

ECE 264 Purdue University

Yung-Hsiang Lu

Please send corrections, questions, suggestions to yunглу@purdue.edu.

updated on Thursday 3rd January, 2013

Contents

I	C Language Features	1
1	Functions	3
1.1	Purposes of Functions	3
1.2	main Function	4
1.3	Function with Return Value	11
1.4	Function with Control	13
1.5	Function Calls Function	14
1.6	Practice Problems	18
2	Computer Memory (Stack)	19
2.1	Address and Value	19
2.2	Stack Memory	21
2.3	Function Return Location	22
2.4	Call Stack	25
2.5	Scope	28
2.6	Return Value	31
2.7	Practice Problems	35

CONTENTS

CONTENTS

3 Recursion	37
3.1 Examples	37
3.1.1 Select Balls	37
3.1.2 One-Way Street	38
3.1.3 Integer Partition	40
3.1.4 Tower of Hanoi	41
3.2 C Implementation	44
3.2.1 Select Balls	45
3.2.2 One-Way Street	53
3.2.3 Integer Partition	54
3.2.4 Tower of Hanoi	57
3.2.5 Fibonacci Numbers	59
3.3 Practice Problems	62
4 Pointer	67
4.1 Pointer and Address	67
4.2 Pointer and Argument	69
4.3 Array and Argument	72
4.4 Practice Problems	73
5 Memory Management (Heap)	77
5.1 Create a Fixed-Size Array	77
5.2 Create an Array using malloc	78
5.3 Stack and Heap	80
5.4 Array and Address	84
5.5 Function Return a Heap Address	85
5.6 Two-Dimensional Array in C	87
5.7 Pointer and Argument	89
5.8 Use valgrind to Detect Memory Leak	92

CONTENTS

CONTENTS

6 String	95
6.1 Compare Strings using <code>strcmp</code>	96
6.2 Use <code>strstr</code> to Detect Substring	97
6.3 Understand <code>argv</code>	99
6.4 Count Substrings	102
6.5 Practice Problems	106
7 Structure	107
7.1 Vector	107
7.2 Object as an Argument	109
7.3 Object and Pointer	112
7.4 Return an Object	114
7.5 Object and <code>malloc</code>	115
7.6 Constructor and Destructor	118
7.7 Structure in Structure	125
8 File Operations	129
8.1 Read Character by Character Using <code>fgetc</code>	130
8.2 Read Integer	132
8.3 Read and Write Integers	134
8.4 Read String	136
8.5 <code>fseek</code> and <code>ftell</code>	137
8.6 Binary File and Object	137
8.7 Practice Problems	141

9 Program with Multiple Files	143
9.1 Header File	144
9.2 Compile and Link	147
9.3 Makefile	149
9.4 Use Makefile for Testing	152
9.5 Practice Problems	154
10 Dynamic Structures	155
10.1 Linked List	156
10.1.1 Node	156
10.1.2 Insert	157
10.1.3 Delete	161
10.1.4 C Program	165
10.2 Binary Tree	171
10.2.1 Node	171
10.2.2 Insert	172
10.2.3 Search	177
10.2.4 Print	178
10.2.5 Delete	179
10.2.6 Iterator	182
10.2.7 Destruct	187
10.2.8 main	187
10.2.9 Makefile	189
10.3 Practice Problems	191

11 More Practice Problems	193
11.1 Anagrams	193
11.1.1 Distinct Letters	193
11.1.2 Unique Words	193
11.2 Integer Partition	193
11.2.1 All Partitions	193
11.2.2 Distinct Numbers	194
11.2.3 Increasing Numbers	195
11.2.4 Only 1-5 Can Be Used	195
11.3 Arrange Three Balls	196
11.3.1 List All Options	196
11.3.2 Count without Listing All Options	196
11.4 Encryption	196
II Programming Tools	199
12 Debugger <code>gdb</code> and <code>ddd</code>	201
12.1 Trace a Program	202
12.2 Show Call Stack	206
12.3 Locate Segmentation Fault	209
12.4 Print Object	212
12.5 DDD	214
12.6 Prevent Bugs	219
12.7 Controversy about <code>assert</code>	223
12.8 Practice Problems	224
13 Test Coverage	227

CONTENTS

CONTENTS

List of Figures

1.1	When a function calls another function, the program executes the code in the called function. When the called function finishes, the program returns to the caller, right below the function call. The arrows mean the flow of the program. The numbers indicates the sequence of execution.	17
2.1	Push and pop of stack. (a) Data are pushed (stored) onto a stack. Originally, the top of the stack stores number 720. Number 653 is pushed to the top of the stack. (b) Data are retrieved (popped) from the stack. Push and pop can occur at only the top of the stack. This figure uses integers as an example. A stack can store any type of data: integer, character, double, string, or programmer-defined types as explained in Chapter 7.	22
2.2	(a) The stack changes after f_1 , f_2 , and f_3 are called. Each function call pushes the called function onto the stack. (b) When f_3 finishes, it is popped from the stack and now f_2 is at the top. When f_2 finishes, it is popped from the stack and f_1 is at the top.	22
2.3	Compilers (such as <code>gcc</code>) mark the return location (RL) after each function call. . . .	23
3.1	To decide $f(n)$, we consider the options for the first ball. If the first ball is blue, the remaining $n - 1$ ball have $f(n - 1)$ options. If the first ball is red, the second ball must be blue and remaining $n - 2$ ball have $f(n - 2)$ options.	38
3.2	The city's streets are either east-west bound or north-south bound. A car can move only east bound or north bound.	39
3.3	(a) A driver cannot turn anywhere from A to B. (b) A driver cannot turn anywhere from C to D. (c) There are some turning options from E to F. At E, the driver can go north bound first or east bound first, as indicated by the two arrows.	39
3.4	Tower of Hanoi. (a) Some disks are arranged along pole A and the goal is to move all disks to pole B, as shown in (b). A larger disk can never be placed above a smaller disk. A third pole, C, is used when necessary.	41

3.5	Moving one disk from A to B requires only one step.	41
3.6	Moving two disks from A to B requires three steps.	42
3.7	Moving two disks from A to B requires three steps.	43
3.8	Computating $f(5)$ requires calling $f(4)$ and $f(3)$. Computing $f(4)$ requires calling $f(3)$ and $f(2)$	60
3.9	Redraw Figure 3.8. This looks like a binary tree: computing each value requires the sum of two values.	61
5.1	(a) Call stack. After calling <code>malloc</code> , <code>arr2</code> 's value is an address in the heap memory. In this example, we use 10000 for the value. Six memory locations are allocated and they are adjacent. (b) Heap memory. The values have not been assigned so "?" is used for their values. We need to use a symbol in the stack (<code>arr2</code>) to access heap memory so the heap memory has no column for symbols.	81
5.2	A two-dimensional array is created in two steps. The first step creates an array of integer pointers. In the second step, each pointer stores the address of the first element in an integer array.	88
8.1	A text file is like a stream. This example uses <code>file2.c</code> as the input file. (a) After calling <code>fopen</code> , the stream starts from the very first character of the file and ends with EOF. (b)(c) Calling <code>fgetc</code> each time takes one character out from the stream. (d) After calling <code>fgetc</code> enough times, the characters in the file are retrieved. (e) Finally, the end of file character EOF is returned.	132
8.2	A program uses <code>fscanf</code> to read integers from a file; <code>fscanf</code> is able to read numbers separated by space or new line (' <code>\n</code> ').	134
10.1	A linked starts with no Node. This figure shows three views: (a) the source code view, (b) illustration, and (c) stack memory. The illustration view is the most common in literature. The three views are shows simultaneously so that you can understand how to write code and what happens in computer's memory. In (b), when a pointer's value is NULL, it is common to make it point to "Ground."	157
10.2	Creating a Node object whose value is 917.	157
10.3	Replacing the three lines by using <code>List_insert</code>	158
10.4	Calling <code>List_insert</code> again to insert another Node object.	159
10.5	Insert the third object by calling <code>List_insert</code> again.	160
10.6	Insert the third object by calling <code>List_insert</code> again.	161

10.7 To delete a <code>Node</code> object, we create a new pointer <code>p</code> . Its value is the same as <code>head</code> 's value.	162
10.8 We create another pointer <code>q</code> ; its value is the same as <code>p->next</code>	163
10.9 Modify <code>p->next</code> and bypass the node that is about to be deleted.	164
10.10 Release the memory pointed by <code>q</code>	165
10.11 An empty tree has one pointer called <code>root</code> and its value is <code>NULL</code>	172
10.12 A binary tree with only one <code>Node</code> . Both <code>left</code> and <code>right</code> are <code>NULL</code> . This node is the root because it has no parent. It is also a leaf node because it has no child.	173
10.13 A binary tree with two <code>Nodes</code> . The node with value 917 is still the root; it is no longer a leaf node because it has one child. The node with value -504 is a leaf node because it has no child.	174
10.14 A binary tree with three <code>Nodes</code>	175
10.15 A binary tree with five <code>Nodes</code> . We create a new view (d) to simplify the representation of a tree.	176
10.16 (a) The original binary search tree. (b) Deleting the node "5". This node has no child. 180	
10.17 Deleting the node "14". This node has one child.	180
10.18 Deleting the node "8". The node has two children. (a) Exchange the values of this node and its successor. (b) Deleting the successor.	181
10.19 The iterator after calling <code>TreeIterator_construct</code>	183
10.20 The linked has three nodes after calling <code>TreeIterator_visitNext</code> because the root's two children are also inserted.	184
12.1 Start-up window of DDD.	214
12.2 DDD opens the <code>vector.c</code> program.	215
12.3 A breakpoint is set. A <code>STOP</code> symbol is shown at the line.	216
12.4 The program stops at the breakpoint.	217
12.5 DDD shows the attributes of <code>v1</code>	218

LIST OF FIGURES

LIST OF FIGURES

List of Tables

3.1	The first ten values of Fibonacci Numbers.	60
3.2	Coefficients for computing $f(n)$	61
3.3	Coefficients for computing $f(n)$	62
4.1	Four different usages of * in C programs. Sometimes you can see LHS for “left hand side” and RHS for “right hand side”.	69
5.1	The first part of this table repeats Table 4.1. The second part summarizes how to create an array on heap memory and to access the array’s elements.	84
8.1	Functions for opening, writing to, and reading from text and binary files.	141

Part I

C Language Features

Chapter 1

Functions

1.1 Purposes of Functions

C programs are composed of functions. A special function called `main` is the starting point. We should not put everything inside the `main` function. Putting everything inside the `main` function is like writing an article with only one paragraph. Most articles have multiple paragraphs and most programs have many functions. As a rule of thumb, a function should contain at most 50 lines¹. Why? If a function is too long, it becomes difficult to understand and error-prone. Functions serve many purposes, for example

- Computation is conducted by several functions. This can make each function shorter and easier to understand. Each function should focus on doing only one thing.
- If something needs to be done over and over again, it can be implemented in a function and the function is called multiple times. If slight changes are needed each time, the differences can be specified using the function's arguments. This is an important principle when creating a function. A function's behavior should be controlled by the arguments *only*. If a function's behavior is affected by anything more than the arguments (for example, by a global variable), the function may behave differently even if the arguments have the same values. When this occurs, understanding the function becomes difficult.
- When a program becomes longer, it may be organized into several files. Functions may be implemented in different files and `gcc` can *link* several files into a single executable program. We will talk about linking in Section 9.2.

In your first C course, you should have seen some functions such as `scanf` and `printf`. They are provided by the C language. This chapter talks about “programmer-defined” functions. Before we do that, let's review what you have learned in the first programming class.

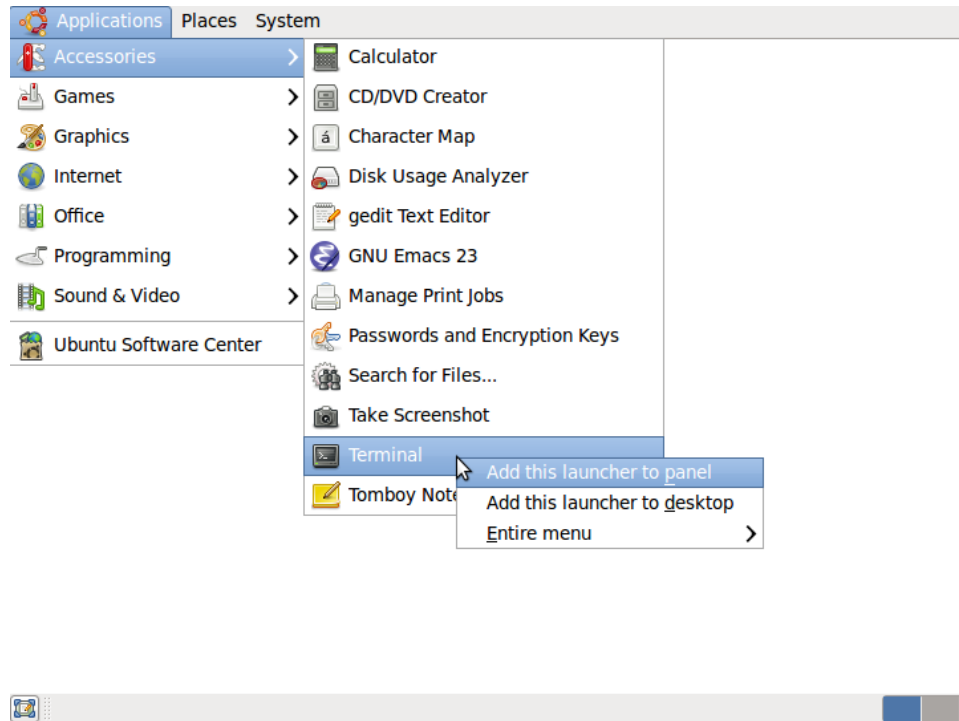
¹The number is for *reference*. This is not a strict rule.

1.2 main Function

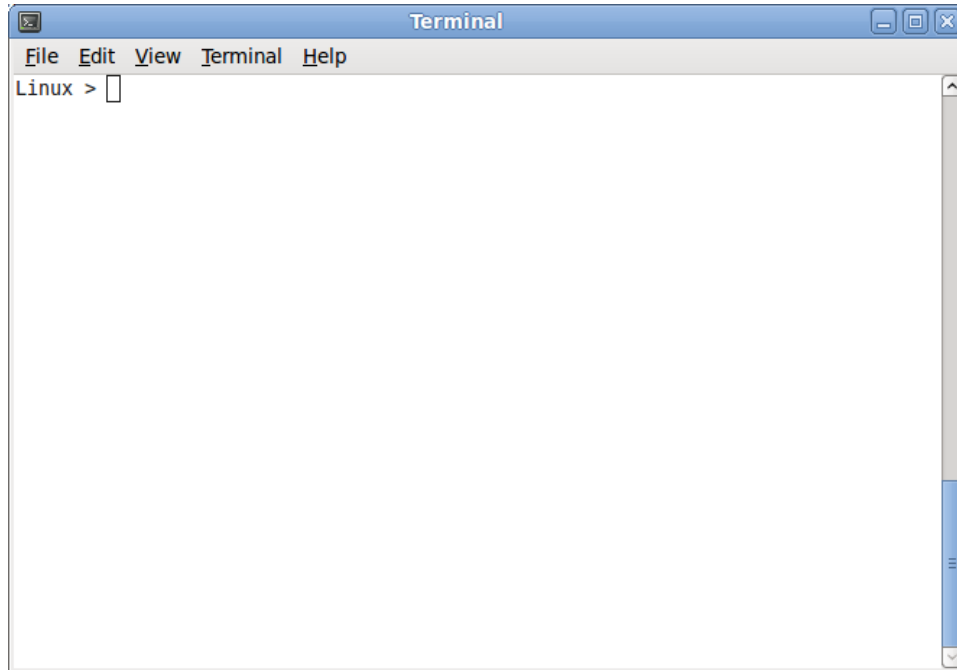
A C program starts at the `main` function: a function whose name is `main`. This is a special function and every program must have it. Let's consider our first sample program in this book:

```
/* file: args.c
   purpose: print the arguments
   */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
    int cnt;
    printf("There are %d arguments.\n", argc);
    printf("The arguments are:\n");
    for (cnt = 0; cnt < argc; cnt ++)
    {
        printf("argv[%d] is %s\n", cnt, argv[cnt]);
    }
    return EXIT_SUCCESS;
}
```

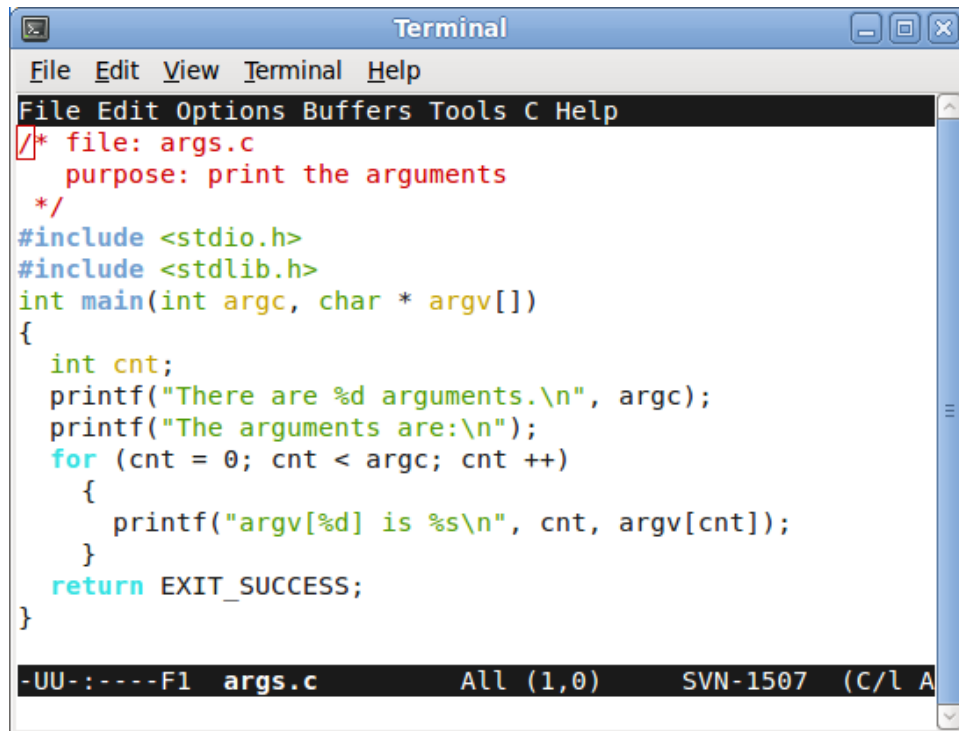
This book uses Linux for the programming environment. To compile this program, start a Linux Terminal. In Ubuntu Linux, click Applications, select Accessories and scroll down to Terminal. You can add Terminal to the launch panel as shown below.



This figure shows a terminal window. `Linux >` is the *command prompt*. Your command prompt may be different from the one shown here. A commonly used prompt (the default one) is the dollar sign `$`.



Many editors are available in Linux. I prefer `emacs` because of the syntax highlight feature shown below.



```
File Edit Options Buffers Tools C Help
[*] * file: args.c
    purpose: print the arguments
 */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
    int cnt;
    printf("There are %d arguments.\n", argc);
    printf("The arguments are:\n");
    for (cnt = 0; cnt < argc; cnt++)
    {
        printf("argv[%d] is %s\n", cnt, argv[cnt]);
    }
    return EXIT_SUCCESS;
}
```

-UU-:----F1 args.c All (1,0) SVN-1507 (C/l A

You can invoke `emacs` in two ways.

1. `emacs -nw args.c`

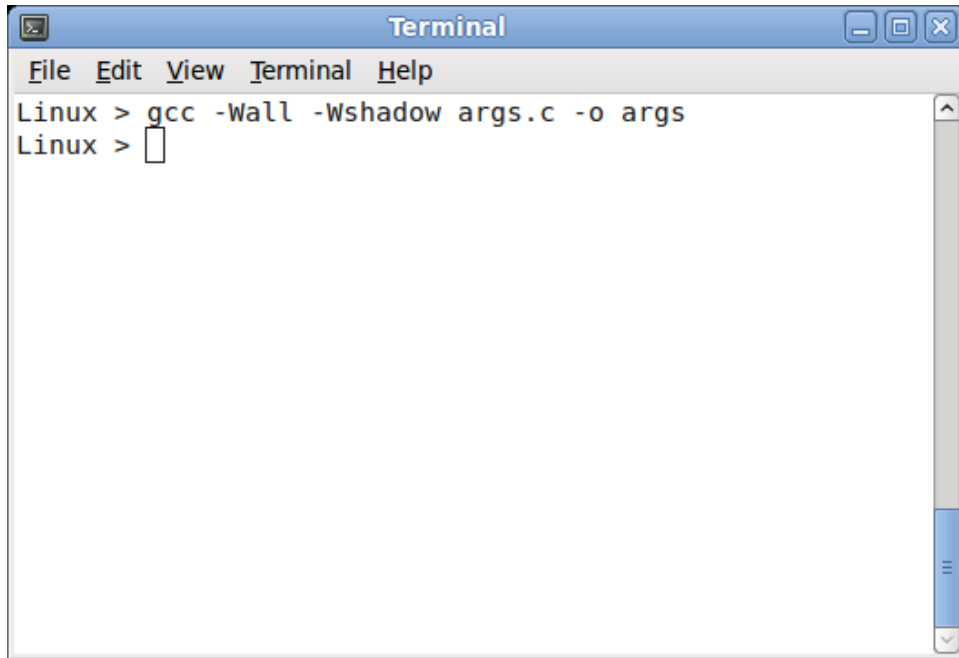
This uses `emacs` inside the terminal as shown above; the option `-nw` means not to create a new window.

2. `emacs args.c &`

This launches `emacs` in a separate window and the symbol `&` allows you to use the terminal and `emacs` at the same time.

Type this command in the terminal to *compile* the program.

```
> gcc -Wall -Wshadow args.c -o args
```



The first symbol `>` is the shell prompt and you do not type it. The command starts from `gcc`. This command means

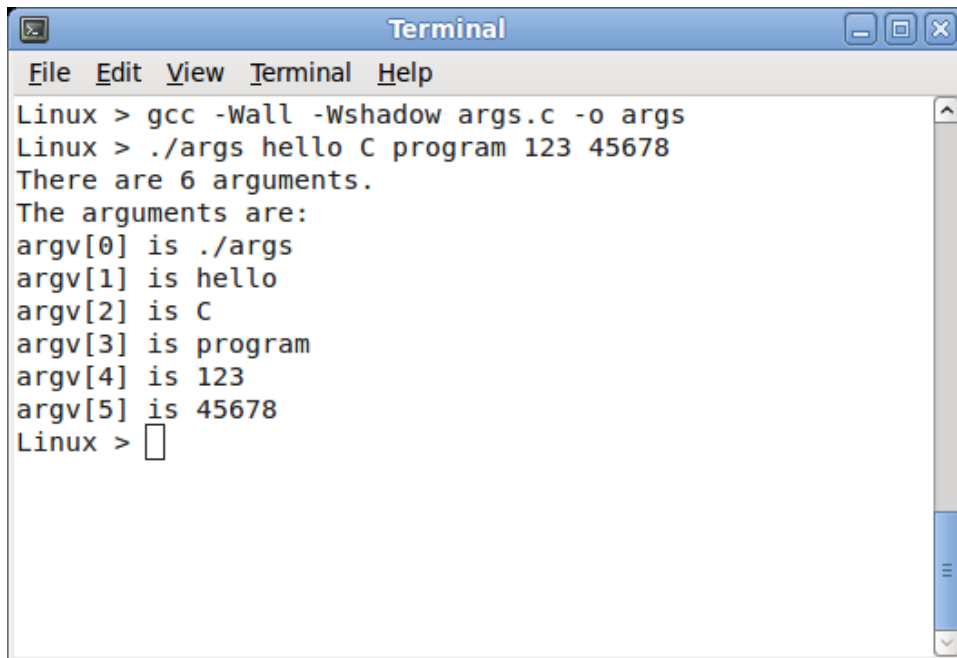
- execute the `gcc` program
- turn on warning messages by adding the `-Wall -Wshadow` options.
- the input C source code is `args.c`. By convention, the source code of a C program is stored in files with `.c` extension.
- the output executable program, after `-o`, is called `args`. By convention in UNIX, a program has no extension. This is different from Windows; in Windows programs have the `.exe` extension in the file names.

The following shows one execution of the program.

```
> ./args hello C program 123 45678
There are 6 arguments.
The arguments are:
argv[0] is ./args
```

```
argv[1] is hello
argv[2] is C
argv[3] is program
argv[4] is 123
argv[5] is 45678
```

The leading `./` means to find a program inside the current directory (also called *folder*). This ensures that the program is the program you have just created. It is possible to have several programs in different directories. By adding `./`, the `args` program in the current directory is executed.



```
Linux > gcc -Wall -Wshadow args.c -o args
Linux > ./args hello C program 123 45678
There are 6 arguments.
The arguments are:
argv[0] is ./args
argv[1] is hello
argv[2] is C
argv[3] is program
argv[4] is 123
argv[5] is 45678
Linux > 
```

The line `./args hello ...` is the command line and they form *command line arguments*. This command has six arguments stored in the `argv` array. The first argument is always the name of the program itself. Hence, `argv[0]` is `"./args"`. In C programs, an array's indexes start from zero, not one. The terminal automatically separates the arguments based on space. You can use one, two, or more spaces to separate the arguments. The following two commands do exactly the same thing even though the latter has more spaces between arguments.

```
> ./args hello C program 123 45678
> ./args      hello      C      program      123      45678
```

This main function

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

takes two arguments.

By convention, the two arguments are called `argc` and `argv`. You should use `argc` and `argv` because every C programmer knows what they mean. The first argument `argc` means the number of arguments in the command line. Here, 'c' means count. The second argument is called `argv`; here 'v' means values. Each argument is a string. C does not really have strings; instead, C uses an array of characters for a string. A string must end with the special character `'\0'`.

What does `-Wall` mean? It tells `gcc` to turn on warning messages. What does `-Wshadow` mean? It tells `gcc` to detect shadow variables. The following example contains a shadow variable, `val`. Its value is initialized to 5. Suppose inside the `{...}` block, we define another variable of the same name and the value is -8. When the block finishes, the inner `val` is no longer valid so the value goes back to 5.

```
/* file: shadow.c
   purpose: example of a shadow variable
 */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
    int val = 5;
    printf("val = %d\n", val);
    {
        int val = -8;
        printf("val = %d\n", val);
    }
    printf("val = %d\n", val);
    return EXIT_SUCCESS;
}
```

The output of this program is

```
val = 5
val = -8
val = 5
```

Why are shadow variables bad? They make your programs difficult to understand. You think you have changed `val` to -8 but its value goes back to 5 later. By adding `-Wshadow` after `gcc`, you will get this message

```
shadow.c: In function 'main':
shadow.c:11: warning: declaration of 'val' shadows a previous local
shadow.c:8: warning: shadowed declaration is here
```

This helps you detect and remove shadow variables.

You should always turn on `gcc` warning messages and remove all warnings as soon as possible. Warning messages often indicate errors. Ignoring warning messages will make your program buggy.

So far, I have emphasized more than once the importance of making your programs easy to understand. As your programs become more complex, this will become increasingly important.

1.3 Function with Return Value

The following is an example showing the programmer-defined `add` function.

```
/* file: add.c
   purpose: show a programmer-defined function called add
   */
#include <stdio.h>
#include <stdlib.h>
int add(int a, int b, int c)
{
    return (a + b + c);
}

int main(int argc, char * argv[])
{
    if (argc != 4)
    {
        printf("need 3 integers.\n");
        return EXIT_FAILURE;
    }
    int x, y, z, s;
    x = (int) strtol(argv[1], NULL, 10);
    y = (int) strtol(argv[2], NULL, 10);
    z = (int) strtol(argv[3], NULL, 10);

    s = add (x, y, z);

    printf("%d + %d + %d = %d.\n", x, y, z, s);
    return EXIT_SUCCESS;
}
```


This program needs three arguments (in addition to the program's name). Therefore, the program checks whether `argc` is four.

You should always check `argc` before using `argv[1]` (or `argv[2]` or `argv[3], ...`). If `argc` is 2 and your program uses `argv[3]`, the program will crash.

The program converts the arguments into three integers using the `strtol` function provided by C. The first argument of `strtol` is the string to be converted. The second argument is a pointer to a string for storing the address of the string which is not a valid number. You can use `NULL` for the second argument to indicate that you do not care what is left after conversion. The third argument is the base. The program uses 10 for decimal values. You may have heard the function `atoi`, which can also convert a string to an integer. The problem of `atoi` is that it does not handle errors (i.e. the string is not a valid integer) well. Thus, `strtol` is a better choice for making your program more robust. This program calls the `add` function. The function adds the three integers and return the result. This value is stored in `s`. Use the `gcc` command to compile the program and generate the executable called `add`.

```
> gcc -Wall -Wshadow add.c -o add
```

The following shows two executions of the program.

```
> ./add 3 18 71
3 + 18 + 71 = 92.

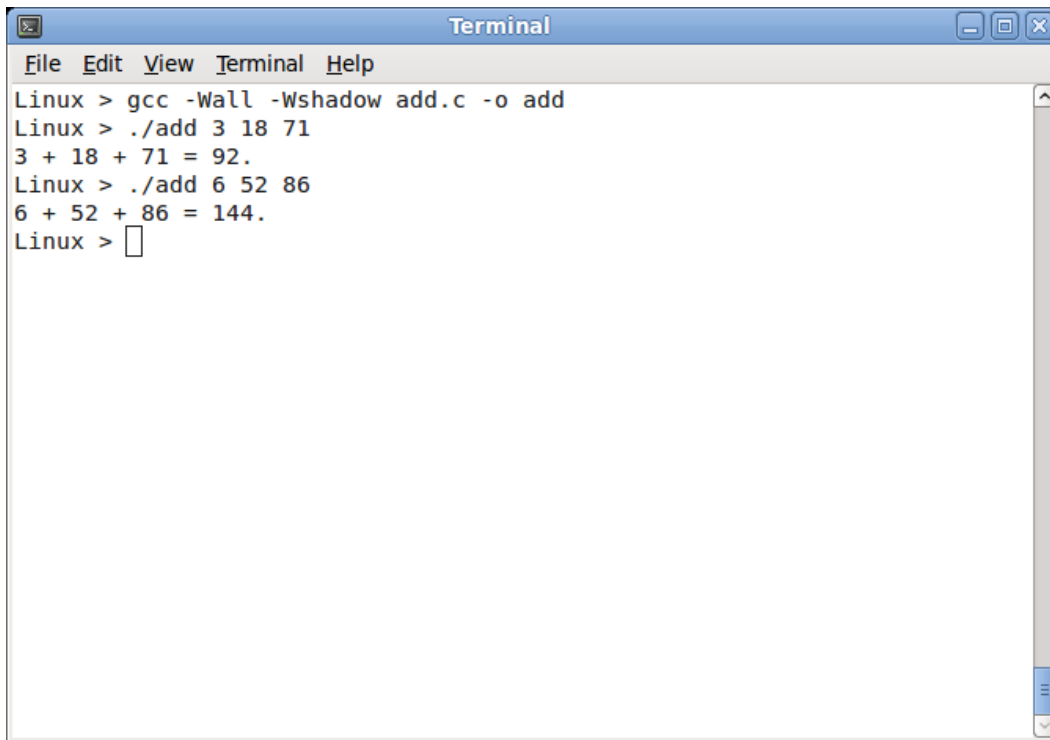
> ./add 6 52 86
6 + 52 + 86 = 144.
```

A C program may call a function and discard the returned value. For example, the program may call

```
add(x, y, z);
```

without assigning the returned value to `s` (or anything), even though `add` returns an integer.

If a function does not return anything, assigning the returned value to a variable is a syntax error. For example,



```
Terminal
File Edit View Terminal Help
Linux > gcc -Wall -Wshadow add.c -o add
Linux > ./add 3 18 71
3 + 18 + 71 = 92.
Linux > ./add 6 52 86
6 + 52 + 86 = 144.
Linux > 
```

```
void f(void)
{
    /* not return anything */
}

void g(void)
{
    int a = f(); /* syntax error */
}
```

In this example `f` does not return anything. When `g` calls `f`, `g` assigns `f`'s return value to an integer variable `a`. This is invalid.

1.4 Function with Control

A function can have `if` condition inside to determine the return value of the function.

```
/* file: funcif.c
   purpose: a function whose output is controled by the input
   */
#include <stdio.h>
#include <stdlib.h>
```

```

int func(int a, int b)
{
    if (a > b)
        {
            return 1;
        }
    if (a < b)
        {
            return -1;
        }
    return 0;
}

int main(int argc, char * argv[])
{
    if (argc != 3)
        {
            printf("need 2 integers.\n");
            return EXIT_FAILURE;
        }
    int x, y;
    x = (int) strtol(argv[1], NULL, 10);
    y = (int) strtol(argv[2], NULL, 10);
    printf("func(%d, %d) = %d.\n", x, y, func(x, y));
    return EXIT_SUCCESS;
}

```

The following shows three executions with different input arguments

```

> ./funcif 5 9
func(5, 9) = -1.
> ./funcif 9 5
func(9, 5) = 1.
> ./funcif 9 9
func(9, 9) = 0.

```

1.5 Function Calls Function

A programmer-defined function can call a C library function (such as `printf` and `scanf`) as well as one or more programmer-defined functions.

```

/* file: func12.c
   purpose: func1 and func2 are two programmer-defined functions.
   main calls func1 and func1 calls func2

```

```

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

int func1(int a);
int func2(int a);

int func1(int a)
{
    printf("beginning of func1(%d)\n", a);
    func2(a + 5);
    printf("end of func1 (%d)\n", a);
    return 0;
}

int func2(int a)
{
    printf("beginning of func2(%d)\n", a);
    printf("end of func2 (%d)\n", a);
    return 0;
}

int main(int argc, char * argv[])
{
    func1(5);
    return EXIT_SUCCESS;
}

```

The file first *declares* two functions `func1` and `func2` as shown in these two lines:

```

int func1(int a);
int func2(int a);

```

What is a function declaration? It includes the following three pieces of information:

1. the return type, for example `int` or `double`. When we talk about user-defined types in Chapter 7, it can also be a user-defined type. If a function does not return anything, we use `void`.
2. the name of the function. Each function must have a unique name. In this example, `func1` and `func2` are the names.
3. the argument types enclosed by parentheses. Usually, we also give names to the arguments. In this example, each function needs an integer as the argument and we give `a` for the name. If there are two or more arguments, they are separated by commas. If the function needs no argument, put `void` inside the parentheses.

A semicolon is added after the parentheses. This means that it is a function declaration.

If we remove the semicolon and add open and close brackets `{...}`, we *define* the function. It is also called the function's or *function's body*.

