a function `void Copy(int *psource, int *pdest, int n)` which will copy the elements of the first block of memory to the second block. For example: let `n = 5`, and the elements of the first block are 10, -5, 23, -8, and 74 respectively. The contents of the second block after calling `Copy(psource, pdest, 5)` will also be 10, -5, 23, -8, and 74 respectively.

Problem 1.12. Write a function `void ReversedCopy(int *psource, int *pdest, int n)` which will copy the elements of the first block of memory to the second block in *reversed order*. For example: let `n = 5`, and the elements of the first block are 10, -5, 23, -8, and 74 respectively. The contents of the second block after calling `ReversedCopy()` will be 74, -8, 23, -5, 10 respectively.

For the next problem, assume that there are three blocks of memory which are NOT

necessarily of the same size. Let `plist1`*,* `plist2` *and* `plist3` *be the pointers to the 3 lists with* `n1`*,* `n2` *and* `n3` *elements respectively.*

Problem 1.13. Write a function `void Concatenate(int *plist1, int n1, int *plist2, int n2, int *plist3)` which will copy the contents of the first and second lists onto the third list. For example: let 1, 3 and 5 be the elements of the first list, and 2, 4, 6, 8, and 10 be the elements of the second list. Calling `Concatenate(plist1, 3, plist2, 5, plist3)` will result into a third list containing 1, 3, 5, 2, 4, 6, 8, and 10 as its elements. On the other hand, calling `Concatenate(plist2, 5, plist1, 3, plist3)` will result into a third list containing 2, 4, 6, 8, 10, 1, 3, and 5 as its elements.

References

Consult the following resources for more information.

1. Binky Pointer Fun Video. `http://cslibrary.stanford.edu/104/`

2. comp.lang.c Frequently Asked Questions. `http://c-faq.com/`

3. Kernighan, B. and Ritchie, D. (1998). *The C Programming Language, 2nd Edition*. Prentice Hall.

Chapter 2

Arrays

2.1 Motivation

Let us start this chapter with a "fun" brain exercise. SUDOKU is an original Japanese number puzzle. It is presented as a table, as shown below, which is made up of 9 rows and 9 columns. Some spaces are blank, while some are filled up with an integer ranging from 1 to 9. Within a SUDOKU table there are "regions". Each region is made up of 3 rows and 3 columns. Regions are drawn bounded by thick lines.

The objective of the puzzle is to complete the table by filling up the missing numbers. The rule is very simple: a number cannot be repeated in the same column, or same row or region. Have fun solving the puzzle below!

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

Solving SUDOKU puzzles requires sound logic. It can be both entertaining, and intellectually fulfilling. SUDOKU became a big craze several years ago in many countries other than Japan. Today, you will find SUDOKU puzzles together with crossword puzzles in newspapers and magazines.

The popularity of SUDOKU prompted companies, and even university research laboratories to develop algorithms and computer programs for automatically generating and solving SUDOKU puzzles. There are a lot of websites offering daily SUDOKU puzzles with different levels of difficulty – catering from young children to adults alike. SUDOKU puzzles are available even in mobile devices such as cellular phones.

There are a host of other puzzles, games, and other computing problems requiring the use of a tabular structure. Examples include:

a. contact numbers in a phone book

b. ID numbers of students enrolled in a class

c. entries in an accounting ledger

d. chess board game

e. matrix for a system of linear equations

In this chapter, we will learn how we can represent and manipulate lists and tables such as SUDOKU puzzles in a computer using the C programming language.

2.2 What is an Array?

An array is a group of elements of the same data type, i.e., it is homogeneous. An array is characterized by four attributes, namely: (i) name, (ii) element type, (iii) size, and (iv) dimension.

The array's *name* is a user–defined identifier which is used to refer to the array's individual elements. The *size* attribute is a positive integer indicating the number of elements in the array. The *element type* is the data type of the elements in the array. The data type can be a simple data type (i.e., `char`, `int`, `float`, `double`) or a user–defined type (for example, structure). An array can be one–dimensional or multi–dimensional (i.e., dimension is greater than one).

In Chapter 1, we discussed how to use `malloc()` to allocate blocks of memory for storing groups of elements. An array is actually very similar to such a group in terms of "structure", i.e., the elements also reside in contiguous block of memory space. Memory addressing is exactly the same, i.e., the address of an array element can be computed by adding the base address with an element's index.

The main difference, however, is in the way memory space are allocated for arrays. The number of bytes needed by an array are dependent on two factors, namely (i) number of elements and (ii) size of each element. These are both known during compile time – thus, a contiguous block of memory space for an array are set aside using *static memory allocation*. This means that unlike dynamically allocated memory, an array's size remains fixed, i.e., the number of array elements cannot be decreased or increased, during run–time.

In the following discussions, we first deal with one–dimensional arrays. The concepts will then be extended to two–dimensional arrays, and higher–dimensional arrays.

2.3 One–Dimensional Array

For brevity we will write 1D to mean one dimension. We can graphically represent a 1D array as a vertical list of elements as shown in the example below. The first element is 8, while the last element is 3. Think of the 1D array drawn in this manner as one of the columns, for example, of a SUDOKU table. In general, the data type, number of elements and the value of each element of a 1D array will be dependent on the problem being solved.

8
9
7
2
5
1
6
4
3

Alternatively, a 1D array can also be drawn as a horizontal list of elements as shown in the following example. In this case, think of the 1D array as one of the rows of the SUDOKU table.

8	9	7	2	5	1	6	4	3

1D arrays are not limited to integer elements. Illustrated below are arrays of characters and floating point values.

'C'	'O'	'M'	'P'	'R'	'O'	'2'

42.75	-10.23	63.54	89.75	101.23

2.3.1 1D Array Declaration

The syntax for declaring a 1D array in the C programming language is as follows:

```
<data type> <name> [<size>]
```

where `data type` is the element's data type, `name` is the array's name, `size` is the array size, and a single pair of square brackets means that the array is one–dimensional.

Some example declarations are:

```
char string[10];   /* 1D array of 10 characters */
int A[5];          /* 1D array of 5 integers */
float Value[50];   /* 1D array of 50 floats */
double Data[100];  /* 1D array of 100 doubles */
```

By default, array elements are uninitialized. It is the programmer's responsibility to initialize individual elements.

Several arrays of the same data type can be declared on the same line. For example:

```
int X[5], Y[10], Z[20];  /* three 1D arrays */
```

Good programming practice, however, recommends that variables be declared one at a time per line of code for better readability.

2.3.2 1D Array Element Definition

In C, a variable declaration tells the compiler how much memory will be needed by a variable based on its type. By default, the value of the variable is garbage. A *variable definition*, on the other hand, indicates not only the memory requirements of a variable, but also its initial value. Just like variables of simple data types, an array may be defined.[1] The syntax is as follows:

```
<data type> <name> [<size>] = {<list of values>}
```

where the `<list of values>` is the set of values to be assigned to array elements in the order that they appeared. The values need to be separated by comma and placed inside a pair of curly brackets. Examples of definitions are:

```
char coursecode[7] = {'C', 'O', 'M', 'P', 'R', 'O', '2'};
int M[3] = {10, 20, 30};
double F[5] = {42.75, -10.23, 63.54, 89.75, 101.23};
```

2.3.3 Referencing an Element in a 1D Array

The syntax for referencing or accessing a particular array element is:

```
<name> [<index>]
```

where `index` takes on the same meaning as the discussions in Chapter 1. `index` is a whole number, and its range of values is from `0` to `size - 1`. Note that some textbooks and C compilers such as `gcc` use the term *subscript* instead of *index*. In this document, these two words mean the same thing.

Examples of valid array references based on the arrays declared above are:

```
string[0] /* refers to the 1st element of array string */
A[4]      /* refers to the last element of array A */
Value[9]  /* refers to the 10th element of array Value */
F[2]      /* refers to the 3rd element of array F */
```

[1] Actually, the array elements may be defined.

Examples of invalid array references are:

```
Data[-1]  /* semantic error: negative index */
X[5]      /* semantic error: index is more than size - 1 */
Y[1.25]   /* syntax error: index is not a whole number */
```

Listing 2.1 shows a very simple example program incorporating what we learned so far: (i) array declaration and (ii) how to reference array elements. There are two arrays in the example program, namely `coursecode` and `A`. The elements of `coursecode` are defined, while those of `A` are assigned after the array declaration.

Listing 2.1: Array declaration and accessing elements

```
1  #include <stdio.h>
2  int main()
3  {
4      char coursecode[7] = {'C', 'O', 'M', 'P', 'R', 'O', '2'};
5      int A[5];
6      int i;
7
8      /* note: for loop is often used together with arrays */
9      for (i = 0; i <  7; i++)
10         printf("%c", coursecode[i]);
11     printf("\n");
12
13     /* initialize elements of array A */
14     A[0] = 3;
15     A[1] = -5;
16     A[2] = 1000;
17     A[3] = 123;
18     A[4] = 47;
19
20     for (i = 0; i < 5; i++)
21         printf("Value of A[%d] is %d.\n", i, A[i]);
22
23     return 0;
24 }
```

► Self–Directed Learning Activity ◄

1. Encode and run the program in Listing 2.1. What is the output of the program?

2. Change the initial values of the elements of array `A` to any integer value that you want. Run the program and see the corresponding output.

3. It is possible to assign the result of an expression as initial value to array elements. For example, change the initialization of the first element of array `A` to `A[0] = 60 / 2 + 5;`. Initialize the remaining array elements (i.e., `A[2]` to `A[4]`) using any valid expression that you want. Run and test the program to see the result.

4. Valid array indices must be from `0` to `size - 1`. Use of indices outside of this range will result into a semantic error. Change the second `for` loop initialization to `i = -2`, and the condition to `i < 8`. Compile the program. Is there a compilation error? If there is no error, run the program. Is there any run–time error?

5. Use the original program in Listing 2.1. Find out what will happen if the statement: `A[5] = 789;` is inserted before the `for` loop. Compile the program. Is there a compilation error? If there is no error, run the program. Is there any run–time error?

6. Edit the `for` loop in original program in Listing 2.1 such that the output is a list of array elements in reversed order, i.e., from the last element down to the first element.

Array elements are usually initialized to zero. Listing 2.2 shows how this can be done using a `for` loop.

Listing 2.2: Initializing elements to zero

```
1  #include <stdio.h>
2  int main()
3  {
4      int A[5];  /* integer array A with size of 5 */
5      int i;     /* i will be used as array index */
6
7      /* initialize all array elements to zero */
8      for (i = 0; i < 5; i++)
9          A[i] = 0;
10
11     /* print the array element values */
12     for (i = 0; i < 5; i++)
13         printf("Value of A[%d] is %d\n", i, A[i]);
14
15     return 0;
16 }
```

► Self–Directed Learning Activity ◄

1. Encode and run the program in Listing 2.2. What is the output of the program?

2. It is possible to initialize the array elements with values other than zero. Edit the program by changing `A[i] = 0;` to `A[i] = (i + 1) * 5;`. Based on this, what will be the values of the array elements? Confirm your answer by running the program and taking note of the printed results.

3. Modify the program such that the values of the elements (from the first to the last) will be initialized as: -10, -8, -6, -4, and -2 respectively.

4. It is possible to initialize the array elements using `scanf()`. Edit the original program in Listing 2.2 by changing `A[i] = 0;` to `scanf("%d", &A[i]);`. Run and test the program to see if it works properly. Try to input a sequence of numbers that are not necessarily in any pattern, for example: 5, -3, 11, 2, 1000.

2.3.4 Working with multiple 1D arrays

Some programming problems require manipulation of multiple arrays within the same function as illustrated in Listing 2.3. The program declares two integer arrays named `A` and `B`, both of size 5. Thereafter, elements of array `A` are initialized via `scanf()`. The second `for` loop copies the elements of array `A` to array `B`. The last `for` loop outputs the values of array `B`.

Listing 2.3: Copying elements of array `A` to array `B`

```
1  #include <stdio.h>
2  int main()
3  {
4      int A[5];
5      int B[5];
6      int i;      /* i will be used as array index */
7
8      /* input elements of array A */
9      for (i = 0; i < 5; i++) {
10         printf("Input value of element %d: ", i);
11         scanf("%d", &A[i]);
12     }
13
14     /* copy elements of A to B */
```

```
15      for (i = 0; i < 5; i++)
16          B[i] = A[i];
17
18      /* output elements of B */
19      for (i = 0; i < 5; i++)
20          printf("Value of element %d is %d.\n", i, B[i]);
21
22      return 0;
23  }
```

At this point in time, an alert reader is probably asking, *"Can we not assign ALL elements of array* `A` *to* `B` *by just one assignment statement of the form* `B = A`? In the C programming language, operations on arrays have to be done on a per–element basis. It is syntactically incorrect to manipulate one whole array. Thus, an array–to–array assignment such as `B = A`, or the addition of two arrays such as `A + B` is not defined!

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 2.3. What is the output of the program?

2. Edit the code such that the program will copy the elements of `A` are copied to `B` in reversed order. That is, the last element of `A` is copied as the first element of `B` and that the first element of `A` is copied as the last element of `B`.

3. Verify that an array–to–array assignment is not allowed by inserting `B = A;` immediately after the first `for` loop. What is the compilation error?

4. Write a new program that will declare 3 integer arrays, say with the names `A`, `B`, and `C` which will store 5 elements each. Initialize the contents of array `A` using a `for` loop. Afterwards, initialize the contents of `B` using another loop. In a third loop, compute `C[i] = A[i] + B[i]`. Finally, output the contents of array `C` in another loop.

2.3.5 1D array and parameter passing

Can we pass a whole **array** as parameter to a function? The answer to this question is NO, and the reason was already explained above, i.e., *array–to-array assignment is not defined in the C programming language.*

A more detailed explanation is given as follows. Consider, for the meantime, a function whose prototype is `void Test(int x);`. It can be called as `Test(y)` where variable `y` contains some integer value. Based from what we learned about parameter passing (in COMPRO1), the *value* of the actual parameter `y` will be passed and *copied* to formal parameter `x`. The same concept, i.e., *copying the value of the actual parameter to the formal parameter*, does not extend to arrays because, as mentioned again above, *array–to–array to assignment is not defined.*

There is a reason why a whole array is not passed as parameter. For argument's sake, let us imagine, for the meantime, that arrays can be passed as parameters. What can happen when you have an array parameter that has a VERY BIG size, for example, 5 million double data type integers? (1) There might not be enough memory for the allocation of the formal parameter. (2) Even if memory was successfully allocated for a VERY BIG array, copying the elements from the actual to the formal parameter *may* take some precious time. This can cause slow program execution especially if the function involved will be called many times, for example, inside the body of a loop with 1000 iterations. Due to these concerns in both the dimensions of memory space and time, whole arrays are not passed as parameter in C.

It is, however, possible to pass the (value of the) address of the array as function parameter. Thereafter, the array elements can be accessed by dereferencing the memory addresses of the individual elements. We will not show the actual codes for this on the assumption that you have already *mastered* the concept of pointers, `&` and `*` operators and memory addressing from Chapter 1. In any case, it is suggested that you implement the program as a simple exercise.

We describe instead the "array notation" for accomplishing this task as illustrated in Listing 2.4.

The input of array elements are done in function `void InputElements(int A[], int n)`. The first formal parameter, i.e., `int A[]` tells the compiler that `A` is the *name* of a group of integer elements.[2] Note that it is not necessary to write (in–between the square brackets) a constant value corresponding to the actual array size. This function is called inside `main()` as `InputElements(A, 5)`.

[2]The square brackets `[]` is the indicator that A is the name associated to a group.

Listing 2.4: 1D array and parameter passing

```
1  #include <stdio.h>
2  void InputElements(int A[], int n)
3  {
4      int i;
5
6      for (i = 0; i < n; i++) {
7          printf("Input the value of element %d: ", i);
8          scanf("%d", &A[i]);
9      }
10 }
11
12 void PrintElements(int A[], int n)
13 {
14     int i;
15
16     for (i = 0; i < n; i++)
17         printf("Value of element %d = %d.\n", i, A[i]);
18 }
19
20 int main()
21 {
22     int A[5];
23
24     InputElements(A, 5);
25     PrintElements(A, 5);
26
27     return 0;
28 }
```

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 2.4. What is the output of the program?
2. Modify the program such that the array size is for 10 integer elements.
3. Implement a new function `void ReversedPrintElements(int A[], int n)`. This function will print the elements of the array in reversed sequence, i.e., from the last element down to the first element. Call this function inside `main()`.
4. Verify that an array–to–array assignment is not allowed by inserting `B = A;` immediately after the first `for` loop. What is the compilation error?

5. Write a function `int GetSum(int A[], int n)` that will compute and return the sum of the elements in array `A`.

6. Write a function `int CountPositive(int A[], int n)` that will count and return the value of the number of positive integer elements in array `A` with `n` elements.

7. Write a function `int CountNegative(int A[], int n)` that will count and return the value of the number of negative integer elements in array `A` with `n` elements.

8. Write a function `int Search(int A[], int n, int key)` that will search `key` in array `A`. If it is in the list, the function should return the index of the element, otherwise it should return a value of -1.

9. Write a function `void RotateRight(int A[], int n, int x)` which will rotate the elements of array `A` towards the right by `x` number of positions. Assume that `x > 0`.

10. Same as the previous problem except that rotation is towards the left. Implement function `void RotateLeft(int A[], int n, int x)`. Assume that `x > 0`.

11. Write a function `void CopyArray(int A[], int B[], int n)` that will copy the contents of array `A` to array `B`.

12. Write a function `void ReversedCopyArray(int A[], int B[], int n)` that will copy the contents of array `A` in reversed order to array `B`.

2.4 Relationship Between Arrays and Pointers

To a beginning programmer, the codes in Listing 2.4 present an illusion, i.e., it seems that what is being passed as parameter is the **whole** array `A`. As was explained above, this is really not true.

So what is it that is actually passed as parameter if it is not the array itself? The answer is: the **name** of the array. We quote, verbatim, the following from page 99 of (Kernighan & Ritchie, 1988)

> "...the name of an array is a synonym for the location of the initial element, ..."

Furthermore, on the same page, (Kernighan & Ritchie, 1988) wrote

> "When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable and so an array name parameter is a pointer, that is, a variable containing an address."

As indicated by the quotations above, arrays and pointers are very much related to each other. The name of the array corresponds to the address of the first element, i.e., the base address. We can easily verify this by printing the addresses of the array elements. We present two versions; the first one uses the address–of operator as shown in Listing 2.5. There is really nothing new about this program since we all know the meaning and how to use the `&` operator.

Listing 2.5: Display array element addresses using `&`

```
1  #include <stdio.h>
2  int main()
3  {
4      int A[5];
5      int i;
6      for (i = 0; i < 5; i++)
7          printf("Address of A[%d] is %p\n", i, &A[i]);
8
9      return 0;
10 }
```

The second version, shown in Listing 2.6, is more interesting! Recall that the name of the array is synonymous with the base address. We can therefore use the memory addressing scheme we learned in Chapter 1. To get the memory address of an array element, we simply need to add the base address with the element's index. In the case of array `A`, the address of element `i` is computed as `A + i`.

Listing 2.6: Display array element addresses using address arithmetic

```
1  #include <stdio.h>
2  int main()
3  {
4      int A[5];
5      int i;
6      for (i = 0; i < 5; i++)
7          printf("Address of A[%d] is %p\n", i, A + i);
8
9      return 0;
10 }
```

► Self–Directed Learning Activity ◄

1. Encode and run the program in Listing 2.5. Run the program more than once. Take note of the addresses of the elements. What do you notice?
2. Encode and run the program in Listings 2.6. Verify that the result of `A + i` is a memory address. Run the program more than once. Take note of the addresses of the elements.
3. For both programs, try changing the data type of the array to (a) `char`, (b) `float` and (c) `double`. Check again the addresses of each array element.

The concept presented in the previous discussion is very important, we highlight the equivalence:

```
&A[i] == A + i
```

Applying the dereference operation on both sides of the equality symbol yields:

```
*&A[i] == *(A + i)
```

Since `*` and `&` cancels each other, the final equivalence relationship is given by:

```
A[i] == *(A + i)
```

`A[i]` is the *array indexing notation* for accessing the `i`'th element of array `A`. The *pointer dereference notation* for achieving the same effect is `*(A + i)`. The following is a verbatim quote from page 99 of (Kernighan & Ritchie, 1988)

> "In evaluating `a[i]`, C converts it to `*(a + i)` immediately; the two forms are equivalent."

Most beginning programmers find the pointer notation difficult, so they prefer to use the array indexing notation for accessing elements. With lots of practice and more experience, a programmer will become proficient in both forms.

We illustrate in Listing 2.7 how to access array elements by dereferencing the element address `A + i`. The codes are reminiscent of what we learned in Chapter 1.

Listing 2.7: Access array elements via dereference operator

```
#include <stdio.h>
int main()
{
    int A[5];
    int i;

    /* initialize array elements */
    for (i = 0; i < 5; i++)
        *(A + i) = (i + 1) * 5;

    for (i = 0; i < 5; i++)
        printf("A[%d] = %d\n", i, *(A + i));

    return 0;
}
```

► Self–Directed Learning Activity ◄

1. Encode and run the program in Listing 2.7.
2. Refer back to the problems given in the previous sections. Convert all the programs you have written from an array indexing notation to a pointer dereference form. Verify that the semantics are the same by testing and running your programs.

2.5 Two–Dimensional Array

For brevity we will write 2D to mean two dimension. A 2D array is a collection of homogeneous elements organized in a set of rows and columns. The more common term for 2D array is *table*. In mathematics, the term *matrix* is also used to refer to a 2D array. If the number of rows is the same as the number of columns, the matrix is called a *square matrix*. An example of a square matrix is a 9x9 SUDOKU puzzle we encountered at the start of this chapter.[3]

Consider a 2x3 array of integer shown below:

50	19	48
35	64	70

In the C programming language, rows and columns are indexed starting from 0. Thus, the first and second rows are indexed as row 0 and row 1 respectively. The same is true for columns. Thus, the columns in the example above are numbered as column 0, column 1 and column 2. A pair of row and column indices are needed to refer to an array element. The value of the array element at row 1, column 2 is 70.

We show in the next example a 5x5 square matrix of characters; the single quotes were omitted for brevity. The character at row 2, column 3 is `'D'`. Can you find combinations of letters that form words in English (similar to the game Boggle or Word Factory)?

R	I	A	H	C
B	A	G	I	A
A	R	O	D	T
T	E	N	S	E
S	I	G	N	G

2.5.1 2D Array Declaration

The syntax for declaring a 2D array in the C programming language is as follows:

```
<data type> <name> [<row size>][column size]
```

[3]In English,the notation `9x9` is read as "9 by 9" where the first number is the number of rows and the second number is the number of columns.

where `data type` is the element's data type, `name` is the array's name, `row size` is the number of rows, `column size` is the number of columns, and that two pairs of square bracket means that the array is 2D.

The total number of elements in the array is equal to the product of `row size` with `column size`. For example, a 2D array made of 2 rows and 3 columns has 6 elements. A SUDOKU puzzle has 81 elements.

Just like a 1D array, a block of contiguous memory space for storing the 2D array elements will be allocated using static memory allocation.

Some example declarations are:

```
char Boggle[5][5];   /* 2D character array with 5 rows, 5 columns */
int Table[2][3];     /* 2D integer array with 3 rows, 2 columns */
int Sudoku[9][9];    /* for Sudoku puzzle of 9 rows, 9 columns */
float Matrix[4][3];  /* 2D float matrix 4 rows, 3 columns */
```

2.5.2 2D Array Element Definition

Elements of a 2D array can be defined as follows

```
<data type> <name> [<row size>][<column size>] = {<list of values>}
```

where the `<list of values>` is the set of values to be assigned to array elements in the order that they appeared. The values need to be separated by commas and placed inside a pair of curly brackets.

The following are example of 2D array element definitions. The `Table` array and `Boggle` definitions correspond to the 2D arrays graphically represented in Section 2.5.

```
int Table[2][3] = { {50, 19, 48}, {35, 64, 70} };

char Boggle[5][5] = { {'R', 'I', 'A', 'H', 'C'},
                      {'B', 'A', 'G', 'I', 'A'},
                      {'A', 'R', 'O', 'D', 'T'},
                      {'T', 'E', 'N', 'S', 'E'},
                      {'S', 'I', 'G', 'N', 'G'} };
```

2.5.3 Referencing an Element in a 2D Array

The syntax for referencing or accessing a particular array element in a 2D array is:

```
<name> [<row index>][<column index>]
```

where `row index` and `column index` are the indices of the element. The `row index` is an integer from `0` to `row size - 1`, while `column index` is from `0` to `column size - 1`. Examples of valid array references, based on the arrays declared above, are:

```
Boggle[0][0]    /* refers to element at row 0, column 0 */
Table[1][0]     /* refers to element at row 1, column 0 */
Sudoku[2][5]    /* refers to element at row 2, column 5 */
Sudoku[8][8]    /* refers to last element at row 8, column 8 */
Matrix[2][1]    /* refer to element at row 2, column 1 */
```

Examples of invalid array references are:

```
Boggle[-1][0]  /* semantic error: negative index */
Sudoku[0][9]   /* semantic error: column index is
                  more than column size - 1 */
Table[3.4][1]  /* syntax error: row index is not a whole number */
```

2.6 Mapping a 2D Array into the Primary Memory

We think of 2D arrays as two–dimensional in concept. It should be noted, however, that physically, it is mapped internally (i.e., stored internally) as a collection of 1D arrays into the primary memory. The C programming language in particular maps 2D array elements into the primary memory in *row–major order*.

Consider, for example, the 2D array defined with the name `M` below.

```
int M[2][3] = { {'A', 'B', 'C'}, {'X', 'Y', 'Z'} };
```

The 2D graphical representation of `M` is

'A'	'B'	'C'
'X'	'Y'	'Z'

In the softcopy of this document, we used red color as a visual cue to indicate the elements of row 0, while blue color indicates elements of row 1.

In the primary memory, `M` is mapped in row–major order. This means that the elements are ordered from row 0 to row 1 (in increasing row index). Within the same row, elements are ordered from column 0 to column 2 (in increasing column index). The corresponding graphical representation of the 2D array in row–major order is as follows:

'A'
'B'
'C'
'X'
'Y'
'Z'

More detailed information about array `M` with regards to the element ordering and the corresponding addresses are shown below.

Element	Address	Value
`M[0][0]`	`&M[0][0]`	'A'
`M[0][1]`	`&M[0][1]`	'B'
`M[0][2]`	`&M[0][2]`	'C'
`M[1][0]`	`&M[1][0]`	'X'
`M[1][1]`	`&M[1][1]`	'Y'
`M[1][2]`	`&M[1][2]`	'Z'

We use the codes in Listing 2.8 to demonstrate that 2D array elements are really mapped in row–major order. The program shows both brute force and nested `for` loop approach when working with 2D array elements.

Listing 2.8: Row–major order demo program

```
#include <stdio.h>
#include <stdio.h>
int main()
{
    int M[2][3] = { {'A', 'B', 'C'}, {'X', 'Y', 'Z'} };
    int i, j;

    printf("Brute force approach:\n");
    /* display row 0 element addresses */
    printf("&M[0][0] = %p\n", &M[0][0]);
    printf("&M[0][1] = %p\n", &M[0][1]);
    printf("&M[0][2] = %p\n", &M[0][2]);

   /* display row 1 element addresses */
   printf("&M[1][0] = %p\n", &M[1][0]);
   printf("&M[1][1] = %p\n", &M[1][1]);
   printf("&M[1][2] = %p\n", &M[1][2]);

   printf("\n");
   printf("Nested for loop approach:\n");
   for (i = 0; i < 2; i++)
      for (j = 0; j  < 3; j++)
          printf("&M[%d][%d] = %p\n", i, j, &M[i][j]);

   return 0;
}
```

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 2.8.
2. Modify the program by changing the number of rows from 2 to 4. Define the new elements using any value that you want.
3. Immediately after the nested `for` loop, insert the following codes:

   ```
   printf("\n");
   for (i = 0; i < 4; i++)
       printf("M + %d = %p\n", M + i); /* what does M + i represent? */
   ```

   Take note of the printed values. What do you think does the expression `M + i` represent?

4. Using the program from the previous problem, change the element data type from `int` to (a) `char`, (b) `float` and (c) `double`. Take note of the addresses of the elements.

Listing 2.9 shows a very simple example program incorporating what we learned so far: (i) 2D array definition, and (ii) how to reference array elements. It is very important to notice that a loop within a loop, i.e., a nested loop, is usually present when manipulating elements of a 2D array.

Listing 2.9: Accessing 2D array elements

```
#include <stdio.h>
int main()
{
    int Table[2][3] = { {50, 19, 48}, {35, 64, 70} };
    int row; /* row index */
    int col; /* column index */


    /* print elements of array named as Table */
    /* note that a double loop is commonly used when
       accessing elements of a 2D array */
    for (row = 0; row < 2; row++) {
        for (col = 0; col < 3; col++)
            printf("%d ", Table[row][col]);

        printf("\n");
    }
    return 0;
}
```

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 2.9.

2. Edit the program by changing the array `Table` definition to an array declaration. Thereafter, insert new codes that will initialize the elements of `Table` as indicated in the drawing below:

0	0	0
0	0	0

3. Do the same requirements as item 2 above, except that the initial values for `Table` should be set as follows:

10	20	30
40	50	60

4. Write a new program that will define a 2D array named `Boggle` following the example in Section 2.5.2. Print the elements. The first line of output should be the elements of row 0; next line of output should be the elements of row 1, and so on.

5. Write a new program that will declare a 2D array named as `MyTable` with 4 rows and 3 columns with double data type elements. Thereafter, input the elements of the `MyTable` using `scanf()`. Finally, output the elements of the array similar to the sequencing mentioned in the previous problem. Print 2 digits only after the decimal point.

2.6.1 2D Array and Parameter Passing

Similar to the discussion in "1D Array and Parameter Passing" in subsection 2.3.5, the `name` of the 2D array can be passed as function parameter.

We show how to do this in Listing 2.10. The function `void Out2DArray(int Table[][3])` prints the elements in row–major order. The formal parameter `int Table[][3]` indicates that the name `Table` is the base address of a 2D array. There is no need to specify any value between the first pair of square brackets, but it is necessary to put the column size in the second pair of square brackets.

Listing 2.10: 2D Array and parameter passing

```
1  #include <stdio.h>
2  void Output2DArray(int Table[][3])
3  {
4      int i;    /* row index */
5      int j;    /* column index */
6
7     for (i = 0; i < 2; i++) {
8          for (j = 0; j < 3; j++)
9              printf("%d ", Table[i][j]);
10
11         printf("\n");
12     }
13 }
```

```
14
15 int main()
16 {
17     int Table[2][3] = { {50, 19, 48}, {35, 64, 70} };
18
19     Output2DArray(Table);
20     return 0;
21 }
```

We modify the program by introducing a new function for input of the `Table` elements. The function prototype for this function is `void Input2DArray(int Table[][3])`. We pass the name of the array as parameter as shown in Listing 2.11. Notice that both input and output of array elements are done using row–major order.