The definitions specify what the functions actually do. Obviously, we need to define functions. Why do we need to declare functions? Sometimes, we have to inform `gcc` of the existence of a function before giving `gcc` the definition. By declaring a function, we tell `gcc` that the function exists so that it may be called by another function, even though `gcc` has not seen the definition yet. Function declaration becomes essential when we discuss programs with multiple files in Chapter 9.

If we remove the declaration, `gcc` will give a warning message:

```
In function func1:
warning: implicit declaration of function func2
```

This means `func1` calls `func2` even though `gcc` does not know anything about `func2` yet. It is a warning message because `gcc` finds the definition later in the file.

If we also remove the definition, `gcc` will produce an error message:

```
In function func1:
warning: implicit declaration of function func2
In function 'func1':
undefined reference to 'func2'
```

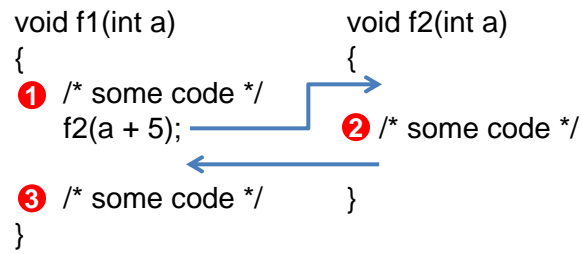
Now, let's see the output of the program:

```
beginning of func1(5)
beginning of func2(10)
end of func2 (10)
end of func1 (5)
```

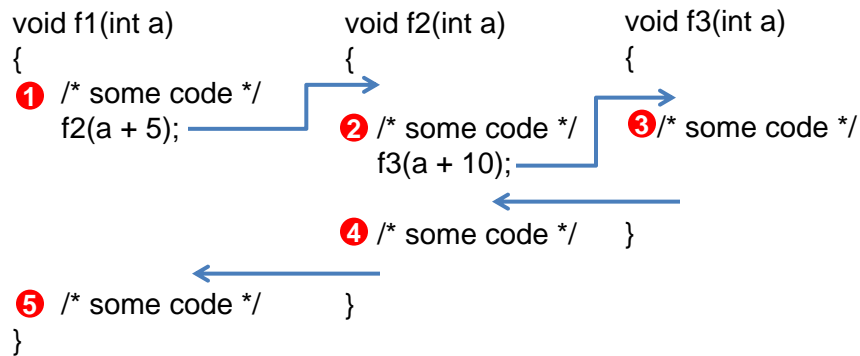
Notice the order of the output. The program enters `func1` first and prints the message `beginning of func1(5)`. Then, `func1` calls `func2`. The program prints the message `beginning of func2(10)`. The program exits `func2` and returns to `func1`. If a function calls another function, when the called function finishes, the program returns to the caller. The compiler (i.e. `gcc`) ensures that the program continues after the function call. Consider this example calling a function

```
function(/* some arguments */);
/* <-- This is the place immediately after the call */
```

After the function finishes, the program continues from the point immediately after the function call. This is illustrated in Figure 1.1.



(a)



(b)

Figure 1.1: When a function calls another function, the program executes the code in the called function. When the called function finishes, the program returns to the caller, right below the function call. The arrows mean the flow of the program. The numbers indicates the sequence of execution.

You may have noticed that both `func1` and `func2` can have variable called `a`. However, you cannot have code like this

```

void f1(int a)
{
  int b = 6;
  int b = 16;
  ...
}

```

inside the same function. If you do so, `gcc` will tell you

```
error: redefinition of b
```

Next chapter explains how programs use memory in computers. The knowledge will help you understand why both `func1` and `func2` can have a variable called `a`. You will also understand why the value of `a` is changed back to 5 when the program is back at `func1`.

1.6 Practice Problems

1. Write a program that can find and report the largest number in the command line arguments. Suppose this program is called `findMax`. The following shows two executions of the program.

```
> ./findMax 6 4 -9 8 7 3
The largest number is 8.
```

```
> ./findMax 1 2 3 4
The largest number is 4.
```

2. Write a program that can determine whether the input arguments are in the ascending order. Suppose this program is called `checkOrder`. The following shows two executions of the program.

```
> ./checkOrder 1 2 3 4
Yes.
```

```
> ./checkOrder 6 4 -9 8 7 3
No.
```

Chapter 2

Computer Memory (Stack)

2.1 Address and Value

Memory is an important part of computers. When you buy a computer, you usually need to specify how much memory to be installed. A typical laptop computer today has several GB memory. G means “giga”, a billion. B means “byte”. One byte is 8 bits. One bit means enough space to store either zero or one.

Computers’ memory is organized as *address - value* pairs. One analogy is the families living on the Main Street of a town. Consider a scenario

- Family Jones lives at first Main Street.
- Family Smith lives at second Main Street.
- Family Brown lives at third Main Street.
- Family Taylor lives at fourth Main Street.
- Family Clark lives at fifth Main Street.

We can express this information in a table

Address	Family
first	Jones
second	Smith
third	Brown
fourth	Taylor
fifth	Clark

Similarly, in a computer's memory, either zero or one is stored in each memory location, something like the following

- zero is stored at the first location.
- zero is stored at the second location.
- one is stored at the third location.
- zero is stored at the fourth location.
- one is stored at the fifth location.

We can also express this as a table

Address	Value
first	zero
second	zero
third	one
fourth	zero
fifth	one

For the most part of this book, we consider more than one bit. Actually, for the most part of this book, we do not worry about the *size* of data because we do not need to worry about the details.

Remembering billions of locations would be quite difficult for a programmer, even though a computer has no problem at all. A simpler way uses *symbols*, such as `counter` or `sum`. Compilers (such as `gcc`) translate the symbols into addresses. You, as a programmer, do not need to worry about what addresses to use. Symbols are for human; computers do not need symbols. Inside a computer's memory, there are only addresses and values.

Consider the following sample code

```
int a = 5;
double b = 6.7;
char z = 'c';
```

They will be stored somewhere inside a computer's memory, something like the following

Symbol	Address	Value
a	100	5
b	104	6.7
z	112	'c'

The address of b is $4 +$ the address of a because we assume an integer takes 4 bytes. We assume a double-precision floating point number takes 8 bytes. Thus, z 's address is $8 + b$'s address. Again, in most cases, we don't need to worry whether a symbol's size is 4 or 8 or something else, as long as each symbol has a distinct address. The addresses are not reused: two variables (a , b , and z in this example) will always have different addresses.

2.2 Stack Memory

In computers, memory is organized into two types: *stack memory* and *heap memory*. This chapter focuses on stack memory; Chapter 5 will explain heap memory. Stack memory follows a strict rule: *first in, last out*. New data can enter the stack memory at only the top. Data can be retrieved at only the top. By convention, we say top instead of bottom, even though the concept is the same. When a piece of data is stored into the stack memory, we say the piece is *pushed* into the stack. When a piece of data is retrieved from the stack memory, we say the piece is *popped* from the stack. Figure 2.1 illustrates the two operations (push and pop) of stack memory.

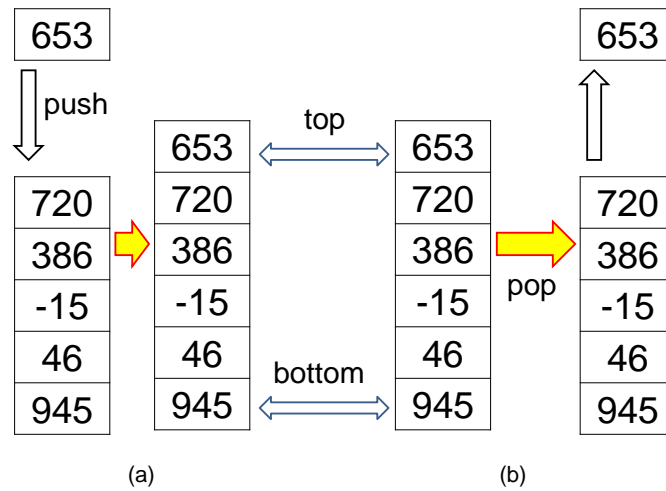
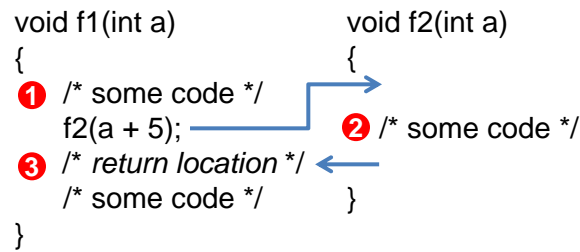


Figure 2.1: Push and pop of stack. (a) Data are pushed (stored) onto a stack. Originally, the top of the stack stores number 720. Number 653 is pushed to the top of the stack. (b) Data are retrieved (popped) from the stack. Push and pop can occur at only the top of the stack. This figure uses integers as an example. A stack can store any type of data: integer, character, double, string, or programmer-defined types as explained in Chapter 7.

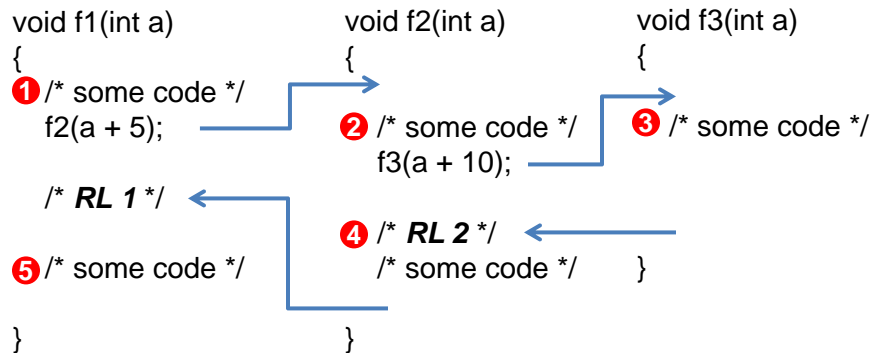
2.3 Function Return Location



Figure 2.2: (a) The stack changes after f_1 , f_2 , and f_3 are called. Each function call pushes the called function onto the stack. (b) When f_3 finishes, it is popped from the stack and now f_2 is at the top. When f_2 finishes, it is popped from the stack and f_1 is at the top.



(a)



(b)

Figure 2.3: Compilers (such as `gcc`) mark the return location (RL) after each function call.

Why is stack memory important? Stack memory stores the *order* of function calls. Recall Figure 1.1 (b). Function `f1` calls function `f2` and `f2` calls function `f3`. When `f3` finishes, the program continues `f2`, from the place just below calling `f3`. Figure 2.2 shows what is in the stack memory

Actually, the stack does not store the *names* of the functions (i.e. not `f1`, `f2`, or `f3`). The stack stores the *return locations* (RL) after the function calls. The return locations are the locations immediately after the function calls. Figure 2.3 redraws Figure 1.1 by adding return locations. Return locations are added by compilers (such as `gcc`). As a programmer, you do not need to do anything (actually, you cannot do anything) for creating return locations. Return locations are stored to the *call stack*. In Figure 2.3(b), RL1 is pushed to the stack first before `f2` starts. When `f2` calls `f3`, RL2 is pushed to the stack before `f3`. When `f3` finished, RL2 is popped from the stack and the program continues from RL2. When `f2` finishes, RL1 is popped from the stack and the program continues from there.

Why are the return locations stored, not the functions' names? The reason is quite simple: a function may call another function multiple times. Consider this example:

```

void f1(int a)
{
  /* some code */
  f2(a + 5); /* first call */
}

```

```
/* return location 1 */  
f2(a + 10); /* second call */  
/* return location 2 */  
/* some more code */  
}
```

Function `f2` is called in two different locations inside `f1`. After the first call, the program continues from return location 1. After the second call, the program continues from return location 2. In a program, every statement has a location. Right now, we need to worry about the statement after a function call. This is the location to continue the program after the function finishes. Section 12.1 uses the `gdb` debugger to show the locations of different statements.

2.4 Call Stack

The place where the return locations are stored is the *call stack*. The call stack stores more than return locations. The call stack also stores *function arguments* and *local variables*. In Figure 2.3, each function has an argument called *a*. The arguments are also stored in the call stack.

```
void f1(int a)
{
    int b = 4;
    char s = 'm';
    /* some code */
}
void f2(int a)
{
    /* some code */
    f1(5);
    /* return location */
    /* some more code */
}
```

When `f1` is called by `f2`, the return location is pushed to the call stack:

Symbol	Address	Value
-	100	return location

Your program cannot directly access the return location, so there is no symbol associated with the return location. The address is given by the operating system. For simplicity, we use 100 here. Then, argument `a` is pushed to the call stack and the value 5 is assigned to `a`.

Symbol	Address	Value
a	101	5
-	100	return location

Next, local variable `b` is pushed to the call stack and its value is 4.

Symbol	Address	Value
b	102	4
a	101	5
-	100	return location

Last, local variable `s` is pushed to the call stack and its value is `'m'`;

Symbol	Address	Value
<code>s</code>	103	<code>'m'</code>
<code>b</code>	102	<code>4</code>
<code>a</code>	101	<code>5</code>
<code>-</code>	100	return location

This is called the *frame* of function `f1`. A frame includes (1) the return location after the function finishes, (2) arguments, and (3) local variables. When `f1` finishes and the program continues execution in `f2`, the whole frame is popped from the call stack. Operating systems ensure that addresses are not shared. It is impossible to have `a` and `b` occupying the same memory address.

Now, let's consider a more complex situation with three functions.

```
void f1(int a)
{
    int b = 4;
    char s = 'm';
    /* some code */
}

void f2(int a)
{
    int b = 19;
    int s = -36;
    /* some code */
    f1(a + 5);
    /* return location 2 */
    /* some more code */
}

void f3(int a)
{
    /* some code */
    f2(5);
    /* return location 1 */
    /* some more code */
}
```

When `f3` calls `f2(5)`, return location 1 is pushed to the call stack.

Symbol	Address	Value
-	100	return location 1

Next, $f2$'s argument a is pushed to the stack and its value is 5.

Symbol	Address	Value
a	101	5
-	100	return location 1

The local variables of $f2$ are pushed to the stack.

Symbol	Address	Value
s	103	-36
b	102	19
a	101	5
-	100	return location 1

Here comes the interesting part: $f2$ calls $f1$. What happens to the call stack? We just follow the same procedure: push the return location, the argument (or arguments if there are multiple), and the local variables.

Symbol	Address	Value
s	107	'm'
b	106	4
a	105	10
-	104	return location 2
s	103	-36
b	102	19
a	101	5
-	100	return location 1

Please remember that the call stack is divided into frames; each frame corresponds to one function call.

Frame	Symbol	Address	Value
$f1$	s	107	'm'
	b	106	4
	a	105	10
	-	104	return location 2
$f2$	s	103	-36
	b	102	19
	a	101	5
	-	100	return location 1

When `f1` finishes, its frame is popped from the stack. When `f2` finishes, its frame is also popped from the stack. You probably want to ask whether there is a frame for `f3`. The answer is yes. The frame for `f3` is constructed using the same procedure when `f3` is called. Every function has its own frame. The frame for the `main` function should be at the very bottom of the call stack.

A function's frame is pushed onto the call stack when the function is called during execution. If the function is not called, its frame is not on the call stack. Consider this example

```
void f1(int a)
{
    if (a == 0)
    {
        f2(a);
    }
    else
    {
        f3(a);
    }
}
int main(int argc, char * argv[])
{
    f1(1);
    return EXIT_SUCCESS;
}
```

The `main` function calls `f1` with `a = 1`. Because of the condition, `f2` is not called and the call stack never has `f2`'s frame.

2.5 Scope

After understanding the call stack and the frame of each function, we can answer the question on page 17: We cannot use

```
int b = 6;
int b = 16;
```

inside the same function. Otherwise, `gcc` says,

```
error: redefinition of b
```

Why can we use the same variable name *a*, *b*, or *s* in multiple functions without problems? The answer is that each frame is a separate *scope*. We can use the same symbols in different scopes. In a C program, each function has to follow this rule:

A function can read or write variables and arguments only inside its own frame. Within each frame, a symbol can appear only once.

In the call stack shown on page 27, for *f1*, the symbol *a* refers to the symbol at address 105. The function cannot read or modify the symbol *a* at address 101. Similarly, for *f1*, the symbol *b* refers to address 106. If we change *f1* to

```
int f1(int a)
{
    int b = 29;
    char s = 'u';
    /* some code */
}
```

The value stored at address 106 is changed to 29; the value stored at address 102 is unchanged.

Frame	Symbol	Address	Value
f1	s	107	'u'
	b	106	29
	a	105	10
	-	104	return location 2
f2	s	103	-36
	b	102	19
	a	101	5
	-	100	return location 1

Consider another example:

```
/* file: fab.c
   purpose: changing the arguments would not change the values
   in the caller
*/
#include <stdio.h>
#include <stdlib.h>
void f1(int a, int b)
{
```

```

printf("beginning of f1(%d,%d)\n", a, b);
a = 5;
b = -5;
printf("end of f1(%d,%d)\n", a, b);
}

void f2(void)
{
int a = 10;
int b = -10;
printf("beginning of f2(%d, %d)\n", a, b);
f1(a, b);
/* return location 2 */
printf("end of f2(%d, %d)\n", a, b);
}

int main(int argc, char * argv[])
{
f2();
/* return location 1 */
return EXIT_SUCCESS;
}

```

The output of this program is

```

beginning of f2(10, -10)
beginning of f1(10,-10)
end of f1(5,-5)
end of f2(10, -10)

```

Please notice that `a` and `b` are changed inside `f1`. However, when `f1` finishes, the values of `a` and `b` are unchanged. Let's use the call stack to understand what has happened.

When `main` calls `f2`, there is no argument. There are two local variables `a` and `b`; `f2`'s frame is shown below.

Frame	Symbol	Address	Value
f2	b	102	-10
	a	101	10
	-	100	return location 1

Right at the beginning of `f1`, the call stack contains two frames. The values of `a` and `b` are 10 and -10 because the caller (i.e. `f2`) assigns 10 and -10 to `a` and `b`.

Frame	Symbol	Address	Value
f1	b	105	-10
	a	104	10
	-	103	return location 2
f2	b	102	-10
	a	101	10
	-	100	return location 1

Inside f1, a and b are changed to 5 and -5. The changes are reflected in the call stack.

Frame	Symbol	Address	Value
f1	b	105	-5
	a	104	5
	-	103	return location 2
f2	b	102	-10
	a	101	10
	-	100	return location 1

When f1 finishes, its frame is popped and the call stack has only f2's frame.

Frame	Symbol	Address	Value
f2	b	102	-10
	a	101	10
	-	100	return location 1

The values of a and b are 10 and -10, not affected by the changes inside f1.

2.6 Return Value

If a function has no return value, `void` is added in front of the function. Otherwise, the return type is added. The following example contains three functions (including `main`) returning `int` and `char`.

```

/* file: freturn.c
   purpose: functions with return values
*/
#include <stdio.h>
#include <stdlib.h>
int f1(int a, int b)
{

```

```

int c;
c = a + b;
return c;
}

char f2(void)
{
    int a = 10;
    int b = -10;
    int d;
    d = f1(a, b);
    /* return location 2 */
    if (d == 0)
    {
        return 'y';
    }
    return 'n';
}

int main(int argc, char * argv[])
{
    char s;
    s = f2();
    /* return location 1 */
    printf("s = %c\n", s);
    return EXIT_SUCCESS;
}

```

How does `gcc` handle return values? It pushes the *address* of the variable to the frame so that the returned value is assigned to the correct address. Earlier on page 26, we said a frame includes (1) the return location after the function finishes, (2) arguments, and (3) local variables. Now, we need to add the fourth: the address for storing the return value.

Shown below is the call stack after `f2` is called, before `f1` is called. The value returned by `f2` is stored in `s`; thus, `f2`'s frame includes the address of `s`. This address is stored just above the return location.

Frame	Symbol	Address	Value
f2	d	105	?
	b	104	-10
	a	103	10
	-	102	value address 100
	-	101	return location 1
main	s	100	?

The frame of `f1` is pushed to the call stack when the program starts executing `f1`, just before

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

Frame	Symbol	Address	Value
f1	c	110	?
	b	109	-10
	a	108	10
	-	107	value address 105
	-	106	return location 2
f2	d	105	?
	b	104	-10
	a	103	10
	-	102	value address 100
	-	101	return location 1
main	s	100	?

The value returned from `f1` is stored in `d` and its address, 105, is stored in `f1`'s frame.

The values of `c` in `f1`, `d` in `f2`, and `s` in `main` are unknown right now, because they have not been assigned yet (waiting for the called functions). Therefore, we use "?" for the values. *You cannot assume that uninitialized variables are zero.* If we do not assign values, the values can be anything.

Next, inside `f1`, `c` is the sum of `a` and `b`. Since `a` is 10 and `b` is -10, their sum is 0 and this value is assigned to `c`.

Frame	Symbol	Address	Value
f1	c	110	0
	b	109	-10
	a	108	10
	-	107	value address 105
	-	106	return location 2
f2	d	105	?
	b	104	-10
	a	103	10
	-	102	value address 100
	-	101	return location 1
main	s	100	?

Function `f1` returns `c`'s value and this value is stored at address 105, i.e. `d`'s address. Just before popping `f1`'s frame, `d`'s value has been changed to 0.

Frame	Symbol	Address	Value
f1	c	110	0
	b	109	-10
	a	108	10
	-	107	value address 105
	-	106	return location 2
f2	d	105	0
	b	104	-10
	a	103	10
	-	102	value address 100
	-	101	return location 1
main	s	100	?

Afterwards, f1's frame is popped.

Frame	Symbol	Address	Value
f2	d	105	0
	b	104	-10
	a	103	10
	-	102	value address 100
	-	101	return location 1
main	s	100	?

Since d is 0, f2 will return 'y'. This return value is copied to the address of 100, i.e. s inside main.

Frame	Symbol	Address	Value
f2	d	105	0
	b	104	-10
	a	103	10
	-	102	value address 100
	-	101	return location 1
main	s	100	'y'

The frame of f2 is popped and the call stack is shown below.

Frame	Symbol	Address	Value
main	s	100	'y'

The procedure seems complex. Fortunately, `gcc` takes care of copying values to arguments when calling functions, and copying return values when functions finish. We have been using 100, 101, ... for the addresses in these examples. As mentioned earlier, the addresses of the call stack are determined by operating systems and we have no control. Thus, the addresses are for explanation purposes only and do not reflect the actual addresses in any specific computer.

Section 1.3 mentioned that a C program may call a function and discard the returned value. For example,

```
int f(int x)
{
    return (x + 5);
}

int g(int y)
{
    f(y); // do not store the returned value
}
```

Even though `f` returns an integer, the value is not stored anywhere in the caller `g`. No value address is pushed to the call stack.

2.7 Practice Problems

1. Draw the call stack of this program just before

```
int c = 3;
```

```
#include <stdio.h>
#include <stdlib.h>
void f(int i, int j)
{
    int a = i + j;
    int b = i - j;
    int c = 3;
}
int main(int argc, char * argv[])
{
    f(6, 9);
    return EXIT_SUCCESS;
}
```


2. Draw the call stack of this program when the program reaches Label 1, Label 2, and Label 3.

```
#include <stdio.h>
#include <stdlib.h>
int f(int i, int j)
{
    int a, b, c;
    a = i + j;
    b = i - j;
    c = 3 + b;
    /* Label 2 */
    return c;
}
int main(int argc, char * argv[])
{
    int a = 6;
    int b = 9;
    int c = 264;
    /* Label 1 */
    c = f(a, b);
    /* Label 3 */
    return EXIT_SUCCESS;
}
```

Chapter 3

Recursion

Recursion uses the concept of *divide and conquer*: divide a complex problem into smaller problems and solve (conquer) the smaller problems. Recursion is a way to solve problems by following these steps:

1. identify the parameters of a problem.
2. express the solution based on the parameters.
3. determine the simple cases when the solutions are “obvious”.
4. derive the relationships between complex cases and simpler cases.

Let’s consider three examples.

3.1 Examples

3.1.1 Select Balls

Suppose you have many (unlimited) red and blue balls. You want to select n balls but red balls cannot be adjacent. How many ways can you select the balls?

Step 1: For this problem, the number of balls, n , is the parameter.

Step 2: Let $f(n)$ be the answer: how many ways we can select the balls.

Step 3: If only one ball is selected, we have two choices: red or blue. Thus, $f(1)$ is 2. If only two balls are selected, there are three options: (i) Red, Blue, (ii) Blue, Red, and (iii) Blue, Blue. Therefore, $f(2)$ is 3.

Step 4: If there are n balls ($n > 2$), the first ball can be red or blue. If the first ball is blue, it cannot be red, and vice versa. Therefore, there are exactly two choices for the first ball. We consider the two different scenarios: (i) If the first ball is blue, the remaining $n - 1$ balls must follow the same rule: no red balls are adjacent. There are $f(n - 1)$ options. (ii) If the first ball is red, the second ball must be blue. The remaining $n - 2$ balls must follow the same rule and there are $f(n - 2)$ options. Thus, we can express $f(n)$ as the sum $f(n - 1) + f(n - 2)$.

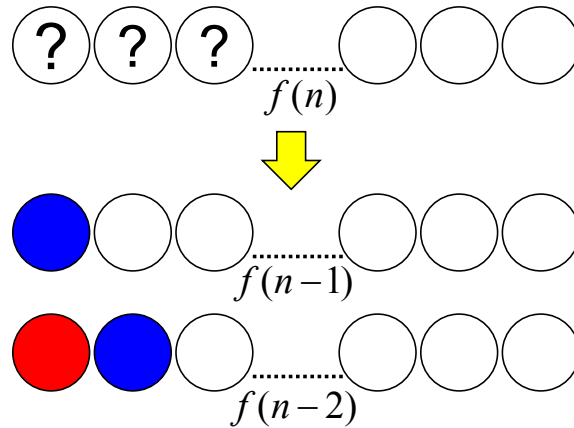


Figure 3.1: To decide $f(n)$, we consider the options for the first ball. If the first ball is blue, the remaining $n - 1$ balls have $f(n - 1)$ options. If the first ball is red, the second ball must be blue and remaining $n - 2$ balls have $f(n - 2)$ options.

$$f(n) = \begin{cases} 2 & \text{if } n \text{ is } 1 \\ 3 & \text{if } n \text{ is } 2 \\ f(n - 1) + f(n - 2) & \text{if } n > 2 \end{cases} \quad (3.1)$$

3.1.2 One-Way Street

Imagine a city in which the streets are either east-west bound or north-south bound shown as the grids in Figure 3.2. The city has severe traffic jam during rush hours so the city government considers to adopt a rule: during rush hours, cars can move only east bound or north bound¹. If you have the locations of the origin and the destination, how many ways can you reach the destination by driving?

Figure 3.2 marks three pairs of origins and destinations: $A \rightarrow B$, $C \rightarrow D$, and $E \rightarrow F$. How many turning options does a driver have going from one origin to the corresponding destination? For the first two pairs $A \rightarrow B$ and $C \rightarrow D$, the driver has only one option. This is shown in Figure 3.3 (a) and (b). In contrast, there are some options for $E \rightarrow F$. At E , the driver can go east bound first or north bound first, as indicated by the two arrows in Figure 3.3 (c). The question is the number of options.

¹All right, this example does not make sense in reality but this is a nice example for recursion.

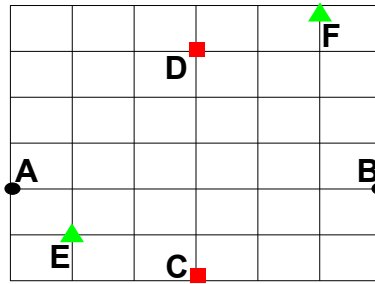


Figure 3.2: The city's streets are either east-west bound or north-south bound. A car can move only east bound or north bound.

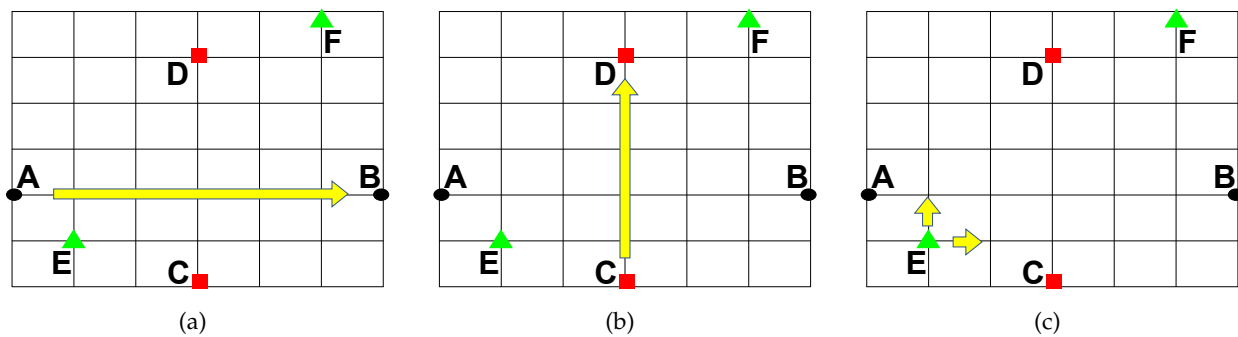


Figure 3.3: (a) A driver cannot turn anywhere from A to B. (b) A driver cannot turn anywhere from C to D. (c) There are some turning options from E to F. At E, the driver can go north bound first or east bound first, as indicated by the two arrows.

To answer this question, we have to first identify the parameters. Suppose E is at the intersection of (x_1, y_1) and F is at the intersection of (x_2, y_2) . The distance between them can be expressed as $(\Delta x, \Delta y) = (x_2 - x_1, y_2 - y_1)$. This is the first step.

Step 2: Let $f(\Delta x, \Delta y)$ express the number of solutions.

Step 3: If $\Delta x < 0$, the destination is at the west side of the origin. Thus, there is no solution. Also, there is no solution if $\Delta y < 0$. $\Delta x = \Delta y = 0$ is a special case. It means that the destination is the same as the origin. We can say that there is no solution or there is one solution— do not do anything. Either definition makes sense but we choose the latter definition. If $\Delta x > 0$ and $\Delta y = 0$ (the case $A \rightarrow B$), there is one solution. If $\Delta x = 0$ and $\Delta y > 0$ (the case $C \rightarrow D$), there is also one solution. These are the simple cases whose answers can be found easily.

$$f(\Delta x, \Delta y) = \begin{cases} 0 & \text{if } \Delta x < 0 \text{ or } \Delta y < 0 \\ 1 & \text{if } \Delta x = 0 \text{ and } \Delta y \geq 0 \\ 1 & \text{if } \Delta x \geq 0 \text{ and } \Delta y = 0 \end{cases} \quad (3.2)$$

Step 4: When $\Delta x > 0$ and $\Delta y > 0$ (the case $E \rightarrow F$), the driver has options at the origin (i.e. E): going north first or going east first. If the driver goes north first, the new origin is at $(x_1, y_1 + 1)$.

There are $f(\Delta x, \Delta y - 1)$ options left. If the driver goes east first, the new origin is at $(x1 + 1, y1)$. There are $f(\Delta x - 1, \Delta y)$ options left. These are the only two possible options at E and they are exclusive. Therefore, we can express $f(\Delta x, \Delta y) = f(\Delta x, \Delta y - 1) + f(\Delta x - 1, \Delta y)$ if $\Delta x > 0$ and $\Delta y > 0$.

$$f(\Delta x, \Delta y) = \begin{cases} 0 & \text{if } \Delta x < 0 \text{ or } \Delta y < 0 \\ 1 & \text{if } \Delta x = 0 \text{ and } \Delta y \geq 0 \\ 1 & \text{if } \Delta x \geq 0 \text{ and } \Delta y = 0 \\ f(\Delta x, \Delta y - 1) + f(\Delta x - 1, \Delta y) & \text{if } \Delta x > 0 \text{ and } \Delta y > 0 \end{cases} \quad (3.3)$$

3.1.3 Integer Partition

Integer partition means breaking a positive integer into the sum of some positive integers. For example, 5 can be broken into the sum of $1 + 2 + 2$ or $2 + 3$. Two partitions are different if they use different sets of integers. For example, the first partition uses 1 but not 3. The second partition uses 3 but not 1. These two partitions are different. We also consider the order of the integers used: $1 + 2 + 2$ and $2 + 1 + 2$ are different because 1 appears in different positions. The question is the number of different partitions for a positive integer n .

Step 1: The number n is naturally the parameter for the problem.

Step 2: Let $f(n)$ be the number of different partitions for integer n .

Step 3: When n is 1, there is only one way to partition the number: itself. When n is 2, there are two ways: $1 + 1$ and 2. Thus, $f(1) = 1$ and $f(2) = 2$.

Step 4: When n is larger than 2, we consider the possible options for the first number.

Total	First Number	Remaining Value
n	1	$n - 1$
	2	$n - 2$
	3	$n - 3$
	...	
	$n - 2$	2
	$n - 1$	1
	n	0

These are all possible cases and they are exclusive. If the first number is 1, the first number cannot be 2.

If the first number is 1, the remaining value is $n - 1$. How many ways can this value be partitioned? By definition, it is $f(n - 1)$. If the first number is 2, the remaining value is $n - 2$. How many ways can this value be partitioned? By definition, it is $f(n - 2)$.

The value of $f(n)$ is the sum of all different cases when the first number is 1, or 2, or 3, ..., or $n - 1$, or n itself. Now, we can express $f(n)$ as

$$f(n) = \begin{cases} 1 & \text{if } n \text{ is } 1 \\ f(n-1) + f(n-2) + \dots + f(1) + 1 = 1 + \sum_{i=1}^{n-1} f(i) & \text{if } n > 1 \end{cases} \quad (3.4)$$

3.1.4 Tower of Hanoi

This is a popular example for explaining recursion. Some disks of different sizes are stacked on one pole. The disks are arranged so that smaller disks are above larger disks. The problem is to move the disks to another pole. Only one disk can be moved each time. At any moment smaller disks are always above larger disks. A third pole is used to hold some disks when necessary. Figure 3.4 illustrates the problem. If there are n disks, how many steps are needed?

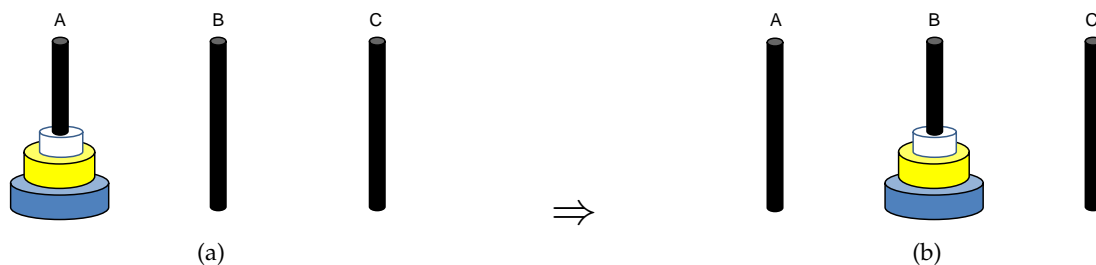


Figure 3.4: Tower of Hanoi. (a) Some disks are arranged along pole A and the goal is to move all disks to pole B, as shown in (b). A larger disk can never be placed above a smaller disk. A third pole, C, is used when necessary.

Let's first consider moving only one disk from A to B. This is the simplest case and we can move that disk directly from A to B as shown in Figure 3.5.

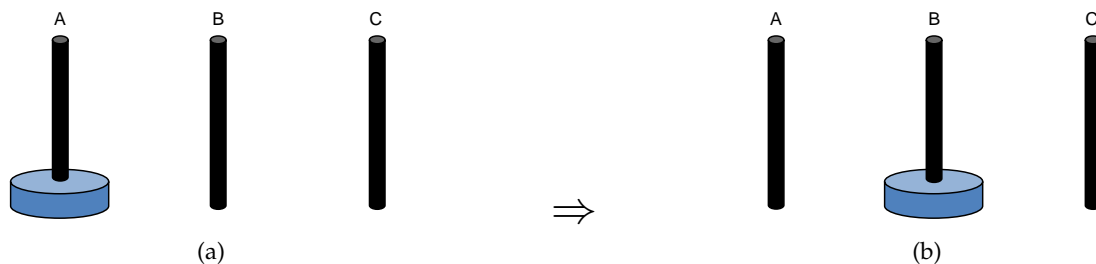


Figure 3.5: Moving one disk from A to B requires only one step.

Moving two disks requires more work. We cannot move the smaller disk (yellow disk) to B and then move the larger disk (blue disk) to B. Doing so would place the larger disk above the smaller disk. This violates the rule. Instead, we have to move the smaller disk "somewhere else",

i.e. C, before moving the larger disk to B. Then, we move the larger disk from A to B and move the smaller disk from C to B. The steps are illustrated in Figure 3.6.

As illustrated in Figure 3.6, when there are two disks, the problem can be solved in three steps. Can you think of a solution that requires fewer steps for two disks?

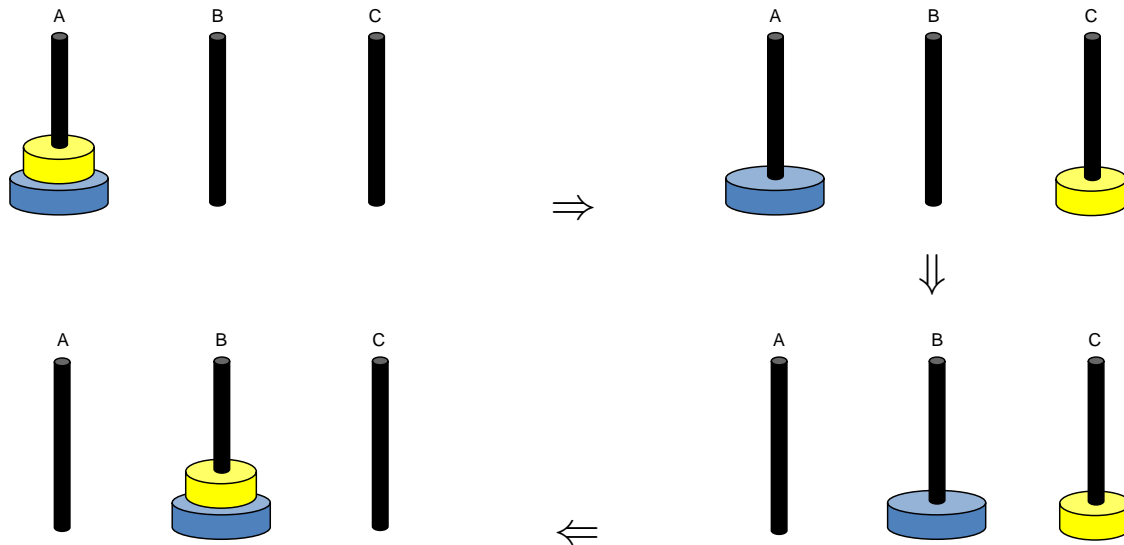


Figure 3.6: Moving two disks from A to B requires three steps.

If you are not quite sure how to solve the problem yet, let's try one more example using three disks. Figure 3.7 illustrates how to move three disks. Totally seven steps are needed. If you study the figure carefully, you can find that the first three steps and the last three steps are somewhat similar. The first three steps move the top two disks from A to C. The last three steps move the top two disks from C to B. Between these steps is the fourth step and it moves the largest disk from A to B.

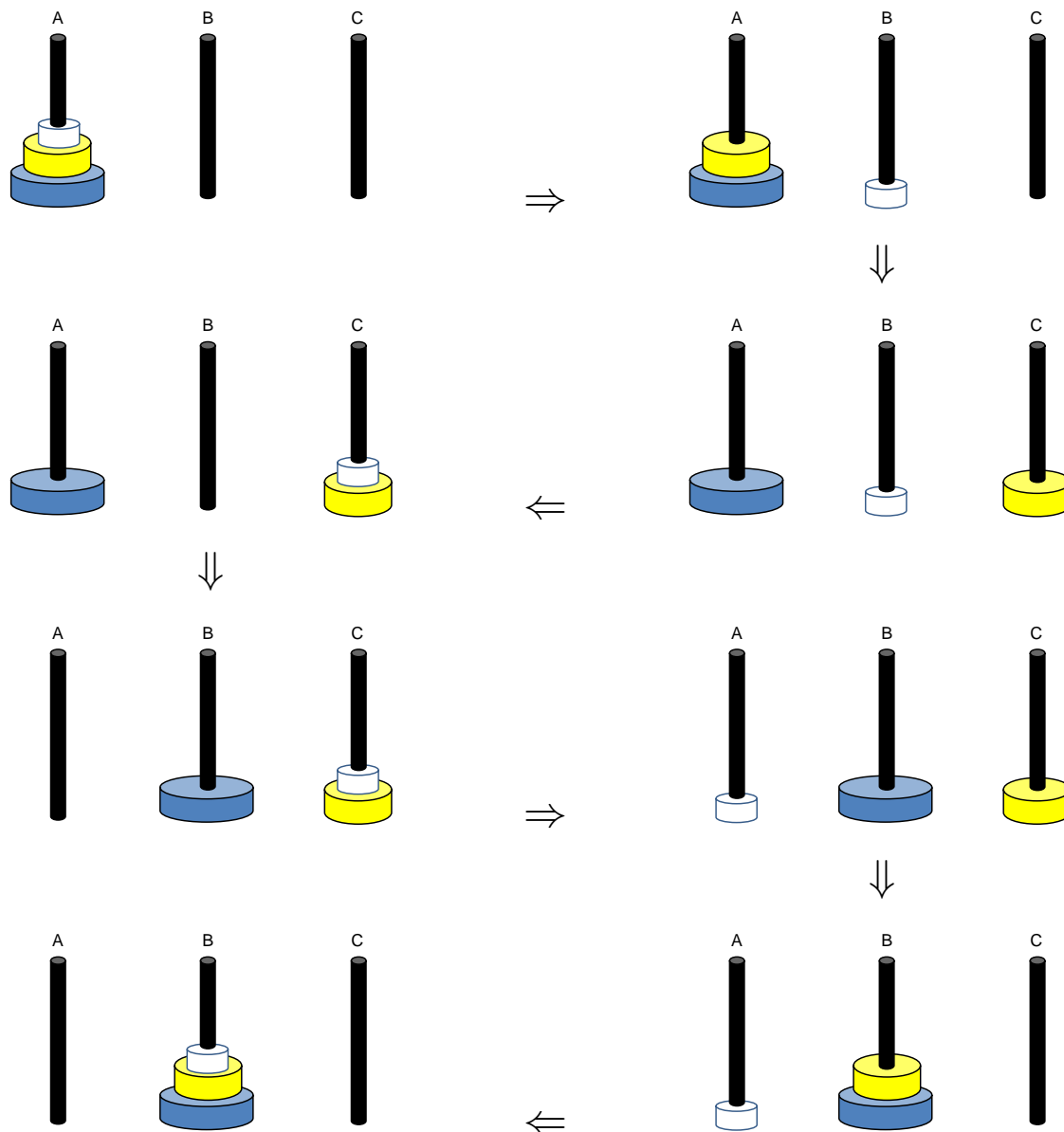


Figure 3.7: Moving two disks from A to B requires three steps.

Can you develop a general strategy for n disks? If there is only one disk (i.e. n is one), the problem can be solved easily. Otherwise, our solution is divided into three parts: (i) move the first $n - 1$ disks from A to C, (ii) move the largest disk from A to B, and (iii) move the first $n - 1$ disks from C to B. How many steps are needed for moving n disks?

Step 1: The number n is naturally the parameter for the problem.

Step 2: Let $f(n)$ be the answer: how many steps are needed to move n disks from A to B.

Step 3: If n is one, only one step is sufficient; thus, $f(1)$ is 1.

Step 4: When n is greater than one, the problem can be divided into three parts. Part (i) needs to move $n - 1$ disks so it needs $f(n - 1)$ steps. Part (ii) needs to move only one disk so it needs one step. Part (iii) needs to move $n - 1$ disks so it needs $f(n - 1)$ steps. Thus, we can obtain the following formula:

$$f(n) = \begin{cases} 1 & \text{if } n \text{ is } 1 \\ f(n - 1) + 1 + f(n - 1) = 2f(n - 1) + 1 & \text{if } n \geq 2 \end{cases} \quad (3.5)$$

3.2 C Implementation

This section shows how to convert the mathematical formulas to C programs. At the beginning of this chapter, we said there are four steps for solving a problem using recursion. The steps are translated into the following C code in a recursive function.

Step 1: identify the parameters of a problem \Rightarrow the arguments of a function

Steps 2 and 3: express the solution based on the parameters and determine the simple cases when the solutions are “obvious” \Rightarrow each simple case is detected by a condition, i.e. an `if` statement. If one condition is true, the problem is solved directly.

Step 4: derive the relationships between complex cases and simpler cases \Rightarrow If none of the `if` conditions is true, this is a complex case. Call the function itself with a (or several) modified argument (or arguments).

A recursive function has the following structure:

```
return_type func(arguments)
{
    if (this is a simple case) /* by checking the arguments */
    {
        solve the problem
    }
    else
    {
        call func by simplifying the arguments
    }
}
```

The `if` condition (or conditions) is the *terminating condition* (or conditions) of the recursive function. When the condition is true, the problem is simple enough and recursive calls are unnecessary.

3.2.1 Select Balls

```
/* file: balls.c
   purpose: implement the recursive relation for selecting balls
   */
#include <stdio.h>
#include <stdlib.h>
int f(int m)
/* use m instead of n so that we can distinguish m in f and n in
   main */
{
    int a, b;
    if (m <= 0)
        {
            printf("Invalid Number %d. It must be positive.\n", m);
            return -1;
        }
    if (m == 1)
        {
            /* two options when only one ball is selected */
            return 2;
        }
    if (m == 2)
        {
            /* three options when only two balls are selected */
            return 3;
        }
    a = f(m - 1);
    /* return location 2 */
    b = f(m - 2);
    /* return location 3 */
    return (a + b);
}

int main(int argc, char * argv[])
{
    int c;
    int n;
    if (argc < 2)
        {
            printf("need 1 integer.\n");
            return EXIT_FAILURE;
        }
    n = (int) strtol(argv[1], NULL, 10);
    c = f(n);
    /* return location 1 */
}
```

```

printf("f(%d) = %d.\n", n, c);
return EXIT_SUCCESS;
}

```

Executing the program with different arguments can produce the following results:

n	$f(n)$
1	2
2	3
3	5
4	8
5	13
6	21

Let's examine the call stack when n (in main) is 4. Assume that there are two arguments and `argv[1]` is "4".

Frame	Symbol	Address	Value
main	n	103	4
	c	102	?
	argv	101	‡
	argc	100	2

‡ `argv` stores the addresses of the arguments.
 We do not need to worry about it for now.

Calling f will change the call stack as follows

Frame	Symbol	Address	Value
f	b	108	?
	a	107	?
	m	106	4
	-	105	value address 102
	-	104	return location 1
main	n	103	4
	c	102	?
	argv	101	‡
	argc	100	2

Since m is not negative, 0, 1, or 2, the program will call $f(m - 1)$ and assign the result in `a`. Recursive call follows the same procedure pushing return location, value address, arguments, and local variables onto the call stack.

Frame	Symbol	Address	Value
f	b	113	?
	a	112	?
	m	111	3
	-	110	value address 107
	-	109	return location 2
f	b	108	?
	a	107	?
	m	106	4
	-	105	value address 102
	-	104	return location 1
main	n	103	4
	c	102	?
	argv	101	‡
	argc	100	2

The value of m is still not negative, 0, 1, or 2. Thus, the program calls $f(m - 1)$ again.

Frame	Symbol	Address	Value
f	b	118	?
	a	117	?
	m	116	2
	-	115	value address 112
	-	114	return location 2
f	b	113	?
	a	112	?
	m	111	3
	-	110	value address 107
	-	109	return location 2
f	b	108	?
	a	107	?
	m	106	4
	-	105	value address 102
	-	104	return location 1
main	n	103	4
	c	102	?
	argv	101	‡
	argc	100	2

The value of m is 2 so the function returns 3. This return value is written to the value address 112.

Frame	Symbol	Address	Value
f	b	118	?
	a	117	?
	m	116	2
	-	115	value address 112
	-	114	return location 2
f	b	113	?
	a	112	3
	m	111	3
	-	110	value address 107
	-	109	return location 2
f	b	108	?
	a	107	?
	m	106	4
	-	105	value address 102
	-	104	return location 1
main	n	103	4
	c	102	?
	argv	101	‡
	argc	100	2

Then, the top frame is popped.

Frame	Symbol	Address	Value
f	b	113	?
	a	112	3
	m	111	3
	-	110	value address 107
	-	109	return location 2
f	b	108	?
	a	107	?
	m	106	4
	-	105	value address 102
	-	104	return location 1
main	n	103	4
	c	102	?
	argv	101	‡
	argc	100	2

Next, the program calls $f(m - 2)$. The return value is stored in b.

Frame	Symbol	Address	Value
f	b	118	?
	a	117	?
	m	116	1
	-	115	value address 113
	-	114	return location 3
f	b	113	?
	a	112	3
	m	111	3
	-	110	value address 107
	-	109	return location 2
f	b	108	?
	a	107	?
	m	106	4
	-	105	value address 102
	-	104	return location 1
main	n	103	4
	c	102	?
	argv	101	‡
	argc	100	2

This time the function returns 2 because m is 1. The return value is written to the address 113.

Frame	Symbol	Address	Value
f	b	118	?
	a	117	?
	m	116	1
	-	115	value address 113
	-	114	return location 3
f	b	113	2
	a	112	3
	m	111	3
	-	110	value address 107
	-	109	return location 2
f	b	108	?
	a	107	?
	m	106	4
	-	105	value address 102
	-	104	return location 1
main	n	103	4
	c	102	?
	argv	101	‡
	argc	100	2

The top frame is popped.

Frame	Symbol	Address	Value
f	b	113	2
	a	112	3
	m	111	3
	-	110	value address 107
	-	109	return location 2
f	b	108	?
	a	107	?
	m	106	4
	-	105	value address 102
	-	104	return location 1
main	n	103	4
	c	102	?
	argv	101	‡
	argc	100	2

The values of a and b are available so the function returns their sum and this value is stored at address 107.

Frame	Symbol	Address	Value
f	b	113	2
	a	112	3
	m	111	3
	-	110	value address 107
	-	109	return location 2
f	b	108	?
	a	107	5
	m	106	4
	-	105	value address 102
	-	104	return location 1
main	n	103	4
	c	102	?
	argv	101	‡
	argc	100	2

The top frame is popped.

Frame	Symbol	Address	Value
f	b	108	?
	a	107	5
	m	106	4
	-	105	value address 102
	-	104	return location 1
main	n	103	4
	c	102	?
	argv	101	‡
	argc	100	2

The program calls $f(m - 2)$ and stores the result in b.

Frame	Symbol	Address	Value
f	b	113	-
	a	112	-
	m	111	2
	-	110	value address 108
	-	109	return location 3
f	b	108	?
	a	107	5
	m	106	4
	-	105	value address 102
	-	104	return location 1
main	n	103	4
	c	102	?
	argv	101	‡
	argc	100	2

The function returns 3 and this value is written to address 108.

Frame	Symbol	Address	Value
f	b	113	-
	a	112	-
	m	111	2
	-	110	value address 108
	-	109	return location 3
f	b	108	3
	a	107	5
	m	106	4
	-	105	value address 102
	-	104	return location 1
main	n	103	4
	c	102	?
	argv	101	‡
	argc	100	2

The top frame is popped.

Frame	Symbol	Address	Value
f	b	108	3
	a	107	5
	m	106	4
	-	105	value address 102
	-	104	return location 1
main	n	103	4
	c	102	?
	argv	101	‡
	argc	100	2

The function adds the values of a and b and writes the result to address 102.

Frame	Symbol	Address	Value
f	b	108	3
	a	107	5
	m	106	4
	-	105	value address 102
	-	104	return location 1
main	n	103	4
	c	102	8
	argv	101	‡
	argc	100	2

The top frame is popped.

Frame	Symbol	Address	Value
main	n	103	4
	c	102	8
	argv	101	‡
	argc	100	2

Finally, we know the value of $f(4)$ is 8.

You may think this is tedious. Fortunately, pushing, popping, determining value addresses and return locations are all handled by the compilers. As programmers, we need to understand what happens.

3.2.2 One-Way Street

```
/* file: streets.c
   purpose: implement the recursive relation for calculating
   the number of options in a city where cars can move only
   north bound or east bound
*/
#include <stdio.h>
#include <stdlib.h>
int f(int dx, int dy)
/* Do not need to worry about dx < 0 or dy < 0
   This is already handled at main */
{
    int a, b;
    if ((dx == 0) || (dy == 0))
    {
        return 1;
    }
    a = f(dx - 1, dy);
    /* return location 2 */
    b = f(dx, dy - 1);
    /* return location 3 */
    return (a + b);
}

int main(int argc, char * argv[])
{
    int deltax, deltay;
    int c;
```

```

if (argc < 3)
{
    printf("need 2 positive integers.\n");
    return EXIT_FAILURE;
}
deltax = (int) strtol(argv[1], NULL, 10);
deltay = (int) strtol(argv[2], NULL, 10);
if ((deltax < 0) || (deltay < 0))
{
    printf("need 2 positive integers.\n");
    return EXIT_FAILURE;
}
c = f(deltax, deltay);
/* return location 1 */
printf("f(%d, %d) = %d.\n", deltax, deltay, c);
return EXIT_SUCCESS;
}

```

You should study how the call stack changes for some inputs, for example $(\Delta x, \Delta y) = (2, 3)$.

In this program, we use local variables to store the values returned from recursive calls:

```

a = f(dx - 1, dy);
/* return location 2 */
b = f(dx, dy - 1);
/* return location 3 */
return (a + b);

```

Actually, we don't need the local variables. Instead, we can write the following

```

return f(dx - 1, dy) + f(dx, dy - 1);

```

The program will do exactly the same things. If we do not use local variables, the compiler will create temporary variables for us. Using local variables makes explanation easier because we can pinpoint the return values' addresses.

3.2.3 Integer Partition

Implementing Formula (3.4) in C is shown below:

```

sum = 0;
for (i = 1; i < n; i++)

```

```

    {
        /* the first value is i */
        /* f(n - i) ways for the remaining value n - i */
        sum += f(n - i);
    }
    sum ++; /* the first value is n and the remaining is zero */

```

Notice the nearly one-to-one mapping from the mathematical expression to the C code. The complete program is shown below.

```

/* file: partition.c
   purpose: implement the recursive relation for calculating
   the number of partitions for a positive integer
*/
#include <stdio.h>
#include <stdlib.h>
int f(int n)
{
    int i;
    int sum = 0;
    if (n == 1)
    {
        return 1;
    }
    for (i = 1; i < n; i ++)
    {
        sum += f(n - i);
    }
    sum ++;
    return sum;
}

int main(int argc, char * argv[])
{
    int n;
    if (argc < 2)
    {
        printf("need one positive integer.\n");
        return EXIT_FAILURE;
    }
    n = (int) strtol(argv[1], NULL, 10);
    if (n <= 0)
    {
        printf("need one positive integer.\n");
        return EXIT_FAILURE;
    }
}

```

```

printf("f(%d) = %d.\n", n, f(n));
return EXIT_SUCCESS;
}

```

Executing the program with different arguments can produce the following results:

n	$f(n)$
1	1
2	2
3	4
4	8
5	16
6	32

You may notice a pattern: $f(n) = 2^{n-1}$. This observation is actually correct for any positive integer n . You can prove this using *mathematical induction*. Another way to prove it is shown below

$$\begin{aligned}
 f(n) &= f(n-1) + f(n-2) + \dots + f(1) + 1 \\
 f(n+1) &= f(n) + f(n-1) + f(n-2) + \dots + f(1) + 1 \\
 &= f(n) + [f(n-1) + f(n-2) + \dots + f(1) + 1] \\
 &= f(n) + f(n) = 2f(n).
 \end{aligned}
 \tag{3.6}$$

The following program prints the partitions

```

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

void printPartition(int * part, int length)
{
    int ind;
    for (ind = 0; ind < length - 1; ind ++)
    {
        printf("%d + ", part[ind]);
    }
    printf("%d\n", part[length - 1]);
}

void partition(int * part, int ind, int left)
{
    int val;
    if (left == 0)
    {

```

```

        printPartition(part, ind);
        return;
    }
    for (val = 1; val <= left; val ++){
        part[ind] = val;
        partition(part, ind + 1, left - val);
    }
}

int main(int argc, char * argv[])
{
    if (argc != 2) {
        printf("usage: q2 <number>\n");
        return EXIT_FAILURE;
    }
    int n = (int) strtol(argv[1], NULL, 10);
    if (n <= 0) {
        printf("must be a positive number\n");
        return EXIT_FAILURE;
    }
    int * arr;
    arr = malloc(sizeof(int) * n);
    partition(arr, 0, n);
    free (arr);
    return EXIT_SUCCESS;
}

```

3.2.4 Tower of Hanoi

By now, implementing the formula should be straightforward for you. The program is shown below.

```

/* file: hanoi1.c
   purpose: calculate the number of steps needed to move n disks
*/
#include <stdio.h>
#include <stdlib.h>
int f(int n)
{
    if (n == 1)
    {
        return 1;
    }
    return 2 * f(n - 1) + 1;
}

```

```

}

int main(int argc, char * argv[])
{
    int n;
    if (argc < 2)
        {
            printf("need one positive integer.\n");
            return EXIT_FAILURE;
        }
    n = (int) strtol(argv[1], NULL, 10);
    if (n <= 0)
        {
            printf("need one positive integer.\n");
            return EXIT_FAILURE;
        }
    printf("f(%d) = %d.\n", n, f(n));
    return EXIT_SUCCESS;
}

```

A more interesting program is to print which disk is moved in each step. We create a function called `move`. This function has four arguments: the number of disks to move, the source, the destination, and the additional poles. We will follow the procedure on page 43: move the top $n - 1$ disk to the additional pole (i.e. C), move the n^{th} disk from A to B, and then move the the top $n - 1$ disk from C to B.

```

/* file: hanoi2.c
   purpose: print the steps moving n disks
*/
#include <stdio.h>
#include <stdlib.h>
void move(int disk, char src, char dest, char additional)
{
    if (disk == 1)
        {
            printf("move disk 1 from %c to %c\n", src, dest);
            return;
        }
    move(disk - 1, src, additional, dest);
    printf("move disk %d from %c to %c\n", disk, src, dest);
    move(disk - 1, additional, dest, src);
}

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

```

```

if (argc < 2)
{
    printf("need one positive integer.\n");
    return EXIT_FAILURE;
}
n = (int) strtol(argv[1], NULL, 10);
if (n <= 0)
{
    printf("need one positive integer.\n");
    return EXIT_FAILURE;
}
move(n, 'A', 'B', 'C');
return EXIT_SUCCESS;
}

```

When the input is 3, the program's output is

```

move disk 1 from A to B
move disk 2 from A to C
move disk 1 from B to C
move disk 3 from A to B
move disk 1 from C to A
move disk 2 from C to B
move disk 1 from A to B

```

You can verify whether this output matches the steps in Figure 3.7.

3.2.5 Fibonacci Numbers

This is another popular example for teaching recursion. The numbers are defined as follows

$$f(n) = \begin{cases} 1 & \text{if } n \text{ is } 1 \\ 2 & \text{if } n \text{ is } 2 \\ f(n-1) + f(n-2) & \text{if } n > 2. \end{cases} \quad (3.7)$$

The following table lists the values of $f(n)$ for $n \leq 10$:

n	$f(n)$
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55

Table 3.1: The first ten values of Fibonacci Numbers.

This formula is similar to (3.1) on page 38; the difference is the starting values of $f(1)$ and $f(2)$. Instead of repeating the same explanation, here I describe a new concept: the sequence of computation. Figure 3.8 illustrates the concept. We want to compute $f(5)$ and we need to compute $f(4)$ and $f(3)$. To compute $f(4)$, we need to compute $f(3)$ and $f(2)$.

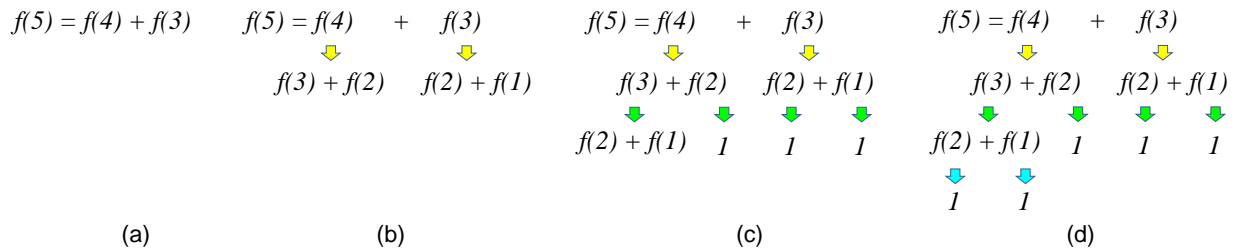


Figure 3.8: Computing $f(5)$ requires calling $f(4)$ and $f(3)$. Computing $f(4)$ requires calling $f(3)$ and $f(2)$.

$f(n-1)$	1
$f(n-2)$	2
$f(n-3)$	3
$f(n-4)$	5
$f(n-5)$	8
$f(n-6)$	13

Table 3.2: Coefficients for computing $f(n)$.

Figure 3.9 redraws Figure 3.8. This looks like a “tree”. You need to use some imagination because the tree’s “root” is at the top and the branches go downwards. Computing each value requires the sum of two values, until we reach the bottom, called *leaves*, where we can obtain the values of $f(2)$ and $f(1)$ without calling the function itself again.

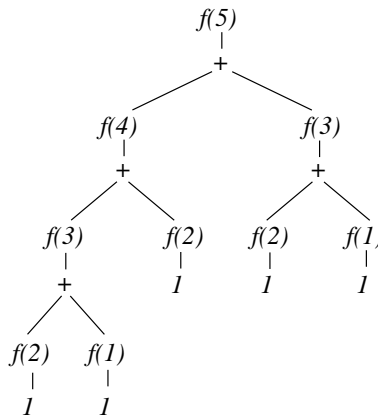


Figure 3.9: Redraw Figure 3.8. This looks like a binary tree: computing each value requires the sum of two values.

Let’s divert and do a little more mathematics here. By definition, $f(n)$ is the sum of $f(n - 1) + f(n - 2)$ and $f(n - 1)$ is the sum of $f(n - 2) + f(n - 3)$. Therefore, $f(n) = f(n - 1) + f(n - 2) = (f(n - 2) + f(n - 3)) + f(n - 2) = 2f(n - 2) + f(n - 3)$. We can continue this derivation for a few more steps:

$$\begin{aligned}
 f(n) &= f(n - 1) + f(n - 2) \\
 &= 2f(n - 2) + f(n - 3) \\
 &= 2(f(n - 3) + f(n - 4)) + f(n - 3) = 3f(n - 3) + 2f(n - 4) \\
 &= 3(f(n - 4) + f(n - 5)) + 2f(n - 4) = 5f(n - 4) + 3f(n - 5) \\
 &= 5(f(n - 5) + f(n - 6)) + 3f(n - 5) = 8f(n - 5) + 5f(n - 6) \\
 &= 8(f(n - 6) + f(n - 7)) + 5f(n - 6) = 13f(n - 6) + 8f(n - 7)
 \end{aligned} \tag{3.8}$$

The following table lists the coefficients for computing $f(n)$:

$f(n-1)$	1	$f(2)$
$f(n-2)$	2	$f(3)$
$f(n-3)$	3	$f(4)$
$f(n-4)$	5	$f(5)$
$f(n-5)$	8	$f(6)$
$f(n-6)$	13	$f(7)$

Table 3.3: Coefficients for computing $f(n)$.

If you compare this table with the values in Table 3.1, you may find that the coefficient of $f(n-k)$ is actually $f(k+1)$.

This table and Figure 3.9 express similar concepts. Figure 3.9 computes $f(5)$ so n is 5 and $f(2)$ is $f(n-3)$. The coefficient for $f(n-3)$ is 3. If you count the occurrences of $f(2)$ in Figure 3.9, you will find that it is called three times.

3.3 Practice Problems

1. In the integer partition problem, the used numbers must be positive. We add a new restriction: the number must be *non-decreasing*. How many ways can a positive integer n be partitioned? Write a program to find the answer. As an example, 5 can be partitioned as $1 + 2 + 2$ or $2 + 3$. However, $2 + 1 + 2$ is invalid because $2 \rightarrow 1$ is decreasing.
2. Continue from the previous question. We change the restriction to be *increasing*. How many ways can a positive integer n be partitioned? Write a program to find the answer. As an example, 5 can be partitioned as $2 + 3$ or $1 + 4$. However, $1 + 2 + 2$ is no longer invalid because 2 is used twice.
3. What is the output of this program? Write down the answer on paper *without* typing it into a computer.

```

/*
  file: recursiveprint.c
  purpose: show the values before and after recursive calls
*/
#include <stdio.h>
#include <stdlib.h>
void f(int n)
{
    int iter;
    if (n <= 0) { return; }
    if (n == 1)
    {
        printf("1 0\n");
    }
}

```

```

        return;
    }
    for (iter = 1; iter < n; iter++)
    {
        printf("%d %d\n", iter, n - iter);
        f(n - iter);
        printf("%d %d\n", iter, n - iter);
    }
}

int main(int argc, char * argv[])
{
    f(3);
    return EXIT_SUCCESS;
}

```

4. Write a program that takes one positive integer as a command argument and print the following pattern.

If the input is 1, the program prints

1

If the input is 2, the program prints

1 2 1

If the input is 3, the program prints

1 2 1 3 1 2 1

If the input is 4, the program prints

1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

Let X be the output when the input is $n - 1$. When the input is n , the output is

$X \ n \ X$

5. Consider the ball selection problem again with some changes. There are three colors: red, blue, and yellow. Assume there is an unlimited supply of all colors. Select the balls and arrange them (i.e. orders matter). Two red balls cannot be next to each other. How many options do you have for selecting n balls? For this question, you need to find a mathematical equation to determine the answer *without* listing the options.
6. Continue from the previous question. Write a program that can list the options.

7. Consider the following recursion function. The function has two arguments x and c . The first argument is used for the terminating condition. The second is a counter. Every time this function is called, the counter increments. This question asks you how many times the function is called when given different values for the first argument.

```
#include <stdio.h>
#include <stdlib.h>
int f(int x, int * c)
{
    (*c) ++;
    if (x <=2)
        {
            return x;
        }
    return 2 * f(x - 1, c) + 3 * f(x - 3, c);
}
int main(int argc, char * argv[])
{
    int a = 0;
    int b = 0;
    int v1, v2;
    v1 = f(3, & a);
    printf("v1 = %d, a = %d\n", v1, a);
    v2 = f(5, & b);
    printf("v2 = %d, b = %d\n", v2, b);
    return EXIT_SUCCESS;
}
```

What is the output of this program? Your answer should have 4 values: $v1$, a , $v2$, and b .

8. Consider integer partition. When partitioning 3,

$$\begin{aligned} 3 &= 1 + 1 + 1 \\ &= 1 + 2 \\ &= 2 + 1 \\ &= 3 \end{aligned}$$

There are 4 different ways (4 "=") to partition the value 3. As another example, we consider partitioning 4:

$$\begin{aligned} 4 &= 1 + 1 + 1 + 1 \\ &= 1 + 1 + 2 \\ &= 1 + 2 + 1 \\ &= 1 + 3 \\ &= 2 + 1 + 1 \\ &= 2 + 2 \\ &= 3 + 1 \\ &= 4 \end{aligned}$$

There are 8 ways to partition value 4. In general, there are 2^{n-1} ways to partition value n . You do not have to prove it. You can assume that this is true.

This question asks you to count how many numbers are used in these partitions. In the example for partitioning 3, 8 numbers are used:

- 1 is used 5 times
- 2 is used 2 times
- 3 is used once

In the example for partitioning 4, 20 numbers are used.

- 1 is used 12 times
- 2 is used 5 times
- 3 is used twice
- 4 is used once

How many numbers are used when you partition value 5?

Find a general rule for a positive integer n . When partitioning n , $f(n)$ numbers are used. Write down the recursive relation to compute $f(n)$.

Chapter 4

Pointer

4.1 Pointer and Address

What is a pointer? A pointer is a variable and its value is a memory address. Even though a program cannot decide which address is used to store a particular variable (such as `a`), the program can obtain the address given by the operating system by adding `&` in front of a variable.

```
int a = -7;
int * ptr; /* ptr is a pointer */
ptr = &a; /* ptr's value is a's address */
```

The first statement creates an integer variable `a`; its value is `-7`. The second statement creates an integer *pointer* `ptr`. The statement means `ptr`'s value is an address. An integer is stored at that address. The third statement assigns `a`'s address to `ptr`'s value. The call stack looks like the following (we ignore the frame in this case).

Symbol	Address	Value
<code>ptr</code>	1021	1020
<code>a</code>	1020	-7

Since `ptr` is a pointer, its value stores an address. In this example, `a`'s address is 1020 so `ptr`'s value is 1020. Please notice that `ptr` itself also occupies space in the call stack; `ptr`'s address is 1021. In Chapter 2, we always used 100 for the starting address. Here I use 1020 to remind you that we have no control of the address. The address is determined by the operating system.