Listing 2.11: Input array elements in another function

```
1  #include <stdio.h>
2
3  void Input2DArray(int Table[][3])
4  {
5     int i;    /* row index */
6     int j;    /* column index */
7
8     for (i = 0; i < 2; i++) {
9          for (j = 0; j < 3; j++)  {
10             printf("Input element %d, %d: ", i, j);
11             scanf("%d", &Table[i][j]);
12         }
13         printf("\n");
14     }
15 }
16
17 void Output2DArray(int Table[][3])
18 {
19     int i;    /* row index */
20     int j;    /* column index */
21
22    for (i = 0; i < 2; i++) {
23         for (j = 0; j < 3; j++)
24             printf("%d ", Table[i][j]);
25
26         printf("\n");
27     }
28 }
```

```
29
30 int main()
31 {
32     int Table[2][3];
33
34     Input2DArray(Table);
35     Output2DArray(Table);
36     return 0;
37 }
```

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listings 2.10 and 2.11.
2. Modify the program by considering a 2D array with more rows and columns. For example, change row size to 4 and column size to 5.
3. Modify the programs by changing the data type from `int` to `double`.

In the preceding example programs, all the array elements were referenced inside a nested `for` loop. Is it possible to access only the elements of a particular row or column? Listing 2.12 shows how this can be done. Function `void OutputRowElements(int Table[][3], int row)` references only the elements of a specified row – the value of which is passed as a parameter. In this case, there is no need to use a nested loop. Only one loop will be needed to generate the indices of the column numbers within a specified row.

Listing 2.12: Access elements of one row only

```
1  #include <stdio.h>
2
3  void OutputRowElements(int Table[][3], int row)
4  {
5     int col;   /* column index */
6
7     for (col = 0; col < 3; col++)
8         printf("%d ", Table[row][col]);
9
10    printf("\n");
11 }
12
```

```
13 int main()
14 {
15     int Table[2][3] = { {50, 19, 48}, {35, 64, 70} };
16     int row;
17
18     printf("Input row number of the elements you want to print: ");
19     scanf("%d", &row);
20
21     OutputRowElements(Table, row);
22     return 0;
23 }
```

► Self–Directed Learning Activity ◄

1. Encode and run the program in Listing 2.12. What is the output when `row = 0`? when `row = 1`? when `row` is more than `1`, for example, `row = 2`?

2. Write a new function `void OutputColumnElements(int Table[][3], int col)` which will output only the elements of a particular column from the first row to the last row. For example, the call `OutputColumnElements(Table, 1)` will print 19 followed by 64 using the `Table` definition in Listing 2.12.

3. Write a function `int GetRowSum(int Table[][3], int row)` that will compute and return the sum of the elements in a specified row. For example: `GetRowSum(Table, 1)` will return a value of `169` (computed as `35 + 64 + 70`).

4. Write a function `int GetColumnSum(int Table[][3], int col)` that will compute and return the sum of the elements in a specified column. For example: `GetColumnSum(Table, 2)` will return a value of `118` (computed as `48 + 70`).

2.7 Representative Problems

In this section we discuss representative problems which naturally requires 1D and 2D arrays.

2.7.1 Sorting

The *sorting* problem takes as input a list of numbers. Its output is a list of the same numbers but re–arranged into either increasing or decreasing order. For example, if the original list is {`40, 30, 10, 50, 20`}, the sorted list in increasing order is {`10, 20, 30, 40, 50`}, and the sorted list in decreasing order is {`50, 40, 30, 20, 10`}.

There are several sorting algorithms in the literature. Some of these are bubble sort, straight selection sort, insertion sort, merge sort, shell sort and quicksort. We will discuss one sorting algorithm only, specifically, straight selection sort.[4]

In an increasing order of elements, the first element should be the smallest element, and the last element is the largest. We use the following as an example of an initially unsorted array.

40	30	10	50	20

The algorithm is described below.

1. Assume, for the meantime, that `A[0]` is the "current" smallest value in the (unsorted) list. Determine which, among the remaining elements, is the element with the least value compared with `A[0]`. If there is such an element, swap it with `A[0]`.

   Using the sample array above, `A[0]` is 40. We find among the remaining elements the least value compared with `A[0]`. This value corresponds to 10 which has an index of 2. We then swap `A[0]` with `A[2]`. The resulting array after swapping is

10	30	40	50	20

[4] A more detailed discussion of sorting algorithms will be given in a subject called Data Structures and Algorithms.

where the element in red color is the element that should rightfully occupy the first position, while the element in blue was the original element occupying said position.

2. We then repeat the same logic as above for the remaining elements `A[1]` to `A[4]`. Let `A[1]` be the current smallest element. Among the remaining elements, the one with least value compared with `A[1]` is `A[4]`. We swap these two elements resulting to:

10	20	40	50	30

3. Continuing with this logic, the contents of the arrays will change as follows:

10	20	30	50	40

10	20	30	40	50

Notice that we repeated the same instructions for four times on five elements. In general, it takes a maximum of `n-1` iterations to sort `n` elements using straight selection sort.

We show in Listing 2.13 the implementation of the sorting algorithm.

Listing 2.13: Straight selection sort

```
1  #include <stdio.h>
2
3  void Swap (int *px, int *py)
4  {
5      int temp;
6
7      temp = *px;
8      *px = *py;
9      *py = temp;
10 }
11
12 void Sort(int A[], int n)
13 {
14     int i;
15     int j;
16     int min; /* index of the smallest value */
17
18     for (i = 0; i < n-1; i++) { /* note: n-1 steps only */
```

```
19          min = i;
20
21          for (j = i + 1; j < n; j++) {
22              if (A[min]  > A[j])
23                  min = j;
24          }
25          Swap(&A[i], &A[min]);
26      }
27  }
28
29  void Output1DArray(int A[], int n)
30  {
31      int i;
32      for (i = 0; i < n; i++)
33          printf("%d\n", A[i]);
34  }
35
36  int main()
37  {
38      int A[5] = {40, 20, 30, 50, 10};
39
40      printf("Array before sorting: \n");
41      Output1DArray(A, 5);
42
43      Sort(A, 5);  /* sort the array elements */
44      printf("\n");
45      printf("Array after sorting: \n");
46      Output1DArray(A, 5);
47
48      return 0;
49  }
```

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 2.13. Next, edit the program by increasing the size of the array from 5 to 10. Initialize the contents using an integer value that you want. Execute the program to see the results. Change the contents several more times and see the corresponding results.

2. Write a new program to implement a sort function that performs straight selection sorting resulting to a list that is in decreasing order.

3. Visit `http://people.cs.ubc.ca/~harrison/Java/sorting-demo.html` to

see a visualization/animation of different sorting algorithms including straight selection sort.

2.7.2 Matrix Algebra

Matrices (containing numeric values) are used in many problems such as in solving a system of simultaneous equation. There is even a specific area in mathematics called Matrix Algebra that studies matrices and operations on matrices.

We will describe simple matrix operations only. In the following discussions, let `s` represent a scalar value and `A`, `B` and `C` represent matrices.

The basic operations on matrices are:

1. Scalar multiplication: `B = s * A`

   This multiplies each element of `A` with a scalar value `s`. More specifically, `B[i][j] = s * A[i][j]`.

2. Matrix Addition: `C = A + B`

   This operation adds the elements of `A` with `B`, and stores the result into matrix `C`. The per–element relationship is given by `C[i][j] = A[i][j] + B[i][j]`. Note that the matrices should have the same operand.

Functions which implement these matrix operations are given below. Assume that arrays are 5x5.

```
void ScalarMultiplication(int B[][5], int A[][5], int s)
{
     int i, j;

     for (i = 0; i < 5; i++)
         for (j = 0; j < 5; j++)
              B[i][j] = s * A[i][j];
}

void MatrixAddition(int C[][5], int A[][5], int B[][5])
{
     int i, j;
```

```
        for (i = 0; i < 5; i++)
            for (j = 0; j < 5; j++)
                 C[i][j] = A[i][j] + B[i][j];
}
```

► **Self–Directed Learning Activity** ◄

1. Encode and incorporate the codes above in a program. Create your own `main()` function which will contain the declarations for the matrices and other variables. The `main()` function should call `ScalarMultiplication()` and `MatrixAddition()`. Print the contents of the resulting matrix.
2. Create a new function `void MatrixSubtraction(int C[][5], int A[][5], int B[][5])` that will compute `C = A - B`, i.e., the difference between matrix `A` and `B`.

2.8 Experiments for the Curious

Let us try some experiments. We will deliberately commit mistakes and see what will happen when we compile and run the codes.

1. Can we declare a 1D array with a negative size? For example, will the declaration `int A[-5];` be accepted by the compiler?
2. Can we declare a 1D array with a size of 0? For example, will `int A[0];` be accepted by the compiler? If this was accepted by your compiler, do the following: declare array `A` then assign 5 to `A`, i.e., `A[0] = 5;`. Compile and then run the program. Is there any run–time error? If you answered yes, what is the run–time error?
3. Can we define a 1D array with a size of 0? For example, will the definition `int A[0] = 5;` be accepted by the compiler?
4. Can we declare a 1D array with a size of 1? For example, will `int A[1];` be accepted by the compiler? Does it make sense to declare a 1D array of just one element?
5. Recall the that the name of the array is synonymous with the base address. Is it possible to increment the array's name in the function where it was declared? For example, will the following program work? Give a reason why or why not it will work.

```
int main()
{
    int A[5];

    A++;   /* will this work? */
    return 0;
}
```

6. Try next the following program. Will it work? Why or why not?

```
void Test(int A[], int n)
{
     A[0] = 5;
     A++;        /* will this work? */
     *A = 10;
}

int main()
{
     int A[5];

     printf("A[0] = %d\n", A[0]);
     printf("A[1] = %d\n", A[1]);
     return 0;
}
```

7. Is it possible to declare a 2D array with 1 row and several columns? For example, is the declaration `int A[1][3];` syntatically correct? Does it make sense to declare an array with just one row or just one column?

8. Assume an array with 3 rows and 5 columns. What do you think will happen if it was passed to a function which accepts fewer columns? Test this with the following function:

```
void Test(int M[][2])
{
     int i, j;
     for (i = 0; i < 3; i++)
         for (j = 0; j < 2; j++) {
             printf("%d ", M[i][j]);

          printf("\n");
     }
}
```

2.9 Chapter Summary

The key ideas presented in this chapter are summarized below:

- An array is a finite group of homogeneous elements.
- An array is characterized by four attributes, namely: name, size, element type and dimension.
- Memory space for arrays are allocated using static memory allocation.
- An array element is accessed by indicating its name and its correponding index. 2D arrays require two indices, namely the row index and column index.
- The name of the array corresponds to its base address. It can be passed as function parameter.
- Elements of a 1D array can also be accessed via pointer dereference.

Problems for Chapter 2

Problem 2.1. Write a function `int IsIncreasingOrder(int A[], int n)` which will return 1 if `A[i] < A[i+1]` for `i = 0` to `n-2`, otherwise it will return 0. Assume that elements are unique, i.e, no two elements have the same value.

Problem 2.2. Write a function `int CountOdd(int A[], int n)` which will count and return the number of odd values in array `A`.

Problem 2.3. Write a function `int Minimum(int A[], int n)` which will determine and return the smallest element in array `A`.

Problem 2.4. Write a function `int Maximum(int A[], int n)` which will determine and return the largest element in array `A`.

Problem 2.5. Write a function `int Sum(int A[], int n)` which will determine and return the sum of all the elements in array `A`.

Problem 2.6. Write a function `float Average(int A[], int n)` which will determine and return the average of the elements in array `A`. Note that the function is of type `float`.

Problem 2.7 Write a function `int CountUpper(char S[], int n)` which will count the number of upper case letters in array `S`. For example, assume an array `S` defined as follows: `char S[7] = {'C', 'o', 'M', 'p', 'r', 'o', '2'};` A call to `CountUpperCase(S, 7)` will return 2 since there are two upper case letters, namely, 'C' and 'M'.

Problem 2.8 Write a function `void ConvertUpper(char S[], int n)` which will convert all letters in the array to upper case. For example, `ConverUpper(S, 7)` using the array `S` defined in the previous problem will result into a modified array `S` containing 'C', 'O', 'M', 'P', 'R', 'O', '2'.

Problem 2.9. Write a function `void MaxCopy(int C[], int A[], int B[], int n)`. Assume that arrays `A` and `B` contain values. Determine which of the two values between `A[i]` and `B[i]` is higher. The higher value is assigned to `C[i]` for `i = 0` to `n-1`.

Problem 2.10. An identity matrix is a square matrix whose main diagonal elements are all 1, and the remaining elements are all 0. Write a function `int IsIdentityMatrix(int M[][5])` that will return 1 if the 5x5 matrix `M` is an identity matrix; otherwise, it should return 0.

Problem 2.11 Find out (for example, using Google search) what is the *transpose* of a matrix. Write a function `void TransposeMatrix(int M[][5], int T[][5])` that will compute and store the transpose of a given matrix `M` to matrix `T`.

Problem 2.12 Find out how to compute the product of two matrices, say `A` and `B`. Please see the following site: `http://people.hofstra.edu/Stefan_Waner/realWorld/tutorialsf1/frames3_2.html`. Implement a function for multiplying matrices `A` with `B` with the result stored in matrix `C`. The function prototype is `void MatrixMultiply(int C[][5], int A[][5], int B[][5]);`.

Problem 2.13. You were familiarized with row–major order in this chapter. Find out what is column–major order. Explain in not more than two sentences what is column–major order.

Problem 2.14. Assume a 2D array `M` of 3 rows and 5 columns. Write a function `void PrintColumnMajorOrder(int M[][5])` which will print the values of the array in column major order. Print one number only per line of output.

References

Consult the following resources for more information.

1. comp.lang.c Frequently Asked Questions. `http://c-faq.com/`

2. Kernighan, B. and Ritchie, D. (1998). *The C Programming Language, 2nd Edition*. Prentice Hall.

Chapter 3

Strings

3.1 Motivation

The document that you are reading right now is a collection of letters and other characters (such as period, left parentheses, question mark, and brackets). Letters when combined properly form words; words form phrases and sentences; sentences form paragraphs.

Text editors and wordprocessors are software that we use for editing, storing, and manipulating characters. Web browsers, email utilities, messengers, chat programs are just some of the other software tools that we use to create, store, retrieve and view information encoded as characters.

Aside from numeric data, a programmer will have to learn how to represent, store and manipulate a collection of characters commonly referred to as *string*.

3.2 What is a string?

A *string* is a combination of zero or more characters from a given character set.[1] For example, `"Hello world!"` is a string constant.

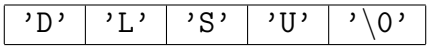
A string constant is denoted in the C programming language by writing a sequence of characters enclosed within a pair of double quotes. The following are examples of string constants:

[1] We will use the ASCII characters.

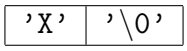
```
"ABC"
"DLSU"
"COMPRO2"
"Hello world!\n"
"http://www.abc_def_123.org"
"X"
" "
""
```

3.3 String Representation

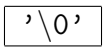
A string is internally represented as a sequence of characters and stored in contiguous bytes of memory space. For example, the string `"DLSU"` is stored internally as follows



The string `"X"` is stored internally as



and the empty string `""` is stored as



The last character in a string is always a *null byte* denoted by `'\0'` (backslash character followed by zero). The null byte has an ASCII code of 0. It is a hidden character so it will not be visible on the display screen or printout. Notice that the `'\0'` is not explicitly written as part of the string constant. It will be automatically appended in behalf of the programmer or user.

The *length* of a string is the number of characters in the string *excluding* the null byte. For example, the length of the string `"COMPRO2"` is 7. Notice that the string `" "` contains a space character and its length is 1. It is different from the string `""` which is used to denote an *empty* string. An empty string has a length of 0. It should also be noted that the string constant `"X"` is different from the character constant `'X'`.

3.4 Memory Allocation for Strings

While there is an `int` keyword corresponding to an integer data type, there is no `str` keyword because the C programming language does not explicitly provide a string data type. Instead, it uses contiguous bytes for storing the individual elements of a string.

The storage space can be allocated by declaring a one–dimensional array of characters (static memory allocation) or via `malloc()` (dynamic memory allocation). We will limit the discussion in this chapter to character arrays for simplicity reason.

Let us assume for example that we would like to allocate memory space for storing at most 3 characters *excluding* the null byte. We can achieve this by declaring a character array as shown in the following code:

```
char str[4];  /* static mem. allocation */
```

Notice that the total size is 4 bytes (not 3) in order to store the null byte. In general, if the length of the string is n then we should allocate memory space for at least $n + 1$ number of bytes.

3.5 String Initialization

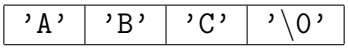
In Chapter 2, we learned how to define an array of characters. For example, we can define the value of the character array `str` as `"ABC"` by

```
char str[4] = {'A', 'B', 'C', '\0'}; /* variable definition */
```

An equivalent but more compact initialization can be achieved as shown in the following definition:

```
char str[4] = "ABC";  /* variable definition */
```

This form of initialization is the preferred method and the one that we will use from hereon. Its internal representation is as follows.



If the array size is more than the length of the string constant, the unused bytes will remain uninitialized, i.e., they contain garbage values. For example, the definition:

```
char str[10] = "ABC";  /* variable definition */
```

is represented internally as:

'A'	'B'	'C'	'\0'	???	???	???	???	???	???

where we used `???` to mean garbage value.

It should be noted that the use of the assignment operator `=` with a string constant is valid only if done as a variable definition.[2] The following assignment operation is not allowed and will cause a syntax error.

```
char str[4];  /* note: this is a variable declaration only */

str = "ABC";  /* syntax error! */
```

▶ Self–Directed Learning Activity ◀

Encode and compile the following program.

```
int main()
{
    char str[4] = "ABC";
    char name[5];

    return 0;
}
```

1. Decrease the size of array `str` from 4 to 2. Compile the program again. Is there any compilation error? If yes, what is the nature of the error? What can you conclude based on this experiment?

2. Change the size of array `str` back to 4. Insert the assigment statement `name = "JUAN";` just before the `return` statement. Compile the program. Is there any compilation error? If yes, what is the nature of the error?

[2]Recall that variable declaration does not involve an initialization unlike a variable definition.

3.6 String I/O with scanf() and printf()

The pre–defined functions `printf()` and `scanf()` have already been introduced in COMPRO1. They are also used, respectively, for output and input of strings with `"%s"` as formatting symbol.

Listing 3.1 shows an example of string I/O with `printf()` and `scanf()`. Notice in particular that it is incorrect to put an address–of operator (ampersand) before `str` in the `scanf()`. Recall that the name of the array is synonymous to the address of the first element.

Listing 3.1: `printf()` and `scanf()` with strings

```
1  #include <stdio.h>
2  int main()
3  {
4      char prompt[30] = "Input a word (max. of 10 characters): ";
5      char str[11];
6
7      printf("%s", prompt);
8      scanf("%s", str);
9      printf("You entered %s\n", str);
10
11     return 0;
12 }
```

► Self–Directed Learning Activity ◄

1. Encode and run the program in Listing 3.1. Try the following as inputs (do not include a pair of double quotes):

   - `X`
   - `test`
   - `abc@def`
   - `COMPRO2`
   - `VeryLongWord`
   - `brown fox`

   Note that there is no space in the fifth item, but there is a space in the last item. What are the corresponding results?

Try other inputs. Try it with shorter than 10 characters, with exactly 10 characters, with more than 10 characters and with inputs that include spaces.

(a) What happens when the number of characters is more than the size of array `str`?

(b) What happens when the user inputs something that has a space in between?

2. A run–time error occurs if the user inputs characters more than the size of the character array. To limit the number of characters accepted by the `scanf()` function, specify the limit as a positive integer between `%` and `s`. For example, to limit the maximum number of input characters to 10, the format should be `"%10s"`. Modify and test the program above to verify that this is true.

3. Modify the program above by putting an ampersand immediately before variable `str` in `scanf()`. Compile and run the program. What is the result? What is the cause of the error?

4. Hangman is a game that allows the player to guess the letters in a word, i.e. a string. Try playing a Flash Hangman game at `http://www.manythings.org/hmf/`. How would you implement your own Hangman game in C? (do not think of the graphical user interface at this point in time).

3.7 String and `char *`

The individual elements of the array defined, for example, as

```
char str[4] = "ABC";
```

can be accessed using array indexing notation. The following equality relationship hold.

```
str[0] == 'A'
str[1] == 'B'
str[2] == 'C'
str[3] ==  '\0'
```

The address of each byte storing a character value has a data type of `char *`. Moreover, in the C programming languge, the name of the array is synonymous with the address of the first array element. Thus, `str[i] == *(str + i)`. This means that the following equality relationship also hold.

```
*(str + 0) == 'A'
*(str + 1) == 'B'
*(str + 2) == 'C'
*(str + 3) ==  '\0'
```

3.8 String Manipulation Functions

There are several pre–defined functions for manipulating strings. Their function prototypes can be found in the header file `string.h`. In this document, we will learn four of the most commonly used string manipulation functions.

The function prototype that we will encounter in the following subsections pass the base address (i.e., address of the first byte in a character array) as parameter. In the actual function call, we simply need to supply the name of the array.

3.8.1 Determining the Length of a String

The `strlen()` function is used to determine the length of a string.[3] Its function prototype is as follows:

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

Listing 3.2 shows an example on how to use the `strlen()` function.

Listing 3.2: Example program illustrating `strlen()`

```
1 #include <stdio.h>
2 #include <string.h>  /* don't forget to include this file */
3
4 int main()
5 {
```

[3]Remember that the length of the string does not include the null byte.

```
6       char str[4] = "ABC";
7       char name[21];
8
9       printf("The length of str is %d\n\n", strlen(str));
10      printf("Input your name (max. of 20 chars.): ");
11      scanf("%20s", name);
12      printf("The length of your name is %d characters.\n",
13              strlen(name));
14
15      return 0;
16  }
```

Encode and run the program to see how it works. Experiment by trying strings of different lengths.

Listing 3.3 shows an example program that asks the user to input a string. Thereafter, it prints the input string one character per line.

Listing 3.3: Another example illustrating `strlen()`

```
1   #include <stdio.h>
2   #include <string.h>  /* don't forget to include this file */
3
4   int main()
5   {
6       char data[21];
7       int i;
8       int len;
9
10      printf("Input a string (max. of 20 chars.): ");
11      scanf("%20s", data);
12
13      len = strlen(data);
14      for (i = 0; i < len; i++)
15          printf("data[%d] = %c\n", i, data[i]);
16
17      return 0;
18  }
```

3.8.2 Copying a String

The `strcpy()` function is used to copy the value of a string (including the null byte) from a source to a destination variable. Its function prototype is shown below.

```
char *strcpy(char *str1, const char *str2)
```

Here, `str2` is the pointer to the source string and `str1` is the pointer to the destination. The function returns the pointer to the source string `str1`. In actual programming practice, the return value is ignored.

We already know that the assignment statement `str = "ABC";` is incorrect. The correct initialization is done via `strcpy(str, "ABC");` as shown in Listing 3.4. Note that we can simply ignore the function's return value.

Listing 3.4: Example program illustrating `strcpy()`

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6      char source[8] = "COMPRO2";
7      char str[4];
8      char name[21];
9      char destination[8];
10
11      /* copy a string constant */
12     strcpy(str, "ABC");   /* str = "ABC"; is incorrect! */
13     printf("The value of str = %s.\n", str);
14
15     /* copy a string constant with spaces in between */
16     strcpy(name, "Juan dela Cruz");
17     printf("The value of name = %s.\n", name);
18
19     /* copy the value of a string variable */
20     strcpy(destination, source);
21     printf("The value of destination = %s.\n", destination);
22
23     return 0;
24 }
```

3.8.3 Concatenating Strings

The `strcat()` function is used to concatenate two strings. Its function prototype is given below.

```
char *strcat(char *str1, const char *str2)
```

Here `str1` and `str2` are pointers to the first and second strings respectively. Characters from the second string are copied onto the first string starting at the memory space associated with the first string's null byte. We should ensure that the size of `str1` is big enough to accomodate its original characters and the characters from `str2`; otherwise, the result is undefined. The second string remains the same after calling `strcat()`.

Listing 3.5 shows an example program how to use `strcat()`.

Listing 3.5: Example program illustrating `strcat()`

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6      char str1[4] = "ABC";
7      char str2[4] = "DEF";
8      char string[20];
9
10     /* initialize as empty string */
11     strcpy(string, "");
12
13     /* concatenate empty string "" and "Hello" */
14     strcat(string, "Hello");
15     printf("string = %s\n", string);
16
17     strcpy(string, "");
18     strcat(string, str1);
19     strcat(string, " "); /* append space */
20     strcat(string, str2);
21     printf("string = %s\n", string);
22
23     return 0;
24 }
```

3.8.4 Comparing Strings

The `strcmp()` function is used to compare two strings lexicographically. It was made available because we cannot use the relational operators such as `==`, `>` and `<` to compare strings. Its function prototype is given below.

```
int strcmp(const char *str1, const char *str2)
```

There are three possible return values, namely:

- 0 is returned when the strings are equal

  For example, `strcmp("ABC", "ABC")` returns 0.

- -1 is returned if the first string is less than the second string

  For example, `strcmp("ABC", "XYZ")` returns -1.

- 1 is returned if the first string is greater than the second string

  For example, `strcmp("XYZ", "ABC")` returns 1.

Note that the strings need not be of the same length. For example, the function call `strcmp("Hey", "Jude")` returns -1.

A example program illustrating how to use `strcmp()` is shown in Listing 3.6.

Listing 3.6: Example program illustrating `strcmp()`

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6      char str1[4] = "ABC";
7      char str2[4] = "XYZ";
8
9      /* "ABC" is equal to "ABC", result is 0 */
10     printf("%d\n", strcmp(str1, "ABC"));
11
12     /* "ABC" is less than "XYZ", result is -1 */
13     printf("%d\n", strcmp(str1, str2));
14
15     /* "XYZ" is greater than "ABC", result is 1 */
```

```
16      printf("%d\n", strcmp(str2, str1));
17
18      /* "ABC" is less than "X", result is -1 */
19      printf("%d\n", strcmp("ABC", "X"));
20
21      /* "aBC" is greater than "ABC", result is 1 */
22      printf("%d\n", strcmp("aBC", "ABC"));
23
24      return 0;
25  }
```

3.9 Strings and `typedef`

The C programming language has a keyword `typedef` for creating a synonym or alias of a specified data type name. The syntax for declaring an alias is

```
typedef <type name> <alias>;
```

Once declared, the alias can be used in place of the original type name.

The following example declares an alias for an `int` type called `Boolean`. Thereafter, the `Boolean` alias is used to declare `flag` as a variable that will be limited by the programmer to a value of either `FALSE` or `TRUE`.

```
#define FALSE (0)
#define TRUE (!(FALSE))

typedef int Boolean;

int main()
{
    Boolean flag = FALSE;

    /* -- other statements follow --*/
}
```

The `typedef` is not limited to the basic data types. It can be used together with pointer data type, with array and as well as `struct` data type.[4]

[4] `struct` data type will be discussed in Chapter 4.

We have already mentioned the fact that the C programming language does not have a data type for string. An artificial way to support a string data type is to use `typedef` with a character array as shown in the following example code.

```
typedef char String[51];

void test(String str)
{
     /*-- some statements here --*/
}

int main()
{
     String sentence = "Hello world!";
     String greetings;

     test(sentence);

      /* -- other statements follow --*/

     return 0;
}
```

The program defines `String` as a global alias for a character array of size 51. Thereafter, `sentence` and `greetings` were defined and declared as variables of "type" `String` respectively in the `main()` function. The alias can also be used as parameter type as shown in the `test()` function definition.

In actual programming practice, it is better if we specify the maximum size for the string as part of the alias name. In the example program shown in Listing 3.7, the alias `String20` was defined and later used as the data type for variables `lastname` and `firstname`. The number 20 indicated as part of the alias name will help the programmer remember that at most 20 characters (excluding the null byte) can be stored in the associated variable. Notice that the actual size of the character array in the `typedef` is 20 + 1. The last byte ensures that there is a space for the null byte in case all the first 20 bytes are used for non–zero characters.

Listing 3.7: Example program illustrating `typedef`

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

typedef char String20[21];
typedef char String50[51];

void Input(String50 prompt, String20 str)
{
     printf("%s", prompt);
     scanf("%20s", str);
}

void Greet(String20 lastname, String20 firstname)
{
     String50 greet;

     strcpy(greet, "Hello ");  /* note space after '0' */;
     strcat(greet, firstname);
     strcat(greet, " ");
     strcat(greet, lastname);
     strcat(greet, "! How are you today?");

     printf("%s\n", greet);
}

int main()
{
    String20 lastname;
    String20 firstname;

    Input("Enter your last name: ", lastname);
    Input("Enter your first name: ", firstname);
    Greet(lastname, firstname);

    return 0;
}
```

3.10 Array of Strings

The need to represent, store and manipulate a collection of strings arise in several programming problems. For example, how can we store the 32 keywords in the ANSI C programming language? One way to do this is to declare 32 string variables each with a distinct name, and initialize them with the name of the C keyword. For example:

```
typedef char String10[11];

String10 keyword0;
String10 keyword1;
      :
      :  /* more variable declarations */
      :
String10 keyword31;
```

This is a brute force approach and should be avoided in actual practice.

The recommended practice is to declare an array of strings. Listing 3.8 shows an example of how this is done. We first declare an alias for a string, and thereafter, we declare (or define) an array of string.

Listing 3.8: Array of strings example program 1

```
1  #include <stdio.h>
2
3  typedef char String10[11];
4
5  int main()
6  {
7      /* keywords is declared/defined as an array of String10.
8       We show the initialization for the 1st 5 keywords only. */
9      String10 keywords[32] = {"auto", "break", "case",
10                               "char", "const"};
11     int i;
12
13     printf("The C keywords are:\n\n");
14     for (i = 0; i < 5; i++)
15         printf("%s\n", keywords[i]);
16
17     return 0;
18 }
```

In this example, variable `keywords[]` is an array of size 32. Each element of the array, i.e., `keywords[i]` is of type `String10`.

Listing 3.9 shows another example. The alias `String10` is first declared. Thereafter, it is used to declare variable `friends[]` as an array of size 5 with each element of type `String10` in `main()`. The program also demonstrates how to pass the name of the array of strings as function parameter as shown in `InputNicknames()` and `PrintNicknames()` functions.