If the program continues with

```
* ptr = 94;
```


The call stack becomes

Symbol	Address	Value
ptr	1021	1020
a	1020	94

The value of a has changed to 94. Why? The statement

```
* ptr = 94;
```

means *taking ptr's value as an address and modifying the value at that address*. Since ptr's value is 1020, the statement assigns 94 to the address. Now, a's value is 94. The next line puts * ptr at the right side of the assignment.

```
int c = * ptr;
```

Symbol	Address	Value
c	1022	94
ptr	1021	1020
a	1020	94

Another integer variable is created and its value is 94. Is this the same as

```
int c = a;
```

When * ptr is at the right side of an assignment, it means *taking ptr's value as an address and read the value at that address*. Since ptr's value is 1020, the statement reads the value at 1020 and this value is 94.

There are four different ways to use * in C programs. This table summarizes the usages.

Example	Meaning
<code>5 * 17</code>	Multiplication. <code>5 * 17</code> is 85
<code>int * ptr;</code>	create a pointer variable. <code>ptr</code> 's value is an address. An integer is stored at that address
<code>ptr = & val</code>	assign <code>val</code> 's address to <code>ptr</code> 's value
<code>* ptr =</code>	(left hand side of assignment) take <code>ptr</code> 's value as an address and modify the value at that address
<code>= * ptr</code>	(right hand side of assignment) take <code>ptr</code> 's value as an address and read the value at that address

Table 4.1: Four different usages of `*` in C programs. Sometimes you can see LHS for “left hand side” and RHS for “right hand side”.

The same symbol `*` has four different meanings. Make sure you fully understand this table.

We summarize using the following example.

```
int a;
int b;
int * ptr;      /* ptr is a pointer */
ptr = & a;      /* ptr's value is a's address */
* ptr = 10;     /* a's value is 10 */
b = * ptr;     /* b's value is also 10 */
* ptr = a * b; /* a's value becomes 100 */
```

In a C statement, the right hand side is evaluated before assigning the result to the left hand side. Hence, the last statement gets $10 \times 10 = 100$ first and then assigns 100 to `* ptr`. This makes `a`'s value 100.

4.2 Pointer and Argument

Calling a function usually has the intention of changing some variables. This example changes one variable by using the function's return value:

```
int a;
a = f(/* some arguments */);
```

The value of `a` is changed by the value returned from `f`. Is it possible to change two or more variables through a function call? Is it possible to write a swap function?

```
int a = 4;
int b = 7;
/* call swap function */
/* a is 7 and b is 4 now */
```

The previous chapter explains that a function can see only its own frame. As a result, the function has no access to variables in the calling function's frame. Fortunately, pointers allow a function to access the caller's frame. This is how to write the swap function.

```
/* file: swap.c
   purpose: swap two integers, using pointers
*/
#include <stdio.h>
#include <stdlib.h>
void swap(int * s, int * t)
{
    int u = * s;
    * s = * t;
    * t = u;
}

int main(int argc, char * argv[])
{
    int a = 4;
    int b = 7;
    printf("before calling swap a = %d, b = %d.\n", a, b);
    swap (& a, & b); /* remember & */
    /* return location 1 */
    printf("after calling swap a = %d, b = %d.\n", a, b);
    return EXIT_SUCCESS;
}
```

How does this work? Let's trace this program by using the call stack. We start with `main`'s frame.

Frame	Symbol	Address	Value
main	b	101	7
	a	100	4

We consider the frame when `swap` is called. As always, the return location is pushed to the call stack. The function returns nothing (`void`) so there is no value address. The function has two arguments `s` and `t` and both are pointers. *Their values are the addresses of `a` and `b` because `main` uses `& a` and `& b` when calling `swap`.*

Frame	Symbol	Address	Value
f2	u	105	?
	t	104	101
	s	103	100
	-	102	return location 1
main	b	101	7
	a	100	4

Inside `swap`, `u` is an integer. Using the rules in Table 4.1, `u`'s value is 4 because it is `* s`. Remember that `* s` reads the value at address 100 since it is at the right hand side.

Frame	Symbol	Address	Value
f2	u	105	4
	t	104	101
	s	103	100
	-	102	return location 1
main	b	101	7
	a	100	4

The next statement looks confusing but it can be understood using the rules in Table 4.1.

```
* s = * t;
```

First, `* t` takes `t`'s value as an address and read the value at that address. Since `t`'s value is 101, the value at address 101 is 7. Therefore, the right hand side is 7. The left hand side is `* s` so the value 7 is assigned to the address 100. This changes `a`'s value to 7.

Frame	Symbol	Address	Value
f2	u	105	4
	t	104	101
	s	103	100
	-	102	return location 1
main	b	101	7
	a	100	7

The last statement in `swap` is

```
* t = u;
```

This assign u's value, 4, to the address 101 and changes b's value to 4.

Frame	Symbol	Address	Value
f2	u	105	4
	t	104	101
	s	103	100
	-	102	return location 1
main	b	101	4
	a	100	7

The `swap` function finishes and its frame is popped. Back to the `main` function, the values of `a` and `b` have been swapped.

Frame	Symbol	Address	Value
main	b	101	4
	a	100	7

This `swap` program is helpful understanding pointers and the call stack. Please study the program carefully.

4.3 Array and Argument

This section describes how to pass an array as an argument and to find certain properties of the array. Specifically, we are going to write functions that

- find the maximum value in the array's elements
- find the minimum value in the array's elements
- add all array's elements
- compute the average of the array's elements
- determine whether the array's elements are in the ascending order

4.4 Practice Problems

1. For the following program

```
/* file: pointer1.c
   purpose: practice problems for pointer
*/
#include <stdio.h>
#include <stdlib.h>
void f(int * a, int * b)
{
    int * c;
    c = a;
    a = b;
    /* draw the call stack when the program reaches here */
    b = c;
}
int main(int argc, char * argv[])
{
    int s = 5;
    int t = 19;
    printf("s = %d, t = %d\n", s, t);
    f(& s, & t);
    printf("s = %d, t = %d\n", s, t);
    return EXIT_SUCCESS;
}
```

draw the call stack when the program finishes

```
a = b;
```

What is the output of this program?

2. For the following program

```
/* file: pointer2.c
   purpose: practice problems for pointer
*/
#include <stdio.h>
#include <stdlib.h>
void f(int * a, int * b)
{
    int c;
    c = (* a) * (* b);
    /* draw the call stack when the program reaches here */
    * a = c;
}
```

```

    /* draw the call stack when the program reaches here */
}
int main(int argc, char * argv[])
{
    int s = 5;
    int t = 19;
    printf("s = %d, t = %d\n", s, t);
    f(& s, & t);
    printf("s = %d, t = %d\n", s, t);
    return EXIT_SUCCESS;
}

```

draw the call stack when the program finishes

```
c = (* a) * (* b);
```

and when the program finishes

```
* a = c;
```

What is the output of this program?

3. Draw the call stack when the program reaches each Label.

```

/*
   file: pointer3.c
*/

#include <stdio.h>
#include <stdlib.h>
int fun1(int a, int * b);
int fun2(int a, int * b);

int main(int argc, char * argv[])
{
    int a = 2;
    int b = 6;
    int * p1 = & a;
    /* Label 1 */
    printf("at Label 1, a = %d, b = %d, * p1 = %d\n", a, b, * p1);
    *p1 = fun2(b, p1);
    /* Label 7 */
    printf("at Label 7, a = %d, b = %d, * p1 = %d\n", a, b, * p1);
    return EXIT_SUCCESS;
}

```

```

int fun1(int a, int * b)
{
    int s;
    /* Label 4 */
    printf("at Label 4, a = %d, *b = %d\n", a, * b);
    * b = a;
    s = (* b) * 2;
    /* Label 5 */
    printf("at Label 5, s = %d, a = %d, *b = %d\n", s, a, * b);
    return s;
}

int fun2(int a, int * b)
{
    int s, t, u;
    s = a;
    /* Label 2 */
    printf("at Label 2, s = %d, a = %d, * b = %d\n", s, a, *b);
    t = (* b) + a;
    /* Label 3 */
    printf("at Label 3, s = %d, t = %d, a = %d, * b = %d\n", s, t, a, *b);
    u = fun1(t, b);
    * b = u;
    /* Label 6 */
    printf("at Label 6, s = %d, t = %d, u = %d, a = %d, * b = %d\n",
           s, t, u, a, *b);
    return (s + u);
}

```


Chapter 5

Memory Management (Heap)

Chapter 2 describes one type of memory: stack. Stack memory follows a simple rule: first in, last out. The call stack is managed by compilers and operating systems.

Sometimes, programmers want to have more control of memory space. They want to be able to *allocate* memory when necessary and *release* memory when the allocated memory is no longer needed. For this purpose, computers have another type of memory: *heap*. Before talking about how to use heap memory, let's review how to create a fixed-size array.

5.1 Create a Fixed-Size Array

C allows two ways to create an array. The first is a fixed-size array and its size is already known when the program is written. The following example creates an array of six integers.

```
int arr[6];
```

C does not initialize array elements. After creating this array, the elements' values are unknown. They may be any value. A common mistake is to assume that the elements are zero.

The following code assigns the first three elements to 11, -29, and 74.

```
arr[0] = 11;  
arr[1] = -29;  
arr[2] = 74;
```

The values of the other three elements are still unknown.

In C programs, indexes start from zero. If an array has six elements, the indexes can be 0 to 5 (inclusively). A common mistake is to use 1 to 6 for the indexes. C does not check whether an index is invalid. If your program uses an invalid index, the program's behavior is undefined. Sometimes, the program works; sometimes, the program does not. You need to be careful not to make this mistake.

5.2 Create an Array using `malloc`

Creating a fixed-size array must specify the array's size when the program is written. This is problematic because the size may be unknown. Sometimes, we need to create an array whose size is known after the program starts. We will use `malloc` to create an array. The following code creates an array of integers.

```
int * arr2;
int length;
scanf("%d", & length);
arr2 = malloc(length * sizeof(int));
```

Notice `*` in front of `arr2`. The value of `length` is entered by a user. Thus, the value is known only after the program starts running. If we want to create an array of integers, we need an integer pointer. We need to use `sizeof(int)` because the size of an integer may be different on different machines. Also, make sure the types match: using `int` in creating the pointer and in allocating memory. If you do the following

```
int * arr2;
arr2 = malloc(length * sizeof(char));
/* WRONG */
/* types do not match, one is int, the other is char */
```

The program is wrong because the types do not match. Of course, the following is also problematic

```
int * arr2;
arr2 = malloc(length * sizeof(double));
/* WRONG */
/* types do not match, one is int, the other is double */
```

To create an array of `length` characters, we need to specify `char` in both places:

```
char * arr2;
arr2 = malloc(length * sizeof(char));
/* correct */
/* an array of characters */
```

Some books suggest casting before `malloc` like the following

```
int * arr2;
arr2 = (int *) malloc(length * sizeof(int));
```

Adding `(int *)` in front of `malloc` is an old way of writing C programs. You may see that in some books but you don't need to do that any more.

Your programs should check whether `malloc` succeeds. If it fails, `arr2` is `NULL`. Your programs should handle this problem before doing anything else. Why does `malloc` fail? When the system cannot provide as much space as requested. This is more likely in embedded systems where memory is limited. C uses `NULL` to indicate an invalid value for a memory address.

After creating this array, we can assign values to the elements in the same way

```
arr2[0] = 11;
arr2[1] = -29;
arr2[2] = 74;
```

When we no longer need this array, we have to release the memory occupied by this array.

```
free(arr2);
```

You should develop a habit typing `free` right after typing `malloc` so that you do not forget to call `free`. You can insert code between them in this way:

```
malloc
/* insert code here */
free
```

When you use `malloc`, you need to specify the number of elements in the array. When you use `free`, you should not (and cannot) specify the number of elements. If your program calls `malloc` without calling `free`, the program has *memory leak*. Memory leak is a serious problem because a program has a limit about the amount of memory that can be allocated by calling `malloc`. The limit depends on the hardware and also the operating systems. You can use `valgrind` to detect memory leak. Section 5.8 explains how to use `valgrind`.

5.3 Stack and Heap

When a program declares a pointer, the call stack has this pointer. For example,

```
int * arr2;
```

Symbol	Address	Value
arr2	200	?

For simplicity of explanation, we are going to assume `length` is 6 here. This statement

```
arr2 = malloc(6 * sizeof(int));
```

assigns a heap memory address to `arr2`. The heap memory is pretty far away from the stack memory, i.e. their addresses are quite different. In this example, I use 10000 for the heap address.

Symbol	Address	Value
arr2	200	10000

Address	Value
10005	?
10004	?
10003	?
10002	?
10001	?
10000	?

(a)

(b)

Figure 5.1: (a) Call stack. After calling `malloc`, `arr2`'s value is an address in the heap memory. In this example, we use 10000 for the value. Six memory locations are allocated and they are adjacent. (b) Heap memory. The values have not been assigned so "?" is used for their values. We need to use a symbol in the stack (`arr2`) to access heap memory so the heap memory has no column for symbols.

The following assignment changes the first element of the array.

```
arr2[0] = 11;
```

The heap memory at 10000 changes

Symbol	Address	Value
arr2	200	10000

Address	Value
10005	?
10004	?
10003	?
10002	?
10001	?
10000	11

(a)

(b)

```
arr2[2] = 74;
```

changes the heap memory at 1002:

Symbol	Address	Value
arr2	200	10000

Address	Value
10005	?
10004	?
10003	?
10002	74
10001	?
10000	11

(a)

(b)

How does this work?

The program takes `arr2`'s value as an address; in this example the value is 10000. If the index is 0, the value stored at that address is modified. If the index is 1, the value stored at the next address is modified.

The following example creates an array whose length is determined by `argc`. The program converts the arguments into integers, since each element of `argv` is a string. Then, the program adds the integers' values and prints the sum.

```
/* file: egmalloc.c
   purpose: create an array whose size is specified at run time.
   The array's elements are the command line arguments.
   The program adds the elements and prints the sum.
*/
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
    int * arr2;
    int iter;
    int sum = 0;
    if (argc < 2)
    {
        printf("Need to provide some integers.\n");
        return EXIT_FAILURE;
    }
    arr2 = malloc(argc * sizeof(int));
    if (arr2 == NULL)
    {
        printf("malloc fails.\n");
        return EXIT_FAILURE;
    }
    /* iter starts at 1 because argv[0] is the program's name */
    for (iter = 1; iter < argc; iter++)
    {
        arr2[iter] = (int) strtol(argv[iter], NULL, 10);
    }
    printf("The sum of ");
    for (iter = 1; iter < argc; iter++)
    {
        printf("%d ", arr2[iter]);
        sum += arr2[iter];
    }
    printf("is %d.\n", sum);

    free (arr2);
}
```

```
    return EXIT_SUCCESS;
}
```

The following shows two examples running the program. If an argument is not an integer (“hello” and “C” in the second example), the value is zero.

```
> ./egmalloc 5 8 -11 4 3 27
The sum of 5 8 -11 4 3 27 is 36.
> ./egmalloc 7 9 hello 1 6 C 2 4 8
The sum of 7 9 0 1 6 0 2 4 8 is 37.
```

We expand Table 4.1 by adding the rules for an array created using `malloc`.

Example	Meaning
<code>5 * 17</code>	Multiplication. <code>5 * 17</code> is 85
<code>int * ptr;</code>	create a pointer variable. <code>ptr</code> 's value is an address. An integer is stored at that address
<code>ptr = & val</code>	assign <code>val</code> 's address to <code>ptr</code> 's value
<code>* ptr =</code>	(left hand side of assignment) take <code>ptr</code> 's value as an address and modify the value at that address
<code>= * ptr</code>	(right hand side of assignment) take <code>ptr</code> 's value as an address and read the value at that address
<code>int * arr;</code>	create a pointer variable. <code>arr</code> 's value is an address.
<code>arr = malloc(sizeof(int) * n);</code>	allocate an array on the heap memory. This array has <code>n</code> elements. Each element is an integer, as specified by <code>sizeof(int)</code> . Please notice that the asterisk inside the parenthesis means multiplication. The address of the first element (index is 0) is stored as <code>arr</code> 's value.
<code>arr[k] =</code>	(left hand side of assignment) take <code>arr</code> 's value as an address, add <code>k</code> (called <i>offset</i>) to get a new address, modify the value at that address. Here, <code>k</code> must be between 0 and <code>n - 1</code> . Otherwise, the program's behavior is undefined.
<code>= arr[k]</code>	(right hand side of assignment) take <code>arr</code> 's value as an address, add <code>k</code> (called <i>offset</i>) to get a new address, read the value at that address. Here, <code>k</code> must be between 0 and <code>n - 1</code> . Otherwise, the program's behavior is undefined.

Table 5.1: The first part of this table repeats Table 4.1. The second part summarizes how to create an array on heap memory and to access the array's elements.

5.4 Array and Address

In your first C class, you may have heard that arrays and pointers are related. In fact, the name of an array stores the address of the first element. The following program shows an example of the addresses of array elements

```

/* file: address.c
   purpose: print the addresses of array elements
*/
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
    int arr[6];
    printf("arr      = %p, &arr[0] = %p.\n", arr, &arr[0]);
    printf("&arr[0] = %p, &arr[1] = %p.\n", &arr[0], &arr[1]);
    printf("sizeof(int) = %ld.\n", sizeof(int));
    return EXIT_SUCCESS;
}

```

Running this program produces an output like the following

```
> ./address
arr      = 0x7fffe3a7fea0, &arr[0] = 0x7fffe3a7fea0.
&arr[0] = 0x7fffe3a7fea0, &arr[1] = 0x7fffe3a7fea4.
sizeof(int) = 4.
```

We said “like the following” because the outputs are not the same every time. Running the same program two more times and the outputs are

```
> ./address
arr      = 0x7fff673a2880, &arr[0] = 0x7fff673a2880.
&arr[0] = 0x7fff673a2880, &arr[1] = 0x7fff673a2884.
sizeof(int) = 4.
> ./address
arr      = 0x7ffff33f2580, &arr[0] = 0x7ffff33f2580.
&arr[0] = 0x7ffff33f2580, &arr[1] = 0x7ffff33f2584.
sizeof(int) = 4.
```

As you can see, the actual value of `&arr[0]` or `&arr[1]` changes in different executions. However, the following observations are true:

- The value of `arr` is always the same as `&arr[0]` in each execution.
- The difference of `&arr[1]` and `&arr[0]` is always four.
- The value of `sizeof(int)` is always four because my computer uses 4 bytes for an integer. If you use a different computer, the value can be different.

This program shows that `arr` is actually a pointer, storing the address of the first element, i.e. `&arr[0]`. The difference of `&arr[1]` and `&arr[0]` is four because each element is an integer. Each integer takes four bytes, as indicated by `sizeof(int)`. As I mentioned earlier, in most cases, we do not need to worry about the actual values of the addresses. We can simply treat the addresses as continuous integers (10000, 10001, ...).

5.5 Function Return a Heap Address

A function may return the address of a heap memory. For example

```

int * f1(int n)
{
    int * ptr;
    ptr = malloc (n * sizeof(int));
    return ptr;
}
void f2(void)
{
    int * arr;
    arr = f1(6);
    /* return location */
    arr[4] = 7;
    free (arr);
}

```

Let's consider the call stack just before `f1` returns `ptr`;

Symbol	Address	Value
<code>ptr</code>	103	10000
-	102	value address 100
-	101	return location
<code>arr</code>	100	?

(a)

Address	Value
10005	?
10004	?
10003	?
10002	?
10001	?
10000	?

(b)

After `f1` returns, this is what in the call stack and the heap memory.

Symbol	Address	Value
<code>arr</code>	100	10000

(a)

Address	Value
10005	?
10004	?
10003	?
10002	?
10001	?
10000	?

(b)

The allocated heap memory is still available because the program has not called `free` yet. This is a fundamental difference between stack and heap memory: when a function finishes, its frame is popped. For heap memory, it is still available until the program calls `free`.

The statement

```
arr[4] = 872;
```

changes an element in the array:

Symbol	Address	Value
arr	100	10000

(a)

Address	Value
10005	?
10004	872
10003	?
10002	?
10001	?
10000	?

(b)

Before `f2` finishes, it should call `free`. Otherwise, the program leaks memory. The purpose of this example is to show that memory allocated by `malloc` can be passed between functions. Please be aware that this example does not follow the principle mentioned on page 79. In this example, `malloc` and `free` are called in two different functions. This is a bad programming style. It is easy to forget calling `free` in `f2` because `f2` does not call `malloc`.

5.6 Two-Dimensional Array in C

In C programs, creating a fixed-size two-dimensional array can be achieved by adding another dimension:

```
int arr2d[5][3]; /* an array with 5 rows and 3 columns */
arr2d[0][2] = 4; /* assign 4 to the third column of the first row */
arr2d[3][1] = 6; /* assign 6 to the second column of the fourth row */
```

In this example, the first dimension has five rows and the indexes are between zero and four (inclusively). The second dimension has three columns; the indexes are between zero and two (inclusively).

It is more complicated creating a two-dimensional array whose size is known only at run time. We first create a one-dimensional array of integer pointers (`int *`) and then each pointer is used to create an integer array. Figure 5.2 illustrates this concept.

```
/* file: twodarray.c
   purpose: show how to create a two-dimensional array
   The size of the array is 8 rows x 3 columns
*/
```

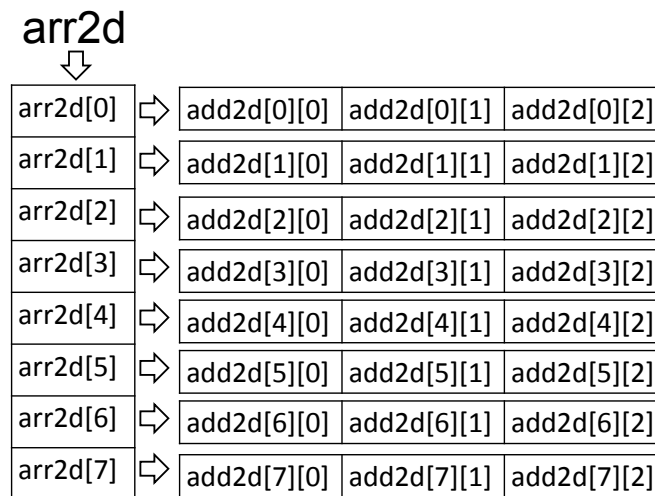


Figure 5.2: A two-dimensional array is created in two steps. The first step creates an array of integer pointers. In the second step, each pointer stores the address of the first element in an integer array.

```
#include <stdio.h>
#include <stdlib.h>
#define NUMROW 8
#define NUMCOLUMN 3
int main(int argc, char * argv[])
{
    int * * arr2d;
    int row;
    /* step 1: create an array of integer pointers */
    arr2d = malloc(NUMROW * sizeof (int *));
    for (row = 0; row < NUMROW; row ++)
    {
        /* step 2: for each row (i.e. integer pointer), create an
           integer array */
        arr2d[row] = malloc(NUMCOLUMN * sizeof (int));
    }
    arr2d[4][1] = 6;
    arr2d[6][0] = 19;
    /* the first index can be 0 to 7 (inclusive) */
    /* the second index can be 0 to 2 (inclusive) */

    /* memory must be released in the reverse order */
    for (row = 0; row < NUMROW; row ++)
    {
        /* release the memory of each row first */
        free (arr2d[row]);
    }
}
```

```

    }
    /* release the array of integer pointer */
    free (arr2);
    return EXIT_SUCCESS;
}

```

After creating the two-dimensional array, it can be used in the same way as a fixed-size array. Before the program ends, the allocated memory must be released. Memory must be released in the *reverse order* as it is allocated.

Let's review the *types* used in creating arrays. An array's name stores the address of the first element. Therefore, the type of an array is an integer pointer. If we want to create an array using `malloc`, we need to use `int *`.

```

int arr[6]; /* an array of fixed size, 6 elements */
int * arr2;
arr2 = malloc(9 * sizeof(int)); /* 9 elements */
arr2[4] = 19; /* arr2[4] is an integer */
free(arr2);

```

A two-dimensional array is composed of a one dimensional array of integer pointers. Therefore, the type is `int **`. The same reason makes the argument of `malloc` use `sizeof(int *)`, not `sizeof(int)`.

```

int ** arr2d;
arr2d = malloc(10 * sizeof (int *));
arr2d[4] = malloc(3 * sizeof(int)); /* arr2d[4] is an integer pointer */
arr2d[4][2] = 8; /* arr2d[4][2] is an integer */

```

Each element of `arr2d`, for example `arr2d[2]`, is an integer pointer. An index removes one asterisk; therefore, an element is an integer pointer. This pointer can be used to create an array of the second dimension.

5.7 Pointer and Argument

We can pass the address of the heap memory (i.e. a pointer) as a function argument. The following example passes an array allocated on the heap memory to a function.

```

/* file: heaparg.c
   purpose: pass the address of heap memory as a function argument
*/

```

```

#include <stdio.h>
#include <stdlib.h>
int sum(int * array, int length)
{
    int iter;
    int answer = 0;
    for (iter = 0; iter < length; iter ++)
        {
            answer += array[iter];
        }
    return answer;
}
int main(int argc, char * argv[])
{
    int * arr;
    int iter;
    int length = 12;
    int total;
    arr = malloc(length * sizeof(int));
    if (arr == NULL)
        {
            printf("malloc fails.\n");
            return EXIT_FAILURE;
        }
    for (iter = 0; iter < length; iter ++)
        {
            arr[iter] = iter;
        }
    total = sum(arr, length);
    /* return location 1 */
    printf("Total is %d.\n", total);
    free (arr);
    return EXIT_SUCCESS;
}

```

In this example, `arr` is passed to function `sum` as an argument. Remember that C programs *copy values* to function arguments. The call stack and the heap memory looks like the following inside the `sum` function, just before the `for` block starts.

Frame	Symbol	Address	Value
sum	answer	211	0
	iter	210	-
	length	209	12
	array	208	10000
	-	207	value address 205
	-	206	return location 1
main	total	205	?
	length	204	12
	iter	203	13
	arr	202	10000
	argv	201	-
	argc	200	-

call stack

Address	Value
10011	11
10010	10
10009	9
10008	8
10007	7
10006	6
10005	5
10004	4
10003	3
10002	2
10001	1
10000	0

heap memory

The argument `array` is a pointer so its value is an address. When `main` calls `sum`, the value of `arr` (i.e. 10000) is copied to the value of `array`. Inside `sum`, `array[0]` refers to the value stored at 10000 and it is 0. Similarly, `array[7]` refers to the value stored at 10007 and it is 7.

Passing heap address makes your programs more flexible. A function can modify the data and the changes are visible to the caller. In the following example, the function `multi2` doubles the array elements.

```

/* file: multi2.c
   purpose: double each array element
*/
#include <stdio.h>
#include <stdlib.h>
void multi2(int * array, int length)
{
    int iter;
    for (iter = 0; iter < length; iter ++)
    {
        array[iter] *= 2;
    }
}
int main(int argc, char * argv[])
{
    int * arr;
    int iter;
    int length = 12;
    arr = malloc(length * sizeof(int));
    if (arr == NULL)
    {
        printf("malloc fails.\n");
    }
}

```



```

        return EXIT_FAILURE;
    }
    for (iter = 0; iter < length; iter ++)
    {
        arr[iter] = iter;
    }

    printf("Original array: ");
    for (iter = 0; iter < length; iter ++)
    {
        printf("%2d ", arr[iter]);
    }
    printf("\n");

    multi2(arr, length);

    printf("New array:      ");
    for (iter = 0; iter < length; iter ++)
    {
        printf("%2d ", arr[iter]);
    }
    printf("\n");

    free (arr);
    return EXIT_SUCCESS;
}

```

The output of this program is shown below

```

Original array:  0  1  2  3  4  5  6  7  8  9 10 11
New array:      0  2  4  6  8 10 12 14 16 18 20 22

```

Remember to call `free` before the program ends. Otherwise, the program has memory leak. Also, to make your program easier to understand and easier to debug, *your program should call `malloc` and `free` in the same function*. If your program calls `malloc` and `free` in different functions, you will soon lose track whether

1. memory allocated by calling `malloc` is released by calling `free` later or
2. memory released by calling `free` has been allocated by calling `malloc` earlier.

5.8 Use `valgrind` to Detect Memory Leak

In Linux, you can use `valgrind` to detect memory leak. Suppose we remove

```
free(arr);
```

near the end of `main`. In a Linux terminal, type the following two lines

```
> gcc -g -Wall -Wshadow multi2.c -o multi2
> valgrind --leak-check=yes ./multi2
```

The first line uses `gcc` to create an executable file called `multi2`. The second line uses `valgrind` to detect memory leak. The output includes a lot of things. Pay attention to the following message

```
LEAK SUMMARY:
    definitely lost: 48 bytes in 1 blocks
```

Why does the program leak 48 bytes? The program allocates space for 12 integers by calling `malloc`. On this machine, each integer occupies 4 bytes. Therefore, the program leaks 48 bytes. If we put back the statement

```
free(arr);
```

`valgrind` reports

```
All heap blocks were freed -- no leaks are possible
```


Chapter 6

String

What is a string? Anything between two double quotations is a string. For example,

- “Hello”
- “C Language”
- “write 2 programs.”
- “symbols \$%# can be parts of a string”

As you can see, a string can include alphabets, digits, space, and symbols. C does not have a specific type for strings. Instead, C uses arrays of characters for strings. Each string must end with a special character `'\0'`. It is not shown above but it is important and often forgotten. Thus, if we want to store the string “Hello”, we need an array that have 6 elements as shown below

```
char str[6];           /* create an array with 6 characters */
strcpy(str, "Hello"); /* copy "Hello" to the array */
```

The function `strcpy` copies a string. This function takes two arguments: the first is the destination and the second is the source. If you count, there are five characters in “Hello” but we need space for the special `'\0'` character. The function does *not* check whether the destination has enough space. In fact, the manual page says, “If the destination string of a `strcpy()` is not large enough, then anything might happen. Overflowing fixed-length string buffers is a favorite cracker technique for taking complete control of the machine. Any time a program reads or copies data into a buffer, the program first needs to check that there’s enough space.”

C provides a function, `strlen`, to calculate the length of a string. This function does not count the special `'\0'` character. Thus, `strlen("Hello")` is 5. If we want to copy a string from `src` to `dest`, the following code is commonly used.

```
char * dest = malloc(sizeof(char) * (strlen(src) + 1));
strcpy(dest, src);
```

The first statement allocates space for the destination. We need to add one to the value from `strlen` to accommodate the special `'\0'` character. What happens if a string does not end with the `'\0'` character? The program's behavior is undefined because many functions for strings go through the elements one by one until reaching this special character.

The C function `strlen` does not include `'\0'`. You must remember to add 1 for this additional character.

Every string must end with the special `'\0'` character. This must be an element of the character array where the string is stored. What happens if the special `'\0'` character is not the last element of a character array? This is not a problem. It simply means the string ends without using all elements of the array. For example, we can create a character array with 10 elements and fill the first five elements as "Hello".

```
char str[10];
str[0] = 'H';
str[1] = 'e';
str[2] = 'l';
str[3] = 'l';
str[4] = 'o';
str[5] = '\0';
```

When we call `strlen(str)`, the function returns 5 because there are 5 elements before `'\0'`. It does not matter what is stored at `str[7]` or `str[8]`.

6.1 Compare Strings using `strcmp`

We can use `strcmp` to compare two strings. This function takes two arguments:

```
strcmp(str1, str2);
```

The function returns a negative integer, zero, or a positive integer dependings on whether `str1` is smaller, equal, or greater than `str2`. How do we define the order of two strings? It is

similar to the orders of words in a dictionary, called lexicographical order. For example, “about” is smaller than “forever” because a is before f in the alphabetic order. “Education” is after “Change”.

How do we compare upper cases and lower cases? How do we define the order if one or both strings contain digits or symbols? The order is determined by the ASCII (American Standard Code for Information Interchange) values. ASCII assigns an integer value to each character. The value for ‘A’ is 65 and the value of ‘a’ is 97. ASCII also assigns a value to each symbol or digit. The value is 35 for ‘#’ and 55 for digit ‘7’. The last statement may sound strange. The value of digit ‘7’ is 55. What does this mean?

C has different types, including `char` for characters and `int` for integers. The two types are treated differently. This can be shown using the following example:

```
/*
 * charint.c
 * how how C treats integer and character differently
 */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
    int v = 55;
    printf("%d\n", v);
    printf("%c\n", v);
    return EXIT_SUCCESS;
}
```

The output of this program is

```
55
7
```

Why do the two lines print different values, even though both use `v`? The first `printf` treats `v` as an integer by using `%d`; hence, the printed value is 55. The second `printf` treats `v` as a character by using `%c`. Since 55 is the ASCII value of character ‘7’, 7 is printed on screen.

6.2 Use `strstr` to Detect Substring

If string `str1` is part of another string `str2`, we say `str1` is a *substring* of `str2`. For example, “str” is a substring of “structure”. “W” is a substring of “Welcome”; “sea” is not a substring of “sightseeing”. If we want to determine whether one string is part of another string, we can use the `strstr` function. This function takes two arguments. If `shorterstr` is a substring of

longerstr, strstr(longerstr, shorterstr) returns a value that is not NULL. Please notice the order of the two arguments.

The following program checks whether argv[2] is a substring of argv[1].

```
/*
 * substr.c
 * find whether one string is another string's substring
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char * argv[])
{
    if (argc < 3)
    {
        printf("Please enter two strings.\n");
        return EXIT_FAILURE;
    }
    printf("argv[1] = %s\n", argv[1]);
    printf("argv[2] = %s\n", argv[2]);
    printf("argv[2] is ");
    if (strstr(argv[1], argv[2]) == NULL)
    {
        printf("not ");
    }
    printf("a substring of argv[1].\n");
    return EXIT_SUCCESS;
}
```

```
> gcc substr.c -o substr
```

```
> ./substr hello lo
argv[1] = hello
argv[2] = lo
argv[2] is a substring of argv[1].
```

```
> ./substr hello low
argv[1] = hello
argv[2] = low
argv[2] is not a substring of argv[1].
```

We can use quotation marks to enclose arguments. For example,

```

> ./substr "have a nice day" "ce day"
argv[1] = have a nice day
argv[2] = ce day
argv[2] is a substring of argv[1].

> ./substr "have a nice day" "hot day"
argv[1] = have a nice day
argv[2] = hot day
argv[2] is not a substring of argv[1].

```

6.3 Understand argv

We have seen

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

many times. What is `argv` really? It is a two-dimensional array of characters. Figure 5.2 shows how a two-dimensional array is organized in a C program. The program `twodarray.c` on page 87 has the same number of elements for each row by using this statement to allocate memory:

```
arr2d[row] = malloc(NUMCOLUMN * sizeof (int));
```

The value `NUMCOLUMN` specifies the length of each row. However, the rows are allowed to have different lengths.

Where is the memory holding `argc` and `argv`? Since `argv` is an array of arrays, where is the memory holding the elements? The `main` function is a special function; the arguments are provided by operating systems (more precisely, by the shell program in the terminal). Since `main` is also a function, the arguments are stored on the call stack, as any other function.

Frame	Symbol	Address	Value
main	argv	101	?
	argc	100	?

Let's execute the `substr` program with these arguments:

```
> ./substr nice ice
```


There are three arguments so `argc` is 3. The value of `argv` is the address of the first element, i.e. `& argv[0]`. Where is `argv[0]` stored? Depending on the processor and the operating system, the elements of `argv` may be stored on the call stack or the heap. Fortunately, we do not need to know where they are stored. In this example, we assume the elements are stored on the call stack. The following shows the call stack. The value of `argv` is the address of `argv[0]`. For now, we use - for the values of `argv[0]`, `argv[1]`, and `argv[2]` to simplify the call stack.

Frame	Symbol	Address	Value
main	<code>argv[2]</code>	104	-
	<code>argv[1]</code>	103	-
	<code>argv[0]</code>	102	-
	<code>argv</code>	101	102
	<code>argc</code>	100	3

Since `argv` is a two-dimensional array, `argv[0]`, `argv[1]`, and `argv[2]` each stores the starting address of an array. The first argument is `"/substr"` and it is stored on the call stack as shown below.

Frame	Symbol	Address	Value
main	<code>argv[0][8]</code>	113	<code>\0</code>
	<code>argv[0][7]</code>	112	<code>r</code>
	<code>argv[0][6]</code>	111	<code>t</code>
	<code>argv[0][5]</code>	110	<code>s</code>
	<code>argv[0][4]</code>	109	<code>b</code>
	<code>argv[0][3]</code>	108	<code>u</code>
	<code>argv[0][2]</code>	107	<code>s</code>
	<code>argv[0][1]</code>	106	<code>/</code>
	<code>argv[0][0]</code>	105	<code>.</code>
	<code>argv[2]</code>	104	-
	<code>argv[1]</code>	103	-
	<code>argv[0]</code>	102	105
	<code>argv</code>	101	102
	<code>argc</code>	100	3

Each string must end with the special `'\0'` character. Thus, at address 113 stores this character to indicate the end of `argv[0]`.

Next, we show the elements of `argv[1]` on the call stack.

Frame	Symbol	Address	Value
main	argv[1][4]	118	\0
	argv[1][3]	117	e
	argv[1][2]	116	c
	argv[1][1]	115	i
	argv[1][0]	114	n
	argv[0][8]	113	\0
	argv[0][7]	112	r
	argv[0][6]	111	t
	argv[0][5]	110	s
	argv[0][4]	109	b
	argv[0][3]	108	u
	argv[0][2]	107	s
	argv[0][1]	106	/
	argv[0][0]	105	.
	argv[2]	104	-
	argv[1]	103	114
	argv[0]	102	105
	argv	101	102
	argc	100	3

Last, we show the elements of `argv[2]` on the call stack.

Frame	Symbol	Address	Value
main	argv[2][3]	122	\0
	argv[2][2]	121	e
	argv[2][1]	120	c
	argv[2][0]	119	i
	argv[1][4]	118	\0
	argv[1][3]	117	e
	argv[1][2]	116	c
	argv[1][1]	115	i
	argv[1][0]	114	n
	argv[0][8]	113	\0
	argv[0][7]	112	r
	argv[0][6]	111	t
	argv[0][5]	110	s
	argv[0][4]	109	b
	argv[0][3]	108	u
	argv[0][2]	107	s
	argv[0][1]	106	/
	argv[0][0]	105	.
	argv[2]	104	119
	argv[1]	103	114
argv[0]	102	105	
argv	101	102	
argc	100	3	

If you print the length of `argv[1]`, it is 4 because the special character is not counted. However, `argv[1]` must have space for that character to indicate the end of the string.

6.4 Count Substrings

Sometimes, we want to count the occurrences of a substring. For example, “ice” is a substring of “nice” and it occurs only once. In this string, “This is his history book”, the substring “is” occurs 4 times. We can use `strstr` to count the occurrence of a substring but we need to understand `strstr` more deeply. The previous section says that `strstr` returns `NULL` if the second argument is not a substring of the first argument. What does `strstr` return if the second argument is a substring of the first argument?

If the second argument is a substring of the first argument, `strstr` returns the address of the first occurrence of the substring. Let’s use the call stack shown earlier. Calling

```
strstr(argv[1], argv[2]);
```

returns 115 because it is the address of the first occurrence of the substring “ice” in “nice”. We can store this address in a pointer:

```
char * ptr;
ptr = strstr(argv[1], argv[2]);
```

The call stack is shown below

Frame	Symbol	Address	Value
main	ptr	123	115
	argv[2][3]	122	\0
	argv[2][2]	121	e
	argv[2][1]	120	c
	argv[2][0]	119	i
	argv[1][4]	118	\0
	argv[1][3]	117	e
	argv[1][2]	116	c
	argv[1][1]	115	i
	argv[1][0]	114	n
	argv[0][8]	113	\0
	argv[0][7]	112	r
	argv[0][6]	111	t
	argv[0][5]	110	s
	argv[0][4]	109	b
	argv[0][3]	108	u
	argv[0][2]	107	s
	argv[0][1]	106	/
	argv[0][0]	105	.
	argv[2]	104	119
	argv[1]	103	114
argv[0]	102	105	
argv	101	102	
argc	100	3	

If we increment `ptr`, what happens? Its value becomes 116, the address of the next element.

Now, let’s consider a longer argument. Suppose `argv[1]` is “This is his history book.” and `argv[2]` is “is”. The substring “is” occurs four times in `argv[1]`. We are going to focus on these two arguments and omit the other parts of the call stack. We also change the address of `argv[1][0]` to 200 as a reminder that we have no control of the addresses.

Frame	Symbol	Address	Value
main	ptr	229	-
	argv[2][2]	228	\0
	argv[2][1]	227	s
	argv[2][0]	226	i
	argv[1][25]	225	\0
	argv[1][24]	224	.
	argv[1][23]	223	k
	argv[1][22]	222	o
	argv[1][21]	221	o
	argv[1][20]	220	b
	argv[1][19]	219	
	argv[1][18]	218	y
	argv[1][17]	217	r
	argv[1][16]	216	o
	argv[1][15]	215	t
	argv[1][14]	214	s
	argv[1][13]	213	i
	argv[1][12]	212	h
	argv[1][11]	211	
	argv[1][10]	210	s
	argv[1][9]	209	i
	argv[1][8]	208	h
	argv[1][7]	207	
	argv[1][6]	206	s
	argv[1][5]	205	i
	argv[1][4]	204	
	argv[1][3]	203	s
	argv[1][2]	202	i
	argv[1][1]	201	h
	argv[1][0]	200	T

We can use `strstr` to find the first occurrence of "is":

```
char * ptr;
ptr = strstr(argv[1], argv[2]); /* ptr's value is 202 */
```

How do we find the second occurrence of "is"? Can we call `strstr` again using `argv[1]` and `argv[2]`? If we do that, `ptr` is 202 again. We cannot proceed to find the second occurrence. Instead, we should move *after* the first occurrence if we want to find the second occurrence. This is how we can find the second occurrence:

```
if (ptr != NULL)
```

```

{
    ptr++; /* move after the first occurrence */
    ptr = strstr(ptr, argv[2]);
}

```

We check whether `ptr` is `NULL`. If it is `NULL`, there is no first occurrence. Otherwise, we increment `ptr` so that `strstr` will start from address 203, not 200. To find the third occurrence, we execute the same code again. Since we do not know the number of occurrences, we should replace `if` by `while`. The complete program is shown below:

```

/*
 * countsubstr.c
 * count the occurrence of a substring
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char * argv[])
{
    int count = 0;
    char * ptr;
    if (argc < 3)
        {
            printf("Please enter two strings.\n");
            return EXIT_FAILURE;
        }
    printf("argv[1] = %s, strlen = %d\n", argv[1], (int) strlen(argv[1]));
    printf("argv[2] = %s, strlen = %d\n", argv[2], (int) strlen(argv[2]));
    ptr = argv[1];
    do
        {
            ptr = strstr(ptr, argv[2]);
            if (ptr != NULL)
                {
                    printf("%s\n", ptr);
                    count++;
                    ptr++;
                }
        } while (ptr != NULL);
    if (count == 0)
        {
            printf("argv[2] is not a substring of argv[1].\n");
        }
    else
        {

```

```
    printf("argv[2] occurs %d times in argv[1].\n", count);
}
return EXIT_SUCCESS;
}
```

6.5 Practice Problems

1. Implement the `int strlen(char * str)` function. This function counts the number of characters until reaching the special character `'\0'`. To distinguish it from the `strlen` function provided by C, we call our function `mystrlen`.
2. Implement the `int strlen(char * str)` function using recursion. If the value of `str` is `'\0'`, the function returns zero. Otherwise, the function calls itself with `str++` as the argument. When the recursive call returns, add one to the value and return the sum.
3. Implement the `int strcmp(char * str1, char * str2)` function. To distinguish it from the `strcmp` function provided by C, we call our function `mystrcmp`.
4. Write a function `int countA2M(string * str)` that counts the occurrences of characters between `'A'` and `'M'`, including these two characters.

Chapter 7

Structure

What are the differences between `int` and `double`? They are *data types*. Data types specify

- format of data. Integers and double-precision floating-point numbers are stored differently.
- the range of data and the size for storage. For a 32-bit machine, valid integers are between -2,147,483,648 (-2^{31} , about -10^9) and 2,147,483,647 ($2^{31} - 1$). In contrast, the absolute value of a double-precision floating number can be as small as 10^{-308} and as large as 10^{308} (approximately).
- operation. Because the two types have different formats, when a program has a statement `a + b`, the actual operations depends on whether `a` and `b` are integers or floating-point numbers. Also, some operations are restricted to certain data types. For example, `switch` must use an integer; `double` cannot be used for `switch`.

C also supports other data types, such as `char` and `float`. C does not have a separate type for strings; instead, C uses arrays of `char` for a strings, as explained in the previous chapter. A natural question is whether C allows programmers to create new types. The answer is yes. Why would programmers want to create new types? The most obvious reason is to put related data together. For example, a college student has a name (string), year (integer, between 1 and 4), grade point average (floating-point), and so on.

7.1 Vector

A vector has three components: x , y , and z . It is desirable to create a new type called `Vector` and put these data together. Programmers can create new data types by using `typedef struct`. This means creating a *structure* that contains multiple pieces of data. By borrowing the terms used in C++ and Java, we call each piece of data an *attribute*. The structure's name is given at the end of the structure, after `}` before `;`.


```

/* file: vector.c
   purpose: a programmer-defined type called Vector
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct
{
    int x;
    int y;
    int z;
} Vector; /* don't forget ; */

int main(int argc, char * argv[])
{
    Vector v1;
    v1.x = 3;
    v1.y = 6;
    v1.z = -2;
    printf("The vector is (%d, %d, %d).\n", v1.x, v1.y, v1.z);
    return EXIT_SUCCESS;
}

```

After creating the structure, we can use it to create an *object* (borrowing the term from C++ and Java again). In this program, `v1` is a `Vector` object. This object has three attributes. To access an attribute, we need to add a period after `v1`, such as `v1.x` and `v1.y`. The program's output is shown below

The vector is (3, 6, -2).

Can you compare two objects using `==` or `!=`? Can you do something like

```

Vector v1;
Vector v2;
v1.x = 1;
v1.y = 2;
v1.z = 3;
v2.x = 0;
v2.y = -1;
v2.z = -2;

if (v1 != v2)
{
    printf("v1 and v2 are different.\n");
}

```

The answer is no. If you do so, gcc will tell you

```
invalid operands to binary !=
```

If you want to compare two `Vector` objects, you have to compare each attribute separately.

7.2 Object as an Argument

A `Vector` object can be passed as a function argument. *All attributes are copied to the argument of the called function.* The following example moves printing to a different function.

```
/* file: vectorarg.c
   purpose: pass a Vector object as a function argument
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct
{
    int x;
    int y;
    int z;
} Vector;

void printVector(Vector v)
{
    printf("The vector is (%d, %d, %d).\n", v.x, v.y, v.z);
}

int main(int argc, char * argv[])
{
    Vector v1;
    v1.x = 3;
    v1.y = 6;
    v1.z = -2;
    printVector(v1);
    return EXIT_SUCCESS;
}
```

How do we know that the attributes are copied? The following program shows that `changeVector` changes the argument. However, inside `main`, the attributes of `v1` are unchanged.

```

/* file: vectorarg2.c
   purpose: change a Vector object in a function
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct
{
    int x;
    int y;
    int z;
} Vector;

void printVector(Vector v)
{
    printf("The vector is (%d, %d, %d).\n", v.x, v.y, v.z);
}

void changeVector(Vector v)
{
    v.x = 5;
    v.y = -3;
    v.z = 7;
    printVector(v);
}

int main(int argc, char * argv[])
{
    Vector v1;
    v1.x = 3;
    v1.y = 6;
    v1.z = -2;
    printVector(v1);
    changeVector(v1);
    printVector(v1);
    return EXIT_SUCCESS;
}

```

The output of this program is

```

The vector is (3, 6, -2).
The vector is (5, -3, 7).
The vector is (3, 6, -2).

```

What really happens when a function's argument is an object? This can be explained by showing the call stack before calling `changeVector`.

Frame	Symbol	Address	Value
main	v1.z	102	-2
	v1.y	101	6
	v1.x	100	3

Calling `changeVector` pushes its frame to the call stack. The argument is an object and has three attributes. The values are copied from the calling function.

Frame	Symbol	Address	Value
changeVector	v.z	106	-2
	v.y	105	6
	v.x	104	3
	-	103	return location
main	v1.z	102	-2
	v1.y	101	6
	v1.x	100	3

The function `changeVector` changes the attributes. However, all these changes occur in the function's frame.

Frame	Symbol	Address	Value
changeVector	v.z	106	7
	v.y	105	-3
	v.x	104	5
	-	103	return location
main	v1.z	102	-2
	v1.y	101	6
	v1.x	100	3

When `changeVector` finishes, the frame is popped and the program resumes at `main`. The call stack is shown below.

Frame	Symbol	Address	Value
main	v1.z	102	-2
	v1.y	101	6
	v1.x	100	3

The attributes of `v1` are unchanged.

7.3 Object and Pointer

Is it possible to change an object's attributes inside a function and keep the changes even after the function returns? The answer is yes. We need to use pointers.

```
/* file: vectorptr.c
   purpose: change a Vector object in a function by using a pointer
   */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct
{
    int x;
    int y;
    int z;
} Vector;

void printVector(Vector v)
{
    printf("The vector is (%d, %d, %d).\n", v.x, v.y, v.z);
}

void changeVector(Vector * p)
{
    p -> x = 5;
    p -> y = -3;
    p -> z = 7;
    printVector(* p);
}

int main(int argc, char * argv[])
{
    Vector v1;
    v1.x = 3;
    v1.y = 6;
    v1.z = -2;
    printVector(v1);
    changeVector(& v1);
    printVector(v1);
    return EXIT_SUCCESS;
}
```

The `changeVector` function's argument is a pointer.

```
void changeVector(Vector * p)
```

Remember the four different ways using `*` explained in Table 4.1. The argument is the second way using `*`: declaring `p` is a pointer.

When calling `changeVector`, `main` has to provide the *address* of a `Vector` object, as shown below

```
changeVector(& v1);
```

This can be better understood by showing the call stack

Frame	Symbol	Address	Value
changeVector	p	104	100
	-	103	return location
main	v1.z	102	-2
	v1.y	101	6
	v1.x	100	3

Instead of copying the whole object, attribute by attribute, the argument `p` stores only the address of the object `v1` in `main`. What is the `->` symbol inside `changeVector`? This symbol means *taking the value as an address and adding the appropriate offset to read or write an attribute*. If it is at the left hand side (LHS) of an assignment, the attribute is modified (i.e. written). If it is at the right hand side (RHS) of an assignment, the attribute is read. What is an offset? It means the distance from the beginning of the object to the corresponding attribute. For a `Vector` object, `x` has no offset because it is the first attribute. The second attribute `y` has an offset equal to `sizeof(int)` because `x` is an integer. The third attribute `z` has offset equal to `sizeof(int) + sizeof(int)` because `x` and `y` are both integers. The compiler calculates offsets so programmers do not need to worry about the details. The statement

```
p -> x = 5;
```

changes the value at address 100.

Frame	Symbol	Address	Value
changeVector	p	104	100
	-	103	return location
main	v1.z	102	-2
	v1.y	101	6
	v1.x	100	5

```
p -> y = -3;
```

changes the value at address 101.

Frame	Symbol	Address	Value
changeVector	p	104	100
	-	103	return location
main	v1.z	102	-2
	v1.y	101	-3
	v1.x	100	5

Why do we need to add `*` in front of `p` when `changeVector` calls `printVector`? In `changeVector`, `p` is a pointer. However, `printVector` expects an object because there is no `*` in the line

```
void printVector(Vector v)
```

In `changeVector`, adding `*` in front of `p` is the fourth way using `*` explained in Table 4.1. The object stored at addresses 100-102 is copied to the argument of `printVector`. How do we know the object is copied? Because there is no `*`. You can change `v`'s attribute inside `printVector`. The change is lost when the function finishes.

7.4 Return an Object

Can a function return a `Vector` object? Yes. The following example shows a C++ style constructor.

```
/* file: vectorconstructor.c
   purpose: create a constructor for Vector
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct
{
    int x;
    int y;
    int z;
} Vector; /* don't forget ; */

Vector Vector_construct(int a, int b, int c)
{
    Vector v;
```

```

    v.x = a;
    v.y = b;
    v.z = c;
    return v;
}

void Vector_print(Vector v)
{
    printf("The vector is (%d, %d, %d).\n", v.x, v.y, v.z);
}

int main(int argc, char * argv[])
{
    Vector v1 = Vector_construct(3, 6, -2);
    Vector_print(v1);
    return EXIT_SUCCESS;
}

```

What is the advantage of creating a constructor? It makes the program easier to read. Moreover, it reminds a programmer that a `Vector` object has three attributes because the function has three arguments. This helps programmers remember to initialize all attributes.

7.5 Object and `malloc`

Is it possible to create an object using heap memory, instead of stack memory? Yes.

```

/* file: vectorconmalloc.c
   purpose: create a Vector object using malloc
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct
{
    int x;
    int y;
    int z;
} Vector;

Vector * Vector_construct(int a, int b, int c)
/* notice * */
{
    Vector * v;

```



```

v = malloc(sizeof(Vector));
if (v == NULL) /* allocation fail */
{
    printf("malloc fail\n");
    return NULL;
}
v -> x = a;
v -> y = b;
v -> z = c;
return v;
}

void Vector_destruct(Vector * v)
{
    free (v);
}

void Vector_print(Vector * v)
{
    printf("The vector is (%d, %d, %d).\n", v -> x, v -> y, v -> z);
}

int main(int argc, char * argv[])
{
    Vector * v1 = Vector_construct(3, 6, -2);
    if (v1 == NULL)
    {
        return EXIT_FAILURE;
    }
    Vector_print(v1);
    Vector_destruct(v1);
    return EXIT_SUCCESS;
}

```

To create an object on the heap memory, we need to declare a `Vector` pointer and use `malloc`. Since it is a pointer, we need to use `->` for the attributes. How does the program work? Let's trace the program by showing the call stack. Before calling `Vector_construct`, the call stack has only the frame for the `main` function:

Frame	Symbol	Address	Value
main	v1	100	?

Calling `Vector_construct` pushes a frame to the stack.

Frame	Symbol	Address	Value
Vector_construct	v	106	?
	c	105	-2
	b	104	6
	a	103	3
	-	102	value address 100
	-	101	return location
main	v1	100	?

Calling `malloc` allocates a piece of memory on heap. The size is sufficient to accommodate three integers. Suppose `malloc` returns 60000. The call stack becomes

Frame	Symbol	Address	Value
Vector_construct	v	106	60000
	c	105	-2
	b	104	6
	a	103	3
	-	102	value address 100
	-	101	return location
main	v1	100	?

The heap memory is shown below

Address	Value
60002	?
60001	?
60000	?

The statement

```
v -> x = a;
```

modifies the value at the address 60000.

The heap memory is changed to

Address	Value
60002	?
60001	?
60000	3

When `Vector_construct` returns, `v`'s value is written to the return address 100. Therefore, the call stack becomes

Frame	Symbol	Address	Value
main	v1	100	60000

The memory allocated on heap has to be released by calling `free`. This is the purpose of the destructor `Vector_destruct`.

7.6 Constructor and Destructor

An object may contain pointers and dynamically allocated memory. To handle this type of objects, you need four functions.

- *constructor*: allocate memory and assign values to attributes.
- *destructor*: release memory.
- *copy constructor*: create a new object from an existing object. The existing object may be created by using the constructor or the copy constructor. We borrow terminology from C++ again. This is called *deep copy*.
- *assignment*: modify an object that has already been created by using the constructor or the copy constructor.

Consider a structure called `Person`:

```
/*
  file: person.c
  purpose: constructor, destructor, copy constructor, assignment
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct
{
  char * name;
  int year;
  int month;
  int date;
} Person;

Person * Person_construct(char * n, int y, int m, int d);
```

```

void Person_destruct(Person * p);
Person * Person_copy(Person * p);
/* create a new Person object by copying the attributes of p */
Person * Person_assign(Person * p1, Person * p2);
/* p1 is already a Person object, make its attribute the same
   as p2's the attributes */
void Person_print(Person * p);

int main(int argc, char * argv[])
{
    Person * p1 = Person_construct("Amy", 1989, 8, 21);
    Person * p2 = Person_construct("Jennifer", 1991, 2, 17);
    Person * p3 = Person_copy(p1); /* create p3 the first time */
    Person_print(p1);
    Person_print(p2);
    Person_print(p3);
    p3 = Person_assign(p3, p2);
    Person_print(p3);
    Person_destruct(p1);
    Person_destruct(p2);
    Person_destruct(p3);
    return EXIT_SUCCESS;
}

Person * Person_construct(char * n, int y, int m, int d)
{
    Person * p;
    p = malloc(sizeof(Person));
    if (p == NULL)
    {
        printf("malloc fail\n");
        return NULL;
    }
    p -> name = malloc(sizeof(char) * (strlen(n) + 1));
    /* + 1 for the ending character '\0' */
    strcpy(p -> name, n);
    p -> year = y;
    p -> month = m;
    p -> date = d;
    return p;
}

void Person_destruct(Person * p)
{
    /* notice the order, p must be released after p -> name has been

```

```

        released */
    free (p -> name);
    free (p);
}

Person * Person_copy(Person * p)
{
    return Person_construct(p -> name, p -> year, p -> month, p -> date);
}

Person * Person_assign(Person * p1, Person * p2)
{
    Person_destruct(p1);
    return Person_copy(p2);
}

void Person_print(Person * p)
{
    printf("Name: %s. ", p -> name);
    printf("Date of Birth: %d/%d/%d\n", p -> year, p -> month, p -> date);
}

```

The output of this program is

```

Name: Amy. Date of Birth: 1989/8/21
Name: Jennifer. Date of Birth: 1991/2/17
Name: Amy. Date of Birth: 1989/8/21
Name: Jennifer. Date of Birth: 1991/2/17

```

Let's examine the program from the constructor, `Person_construct`. The function allocates memory for a `Person` object. Each `Person` object contains a name. We want to allocate just enough memory for the name; thus, we use a `char` pointer. The space is allocated based on the length of the name. We need to add one to accommodate the ending character `'\0'`. Forgetting to add one is a common mistake.

We need to allocate the space for the object, i.e. `p`, before allocating memory for `p -> name`. Without the space for `p`, `p -> name` does not exist yet. With the space for `p -> name`, we can copy the name from the argument to the attribute. The remaining statements in the constructor are similar to the ones we have seen in the previous section. The destructor is *symmetric* to the constructor. The constructor allocates memory for `p` before `p -> name`. In the destructor, the memory for `p -> name` is released first.

General rule: the destructor releases memory in the *reverse order* as the memory is allocated in the constructor.

Why do we need the copy constructor? Why can't we simply write

```
p3 = p1;
```

Does this cause syntax error? No, there is no syntax error. This simply makes p1 and p3 have the same value, i.e. they refer to the same address in heap memory.

What is the difference between p1 and p4 in the following code?

```
Person * p1;  
Person p4;  
p1 = Person_construct("Amy", 1989, 8, 21);  
p4.year = 1990;
```

With *, p1 is a pointer; its value is an address. This is the second usage of *, as explained in Table 4.1. In contrast, p4 is not a pointer; it already has space for the attributes (uninitialized). The call stack and heap memory looks like the following (we ignore p2 and p3 for now)

Symbol	Address	Value
p4.date	204	?
p4.month	203	?
p4.year	202	?
p4.name	201	?
p1	200	10000

call stack

Address	Value
20003	'\0'
20002	'y'
20001	'm'
20000	'A'
10003	21
10002	8
10001	1989
10000	20000

heap memory

Notice that p1 is a pointer so its value is an address in the heap memory. In this example, we use 10000 for the value of p1. The first attribute p1 -> name is also a pointer. Thus, at 10000 stores another address. We use 20000 for this example. The actual name, starting with the character 'A', is stored at address 20000.

We can assign values to p4's attributes

```
p4.year = 1991;  
p4.month = 1;  
p4.date = 17;
```

Symbol	Address	Value
p4.date	204	17
p4.month	203	1
p4.year	202	1991
p4.name	201	?
p1	200	10000

call stack

Address	Value
20003	'\0'
20002	'y'
20001	'm'
20000	'A'
10003	21
10002	8
10001	1989
10000	20000

heap memory

Assign Alice to p4's name.

```
p4.name = malloc(sizeof(char) * (strlen("Alice") + 1));
strcpy(p4.name, "Alice");
```

Suppose malloc returns address 9000. The call stack and the heap memory becomes

Symbol	Address	Value
p4.date	204	17
p4.month	203	1
p4.year	202	1991
p4.name	201	9000
p1	200	10000

call stack

Address	Value
20003	'\0'
20002	'y'
20001	'm'
20000	'A'
10003	21
10002	8
10001	1989
10000	20000
9005	'\0'
9004	'e'
9003	'c'
9002	'i'
9001	'l'
9000	'A'

heap memory

What happens if we have the following code?

```
Person p5;
```

Since p5 is not pointer, it has space for the attributes.

Symbol	Address	Value
p5.date	208	?
p5.month	207	?
p5.year	206	?
p5.name	205	?
p4.date	204	17
p4.month	203	1
p4.year	202	1991
p4.name	201	9000
p1	200	10000

call stack

Address	Value
20003	'\0'
20002	'y'
20001	'm'
20000	'A'
10003	21
10002	8
10001	1989
10000	20000
9005	'\0'
9004	'e'
9003	'c'
9002	'i'
9001	'l'
9000	'A'

heap memory

The following statement copies p4's attributes to p5's attributes, one by one.

```
p5 = p4;
```

Symbol	Address	Value
p5.date	204	17
p5.month	203	1
p5.year	202	1991
p5.name	203	9000
p4.date	203	17
p4.month	203	1
p4.year	202	1991
p4.name	201	9000
p1	200	10000

call stack

Address	Value
20003	'\0'
20002	'y'
20001	'm'
20000	'A'
10003	21
10002	8
10001	1989
10000	20000
9005	'\0'
9004	'e'
9003	'c'
9002	'i'
9001	'l'
9000	'A'

heap memory

There is no problem for the date of birth: `p4` and `p5` have *separate* space. The name attributes are pointers and they have the same value; thus, they refer to the same address in the heap memory. This is called *shallow copy*. Is this a problem? In most cases, yes.

Consider this code

```
int x = 61;
int y = x;
x = 92;
```

What is `y`'s value? It is still 61, even though `x` has been changed to 92. The problem of shallow copy is that they have the same value and refer to the same address in the heap memory. Changing `p4.name` also changes `p5.name`; changing `p5.name` also changes `p4.name`. Since we never allocate memory for `p5.name`, we should not release the memory, i.e. we should *not* have the following statement:

```
free(p5.name);
```

We should release only `p4.name`:

```
free(p4.name);
```

If we release the same heap memory twice, the program will crash.

Now, we can understand why we need to have copy constructor and assignment. We need to handle memory allocation and release. The copy constructor allocates separate memory space so that changing one later does not affect the other. This is called *deep copy*. If one object already has memory space, the assignment function has to provide a sufficient amount of memory. In this program, we want to change `p3 -> name` from Amy to Jennifer. Since Jennifer is longer, the new name needs more space. If there is already enough memory (for example, changing from Jennifer to Amy), the memory can be used. This requires the program to keep track of the amount of space allocated. This program takes a simpler approach: always releases memory first and then allocates memory again.

Earlier, I said shallow copy is problematic in most cases. What are the cases when it is not problematic? When would we actually want to have shallow copy? First, you need to know that your program performs shallow copy and objects share the same address in the heap memory. This is an advance topic so I will only describe the concept. Sometimes, you do want several objects to share the same memory space. This can be useful when the required memory space is very large and you don't want every object to have a copy. If the data are not changed, the objects share the memory. The program needs to keep track of the number of objects using this piece of memory so that the memory is released when no object uses the memory any more. If one object wants to change the data, the program does make a copy. This is called *copy on write*.

7.7 Structure in Structure

Can a structure's attribute be a structure? Yes. This is an example moving a `Person`'s date of birth from three integers to one `Date` object.

```
/*
   file: person2.c
   purpose: a object's attribute is an object
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct
{
    int year;
    int month;
    int date;
} DateOfBirth;

DateOfBirth DateOfBirth_construct(int y, int m, int d)
{
    DateOfBirth dob;
    dob.year = y;
    dob.month = m;
    dob.date = d;
    return dob;
}

void DateOfBirth_print(DateOfBirth d)
{
    printf("Date of Birth: %d/%d/%d\n", d.year, d.month, d.date);
}

typedef struct
{
    char * name;
    DateOfBirth dob;
} Person;

Person * Person_construct(char * n, int y, int m, int d);
void Person_destruct(Person * p);
Person * Person_copy(Person * p);
/* create a new Person object by copying the attributes of p */
Person * Person_assign(Person * p1, Person * p2);
/* p1 is already a Person object, make its attribute the same
   as p2's the attributes */
```

```

void Person_print(Person * p);

int main(int argc, char * argv[])
{
    Person * p1 = Person_construct("Amy", 1989, 8, 21);
    Person * p2 = Person_construct("Jennifer", 1991, 2, 17);
    Person * p3 = Person_copy(p1); /* create p3 the first time */
    Person_print(p1);
    Person_print(p2);
    Person_print(p3);
    p3 = Person_assign(p3, p2);
    Person_print(p3);
    Person_destruct(p1);
    Person_destruct(p2);
    Person_destruct(p3);
    return EXIT_SUCCESS;
}

Person * Person_construct(char * n, int y, int m, int d)
{
    Person * p;
    p = malloc(sizeof(Person));
    if (p == NULL)
    {
        printf("malloc fail\n");
        return NULL;
    }
    p -> name = malloc(sizeof(char) * (strlen(n) + 1));
    /* + 1 for the ending character '\0' */
    strcpy(p -> name, n);
    p -> dob = DateOfBirth_construct(y, m, d);
    return p;
}

void Person_destruct(Person * p)
{
    /* notice the order, p must be released after p -> name has been
       released */
    free (p -> name);
    free (p);
}

Person * Person_copy(Person * p)
{
    return Person_construct(p -> name, p -> dob.year,

```

```

        p -> dob.month, p -> dob.date);
}

Person * Person_assign(Person * p1, Person * p2)
{
    Person_destruct(p1);
    return Person_copy(p2);
}

void Person_print(Person * p)
{
    printf("Name: %s. ", p -> name);
    DateOfBirth_print(p -> dob);
}

```

We create a *hierarchy* of structures. `Person_construct` calls `DateOfBirth_construct`. `Person_print` calls `DateOfBirth_print`. What is the advantage? As your programs become more complex, you may find such a hierarchy helpful for organization. You don't want to create one structure that has everything. Instead, you should put related data together and create a structure, then organize related structures into another structure. Each structure has a constructor to initialize all attributes. If a structure has pointers for dynamically allocated memory, make sure you also have a destructor. If deep copy is what you want (in most cases), remember the copy constructor and the assignment function.

Chapter 8

File Operations

A program has many ways to obtain input data, for example

- using `scanf` to get data from a user through keyboard.
- using `argc` and `argv` to get data from the command line.
- using file operations to get data stored on a disk.

For the third option, the program must obtain the file's name first. The file's name itself is another piece of data. The following examples use `argv[1]` as a file's name. The program must check whether `argc` is at least two to decide whether `argv[1]` is available.

```
/* file: file1.c
   purpose: check whether argv[1] exists
*/
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
    if (argc < 2)
    {
        printf("Need to provide the file's name.\n");
        return EXIT_FAILURE;
    }
    printf("The name of the file is %s.\n", argv[1]);
    return EXIT_SUCCESS;
}
```

Running the program without the file's name will produce a message and the program returns `EXIT_FAILURE`.

```
> gcc -Wall -Wshadow file1.c -o file1
> ./file1
Need to provide the file's name.
```

When the main function returns, the program terminates. By returning `EXIT_FAILURE`, this program informs the terminal that this program ends abnormally. If the file's name is given, the program prints the file's name:

```
> ./file1 data
The name of the file is data.
```

8.1 Read Character by Character Using `fgetc`

After getting the file's name, we need to open the file for reading. This is accomplished by calling the `fopen` function. The function has two arguments: the first is the name of the file; the second is the mode to open this file. In this example, we open a text file to read. This mode is specified by "r" in the second argument.

Calling `fopen` does not always open the file successfully. Many reasons can cause failures. One reason is that the file does not exist in the current directory. Another reason is that the file cannot be read even if it exists because the user does not have the permission to read this file. When `fopen` fails, it returns `NULL`.

After opening the file, there are many ways to read the data in the file. The first method is `fgetc`.

```
/* file: file2.c
   purpose: show how to use fgetc. This program counts the number of
   characters in the input file.
*/
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
    FILE * fptr;
    int ch;
    int counter = 0;
    if (argc < 2)
    {
        printf("Need to provide the file's name.\n");
        return EXIT_FAILURE;
    }
    fptr = fopen(argv[1], "r");
```

```

if (fptr == NULL)
{
    printf("fopen fail.\n");
    return EXIT_FAILURE;
}
printf("The name of the file is %s.\n", argv[1]);
do
{
    ch = fgetc(fptr);
    if (ch != EOF)
    {
        counter ++;
    }
} while (ch != EOF);
fclose(fptr);
printf("The file has %d characters.\n", counter);
return EXIT_SUCCESS;
}

```

This program counts the number of characters. A character may be an English letter ('a' to 'z' or 'A' to 'Z'), a digit ('0' to '9'), a punctuation mark (such as ',' and ';'), and space. At the end of each line, a new line character ('\n') is also counted. When the program reaches the end of the file, a special character is returned by `fgetc`. This character is called *end of file* and it is `EOF` defined in `stdio.h`. This program reports the number of characters. Suppose we use the program to count the number of characters in this C program, this is the result

```

> gcc -Wall -Wshadow file2.c -o file2
> ./file2 file2.c
The name of the file is file2.c.
The file has 716 characters.