Listing 3.9: Array of strings example program 2

```
1  #include <stdio.h>
2
3  typedef char String10[11];
4
5  void InputNicknames(String10 friends[], int n)
6  {
7       int i;
8
9       for (i = 0; i < n; i++) {
10           printf("Input the nickname of your friend: ");
11           scanf("%s", friends[i]);
12      }
13 }
14
15 void PrintNicknames(String10 friends[], int n)
16 {
17     int i;
18
19     printf("\n");
20     printf("The nicknames of your friends are:\n");
21     for (i = 0; i < n; i++) {
22          printf("%s\n", friends[i]);
23     }
24 }
25
26 int main()
27 {
28     String10 friends[5];
29
30     InputNicknames(friends, 5);
31     PrintNicknames(friends, 5);
32
33     return 0;
34 }
```

3.11 Chapter Summary

The key ideas presented in this chapter are summarized below:

- A string is a combination of zero or more characters from a given character set.
- A string constant is specified by enclosing a group of characters enclosed within a pair of double quotes.
- A null byte indicates the end of a string. It is used by string manipulation functions such as `strlen()` and `strcat()` to determine the last character within the string.
- The data type associated with the address of each string element is `char *`.
- Memory space for a string can be allocated as a one dimensional array of characters.
- `printf()` can be used to output a string with `"%s"` as format.
- `scanf()` can be used to input the value of a string with `"%s"` as format. The maximum number of characters in the input can be indicated by putting the limiting value between `"%"` and `"s"`.
- Pre–defined functions for string manipulations exist with their function prototypes declared in the file `stdio.h`.
- `typedef` can be used to declare an alias for a character array.
- An array of strings can be represented, stored and manipulated by first declaring an alias for the character array, and then use the said alias as a data type for declaring an array of strings.

Problems for Chapter 3

Problem 3.1. Assume the following declarations and definitions:

```
typedef char String7[8];
typedef char String30[31];

String7 subject1 = "COMPRO2";
String7 subject2 = "FORMDEV";
String30 sentence;
```

What is output corresponding to the following? Items are independent from each other.

```
a.  printf("%d\n", strlen("Hello World!"));
b.  printf("%d\n", strlen("X"));
c.  printf("%d\n", strlen("X\n"));
d.  printf("%d\n", strlen(" "));
e.  printf("%d\n", strlen(""));
f.  printf("%d\n", strlen(strcpy(sentence, "Nanja kore???")));
g.  printf("%d\n", strlen(strcat(strcpy(sentence, "A*B*"), "C*D*F")));
h.  printf("%d\n", strcmp(subject1, subject2));
i.  printf("%d\ns, strcmp(subject2, subject1));
j.  printf("%d\n", strcmp("formdev", subject1));
k.  printf("%d\n", strcmp(subject2, "compro2"));
l.  printf("%d\n", strcmp(subject2, "formdev"));
```

Problem 3.2. Implement `void PrintReversed(char *str);` which will output the string in reversed order. For example, the function call `PrintReversed("ABC");` will output `"CBA"`.

Problem 3.3. Implement `int IsPalindrome(char *str);` which will return a 1 if the string parameter is a palindrome otherwise it returns a 0. A palindrome is a word or phrase that when read backwards (or in reversed order) produces the same word or phrase. For example, the word `"ROTOR"` is a palindome. The phrase `"STAR RATS"` is a palindrome. The word `"GOOD CAT"` is not a palindrome.

Problem 3.4. Implement `char *Capitalize(char *str);` which will capitalize all lower case letters in the string. The functions returns the pointer to the first byte of the modified string. For example, let `str1` and `str2` be strings. Also, let `str1` contain `"Hello World!"`. The function call `strcpy(str2, Capitalize(str1));` will copy `"HELLO WORLD!"` as the value of variable `str2`.

Problem 3.5. Implement a function `int GetPassword(char *password)` that will do the following in sequence:

1. Ask the user to input a string which will be stored into a local variable named `str` (maximum of 20 characters).
2. Check if `str` is equivalent to `password`. If it is, the function returns a 1. (Note: user entered the correct password.).
3. If they are not equivalent, i.e., the password is incorrect, repeat from step 1. The user is given three chances to enter the correct password. If after three tries, the correct password was not supplied, the function will return a value of 0.

Problem 3.6. The program in Listing 3.8 initializes the array of strings named as `keywords[]` with the first 5 C keywords only. Modify the program by supplying the remaining 27 keywords; see `http://www.cprogrammingreference.com/Tutorials/Basic_Tutorials/ckeywords_home.html` for the complete list. Thereafter, modify the `for` loop such that all the 32 keywords will be displayed on the screen.

Problem 3.7. Refer to Listing 3.9. The task in this problem is to implement the function `void SortNicknames(String10 friends[], int n)` which will sort the `friends[]` array alphabetically (in increasing order). Use the straight selection sorting algorithm discussed in Chapter 2. Test the function by calling it inside `main()` immediately before the call to `PrintNicknames()` function.

Problem 3.8. Create and implement your own algorithms for the following:

- String length, function prototype is: `int mystrlen(char *str);`
- String copy, function prototype is: `char *mystrcpy(char *str1, char *str2);`
- String concatenation, function prototype is: `char *mystrcat(char *str1, char *str2);`
- String comparison, function prototype is: `int mystrcmp(char *str1, char *str2);`

Chapter 4

Structures

4.1 Motivation

Many problems and applications in computing require representation and manipulation of data that are made up of a group of elements. These elements may be of the same data type (i.e., homogeneous) or of different data types (i.e., heterogeneous).

Consider for example how we can represent date. Although we usually abstract date as a single entity, it is actually a group of simple elements which include month, day and year. These elements can be declared as variables of type `int` as shown in the following code snippet:

```
int month;
int day;
int year;
```

The name (of a person) is another object that we abstract as a single entity. We can represent it as a group of elements as shown in the following:

```
char firstname[20];
char middlename[20];
char lastname[20];
```

For trivial programming problems that deal with just one name or just one date, the above representation would be sufficient. However, what if there is a

need to maintain several names and dates? Institutions such as banks, hospitals, hotels collect and store information about their clients (names and birthdates) in a database. Such databases will normally store hundreds or thousands of client information. The simple name and date representations in the code snippets above will not be appropriate.

We will learn in this chapter how to represent, store and manipulate entities that are composed of an aggregate of data values. In C programming language, these are referred to as *structures*.

4.2 What is a Structure?

Simply stated, a *structure* is a group of elements. Unlike arrays, however, the elements of a structure can be heterogeneous, i.e., of different data types. In the C programming language, an element of a structure is referred to as *member*. A member of a structure can be of a simple data type (`char`, `int`, `float`, `double`), a pointer data type, an array or even another structure data type.

The word structure may sound technical but we can easily understand the concept if we associate it with the word *record* – a name used in everyday language. Here are some examples of structures:

1. A mobile phone book directory is made up of several phone book records. Each phone book record (structure) has two members, namely number and name. Number is a numeric value while name is made up of several characters (character array).

2. Consider next an email or a social networking account. We can think of an account as a structure with the following as members: login name and password.

3. Date, for example 12/25/2009 (Christmas of 2009), as noted in the previous section is actually a structure made up of three members namely: month, day, and year. The month member can be represented as a string (for example, "December") or as an integer (for example, 12).

A structure is governed by a parent–child relationship which can be visualized as shown in Figure 4.1. The parent is the structure, and the children are the structure's members. The corresponding diagrams for the sample structures mentioned above are shown in Figure 4.2.

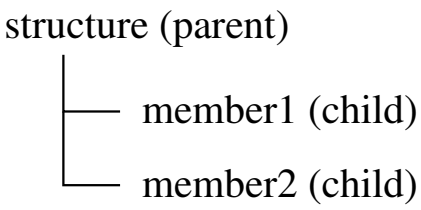


Figure 4.1: Graphical representation of a structure and its members

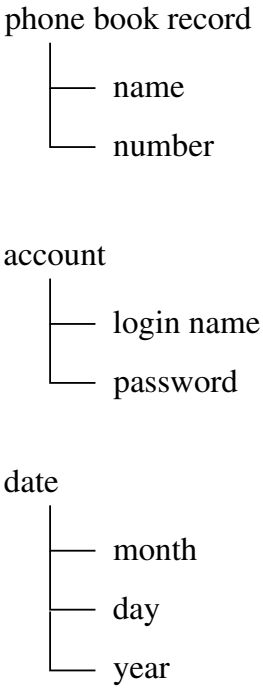


Figure 4.2: Graphical representations of structures in the previous example

4.3 `struct` Type and Structure Variable

A structure variable is a variable whose data type is a *structure type*. C uses `struct` as keyword to denote a structure type.[1]

The syntax for declaring a `struct` type and structure variable is

```
struct [tag-name] {
   <data type> <member-name>;
   <data type> <member-name>;
               :
               :
   <data type> <member-name>;
} [structure-var-name];
```

There are four different variations by which the syntax can be applied.

Variation #1 - Unnamed struct without an instance. The simplest variation contains only the `struct` keyword and a list of member declarations inside a pair of curly brackets. An example declaration of a `struct` type made up of four members representing the four basic data types is shown below:

```
struct {
   char ch;
   int i;
   float f;
   double d;
};
```

Try to compile a program that has a similar declaration as in above. Notice that `gcc` reports a ‘‘`unnamed struct/union that defines no instances`’’ warning message. The “unnamed struct” part of the warning means that the structure type does not have a tag–name. On the other hand, the “defines no instances” portion means that a structure variable of such structure type was not declared.

Variation 1 is by itself not useful. In a latter section, we will see how to use it together with `typedef`.

[1] Note that `struct` is a *user–defined data type*, i.e., it is the programmer who specifies the actual details about the type.

Variation #2 - Named struct without an instance. When a tag–name is added to variation 1, we will have the 2nd variation. For example:

```
struct sampleTag {
   char ch;
   int i;
   float f;
   double d;
};
```

This declaration specifies that the name of the `struct` type is `sampleTag`. Try to compile a program that has a declaration similar to the example above. Notice that the warning "unnamed struct ..." will no longer appear because the `struct` has been named.

Variation 2 is prevalent in actual programming practice.

Variation #3 - Unnamed struct with an instance. Adding a struct–var–name to variation 1 results into the 3rd variation. For example:

```
struct {
   char ch;
   int i;
   float f;
   double d;
} x;
```

This variation declares (i) an unnamed `struct` type, and (ii) a structure variable, i.e., an instance of the `struct` type named as `x`.

What happens if we want to declare multiple instances? This can be done simply by specifying the name of additional instances separated by a comma. For example, three instances named `x`, `y` and `z` are declared in the following code.

```
struct {
   char ch;
   int i;
   float f;
   double d;
} x, y, z;
```

Variation #4 - Named struct with an instance. The fourth variation has a tag–name as well as a struct–var–name. For example:

```
struct sampleTag {
   char ch;
   int i;
   float f;
   double d;
} x;
```

This variations declares both (i) `struct` type with a tag–name of `sampleTag` and (ii) an instance of the `struct` type, i.e., a structure variable, named as `x`. Similar to variation 3, several instances maybe declared by giving the names separated by comma.

Once a named `struct` type is made, as in the case of Variations 2 and 4, instances of such a type can be declared in subsequent codes. The tag–name serves as a substitute for the list of member declarations. The following shows how this is done.

```
/* struct type declaration from Variation 2 */
struct sampleTag {
    char ch;
    int i;
    float f;
    double d;
};

/* declare one instance */
struct sampleTag x0;

/* declare two more instances on separate lines*/
struct sampleTag x1;
struct sampleTag x2;

/* declare three more instances on the same line */
struct sampleTag x3, x4, x5;
```

In C, the declaration of `struct sampleTag` is also referred to as *structure template*. It is a common practice among C programmers to do a two–step declaration similar to the preceding example above. The steps are:

Step 1. Following variation 2 - declare the structure template.

Step 2. Declare the instance, i.e., structure variable, with the structure template as its data type.

Let us apply what we learned by declaring the structures mentioned in Section 4.2. The corresponding C declarations are shown in Listing 4.1. We included the `<string.h>` header file for string related operations which will be needed to manipulate character array structure members such as `name`, and `password`.

Notice that we declared the structure templates outside of the `main()` function. The reason for this will be explained when we go to the discussion of functions and structures in the latter sections.

Listing 4.1: Example `struct` type and instance declarations

```
1  #include <stdio.h>
2  #include <string.h>  /* for string related functions */
3
4  /* first: declare the structure templates */
5  struct phoneTag {
6     int number;
7     char name[30];
8  };
9
10 struct accountTag {
11    char login_name[7];
12    char password[20];
13 };
14
15 struct dateTag {
16    int month;
17    int day;
18    int year;
19 };
20
21 int main()
22 {
23       /* next: declare the instances */
24       struct phoneTag landline;
25
26       struct accountTag email_account;
```

```
27      struct accountTag facebook_account;
28
29      struct dateTag today;
30      struct dateTag birthdate;
31
32      /*--- codes to manipulate structures follow --- */
33      return 0;
34  }
```

▶ Self–Directed Learning Activity ◀

1. What do you think is the minimum possible number of members in a C `struct`? Verify your answer by writing and compiling a sample C declaration.

2. Encode the examples mentioned in pages 104 to 106, i.e., the declarations in Variations 1 to 4. Note that you have to declare them inside a function such as `main()`. Compile and test the codes. Take special note of the compiler warning for Variation 1 declaration.

3. Encode and compile the program in Listing 4.1.

4. Think of two or more structures. Thereafter, add your own declarations using the program you encoded in the previous item. Compile your program to see if your declarations are correct.

4.4 Operations on Structures

There are only three possible operations that can be performed on a structure variable, namely:

1. access its members

2. assign a structure to another structure variable of the same type

3. get the memory address of the structure variable via the "address–of" `&` operator

We will explain the details of these operations in the following sections.

4.5 Accessing a Member of a Structure

Any member of a structure variable can be accessed by using the *structure member operator* which is denoted by a dot symbol. The syntax is:

```
<struct-var-name>.<member-name>
```

The following codes show how to initialize, and then print the values of the members of structure variable `birthdate` (refer back to Listing 4.1):

```
/* initialize birthdate members */
birthdate.month = 12;
birthdate.day = 25;
birthdate.year = 2009;

/* print birthdate members */
printf("Birthdate is %d/%d/%d\n",
        birthdate.month, birthday.day, birthdate.year);
```

Insert these codes after the comment ("codes to manipulate structures follow") in Listing 4.1 and verify that it indeed works.

We can input the values of individual members as well using `scanf()` as shown in the following example:

```
/* input today members */
printf("Input today's month day and year: ");
scanf("%d %d %d", &today.month, &today.day, &today.year);

/* print today members */
printf("today is %d/%d/%d\n",
        today.month, today.day, today.year);
```

The pre–defined function `strcpy()` can be used to initialize members which are of type character array. Input can be done using `scanf()` function with a `"%s"` as format symbol corresponding to character arrays. For example:

```
/* initialize using strcpy() */
strcpy(email_account.login_name,  "pusa");
strcpy(email_account.password,  "MuNinG123");

/* input using scanf("%s", ...) */
printf("Input login name: ");
scanf("%s", facebook_account.login_name);
printf("Input password: ");
scanf("%s", facebook_account.password);
```

4.6 Nested Structures

It is possible to declare a member which is also of type `struct`. We refer to this construct as nested structures. We declare, for example, `employee` as a nested structure below.

```
struct nameTag {
   char first[20];
   char middle[20];
   char last[20];
};

struct employeeTag {
    int IDnumber;
    struct nameTag name;
    struct dateTag birthdate;
};

struct employeeTag employee;
```

The corresponding parent-child diagram for the `employee` nested structure is shown in Figure 4.3. The diagram depicts that the structure (parent) `employee` has three members (children), namely `IDnumber`, `name` and `birthdate`. The last two members are also structures (parents) on their own. In particular, the members (children) of `name` are `first`, `middle` and `last`.

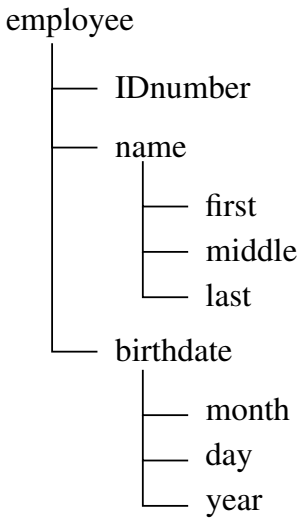


Figure 4.3: Graphical representation of the `employee` nested structure

Members of nested structures can then be accessed using the *structure member operator*. We show how to initialize the `employee` variable in the following code snippet.

```
employee.IDnumber = 123456;
strcpy(employee.name.first, "Jose");
strcpy(employee.name.middle, "Cruz");
strcpy(employee.name.last, "Santos");
employee.birthdate.month = 10;
employee.birthdate.day = 26;
employee.birtdate.year = 1980;
```

Alternatively, we can use `scanf()` to input the `employee` members. We omitted the `printf()` prompts for brevity in the following code.

```
scanf("%d", &employee.IDnumber);
scanf("%s", employee.name.first);
scanf("%s", employee.name.middle);
scanf("%s", employee.name.last);
scanf("%d", &employee.birthdate.month);
scanf("%d", &employee.birthdate.day);
scanf("%d", &employee.birtdate.year);
```

► **Self–Directed Learning Activity** ◄

1. Encode the program incorporating the code snippets above. Compile and test the codes.

2. Assume the following declarations:

```
struct A {
   int i;
   float f;
};

struct B {
   int x;
   double d;
};

struct C {
    int y;
    struct A a;
    struct B b;
};

struct C c;
```

   (a) Initialize the contents of variable `c` using direct assignments.

   (b) Initialize the contents of variable `c` using `scanf()`.

4.7 Structure to Structure Assignment

Let us say that we want to assign the value of `today` to `birthdate`. We can accomplish this by assigning each member of `today` to the corresponding member of `birthdate`, i.e.,

```
birthdate.month = today.month;
birthdate.day = today.day;
birthdate.year = today.year;
```

If a structure is made up of m number of members, then it would require m number of assignment statements to do this.

A better way to achieve the same effect is to assign a structure to another structure using the usual assignment operator. For example

```
birthdate = today;
```

As a rule, a structure to structure assignment is possible only if the variables involved have the same data type. If they are not, a syntax error will be reported by the compiler. In gcc, the corresponding error message is "error: incompatible types in assignment". The following is an example of an invalid assignment

```
landline = today;
```

because `struct phoneTag` is not the same as the data type `struct dateTag`.

▶ Self–Directed Learning Activity ◀

1. Verify that the assignment `birthdate = today;` is syntactically correct using the program in Listing 4.1.
2. Verify that the assignment `landline = today;` results into a compiler error.
3. Assume the following declarations:

```
struct A {
   int i;
   float f;
};

struct B {
   int i;
   float f;
};

struct C {
   float f;
   int i;
};

struct D {
```

```
    struct A a;
    struct C c;
};

struct A a1, a2;
struct B b1, b2;
struct C c1, c2;
struct D d1, d2;
```

Write the word VALID if the assignment statement below is syntactically correct, otherwise write INVALID.

(a) `a2.i = a1.i;`

(b) `b1.f = a2.f;`

(c) `b2.f = a1.i;`

(d) `c1.f = a1.f;`

(e) `a2 = a1;`

(f) `b1 = b2;`

(g) `a1 = b1;`

(h) `b2 = a1;`

(i) `c1 = a2;`

(j) `b1 = c2;`

(k) `d1.a = a1;`

(l) `d2.c.f = a2;`

(m) `c2 = d2.a;`

4.8 Passing a Structure as Function Parameter

A structure can be passed as a function parameter just like variables of simple data type. This is in consonance with the fact that structure to structure assignment is one of the operations allowed with structures.

Listing 4.2 shows how to define a function that has a structure as a parameter, and how the function can be called. It should be noted that the structure template must be declared prior to the declaration or definition of functions referring to the structure tag–name.

The `main()` function calls `PrintDate(today)`. The actual parameter `today` is assigned as the value of the formal parameter `myDate`.

Listing 4.2: Structure as a Function Parameter

```
1  #include <stdio.h>
2
3  /* take note of this global declaration */
4  struct dateTag {
5      int month;
6      int day;
7      int year;
8  };
9
10 void PrintDate(struct dateTag myDate)
11 {
12     printf("%d/%d/%d", myDate.month, myDate.day, myDate.year);
13 }
14
15 int main()
16 {
17      struct dateTag today;
18
19      today.month = 12;
20      today.day = 25;
21      today.year = 2009;
22
23      printf("Today's date is ");
24      PrintDate(today);
25
26      return 0;
27 }
```

► **Self–Directed Learning Activity** ◄

1. Encode Listing 4.2. Compile and test the program.
2. Add a declaration for `birthdate` variable with a data type of `struct dateTag`. Initialize the contents of `birthdate` using `scanf()`. Thereafter, call `PrintDate()` with `birthdate` as actual parameter.
3. Write a new function `int Equal(struct dateTag date1, struct dateTag date2)` which will return 1 if the two dates are equal. Otherwise, it should return 0. Assume that the two parameters have already been initialized before calling this function.
4. Write a new function `int Latest(struct dateTag date1, struct dateTag date2)` which will return 0 if the first parameter is the latest date between the two parameters, otherwise it should return a 1.

4.9 Function Returning a Structure

Functions that return a structure can also be defined. The value returned by such a function can be assigned to a recipient structure variable.

Listing 4.3 shows an example of how to define a function returning a structure. The return value of `GetDate()` is assigned in the `main()` function to another variable named `today`. Note that function `GetDate()` has a data type of `struct dateTag`.

Listing 4.3: Function Returning a Structure

```
1  #include <stdio.h>
2
3  /* take note of this global declaration */
4  struct dateTag {
5      int month;
6      int day;
7      int year;
8  };
9
10 struct dateTag GetDate(void)
11 {
12     struct dateTag myDate;
13
14     printf("Input month, day and year: ");
```

```
15      scanf("%d %d %d", &myDate.month, &myDate.day, &myDate.year);
16
17      return myDate;
18  }
19
20  void PrintDate(struct dateTag myDate)
21  {
22      printf("%d/%d/%d", myDate.month, myDate.day, myDate.year);
23  }
24
25  int main()
26  {
27       struct dateTag today;
28
29       today = GetDate();
30       printf("Today's date is ");
31       PrintDate(today);
32
33       return 0;
34  }
```

4.10 Pointers and Structures

4.10.1 Address of a Structure Variable

The address of a structure variable can be determined using the address–of operator `&`. For example, the addresses of the variables declared in Listing 4.1 will be displayed in the following `printf()` statements.

```
printf("&landline = %p\n.", &landline);
printf("&email_account = %p\n.", &email_account);
printf("&facebook_account = %p\n.", &facebook_account);
printf("&today = %p\n.", &today);
printf("&birthdate = %p\n.", &birthdate);
```

4.10.2 Pointer to a Structure

The address of a structure variable can be assigned to a structure pointer variable. Thereafter, the pointer variable can be used to access the members of the structure indirectly.

Listing 4.4 shows how to declare a structure pointer variable named `ptr`. It is basically the same as declaring a pointer to a simple data type. Variable `ptr` is then initialized as `ptr = &today`. Thereafter, the members of `today` are accessed indirectly by dereferencing `ptr`. Note that `*ptr` is the indirect expression corresponding to `today`. Try to see the similarity and the difference of an earlier program in Listing 4.2 with the program below.

Listing 4.4: Pointer to Structure

```
1  #include <stdio.h>
2
3  struct dateTag {
4      int month;
5      int day;
6      int year;
7  };
8
9  void PrintDate(struct dateTag myDate)
10 {
11     printf("%d/%d/%d", myDate.month, myDate.day, myDate.year);
12 }
13
14 int main()
15 {
16      struct dateTag today;
17      struct dateTag *ptr;   /* pointer to structure dateTag */
18
19      ptr = &today;        /* initialize pointer variable */
20      (*ptr).month = 12;  /* access today members indirectly */
21      (*ptr).day = 25;
22      (*ptr).year = 2009;
23
24      printf("Date today is ");
25      PrintDate(today);
26      printf("\n");
27
28      return 0;
29 }
```

Table 4.1: Direct and Indirect Access to Structure and its Members

Direct Access	Indirect Access
`today`	`*ptr`
`today.month`	`(*ptr).month`
`today.day`	`(*ptr).day`
`today.year`	`(*ptr).year`

Table 4.1 summarizes the direct access and the indirect access to the structure and its members. The `.` has a higher priority than `*`. This the reason why `*ptr` had to be enclosed within a pair of parentheses.

▶ Self–Directed Learning Activity ◀

1. Encode Listing 4.4. Compile and test the program.
2. In the function call to `PrintDate()` replace the parameter today with `*ptr`. Will it work?
3. What will happen if the initialization `ptr = &today;` is removed from the program?
4. Using the original codes of Listing 4.4, try to see what will happen if the parentheses enclosing `*ptr` are removed.
5. Refer to the declaration of `struct employeeTag employee` in Section 4.6. Create a C program that will do the following in sequence:
   (a) Declare a structure pointer variable of type `struct employeeTag *`.
   (b) Initialize the pointer variable with the address of `employee`.
   (c) Initialize the members of `employee` indirectly by dereferencing the pointer variable.

4.10.3 Structure Pointer Operator

The *structure pointer operator* denoted by ->, i.e. a dash immediately followed by a greater than symbol, is used to access a member indirectly via a pointer. Note that it can only be used by pointers to structure variables. The syntax for using the *structure pointer operator* is

```
<structure-pointer-var-name> -> <member-name>
```

The indirect initialization of `today` can then be rewritten as:

```
ptr->month = 12;   /* original: (*ptr).month = 12;  */
ptr->day = 25;     /* original: (*ptr).day = 25;    */
ptr->year = 2009;  /* original: (*ptr).year = 2009; */
```

where `(*ptr).member-name` is equivalent to `ptr->member-name`. Experienced C programmers prefer the `->` notation because it is shorter to write, and "easier" to read.

Table 4.2: Indirect Access via `*` and `->` Operators

Direct Access	Indirect Access via *	Indirect Access via ->
`today`	`*ptr`	not applicable
`today.month`	`(*ptr).month`	`ptr->month`
`today.day`	`(*ptr).day`	`ptr->day`
`today.year`	`(*ptr).year`	`ptr->year`

Table 6.1 summarizes the direct access and the two alternative ways for indirect access to the structure's members.

► Self–Directed Learning Activity ◄

1. Modify Listing 4.4 by replacing the initialization of variable `today` using the *structure pointer operator* `->`. Compile and test the program.
2. Using the codes in the previous item, find out what will happen if a space is inserted between `-` and `>` symbol.
3. Refer to the declaration of `struct employeeTag employee` in Section 4.6. Create a C program that will do the following in sequence:

(a) Declare a structure pointer variable of type `struct employeeTag *`.

(b) Initialize the pointer variable with the address of `employee`.

(c) Initialize the members of `employee` indirectly by using the structure pointer operator.

4.10.4 Pointer to a Structure as Parameter

We learned in Section 4.8 that structures can be passed as parameters to functions. If the size of the structure is large, it may not be a good idea to pass the entire structure because of space and time considerations. It will be faster and economical (in terms of memory space) to pass instead a pointer to a structure as function parameter.

The more important reason for passing a pointer to structure is the need to change the values of the members of the structure outside of the function where the structure variable was declared.

Listing 4.5 shows an example program which passes a pointer to structure as parameter. In functions `InputDate()` and `OutputDate()`, the formal parameter is `ptr` which has a data type of `struct dateTag *`. These functions are called in `main()` with the address of `today`, i.e., `&today` as parameter.

Try to recall Listing 4.3 and compare it with Listing 4.5. In particular, notice the difference between `GetDate()` with `InputDate()`. Compare also `PrintDate()` with `OutputDate()`.

Listing 4.5: Passing a Structure Pointer as Parameter

```
1  #include <stdio.h>
2
3  /* take note of this global declaration */
4  struct dateTag {
5      int month;
6      int day;
7      int year;
8  };
9
10 void InputDate(struct dateTag *ptr)
11 {
12     printf("Input month, day and year: ");
13     scanf("%d %d %d", &ptr->month, &ptr->day, &ptr->year);
14 }
```

```
15
16 void OutputDate(struct dateTag *ptr)
17 {
18     printf("%d/%d/%d", ptr->month, ptr->day, ptr->year);
19 }
20
21 int main()
22 {
23      struct dateTag today;
24
25      InputDate(&today);
26      printf("Today's date is ");
27      PrintDate(&today);
28
29      return 0;
30 }
```

► Self–Directed Learning Activity ◄

Refer to the declaration of `struct employeeTag employee` in Section 4.6. Create a C program such that it contains the definitions for the following functions:

1. `void InputEmployee(struct employeeTag *ptr)`

   This function will be used to input the values of the members of the structure.

2. `void OutputEmployee(struct employeeTag *ptr)`

   This function will be used to output the values of each member of the structure.

3. `int main()`

   This function will declare a structure variable named `employee` of type `struct employeeTag`. It should call the functions in-charge for input and output of `employee` member values.

4.11 Dynamic Memory Allocation of Structures

Memory space for a single instance or multiple instances of a structure type can also be allocated dynamically using the pre–defined function `malloc()`.

4.11.1 Single Instance Allocation

The program in Listing 4.6 demonstrates how to declare a structure pointer variable, how to dynamically allocate space for one instance of the structure type, how to initialize the members of the structure and how to free up the memory space. Notice that the parameter to `malloc()` is the size of the structure.

Listing 4.6: Dynamic Memory Allocation of One Structure Instance

```
1  #include <stdio.h>
2  #include <stdlib.h>  /* dont forget this header file */
3
4  struct dateTag {
5      int month;
6      int day;
7      int year;
8  };
9
10 int main()
11 {
12      struct dateTag *ptr;
13
14      if ((ptr = malloc(sizeof(struct dateTag))) == NULL) {
15         printf("Error: not enough memory space.\n");
16         exit(1);
17      }
18
19      ptr->month = 12;
20      ptr->day = 25;
21      ptr->year = 2009;
22      printf("%d/%d/%d\n", ptr->month, ptr->day, ptr->year);
23      free(ptr);
24
25      return 0;
26 }
```

► **Self–Directed Learning Activity** ◄

1. Encode and test the program in Listing 4.6.
2. Modify the program by replacing `ptr->member-name` with `(*ptr).member-name`.
3. Create a new C program that applies that same idea using `struct employeeTag`.

4.11.2 Multiple Instance Allocation

The program in Listing 4.7 demonstrates how to dynamically allocate memory space for multiple instances of a structure type. Allocation is done via function call to `ptr = malloc(sizeof(struct dateTag) * n)` where `n` is the number of instances.

The value of `ptr` is the base address; the address of the `i`'th element is given by the expression `ptr + i`.[2] The `month` member of the `i`'th element can then be accessed using the structure pointer operator, i.e., `(ptr + i)->month`. The parentheses are necessary in `(ptr + i)`. Without the parentheses `->` will be applied before `+` which will result into an error. The statement `scanf("%d %d %d", &(ptr+i)->month, &(ptr+i)->day, &(ptr+i)->year);` inputs the members of the `i`'th structure.