```

Linux has a program called `wc` and it reports the numbers of lines, words, and characters in a file. The program also reports 712 characters:

```

> wc file2.c
35 105 716 file2.c

```

Many operating systems restrict the number of opened files. Your programs should call `fclose` after finishing all operations. It is a good habit to type `fclose` right after typing `fopen` and then *insert* appropriate code between them. This can prevent you from forgetting calling `fclose`.

How does `fgetc` work? Calling `fopen` creates a `FILE` pointer. A text file is considered a *stream* of characters, as illustrated in Figure 8.1. This stream starts at the beginning of the file. Every time `fgetc` is called, one character is taken out from the stream. If the program keeps calling `fgetc`, eventually all characters are taken and `EOF` is returned.

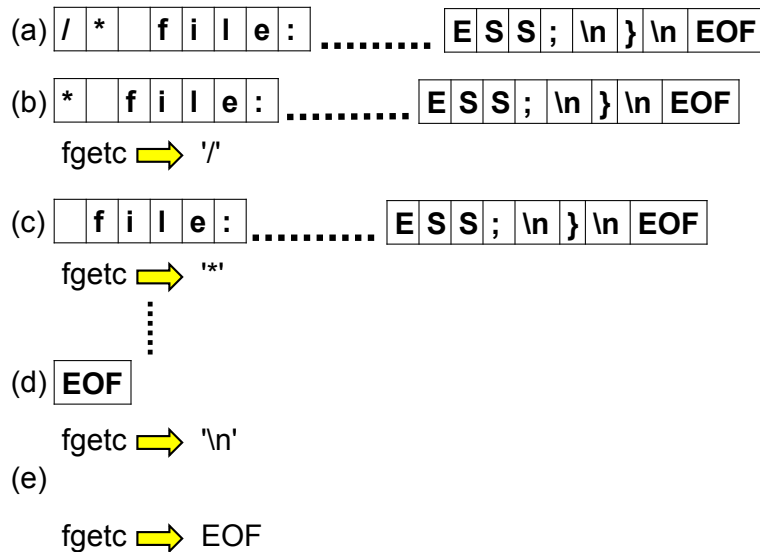


Figure 8.1: A text file is like a stream. This example uses `file2.c` as the input file. (a) After calling `fopen`, the stream starts from the very first character of the file and ends with `EOF`. (b)(c) Calling `fgetc` each time takes one character out from the stream. (d) After calling `fgetc` enough times, the characters in the file are retrieved. (e) Finally, the end of file character `EOF` is returned.

8.2 Read Integer

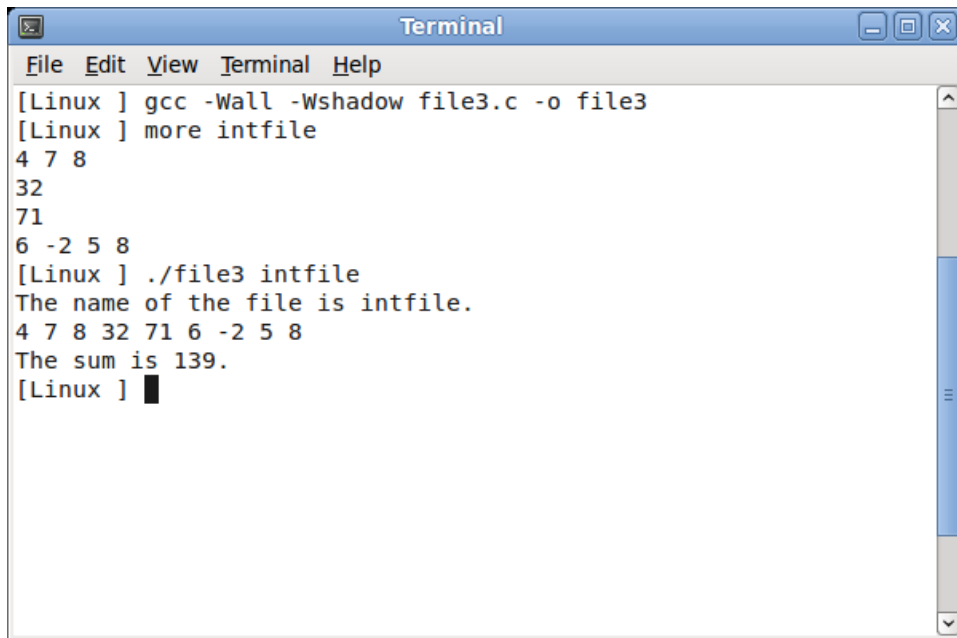
In addition to `fgetc`, C provides many functions to read data from a file. One of them is `fscanf`. It is very similar to `scanf`, except that it needs one more argument. The first argument is a `FILE` pointer. The following program adds the numbers in a file.

```

/* file: file3.c
   purpose: show how to use fscanf. This program reads numbers from a
   file and add them together
*/
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
    FILE * fptr;
    int val;
    int sum = 0;
    if (argc < 2)
    {
        printf("Need to provide the file's name.\n");
        return EXIT_FAILURE;
    }
    fptr = fopen(argv[1], "r");

```

```
if (fptr == NULL)
{
    printf("fopen fail.\n");
    return EXIT_FAILURE;
}
printf("The name of the file is %s.\n", argv[1]);
while (fscanf(fptr, "%d", & val) == 1)
{
    printf("%d ", val);
    sum += val;
}
fclose(fptr);
printf("\nThe sum is %d.\n", sum);
return EXIT_SUCCESS;
}
```



```
Terminal
File Edit View Terminal Help
[Linux ] gcc -Wall -Wshadow file3.c -o file3
[Linux ] more intfile
4 7 8
32
71
6 -2 5 8
[Linux ] ./file3 intfile
The name of the file is intfile.
4 7 8 32 71 6 -2 5 8
The sum is 139.
[Linux ]
```

Figure 8.2: A program uses `fscanf` to read integers from a file; `fscanf` is able to read numbers separated by space or new line (`'\n'`).

The program keeps reading until no more integer can be read. The return value of `fscanf` is the number of successful data retrieval. This example reads one integer each time so the condition is

```
while (fscanf(fp, "%d", &val) == 1)
```

8.3 Read and Write Integers

Write a program that takes command-line arguments: `argv[1]`: name of first input file `argv[2]`: name of second input file `argv[3]`: name of output file

The input files contain integers, one integer per line.

This program reads one integer from each input file, add them, and store the sum (another integer) into the output file, one integer per line.

You can assume that the two input files have the same number of integers, but you do not know the number in advance.

For example, input file 1 is 8 19 -7 6

Input file 2 is 21 36 81 -7

The output file should be 29 55 74 -1

Your program should ignore space (if any) in each line. Your program should also ignore empty lines (if any).

Do not worry about overflow or underflow of integers.

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

int main(int argc, char * argv[])
{
    /* check whether enough command-line arguments are given */
    /* If not enough, return EXIT_FAILURE */
    if (argc < 4)
    {
        return EXIT_FAILURE;
    }

    /* open the two input files */
    FILE * fin1;
    FILE * fin2;
    fin1 = fopen(argv[1], "r");
    fin2 = fopen(argv[2], "r");

    /* open the output file */
    FILE * fout;
    fout = fopen(argv[3], "w");

    /* check whether the files are open successfully */
    /* if any one fails to open, return EXIT_FAILURE */
    if ((fin1 == NULL) || (fin2 == NULL) || (fout == NULL))
    {
        return EXIT_FAILURE;
    }

    /* repeat until reaching the end of file */
    /* read one integer from input file 1 */
    /* read one integer from input file 2 */
    /* add the two integers and write the sum to the output file */
    while ((! feof(fin1)) && (! feof(fin2)))
    {
        int val1;
        int val2;
        int vals;
        int rt1;
        int rt2;
```

```

    rt1 = fscanf(fin1, "%d", & val1);
    rt2 = fscanf(fin2, "%d", & val2);
    if ((rt1 == 1) && (rt2 == 1))
    {
        vals = val1 + val2;
        fprintf(fout, "%d\n", vals);
    }
}

/* close the files */
fclose (fin1);
fclose (fin2);
fclose (fout);

return EXIT_SUCCESS;
}

```

8.4 Read String

Another commonly function for reading data from a file is `fgets`. This function takes three arguments:

1. an array of characters to store the data
2. the number of characters to read
3. a `FILE` pointer

If the second argument is `size`, the function reads as many as `size - 1` characters from the file. The function adds the ending character, `'\0'`, automatically. The function may reads fewer characters if (i) a new line character occurs before reading `size - 1` characters or (ii) the file has reached the end.

The following program reads from a file line by line and counts the number of lines in the file. We assume that the maximum length of each line is 80 characters, i.e. at least one `'\n'` within every 80 characters.

```

/* file: file4.c
   purpose: show how to use fgets. Read a file line by line.
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

```

```

#define MAX_LINE_LENGTH 81
int main(int argc, char * argv[])
{
    FILE * fptr;
    int numLine = 0;
    char oneLine[MAX_LINE_LENGTH];
    if (argc < 2)
        {
            printf("Need to provide the file's name.\n");
            return EXIT_FAILURE;
        }
    fptr = fopen(argv[1], "r");
    if (fptr == NULL)
        {
            printf("fopen fail.\n");
            return EXIT_FAILURE;
        }
    printf("The name of the file is %s.\n", argv[1]);
    while (fgets(oneLine, MAX_LINE_LENGTH, fptr) != NULL)
        {
            numLine ++;
        }
    fclose(fptr);
    printf("The file has %d lines.\n", numLine);
    return EXIT_SUCCESS;
}

```

When the program cannot read from the file any more, `fgets` returns `NULL`. This means that the end of the file has been reached.

8.5 `fseek` and `ftell`

When you use `fopen` to open a file, you start from the very beginning of the file. Every time you read some data, you move forward, until reaching the end of the file. You can use `fseek` to move to a specific location in the file. You can use `ftell` to know the current location in the file.

8.6 Binary File and Object

This section explains how to write an object to a file and how to read an object from a file. We will talk about both text file and binary file. We consider `Vector` objects explained on page 115.

```

/* file: vectorfile.c
   purpose: show how to write and read Vector objects using files
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct
{
    int x;
    int y;
    int z;
} Vector;

Vector Vector_construct(int a, int b, int c)
{
    Vector v;
    v.x = a;
    v.y = b;
    v.z = c;
    return v;
}

void Vector_print(char * name, Vector v)
{
    printf("%s is (%d, %d, %d).\n", name, v.x, v.y, v.z);
}

void Vector_writet(char * filename, Vector v)
/* writet means write to a text file */
{
    FILE * fptr;
    fptr = fopen(filename, "w"); /* text file */
    if (fptr == NULL)
    {
        printf("Vector_writet fopen fail\n");
        return;
    }
    fprintf(fptr, "%d %d %d", v.x, v.y, v.z);
    fclose (fptr);
}

Vector Vector_readt(char * filename)
/* readt means read from a text file */
{
    Vector v = Vector_construct(0, 0, 0);
}

```

```

FILE * fptr;
fptr = fopen(filename, "r"); /* text file */
if (fptr == NULL)
{
    printf("Vector_readt fopen fail\n");
    return v;
}
if (fscanf(fptr, "%d %d %d", & v.x, & v.y, & v.z) != 3)
{
    printf("fscanf fail\n");
}
fclose (fptr);
return v;
}

void Vector_writeb(char * filename, Vector v)
/* writeb means write to a binary file */
{
    FILE * fptr;
    fptr = fopen(filename, "wb"); /* binary file */
    if (fptr == NULL)
    {
        printf("Vector_writeb fopen fail\n");
        return;
    }
    if (fwrite(& v, sizeof(Vector), 1, fptr) != 1)
    {
        printf("fwrite fail\n");
    }
    fclose (fptr);
}

Vector Vector_readb(char * filename)
/* readb means read from a binary file */
{
    FILE * fptr;
    Vector v = Vector_construct(0, 0, 0);
    fptr = fopen(filename, "rb");
    if (fptr == NULL)
    {
        printf("Vector_readb fopen fail\n");
        return v;
    }
    if (fread(& v, sizeof(Vector), 1, fptr) != 1)
    {

```



```

        printf("fread fail\n");
    }
    return v;
}

int main(int argc, char * argv[])
{
    Vector v1 = Vector_construct(13, 206, -549);
    Vector v2 = Vector_construct(-15, 8762, 1897);
    Vector_print("v1", v1);
    Vector_print("v2", v2);
    printf("=====\n");
    Vector_writet("vectort.dat", v1);
    v2 = Vector_readt("vectort.dat");
    Vector_print("v1", v1);
    Vector_print("v2", v2);

    v1 = Vector_construct(2089, -3357, 1234);
    v2 = Vector_construct(7658, 0, 1876);
    printf("=====\n");
    Vector_print("v1", v1);
    Vector_print("v2", v2);

    Vector_writeb("vectorb.dat", v1);
    v2 = Vector_readb("vectorb.dat");
    printf("=====\n");
    Vector_print("v1", v1);
    Vector_print("v2", v2);
    return EXIT_SUCCESS;
}

```

The program contains two write functions and two read functions. `Vector_writet` and `Vector_readt` use text files. `Vector_writeb` and `Vector_readb` use binary files. When we use text files, writing and reading objects are natural extensions of writing and reading operations explained earlier. The two functions simply write and read one attribute after another. Make sure the two functions have the same orders of attributes. `Vector_writeb` and `Vector_readb` open files in the binary mode by adding `b` in the second argument when calling `fopen`¹. The following table describes the differences of text and binary files.

¹Some operating systems, including Linux, actually ignore `b`. It is used primarily for compatibility among different systems.

Operation	Text File	Binary File
open a file	<code>fopen</code>	<code>fopen‡</code>
write	<code>fprint</code>	<code>fwrite</code>
read	<code>fgetc</code> , <code>fgets</code> , or <code>fscanf</code>	<code>fread</code>

‡ Opening a binary file needs to add `b` in the second argument.

Table 8.1: Functions for opening, writing to, and reading from text and binary files.

To write an object in the binary mode, we use `fwrite`. This function needs four arguments:

1. the *address* of the object. We need to add `&` before the object.
2. the size of the object. We use `sizeof` to find the size of the object. For this example, the size of a `Vector` object is `sizeof(Vector)`.
3. the number of objects. This example writes only one object so the value is 1. If your program has an array of objects, this argument is the number of elements of the array.
4. the `FILE` pointer

The return value of `fwrite` is the number of objects written. This number can be different from the third argument because, for example, the disk is full. It is a good programming habit checking whether the return value is the same as the third argument. The data written by `fwrite` needs to be read by `fread`, not `fscanf`. We also need to give four arguments to `fread` and the order of the arguments is the same as that for `fwrite`.

What are advantages and disadvantages of text or binary files? If data are stored in a text file, you can read it using the `more` command in terminal or editing it using a text editor. However, writing and reading objects need to handle attributes one by one. The order in `Vector_writet` must be the same as the order in `Vector_readt`. Moreover, if one more attribute is added to `Vector`, both functions must be changed. These requirements increase the chances of mistakes; it is easy to change one place and forget to change the other. In contrast, `Vector_writeb` and `Vector_readt` automatically handle the order of attributes. No change is needed if an attribute is added to `Vector`.

8.7 Practice Problems

1. Write a program that takes two arguments (in addition to the program's own name): `argv[1]` is a string to search for; `argv[2]` is the name of a file. Read the file line by line. If a line contains the string, print the line. This is similar to the `grep` function in UNIX. You can find whether one string contains another string by using the `strstr` function. Please note that the string does not need to be a complete word.

2. Improve the solution for the previous problem by reporting the number of times the string occurs. For example, if a line is

```
This is a book on C programming. This book is written for you.
```

In this line, the word “book” appears twice. Therefore, the count is two if the string to search is “book”.

Chapter 9

Program with Multiple Files

So far each program has only one file. It is impossible to write a complex C program using a single file. In fact, when you include a header file, such as

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

you are already using multiple files. You do not implement `printf` in your program, right? Where does it come from? It comes from C libraries— `libc.so` in `/usr/lib/`. You don't need to know where libraries are stored; `gcc` handles this for you. Your programs use multiple files already. In fact, every program you write uses multiple files. Why is C designed in this way? What are the advantages of writing a program using multiple files?

- Reduce redundant work. Many frequently used functions are supported by libraries. You don't have to write your own `printf`, `scanf`, or `malloc`.
- Improve *portability*. In computing, portability means running the same program across different operating systems, such as Linux, Windows, and MacOS. Some functions handle low-level activities related to hardware, such as reading files from a disk or sending packets through a network card. It is better to implement the hardware-specific details in libraries so that your program can run on different computers, as long as your program uses the correct libraries. Usually you need to recompile the source code using the compiler at the target machine.
- Enhance performance. Libraries are usually well optimized so your program can have better performance.
- Partition work for collaboration. When you write a large program with other people, it is natural to break the work into smaller units. Each person is responsible for some units and each unit is stored in a single file.

- Save compilation time. Compiling a large program (such as an operating system) can take many minutes. If the whole program is in a single file and you change only one line, you have to wait for a long time. A better solution is to break the program into many files and compile only the files affected by your change. This may take a few seconds, much better than many minutes.

9.1 Header File

A header file has the `.h` extension but it is the content, not the file name, that makes a particular file a header file. A header may include the following definitions or declarations.

- defining symbolic constants, for example,

```
#define NAME_LENGTH 50
```

- declare functions, for example,

```
void printStudent(Student s);
```

This is a *declaration*, not a *definition* or *implementation*. The declaration provides only the return type, the function's name, and the argument's (or arguments') type (or types). Do not forget `;` after the closing parenthesis.

- define programmer-created data types, for example

```
typedef struct
{
    char name[NAME_LENGTH];
    int year;
    float GPA;
} Student;
```

- include other header files, for example

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

The implementation of functions should be in `.c` files, not header files. Header files are *included*; `.c` files are *compiled* and *linked*. The following is a program using one `.h` file and one `.c` file.

```

#ifndef STUDENT2_H
#define STUDENT2_H
/* file: student2.h
   purpose: a programmer-defined type called Student
*/
#define NAME_LENGTH 50
typedef struct
{
    char name[NAME_LENGTH];
    int year;
    float GPA;
} Student;
#endif

```

```

/* file: student2.c
   purpose: include a programmer-defined header file
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "student2.h"
int main(int argc, char * argv[])
{
    Student s1;
    strcpy(s1.name, "Amy Smith");
    s1.year = 3;
    s1.GPA = 3.6;
    printf("The student's name is %s.\n", s1.name);
    switch (s1.year)
    {
        case 1:
            printf("    first-year.\n");
            break;
        case 2:
            printf("    sophomore.\n");
            break;
        case 3:
            printf("    junior.\n");
            break;
        case 4:
            printf("    senior.\n");
            break;
    }
    printf("    GPA = %4.2f.\n", s1.GPA);
}

```

```
    return EXIT_SUCCESS;
}
```

The first two lines of a header file is

```
#ifndef FILENAME_H
#define FILENAME_H
```

The last line of a header file is

```
#endif
```

This can prevent *multiple inclusion*. Sometimes, the same header file is included multiple times, for example included by two different header files and both of them are included by the same `.c` file. Without this three lines at the very top and the very bottom, `gcc` will report an error when the same header file is included multiple times.

This header file is included by the `student2.c`

```
#include "student2.h"
```

Notice that we use double quotations for programmer-defined header files. For header files provided by C, we use `<` and `>`, for example

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

Compiling this program requires nothing special.

```
> gcc student2.c -o student2
```

Notice that the header file does not appear in the command.

9.2 Compile and Link

The previous section shows a program with one `.h` and one `.c` file. As programs become even more complex, we need to use even more files for better organization. This example moves `printStudent` into another `.c` file called `student3.c`. The main function is in another file called `student3main.c`.

```
/* file: student3.h
 */
#ifndef STUDENT3_H
#define STUDENT3_H
#define NAME_LENGTH 50
typedef struct
{
    char name[NAME_LENGTH];
    int year;
    float GPA;
} Student;
void printStudent(Student s);
#endif
```

```
/* file: student3.c
   purpose: demonstrate how to write a program with 2 .c files
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "student3.h"

void printStudent(Student s)
{
    printf("The student's name is %s.\n", s.name);
    switch (s.year)
    {
        case 1:
            printf("    first-year.\n");
            break;
        case 2:
            printf("    sophomore.\n");
            break;
        case 3:
            printf("    junior.\n");
            break;
        case 4:
```



```

        printf("    senior.\n");
        break;
    }
    printf("    GPA = %4.2f.\n", s.GPA);
}

```

```

/* file: student3main.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "student3.h"

int main(int argc, char * argv[])
{
    Student s1;
    strcpy(s1.name, "Amy Smith");
    s1.year = 3;
    s1.GPA = 3.6;
    printStudent(s1);
    return EXIT_SUCCESS;
}

```

In terms of functionality, this program is the same as the one in the previous section. How do we compile the program? There are two ways. The first is a simple extension of the old way:

```
> gcc -Wall -Wshadow student3.c student3main.c -o student3
```

We simply put the names of *both* .c files after gcc. If the program needs three or more files, put the names of all .c files after gcc.

The second way is composed of two two stages: *compilation* and *linking*. Compilation means converting a source file into an *object* file. Linking takes the needed object files and creates an executable program. Each .c file can be compiled separately. This is called *separate compilation*.

```
> gcc -Wall -Wshadow -c student3.c
> gcc -Wall -Wshadow -c student3main.c
```

These two commands generate *object files*: student3.o and student3main.o. The next command links them to create an executable.

```
> gcc student3.o student3main.o -o student3
```

The second way seems to be a lot of unnecessary trouble. Why does it even exist? If you modify only `student3main.c` without changing `student3.c`, the object file `student3.o` should still be up-to-date. In this case, you do not have to compile `student3.c` again and recreate `student3.o`. You need to compile `student3main.c` and link the object files. This can save a lot of time if a program contains many `.c` files and you change only one of them—you need to compile only the changed one and then link all object files.

9.3 Makefile

Even if you are convinced of the advantage of separate compilation, you are probably not excited about typing

```
> gcc -Wall -Wshadow -o file1.c
> gcc -Wall -Wshadow -o file2.c
> gcc -Wall -Wshadow -o file3.c
...
> gcc file1.o file2.o file3.o ... -o program
```

every time you modify one or several files. Worse, if you want to take advantage of separate compilation—compiling only the files that have been modified—you have to keep track which object files are obsolete. You need to know which `.c` or `.h` files have been modified since the last compilation. This is too much work. Fortunately, `make` in Linux can help. This command takes an input file called `Makefile`. This section explains how to write simple `Makefile`.

The main purpose of `Makefile` is to decide which files need to be recompiled by comparing the time of an object file and the relevant `.c` and `.h` files. If the object file is newer, the `.c` file is not recompiled. If any of the relevant `.c` and `.h` files is newer, `make` generates a new object file.

The `Makefile` for the previous program is shown below

```
# filename: Makefile
GCC = gcc -Wall -Wshadow

student3: student3.o student3main.o
    $(GCC) student3.o student3main.o -o student3

student3.o: student3.c student3.h
    $(GCC) -c student3.c

student3main.o: student3main.c student3.h
    $(GCC) -c student3main.c
```

The first line is a comment. In `Makefile`, anything after `#` is a comment. The next line defines a symbol called `GCC`. It is the `gcc` command with warning messages enabled. We can use `$(GCC)` later and it is replaced by `gcc -Wall -Wshadow`. Defining this symbol makes changes easier later. For example, we can add `-g` to enable debugging. When you are confident that the program is working, you can replace `-g -Wall -Wshadow` by `-O` to enable compiler optimization. This may make your program noticeably faster.

The next two lines

```
student3: student3.o student3main.o
    $(GCC) student3.o student3main.o -o student3
```

mean, “The file `student3` depends on two files `student3.o` and `student3main.o`. If either file is new, execute the following command `$(GCC) student3.o student3main.o -o student3`” namely,

```
gcc -Wall -Wshadow student3.o student3main.o -o student3
```

This is the linking command for creating the executable file `student3`.

Before `$(GCC)` is a tab, not space. This is a common problem in `Makefile`. How do you get a tab? Hit the tab key (depending on the keyboard layout, the key is usually at the left side above `CAPS LOCK` or `CTRL`).

How does `make` know whether `student3.o` or `student3main.o` is new? The next two lines in `Makefile`

```
student3.o: student3.c student3.h
    $(GCC) -c student3.c
```

mean, “`student3.o` depends on `student3.c` and `student3.h`. If either file is new, execute the following command.”

```
gcc -Wall -Wshadow -c student3.c
```

Similarly, these two lines

```
student3main.o: student3main.c student3.h
    $(GCC) -c student3main.c
```

mean “student3main.o depends on student3main.c and student3.h. If either file is new, execute the following command.”

```
gcc -Wall -Wshadow -c student3main.c
```

In Linux, if you type make, the following messages appear

```
> make
gcc -Wall -Wshadow -c student3.c
gcc -Wall -Wshadow -c student3main.c
gcc -Wall -Wshadow student3.o student3main.o -o student3
```

If we type make for the first time, student3.c and student3.h are both new (because student3.o does not exist yet). Thus, make compiles student3.c. Also, student3main.c and student3.h are new; thus, make compiles student3main.c. Now, both student3.o and student3main.o are new and make links them to create student3.

If you type make again, the following message appears.

```
make: `student3' is up to date.
```

Why? The file student3 depends on student3.o and student3main.o but the time of these two files are earlier than the time of student3. Therefore, make knows student3 is up-to-date.

Actually, make also checks whether student3.o is up-to-date by comparing its time with the time of student3.c and student3.h. If student3.o is newer, make will not recompile student3.c. Similarly, make checks whether student3main.o is newer than student3main.c and student3.h.

If you change student3main.c and type make, this is what happens

```
gcc -Wall -Wshadow -c student3main.c
gcc -Wall -Wshadow student3.o student3main.o -o student3
```

Notice that student3.o is still up-to-date so student3.c is not recompiled.

If you change student3.h and type make, both object files will be regenerated because both depend on student3.h. Then, the executable is generated from the new object files.

9.4 Use Makefile for Testing

You can add more commands in Makefile so that you do not have to type frequently used long commands over and over again. Section 5.8 mentions `valgrind` to detecting memory leak. We can add an option in Makefile to invoke `valgrind`.

```
# filename: Makefile
GCC = gcc -Wall -Wshadow

student3: student3.o student3main.o
    $(GCC) student3.o student3main.o -o student3

student3.o: student3.c student3.h
    $(GCC) -c student3.c

student3main.o: student3main.c student3.h
    $(GCC) -c student3main.c

memory: student3
    valgrind --leak-check=yes ./student3
```

Type

```
> make memory
```

The dependence first checks whether `student3` needs to be recompiled. Then, the `valgrind` command runs to check whether the program leaks memory. This is more convenient than typing the long command every time.

You can extend this further. This program's output is

```
The student's name is Amy Smith.
  junior.
  GPA = 3.60.
```

You can save it in a file called `expectedoutput` and change Makefile as follows

```
# filename: Makefile
GCC = gcc -Wall -Wshadow

student3: student3.o student3main.o
    $(GCC) student3.o student3main.o -o student3
```

```
student3.o: student3.c student3.h
    $(GCC) -c student3.c

student3main.o: student3main.c student3.h
    $(GCC) -c student3main.c

memory: student3
    valgrind --leak-check=yes ./student3

test: student3
    ./student3 > newoutput
    diff newoutput expectedout
```

If you type

```
> make test
```

The program will run and save the output in the file `newoutput`. Then, this file is compared with `expectedoutput`. If the two files are the same, no error message occurs. If the two files are different, you will see an error message.

You can combine `memory` and `test` into one as shown below

```
# filename: Makefile
GCC = gcc -Wall -Wshadow

student3: student3.o student3main.o
    $(GCC) student3.o student3main.o -o student3

student3.o: student3.c student3.h
    $(GCC) -c student3.c

student3main.o: student3main.c student3.h
    $(GCC) -c student3main.c

test: student3
    ./student3 > newoutput
    diff newoutput expectedout
    valgrind --leak-check=yes ./student3
```

When you type

```
> make test
```

The program's output is compared with the expected output and `valgrind` checks whether the program leaks memory.

9.5 Practice Problems

1. Consider the list of files:

```
> ls
f1.c f1.h f2.c f2.h f3.c f3.h main.c Makefile
```

This is Makefile

```
GCC = gcc
CFLAGS = -Wall -Wshadow
OBJS = main.o f1.o f2.o
main: $(OBJS)
    $(GCC) $(CFLAGS) $(OBJS) -o main

main.o: main.c f1.h f2.h
    $(GCC) $(CFLAGS) -c main.c

f1.o: f1.h f1.c
    $(GCC) $(CFLAGS) -c f1.c

f2.o: f2.h f2.c
    $(GCC) $(CFLAGS) -c f2.c

clean:
    rm -f *.o main
```

- (a) What is the output on Terminal after you type

```
make clean
make
```

- (b) If you modify `f1.h`, save it, and type `make` in Terminal, what is the output?
- (c) If you modify `f3.c`, save it, and type `make` in Terminal, what is the output?

Chapter 10

Dynamic Structures

We have seen two scenarios about memory allocation. The first is static allocation. The size is known when a program is written. For example

```
int arr[100];
```

This creates an array with 100 elements.

In many cases, the size is unknown when the program is written but known after the program starts. This is the second scenario. An example is shown below; the size is given by a user.

```
int * arr2;
int length;
scanf("%d", & length);
arr2 = malloc(length * sizeof(int));
```

This chapter describes how to handle the third scenario: the size is unknown even after the program starts and memory is allocated and released based on needs. Imagine you are creating a social network program. How many users will register? You certainly want to have millions of users but you cannot control that. You cannot have a pre-determined number, say five million users and reject registrations after there are already five million users. You cannot allocate a huge amount of memory for, say a billion users, because that will waste too much money if they are only two million users. Moreover, users may come and go. Some users register but forget their passwords and create new accounts. You may want to remove accounts that have not been used for more than one year. To manage this type of application, you need to *allocate memory as needs grow and release memory as needs shrink*. This chapter describes how to use *dynamic structures* whose sizes can grow or shrink based on needs. This chapter won't be sufficient for you to build a social network site but it is a starting point.

We want to create a structure that supports the following functions:

- *insert*: adding new data into the structure, allocating memory as needed.
- *search*: determining whether a piece of data is already in the structure.
- *delete*: removing data from the structure, releasing memory if it is no longer needed.
- *print*: print the data stored in the structure.
- *destory*: delete everything before the program ends.

10.1 Linked List

As the name suggests, a linked list is a list and the components are linked. A linked list has some *nodes*. Each node is an object with two attributes.

- a pointer to another node. By convention, this attribute is called `next`. If one node has no next node, the `next` attribute stores `NULL`.
- data. This attribute may be a pointer, an object, or a primitive type such as `int` or `char`.

10.1.1 Node

We will create a new structure that has two attributes. For simplicity, the data attribute is an integer.

```
typedef struct listnode
{
    struct listnode * next;
    int value;
} Node;
```

This structure has two new concepts: (1) We give a temporary name to the structure, `listnode`. This name is needed because (2) one attribute is a pointer to another object *of the same structure*. We call this structure `Node`; `listnode` is used for only creating that attribute and we will not use it any more. Why do we need `listnode` if it is used only once? This structure is “self-referring”. Without `listnode`, `gcc` does not know the type of `next`. Also, `next` must be a pointer. Since `gcc` has not seen the whole structure yet, `gcc` does not know how much memory is needed. If `next` is a pointer, `gcc` knows `next` needs enough memory to store an address. We use `int` as the type of the value. A linked list’s values can have any type: `int`, `char`, or even a structure described in Chapter 7.

The following figure shows three views of the first operation for creating a linked list.

We would like to create a function `List_insert` that does three things: (1) allocating memory, (2) assigning `NULL` to the `next` attribute, and (3) assigning an argument to the `value` attribute. This function makes inserting Nodes easier: we need to call the function, instead of writing the three statements over and over again.

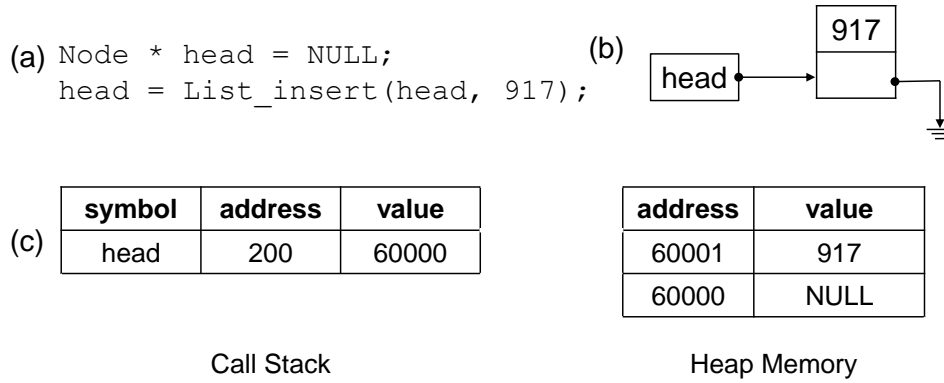


Figure 10.3: Replacing the three lines by using `List_insert`.

If we call `List_insert` one more time with `-504` as the argument, one more `Node` object is created and it is inserted at the beginning of the list. Please notice that the value stored at `head` has changed. Also, there is no guarantee that calling `malloc` twice will obtain consecutive addresses. Thus, we assume there is a gap between the two `Node` objects.

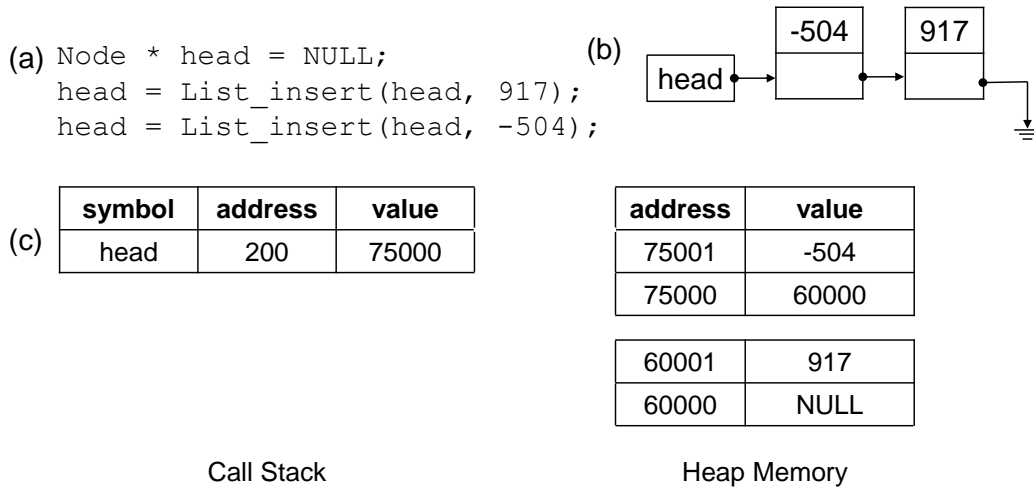


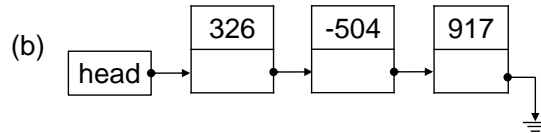
Figure 10.4: Calling `List_insert` again to insert another `Node` object.

Next, we call `List_insert` again with 326 as the argument. A new `Node` object is created and it is inserted at the beginning of the list. The value stored at `head` changes again. We assume there is a gap between this new object and the other two existing objects.