Listing 4.7: Dynamic Memory Allocation of Multiple Instances of a Structure

```
1  #include <stdio.h>
2  #include <stdlib.h>  /* dont forget this header file */
3
4  struct dateTag {
5      int month;
6      int day;
7      int year;
8  };
9
10 void InputDates(struct dateTag *ptr, int n)
11 {
12     int i;
13
14     for (i = 0; i < n; i++) {
15        printf("Input month day year for element %d: ", i);
16        scanf("%d %d %d",
```

[2] It is best to read the contents of Chapter 1 again in case you have forgotten pointer arithmetic.

```
17                &(ptr+i)->month, &(ptr+i)->day, &(ptr+i)->year);
18      }
19  }
20
21  void OutputDates(struct dateTag *ptr, int n)
22  {
23      int i;
24
25      for (i = 0; i < n; i++)
26         printf("Date %d is %d/%d/%d.\n", i,
27            (ptr+i)->month, (ptr+i)->day, (ptr+i)->year);
28  }
29
30  int main()
31  {
32       int n;
33       struct dateTag *ptr;
34
35       printf("How many structures would you like to allocate: ");
36       scanf("%d", &n);
37
38       if ((ptr = malloc(sizeof(struct dateTag) * n)) == NULL) {
39          printf("Error: not enough memory space.\n");
40          exit(1);
41       }
42
43       InputDates(ptr, n);
44       OutputDates(ptr, n);
45
46       free(ptr);
47       return 0;
48  }
```

▶ Self–Directed Learning Activity ◀

1. Encode and test the program in Listing 4.7.

2. Find out what will happen if we remove the pair of parenthess surrounding the address `ptr + i` in function `PrintDates()`.

3. Modify the program by replacing `ptr->member-name` with `(*ptr).member-name`.

4. Create a new C program that applies that same idea using `struct employeeTag`.

4.12 Array of Structures

Static memory allocation of multiple instances of structures other than listing the names of instances separated by a comma is possible. This is done by declaring an array of structure.

Consider for example how to declare 5 instances of `struct dateTag`. One of way of doing this is to have 5 separate variables declared as follows:

```
struct dateTag date1, date2, date3, date4, date5;
```

If the number of instances is increased, say to 1000, this kind of declaration will no longer be practical. A better approach is to declare a 1D array of structure. The syntax for an array of structure declaration is the same with how we declare 1D array of simple data types. For example, to declare 5 instances as an array, we write:

```
struct dateTag dateArray[5];
```

In this declaration, each element of `dateArray` is a structure. We will need to use both array indexing and structure member operator to access a particular member of a structure. The expression `dateArray[i]` is the `i`'th element of the array. To access the member `month` of this element, we write `dateArray[i].month`.

The following code snippet show an example of how to initialize the first array element structure members.

```
dateArray[0].month = 12;
dateArray[0].day = 25;
dateArray[0].year = 2009;
```

Notice that array indexing is performed first before the structure member operation.

If we want to input the members of the `i`'th element we would have to write:

```
scanf("%d %d %d", &dateArray[i].month, &dateArray[i].day,
       &dateArray[i].year);
```

The expression `&dateArray[i].month` is quite involved since there are three operators present. The operations are applied in the following order: first array indexing `[]`, followed by structure member operator `.`, and finally address–of operator `&`.

Listing 4.8 shows a complete program that illustrates an application of the concepts that we have learned so far. The name of the array[3] is passed as parameter to functions `InputDateArray()` and `OutputDateArray()`. Parameter `n` represents the number array elements.

Listing 4.8: Array of Structure

```
1  #include <stdio.h>
2
3  /* take note of this global declaration */
4  struct dateTag {
5      int month;
6      int day;
7      int year;
8  };
9
10 void InputDateArray(struct dateTag dateArray[], int n)
11 {
12     int i;
13
14     for (i = 0; i < n; i++)
15     {
16         printf("Input month, day and year of element %d: ", i);
17         scanf("%d %d %d", &dateArray[i].month,
18                &dateArray[i].day, &dateArray[i].year);
19     }
20 }
21
22 void OutputDateArray(struct dateTag dateArray[], int n)
23 {
24     int i;
25     for (i = 0; i < n; i++)
26          printf("Element %d is %d/%d/%d\n", i,
27                  dateArray[i].month, dateArray[i].day,
28                  dateArray[i].year);
29 }
```

[3]Recall that the name of the array is synonymous with the base address of the array.

```
30
31 int main()
32 {
33      struct dateTag dateArray[5];
34
35      InputDateArray(dateArray, 5);
36      OutputDateArray(dateArray, 5);
37
38      return 0;
39 }
```

Table 4.3: Accessing an Element of an Array of Structures

[ ] Notation	* Notation	-> Notation
A[i]	*(A + i)	not applicable
A[i].member-name	(*(A + i)).member-name	(A + i)->member-name

Table 4.3 summarizes the three alternative ways of accessing a structure and its members. The structure in this case is an element with index `i` in an array of structures which we named as `A` in the table.

► Self–Directed Learning Activity ◄

1. Encode and test the program in Listing 4.8.
2. We learned in Chapter 2 (Arrays) that an array element can be accessed via pointer dereference. Replace `dateArray[i]` above with `*(dateArray + i)`. Compile and run the program to see that you'll get the same results.
3. Create a new C program that applies that same idea using `struct employeeTag`.

4.13 typedef and struct type

typedef can be used to declare an alias for a struct data type. It can be used with Variation 1 (unnamed structure with no instance) of the structure declaration syntax. For example, the code below

```
typedef struct {
   char ch;
   int i;
   float f;
   double d;
} sampleType;
```

declares sampleType as an alias for the unnamed struct type.[4]

typedef can also be used together with Variation 2, i.e., named structure with no instance. For example, the following is a two part declaration.

```
struct sampleTag {
   char ch;
   int i;
   float f;
   double d;
};

typedef struct sampleTag sampleType;
```

The first part declares the structure template, thereafter the second part declares an alias for the structure template. These two separate declarations can be combined into one declaration as:

```
typedef struct sampleTag {
   char ch;
   int i;
   float f;
   double d;
} sampleType;
```

[4]The author actually finds this kind of declaration humorous. Why? Because we are declaring an alias for an "unnamed" structure. The structure does not have a (tag) name, yet we are giving it an alias! :-)

Structure variables may then be declared using the alias as data type. For example:

```
struct sampleTag s;
sampleType t;
```

declares a structure variable `s` using the structure template, while structure variable `t` was declared using the alias. Variables `s` and `t` have the same data type. Thus, the assignment `t = s;` will not result into a data type mismatch error.

Alias declared with `typedef` can also be used to declare structure pointer variables. For example:

```
sampleType *ptr;
```

We show how to use `typedef` in an actual program by rewriting the codes from Listing 4.3. The modifed codes (i.e., with `typedef`) are shown in Listing 4.9. The original program used `struct dateTag`. The modified program replaced the declarations of variables and parameters parameters and the definition of function `GetDate()` with the alias `dateType`.

Listing 4.9: `typedef` and `struct`

```
1  #include <stdio.h>
2
3  struct dateTag {
4      int month;
5      int day;
6      int year;
7  };
8
9  typedef struct dateTag dateType;
10
11 dateType GetDate(void)
12 {
13     dateType myDate;
14
15     printf("Input month, day and year: ");
16     scanf("%d %d %d", &myDate.month, &myDate.day, &myDate.year);
17
18     return myDate;
19 }
20
```

```
21 void PrintDate(dateType myDate)
22 {
23     printf("%d/%d/%d", myDate.month, myDate.day, myDate.year);
24 }
25
26 int main()
27 {
28      dateType today;
29
30      today = GetDate();
31      printf("Today's date is ");
32      PrintDate(today);
33
34      return 0;
35 }
```

The curious reader, at this junction, is probably asking "So which one is better, use a structure template or use a `typedef` alias?" The author choose not to answer this question with a simple yes or no.

There are groups of programmers who will choose to use a structure template over a `typedef` alias because the template clearly states the `struct` nature of an instance, parameter or function.

Another group of programmers prefer using `typedef` alias primarily because it allows them to write codes that are shorter (which are also easier to type because in general it requires fewer keystrokes).

The recommendation of the author to the reader is to make sure that you become conversant with both styles. In actual practice, you may choose one over the other; just observe consistency of use within the same set of codes.

► Self–Directed Learning Activity ◄

1. Verify that the assignment operation `t = s;` will not produce a data type mismatch error. Recall in the previous discussion that the variables were declared as:

   ```
   struct sampleTag s;
   sampleType t;
   ```

2. Rewrite all the sample programs, and your program solutions to the Self–Directed Learning Activities with a `typedef`.

4.14 Chapter Summary

The key ideas presented in this chapter are summarized below:

- A structure is a group of possibly heterogeneous elements.
- There are four variations for declaring a `struct` data type and structure variable.
- There are only three operations allowed on structure, namely:
  1. access an element of a structure
  2. assign a structure to another structure (includes parameter passing of a structure, and functions that returns a structure)
  3. get the address of a structure variable
- Two new operators were introduced, namely the *structure member operator* and *structure pointer operator*.
- Structures can be allocated using static memory allocation and dynamic memory allocation.
- `typedef` can be used to declare an alias to a structure.

Problems for Chapter 4

Assume the following declarations:

```
struct aTag {
    char ch;
    int i;
};

struct bTag {
    float f;
    double d;
};

struct cTag {
    struct aTag a;
    struct bTag b;
    int A[5];
```

```
};

typedef struct aTag aType;
typedef struct bTag bType;
typedef struct cTag cType;

aType a, A[5]
bType b, B[5];
cType c, C[5];

aType *pa;
bType *pb;
cType *pc;
```

Problem 4.1. What is the data type of the following expressions? DO NOT use the alias as answer. Write the word INCORRECT if the expression is incorrect.

1. a	21. &b.f	41. (*pc).b	61. &pb->f	81. C[2].b
2. b	22. &b.d	42. (*pc).A[3]	62. &pb->d	82. C[2].A[1]
3. c	23. &c.a	43. *pc.A[2]	63. &pc->a	83. C[0].a.ch
4. a.ch	24. &c.b	44. (*pc).a.ch	64. &pc->b	84. C[0].a.i
5. a.i	25. &c.a.ch	45. (*pc).a.i	65. &pc->A[4]	85. C[0].b.f
6. b.f	26. &c.a.i	46. (*pc).b.f	66. &pc->a.ch	86. C[0].b.d
7. b.d	27. &c.b.f	47. (*pc).b.d	67. &pc->a.i	87. C[0].A[5]
8. c.a	28. &c.b.d	48. pa->ch	68. &pc->b.f	88. &C[1].a.ch
9. c.b	29. &c.A[0]	49. pa->i	69. &pc->b.d	89. &C[1].b.f
10. c.a.ch	30. &c.A[3]	50. pb->f	70. A[0]	90. &C[1].b.d
11. c.a.i	31. pa	51. pb->d	71. B[4]	91. &C[1].A[2]
12. c.b.f	32. pb	52. pc->a	72. C[2]	92. *(C + 2)
13. c.b.d	33. pc	53. pc->b	73. &A[1]	93. (*(C + 2)).a
14. c.A[0]	34. *pa	54. pc->A[3]	74. &B[3]	94. (C+2)->b
15. c.A[5]	35. *pb	55. pc->a.ch	75. &C[0]	95. (C+3)->a.i
16. &a	36. (*pa).ch	56. pc->a.i	76. A[0].ch	96. &(C+3)->b.f
17. &b	37. (*pa).i	57. pc->b.f	77. A[0].i	97. A[0].i++
18. &c	38. (*pb).f	58. pc->b.d	78. B[1].f	98. B[1]--
19. &a.ch	39. (*pb).d	59. &pa->ch	79. B[1].d	99. C[2].a++
20. &a.i	40. (*pc).a	60. &pa->i	80. C[2].a	100.(C+4)->a.ch--

Problem 4.2. Implement a function `void InputFunc1(cType *ptr)` that will input via `scanf()` the members of the structure pointed to by `ptr`. Assume that the memory space already exist.

Problem 4.3. Implement a function `cType InputFunc2(void)` that will declare a local variable as `cType temp`. The function will then input the values of the members of `temp` using `scanf()`. Finally, the function will return `temp`.

Problem 4.4. Implement a function `void InputFunc3(cType C[], int n)` that will input via `scanf()` the elements of the array of structure `C`. Parameter `n` represents the number of elements in the array.

Problem 4.5. Implement a function `void OutputFunc1(cType *ptr)` that will output the members of the structure pointed to by `ptr`. Assume that the memory space already exist and that the values of the structure are valid. Use dereference operator and structure member operator only. DO NOT use the structure pointer operator `->`.

Problem 4.6. Implement a function `void OutputFunc2(cType *ptr)` similar to the previous problem. The difference here is that it is required to use the structure pointer operator `->`.

Note: In the following problems the second parameter `int n` *represents the number of elements in an array.*

Problem 4.7. Implement a function `void OutputFunc3(cType C[], int n)` that will output the elements of array C using array indexing.

Problem 4.8. Implement a function `void OutputFunc4(cType C[], int n)` that will output the elements of array C using structure pointer operator instead of array indexing.

Problem 4.9. Implement a function `int Total1(aType A[], int n)` which will return the sum of the member `i` of the structures in array A.

Problem 4.10. Implement a function `int Total2(cType c)` which will return the sum of structure `c`'s member array `A`.

Problem 4.11. Implement a function `int Total3(cType C[], int n)` which will return the sum of *ALL* the elements of member array `A` for all structures. Note: the previous problem computes the sum from just one structure. This problem computes the sum of the member `A` for all elements of `C`.

Problem 4.12. Implement a function `int Minimum(cType c)` which will return

the index of the smallest value in `c`'s member array `A`.

Problem 4.13. Implement a function `float fMinimum(bType B[], int n)` which will return the index of the element in array B whose member `f` is the smallest.

Problem 4.14. Implement a function `double dMaximum(bType B[], int n)` which will return the index of the element in array B whose member `d` is the largest.

Problem 4.15. Implement a function `void SortFunc1(aType A[], int n)` that will sort the elements of array `A` in *increasing* order based on member `i`. Use the straight selection sort algorithm discussed in Chapter 2 (Arrays).

Problem 4.16. Implement a function `void SortFunc2(bType B[], int n)` that will sort the elements of array `B` in *decreasing* order based on member `d`. Use the straight selection sort algorithm discussed in Chapter 2 (Arrays).

Problem 4.17. Implement a function `void CopyFunc(cType sourceArr[], cType destArr[], int n)` that will copy elements of the `sourceArr` array to the `destArr` array (same index). Parameter `n` is the number of elements of the arrays.

References

Consult the following resources for more information.

1. comp.lang.c Frequently Asked Questions. `http://c-faq.com/`

2. Kernighan, B. and Ritchie, D. (1998). *The C Programming Language, 2nd Edition*. Prentice Hall.

Chapter 5

Linked List

5.1 Motivation

PDAs such as mobile phones have a software application for storing, editing, deleting and retrieving data about the user's contacts. The usual information includes a contact person's name and phone number. The list of contact information is usually sorted alphabetically based on the contact person's name.

Let us now imagine that you are given the task of implementing a program for maintaining a list of contact information. How will you represent the data? How will you implement the functions for adding, editing, deleting, retrieving and listing contact information? Give yourself some time to think about this problem. Try to develop some preliminary programs to test if your ideas are sound.

One approach that you have probably considered is to use one dimensional array to represent and store the list of contact information. Each array element element is a structure made up of two members, namely, name and phone number. If efficient memory usage is a primary concern, memory for the list can be obtained instead via dynamic memory allocation.

Let us assume at this point that an an array representation is going to be used. How do we implement the functions for adding and deleting contact information? Note that the list should be maintained in sorted order. It will be very inefficient to perform straight selection sort after each add or delete operation.

In this chapter, we will learn another approach using a data structure known as linked list. It is appropriate for problems similar to the one described above.

5.2 What is a Linked List?

A *linked list* is a group of elements. The elements of a linked list are referred to as *nodes*. A node is actually a structure with the following members: (i) data and (ii) one or two pointers to a node structure.

A *single linked list* is a type of linked list with nodes containing only one pointer member each. There is another type of linked list called *double linked list* with nodes that have two pointers each. The qualifiers "single" and "double" refer to the number of pointers in a node.

In the following discussions, we will concentrate our efforts on the representation and manipulation of single linked lists only.[1] Moreover, we will just use the shorter term *linked list* to actually mean single linked list throughout the remainder of this chapter.

5.3 Graphical Representation

We will use graphical symbols, shown in Figure 5.1, for visualizing linked lists.

- A directed line segment (line with an arrow head) represents a pointer.
- A box represents a node. The node as mentioned above is a structure containing two members, namely data and a pointer. The pointer will point to the "next" node in the list.
- An electrical ground symbol represents `NULL`. In the context of linked lists, the `NULL` value is used to indicate the fact that there is no more "next" node in the list, i.e., it is the end of the list.

Empty Linked List

The simplest possible linked list is an *empty* list – it is a list that does not contain any node. We will represent it as a line segment leading to an electrical ground symbol as shown in Figure 5.2. In all the sample codes in this chapter, we will use the identifier `pFirst` as the name for the pointer to the linked list. The instruction `pFirst = NULL` will initialize a linked list to an empty state. In a latter section, we will show how to add nodes into a linked list.

[1] Double linked lists will be covered in another subject called Data Structures and Algorithms (course code: DASALGO).

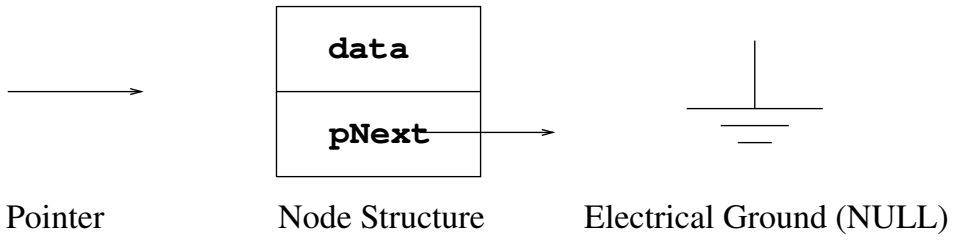


Figure 5.1: Graphical Symbols for Linked List

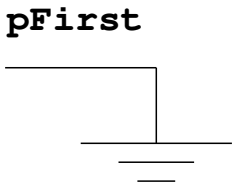


Figure 5.2: Empty Linked List

Linked List With Only One Node

A linked list containing just one node is shown in Figure 5.3. Notice that

- `pFirst` is drawn such that it points to the *first* node and,
- the pointer member of the node has a value of `NULL` and visually represented as a line leading to an electrical ground symbol.

If a linked list has only one node, then that node is the *first* node and at the same time, the *last* node in the linked list. The last node in the list has a pointer member set to `NULL`.

Linked List With More Than One Node

Figure 5.4 shows a linked list containing three nodes. Just like in a one–node list, `pFirst` points to the first node in the list, the pointer of the first node points to the second node, the pointer of the second node points to the third node, and the pointer of the third node is set to `NULL`. Figure 5.4 should suggest by now an

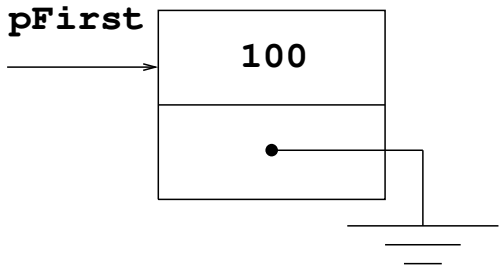


Figure 5.3: Linked List With One Node

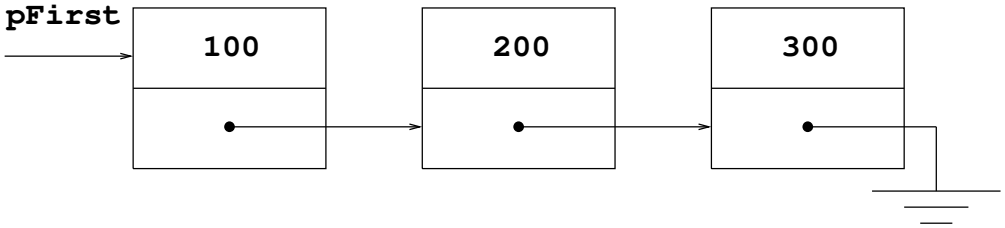


Figure 5.4: Linked List With More Than One Node

explanation why we use the term "linked" prior to "list". The pointer in a node links, i.e., connects it or serves as a bridge to go to the next node in the list. The links allow us to access all the nodes in the list starting from the first node via `pFirst`. The linked nature of the list restricts the access to a sequential mode. Sequential in this case means that before we can access the third node, we have to pass through the first and the second nodes.[2] There is no way we can access the third node directly.

5.4 Linked List Data Type Declaration

We are now ready to discuss how to implement linked lists using the C programming language. In this section, we first consider the data type declarations involved.

[2]In contrast, array elements can be accessed directly without passing through any other elements. We have seen in Chapters 1 and 2 that this is actually done by computing directly the address of the element as the sum of the base address and the element's index.

A node is a structure that basically contains two members: (i) data and (ii) a pointer to the next node. The data member can be of a simple data type or a structure on its own. Listing 5.1 shows an example of a `struct` type declaration for a node:

Listing 5.1: Data and Node Structure Declarations

```
1  /* Note: save the following declarations in "nodetype.h" file */
2  struct dataTag {
3     int key;
4     /* declare additional members here */
5  };
6
7  struct nodeTag {
8     struct dataTag data;
9     struct nodeTag *pNext;  /* pointer to the next node */
10 };
11
12 typedef struct dataTag dataStructType;
13 typedef struct nodeTag nodeStructType;
```

To avoid repetitions in coding in latter sections, let us save the declarations above in a header file named as `"nodetype.h"`. Subsequent example codes will simply have `#include "nodetype.h"` in order to refer to these names.

The `struct dataTag` is a structure template for the `data` member. In the example, there is only one member named `key`. Additional members should be declared depending on the problem being solved. The `struct nodeTag` is a structure template for the node. Its members are `data` and the pointer `pNext`. It is very interesting to note that `struct nodeTag` is already being used inside the `struct` type declaration itself. Structures which use its name as a data type for its member inside its own declaration are called *self–referential structures*.

The `typedef` declarations for `dataStructType` and `nodeStructType` are actually optional. The aliases can be used in place of the `struct` types such as in variable declarations.

The pointer `pFirst` can then be declared in two ways; first, using the `struct` type, i.e.,

```
struct nodeTag *pFirst;
```

and second by using its alias, i.e.,

```
nodeStructType *pFirst;
```

5.5 Operations on Linked Lists

The following are basic operations that can be performed on a linked list:

- Initialize a linked list as an empty list
- Create a new node and initialize its contents
- Add a new node into a list[3]
- Traverse the list
- Retrieve a node (for example get the last node) or retrieve information from a node in the list (for example, search fo a node containing a specified key value)
- Delete a node from a list

The details of these operations will be explained in the following sections.

5.6 How to Create an Initially Empty List

Linked lists are usually initialized such that they are empty, i.e., there are no nodes at the start as illustrated in Figure 5.2. This is done by setting the pointer `pFirst` to a `NULL` value. An example initialization code is shown in Listing 5.2.

Listing 5.2: Linked List Initialization Inside `main()`

```
1 #include "nodetype.h"
2 int main()
```

[3]Sometimes we use the word "insert" instead of "add" to mean the same operation.

```
3 {
4     nodeStructType *pFirst; /* pointer to the linked list */
5
6     pFirst = NULL;
7
8     /*-- other codes follow --*/
9 }
```

In larger programs, the initialization of the linked list is implemented as a separate function. There are two approaches to accomplish this. The first approach is actually the complicated one because it requires the use of a pointer to a pointer variable.

Listing 5.3 shows function `Initialize()` with a parameter of `nodeStructType **pptr`. Most beginning programmers would have difficulty grasping the meaning of the double asterisks in such a data type.[4] Function `main()` calls `Initialize()` with the value of the address of `pFirst`, i.e., `&pFirst` as parameter.

Listing 5.3: Complicated Initialization of `pFirst` Outside of `main()`

```
1  /* notice the double asterisks in the formal parameter */
2  void InitializeList(nodeStructType **pptr)
3  {
4       *pptr = NULL;
5  }
6
7  int main()
8  {
9      nodeStructType *pFirst;
10
11     InitializeList(&pFirst);
12     /* pass the value of the address of pFirst */
13     /* pFirst becomes NULL after calling InitializeList() */
14     return 0;
15 }
```

The second approach is easier to understand and implement. It does not require any parameter passing and pointer dereferencing. This is accomplished by implementing the initialization function such that it returns the value of a

[4]It actually refers to the data type of the address of a pointer variable.

structure pointer. The return value is then assigned to a recipient pointer variable in `main()` as shown in Listing 5.4.

For these reasons, we will use this way of initialization in all subsequent example codes and programs. Moreover, we will implement functions returning a structure pointer to effect a change in a pointer variable outside of the function where it was declared.

Listing 5.4: Easier Initialization of `pFirst` Outside of `main()`

```
1  nodeStructType *InitializeList(void)
2  {
3      return NULL;
4  }
5
6  int main()
7  {
8      nodeStructType *pFirst;
9
10     pFirst = InitializeList();
11     return 0;
12 }
```

► Self–Directed Learning Activity ◄

1. Insert a `printf("pFirst = %p", pFirst);` after initializing `pFirst` in the `main()` of Listings 5.2 to 5.4. Test the programs to see the results.

2. Write a new function `int IsEmpty(nodeStructType *pFirst)` that will determine if a linked is empty or not. If the linked list is empty `IsEmpty()` should return a 1, otherwise it should return a 0.

5.7 How to Create and Initialize a New Node

The next step after initializing a linked list as an empty list is to add new nodes to it. Before we can add new nodes, we must (i) first create a node via dynamic memory allocation, and then (ii) initialize the members of the node. Usually, the pointer field is initialized to `NULL`. Later on, the pointer member can be adjusted to make it point to another node.

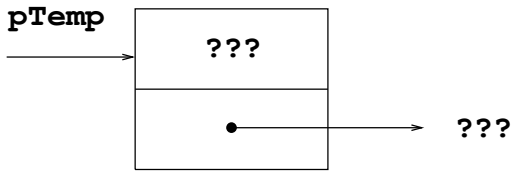
Node creation and initialization is illustrated in Listing 5.5. `CreateNewNode()` accepts `data` as formal parameter. In the function definition, memory space for the node is allocated dynamically via `malloc(sizeof(nodeStructType))`. The address of the node is assigned to a local pointer variable named `ptr`. The members of the node structure are then initialized by dereferencing the pointer `ptr`. Finally, `CreateNewNode()` returns the pointer value, i.e., the address of the node. This address should be stored by the function that called `CreateNewNode()` to another recipient pointer variable; for example, `pTemp = CreateNewNode(data);`.

Take note that we do not want to `free` the memory inside `CreateNewNode()`. Memory deallocation will have to be done in another function, i.e., when we delete the list or free up the entire linked list.

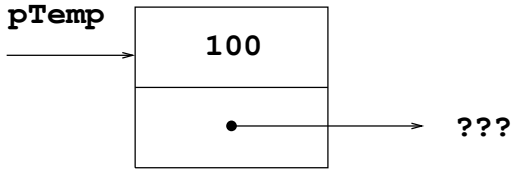
Listing 5.5: Creating and Initializing a Node

```
1  nodeStructType *CreateNewNode(dataStructType data)
2  {
3      nodeStructType *pTemp;
4
5      if ((pTemp = malloc(sizeof(nodeStructType))) == NULL) {
6          printf("ERROR: not enough memory.\n");
7          exit(1);
8      }
9
10     /* initialize node members */
11     pTemp->data = data;
12     pTemp->pNext = NULL;
13
14     return pTemp;
15 }
```

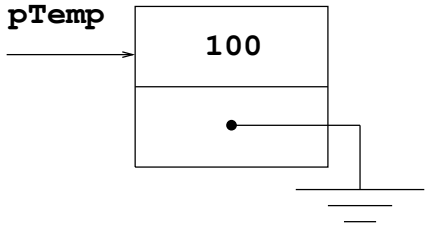
Figure 5.5 shows a graphical explanation of what happens in the node creation and initialization.



(a) node after `pTemp = malloc()` (the ??? means garbage)



(b) node after `pTemp->data = data` (example: `data.key = 100`)



(c) after `pTemp->pNext = NULL`

Figure 5.5: Creation and Initialization of a New Node

5.8 Brute Force Addition of Nodes in a Linked List

We demonstrate the use of `CreateNewNode()` in the following example codes. Take note, however, that the examples are very crude. The objective for the moment is to demonstrate how a linked list can be built.

Example 1. Listing 5.6 shows how to build a linked list with only one node first. The `main()` function called `CreateNewNode()` and its return value was stored in temporary pointer variable `pTemp`. The new node was made the first node in the list by the pointer to pointer assignment `pFirst = pTemp;`. At this point in time, two pointers are pointing to the new node, namely `pTemp` and `pFirst`.

Listing 5.6: Linked List with Only One Node

```
1  /*-- other codes appear before main() --*/
2
3  int main()
4  {
5      nodeStructType *pFirst;
6      nodeStructType *pTemp;
7      dataStructTytpe data;
8
9      /* initialize linked list as an empty list */
10     pFirst = Initialize();
11
12     /*create a new node */
13     data.key = 100;
14     pTemp = CreateNewNode(data);
15
16     /* add new node as the 1st node node in the list */
17     pFirst = pTemp;
18
19     /*-- other codes to follow --*/
20 }
```

Example 2. Listing 5.7 builds a linked list with three nodes. Warning: brute force approach! In real life, we will not create linked lists like in this program. The correct way will be explained in a latter section. The program calls `CreateNewNode()` three times. The return value of `CreateNewNode()` are then stored to temporary pointer variables `pTemp1`, `pTemp2`, and `pTemp3`. At this point in time, the three nodes are not yet linked.

Linking is done by setting the link field of the first node to point to the second node, i.e. `pTemp1->pNext = pTemp2`. The second and third nodes are linked via `pTemp2->pNext = pTemp3`. Finally, the assignment `pFirst = pTemp1` makes these three nodes elements of the linked list. The resulting list can be graphically represented as in Figure 5.4.

Listing 5.7: Brute Force Creation of Linked List With Three Nodes

```
1  /*-- other codes appear before main() --*/
2
3  int main()
4  {
5      nodeStructType *pFirst;
6      nodeStructType *pTemp1, *pTemp2, *pTemp3;
7      dataStructTytpe data;
8
9      /* initialize linked list as an empty list */
10     pFirst = Initialize();
11
12     /*create three nodes */
13     data.key = 100;
14     pTemp1 = CreateNewNode(data);
15
16     data.key = 200;
17     pTemp2 = CreateNewNode(data);
18
19     data.key = 300;
20     pTemp3 = CreateNewNode(data);
21
22     /* link these nodes together */
23     pTemp1->pNext = pTemp2; /* link 1st to 2nd node */
24     pTemp2->pNext = pTemp3; /* link 2nd to 3rd node */
25
26     /* make these nodes elements of the linked list */
27     pFirst = pTemp1;
28
```

```
29      /*-- other codes to follow --*/
30  }
```

5.9 Traversing the Linked List

Traversing the list means to *visit* each node in the list starting from the first node down to the last node. Visit is actually a generic term that means "to access a node". The specific action to be done per node is dependent on the problem being solved. For the meantime, we will assume that the visit action requires that we print the value of the data member of the node.

The algorithm for the list traversal is as follows:

1. Let a temporary pointer, say `pCurrent`, point to the first node in the list.
2. While it is not yet the end of the list, do the following:
   (a) Visit the node
   (b) Go to the next node in the list.

The implementations of the `Visit()` and `Traverse()` functions are shown in Listing 5.8.

Listing 5.8: Link List Traversal

```
1  void Visit(nodeStructType *ptr)
2  {
3       printf("%d ", ptr->data.key);
4
5       /* more printfs if there are more data members */
6  }
7
8  void Traverse(nodeStructType *pFirst)
9  {
10      nodeStructType *pCurrent;
11
12      pCurrent = pFirst;
13      while (pCurrent != NULL) { /* while not yet end of list */
```

```
14            Visit(pCurrent);              /* visit each node */
15            pCurrent = pCurrent->pNext;   /* go to the next node */
16      }
17  }
```

Actually, we can implement `Traverse()` without the need for a local variable. Since `pFirst` is a just local copy of the pointer to the first node, we can change the copy without affecting the original variable. This is advantageous for two reasons: (i) we save memory because there is no need for a local pointer variable, and (ii) we save also on time because there is no more local variable initialization involved. The improved implementation of `Traverse()` is shown in Listing 5.9.

Listing 5.9: A Better Link List Traversal

```
1  void Traverse(nodeStructType *pFirst)
2  {
3       while (pFirst != NULL) { /* while not yet end of list */
4            Visit(pFirst);             /* visit each node */
5            pFirst = pFirst->pNext;      /* go to the next node */
6       }
7  }
```

▶ Self–Directed Learning Activity ◀

1. Add the implementations of `Visit()` and `Traverse()` functions to Listing 5.7. Thereafter, insert the function call `Traverse(pFirst)` after the comment that says "other codes to follow". Compile and test the program.

2. Write a function `int CountNodes(nodeStructType *pFirst)` that will determine and return the number of nodes in a linked list. Test the function to see that it works properly.

5.10 Getting the Last Node in the List

There are problems that will require us to access the last node in the list. Remember that the last node has a link whose value is `NULL`.

Due to the nature of the linked list, the last node can only be accessed by first passing through all the nodes preceding the last node. The following algorithm show how this can be done:

1. Set a local pointer variable to point to the first node.
2. While the pointer is not pointing to the last node in the list, move it to the next node in the list.

Listing 5.10 shows us the implementation of function `GetLastNode()`. This function returns `NULL` if the list is empty, otherwise it returns the address of the last node. Notice that the codes are very much similar to the steps in the traversal algorithm except for the condition in the `while` loop.

Listing 5.10: Retrieving the Last Node in the List

```
1 nodeStructType *GetLastNode(nodeStructType *pFirst)
2 {
3     if (pFirst != NULL)
4        /* while the link field of the current node is not NULL */
5        while (pFirst->pNext != NULL)
6             pFirst = pFirst->pNext;     /* go to the next node */
7
8     return pFirst;
9 }
```

5.11 Freeing the Entire Linked List

How do you free an entire linked list containing more than one node? A naive answer would be to call `free(pFirst)` where `pFirst` is the pointer to the first node. This instruction will definitely free up the memory space allocated to the first node, but not of those allocated to all the other nodes. A memory leak will result. Imagine that if there are one million nodes in the list, only 1 node will be freed and the remaining 999,999 nodes will become inaccessible! The memory space allocated to them cannot be recovered anymore during run–time.

To free up the memory allocated to the entire list, it is imperative to free up each node in the list. The steps for doing these are quite similar to a list traversal

except that we are "destroying" nodes as we travel down the list. The algorithm is as follows:

1. Let `pFirst` be the pointer to the first node. Initialize a temporary pointer, say `pCurrent`, to point also to the first node.
2. While there are still nodes in the list
   (a) Move `pFirst` to point to the next node.
   (b) Free the node pointed to by `pCurrent`
   (c) Make `pCurrent` point to the first node in the list.

The implementation for the algorithm is shown in Listing 5.11.

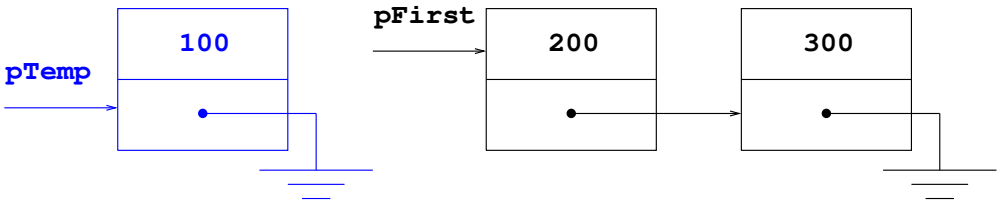
Listing 5.11: Freeing an Entire Linked List

```
1  void FreeList(nodeStructType *pFirst)
2  {
3       nodeStructType *pCurrent;
4
5       pCurrent = pFirst;
6       while (pFirst != NULL) { /* while there are nodes */
7          pFirst = pFirst->pNext; /* move to the next node */
8          free(pCurrent);         /* free up a node */
9          pCurrent = pFirst;      /* move pCurrent to 1st node */
10      }
11 }
```

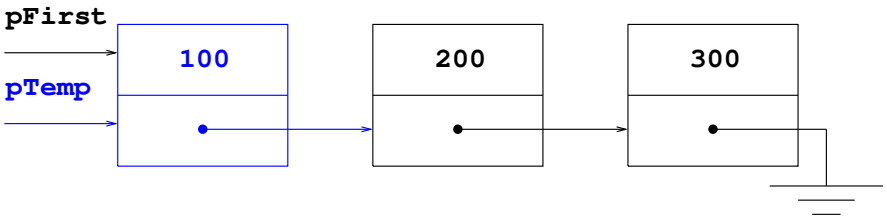
Note that moving `pFirst` to next node, i.e., `pFirst = pFirst->pNext` should be done before `free(pCurrent)`. If the sequence is interchanged, the remaining nodes will become inaccessible resulting into a memory leak.

5.12 How to Add a Node in a Linked List

We now consider a more systematic way of building a linked list. There are three possible cases to consider when we add a new node into a list – which may or may not be empty.



(a) Linked list before adding new node



(b) Linked list after adding new node

Figure 5.6: A new node is added as first node in a non–empty linked list

- Add a new node as the first node in the list.
- Add a new node as the last node in the list.
- Add a new node in between two existing nodes.

We discuss how to achieve the first two cases in the following subsections. The last case will be covered in a later section where we discuss insertion sort on linked lists.

5.12.1 Case 1: Add new node as first node

The algorithm for the addition of a new node as the first node in the list is the easiest among the three cases. The two-step algorithm is as follows:

1. Set the link field of the new node to point to where `pFirst` is pointing to.
2. Set `pFirst` to point to the new node.

Figure 5.6 shows graphically how to add the new node as the first node in the list. A function that implements this algorithm is shown in Listing 5.12. The algorithm works correctly on both an initially empty list, and a non–empty list.

Listing 5.12: Add New Node as First Node

```
1  /* pFirst is the pointer to the first node;
2     pTemp is the pointer to the new node */
3  nodeStructType *Add_As_FirstNode(nodeStructType *pFirst,
4                                   nodeStructType *pTemp)
5  {
6      /* link new node to the "former" first node */
7      pTemp->pNext = pFirst;
8
9      /* new node will be set as first node */
10     pFirst = pTemp;
11
12     return pFirst;
13 }
```

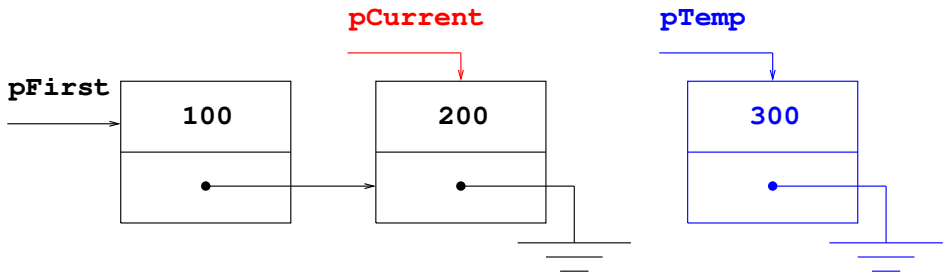
5.12.2 Case 2: Add new node as last node

The algorithm for adding a new node as the last node in the list as follows:

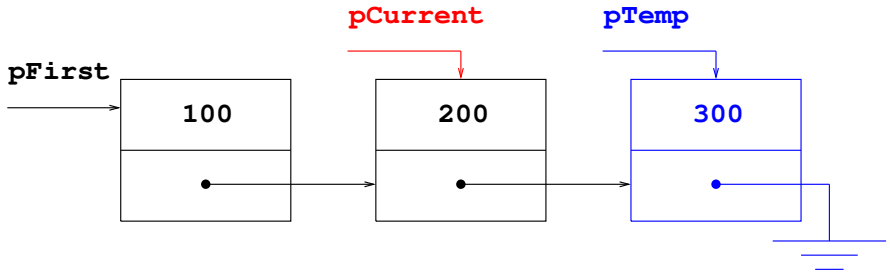
1. If the linked list is empty, set `pFirst` to point to the new node. STOP.
2. If the linked list is not empty
   (a) Set a local pointer variable, say `pCurrent`, to make it point to the last node in the list (refer back to Section 5.10).
   (b) Set the link field of the last node to point to the new node.

A graphical representation of how a node is added as the new last node in a linked list is shown in Figure 5.7.

Two equivalent implementations of the algorithm are shown in Listing 5.13 and 5.14. Note that Listing 5.14 calls the `GetLastNode()` function which was defined way back in Section 5.10.



(a) Linked list before adding new node as last node in the list



(b) Linked list after executing `pCurrent->pNext = pTemp`

Figure 5.7: A new node is added as last node in a non–empty linked list

Listing 5.13: Add New Node as Last Node

```
1  /* pFirst is the pointer to the first node;
2     pTemp is the pointer to the new node */
3  nodeStructType *Add_As_LastNode(nodeStructType *pFirst,
4                                  nodeStructType *pTemp)
5  {
6      nodeStructType *pCurrent;
7
8      if (pFirst == NULL) /* list is empty */
9          pFirst = pTemp;  /* insert new node as last node */
10     else { /* list has at least one node */
11         pCurrent = pFirst;
12         /* make pCurrent point to the last node */
13         while (pCurrent->pNext != NULL)
14               pCurrent = pCurrent->pNext;
15
16         /* add new node as the last node in the list */
17         pCurrent->pNext = pTemp;
```

```
18      }
19      return pFirst;
20  }
```

Listing 5.14: Add New Node as Last Node – calls `GetLastNode()`

```
1   /* pFirst is the pointer to the first node;
2      pTemp is the pointer to the new node */
3   nodeStructType *Add_As_LastNode(nodeStructType *pFirst,
4                                   nodeStructType *pTemp)
5   {
6       nodeStructType *pCurrent;
7
8       if (pFirst == NULL) /* list is empty */
9           pFirst = pTemp;  /* insert new node as last node */
10      else { /* list has at least one node */
11          /* make pCurrent point to the last node */
12          pCurrent = GetLastNode(pFirst);
13
14          /* add new node as the last node in the list */
15          pCurrent->pNext = pTemp;
16      }
17      return pFirst;
18  }
```

5.13 How to Delete a Node From a Linked List

Nodes that are no longer needed should be deleted from a linked list. The `free()` function is used to deallocate the memory associated with the node to be deleted. There are three possible cases to consider:

- Delete the first node from the list.
- Delete the last node from the list.
- Delete a node in between two other nodes.

We discuss how to achieve the first two cases in the following subsections. The last case will be covered in a later when we discuss deletion from a sorted list.

5.13.1 Case 1: Delete the first node

The algorithm for the deletion of the first node is the easiest among the three cases. The algorithm is shown below with Listing 5.15 as its C implementation. Figure 5.8 shows a graphical representation of the deletion steps.

1. Set a temporary pointer, say `pTemp` to point to the first node.
2. Set `pFirst` to point to the next node.
3. Delete the node pointed to by `pTemp` via `free(pTemp)`.

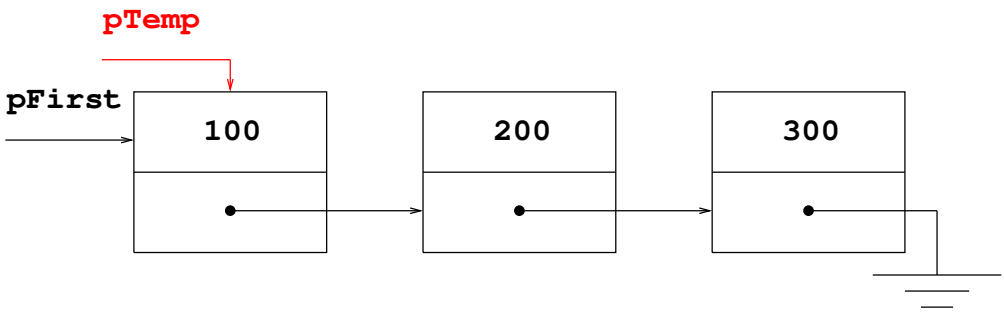
Listing 5.15: Delete First Node From the List

```
1  nodeStructType *Delete_FirstNode(nodeStructType *pFirst)
2  {
3      nodeStructType *pTemp;
4
5      if (pFirst == NULL) /* list is empty */
6         printf("List is empty. No node to delete.\n");
7      else {
8         pTemp = pFirst;
9         pFirst = pFirst->pNext;
10        free(pTemp);
11     }
12     return pFirst;
13 }
```

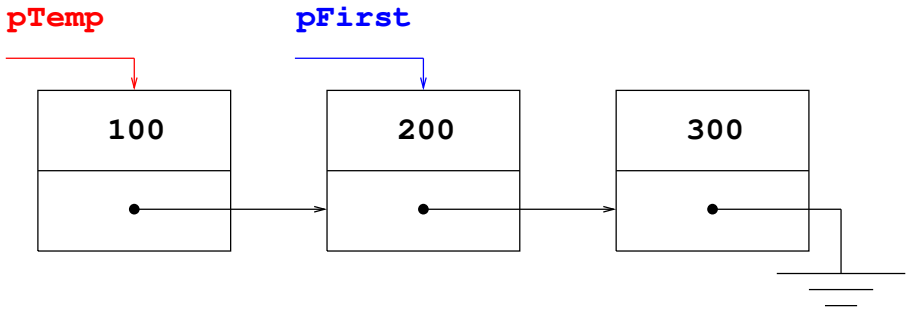
5.13.2 Case 2: Delete the last node

The algorithm for deleting the last node requires two local pointers named `pTrailer` and `pCurrent`. Pointer `pTrailer` is a pointer that is positioned just one node behind (i.e., previous to) the node pointed to by `pCurrent`.

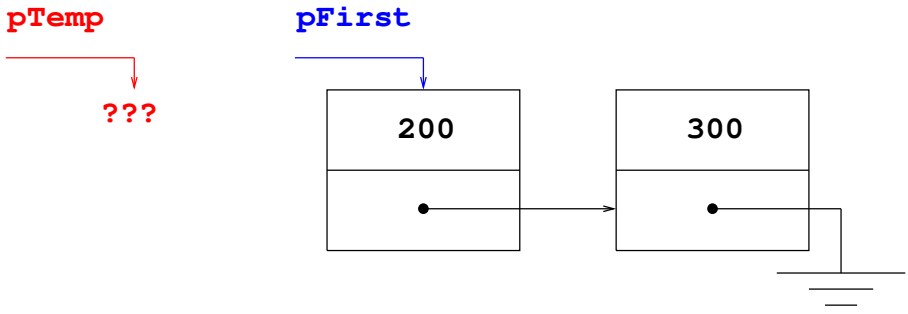
1. Initialize a local pointer variable, say `pTrailer`, to `NULL`, and another variable say `pCurrent`, such that it points to the first node.
2. While `pCurrent` is not yet pointing to the last node, do the following:



(a) Linked list after executing `pTemp = pFirst`



(b) Linked list after executing `pFirst = pFirst->pNext`



(c) Linked list after executing `free(pTemp)`

Figure 5.8: Steps in deleting the first node from a non–empty linked list

   (a) Set `pTrailer` such that it points to the node pointed to by `pCurrent`
   (b) Adjust `pCurrent` by pointing it to the next node in the list.

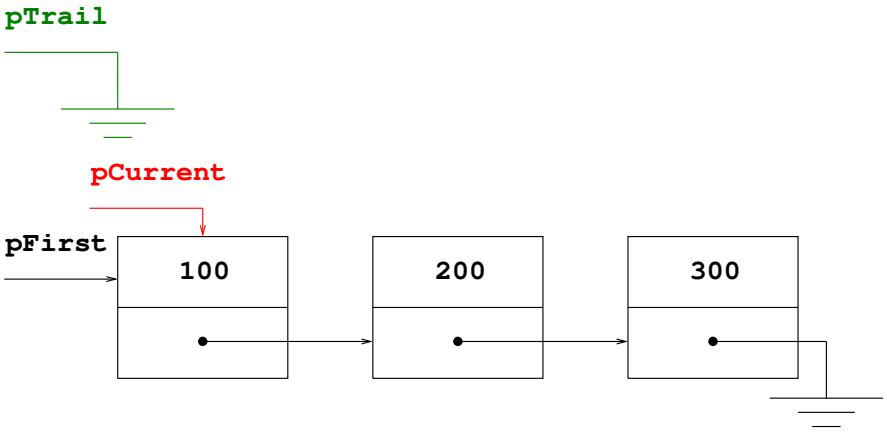
3. Set the link field of the node pointed to by `pTrail` to `NULL`.

4. Delete the node pointed to by `pCurrent` via `free(pCurrent)`.

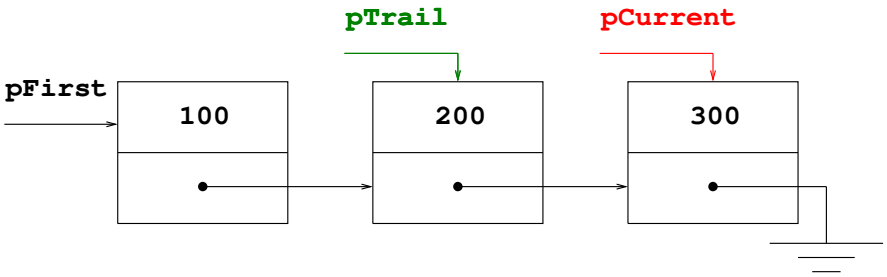
The algorithm's implementation is shown in Listing 5.16 and a graphical representation is depicted in Figure 5.9.

Listing 5.16: Delete Last Node From the List

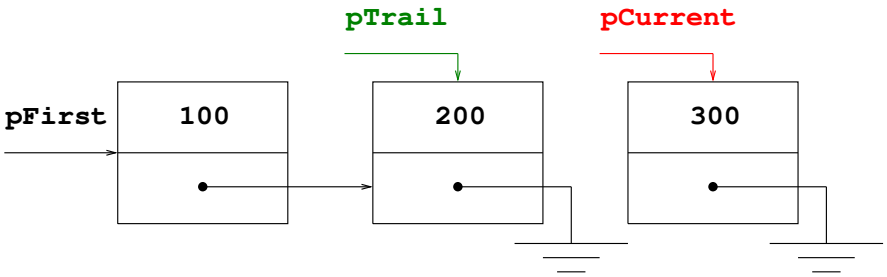
```
1  nodeStructType *Delete_LastNode(nodeStructType *pFirst)
2  {
3      nodeStructType *pTrail;
4      nodeStructType *pCurrent;
5
6      if (pFirst == NULL) /* list is empty */
7         printf("List is empty. No node to delete.\n");
8      else {
9         pTrail = NULL;
10        pCurrent = pFirst;
11        while (pCurrent->pNext != NULL) {
12            pTrail = pCurrent;
13            pCurrent = pCurrent->pNext;
14        }
15        if (pCurrent == pFirst) /* only one node in the list */
16            pFirst = NULL;
17        else /* at least two nodes in the list */
18            pTrail->pNext = NULL;
19        free(pCurrent);
20     }
21     return pFirst;
22 }
```



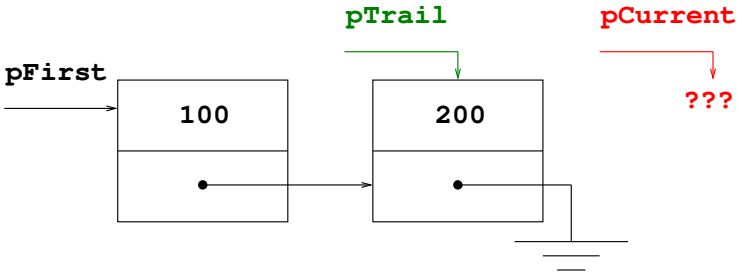
(a) Linked list after executing `pTrail = NULL; pCurrent = pFirst;`



(b) Linked list immediately after right after the `while()` loop



(c) Linked list after executing `pTrail->pNext = NULL`



(d) Linked list after executing `free(pCurrent)`

Figure 5.9: Steps for deleting the last node from a non–empty linked list

5.14 Insertion Sort

We discuss in this section a sorting algorithm called *Insertion Sort* implemented using linked lists.[5] Nodes are added to, and deleted from the linked list while maintaining a sorted order based on the `key` value of each node. We assume in the following discussions that each node has a unique `key` value.

5.14.1 Adding Nodes Into a Sorted List

The Insertion Sort algorithm for building a linked list with nodes containing `key` values in increasing order from the first node down to the last node is as follows.

1. Initialize the linked list as an empty list.
2. Input the value of the `key` and the other `data` members (note: we assume that all key values are unique). If there is no more input, STOP.
3. Create a new node and initialize its members. (Refer to Section 5.7).
4. Determine where to insert the new node within the sorted list (note: this step is implemented inside a loop).
5. Insert the new node in the list. There are three possible cases, namely: (a) insert it as the first node, (b) insert it as the last node, and (c) insert it in between two existing nodes in the list.
6. Repeat from step 2.

The details for steps 3 to 5 are implemented in function `Insert_Sorted()` which is shown in Listing 5.17. The function accepts two parameters: (i) `pFirst`, and (ii) the `data` value to be assigned to the new node. It returns the pointer to the updated list.

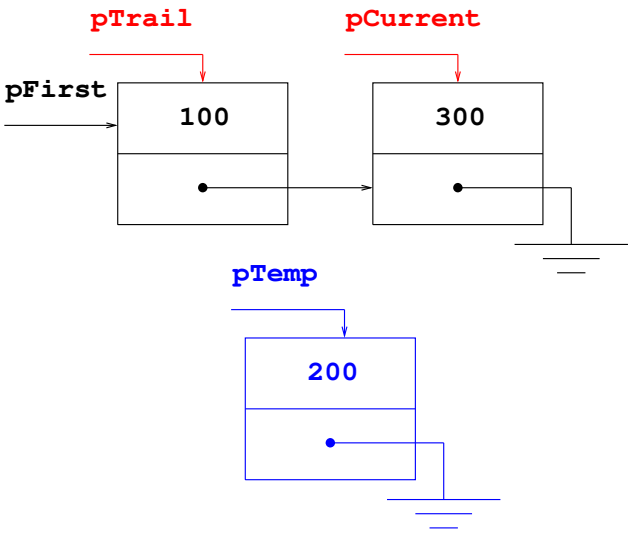
IMPORTANT NOTE: It is assumed that the node containing the new key value does not yet exist in the sorted linked list prior to calling `Insert_Sorted()`.

Figure 5.10 shows an example scenario for addition of a new node in between two existing nodes. Refer to previous figures for addition of nodes as the first (see Figure 5.6) or last node (see Figure 5.7) of the list.

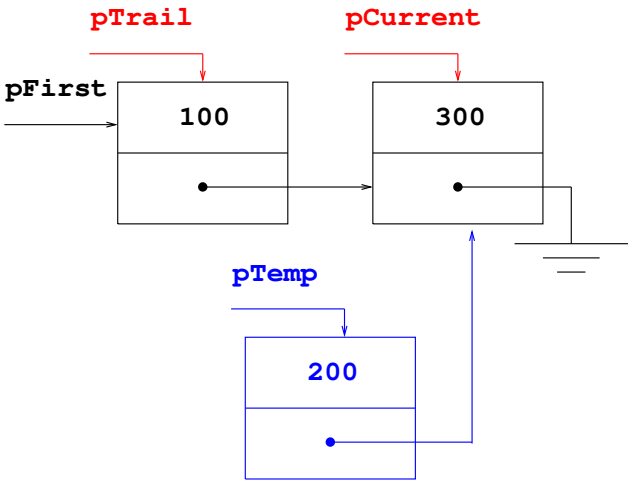
[5]There is also an implementation of Insertion Sort for arrays. It will be discussed in your future subject on Data Structures and Algorithms (course code: DASALGO).

Listing 5.17: Insertion of a New Node in a Sorted Linked List

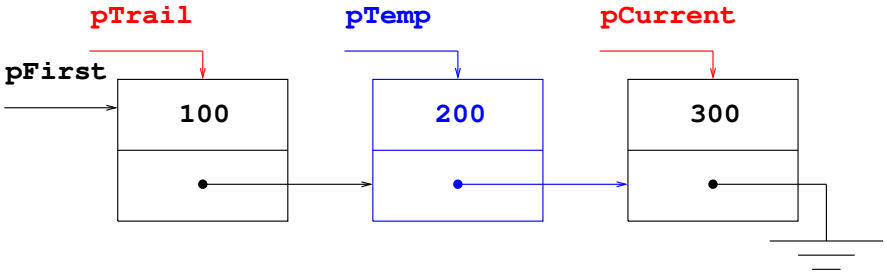
```
1  nodeStructType *Insert_Sorted(nodeStructType *pFirst,
2                                dataStructType data)
3  {
4      int newkey;
5      nodeStructType *pTemp;
6      nodeStructType *pTrail;
7      nodeStructType *pCurrent;
8
9      pTemp = CreateNewNode(data);
10
11     /* determine where to insert the new node */
12     if (pFirst == NULL)
13         pFirst = pTemp; /* insert new node as first node */
14     else { /* there is at least one node */
15         newkey = data.key;
16         pTrail = NULL;
17         pCurrent = pFirst;
18
19         while (pCurrent != NULL) {
20            /* compare new key with key of the current node */
21            if (newkey < pCurrent->data.key)
22               break; /* we found where to insert new node! */
23            else {
24                /* newkey > pCurrent->data.key so we move
25                   to the next node */
26                pTrail = pCurrent;
27                pCurrent = pCurrent->pNext;
28            }
29         }
30
31         pTemp->pNext = pCurrent;
32         /* note: new node is inserted as last node
33                  if pCurrent is NULL */
34
35         if (pTrail != NULL)
36            pTrail->pNext = pTemp; /* insert in between */
37         else
38            pFirst = pTemp; /* insert as first node */
39      }
40      return pFirst;
41 }
```



(a) Linked list before adding new node in between two nodes



(b) Linked list after executing `pTemp->pNext = pCurrent`



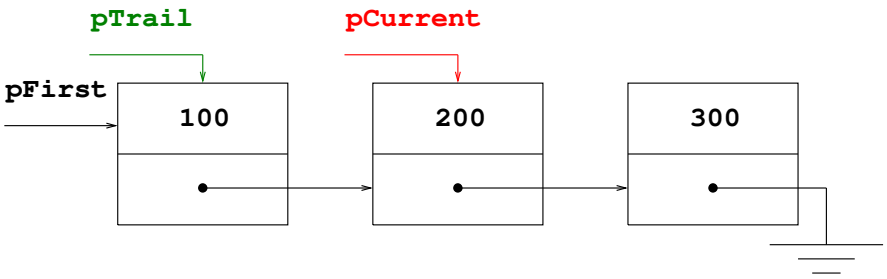
(c) Linked list after executing `pTrail->pNext = pTemp`

Figure 5.10: Steps for adding a node in between two nodes

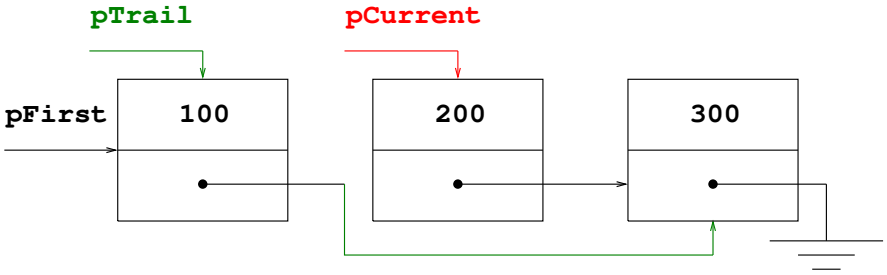
5.14.2 Deleting Nodes From a Sorted List

The algorithm for deleting a node from a sorted list is as follows:

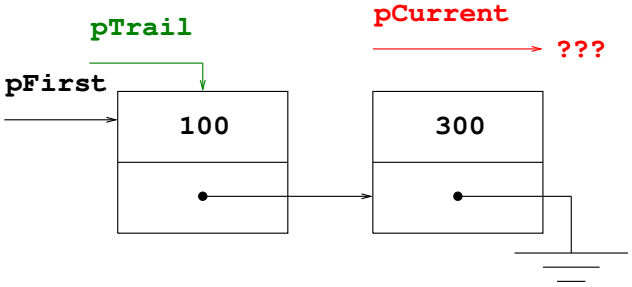
1. Input the `key` value of the node that will be deleted.
2. Search the node that contains the key value. If it exists, delete it. There are three possible cases, namely: (a) delete the first node, (b) delete the last node, and (c) delete a node that exists in between two other nodes.



(a) Linked list before deleting a node in between two nodes



(b) Linked list after executing `pTrail->pNext = pCurrent->pNext`



(c) Linked list after executing `free(pCurrent)`

Figure 5.11: Steps for deleting a node from between two nodes

The details of the algorithm are implemented in function `Delete_Sorted()` which is shown in Listing 5.18. The function accepts two parameters namely, (i) `pFirst`, and (ii) the `key` value of the node to be deleted. It returns the pointer to the updated list.

Listing 5.18: Deletion of a Node from a Sorted Linked List

```
1  nodeStructType *Delete_Sorted(nodeStructType *pFirst, int key)
2  {
3       int found; /* Boolean: false is 0, true is 1 */
4       nodeStructType *pTrail;
5       nodeStructType *pCurrent;
6
7       if (pFirst == NULL)
8           printf("Empty list, no node to delete.\n");
9       else { /* there is at least one node */
10          pTrail = NULL;
11          pCurrent = pFirst;
12          found = 0; /* not found */
13          while (pCurrent != NULL && !found) {
14               if (key == pCurrent->data.key)
15                  found = 1; /* key is in the list */
16               else if (key > pCurrent->data.key) {
17                        pTrail = pCurrent;
18                        pCurrent = pCurrent->pNext;
19                    }
20               else /* key < pCurrent->data.key */
21                   break; /* key is not in the list */
22          }
23
24          if (found) { /* we delete the node */
25             if (pTrail == NULL) /* it's the first node */
26                 pFirst = pFirst->pNext;
27             else
28                 pTrail->pNext = pCurrent->pNext;
29
30            free(pCurrent);
31          }
32          else /* !found */
33              printf("Key %d is not in the list.\n", key);
34      }
35      return pFirst;
36 }
```

Figure 5.11 shows a representative example for deleting a node that exists between two other nodes. Refer to previous figures for deletion of the first or last node from the list.