```

Node * head = NULL;
(a) head = List_insert(head, 917);
    head = List_insert(head, -504);
    head = List_insert(head, 326);

```



(c)

symbol	address	value
head	200	83000

address	value
83001	326
83000	75000
75001	-504
75000	60000
60001	917
60000	NULL

Call Stack

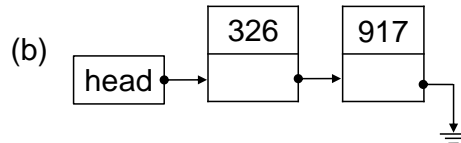
Heap Memory

Figure 10.5: Insert the third object by calling `List_insert` again.

10.1.3 Delete

We want to create a function, `List_delete`. This function deletes the `Node` object whose `value` is the same as the argument. Suppose we want to delete the second `Node` object. After calling `List_delete`, the first and the third objects remain in the linked list. Please notice that the first `Node` object's `next` attribute points to the original third `Node` object (now the second). This ensure that the object is still reachable from the `head`.

```
(a) Node * head = NULL;
    head = List_insert(head, 917);
    head = List_insert(head, -504);
    head = List_insert(head, 326);
    head = List_delete(head, -504);
```



(c)

symbol	address	value
head	200	83000

address	value
83001	326
83000	60000
60001	917
60000	NULL

Call Stack

Heap Memory

Figure 10.6: Insert the third object by calling `List_insert` again.

How do we implement `List_delete`? First, we create a new pointer `p` and make its value the same as `head`'s value. *We have to keep the head's value. If we change head's value, we lose the whole linked list.*

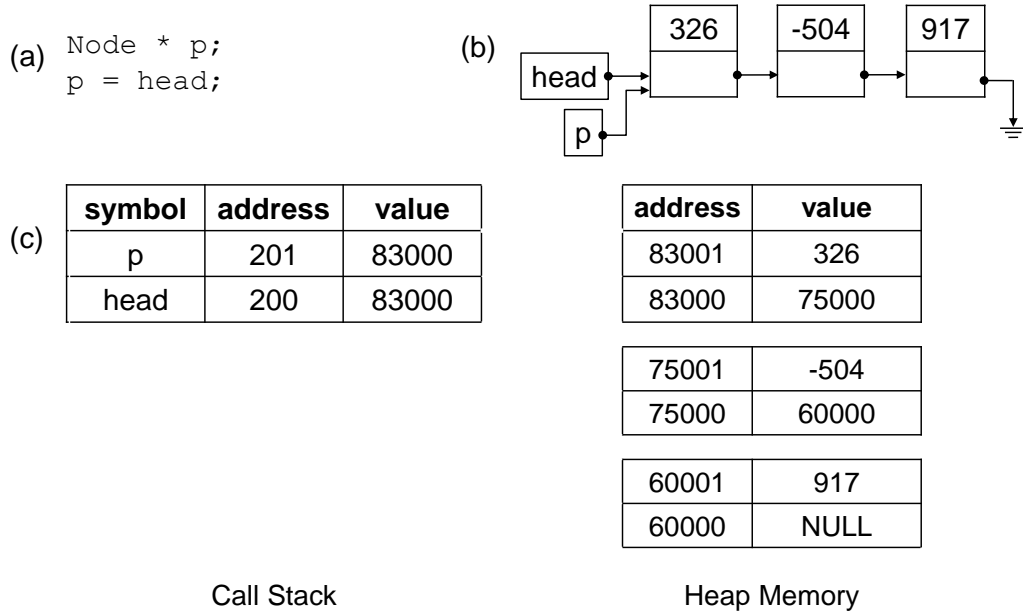
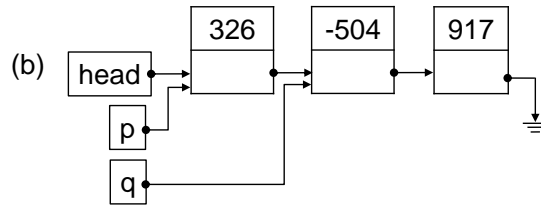


Figure 10.7: To delete a `Node` object, we create a new pointer `p`. Its value is the same as `head`'s value.

Next, we create another pointer q and its value is $p \rightarrow next$.

```
(a) Node * p;
    p = head;
    Node * q;
    q = p -> next;
```



(c)

symbol	address	value
q	202	75000
p	201	83000
head	200	83000

address	value
83001	326
83000	75000

75001	-504
75000	60000

60001	917
60000	NULL

Call Stack

Heap Memory

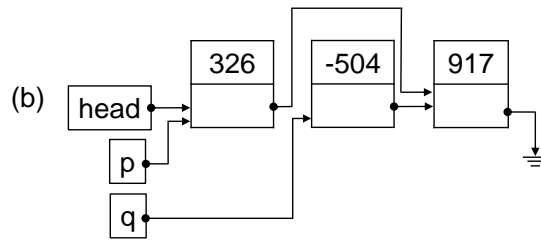
Figure 10.8: We create another pointer q ; its value is the same as $p \rightarrow next$.

Then, we change `p->next` to `q->next`. This bypasses the object that is about to be deleted.

```

Node * p;
(a) p = head;
Node * q;
q = p -> next;
p -> next = q -> next;

```



(c)

symbol	address	value
q	202	75000
p	201	83000
head	200	83000

address	value
83001	326
83000	60000
75001	-504
75000	60000
60001	917
60000	NULL

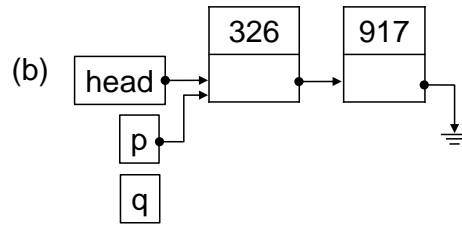
Call Stack

Heap Memory

Figure 10.9: Modify `p->next` and bypass the node that is about to be deleted.

Finally, we release the memory pointed by q.

```
(a) Node * p;
     p = head;
     Node * q;
     q = p -> next;
     p -> next = q -> next;
     free (q);
```



(c)

symbol	address	value
q	202	-
p	201	83000
head	200	83000

address	value
83001	326
83000	60000

60001	917
60000	NULL

Call Stack

Heap Memory

Figure 10.10: Release the memory pointed by q.

10.1.4 C Program

We are ready to show the the code for implementing linked list. The code reflects what is shown in the figures.

```
/*
   file: list.h
*/
#ifndef LIST_H
#define LIST_H
typedef struct listnode
{
    struct listnode * next;
    int value;
} Node;

Node * List_insert(Node * head, int v);
/* insert a value v to a linked list starting with head, return the
   new head */

Node * List_search(Node * head, int v);
/* search a value in a linked list starting with head, return the node
```

```

    whose value is v, or NULL if no such node exists */

Node * List_delete(Node * head, int v);
/* delete the node whose value is v in a linked list starting with
   head, return the head of the remaining list, or NULL if the list is
   empty */

void List_destroy(Node * head);
/* delete every node */

void List_print(Node * head);
/* print the values stored in the linked list */
#endif

```

```

/*
   file: list.c
*/
#include "list.h"
#include <stdio.h>
#include <stdlib.h>
static Node * Node_construct(int v)
/*
   A static function can be called by functions only in this file.
   Functions outside this file cannot call it.
*/
{
    Node * n = malloc(sizeof(Node));
    n -> value = v;
    n -> next = NULL;
    return n;
}

Node * List_insert(Node * head, int v)
{
    printf("insert %d\n", v);
    Node * p = Node_construct(v);
    p -> next = head;
    return p;    /* insert at the beginning */
}

Node * List_search(Node * head, int v)
{
    Node * p = head;
    while (p != NULL)

```

```

    {
        if ((p -> value) == v)
            {
                return p;
            }
        p = p -> next;
    }
return p;
}

Node * List_delete(Node * head, int v)
{
    printf("delete %d\n", v);
    Node * p = head;
    if (p == NULL) /* empty list, do nothing */
        {
            return p;
        }
    /* delete the first node (i.e. head node)? */
    if ((p -> value) == v)
        {
            p = p -> next;
            free (head);
            return p;
        }

    /* not deleting the first node */

    Node * q = p -> next;
    while ((q != NULL) && ((q -> value) != v))
        {
            /* must check whether q is NULL
            before checking q -> value */
            p = p -> next;
            q = q -> next;
        }
    if (q != NULL)
        {
            /* find a node whose value is v */
            p -> next = q -> next;
            free (q);
        }
    return head;
}

```

```

void List_destroy(Node * head)
{
    while (head != NULL)
    {
        Node * p = head -> next;
        free (head);
        head = p;
    }
}

void List_print(Node * head)
{
    printf("\nPrint the whole list:\n");
    while (head != NULL)
    {
        printf("%d ", head -> value);
        head = head -> next;
    }
    printf("\n\n");
}

```

The function `Node_construct` is static. This means the function can be called by another function inside the same file. A static function is invisible outside this file.

We need to pay special attention when deleting the very first Node, i.e., the head. In `List_delete`, we first check whether the list is empty. If it is empty, there is nothing to delete and the function returns immediately. Next, we check whether the first Node's value is the same as the value to delete. If so, the list's head is changed to the second Node. It is possible that the list has only one Node. In this case, `head -> next` is `NULL`. After deleting the first Node, the list is empty. If the first Node is not deleted, we continue to check whether any Node should be deleted by checking

```
while ((q != NULL) && ((q -> value) != v))
```

This checks whether a Node's value is the same as the input argument `v`. We use a feature in C language called *short-circuit*. If `q` is `NULL`, the condition is false and the second part `q->value` is *not* checked.

The following code shows how to use a linked list.

```

/*
file: listmain.c
*/
#include "list.h"
#include <stdlib.h>

```

```

#include <stdio.h>
int main(int argc, char * argv[])
{
    Node * head = NULL; /* must initialize it to NULL */
    head = List_insert(head, 917);
    head = List_insert(head, -504);
    head = List_insert(head, 326);
    List_print(head);
    head = List_delete(head, -504);
    List_print(head);
    head = List_insert(head, 138);
    head = List_insert(head, -64);
    head = List_insert(head, 263);
    List_print(head);

    if (List_search(head, 138) != NULL)
    {
        printf("138 is in the list\n");
    }
    else
    {
        printf("138 is not in the list\n");
    }

    if (List_search(head, 987) != NULL)
    {
        printf("987 is in the list\n");
    }
    else
    {
        printf("987 is not in the list\n");
    }

    /* delete the first Node */
    head = List_delete(head, 263);
    List_print(head);

    /* delete the last Node */
    head = List_delete(head, 917);
    List_print(head);

    /* delete all Nodes */
    List_destroy(head);
    return EXIT_SUCCESS;
}

```

```
}
```

The output of this program is shown below

```
insert 917
insert -504
insert 326
```

```
Print the whole list:
326 -504 917
```

```
delete -504
```

```
Print the whole list:
326 917
```

```
insert 138
insert -64
insert 263
```

```
Print the whole list:
263 -64 138 326 917
```

```
138 is in the list
987 is not in the list
delete 263
```

```
Print the whole list:
-64 138 326 917
```

```
delete 917
```

```
Print the whole list:
-64 138 326
```

10.2 Binary Tree

In a linked list, each `Node` has only one link (`next`). It is certainly possible for each `Node` to have two links. With two links, we can create new dynamic structures. Two commonly used are

- *doubly linked list*. Each `Node` has two links called `previous` and `next`. If `p -> next` is `q`, then `q -> previous` is `p`. When we explain linked lists earlier, we always have to start from `head` and move in only one direction following the `next` attributes. A doubly linked list allows us to move in either direction. We will not discuss doubly linked lists further in this book.
- *binary tree*. Each `Node` has two links called `left` and `right`. A binary tree is different from a doubly linked list because the two links are unrelated.

You can certainly create a new structure in which each object has three or more links. We will not discuss those structures here. This section focuses on binary trees, in particular, *binary search tree*. A binary search tree satisfies the following conditions:

- If `q` is `p -> left`, then `p -> value` is larger than `q -> value`. We say that `q` is `p`'s *left child* and `p` is `q`'s parent. By convention, we use *parent* instead of father or mother.
- If `r` is `p -> right`, then `p -> value` is smaller than `r -> value`. We say that `r` is `p`'s *right child* and `p` is `r`'s parent.

If a node has both children, i.e. neither `q` nor `r` is `NULL`, we say that `q` and `r` are *silblings*. If a node has no child, we call it a *leaf* node.

10.2.1 Node

Let's see the header file for a binary search tree:

```
/*
  file: tree.h
*/
#include <stdio.h>

#ifndef TREE_H
#define TREE_H
typedef struct treenode
{
  struct treenode * left;
  struct treenode * right;
  int value;
} TreeNode;
```



```

TreeNode * Tree_insert(TreeNode * root, int v);
/* insert a value v to a binary search tree starting with root, return
   the new root */

TreeNode * Tree_search(TreeNode * root, int v);
/* search a value in a binary search tree starting with root, return
   the node whose value is v, or NULL if no such node exists */

TreeNode * Tree_delete(TreeNode * root, int v);
/* delete the node whose value is v in a binary search tree starting
   with root, return the root of the remaining tree, or NULL if the
   tree is empty */

void TreeNode_print(TreeNode * n);
/* print the value of one node */

void Tree_print(TreeNode * root);
/* print the values stored in the binary search tree */

void Tree_destruct(TreeNode * root);
/* delete every node */
#endif

```

A binary search tree has similar functionalities as a linked list. Both structures support insert, search, and delete. The differences are the *internal organizations* and more important the efficiency. We will take the same approach as in the previous section by presenting the three views of a binary search tree.

First, we create an empty tree so this is the same as Figure 10.1. The starting point of a binary tree is called `root` instead of `head`.

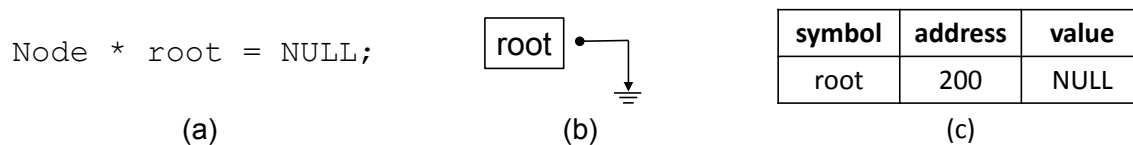


Figure 10.11: An empty tree has one pointer called `root` and its value is `NULL`.

10.2.2 Insert

Next, we create the first tree Node.

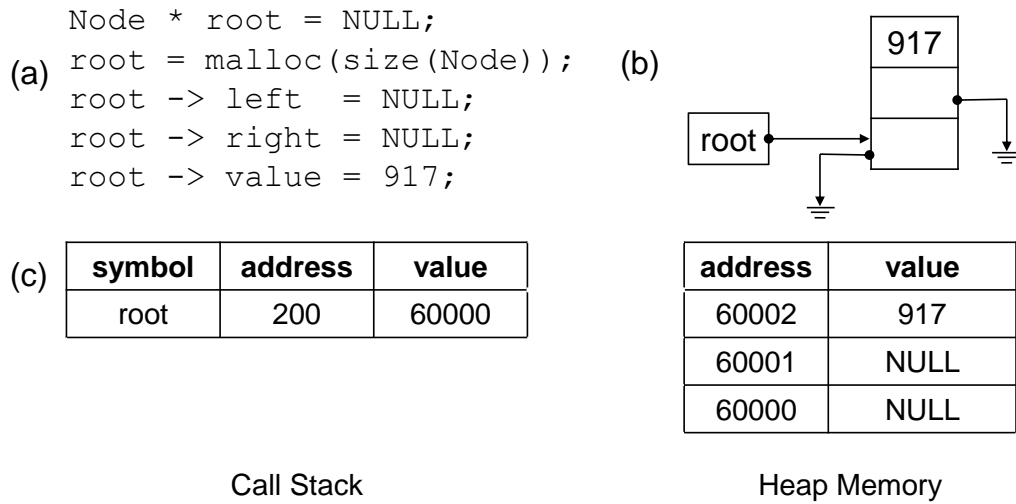


Figure 10.12: A binary tree with only one Node. Both `left` and `right` are `NULL`. This node is the root because it has no parent. It is also a leaf node because it has no child.

We create a function `Tree_insert`. This figure shows the tree after inserting two Nodes. Since `-504` is smaller than `917`, the second Node is on the left side of the first Node. This is an important property of a binary search tree. Also, the `root`'s value does not change after we insert a new Node. In this example, the value remains `60000`.

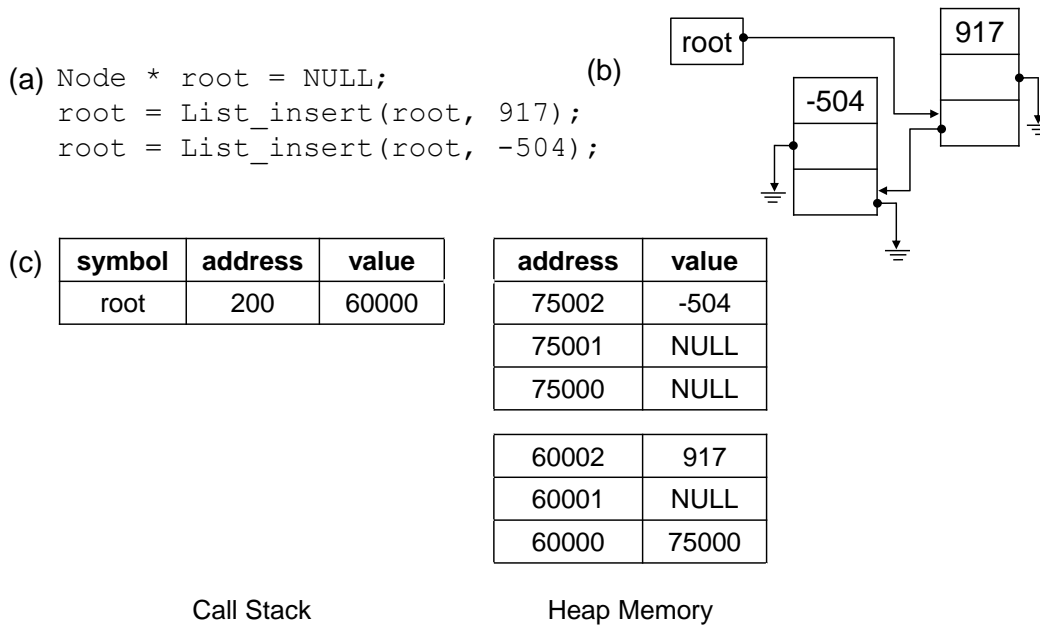


Figure 10.13: A binary tree with two Nodes. The node with value 917 is still the root; it is no longer a leaf node because it has one child. The node with value -504 is a leaf node because it has no child.

Another Node is inserted. The value is 1226 and it is larger than 917; thus, it is at the right side of the first Node.

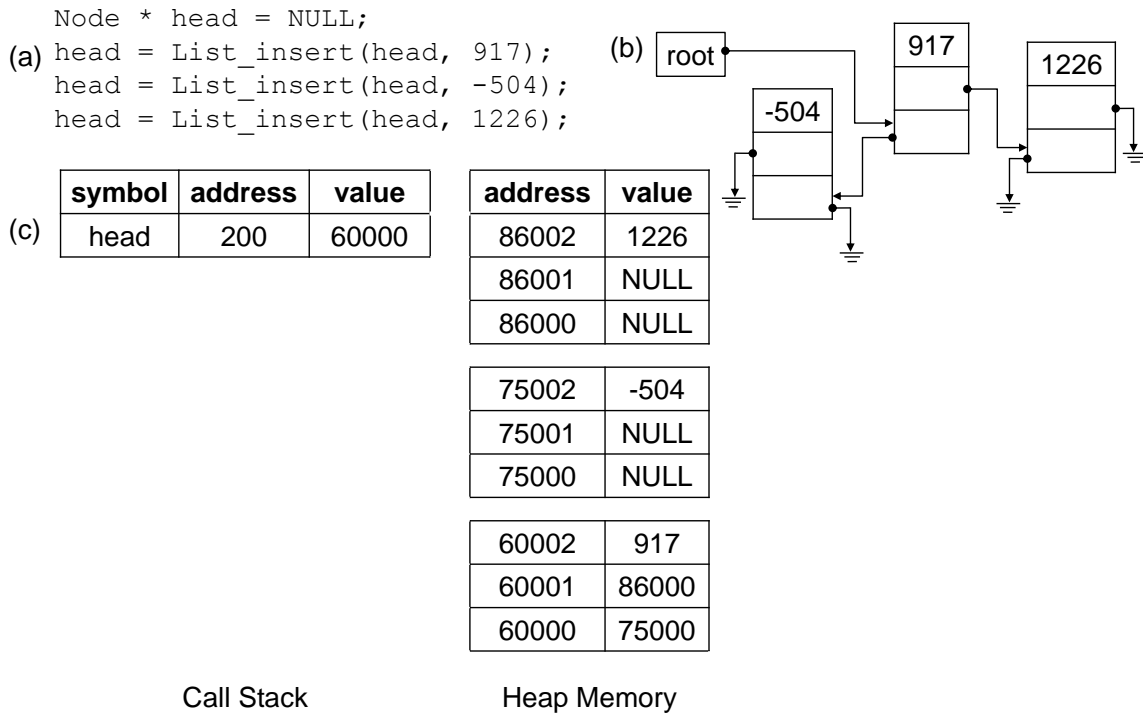


Figure 10.14: A binary tree with three Nodes.

The following figure shows a tree with five Nodes. The second view in (b) becomes complicated; the memory view is also getting quite long. Thus, we create another representation of a tree, as shown in Figure 10.15. In this view, each Node is represented as a circle. In computing, when we draw a “tree”, the root is at the top. This is different from the trees you see in a forest.

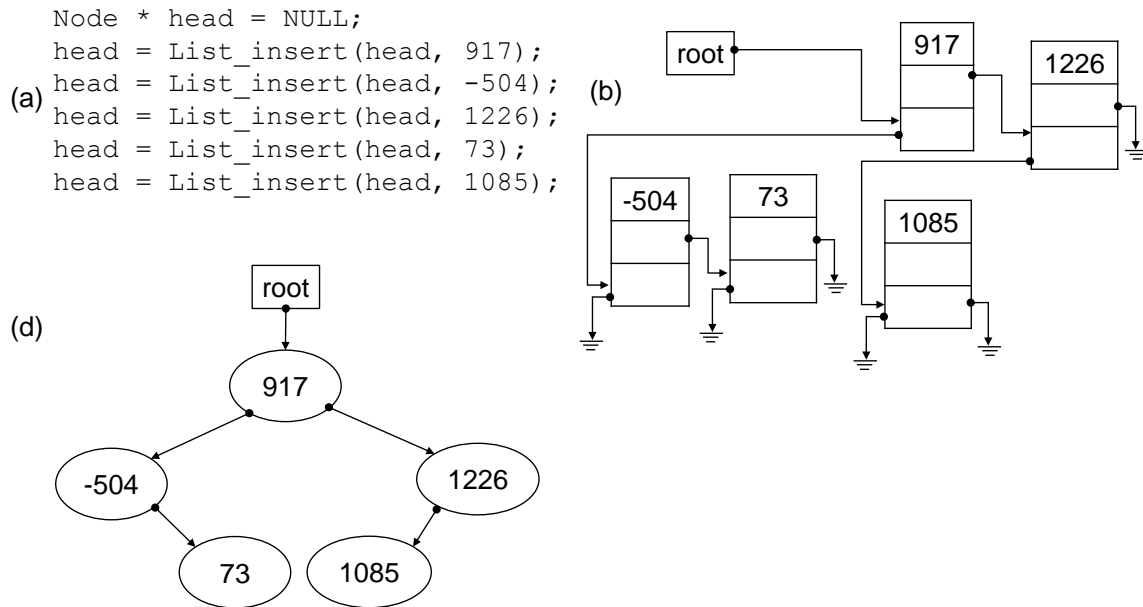


Figure 10.15: A binary tree with five Nodes. We create a new view (d) to simplify the representation of a tree.

The values -504 and 73 are smaller than the value at `root` so both Nodes are at the left side of `root`. Because 73 is larger than -504, 73 is at the right side of -504. Binary search tree has an important property:

For any Node whose value is v , *all* Nodes on the left side have values smaller than v ; *all* Nodes on the right side have values larger than v .

The following shows the functions for implementing `Tree_insert`.

```

#include "tree.h"
#include <stdlib.h>
static TreeNode * TreeNode_construct(int v)
{
    TreeNode * n;
    n = malloc(sizeof(TreeNode));
    n -> left = NULL;
    n -> right = NULL;
    n -> value = v;
    return n;
}

TreeNode * Tree_insert(TreeNode * n, int v)

```

```

{
    if (n == NULL)
    {
        /* tree is empty, create the first node in the tree */
        return TreeNode_construct(v);
    }
    /* tree is not empty, find the correct location to insert the node */
    if (v == (n -> value))
    {
        /* value already stored in the tree */
        return n;
    }
    if (v < (n -> value))
    {
        n -> left = Tree_insert(n -> left, v);
    }
    else
    {
        n -> right = Tree_insert(n -> right, v);
    }
    return n;
}

```

10.2.3 Search

Now, we can understand why it is called a *binary search* tree. If we want to find whether a number is stored in the tree. We first compare this number with the value stored at `root`. If the two values are the same, we know this value is stored in the tree. If the number is greater than the value stored at `root`, we know that it is impossible to find the value on the left side of tree. We can apply this property *recursively* until either (1) we find it somewhere in the tree or (2) we know the number is not stored in the tree. How can a binary search tree can be more efficient than a linked list? If the left side and the right side of `root` have the same number of `Nodes`, after only one comparison with the `root`'s value, we can discard half of the `Nodes` and choose only one side for the next comparison. The following shows how to implement `Tree_search`.

```

/*
    file: treeseach.c
*/
#include "tree.h"
TreeNode * Tree_search(TreeNode * n, int v)
{
    if (n == NULL)
    {
        /* cannot find */
        return NULL;
    }
}

```

```

    }
    if (v == (n -> value))
    {
        /* found */
        return n;
    }
    if (v < (n -> value))
    {
        /* search the left side */
        return Tree_search(n -> left, v);
    }
    return Tree_search(n -> right, v);
}

```

10.2.4 Print

`Tree_print` can be implemented in different ways. The function has three steps: (1) visiting the Node's left side, (2) visiting the Node's right side, and (3) printing the Node's value. There are $3! = 6$ ways to order these three steps but (1) usually precedes (2). Thus, we have only three ways to implement `Tree_print`:

- *pre-order*. The three steps are ordered as (3) - (1) - (2).
- *in-order*. The three steps are ordered as (1) - (3) - (2).
- *post-order*. The three steps are ordered as (1) - (2) - (3).

The `Tree_print` function calls all three orders to show the difference.

```

#include "tree.h"

void TreeNode_print(TreeNode * n)
{
    printf("%d ", n -> value);
}

static void Tree_printPreorder(TreeNode * n)
{
    if (n == NULL)
    {
        return;
    }
    TreeNode_print(n);
    Tree_printPreorder(n -> left);
    Tree_printPreorder(n -> right);
}

```

```

static void Tree_printInorder(TreeNode * n)
{
    if (n == NULL)
        {
            return;
        }
    Tree_printInorder(n -> left);
    TreeNode_print(n);
    Tree_printInorder(n -> right);
}

static void Tree_printPostorder(TreeNode * n)
{
    if (n == NULL)
        {
            return;
        }
    Tree_printPostorder(n -> left);
    Tree_printPostorder(n -> right);
    TreeNode_print(n);
}

void Tree_print(TreeNode * n)
{
    printf("\n\n====Preorder====\n");
    Tree_printPreorder(n);
    printf("\n\n====Inorder====\n");
    Tree_printInorder(n);
    printf("\n\n====Postorder====\n");
    Tree_printPostorder(n);
    printf("\n\n");
}

```

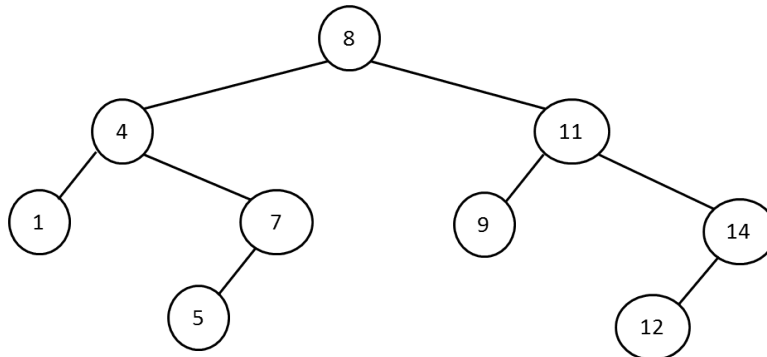
10.2.5 Delete

Deleting a node in a binary search tree is more complex because we have to maintain the tree's properties described on page 176. When deleting a node, we have to consider different scenarios:

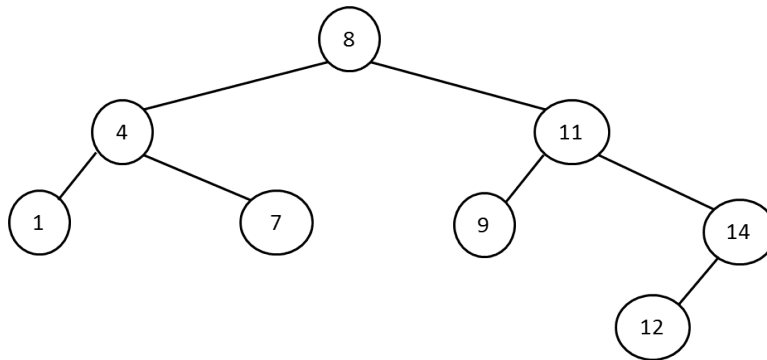
1. If this node has no child, release the memory occupied by this node. Figure 10.16 illustrates this scenario.
2. If this node has only one child, make this node's parent point to this node's child and release the memory occupied by this node. Figure 10.17 illustrates this scenario.

3. If this node has two children, find this node's successor. The successor is the node that appears immediately after this node in in-order traversal. The successor must be on the right side of this node. Exchange the values of this node and the successor. Delete the successor. Figure 10.18 illustrates this scenario. The successor must not have the left child; otherwise, it cannot be the successor.

The following figures show the three scenarios.



(a)



(b)

Figure 10.16: (a) The original binary search tree. (b) Deleting the node "5". This node has no child.

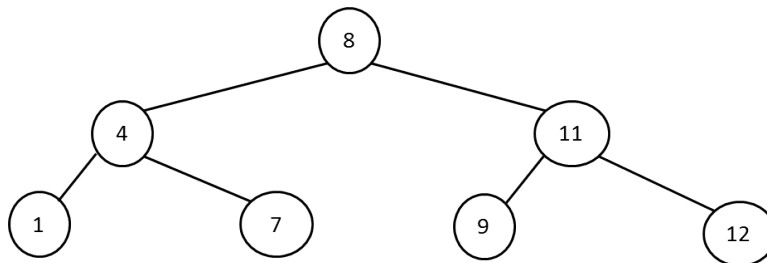


Figure 10.17: Deleting the node "14". This node has one child.

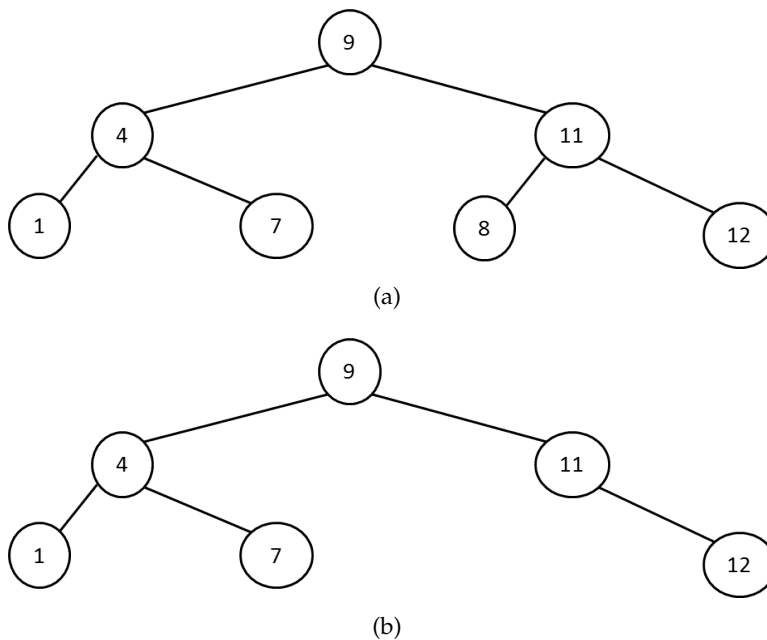


Figure 10.18: Deleting the node “8”. The node has two children. (a) Exchange the values of this node and its successor. (b) Deleting the successor.

The following code implements the delete function. Please notice that this function is recursive.