▶ Self–Directed Learning Activity ◀

1. Download the program named `chap5_prog1.c` and the header file `nodetype.h` from the CCS1 COMPRO2 download site. Study first the contents of the program; i.e., how the functions are defined, the logic involved in each function definition and how the functions are called.

   Compile and run the program. Experiment with the options for adding a new node (as the first or last node in the list) and deleting the first or last node from the list. It is recommended that you invoke the traverse option after addition or deletion of nodes to see the effect on the list.

   Make sure that you fully understand the logic of the functions. The learning activity here is for you to CREATE YOUR OWN set of functions which will also have the same logic. It is very important that you develop the capability to recreate functions based on your understanding of the logic or the algorithm.

2. Download the program named `chap5_prog2.c`. The program implements the routines for adding new nodes and deleting nodes in a sorted list. Compile, run and test the functionalities of the program. As in the previous activity, CREATE YOUR OWN version of the functions with the same logic.

3. Download a visualization program for different data structures, including linked lists, from the following site: `http://www.cs.usfca.edu/~galles/visualization/download.html`. The downloadable file is actually an executable Java (i.e., jar file). Run the program by double clicking on the icon representing the file. Select the option "Algorithms" from the main menu option, and then select "List/Stacks/Queues" from the submenu options. Thereafter left–click on the right most tabbed corresponding to "List Linked List". You will see in the center of a window a visualization for a dummy node with two pointers named "Head" and "Tail". Initially both these pointers point to the dummy node. There is also a variable named "Length" that indicates the number of nodes in the list. Initially, the value is 0 (note that the dummy node is not counted).

   (a) Try the following: type an integer value (for example, 10) in the space before the button named "add". Thereafter, left–click the "add" button. Notice that the program will show you an animation of how a new

node with a data value of 10 will be inserted into the list. Notice that the "tail" pointer will then be adjusted to point to the newly inserted node. Notice also that the "Length" counter is incremented by 1. Next, try adding 20, and see the animation. Try adding a few more nodes. Based on what you observed, describe how the visualization program adds a new node.

(b) There are other things that you can do with the list. Try clicking on the button name "Create Iterator". Notice that there will be new buttons in green color. Note also that a new pointer denoted as "previous" in green color appears. Initially, it points to the dummy node. The node after the node pointed bty the "previous" pointer is the "next" node. Click on the the "next" button to see what will happen. Try clicking next a few more times. Describe the effect of clicking the "next" button. What will happen if the "previous" is pointing to the last node, and you clicked on "next"?

(c) To get back the dummy node, click on the "first" button. Click the "Delete" button. Describe what is the effect on the list. Click the "next" button once, and then click "Delete" once. Describe what happened. Describe (i.e., enumerate the necessary steps on) how to delete a node using the visualization program. Try to delete the remaining (non-dummy) node.

(d) Add new nodes in the list. Position the "previous" pointer to the node prior to the last node in your list. On the space between the "Insert", type a new integer value, and then click "Insert". Describe what happened. Describe how you can insert a new node in between two existing nodes based on the visualization program.

5.15 Chapter Summary

The key ideas presented in this chapter are summarized below:

- A linked list is a group of nodes. Each node is composed of a data member and a number of links. A single linked list is a list of nodes containing one link each which points to the next node in the list.
- Algorithms must be formulated for adding and deleting new nodes in the list.
- It is necessary that we understand how to properly manipulate `pFirst` and the links of each node to maintain the link list.

- *Insertion Sort* is an algorithm for building a list sorted based in terms of the `key` member. The list can either be in ascending or descending order.

Problems for Chapter 5

Problem 5.1. Implement `dataStructType Get_FirstData(nodeStructType *pFirst)` which will return the value of the `data` member of the first node.

Problem 5.2. Implement `dataStructType Get_LastData(nodeStructType *pFirst)` which will return the value of the `data` member of the last node.

NOTE: *Review first the function* `int CountNodes(nodeStructType *pFirst)` *given as an exercise in page 151.*

Problem 5.3. Implement `dataStructType Get_Data(nodeStructType *pFirst, int num)` which will return the value of the `data` member of node number `num`. For example, the function call `Get_Data(pFirst, 3)` will return the `data` member of the third node in the list. NOTE: Assume that `num` is a valid value, i.e., it is an integer between `0` to `CountNodes(pFirst)` inclusive.

Problem 5.4. Implement `nodeStructType *Get_Node(nodeStructType *pFirst, int num)` which will return the pointer to node number `num`. For example, `Get_Node(pFirst, 3)` will return the pointer to the third node in the list. Make the same assumption about `num` as in the previous exercise.

Problem 5.5. Implement `int Sum(nodeStructType *pFirst)` which will return the sum of the key values in the linked list. For example, if there are three nodes containing key values of `100, 200,` and `300` respectively, `Sum(pFirst)` will return `600`. If the linked list is empty is should return `0`.

Problem 5.6. Implement `double Average(nodeStructType *pFirst)` which will return the average of the key values in the linked list. For example, if there are three nodes containing key values of `100, 200,` and `300` respectively, `Average(pFirst)` will return `200.0`.

Problem 5.7. Implement `nodeStructType *Minimum(nodeStructType *pFirst)` which will return the pointer to the node that contains the minimum (smallest) key value. The function should return `NULL` if the list is empty. Assume that no two nodes contain the same key value.

Problem 5.8. Implement `nodeStructType *Maximum(nodeStructType *pFirst)` which will return the pointer to the node that contains the maximum key value.

The function should return `NULL` if the list is empty. Assume that no two nodes contain the same key value.

Problem 5.9. Implement `nodeStructType *Search_Unsorted(nodeStructType, int key)` which will return the pointer to the node whose key value is the same as that of parameter key. If no such node exist or if the list is empty, the function should return a `NULL` value. For example, assume that there are five nodes containing key values of `100, 200, 175, 80,` and `260` respectively. The function call `Search_Unsorted(pFirst, 200)` will return the pointer to the node containing a key of `200`. On the other hand, function call `Search_Unsorted(pFirst, 125)` will return `NULL`. Assume that the list is NOT sorted.

Problem 5.10. Implement `nodeStructType *Search_Sorted(nodeStructType *pFirst, int key)`. It should behave as the search function in the previous problem but assume that in this case the list is sorted. The sample keys therefore will be arranged as `80, 100, 175, 200, 260`. Your algoithm/implementation takes advantage of the apriori knowledge that the list is sorted. For example, the function call `Search_Sorted(pFirst, 90)` should not visit or check the nodes containing `175, 200` and `260` after visiting the node containing `100`.

Problem 5.11 Implement `int IsSortedAscending(nodeStructType *pFirst)` that will return `1` if the nodes in the linked list are in ascending order by their key values; otherwise it should return a `0`. Example #1: if a linked list has three nodes containing `100, 200` and `300` respectively, the function will return a value of `1`. Example #2: if a linked has three nodes containing `100, 300,` and `200` respectively, the function will return a value of `0`. Assume in this problem that the linked list is not empty and that no two nodes contain the same key value.

Problem 5.12 Implement `nodeStructType *Insert_Sorted_Descending (nodeStructType *pFirst, dataStructType data)` which will add a new node containing `data` in a list sorted in descending order by key. Refer to Section 5.14.1.

Problem 5.13 Implement `nodeStructType *Delete_Sorted_Descending (nodeStructType *pFirst, int key)` which will delete a node whose `data.key` member is equal to parameter `key`. Refer to Section 5.14.2.

Problem 5.14. Implement `nodeStructType *CopyList(nodeStructType *pFirst)` which will create a copy of the linked list. The function returns a pointer to the first node of the copy. For example, let the original linked list contain three nodes with key values of `100, 200` and `300` respectively. The function call `pCopy = CopyList(pFirst)` will result into a new link list also with three nodes containing key values of `100, 200,` and `300` respectively. The pointer to the first node of the copy is `pCopy`. Note: you will need to call `malloc()` to create new

nodes for the copy.

Problem 5.15. Implement `nodeStructType *CopyList_Reversed (nodeStructType *pFirst)` which is basically the same as in the previous problem except that the copy contains nodes with key values in reversed sequence. For example, let the original list contains three nodes with key values of `100, 200,` and `300` respetively. The resulting copy will have a first node containing `300` and a last node containing 100 as key values respectively.

Problem 5.16. Let there be two linked lists. The pointer to the first list is `pList1` and the pointer to the second list is `pList2`. Implement `int Longer(nodeStructType *pList1, nodeStructType *pList2)` which will determine which of the two list is longer, i.e., with more nodes. If the first list is longer than the second the function should return a `1`. If the second is list longer than the first, the function should return a `2`. Otherwise, if the lists have the same number of nodes, the function should return a `0`. Hint: your implementation should call the function `CountNodes()`.

Problem 5.17. Assume again two linked list as in the previous problem, with the additional assumption that they have the same number of nodes. Implement `int Equal(nodeStructType *pList1, nodeStructType *pList2)` which will determine if all the keys in the respective nodes of the list are equal or not. If they are equal, the function should return a `1` otherwise it should return a `0`.

Problem 5.18. Implement `nodeStructType *Purge(nodeStructType *pFirst, int x)` which will delete all nodes with key values less than parameter `x`. For example, let the linked list contain five nodes with key values of `100, 75, 300, 40, 500` respectively. After calling `pFirst = Purge(pFirst, 100)` the linked will contain only three nodes with keys `100, 300` and `500` respectively. The nodes with keys of `75` and `40` were deleted from the list.

Problem 5.19. Implement `nodeStructType *RotateRight(nodeStructType *pFirst)` which will rotate the nodes of a linked list *once* towards the right. For example, assume a linked with three nodes containing keys `100, 200` and `300` respectively. After calling `pFirst = RotateRight(pFirst)`, the linked list have the original third as its first node, the original first node as its second node, and the original second node as its last node, i.e., the nodes are in the sequence `300, 100, 200`. Restrictions: (i) Make sure that your algorithm adjusts the related links, not the data member. (ii) New nodes must not be created, i.e., do not call `malloc()` inside the function definition. (iii) The original last node SHOULD NOT be deleted, its link should be modified.

Problem 5.20. Implement `nodeStructType *RotateLeft(nodeStructType`

`*pFirst)`. which will rotate the nodes once towards the left. For example, if the original list contains nodes with key values `100, 200, 300`, then the rotated list will contain nodes in the order `200, 300` and `100` respectively. Restrictions similar to the previous problem should be observed.

Problem 5.21. Implement `nodeStructType *SwapNodePairs(nodeStructType *pFirst)` which will swap the positions of pairs of nodes in the linked list. Assume that the linked list is non–empty and that there are always an even number of nodes. For example, let there be 6 nodes containing the key values `100, 200, 300, 400, 500` and `600` respectively. After calling `pFirst = SwapNodePairs(pFirst)` the link list will now contain nodes with key values rearranged as `200, 100, 400, 300, 600` and `500`. Note that your algorithm or solution should manipulate the links of the nodes, not the `data` member of the nodes.

References

Consult the following resources for more information.

1. comp.lang.c Frequently Asked Questions. `http://c-faq.com/`

2. Kernighan, B. and Ritchie, D. (1998). *The C Programming Language, 2nd Edition*. Prentice Hall.

Chapter 6

File Processing

6.1 Motivation

Several programs that we have written required data to be read via the standard input device and store the values to variables associated with contiguous bytes in the primary memory. An example of an input instruction is `scanf("%d", &n);` which requires the user to input the value of an integer variable named as `n`.

There are several practical problems that would require programs to read data NOT from the standard input device but from a file in the secondary memory.[1] One reason why data have to be read from a file is that it could be be impractical for the user to input (a lot of) values interactively and repetitively such as in the following code snippet

```
for (i = 0; i < 10000; i++)
     scanf("%d", &A[i]);
```

Another reason is that we need to save data from the primary memory into secondary memory because the primary memory is volatile. For example, let us assume that we are typing a research paper that would require several lines, paragraphs and pages. We would normally use a word processor to create and edit the document. At some point in time, we would choose a functionality such as `"File"` followed `"Save"` save the current document that we have edited as a file in the secondary memory. Later on, this file can be opened (i.e., `"File"`, `"Open"`), and the contents will be opened and copied onto the primary memory.

[1] Secondary storage devices include hard disk, mobile disk, DVD, etc.

In this chapter we will learn how to read data from a file, and how to write data to a file. We will consider both text and binary files.

6.2 What is a File?

A file is a group of elements. The elements are essentially a collection of data stored in memory, specifically, in the secondary memory, for example in the hard disk or mobile disk. There are basically two types of files, namely: *text* file and *binary* file.

C language does not actually have keywords or commands for handling input/output. Instead, there are several library functions which are part of what is now known as standard input/output library. The function prototypes are declared inside `stdio.h` file. In this chapter, we will learn several library functions that will allows us to do the following: (i) open a file, (ii) close a file, (iii) create a new file, (iv) read data from an existing file, (v) write data to a new file and (vi) append data to an existing file. Details are provided in the following sections.

6.3 Text File

A *text file* is a file containing elements all of type `char`. These characters are encoded in a standard format such as ASCII (American Standard Code for Information Interchange) or UNICODE (for international encoding).[2]

Text files can be opened by text editors such as `edit`, `Notepad` or `vi`.[3] The contents of a text file can also be displayed in the console by commands such as `type <filename>` in Windows command line or `cat <filename>` in UNIX and Linux.

Example: Assume a text file `"SAMPLE.TXT"` containing two lines of text as follows:

```
1 APPLE
2 BANANAS
```

The characters in the text file are listed in Table 6.1. Notice the presence of the newline character after letter 'E'.

[2] Different cultures use different characters in their written languages.

[3] `Edit` and `Notepad` are text editors in Windows OS and `vi` is an editor in UNIX and Linux.

Table 6.1: Example Characters and Their ASCII Codes

Character	ASCII Code
'1'	49
' '	32
'A'	65
'P'	80
'P'	80
'L'	76
'E'	69
newline	10
'2'	50
' '	32
'B'	66
'A'	65
'N'	78
'A'	65
'N'	78
'A'	65
'S'	83

Question: How do we know when/where the text file ends? That is, how do we know that we have already reached the last character in a text file?

Answer: There is a marker with a macro name of `EOF` which denotes the end of the file. `EOF` is an abbreviation of **E**nd **o**f **F**ile. In the example above, the `EOF` appears after character 'S'.

It is very important to note that the '1' stored in the text file is a character encoded as ASCII code 49. It is not numeric 1 of type integer.

▶ Self–Directed Learning Activity ◀

Determine what is the value of `EOF`. Hint: Write a C program that will print the value of `EOF`.

6.4 File Pointer Declaration

A *file pointer* is a pointer that will be mapped to an external file uniquely specified by its filename and extension. The syntax for file pointer declaration is:

```
FILE *<pointer name>;
```

Note that `FILE` is not a keyword in the C programming language. It is actually a type definition for a structure used in storing information about a file.

Examples of file pointer declarations are:

```
FILE *fp;   /* fp is taken to mean file pointer */

FILE *fp_input, *fp_output;
/* fp pointers to input file and output files */

FILE *fp1, *fp2, *fp3;
```

6.5 Opening and Closing a File

6.5.1 Opening a File

The first operation that should be done on files is to open it. The pre–defined library function for opening a file is `fopen()` with the following prototype:

```
FILE *fopen(<filename>, <mode>);
```

`fopen()` returns a pointer to the file (data type of `FILE *`)if the file exists; otherwise, it returns `NULL`. It has two parameters, namely:

- `<filename>` is a string indicating the filename and extension (in some cases, we may need to specify the relative or the absolute path) of the file
- `<mode>` is a string indicating the mode in which the file will be opened. There are several modes; we will limit our discussions to the following:
  - `"r"` open the file in read mode (i.e., input from the file)

- `"w"` open the file in write mode (i.e., output to a file); if the file exists, the original contents will be overwritten
- `"a"` open the file in append mode (i.e., output to a file); adds new entries at the end of the original file
- `"b"` this letter is added after `"r"` or `"w"` to indicate the fact that the file to be manipulated is a binary file.

Simple examples of using `fopen()` are shown in the following codes:

```
/* open a text file named "input.txt" in read mode */
fp_input = fopen("input.txt", "r");

/* open a text file named "output.txt" in write mode */
fp_output = fopen("output.txt", "w");

/* open a text file named "output.txt" in read mode */
/* file can be found in c:\data subdirectory */
fp1 = fopen("c:\data\sample.txt", "r");

/* open text file named "test.txt" in append mode */
fp2 = fopen("test.txt", "a");

/* open a binary file named "sample.dat" in read mode */
fp3 = fopen("sample.dat", "rb");
```

6.5.2 Closing a File

The last operation that should be performed on a file is to close it. The function prototype for this is:

```
fclose(<file pointer>);
```

WARNING: Make sure to close all files that were opened. It is possible that data maybe lost if the associated file is not closed. Some operating system will automatically close all files after a program's termination, but we should not depend on this behavior. Only one file at a time can be closed in one invokation of `fopen()`. Thus, if there are m number of files that need to be opened, then `fopen()` will need to be called m number of times. Examples of how to close (previously opened) files are given below:

```
/*example shows how to close three files */
fclose(fp);
fclose(fp_input);
fclose(fp_output);
```

ILLUSTRATIVE EXAMPLES

Listing 6.1 shows an example program that demonstrates what we learned so far, (i) how to declare a file pointer, (ii) how to open a file named as `"test.txt"` in read mode and later after performing some other tasks (iii) how to close the file. If the file does not exist in the same directory as the executable program, `fopen()` will return a `NULL` value. Otherwise, if the file exists, the address of the file will be stored in file pointer `fp`. All other file related operations will have the file pointer as a parameter (see sample programs in the latter sections).

Listing 6.1: `fopen()` with `"r"` mode sample program

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      FILE *fp;
7
8      fp = fopen("test.txt", "r");  /* open in read mode */
9
10     if (fp == NULL) {
11        printf("ERROR: file does not exist.\n");
12        exit(1);
13     }
14     else
15        printf("File opened successfully.\n");
16
17     /*-- other statements follow --*/
18
19     fclose(fp);  /* close the file */
20     return 0;
21 }
```

▶ Self–Directed Learning Activity ◀

1. Encode and compile the program in Listing 6.1. Test it for two possible cases:

(a) the file `"test.txt"` is not in the same directory as the executable file

(b) the file `"test.txt"` is present in the same directory.

2. It is possible to open a file that is not in the same directory as the executable file. As an experiment, do the following:

   (a) First, create a new directory named `MYDIR` in drive `D`. At this point in time, the subdirectory `MYDIR` is empty.

   (b) Next, modify `fopen()` command into `fopen("D:/MYDIR/test.txt", "r");`.

   (c) Compile and test the program. What is the result?

   (d) Next, make sure that there is a `test.txt` file in the `MYDIR` subdirectory. You may create a new file using a text editor. Alternatively, you may just move or copy an existing file into the said subdirectory.

   (e) Run the program again. What is the output?

Instead of hardcoding the filename, i.e., the first parameter of `fopen()`, we can ask the user to input it via `scanf()`. Listing 6.2 illustrates this technique.

Listing 6.2: More generalized filename

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      char filename[40];    /* filename and extension */
7      FILE *fp;
8
9      printf("Input filename: ");
10     scanf("%s", filename);
11
12     if ( (fp = fopen(filename, "r")) == NULL) {
13        printf("ERROR: %s does not exist.\n", filename);
14        exit(1);
15     }
16     else
17        printf("%s was opened successfully.\n", filename);
18
19     /*-- other statements follow --*/
20
21     fclose(fp);
22     return 0;
23 }
```

▶ Self–Directed Learning Activity ◀

Encode and compile the program in Listing 6.2. Test it for the following cases:

1. a file that does not exist
2. a file that is in the same directory as the executable file
3. a file that is in a different drive/directory

Listing 6.3 shows how to open a file named as `"sample.txt"` in write mode. Take note that the `"w"` mode is a destructive mode. This means that if a file already exists, the contents of the said file will be erased and will be overwritten.[4]. If it does not exist, a new file with the filename of `"sample.txt"` will be created.

Listing 6.3: `fopen()` with `"w"` mode sample program

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      FILE *fp;
7
8      fp = fopen("sample.txt", "w");  /* open in write mode */
9
10     /*-- other statements follow --*/
11
12     fclose(fp);
13     return 0;
14 }
```

▶ Self–Directed Learning Activity ◀

1. Encode and compile the program in Listing 6.3. Test it for the following cases:
   (a) the file does not exist
   (b) the file exists and contains some texts
2. What is the size of the file in the first case? What happened with the original file contents in the second case?

[4]Be careful! You may accidentally erase the contents of an important file if you incorrectly specified `"w"` instead of `"r"`.

A file can be opened and closed more than once within the same program or even within the same function. The following code snippet shows an example:

```
    FILE *fp;

    fp = fopen("test.txt", "r");   /* open 1st time */

    /*-- other statements --*/

    fclose(fp); /* close 1st time */

    /*--- then later on --*/

    fp = fopen("test.txt", "r");   /* open 2nd time */

    /*-- other statements --*/

    fclose(fp);                    /* close 2nd time */
}
```

Several files maybe opened, possibly in different modes, within the same program or even within the same function. For example:

```
    FILE *fp1, *fp2;

    fp1 = fopen("abc.txt", "r");  /* open first file */
    fp2 = fopen("def.txt", "w");  /* open second file */

    /*-- other statements --*/

    fclose(fp1);  /* close the files */
    fclose(fp2);
```

6.6 Formatted File Output

We are already familiar with the formatted output library function `printf()` which displays the output on the standard output device (display screen). `printf()` can be thought of as a specific case of the more generalized formatted output library function named `fprintf()`. Its function prototype is:

```
int fprintf(<file pointer>, <format string>, <expression>);
```

We first prove that `printf()` is just a specific case of `fprintf()` using Listing 6.4. To do this, we will need to use a predefined pointer `stdout` as the first parameter to `fprintf()`. `stdout` means standard output which in our case refers to the display screen. In C, it is automatically opened (we do not need to call `fopen()` with `stdout` and can be readily used in any part of the program. Everything that we did before using `printf()` can be done using `fprintf(stdout, ...)`.

Listing 6.4: `fprintf()` with `stdout`

```
1  #include <stdio.h>
2
3  int main()
4  {
5      /* no need to call fopen() with stdout */
6
7      fprintf(stdout, "Hello world!\n");
8
9      /* no need to call fclose() with stdout */
10     return 0;
11 }
```

▶ Self–Directed Learning Activity ◀

1. Encode, compile and test the program in Listing 6.4.
2. Add the following statements after `fprintf()` in Listing 6.4. Run and test the program. What are the results?

```
fprintf(stdout, "%d\n", 123);
fprintf(stdout, "%f\n", 6.3745674123456);
fprintf(stdout, "%.2f\n", 6.3745674);
```

The true purpose of `fprintf()` is to output/write data to be stored in secondary memory. Listing 6.5 shows how we can output the values of variables declared using the simple data types `char`, `int`, `float` and `double` into a named text file. It illustrates the four basic steps involved in text file output, namely:

1. Declare a file pointer.
2. Open the file for output using `fopen()` in write mode, that is, the second parameter is `"w"`.
3. Output data to the file using `fprintf()`. Note that `fprintf()` maybe called any number of times as necessary after `fopen()` and before `fclose()`.
4. Close the file using `fclose()`.

Listing 6.5: Output Basic Data Type Values

```
1  #include <stdio.h>
2  int main()
3  {
4      char ch = 'A';
5      int i = 123;
6      float f = 4.0;
7      double d = 3.1415159;
8
9      FILE *fp;
10
11     fp = fopen("sample.txt", "w");
12     fprintf(fp, "%c %d %.2f %lf", ch, i, f, d);
13     fclose(fp);
14     return 0;
15 }
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the program in Listing 6.5. How many characters do you think were written onto the text file?
2. Go to the command line, and change your subdirectory to the directory containing the `exe` file. What are the results for the following commands?
   (a) `dir sample.txt`
   (b) `type sample.txt`

Listing 6.6 is a program that illustrates how to output a sequence of numbers from 0 to 4 generated using a `for` loop to a file. We will compare the result of this program later with another program discussed in the section related to binary files.

Listing 6.6: Output Basic Data Type Values

```
1  #include <stdio.h>
2
3  int main()
4  {
5      char filename[40];
6      int i;
7      FILE *fp;
8
9      fprintf(stdout, "Input filename: ");
10     fscanf(stdin, "%s", filename);
11
12     fp = fopen(filename, "w");
13
14     for (i = 0; i < 5; i++)
15         fprintf(fp, "%d\n", i);
16
17     fclose(fp);
18     return 0;
19 }
```

6.7 Formatted File Input

We are already familiar with the formatted input library function `scanf()`. The counterpart function for formatted file input is the library function `fscanf()`. Its function prototype is as follows.

```
int fscanf(<file pointer>, <format string>, <address>);
```

Similar to the previous section, we first prove that `scanf()` is just a specific case of `fscanf()` using Listing 6.7. To do this, we will need to use a predefined name `stdin` as the first parameter to `fscanf()`. `stdout` means standard input device which in our case refers to the keyboard. In C, it is automatically opened (we do not need to call `fopen()` with `stdin` and can be readily used in any part of the program. Everything that we did before using `scanf()` can be done using `fscanf(stdin, ...)`.

Listing 6.7: `fscanf()` with `stdin`

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int n;
6
7      /* no need to call fopen() and fclose() with stdin and stdout */
8      fprintf(stdout, "Input an integer value: ");
9      fscanf(stdin, "%d", &n);
10     fprintf(stdout, "n = %d\n", n);
11     return 0;
12 }
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the program in Listing 6.4.
2. Modify the program by:
   (a) asking the user to input the value of a float variable and a double data type variable; provide prompts using `fprintf()`.
   (b) afterwards, output the values entered by the user using `fprintf()`.

The input in the previous program come from the standard input device `stdin`. How we can get the input if data are stored in a secondary memory? Listing 6.8 shows an example of how to read data from a named text file using `fscanf()`.[5]

The program illustrates the four basic steps involved in text file input, namely:

1. Declare a file pointer.
2. Open the file for input using `fopen()` in read mode, that is, the second parameter is `"r"`.
3. Input data from the file using `fscanf()`. Note that `fscanff()` maybe called any number. of times as necessary after `fopen()` and before `fclose()`.
4. Close the file using `fclose()`.

[5]In order for the program to work, the file `"sample.txt"` with the correct contents must exist. There will be a logical/run–time error if `"sample.txt"` is not found, or if the contents are incorrect. To ensure that this will not happen, run first the program in Listing 6.5.

Listing 6.8: Read data from a text file

```
#include <stdio.h>

int main(void)
{
      char ch;
      int i;
      float f;
      double d;
      FILE *fp;

      /* open input file */
      fp = fopen("sample.txt", "r");

      /* read data from the file */
      fscanf(fp, "%c %d %f %lf", &ch, &i, &f, &d);

      /* write data to stdout */
      fprintf(stdout, "ch = %c, i = %d, f = %f, d = %lf\n",
                      ch, i, f, d);

      /* close input file */
      fclose(fp);
      return 0;
}
```

▶ Self–Directed Learning Activity ◀

1. Encode, compile and test the program in Listing 6.8. What are the results when you run the program?

2. Modify the contents of `"sample.txt"` using an editor as Notepad. Re–run the program; what are the corresponding results?

6.8 How to Read ALL Data From the Input File

The previous program reads only four data values from the input file. It is possible that there are more data.

Many problems involving file processing would have to read ALL data from the input file. We will give in the following subsections several example problems and their corresponding techniques on how to solve them.

6.8.1 A program similar to the `type` command

The `type` command that we issue in Windows command line reads all the characters stored in the text file and outputs them on the screen. The algorithm for such a program is as follows:

1. Input the name of the file.
2. Open the file in `"r"` mode. Let us assume that it exists.
3. Read one character from the file using `fscanf()`.
4. If the return value of `fscanf()` is 1 it means that `fscanf()` successfully read one character, i.e., it is not yet the end of file.
   (a) Display the character on the screen.
   (b) Repeat step 3.
5. If the return value of `fscanf()` is not 1 it means that we have reached the end of file. Close the file.

The corresponding program is shown Listing 6.9. The technique uses a `fscanf()` inside a `while` loop to read all characters stored in the text file. The loop condition involves two operations actually, (i) read one character from the file using `fscanf()`, (ii) compare the return value of `fscanf()` with 1. A return value of 1 means that `fscanf()` successfully read one character from the file; otherwise, it means that there are no more characters and that we have reached the end of file.[6]

[6] Actually, there is a pair of pre–defined functions dedicated to single character file I/O. These are `fgetc()` and `fputc()`. We chose not to discuss them in these course notes. It is suggested that you consult the references if you are interested in learning more about these functions.

Another new concept introduced in the sample program is the use of the pre–defined name `stderr` which denotes the standard error device. This is the device that should be used when we want to display an error. The standard error device corresponds to the display screen in the platform that we are using.

Listing 6.9: A program similar to the `type` command

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      char filename[40];
7      char ch;
8      FILE *fp;
9
10     fprintf(stdout, "Input filename: ");
11     fscanf(stdin, "%s", filename);
12
13     if ((fp = fopen(filename, "r")) == NULL) {
14        fprintf(stderr, "ERROR: %s does not exist.\n", filename);
15        exit(1);
16     }
17
18     while ((fscanf(fp, "%c", &ch)) == 1)
19         fprintf(stdout, "%c", ch);
20
21     fclose(fp);
22     return 0;
23 }
```

▶ Self–Directed Learning Activity ◀

1. Encode, compile and test the program in Listing 6.9. Test for files that exist and that do not exist.

2. Run the program again. When asked for the filename, input the filename you used when you save Listing 6.9. What is the result?

6.8.2 A program similar to the `copy` command

The `copy` command copies the contents of one file to another file. Its syntax is `copy <source file> <destination file>`. The `copy` command works for both text files and binary files. We show how to implement a similar functionality but it will be limited to copying text files only.

The algorithm for `copy` shown below is very much similar to that for `type`. The difference here is that instead of an `fprintf()` to the `stdout`, we do an `fprintf()` to the destination file.

1. Input the name of the source file.
2. Open the source file. Let us assume it exists.
3. Input the name of the destination file.
4. Open the destination file. (Note: If the file exists, the original contents will be overwritten.)
5. Read one character from the source file using `fscanf()`.
6. If `fscanf()` return value is 1:
   (a) Write the character onto the destination file.
   (b) Repeat Step 5.
7. If `fscanf()` return value is not 1, close the files.

The implementation is shown in Listing 6.10.

Listing 6.10: A program similar to the `copy` command

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      char source_filename[40];
7      char dest_filename[40];
8      char ch;
9      FILE *fp_source;
10     FILE *fp_dest;
11
```

```
12      fprintf(stdout, "Input source filename: ");
13      fscanf(stdin, "%s", source_filename);
14
15      /* open source file */
16      if ((fp_source = fopen(source_filename, "r")) == NULL) {
17         fprintf(stderr, "ERROR: %s does not exist.\n",
18                 source_filename);
19         exit(1);
20      }
21
22      fprintf(stdout, "Input destination filename: ");
23      fscanf(stdin, "%s", dest_filename);
24
25      /* open destination file in "w" mode */
26      fp_dest = fopen(dest_filename, "w");
27
28      /* copy one character at a time from source to dest. */
29      while ((fscanf(fp_source, "%c", &ch)) == 1)
30          fprintf(fp_dest, "%c", ch);
31
32      fclose(fp_source);
33      fclose(fp_dest);
34      return 0;
35  }
```

6.8.3 A program that reads a sequence of numbers from a text file

Let us assume that a text file contains a sequence of integers (actually, the digits corresponding to these numbers are stored as chacters). Furthermore, we assume that each line contains only one integer. What we do not know is how many integers are stored in the file. Our problem is to create a program that will read all the integers from the file and display them on the screen one integer per line of output.

The solution to this problem is very much similar to the `type` algorithm. The difference is that instead of reading one character at a time, we read integers. The reader is advised to create first an algorithm before reading the next page.

The program corresponding to the requirement is shown in Listing 6.11.

Listing 6.11: A program that reads a sequence of numbers from a file

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      char filename[40];
7      int value;
8      FILE *fp;
9
10     fprintf(stdout, "Input filename: ");
11     fscanf(stdin, "%s", filename);
12
13     if ((fp = fopen(filename, "r")) == NULL) {
14        fprintf(stderr, "ERROR: %s does not exist.\n", filename);
15        exit(1);
16     }
17
18     while ((fscanf(fp, "%d", &value)) == 1)
19         fprintf(stdout, "%d\n", value);
20
21     fclose(fp);
22     return 0;
23 }
```

6.8.4 A program that reads a sequence of tuples from a text file

A "tuple" is one group of values which maybe of different types. For this example, let us assume that each tuple is composed of 4 values: (i) a string of at most 20 characters, (ii) one integer, (iii) one float and (iv) one double in the given sequence.

Assume that a text file contains data corresponding to a sequence of tuples. Tuples are stored per line, and values within the same tuple are separated by white spaces (blanks, tabs). For example:

```
PEDRO 1  2.5  3.1
MARIA 6  0.5  8.9
JUAN -4 1.8  3.7
```

Our task is read all the tuples from a named text file and output each tuple one line at a time on the display screen. Take note that we do not know in advance how many tuples are present.

The solution can be patterned from the algorithms in the previous subsections. We read the values for each tuple using `fscanf()`. A successful `fscanf()` should return a value equivalent to 4 which means that all 4 in a tuple were successfully read. In such a case, we output the tuple on the screen. If the `fscanf()` does not return a value equal to 4, then it means there are no more tuples, and therefore it is already the end of the file.

The solution is shown in Listing 6.12.

Listing 6.12: A program that reads a sequence of tuples from a file

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      char filename[40];
7      char name[20];
8      int i;
9      float f;
10     double d;
11
12     int count;
13
14     FILE *fp;
15
16     fprintf(stdout, "Input filename: ");
17     fscanf(stdin, "%s", filename);
18
19     if ((fp = fopen(filename, "r")) == NULL) {
20         fprintf(stderr, "ERROR: %s does not exist.\n", filename);
21        exit(1);
22     }
23
24     while ((fscanf(fp, "%s %d %f %lf", name, &i, &f, &d)) == 4)
25         fprintf(stdout, "%s %d %f %lf\n", name, i, f, d);
26
27     fclose(fp);
28     return 0;
29 }
```

The general rule that we should have learned from this example is that `fscanf()` returns the number of values that were successfully read from the file. If we are trying to read m number of values, and `fscanf()` returned m as a result, then we can conclude that it was a successful read; otherwise it was unsuccessful. An unsuccessful `fscanf()` will return a values less than m. This could mean two possibilities, (i) some values are missing (example, two values instead of the expected four values) or (ii) there are no more values, i.e., we have reached the `EOF`.

6.9 Opening a File in Append Mode

The last topic that we will cover in text file processing is how to open a file in append `"a"` mode. A text file opened in `"a"` mode has the following characteristics:

1. If the file does not exist yet, a new file will be created.
2. If the file exists, its contents are not overwritten (unlike in `"w"` mode). New data written to the file will be added (i.e., appended) towards the end of the file.

Listing 6.13 shows a simple program to demonstrate the concept. Every time the program is executed, it will ask the user to input 3 integers. Each integer is output to a file. If the named file already exists, these integers will be appended to the original contents.

Listing 6.13: `fopen()` with `"a"` mode sample program

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      char filename[40];
7      int i, n;
8      FILE *fp;
9
10     fprintf(stdout, "Input filename: ");
11     fscanf(stdin, "%s", filename);
12
13     /* open file in "a" mode (append) */
14     fp = fopen(filename, "a");
15
```

```
16      /* ask the user to input 3 integers, output
17         each integer to the text file */
18      for (i = 0; i < 3; i++) {
19          fprintf(stdout, "Input an integer value: ");
20          fscanf(stdin, "%d", &n);
21
22          /* output data to file */
23          fprintf(fp, "%d\n", n);
24      }
25
26      fclose(fp);
27      return 0;
28  }
```

► Self–Directed Learning Activity ◄

1. Encode and compile the program in Listing 6.13. Run the program three times or more times. For each execution, always input the same file name to see the effect of the append mode. The input values for the integers need not be the same (varying values are recommended though to see the effect).

2. What are the contents of the file during the first time you executed it? second time? third time?

6.10 Binary File Processing

A *binary file* is a file containing data encoded in binary, i.e., combinations of 0 and 1. Its format should be known in advance to be able to read and decode its contents properly. Examples of binary files are object and executable files.

There are four steps in manipulating binary files:

1. Declare a file pointer
2. Open the binary file
3. Access the binary file
4. Close the binary file

The file pointer declaration, and the commands for opening and closing the files are the same with binary files. The `"rb"` mode should be used as the second parameter of `fopen()` in order to open the file for input; while mode `"wb"` should be used to open it for output. In the following sections, we discuss how to read data from and write data to a binary file using the pre–defined functions `fread()` and `fwrite()`.

6.11 Writing Data into a Binary File

Data can be written to a binary file using the `fwrite()` function. Its corresponding syntax is as follows:

```
size_t fwrite(<buffer pointer>, <size>, <count>, <file pointer>)
```

Parameter `buffer pointer` is the base address of the element (for example, variable of a simple data type) or group of elements (for example, arrays and structures) to be written. The size, in number of bytes, of each element is indicated by parameter `size`, while `count` indicates the number of elements to be written. Thus, the total number of bytes to be written in the files is equivalent to `size` multiplied with `count`. The last parameter `file pointer` is the associated file pointer to the the named binary file.

The function returns an integral value corresponding to the number of elements written to a file. A successful `fwrite()` should return a value equivalent to the

third parameter denoted as `count`. If this is not the case, then a write error has occurred. There is a pre–defined function called `ferror()` that can be used to determine if an error has occurred. We will not discuss `ferror()` at this point in time.

We illustrate in the following subsections how to use `fwrite()` to output data based on the different data types that we learned in the past.

6.11.1 How to Write Values of Simple Data Types

We consider first how to write values of simple data types `char`, `int`, `float` and `double`. Listing 6.14 shows a simple program that writes just one character into a binary file named `"sample_01.dat"`.[7]

Take note of the parameters supplied to the `fwrite()` function call. The first parameter `&ch` is the base address of the memory space containing the value that we want to write to the file, the second parameter is the size of the `char` data type, the third parameter is 1 indicating that there is only one character to be written to the file, and `fp` is the file pointer.

Listing 6.15 extends the previous program by writing one character, one integer, one float and double data type values into a binary file named `"sample_02.dat"`. The first parameter for each of the `fwrite()` calls is the address of the variable, and the second parameter is the size corresponding to the variable's data type.

Listing 6.14: Writing one character into a binary file

```
1  #include <stdio.h>
2
3  int main()
4  {
5      FILE *fp;
6      char ch;
7
8      ch = 'A';
9      fp = fopen("sample_01.dat", "wb");
10     fwrite(&ch, sizeof(char), 1, fp);
11     fclose(fp);
12     return 0;
13 }
```

[7] We will use `"DAT"` as file extension throughout this document to denote data for a binary file.

Listing 6.15: Writing values of variables of simple data type

```
1  #include <stdio.h>
2
3  int main()
4  {
5      char ch = 'A';
6      int i = 123;
7      float f = 4.56789;
8      double d = 3.1415159;
9
10     FILE *fp;
11
12     fp = fopen("sample_02.dat", "wb");
13
14     fwrite(&ch, sizeof(char), 1, fp);
15     fwrite(&i, sizeof(int), 1, fp);
16     fwrite(&f, sizeof(float), 1, fp);
17     fwrite(&d, sizeof(double), 1, fp);
18
19     fclose(fp);
20     return 0;
21 }
```

▶ Self–Directed Learning Activity ◀

1. Encode, compile and test the programs in Listings 6.14 and 6.15. What do you think is the size of the binary file?

2. Go to the command line, and change your subdirectory to the directory containing the `exe` file. What are the results for the following commands?

   (a) `dir sample_01.dat`

   (b) `type sample_01.dat`

   (c) `dir sample_02.dat`

   (d) `type sample_02.dat`

6.11.2 How to Write Values of Group of Elements

We consider next how to write values for group of elements such arrays and structures. Listing 6.16 shows how to write a list of integer values stored in memory space which was allocated dynamically. The function call `fwrite(ptr, sizeof(int), n, fp)` indicate that the first parameter `ptr` is the base address of the memory space, each element has a size equal to `sizeof(int)`, and the number of elements is `n` – which is a user input in the sample program, and `fp` is the corresponding file pointer.

Listing 6.16: Writing values stored in dynamically allocated memory space

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int i;
7      int n;
8      int *ptr;
9      FILE *fp;
10
11     /* input number of integer elements to dynamically allocate */
12     printf("Input n: ");
13     scanf("%d", &n);
14
15     if ((ptr = malloc(sizeof(int) * n)) == NULL) {
16        fprintf(stderr, "ERROR: not enough memory.");
17        exit(1);
18     }
19
20     /* initialize elements */
21     for (i = 0; i < n; i++)
22         *(ptr + i ) = i * 5;
23
24     /* write data to binary file */
25     fp = fopen("sample_03.dat", "wb");
26     fwrite(ptr, sizeof(int), n, fp);
27     fclose(fp);
28     free(ptr);
29     return 0;
30 }
```

Listing 6.17 is essentially the same as the previous example except that the memory space for a list of integers is stored in an array. Recall that memory space for the array is associated with static memory allocation.

Listing 6.17: Writing values stored in an array

```
#include <stdio.h>

int main()
{
    int i;
    int A[5];
    FILE *fp;

    /* initialize array elements */
    for (i = 0; i < 5; i++)
        A[i] = i * 100;

    /* write data to binary file */
    fp = fopen("sample_04.dat", "wb");
    fwrite(A, sizeof(int), 5, fp);
    fclose(fp);
    return 0;
}
```

▶ Self–Directed Learning Activity ◀

1. Encode, compile and test the programs in Listings 6.16 and 6.17. What do you think is the size of the binary file?

2. Change the value of the third parameter of `fwrite()` from 5 to 1. What do you think will be the effect? Try changing it from 5 to 3 also. Verify your answer by running the modified programs and checking the size of the resulting data files.

3. Create new programs similar to the examples above to write list of values with the following types: `char`, `float` and `double`.

Listing 6.18 shows an example of how to write the value of a single structure variable. The example structure contains four members corresponding to the four simple data types.

Listing 6.18: Writing the value of a structure

```
1  #include <stdio.h>
2
3  struct sampleTag {
4      char ch;
5      int i;
6      float f;
7      double d;
8  };
9
10 int main()
11 {
12     struct sampleTag s;
13     FILE *fp;
14
15     /* initialize structure members */
16     s.ch = 'A';
17     s.i = 123;
18     s.f = 4.56789;
19     s.d = 3.1415159;
20
21     /* write structure into file */
22     fp = fopen("sample_05.dat", "wb");
23     fwrite(&s, sizeof(struct sampleTag), 1, fp);
24     fclose(fp);
25     return 0;
26 }
```

▶ Self–Directed Learning Activity ◀

1. Encode, compile and test the program in Listing 6.18.

2. Modify the program to declare an array of structure with 5 elements of type `struct sampleTag`. Initialize the values. Thereafter, write it into a file name `"sample_06.dat"`.

6.12 Reading Data From a Binary File

Data can be read from a binary file using the `fread()` function. Its corresponding syntax is as follows:

```
size_t fread(<buffer pointer>, <size>, <count>, <file pointer>)
```

The meaning of each parameter is the same with those used in `fwrite()`.

The function returns an integer value corresponding to the number of elements read from a file. A successful `fread()` should return a value equivalent to the third parameter. There are two possible scenarios when the return value is not the same as `count` (i) a read error has occurred or (ii) the `EOF` was reached. The program has to be able to handle these cases appropriately to avoid run–time error. There is a pre–defined function called `ferror()` that can be used to determine if a read error has occurred. We will not discuss `ferror()` at this point in time.

In the following subsections, we show how to read data from the sample data files which were created as specified in Listings 6.14 to 6.18.

6.12.1 How to Read Values of Simple Data Types

We consider first how to read from an existing binary file the values of simple data types `char`, `int`, `float` and `double`. Listing 6.19 shows a simple program that reads just one character into a binary file named `"sample_01.dat"` using `fread()`. The value is stored in the local variable `ch`. This value is displayed on the screen using `fprintf()`; note that this particular step is not really required in all programs manipulating binary files.

Listing 6.20 extends the previous program by reading one `char`, one `int`, one `float` and `double` data type values from a binary file named `"sample_02.dat"`.

Listing 6.19: Reading a single character from a binary file

```
#include <stdio.h>

int main()
{
    FILE *fp;
    char ch;

    fp = fopen("sample_01.dat", "rb");
    fread(&ch, sizeof(char), 1, fp);
    fprintf(stdout, "ch = %c\n", ch);
    fclose(fp);

    return 0;
}
```

Listing 6.20: Reading values of simple data types from a binary file

```
#include <stdio.h>

int main()
{
    char ch;
    int i;
    float f;
    double d;
    FILE *fp;

    fp = fopen("sample_02.dat", "rb");

    fread(&ch, sizeof(char), 1, fp);
    fread(&i, sizeof(int), 1, fp);
    fread(&f, sizeof(float), 1, fp);
    fread(&d, sizeof(double), 1, fp);

    fprintf(stdout, "ch = %c, i = %d, f = %f, d = %lf\n",
            ch, i, f, d);

    fclose(fp);
    return 0;
}
```

► **Self–Directed Learning Activity** ◄

1. Encode, compile and test the program in Listings 6.19 and 6.20. Make sure that the binary files `"sample_01.dat"` and `"sample_02.dat"` exist in the same directory as the executable files.

2. Find out what will happen if the data files are not present.

3. Find out what will happen if we interchange the sequence of the first two `fread()` statements in Listing 6.20. That is, the program will read first an integer value, then it will read a character value.

6.12.2 How to Read Values of Group of Elements

We consider next how to read values for group of elements such as arrays and structures from an existing binary file.

Listing 6.21 shows how to read a list of integers from the file `"sample_03.dat"`. The sample program illustrates a technique for reading data from a file without prior knowledge regarding how many values are stored in the file. This is achieved via a `while` loop which checks the return value of `fread()`. If `fread()` returns a value of 1, then there is still at least one integer value that can be read from the file. If the `while` condition becomes false, then it means that we have reached the `EOF`.

Listing 6.21: Reading an unknown number of integer values from a binary file

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int count;
6      int value;
7      FILE *fp;
8
9      fp = fopen("sample_03.dat", "rb");
10
11     while ((count = fread(&value, sizeof(int), 1, fp)) == 1)
12        fprintf(stdout, "count = %d, value = %d\n", count, value);
13
14     fclose(fp);
15     return 0;
16 }
```

We know in advance that there are 5 integer values stored in "sample_04.dat". This apriori information lets us write the codes such that the elements can be stored in an array with a known size. It will only require us to invoke fread() only once as shown in Listing 6.22.

Listing 6.22: Reading a known number of integer values from a binary file

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int i;
6      int A[5];
7      FILE *fp;
8
9      /* read data to binary file */
10     fp = fopen("sample_04.dat", "rb");
11     fread(A, sizeof(int), 5, fp);
12     fclose(fp);
13
14     for (i = 0; i < 5; i++)
15        fprintf(stdout, "A[%d] = %d\n", i, A[i]);
16
17     return 0;
18 }
```

Listing 6.23 shows an example of how to read the value of a single structure variable.

Listing 6.23: Reading a structure from a binary file

```
1  #include <stdio.h>
2
3  struct sampleTag {
4      char ch;
5      int i;
6      float f;
7      double d;
8  };
9
10 int main()
11 {
12     struct sampleTag s;
13     FILE *fp;
14
15     fp = fopen("sample_05.dat", "rb");
16     fread(&s, sizeof(struct sampleTag), 1, fp);
17
18     fprintf(stdout, "ch = %d, i = %d, f = %f, d = %lf\n",
19             s.ch, s.i, s.f, s.d);
20
21     fclose(fp);
22     return 0;
23 }
```

▶ Self–Directed Learning Activity ◀

1. Encode, compile and test the program in Listings 6.21 and 6.23. Make sure that the binary files exist in the same directory as the executable files.

2. Find out what will happen if the data files are not present.

3. In Listing 6.22, change the value of the third parameter of `fread()` from 5 to 1. What do you think will be the effect? Try changing it from 5 to 3 also. Verify your answer by running the modified programs.

6.13 Checking for the EOF

The pre–defined function `feof()` allows us to check if we have reached the `EOF`. Its syntax is given as:

```
int feof(<file pointer>)
```

It returns a non–zero value if the `EOF` was reached; otherwise, it returns a zero. This function can be used for both text and binary files.

We illustrate one way to use `feof()` in Listing 6.24. It is an alternative implementation for Listing 6.21. The program reads all the integer values from `"sample_03.dat"`. The technique requires that we read the first integer value from the file. Thereafter, we test if we have NOT yet reached the `EOF`. This is specified by the expression `!feof(fp)` in the `while` loop.. Take note of the use of the NOT logical operator before `feof()`. If it is not yet the `EOF`, we execute the body of the function which contains another call to `fread()`. The `while` loop condition will become false when `feof()` returns a non–zero value.

Which technique/implementation do you think is more efficient: the technique without `feof()` or the one with `feof()`?

Listing 6.24: Example usage of `feof()`

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int count;
6      int value;
7      FILE *fp;
8
9      fp = fopen("sample_03.dat", "rb");
10
11     count = fread(&value, sizeof(int), 1, fp);
12     while (!feof(fp)) {
13        fprintf(stdout, "count = %d, value = %d\n", count, value);
14        count = fread(&value, sizeof(int), 1, fp);
15     }
16
17     fclose(fp);
18     return 0;
19 }
```

6.14 Random File Access

6.14.1 File Position and `ftell()`

A binary file is actually organized as a group of elements with each element being one byte in size. Each byte is associated with a unique *file position*. The first byte is stored at file position 0, the second byte is at position 1, the next byte is at position 2, ..., and so on. Thus, if there are N bytes, i.e. the file size is N, the last byte is stored at file position *N-1*. This organization is actually the same with how the RAM is organized. It should be noted that for files, the file position N correspond to the position of the `EOF`.

In the following discussions, we will use the name `fpos` to denote file position. We can think of it as a variable whose value is from 0 to N.

Immediately after opening a file (either for read or write mode), the value of `fpos` is zero, i.e., it denotes the beginning or start of the file. A call to `fread()` or `fwrite()` will advance the file position by a certain number of bytes which is equivalent to parameter `size` multiplied with parameter `count`.

The current file position can be obtained via the return value of the pre–defined function `ftell()`.

```
long int ftell(<file pointer>)
```

We illustrate how to use `ftell()` in Listing 6.25. It is actually a modified version of the program in Listing 6.15. We just added several pairs of function calls to `ftell()` and `fprintf()`. When executed, this modified program will clearly show how the file position advances as new data values are written into the file.

Listing 6.25: Example usage of `ftell()`

```
1  #include <stdio.h>
2
3  int main()
4  {
5      char ch = 'A';
6      int i = 123;
7      float f = 4.56789;
8      double d = 3.1415159;
9
10     FILE *fp;
```

```
11      long int fpos;
12
13      fp = fopen("sample_06.dat", "wb");
14      fpos = ftell(fp);
15      fprintf(stdout, "After fopen(), fpos = %d\n", fpos);
16
17      fwrite(&ch, sizeof(char), 1, fp);
18      fpos = ftell(fp);
19      fprintf(stdout, "After writing ch, fpos = %d\n", fpos);
20
21
22      fwrite(&i, sizeof(int), 1, fp);
23      fpos = ftell(fp);
24      fprintf(stdout, "After writing i, fpos = %d\n", fpos);
25
26      fwrite(&f, sizeof(float), 1, fp);
27      fpos = ftell(fp);
28      fprintf(stdout, "After writing f, fpos = %d\n", fpos);
29
30      fwrite(&d, sizeof(double), 1, fp);
31      fpos = ftell(fp);
32      fprintf(stdout, "After writing d, fpos = %d\n", fpos);
33
34      fclose(fp);
35      return 0;
36  }
```

▶ Self–Directed Learning Activity ◀

1. Encode, compile and test the program in Listing 6.25. Take note of the printed values of `fpos`.

2. Modify the program in Listing 6.20 to detemine the file position after `fopen()` and `fread()`. This will allow us to verify and prove that `fread()` advances the file position by a certain number of bytes equivalent to `size` multiplied with `count` parameters.

6.14.2 File Position and `fseek()`

It is possible to reposition the file position anywhere between 0 and the `EOF` using `fseek()`. Its usage is as follows

```
int fseek(<file pointer>, <offset>, <origin>)
```

where the `offset` parameter is an integral value (measured in bytes) by which the file position will be moved relative to a specified `origin`. The new file position is computed as `origin + offset`. There are three possible position for `origin`, namely, the position of the first byte, the current file position, and the position immediately after the last byte, i.e., `EOF`. The `stdio.h` file conveniently defines these as macros which are listed in Table 6.2.

Table 6.2: Macros for `origin`

Macro	Meaning
`SEEK_SET`	beginning of the file, i.e., first byte
`SEEK_CUR`	current file position
`SEEK_END`	end of the file, i.e., after the last byte

Case 1: Reposition relative to start of the file. Example: the call `fseek(fp, 4, SEEK_SET)` will result into a new file position equal to 4. The new position is computed as `origin + offset` where origin is 0 (since it is the start of the file) and `offset` is 4.

Case 2: Reposition relative to current file position. Example: let us assume that `fpos` is currently 5. The call `fseek(fp, 4, SEEK_CUR)` will result into a new file position equal to 9. The new position is computed as `origin + offset` where origin is 5 (the current file position) and `offset` is 4.

Case 3: Reposition relative to `EOF`. Let us assume that the file is 17 bytes in size. The call `fseek(fp, -4, SEEK_END)` will result into a new file position equal to 13. The new position is computed as `origin + offset` where `origin` is 17 (since it is the `EOF`) and `offset` is -4. Take note that the `offset` must be negative when using `SEEK_END` as the `origin`.

Take note that `fseek()` will fail if it attempts to use a negative value for the new position. The `fseek()` function returns zero if repositioning was successful; otherwise, it returns a non–zero value indicating failure.

We illustrate how to use `fseek()` in Listing 6.26.

Listing 6.26: Example usage of `fseek()`

```
#include <stdio.h>

int main()
{
    FILE *fp;
    long int fpos;
    int offset;
    int start_fpos;    /* fpos at start */
    int current_fpos;  /* fpos at current */
    int eof_fpos;      /* fpos at EOF */

    fp = fopen("sample_06.dat", "rb");
    start_fpos = ftell(fp);

    /* Case 1: reposition relative to start of file */
    offset = 5;
    fseek(fp, offset, SEEK_SET);
    fpos = ftell(fp);
    fprintf(stdout, "offset = %d, SEEK_START at %d, fpos = %d\n",
            offset, start_fpos, fpos);

    /* Case 2: reposition relative to the current fpos */
    offset = 4;
    current_fpos = fpos;
    fseek(fp, offset, SEEK_CUR);
    fpos = ftell(fp);
    fprintf(stdout, "offset = %d, SEEK_CUR at %d, fpos = %d\n",
            offset, current_fpos, fpos);

    /* Case 3: reposition relative to EOF position */
    fseek(fp, 0, SEEK_END);
    eof_fpos = ftell(fp);

    offset = -2;  /* note: negative 2 */
    fseek(fp, offset, SEEK_END);
    fpos = ftell(fp);
    fprintf(stdout, "offset = %d, SEEK_END at %d, fpos = %d\n",
            offset, eof_fpos, fpos);
    fclose(fp);
    return 0;
}
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the program in Listing 6.26.
2. Assume a binary file containing 100 bytes. Which of the following `fseek()` call will be successful? Which will fail? If there is a failure, explain why it failed. For the items with `SEEK_CUR` assume that the current `fpos` is at 50. The best way to check if your answer is correct is to write a program, run it and check the return value of `fseek()`.
   (a) `fseek(fp, sizeof(char) * 2, SEEK_SET)`
   (b) `fseek(fp, -2 * sizeof(char), SEEK_END)`
   (c) `fseek(fp, sizeof(char)* 2, SEEK_END)`
   (d) `fseek(fp, -2 * sizeof(char), SEEK_SET)`
   (e) `fseek(fp, 50, SEEK_CUR)`
   (f) `fseek(fp, -50, SEEK_CUR)`
   (g) `fseek(fp, 51, SEEK_CUR)`
   (h) `fseek(fp, -51, SEEK_CUR)`

6.14.3 Random Access With `fseek()`

The programs that we developed so far read data from the binary files in the same sequence that they were written. It is actually possible to read the values in any sequence that we want. It is possible to access directly the first value, the last value or any value anywhere in the file without having to read any other elements. This type of access is called random access (as opposed to sequential access).

The function `fseek()` allows us to reposition `fpos` anywhere in the file starting from the first byte to the `EOF`. Once positioned, we can use `fread()` to read a set of bytes starting at the current `fpos`.

The binary file `"sample_06.dat"` which we created using a previous program (see Listing 6.25) contains one character value, one integer value, one float value and one double data type value in that sequence. We demonstrate in Listing 6.27 the concept of random access using `fseek()`. The program will print the data values in the following order: first the integer value, followed by double, then the character and finally the float value. This order is totally different from the order by which these values were written. Note that we deliberately used different `origin` values. It is actually possible to use just one common `origin`, for example, `SEEK_SET` or `SEEK_END`, and then specify the correct `offset` value.

Listing 6.27: Random access with `fseek()`

```
1  #include <stdio.h>
2
3  int main()
4  {
5      char ch;
6      int i;
7      float f;
8      double d;
9      FILE *fp;
10
11     fp = fopen("sample_02.dat", "rb");
12
13     /* position fpos to the integer data */
14     fseek(fp, 1, SEEK_SET);
15     fread(&i, sizeof(int), 1, fp);
16     fprintf(stdout, "i = %d\n", i);
17
18     /* position fpos to read double data */
19     fseek(fp, -8, SEEK_END);
20     fread(&d, sizeof(double), 1, fp);
21     fprintf(stdout, "d = %lf\n", d);
22
23     /* position fpos to read char data */
24     fseek(fp, 0, SEEK_SET);
25     fread(&ch, sizeof(char), 1, fp);
26     fprintf(stdout, "ch = %c\n", ch);
27
28     /* position fpos to read float data */
29     fseek(fp, 4, SEEK_CUR);
30     fread(&f, sizeof(float), 1, fp);
31     fprintf(stdout, "f = %f\n", f);
32
33     fclose(fp);
34     return 0;
35 }
```

▶ Self–Directed Learning Activity ◀

1. Encode, compile and test the program in Listing 6.27. What values were displayed?

2. The `offset` values that we used to position `fpos` are hardcoded numbers. This is not really a good programming practice. A better approach would

be to specify the `offset` based on the `sizeof()` operator. This will help us avoid problems when we port the source codes from one platform to another platform which may have differing sizes for their data types.

Change the original codes to their improved version as shown in the following table, and then run the program.

Original	**Improved Version**
`fseek(fp, 1, SEEK_SET)`	`fseek(fp, sizeof(char), SEEK_SET)`
`fseek(fp, -8, SEEK_END)`	`fseek(fp, -sizeof(double), SEEK_END)`
`fseek(fp, 4, SEEK_CUR)`	`fseek(fp, sizeof(int), SEEK_CUR)`

3. Modify the codes from the previous item by using only one uniform `origin`. That is, all `fseek()` should have the same `origin` unlike the original program which uses all the three possibilities. Modify the program such it uses only `SEEK_SET`. Thereafter, modify the program such that it uses only `SEEK_CUR`. The last modification should use only `SEEK_END`.

4. Create your own programs for accessing and displaying the data values stored in `"sample_06.dat"` in the sequence given below (one program per sequence). Try to create several variations of the programs, i.e. a variation when the `origin` is uniform for all `fseek()` and another variation where different `origin` values are used.

   (a) `char` value, `float` value, `double` value, `int` value

   (b) `float` value, `int` value, `char` value, `double` value

   (c) `double` value, `char` value, `float` value, `int` value

   (d) `double` value, `int` value, `float` value, `char` value

   (e) `double` value, `float` value, `int` value, `char` value

We present another example of random file access in Listing 6.28. The program first creates a binary file which will contain 10 integer values. Thereafter, the user is prompted to input an `"index"` corresponding to the integer that he/she would like to read from the file. The term `"index"` in this case takes on the same meaning as in the discussion of memory addressing way back in Chapter 1.

There are two key concepts that are worth learning from this particular example program, specifically:

1. Files can be opened, accessed, and closed in functions other than `main()`
2. If the file contains values of homogeneous data type (that is, all values have the same data type, for example an array), the `offset` parameter in the `fseek()` function can be specified as the `sizeof` the data type multiplied by the index of the element with an `origin` set to `SEEK_SET`.

Listing 6.28: Random access (second example)

```
1  #include <stdio.h>
2  #define FILENAME "sample_07.dat"
3
4  void CreateFile(char filename[])
5  {
6      FILE *fp;
7      int value;
8      int i;
9
10     fp = fopen(FILENAME, "wb");
11     for (i = 0; i < 10; i++) {
12         value = i * 100;
13         fwrite(&value, sizeof(int), 1, fp);
14     }
15
16     fclose(fp);
17 }
18
19 int main()
20 {
21     FILE *fp;
22     int value;
23     int index;
24     int count;
25     int retval;
26
```

```
27      /* first, we create the file */
28      CreateFile(FILENAME);
29
30      /* next, we re-open the file */
31      fp = fopen(FILENAME, "rb");
32
33      while (1) {
34         printf("Input index of the element you want to read.");
35         printf("  Enter -1 to exit: ");
36         scanf("%d", &index);
37         if (index == -1)
38            break;
39         else {
40            /* reposition fpos, then read data */
41            /* retval is zero for successful fseek,
42               otherwise it's negative */
43            retval = fseek(fp, sizeof(int) * index, SEEK_SET);
44            fprintf(stdout, "retval = %d, ", retval);
45
46            /* count should be 1 for successful fread */
47            count = fread(&value, sizeof(int), 1, fp);
48            fprintf(stdout, "count = %d, Data at index %d = %d\n\n",
49                    count, index, value);
50         }
51      }
52      fclose(fp);
53      return 0;
54  }
```

▶ Self–Directed Learning Activity ◀

1. Encode, compile and test the program in Listing 6.28. What is the size of the file? What are the contents of the binary file? Write down the value and the corresponding position for all the values stored in the file.

2. Run the program by supplying `index` values starting from 0, then 1, then 2, and so on until 9. Are the output consistent with the values you wrote in the previous item? Take note of the values of `retval` and `tcount` as well.

3. The index values from 0 to 9 are valid. Other values are actually incorrect. Try running the program with an `index` of -5. What is the return value of `fseek()` as stored in variable `retval`?

4. Next try an index value of 12. It will reposition `fpos` way past the `EOF`. What is the value of `retval`? What is the return value of `fread()` as stored in variable `count`?

6.15 Reading and Writing on the Same File

In applications such as Notepad, we can open an existing file, view the contents, edit the contents and save it. Essentially, this means that the file was opened for reading, but later on it performed a write operation as well.

Reading and writing on the same file (though not at exactly the same time) is actually possible using what we call "mixed" modes. These are:

- `"rb+"`or `"r+b"` – open the file for both read and write, note that file must exist.
- `"wb+"` or `"w+b"` – open the file for both read and write. If the file does not exist, a new one will be automatically created. If the file exists, the original contents will be erased and overwritten with a new one.

The presence of the `"+"` allows mixed mode access, i.e., both reading and writing. We will limit our discussion to `"rb+"` mode since this is the mode that will be most prevalent in actual practice.

Listing 6.29 shows an example of mixed mode access. The program opens the file `"sample_07.dat"` with a mode of `"rb+"`. Note that `"sample_07.dat"` was created by the program in the previous section. It then sets the file position via `fseek()` and then reads the data stored in the said position using `fread()`. The value is assigned to local variable `n`. The value of `n` is displayed on the screen using `fprintf()`. Thereafter, value of `n` is decremented by 1. The programs sets the file position again via `fseek()` and then writes the new value of `n` on the same position occupied by the original value of `n` (prior to decrement). The program is a simple illustration of a technique for editing/changing the contents of an existing file.

Listing 6.29: Mixed mode access

```
1  #include <stdio.h>
2  int main()
3  {
4      FILE *fp;
5      int n;
6
7      fp = fopen("sample_07.dat", "rb+");
8
9      fseek(fp, sizeof(int) * 5, SEEK_SET);
10     fread(&n, sizeof(int), 1, fp);
11
12     n = n - 1;
13     fseek(fp, sizeof(int) * 5, SEEK_SET);
14     fwrite(&n, sizeof(int), 1, fp);
15
16     fprintf(stdout, "n = %d\n", n);
17
18     fclose(fp);
19     return 0;
20 }
```

► Self–Directed Learning Activity ◄

1. Encode, compile and test the program in Listing 6.29. What are the contents of the binary file prior to running the program? after running the program?
2. Run the program several times. What do you notice? Try to give an explanation of what the program does everytime you run it.
3. Modify the program such that:
   (a) it will always increment the first data value by 1.
   (b) it will always multiply the last data value by 2.

6.15.1 The `fflush()` function

The `fflush()` function is used to force data to be written onto the output file. The function prototype is as follows:

```
int fflush(<file pointer>)
```

It returns 0 for if the operation is successful, otherwise it returns `EOF`. Use `fflush()` after `fwrite()` in case there are multiple `fread()` and `fwrite()` on the same file inside some loop.

Listing 6.30 shows another example of mixed mode access with `fflush()`. The program gives the user two options. The first option creates a file that will contain the characters `"Hello World!"`. The mode used here is `"wb"`. The second option updates the contents of the file by changing all lower case letters to their upper case equivalents using the predefined function `toupper()`.[8] The mode used here is mixed mode as denoted by `"rb+"`.

Listing 6.30: Mixed mode access (second example)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <ctype.h>
5
6  #define FILENAME "sample_08.dat"
7
8  void CreateFile(char filename[])
9  {
10      FILE *fp;
11      char message[20] = "Hello World!";
12
13      fp = fopen(filename, "wb");
14      fwrite(message, sizeof(char), strlen(message), fp);
15      fclose(fp);
16 }
17
18 void UpdateFile(char filename[])
19 {
20      FILE *fp;
```

[8] Do some information search to learn more about the `toupper()` and `tolower()` functions. Their corresponding header file is `ctype.h`.

```
21      char ch;
22
23      fp = fopen(filename, "rb+");
24
25      while ((fread(&ch, sizeof(char), 1, fp)) == 1) {
26          if (ch >= 'a' && ch <= 'z') {
27             ch = toupper(ch); /* change to upper case */
28             fseek(fp, -1, SEEK_CUR);
29             fwrite(&ch, sizeof(char), 1, fp);
30             fflush(fp);
31          }
32      }
33      fclose(fp);
34  }
35
36  int main()
37  {   int choice;
38
39      printf("Input [1] Create file, [2] Update file: ");
40      scanf("%d", &choice);
41      if (choice == 1)
42         CreateFile(FILENAME);
43      else if (choice == 2)
44              UpdateFile(FILENAME);
45      else fprintf(stderr, "Invalid choice.");
46
47      return 0;
48  }
```

▶ Self–Directed Learning Activity ◀

1. Encode, compile and test the program in Listing 6.30. Run the program twice. Choose option 1 during the first time. Make sure that you examine the contents of the file produced. Thereafter, run the program the second time and choose option 2. Examine again the contents of the file. What do you notice? Can you explain what actually happened?

2. Write your own program for reading the contents of a file containing characters. If the file is an upper case character, replace it with a corresponding lower case. Note: there is a pre–defined function `tolower()` that allows conversion from upper to lower case.

6.16 Chapter Summary

The key ideas presented in this chapter are summarized below:

- A file is a group of elements. We classify the files into two: text files and binary files.
- We learned how to create our own files, and how to read data from existing files using the following pre–defined functions:
  - `fopen()` to open the file
  - `fclose()` to close the file
  - `fprintf()` for formatted output; this function is used to write data to a text file
  - `fscanf()` for formatted input; this function is used to read data from a tex file
  - `fwrite()` to write data to a binary file
  - `fread()` to read data from a binary file
  - `feof()` to check the `EOF` was reached
  - `ftell()` to determine the current file position `fpos`
  - `fseek()` to reposition `fpos`
  - `fflush()` to force data to be written onto the output file.
- Mode `"w"` is for writing to a text file, `"a"` is for appending to a text file, and `"r"` is for reading from a text file.
- Mode `"wb"` is for writing to a binary file, `"rb"` for reading from a binary file.
- Mode `"w+b"` or `"wb+"` and `"rb+"` or `"r+b"` is for mixed mode, i.e., reading and writing to the same file.
- VERY IMPORTANT NOTE: We should have information regarding the data format of a binary file in order for us to be able to read and interpret its contents correctly.

Problems for Chapter 6

For problems 6.1 and 6.2 assume a text file named `"essay.txt"`.

Problem 6.1. Write a program that will read the contents of `"essay.txt"`. The output of the program are the values of the following:

- How many letters are in the file (lower case and upper case)
- How many lower case letters are in the file
- How many upper case letters are in the file
- How many lines are present in the file (note: a line ends with a newline character)
- How many are lower case vowels
- How many are upper case vowels
- How many are lower case consonants
- How many are upper case consonants

Problem 6.2. Write a program that will read the contents of `"essay.txt"` and copy all the lower case letters into a file named `"lower.txt"` and all upper case letters into a file named `"upper.txt"`.

Problem 6.3. Assume that there are two text files. Write a program that will determine whether these two text files are identical or not (i.e., the contents are the same).

For problems 6.4 and 6.5 assume a list of integers stored in a text file named `"integers.txt"`. *We do not know in advance how many integers are present.*

Problem 6.4. Write a program that will read the contents of `"integers.txt"`. The output of the program are the values of the following:

- The smallest integer value (minimum value)
- The largest integer value (maximum value)
- The number of integers read from the file
- The sum of all the integers
- The average of all the integers

Problem 6.5. Write a program that will read the contents of `"integers.txt"`. The output of the program is the frequency count of the following:

- How many integers are greater than or equal to 0
- How many integers are negative
- How many integers are odd numbers
- How many integers are even numbers

Problem 6.6. Assume two lists of integers both in ascending order. The first list was stored in a file named `list1.txt`, the second in `list2.txt`. We do not know how many integers are stored in each file. Write a program that will read data values from these two files and store them also in ascending order in a new file named `list3.txt`.

For the next two problems, assume a list of integers stored in a binary file named `"integers.dat"`. *We do not know in advance how many integers are present.*

Problem 6.7. Write a program that will read the contents of `"integers.dat"`. The output of the program are the values of the following:

- The smallest integer value (minimum value)
- The largest integer value (maximum value)
- The number of integers read from the file
- The sum of all the integers
- The average of all the integers

Problem 6.8. Write a program that will read the contents of `"integers.dat"`. The output of the program is the frequency count of the following:

- How many integers are greater than or equal to 0
- How many integers are negative
- How many integers are odd numbers
- How many integers are even numbers

Problem 6.9. Assume two lists of integers both in ascending order. The first list was stored in binary files named `list1.dat`, the second in `list2.dat`. We do not know how many integers are stored in each file. Write a program that will read data values from these two files and store them also in ascending order in a new binary file named `list3.dat`.

For the following problems, assume the following declarations:

```
struct EmployeeTag
{
     char name[40];
     int ID_number;
     float salary;
};

typedef struct EmployeeTag EmployeeType;
```

Problem 6.10 Write a program that will declare an array of 10 structures. Each structure is of type `EmployeeType`. Initialize the contents of the array. Finally, write the contents of the array onto a binary file named `"employee.dat"`.

Problem 6.11 Write a program that will read employee structure one at a time from the file `"employee.dat"` created in the previous problem. The value of each structure should be displayed on the standard output device.

Problem 6.12 Write a program that will read employee structure in REVERSED ORDER one at a time from the file `"employee.dat"` created in the previous problem. The value of each structure should be displayed on the standard output device.

Problem 6.13 Write a program that will read ALL employee structures from the file `"employee.dat"` created in the previous problem. Data read from the file should be stored into an array of employee structure. Thereafter, the value of each array element should be displayed on the standard output device.

Problem 6.14 Write a program that will ask the user to input an integer value representing an ID number. Determine if there is an employee structure stored in the file `"employee.dat"` with the same ID number. If there is such a structure, display the contents of the employee structure; otherwise, display a message indicating "There is no employee with such ID number". Note that this is essentially a search problem.

Problem 6.15 Write a program that will open the file using `"rb+"` mode. Update

the employee structures stored in the file `"employee.dat"` by giving a salary increase of 10 percent to all employees. For example, if an employee's salary is 10,000.00 then the updated value of the `salary` member should be 10,100.00. Make sure to call `fflush()` right after `fwrite()`.

Problem 6.16 Write a program that will open the file using `"rb+"` mode. Sort the employee structures in ascending order based on ID_number member. DO NOT use an array for this; work directly with the binary file. Refer back to the straight selection sorting algorithm discussed in the chapter on arrays. Apply the algorithm on the binary file. Make sure to call `fflush()` right after `fwrite()`.

Chapter 7

Recursion

7.1 Motivation

When we say that "something" is **recursive**, we mean that that "something" is described or defined in such a way such that it *refers to itself within the definition*. Something that is recursive is said to be *self–similar* or *self–referential*.

Recursion occurs in nature, in languages, in (the visual) art, in mathematics and in programming. Let us have first a visual appreciation of recursion before we delve on the programming aspect. Please see the images in the following websites.

`http://www.its.caltech.edu/~atomic/snowcrystals/photos/photos.htm`
`http://webecoist.com/2008/09/07/17-amazing-examples-of-fractals-in-nature/`
`http://www.flickr.com/photos/gadl/sets/72157594316295785/`
`http://profron.net/pictures/sluggo%20recursive.jpg`
`http://math-art.net/2008/02/14/recursive-cartoon/`

The first two websites feature recursion in nature observable in patterns appearing on snowflakes, seashells and Romanescu brocolli. The images in the third site feature photographs which were digitally altered such that they appear to be recursive. Note that the recursion is patterned after the seashell shape seen in the second website. The last two websites feature cartoon showing a character doing an action which is repeated several times within the same cartoon.

Recursion occurs in languages as well, for example, a story, within a story within a story... The acronym "GNU" is actually a recursive. When expanded, it stands for "GNU is Not Unix". Find out what is the unofficial expansion of the abbreviation "PNG". PNG is an image file format.

7.2 Definition of Recursion in Programming

According to `http://www.itl.nist.gov/div897/sqg/dads/HTML/recursion.html`, the NIST definition for recursion is as follows:

> Recursion is an algorithmic technique where a function, in order to accomplish a task, calls itself with some part of the task.

Why do we need (to learn) recursion? One reason is that the problem itself is recursive. It is but natural to give a recursive solution (example: Quicksort and binary tree traversals). While a non–recursive solution may exist, it may be difficult to formulate, implement and understand. Another reason is that there are languages that do not have looping mechanisms such as LISP. In such languages, the predominant technique for computing through sequences and lists is recursion.

7.3 Bad Example – Infinite Recursion

We deliberately show first a bad example of a recursive function definition. It is bad because, in theory, the recursive function calls will never terminate resulting into an infinite program. In real life, the program will crash and terminate abruptly because it will eventually run out of memory. An infinite recursion is the counterpart of an infinite loop.[1]

Listing 7.1: Bad example of recursion (infinite recursion)

```
1  #include <stdio.h>
2
3  void Bad()
4  {
5     printf("Bad example due to infinite recursion.\n");
6     Bad();  /* recursive call here */
7  }
8
9  int main()
10 {
11    Bad();
12    return 0;
13 }
```

[1] Actually, an infinite recursion is worse than an infinite loop because it eats up memory every time does a recursive call resulting into a stack overflow.

▶ Self–Directed Learning Activity ◀

1. Encode, compile and run the program in Listing 7.1. What is the output?
2. It will probably take a very long time for the program to terminate due to low memory. Try adding a local variable declaration say `double A[10]` inside `Bad()` function. This local variable will eat up memory everytime there's a recursive call. Recompile and re–run the program to see what will happen.
3. Modify the program by adding `printf("Test");` after the recursive function call. Run the program. Notice that you will not get `"Test"` as part of the output. What do you think is the reason for this?

An infinite recursion results into a logical error! The moral of the story from this bad example is that a recursive function must be defined such that it will terminate after a finite number of computational steps.

There is actually something worse than the bad example function presented above. Can you guess? Answer: a `main()` function that calls itself. This is a blunder that beginning programmers commit unknowingly. Although it is syntactically possible, unwritten rules in actual programming practice discourage a recursive `main()` function.

7.4 Concept of Base Case and Recursive Case

A recursive function should be defined such that it has two parts, namely:

1. **Base case** – this is the non–recursive portion of the function that solves the *trivial* part of the problem. The trivial part usually just return a value (often a constant value) in case the return type is not `void`. The base case leads to function termination.
2. **Recursive case** – this is the *non–trivial* part wherein the function has to perform more computation that requires it to call itself.

Note that there maybe several base and recursive cases within the same function.

The base case is separated from the recursive case via a conditional control structure, i.e., an `if` (or `switch case`), to test for a condition which will determine whether the function will compute the base case or the recursive case.

7.5 Recursive Function Definition in C

As in all C functions, the function definition must specify the data type of the value to be returned by the function, this is then followed by the function name and the parameter list. Thereafter, it is followed by the body of the function.

A typical format for a recursive function definition is as follows:

```
<return type> <function name> ( [<parameter list>] )
{
     [local variable declaration]

     if (<expression>)
        [return] <base case>
     else
        [return] <recursive case>
}
```

The `return` keyword SHOULD be written when the return type of the function is NOT `void`. Alternatively, we can invert the logic of the conditional expression such that the `recursive case` can be placed in the body of the `if` clause and the `base case` will be in the `else` clause. That is,

```
     if (<expression>)
        [return] <recursive case>
     else
        [return] <base case>
```

It is possible to define a recursive function such that it just has an `if` without a corresponding `else`. That is,

```
     if (<expression>)
        [return] <recursive case>
```

In this particular scenario, the `base case` corresponds to a function termination when the expression evaluates to false.

We discuss several variations of recursive functions in the following subsections.

7.5.1 Recursive Function Returning a Value

We consider as our first correct example the C implementatoin of the factorial function. The recurrence relation

$$n! = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

is a mathematical function that is defined recursively.[2] The exclamation mark denotes the factorial function. We see that the:

1. **base case** corresponds to the trivial task of returning a value of 1
2. **recursive case** corresponds to the non–trivial task of returning `n * (n-1)!`

The C program that computes the factorial of an integer is shown in Listing 7.2. It is very interesting to note that we actually translated a definition expressed in one language, i.e., the language of mathematics to another language, i.e., the C programming language.

We can see that the language of mathematics is more compact compared with C. The C translation require that we specify the data type of parameter `n`. The exclamation mark notation had to be translated into a user–defined function name `factorial`. The fact that there is a one–to–one correspondence between the mathematical and the C language function definition facilitates seamless translation.

Listing 7.2: Recursive factorial function

```
1  #include <stdio.h>
2
3  int factorial(int n)
4  {
5     if (n == 0)
6        return 1;    /* base case */
7     else
8        return n * factorial(n-1);  /* recursive case */
9  }
10
11 int main()
12 {
```

[2]On a historical note, the function definitions (non–recursive and recursive) in modern programming languages were actually patterned from the nature of functions and relations in mathematics. Recursive functions in particular are based from recurrence relations.

```
13      int n;
14
15      printf("Input n: ");
16      scanf("%d", &n);
17      printf("The factorial of %d = %d", n, factorial(n));
18      return 0;
19  }
```

Let us trace how the program works. The program starts execution from the `main()` function, which calls `printf()` which in turn calls `factorial(n)`.

Consider first the trivial case when `n` is 0. The function call `factorial(0)` returns a value of 1 which is then displayed via the `printf()` statement in `main()`. Next, consider the non–trivial case when `n > 0`. Let us say that we call `function(3)`. The following are the corresponding computation steps in sequence:

1. `main()` calls `printf()`
2. `printf()` calls `factorial(3)`
3. Recursive case `factorial(3)` calls `factorial(2)` via `return 3 * factorial(2)`
4. Recursive case `factorial(2)` calls `factorial(1)` via `return 2 * factorial(1)`
5. Recursive case `factorial(1)` calls `factorial(1)` via `return 1 * factorial(0)`
6. Base case `factorial(0) = 1`
7. Value of `factorial(0)` computed as 1, value of `factorial(1) = 1 * 1`
8. Value of `factorial(1)` computed as 1, value of `factorial(2) = 2 * 1`
9. Value of `factorial(2)` computed as 2, value of `factorial(3) = 3 * 2`
10. Value of `factorial(3)` computed as 6, return result to `printf()` of `main()`.
11. `printf()` outputs a value of 6 on display screen.
12. `main()` terminates.

Notice that the parameter in the recursive call is changed, in this case by decrementing the value by one. It should not be identical as in the original function call; otherwise, the recursive function will never terminate. IMPORTANT: A change

in the state of the parameter is required for a recursive function to eventually reach its base case.

► **Self–Directed Learning Activity** ◄

1. Encode, compile and run the program in Listing 7.2. Run it with the following values for `n` and verify the results with manual computation.

   (a) 0

   (b) 1

   (c) 2

   (d) 3

2. The value of the factorial "grows" at a "fast rate" as we increase the value of `n`. Continue experimenting with the program by running it with values of n = 4, 5, 6.. Notice what happens to the printed value. To handle larger whole numbers, try adding the keyword `long` before `int` to make the data type a long integer.

3. What do you think will happen if the `return` keyword is removed from the `else` clause? Verify it by modifying and running the program.

4. Modify the conditional expression in the original program such that the recursive case will be the body of the `if` clause and the base case will be the body of the `else` clause.

5. What do you think will happen if the body of the function is written such that there is only statement which is `return n * factorial(n-1);`?

7.5.2 Recursive Function of Type `void`

Listing 7.3 shows a simple recursive function that does not return any value. The keyword `return` need not be used in the function.

The condition `n > 0` determines whether the function `Series1()` will call itself recursively or not. The associated action for the base case is to terminate the function which happens when the conditional expression evaluates to false. The recursive case is computed when the condition evaluates to true. Notice that the parameter in the recursive call is changed to reach the base case.

Listing 7.3: Recursive function of type `void`

```
#include <stdio.h>

void Series1(int n)
{
   if (n > 0) {
      printf("n = %d\n", n);
      Series1(n-1);
   }
}

int main()
{
    int n;

    printf("Input n: ");
    scanf("%d", &n);
    Series1(n);
    return 0;
}
```

Let us trace how this recursive function works. The program starts execution from `main()` function, which calls `Series1(n)`. Let us consider first the trivial case when `n` is less than 1 (0 or any negative integer). Calling `Series1(0)` will result into a base case which simply terminates the function call. Next, consider the non–trivial case when `n > 0`. Let us say that we call `Series1(2)`. The following are the corresponding computation steps in sequence:

1. `main()` calls `Series1(2)`
2. Recursive case, `printf()` outputs `n = 2`, then `Series1(2)` calls `Series(1)`
3. Recursive case, `printf()` outputs `n = 1`, then `Series1(1)` calls `Series(0)`
4. Base case, `Series(0)` terminates.
5. `Series1(1)` terminates.
6. `Series1(2)` terminates.
7. `main()` terminates.

► **Self–Directed Learning Activity** ◄

1. Encode, compile and run the program in Listing 7.3. Run it with the following values for `n` and verify the results with manual computation.

   (a) -5

   (b) 0

   (c) 5

   (d) 10

   What do you think is the program doing?

2. Modify the program such that the last output will be zero.

3. What do you think will happen if we change the recursive call parameter from `n-1` to `n`?

7.6 Tail Recursive and Non–Tail Recursive Functions

A recursive function is said to be *tail recursive* if the last action that the function will do is to call itself. On the other hand, a recursive function that has to do some other action after calling itself is said to be *non–tail recursive*.[3]

The `Series1()` function in the previous program is an example of a tail–recursive function. The `factorial()` function is non–tail recursive. Recall the statement `return n * factorial(n-1)` in the function definition. This requires that we compute FIRST the value of `factorial(n-1)` and only AFTER that can we perform the last action which is to multiply it with `n`.

Let us look at another example of non–tail recursive function. What do you think will happen if we interchange the sequence of the `printf()` and the recursive function call in the previous program? This is actually what is shown in Listing 7.4.

Notice that `Series2()` is a non–tail recursive function because there is still another action, i.e., to execute `printf()`, waiting to be executed AFTER the recursive call has finished its task.

[3]Other books and literature use the term *tail–end recursion.*

Listing 7.4: Example of a non–tail recursive function

```
1  #include <stdio.h>
2
3  void Series2(int n)
4  {
5     if (n > 0) {
6        Series2(n-1);
7        printf("n = %d\n", n);
8     }
9  }
10
11 int main()
12 {
13     int n;
14
15     printf("Input n: ");
16     scanf("%d", &n);
17     Series2(n);
18     return 0;
19 }
```

Let us trace the program execution by calling `Series2(3)`. The following are the corresponding computation steps in sequence:

1. `main()` calls `Series2(3)`
2. Recursive case, `Series2(3)` calls `Series2(2)`. VERY IMPORTANT NOTE: the `printf()` statement cannot be executed yet because we have to finish first the task of computing `Series2(2)`.
3. Recursive case, `Series2(2)` calls `Series2(1)`. Same note applies.
4. Recursive case, `Series2(1)` calls `Series2(0)`. Same note applies.
5. Base case `Series2(0)` terminates.
6. `Series2(1)` calls `printf()` which outputs `n = 1`. `Series2(1)` terminates.
7. `Series2(2)` calls `printf()` which outputs `n = 2`. `Series2(2)` terminates.
8. `Series2(3)` calls `printf()` which outputs `n = 3`. `Series2(3)` terminates.
9. `main()` terminates.

The important lesson that we should have learned from this particular example is that any instruction written after a non–tail recursive function call will have to wait until the recursive call gets to finish its task first.

▶ Self–Directed Learning Activity ◀

1. Encode, compile and run the program in Listing 7.4. Run it with the following values for `n` and verify the results with manual computation.

   (a) -5

   (b) 0

   (c) 5

   (d) 10

   What do you think is the program doing?

2. Modify the program such that the first output will be zero.

3. What do you think will happen if we change the recursive call parameter from `n-1` to `n`?

7.7 More Representative Examples of Recursion

Let us look at other examples of recursive functions. Listings 7.5 and 7.6 are programs with recursive functions in charge of printing the elements of a 1D array. Listing 7.7 contains a recursive function that computes the total of array elements. Notice that each of the recursive function call passes `index + 1` in order for the program execution to approach the base case. Try to understand and trace what the programs are doing.

Listing 7.5: Accessing array elements via recursion (example 1)

```
1 #include <stdio.h>
2
3 void AccessArray1(int A[], int index, int n)
4 {
5     if (index < n) {
6         printf("A[%d] = %d\n", index, A[index]);
7         AccessArray1(A, index+1, n);
8    }
9 }
```

```
10
11 int main()
12 {
13     int A[5] = {10, 20, 30, 40, 50};
14
15     AccessArray1(A, 0, 5);
16     return 0;
17 }
```

Listing 7.6: Accessing array elements via recursion (example 2)

```
1  #include <stdio.h>
2  void AccessArray2(int A[], int index, int n)
3  {
4      if (index < n) {
5          AccessArray2(A, index+1, n);
6          printf("A[%d] = %d\n", index, A[index]);
7     }
8  }
9
10 int main()
11 {
12     int A[5] = {10, 20, 30, 40, 50};
13
14     AccessArray2(A, 0, 5);
15     return 0;
16 }
```

Listing 7.7: Computing sum of array elements via recursion

```
1  #include <stdio.h>
2  int SumArray(int A[], int index, int n)
3  {
4      if (index < n)
5           return A[index] + SumArray(A, index+1, n);
6  }
7
8  int main()
9  {
10     int A[5] = {10, 20, 30, 40, 50};
11
12     printf("Sum of array elements = %d\n", SumArray(A, 0, 5));
13     return 0;
14 }
```

Listing 7.8 is a recursive equivalent of a function that traverses a single linked list. The `"nodetype.h"` file is the same as the one we used in Chapter 4 (Linked Lists). Notice that the recursive call passes the pointer to the next node, i.e., `pFirst->pNext` in order to approach the base case.

Listing 7.8: Linked list travesal using recursion

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "nodetype.h"
5
6  void RecursiveTraverse(nodeStructType *pFirst)
7  {
8       if (pFirst != NULL) {
9          printf("%d\n", pFirst->data.key);
10         RecursiveTraverse(pFirst->pNext);
11     }
12 }
```

► Self–Directed Learning Activity ◄

1. Encode, compile and run the programs in Listing 7.5 to 7.8.
2. For recursive function, identify what is the (i) base case and (ii) recursive case.
3. Which of the recursive function is tail–recursive? non–tail recursive?
4. With reference to Listing 7.8, what will happen if we move the `printf()` statement after the recursive function call?

7.8 Chapter Summary

The key ideas presented in this chapter are summarized below:

- Recursion is a technique that allows us to define functions that are self–referential. A recursive function has two parts, namely (i) base case and a (ii) recursive case.
- The base case eventually leads to program termination. The recursive case is the portion of the function where a function calls itself.
- A recursive function can either be classified as tail–recursive or non–tail recursive.
- Recursion is best used in problems that are recursive in nature, and where an iterative solution is difficult to formulate.

Problems for Chapter 7

Problem 7.1. Trace the program in Listing 7.9. Is `Mystery()` tail–end or non–tail end recursive?

Listing 7.9: `Mystery()` recursive function

```
1   #include <stdio.h>
2
3   int Mystery(int x, int y)
4   {
5       if (y == 0)
6          return x;
7       else
8          return (Mystery(y, x % y));
9   }
10
11  int main()
12  {
13       printf("Mystery(2, 0) = %d\n", Mystery(2, 0));
14       printf("Mystery(10, 25) = %d\n", Mystery(10, 25));
15       return 0;
16  }
```

Problem 7.2. Trace the program in Listing 7.10. Is `DisplayDigits()` tail–end or non–tail end recursive?

Listing 7.10: `DisplayDigits()` recursive function

```
#include <stdio.h>

void DisplayDigits(int n)
{
    if (n < 10)
        printf("%d\n", n);
    else {
        printf("%d\n", n % 10);
        DisplayDigits (n/10);
    }
}

int main()
{
    int n;

    printf("Input a positive integer: ");
    scanf("%d", &n);   /* try to input 123456 */
    DisplayDigits(n);
}
```

Problem 7.3. Trace the program in Listing 7.11. Is `BLAP()` tail–end or non–tail end recursive? Is `BLIP()` tail–end or non–tail end recursive?

Listing 7.11: `BLIP()` and `BLAP()` recursive functions

```
#include <stdio.h>

void BLIP(int n)
{
     if (n != 0)  {
        printf("BLIP = %d\n", n);
        BLIP(n-1);
     }
}
```

```
10
11 void BLAP(int n)
12 {
13      if (n!= 0)  {
14         BLIP(n);
15         BLAP(n-1);
16      }
17 }
18
19 int main()
20 {
21     BLAP(4);
22     return 0;
23 }
```

Problem 7.4. The Fibonacci function is given by the following formula:

$$\text{Fibonnaci(n)} = \begin{cases} 1 & \text{if } n \text{ is 0 or } n \text{ is 1} \\ \text{Fibonacci(n-1) + Fibonacci(n-2)} & \text{if } n > 1 \end{cases}$$

- Which is the base case? recursive case?
- Evaluate the value of the following function calls
  - `Fibonacci(0)`
  - `Fibonacci(1)`
  - `Fibonacci(2)`
  - `Fibonacci(5)`
- Write a recursive C function that implements the Fibonacci function. What is the return type of the function? How many parameters are needed? What is the data type of the parameter?
- Is the function tail–recursive or non–tail recursive?
- It should be realized that although the mathematical definition is recursive, it is possible to write an iterative solution (i.e., using loop). Write an iterative solution and compare its performance with the recursive solution. Try your programs on increasing value of `n` starting from 0. Which is bettter in terms of computational performance, the recursive or the iterative solution?

Problem 7.5. The Ackermann function is the following formula (where `m` `n` are whole numbers

$$\text{Ackermann(m, n)} = \begin{cases} n+1 & \text{if } m \text{ is } 0 \\ \text{Ackermann(m-1, 1)} & \text{if } n \ 0 \\ \text{Ackermann(m-1, Ackermann(m, n-1))} & \text{otherwise} \end{cases}$$

- Which is the base case? recursive case?
- Evaluate the value of the following functions
  - `Ackermann(0, 10)`
  - `Ackermann(1, 2)`
  - `Ackermann(2, 1)`
  - `Ackermann(3, 0)`
- Write a recursive C function that implements the Fibonacci function. What is the return type of the function? How many parameters are needed? What is the data type of the parameter?
- Is the function tail–recursive or non–tail recursive?