```

/*
 * file: treedelete.c
 */
#include "tree.h"
#include <stdlib.h>
TreeNode * Tree_delete(TreeNode * tn, int val)
{
    if (tn == NULL) { return NULL; }
    if (val < (tn -> value))
    {
        tn -> left = Tree_delete(tn -> left, val);
        return tn;
    }
    if (val > (tn -> value))
    {
        tn -> right = Tree_delete(tn -> right, val);
        return tn;
    }
    /* v is the same as tn -> value */
    if (((tn -> left) == NULL) && ((tn -> right) == NULL))
    {

```

```

        /* r has no child */
        free (tn);
        return NULL;
    }
    if ((tn -> left) == NULL)
    {
        /* tn -> right must not be NULL */
        TreeNode * rc = tn -> right;
        free (tn);
        return rc;
    }
    if ((tn -> right) == NULL)
    {
        /* tn -> left must not be NULL */
        TreeNode * lc = tn -> left;
        free (tn);
        return lc;
    }
    /* r have two children */
    /* find the immediate successor */
    TreeNode * su = tn -> right; /* su must not be NULL */
    while ((su -> left) != NULL)
    {
        su = su -> left;
    }
    /* su is r's immediate successor */
    /* swap their values */
    tn -> value = su -> value;
    su -> value = val;
    /* delete su */
    tn -> right = Tree_delete(tn -> right, val);
    return tn;
}

```

10.2.6 Iterator

In many cases, we want to go through every node in a binary tree. but we do not need to know the details. For example, we want to add the values stored in the tree. Since $a + b = b + a$, it does not matter which node is the first and which is the last. Another application counts the number of nodes whose values are greater than a given threshold. We need to visit every node once and only once but we do not need to know whether one particular node is the parent of another particular node. For this type of applications, an *iterator* can be helpful. Some languages, such as Perl, provide `foreach` as a way to invoke an iterator. C++ and Java have built-in iterators for some

classes (these are often called *container classes*). C does not provide `foreach` nor container classes; we have to implement iterators ourselves. An iterator needs to support at least these functions:

- *construct*: create an iterator.
- *hasNext*: return 1 if some elements have not been visited; return 0 if all elements have been visited.
- *visitNext*: go to the next element. This is usually called when *hasNext* return 1.
- *destruct*: release all memory occupied by an iterator. This should always be called after an iterator has been created.

A tree iterator contains an object and a linked list. The object has three pointers for the head, the current, and the tail of the linked list. Each node of the linked list has a pointer for one node in the binary tree. Figure 10.19 shows the iterator after calling `TreeIterator_construct`. There is only one node in the linked list and this node's `treenode` points to the tree's root, as shown by the dash arrow. The iterator's three attributes point to the only node in the linked list.

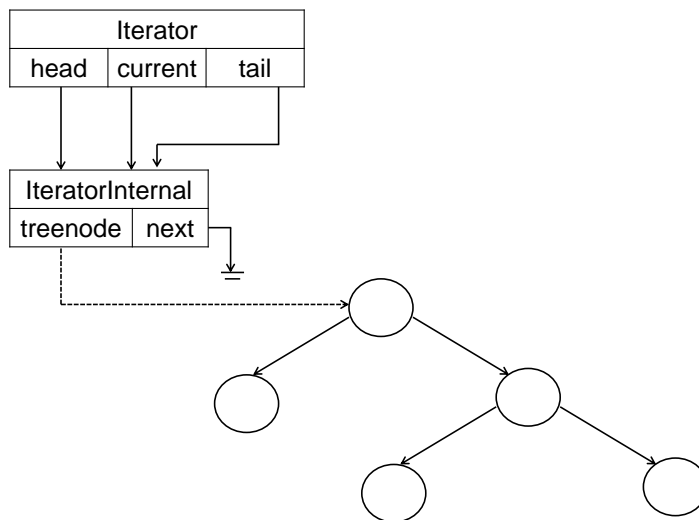


Figure 10.19: The iterator after calling `TreeIterator_construct`.

Calling `TreeIterator_visitNext` performs the following tasks:

- If the currently pointed binary tree node has one or both children, the child (or children) is inserted into the tail of the linked list.
- The iterator's `current` moves to the next node in the linked list. This is illustrated in Figure 10.20.

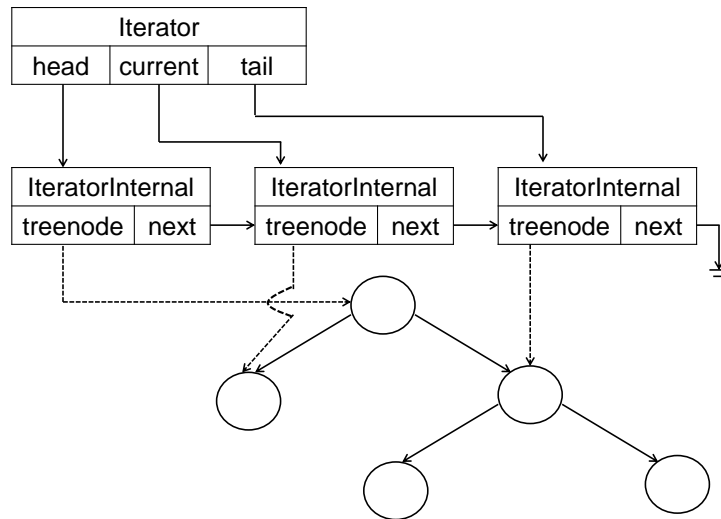


Figure 10.20: The linked has three nodes after calling `TreeIterator_visitNext` because the root's two children are also inserted.

Calling `TreeIterator_destruct` releases the memory occupied by the linked list and the iterator itself. If an iterator is created by calling `TreeIterator_construct`, the memory occupied by this iterator must be released by calling `TreeIterator_destruct`.

When `TreeIterator_visitNext` is called, at most two nodes (the child or children of the binary tree node pointed by `current`) are added to the end of the linked list. Is it possible to create the complete list pointing to every node in the binary tree when `TreeIterator_construct`? The answer is yes but it is undesirable. Why? Because in many cases, we do not have to visit every node in a binary tree. For example, if we want to know whether the tree stores a value greater than 381, we can stop after we find a node whose value is greater than 381. It is unnecessary to build the whole linked list at the beginning. If the binary tree contains many nodes, building the complete linked list takes too long.

The following shows how to implement an iterator for a binary tree.

```
#ifndef ITERATORINTERNAL_H
#define ITERATORINTERNAL_H
#include "tree.h"
/*
```

```

    * should be included by treeiterator.h only
    */
typedef struct iteratornode1
{
    struct iteratornode1 * next;
    TreeNode * treenode;
} IteratorInternal;
#endif

```

```

#ifndef TREEITERATOR_H
#define TREEITERATOR_H
#include "tree.h"
#include "iteratorinternal.h"
typedef struct iteratornode2
{
    IteratorInternal * head;
    IteratorInternal * current;
    IteratorInternal * tail;
} TreeIterator;

TreeIterator * TreeIterator_construct(TreeNode * root);
int TreeIterator_hasNext(TreeIterator * iterator);
TreeNode * TreeIterator_visitNext(TreeIterator * iterator);
void TreeIterator_destruct(TreeIterator * iterator);
#endif

```

```

/*
 * file: treeiterator.c
 */
#include "tree.h"
#include "treeiterator.h"
#include <stdlib.h>
#include <stdio.h>
static IteratorInternal * IteratorInternal_construct(TreeNode * p)
{
    IteratorInternal * internal = malloc(sizeof(IteratorInternal));
    internal -> treenode = p;
    internal -> next = NULL;
    return internal;
}

TreeIterator * TreeIterator_construct(TreeNode * p)

```

```

{
    TreeIterator * iter = malloc(sizeof(TreeIterator));
    IteratorInternal * internal = IteratorInternal_construct(p);
    iter -> head = internal;
    iter -> current = internal;
    iter -> tail = internal;
    return iter;
}

int TreeIterator_hasNext(TreeIterator * iter)
{
    return ((iter -> current) != NULL);
}

static void TreeIterator_insert(TreeIterator * iter)
{
    TreeNode * tailnode = (iter -> tail) -> treenode;
    if ((tailnode -> left) != NULL)
    {
        IteratorInternal * internal =
            IteratorInternal_construct(tailnode -> left);
        (iter -> tail) -> next = internal;
        iter -> tail = internal;
    }

    if ((tailnode -> right) != NULL)
    {
        IteratorInternal * internal =
            IteratorInternal_construct(tailnode -> right);
        (iter -> tail) -> next = internal;
        iter -> tail = internal;
    }
}

TreeNode * TreeIterator_visitNext(TreeIterator * iter)
{
    if (TreeIterator_hasNext(iter) == 0)
    {
        return NULL;
    }
    TreeNode * treenode = (iter -> current) -> treenode;
    TreeIterator_insert(iter);
    iter -> current = (iter -> current) -> next;
    return treenode;
}

```

```

void TreeIterator_destruct(TreeIterator * iterator)
{
    IteratorInternal * internal = iterator -> head;
    while (internal != NULL)
    {
        IteratorInternal * curr = internal;
        internal = internal -> next;
        free (curr);
    }
    free (iterator);
}

```

10.2.7 Destruct

The `Tree_destruct` function destroys the whole tree by releasing the memory occupied by all Nodes.

```

#include "tree.h"
#include <stdlib.h>
void Tree_destruct(TreeNode * n)
{
    if (n == NULL)
    {
        return;
    }
    Tree_destruct(n -> left);
    Tree_destruct(n -> right);
    free(n);
}

```

Notice that both `left` and `right` must be destroyed before this Node's memory is released.

10.2.8 main

Finally, this shows the main function:

```

#include "tree.h"
#include "treeiterator.h"
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
void visitTree(TreeNode * root)

```



```

{
    printf("-----Iterator-----");
    TreeIterator * iterator = TreeIterator_construct(root);
    while (TreeIterator_hasNext(iterator))
    {
        TreeNode * n = TreeIterator_visitNext(iterator);
        TreeNode_print(n);
    }
    printf("\n\n");
    TreeIterator_destruct(iterator);
}
int main(int argc, char * argv[])
{
    TreeNode * root = NULL;
    int num = 0;
    int iter;
    unsigned int seed = time(NULL);
    seed = 0;
    srand(seed);
    if (argc >= 2)
    {
        num = (int) strtol(argv[1], NULL, 10);
    }
    if (num < 8)
    {
        num = 8;
    }
    int * array = malloc(sizeof(int) * num);
    for (iter = 0; iter < num; iter++)
    {
        array[iter] = rand() % 10000;
    }
    for (iter = 0; iter < num; iter++)
    {
        int val = array[iter];
        printf("insert %d\n", val);
        root = Tree_insert(root, val);
        Tree_print(root);
    }
    visitTree(root);
    for (iter = 0; iter < num; iter++)
    {
        int index = rand() % (2 * num);
        if (index < num)
        {

```

```

        int val = array[index];
        printf("delete %d\n", val);
        root = Tree_delete(root, val);
        Tree_print(root);
    }
}
Tree_destruct(root);
free (array);
return EXIT_SUCCESS;
}

```

10.2.9 Makefile

The program for the binary tree now has quite a few `.h` and `.c` files for implementing the different functions. We will explain new techniques that can simplify `Makefile`. If you have forgotten `Makefile`, please review Section 9.3. In Section 9.3, `Makefile` lists the object files as well as the corresponding `.c` and `.h` files. For example,

```

student.o: student.c student3.h
    $(GCC) -c student.c

```

This is acceptable if there are only a few files. When the number of files increases, listing all object files and the dependent header files becomes tedious. One solution uses `Makefile`'s symbolic conversion. We list only the `.c` files and tell `make` to automatically find the corresponding object files. The following lines can do the work:

```

SRCS = treemain.c treesearch.c treedestruct.c treeinsert.c \
    treeprint.c treedelete.c treeiterator.c
OBJS = $(SRCS:%.c=%.o)

```

The lines tell `make` that there are seven source `.c` files. The second line tells `make` that each `.c` file should have a corresponding object `.o` files.

The following two lines tell `make` how to generate an object file from the corresponding source file.

```

.c.o:
    $(GCC) -c $*.c

```

This rule, however, does not specify the dependence of header files. There is another command, called `makedepend`, to do this job. To use this command, simply put the following two lines in `Makefile`.

```
headers:
    makedepend $(SRCS)
```

The command `makedepend` will check which header files are included in the source files and write the dependences at the bottom of `Makefile`. When we type

```
make headers
```

the `makedepend` command generates something like the following:

```
# DO NOT DELETE

treemain.o: tree.h /usr/include/stdio.h /usr/include/features.h
treemain.o: /usr/include/bits/predefs.h /usr/include/sys/cdefs.h
treemain.o: /usr/include/bits/wordsize.h /usr/include/gnu/stubs.h
treemain.o: /usr/include/gnu/stubs-64.h /usr/include/bits/types.h
treemain.o: /usr/include/bits/typesizes.h /usr/include/libio.h
```

This means `treemain.o` depends on these `.h` files. If any of these `.h` files changes, `treemain.c` should be recompiled.

To summarize, this section explains two techniques simplifying `Makefile`. We can list source files and use symbols to automatically generate the list of object files. We can use `makedepend` to generate the list of header files; changing any of these header files triggers recompilation.

The following is our new `Makefile` for binary search trees.

```
CFLAGS = -g -Wall -Wshadow
GCC = gcc $(CFLAGS)
SRCS = treemain.c treesearch.c treedestruct.c treeinsert.c treeprint.c \
      treedelete.c treeiterator.c
OBJS = $(SRCS:%.c=%.o)

tree: $(OBJS)
    $(GCC) $(OBJS) -o tree

memory: tree
    valgrind --leak-check=yes --verbose ./tree 10

.c.o:
    $(GCC) $(CFLAGS) -c $*.c

clean:
    rm -f *.o a.out tree
```

```

headers:
    makedepend $(SRCS)

# DO NOT DELETE

treemain.o: tree.h /usr/include/stdio.h /usr/include/features.h
treemain.o: /usr/include/libio.h /usr/include/_G_config.h
treemain.o: /usr/include/wchar.h /usr/include/xlocale.h treeiterator.h
treemain.o: iteratorinternal.h /usr/include/time.h /usr/include/stdlib.h
treemain.o: /usr/include/alloca.h
treesearch.o: tree.h /usr/include/stdio.h /usr/include/features.h
treesearch.o: /usr/include/libio.h /usr/include/_G_config.h
treesearch.o: /usr/include/wchar.h /usr/include/xlocale.h
treedestruct.o: tree.h /usr/include/stdio.h /usr/include/features.h
treedestruct.o: /usr/include/libio.h /usr/include/_G_config.h
treedestruct.o: /usr/include/wchar.h /usr/include/xlocale.h
treedestruct.o: /usr/include/stdlib.h /usr/include/alloca.h
treeinsert.o: tree.h /usr/include/stdio.h /usr/include/features.h
treeinsert.o: /usr/include/libio.h /usr/include/_G_config.h
treeinsert.o: /usr/include/wchar.h /usr/include/xlocale.h
treeinsert.o: /usr/include/stdlib.h /usr/include/alloca.h
treeprint.o: tree.h /usr/include/stdio.h /usr/include/features.h
treeprint.o: /usr/include/libio.h /usr/include/_G_config.h
treeprint.o: /usr/include/wchar.h /usr/include/xlocale.h
treedelete.o: tree.h /usr/include/stdio.h /usr/include/features.h
treedelete.o: /usr/include/libio.h /usr/include/_G_config.h
treedelete.o: /usr/include/wchar.h /usr/include/xlocale.h
treedelete.o: /usr/include/stdlib.h /usr/include/alloca.h
treeiterator.o: tree.h /usr/include/stdio.h /usr/include/features.h
treeiterator.o: /usr/include/libio.h /usr/include/_G_config.h
treeiterator.o: /usr/include/wchar.h /usr/include/xlocale.h treeiterator.h
treeiterator.o: iteratorinternal.h /usr/include/stdlib.h
treeiterator.o: /usr/include/alloca.h

```

10.3 Practice Problems

1. Modify the linked list program so that the most recently added Node object is at the end of the list. In other words, the new Node is the farthest from head.
2. Modify the linked list program so that the values stored in the list are unique. In other words, if p and q are two different Nodes, $p \rightarrow \text{value}$ must be different from $q \rightarrow \text{value}$.
3. Modify the linked list program so that the values are sorted. In other words, if p is before q , $p \rightarrow \text{value}$ must be smaller than $q \rightarrow \text{value}$.
4. Suppose a linked has n Nodes and the values are not sorted. How many Nodes will `List_search` check if v is not stored in the list?

5. Draw a binary search tree that has a special property: the outputs are the same for pre-order, in-order, and post-order. Please remember that in a binary search tree, the nodes store distinct values.

Chapter 11

More Practice Problems

11.1 Anagrams

11.1.1 Distinct Letters

Write a program that can print the anagrams of a word. An anagram of a word is a word that uses exactly all of the letters of the original word, but in a different order. For example, the anagrams of the word "the" are "teh", "eth", "eht", "het", and "hte". You can assume that the letters in the word are distinct. For example, do not worry about a word like "color" because the letter 'o' appears twice.

11.1.2 Unique Words

Modify the program to handle repeated letters: One or more letters in the word may appear multiple times. For example, the letter 'o' appears twice in the word "color". Your program must print the same anagram only once.

11.2 Integer Partition

11.2.1 All Partitions

Write a program to print all partitions of a positive integer. For example, if the input is 3, the program prints

```
[1 1 1]
[1 2]
[2 1]
[3]
```

If the input is 4, the program prints

```
[1 1 1 1]
[1 1 2]
[1 2 1]
[1 3]
[2 1 1]
[2 2]
[3 1]
[4]
```

If the input is 5, the program prints

```
[1 1 1 1 1]
[1 1 1 2]
[1 1 2 1]
[1 1 3]
[1 2 1 1]
[1 2 2]
[1 3 1]
[1 4]
[2 1 1 1]
[2 1 2]
[2 2 1]
[2 3]
[3 1 1]
[3 2]
[4 1]
[5]
```

11.2.2 Distinct Numbers

Modify the program so that the numbers are distinct. For example, if the input is 5,

```
[1 4]
[2 3]
[4 1]
[3 2]
```

are valid partitions, but

```
[2 2 1]
[1 1 3]
```

are invalid partitions.

11.2.3 Increasing Numbers

Modify the program so that the numbers are increasing. If the input is 5,

```
[1 4]
[2 3]
```

are valid partitions, but

```
[4 1]
[3 2]
```

are invalid partitions.

11.2.4 Only 1-5 Can Be Used

Modify the program so that only 1, 2, 3, 4, and 5 are used for any value. For example, if the value is 16,

```
[1 4 3 4 4]
[2 3 5 4 2]
```

are valid partitions, but

```
[4 1 6 5]
[3 2 11]
```

are invalid partitions.

11.3 Arrange Three Balls

You are given unlimited numbers of Yellow, Red, and Green balls. Y: Yellow ball. R: Red ball. G: Green ball. You want to determine the number of options to select the colors under one rule: two green balls cannot be next to each other. How many options do you have if you want to select n balls?

If n is 1, there are 3 options:

Y
R
G

If n is 2, there are three 8 options ($3 \times 3 - 1$ because GG is not allowed)

YY
YR
YG
RY
RR
RG
GY
GR

11.3.1 List All Options

Write a program that can generate all options for a given value of n .

11.3.2 Count without Listing All Options

Derive a mathematical formula to determine the number of options without listing all options.

11.4 Encryption

This problem requires three programs.

First, write a program to convert a text file into a file of binary digits (i.e. 0 or 1) using the letters' ASCII values. Each letter uses 8 digits. For example, In the ASCII table, 'A' is 65 so its

binary representation is 01000001. For 'n', the ASCII value is 110 and the binary representation is 01101110. The program can also convert a file of binary digits to English letters.

Second, write a program to generate an encryption key. The key is a sequence of random 0 or 1. The program should get one input specifying the number of digits in the key.

Third, write a program to encrypt a binary data file (generated by the first program) using the key (generated by the second program). Encryption is bit-wise exclusive or (XOR) defined as follows

```
0 XOR 0 is 0
0 XOR 1 is 1
1 XOR 0 is 1
1 XOR 1 is 0
```

This is an example:

```
input  00100011010101
key    11000101101101
XOR
result 11100110101000
```

In C, the XOR operator is \wedge . If someone sees the encrypted data without the key, it is difficult for the person to guess the original text. If the data file has fewer bits than the number of bits of the key, the unused key bits are discarded. If the data file has more bits than the number of bits of the key, the bits in the key are reused from the beginning. For example,

```
input  00100011010101
key    110001
```

The key repeats so that it becomes long enough

```
input  00100011010101
key    11000111000111
      |      |
      repeat repeat
```

The encryption program can also be used for decryption. If a person has the key and the encrypted data, the person can restore the original data by applying XOR with the key again.

Part II

Programming Tools

Chapter 12

Debugger `gdb` and `ddd`

A debugger can be very helpful finding problems in your programs. In fact, You can use a debugger for understanding a program's behavior even though your program is working. Compared with adding "debugging messages" printed on screen, a debugger has several advantages:

- You do not have to change your program. This makes using a debugger a very attractive option when you do not really know what is wrong with your program and what should be printed.
- A debugger can show you the call stack and allow you to move among the frames. This can help you understand the sequence of function calls and how the program reaches the current location. Moreover, a debugger can show the values of arguments.
- If your program crashes (i.e. stops abnormally), a debugger can tell you what code is executed just before the program stops. This is much better than printing messages; programs often stop at unexpected places. As a result, it is common that you add a lot of debugging messages and they are irrelevant to the problem.
- A debugger allows you to set breakpoints. The program will stop when it reaches a breakpoint. You can even set a conditional breakpoint so that the program stops only if the condition is met, for example, when a variable is smaller than 0.

Even though a debugger has many advantages, it also has disadvantages:

- Most debuggers are designed for *interactive debugging*. You have to interact with the debuggers. This can require a lot of efforts if the problems do not appear early during program execution. You may need to wait for a long time before the program reaches the place where problems occur. For this reason, printing debugging messages (called *logging*) can still be useful, especially for post-execution analysis.

- When a program is compiled with debugging information (by adding `-g` after `gcc`), the program is slower. This can further increase the amount of time taken to reach the place of problems.
- If a program interacts with another system, stopping or slowing the program's execution by a debugger can cause the other system to behave unexpectedly. If you are writing a program talking to another machine through a network, stopping your program can make that machine believe that the network is broken and close the network connection with your program.

Despite the disadvantages, a debugger is still a good tool, if you know how to use it and its limitations. You can decide whether it is the right tool.

12.1 Trace a Program

To use `gdb`, you need to add `-g` after `gcc`. This tells `gcc` to add information mapping the executable file back to source code. Let's consider the program `args.c` on page 4. The following commands create the executable file called `args` and call `gdb` with the executable as the command-line argument.

```
> gcc -g -Wall -Wshadow args.c -o args
> gdb args
```

Please notice that `gdb` takes the name of an executable file, not a `.c` file. If a program contains multiple `.c` files, `gdb` knows how to map the executable to the correct source files.

After starting `gdb`, it shows some messages like the following

```
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
...
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from ...
(gdb)
```

Here, `(gdb)` is the command prompt. Our first command is to set a breakpoint at the beginning of the `main` function.

```
(gdb) b main
Breakpoint 1 at 0x400573: file args.c, line 9.
```

What is 0x400573? It is the location of the statement. Section 2.3 explains that every statement has a location. This value is the location corresponding to the first statement in the `main` function.

Next, we start the program using the `r` command. It means “run.” We can add arguments after `r`.

```
(gdb) r 1 2 3 hello C programming this is gdb
Starting program: args 1 2 3 hello C programming this is gdb
```

```
Breakpoint 1, main (argc=10, argv=0x7fffe778) at args.c:9
9  printf("There are %d arguments.\n", argc);
```

The program stops at the first line of the function (skipping the declarations of the variable `cnt`); `gdb` prints the next line to execute. As we can see, the value of `argc` is 10. What is 0x7fffe778 for `argv`? Remember that it is a pointer and it stores the address of the string array. The command `n` means “next”: the program executes the next statement.

```
(gdb) n
There are 10 arguments.
10  printf("The arguments are:\n");
```

The program prints `There are 10 arguments`. We can also use the `print` command in `gdb` to show the value of `argc`:

```
(gdb) print argc
$1 = 10
```

After each `n` command, `gdb` prints the next statement. If you want to see a few more lines before and after the next statement, you can use the `list` command.

```
(gdb) list
5 #include <stdlib.h>
6 int main(int argc, char * argv[])
7 {
8     int cnt;
9     printf("There are %d arguments.\n", argc);
10    printf("The arguments are:\n");
11    for (cnt = 0; cnt < argc; cnt++)
12        {
13            printf("argv[%d] is %s\n", cnt, argv[cnt]);
14        }
```


Type `n` twice and the program enters the `for` block.

```
(gdb) n
The arguments are:
11   for (cnt = 0; cnt < argc; cnt ++)
```

```
(gdb) n
13     printf("argv[%d] is %s\n", cnt, argv[cnt]);
```

We can use the `print` command again to print the value of `cnt`.

```
(gdb) print cnt
$2 = 0
```

Let's type `n` three more times and print the value of `cnt`.

```
(gdb) n
argv[0] is args
11   for (cnt = 0; cnt < argc; cnt ++)
```

```
(gdb) n
13     printf("argv[%d] is %s\n", cnt, argv[cnt]);
```

```
(gdb) n
argv[1] is 1
11   for (cnt = 0; cnt < argc; cnt ++)
```

```
(gdb) print cnt
$3 = 1
```

If you do not want to type `print` so many times, you can use the `display` command. The command will keep printing the value.

```
(gdb) display cnt
1: cnt = 1
```

```
(gdb) n
13     printf("argv[%d] is %s\n", cnt, argv[cnt]);
1: cnt = 2
```

```
(gdb) n
argv[2] is 2
11   for (cnt = 0; cnt < argc; cnt ++)
```

```
1: cnt = 2
```

```
(gdb) n
13     printf("argv[%d] is %s\n", cnt, argv[cnt]);
1: cnt = 3
```

As you can see, `cnt`'s value is printed every time we enter the `n` command.

We do not want to type `n` many times. Instead, we set a breakpoint at line 13 when `cnt` is larger than 7.

```
(gdb) b 13 if (cnt > 7)
Breakpoint 2 at 0x40059d: file args.c, line 13.
```

We can continue running the program by using the `c` command; the program will stop when `cnt` is larger than 7.

```
(gdb) c
Continuing.
argv[3] is 3
argv[4] is hello
argv[5] is C
argv[6] is programming
argv[7] is this

Breakpoint 2, main (argc=10, argv=0x7fffe778) at args.c:13
13     printf("argv[%d] is %s\n", cnt, argv[cnt]);
1: cnt = 8
```

We can use the `info b` command to show breakpoints.

```
(gdb) info b
Num      Type           Disp Enb Address                What
1        breakpoint      keep y  0x000000400573 in main
                                                at args.c:9
breakpoint already hit 1 time
2        breakpoint      keep y  0x00000040059d in main
                                                at args.c:13
stop only if (cnt > 7)
breakpoint already hit 1 time
```

You can delete one breakpoint using the `delete` command followed by a number. For example, `delete 2` deletes the second breakpoint.

```
(gdb) delete 2
(gdb) info b
Num      Type           Disp Enb Address                What
1        breakpoint      keep y  0x000000400573 in main
                                                at args.c:9
breakpoint already hit 1 time
```

The `delete` command, without any number, deletes all breakpoints.

```
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) info b
No breakpoints or watchpoints.
```

Since there is no breakpoint now, the `c` command will continue running the program to completion.

```
(gdb) c
Continuing.
argv[8] is is
argv[9] is gdb
```

Program exited normally.

To stop `gdb`, type the `quit` command.

The following table shows the commands we have used so far.

Command	Meaning
<code>b function name</code>	set a breakpoint at the beginning of the function.
<code>b line number</code>	set a breakpoint at this line number.
<code>b line number if condition</code>	set a breakpoint at this line number, stop only when the condition is met.
<code>c</code>	continue running the program until it reaches the next breakpoint, or until the program completes.
<code>delete</code>	delete all breakpoints.
<code>delete number</code>	delete a specific breakpoint.
<code>display variable</code>	display the value of this variable after every <code>n</code> command.
<code>info b</code>	show breakpoints
<code>list</code>	list the lines before and after the next statement.
<code>n</code>	execute the next statement.
<code>print variable</code>	print the value of this variable.
<code>r arguments</code>	run the program with the command-line arguments.

12.2 Show Call Stack

This section shows the call stack of `balls.c` on page 46. We will execute the program with `n` is 5.

```

> gcc balls.c -o balls
> gdb balls
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
...
(gdb) b main
Breakpoint 1 at 0x400637: file balls.c, line 37.
(gdb) r 5
Starting program: balls 5
Breakpoint 1, main (argc=2, argv=0x7fffe7d8) at balls.c:37
37  if (argc < 2)
(gdb)

```

We can use the `n` command to go to the next statement. Since `argc` is 2, the `if` condition is false.

```

(gdb) n
42  n = (int) strtol(argv[1], NULL, 10);

```

Go to the next statement by typing `n` again.

```

(gdb) n
43  c = f(n);

```

We want to see what is happening *inside* the `f` function. Therefore, we are going to use the `s` command to step into the called function. The `bt` (backtrace) function displays the call stack. The call stack currently has two frames for `f` and `main`. After #1 is the return location after `f` finishes.

```

(gdb) s
f (m=5) at balls.c:11
11  if (m <= 0)
(gdb) bt
#0  f (m=5) at balls.c:11
#1  0x000000400678 in main (argc=2, argv=0x7fffe7d8) at balls.c:43

```

Type `n` three times and the program reaches the point calling `f` again.

```

(gdb) n
16  if (m == 1)
(gdb) n
21  if (m == 2)
(gdb) n
26  a = f(m - 1);

```

Type `s` again to enter the called function. Now, `m` is 4.

```
(gdb) s
f (m=4) at balls.c:11
11  if (m <= 0)
```

We can continue this procedure and see `m` become 3.

```
(gdb) n
16  if (m == 1)
(gdb) n
21  if (m == 2)
(gdb) n
26  a = f(m - 1);
(gdb) s
f (m=3) at balls.c:11
11  if (m <= 0)
```

Type `bt` to see the call stack.

```
(gdb) bt
#0  f (m=3) at balls.c:11
#1  0x00000040060a in f (m=4) at balls.c:26
#2  0x00000040060a in f (m=5) at balls.c:26
#3  0x000000400678 in main (argc=2, argv=0x7fffe7d8) at balls.c:43
```

You can see clearly the return locations and `m`'s value in each frame. We can see the value by using the `print` command. Currently, its value is 3. We can also print `m`'s address by adding `&` in front.

```
(gdb) print m
$1 = 3
(gdb) print &m
$2 = (int *) 0x7fffe64c
```

We can use the `f` (frame) command go to another frame. Let's go to frame #2 and print `m`'s value as well as its address.

```
(gdb) f 2
#2  0x00000040060a in f (m=5) at balls.c:26
26  a = f(m - 1);
(gdb) print m
$3 = 5
(gdb) print & m
$4 = (int *) 0x7fffe6ac
```

As you can see, this `m` is stored at an address different from the `m` in frame #0.

You can continue using these commands to see how the call stack changes and the arguments' values, as well as the values of the local variables.

Command	Meaning
<code>bt</code>	Backtrace, show the call stack.
<code>f number</code>	go to another frame.
<code>s</code>	Step into a function.

12.3 Locate Segmentation Fault

If a C program, `segmentation fault` means a program intends to access (read from or write to) an invalid memory location. The following example will cause segmentation fault during execution.

```
/*
  file: segfault1.c
  purpose: an example causing segmentation fault
*/
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
  int * arr;
  arr[2] = 10;
  return EXIT_SUCCESS;
}
```

```
> gcc -Wall -Wshadow segfault1.c -o seg
segfault1.c: In function f1:
segfault1.c:6: warning: arr is used uninitialized in this function
> ./seg
Segmentation fault
```

The program causes Segmentation fault. Which line causes the fault? We can use gdb to find the answer:

```
> gdb seg
GNU gdb (GDB) 7.1-ubuntu
(gdb) r
Starting program: seg

Program received signal SIGSEGV, Segmentation fault.
0x0000004004d7 in main (argc=1, argv=0x7fffe7d8)
    at segfault1.c:6
6   arr[2] = 10;
(gdb) bt
#0  0x0000004004d7 in main (argc=1, argv=0x7fffe7d8)
    at segfault1.c:6
```

The program fails at line 6. This is the line where gcc gives a warning message. From this example, we know gcc warnings should be treated seriously because warnings often indicate errors. The program fails because it wants to modify an element of the array but no memory has been allocated yet.

The previous program *always* causes segmentation fault. This is actually good for a programmer: the programmer knows something is wrong and needs to be fixed. Sometimes, a program has problems but does not crash. The programmer may think the program is working. The following is an example. When I execute the program, it does not always crash.

```
/*
  file: segfault2.c
  purpose: another example causing segmentation fault (through
  function calls with arguments)
*/
#include <stdio.h>
#include <stdlib.h>
void f1(int a, int b)
{
    int * arr;
    arr[a] = b;
}
void f2(int a)
{
    if (a > 5)
    {
        return;
    }
    else
```

```

    {
        f1(a + 3, a - 7);
    }
}int main(int argc, char * argv[])
{
    f2(1);
    return EXIT_SUCCESS;
}

```

However, we do know that the program has the same problem: it does not allocate memory for the array. How can we fix a problem if we do know the existence of the problem? We can use `valgrind` mentioned in Section 5.8.

```

> valgrind --leak-check=yes ./seg
==2731== Memcheck, a memory error detector
==2731== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==2731== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h
    for copyright info
==2731== Command: ./seg
==2731==
==2731== Use of uninitialised value of size 8
==2731==    at 0x4004DE: f1 (segfault2.c:11)
==2731==    by 0x400507: f2 (segfault2.c:21)
==2731==    by 0x400525: main (segfault2.c:25)
==2731==
==2731== Invalid write of size 4
==2731==    at 0x4004DE: f1 (segfault2.c:11)
==2731==    by 0x400507: f2 (segfault2.c:21)
==2731==    by 0x400525: main (segfault2.c:25)
==2731== Address 0x10 is not stack'd, malloc'd or (recently) free'd
==2731==
==2731==
==2731== Process terminating with default action of signal 11 (SIGSEGV)
==2731== Access not within mapped region at address 0x10
==2731==    at 0x4004DE: f1 (segfault2.c:11)
==2731==    by 0x400507: f2 (segfault2.c:21)
==2731==    by 0x400525: main (segfault2.c:25)
==2731== If you believe this happened as a result of a stack
==2731== overflow in your program's main thread (unlikely but
==2731== possible), you can try to increase the size of the
==2731== main thread stack using the --main-stacksize= flag.
==2731== The main thread stack size used in this run was 8388608.
==2731==
==2731== HEAP SUMMARY:
==2731==    in use at exit: 0 bytes in 0 blocks

```



```
==2731==    total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==2731==
==2731== All heap blocks were freed -- no leaks are possible
==2731==
==2731== For counts of detected and suppressed errors, rerun with: -v
==2731== Use --track-origins=yes to see where uninitialised values come from
==2731== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 4 from 4)
Segmentation fault
```

The first line is the command we type at Terminal and the rest is the report. As you can see, `valgrind` detects something wrong at line 11:

```
arr[a] = b;
```

The report shows the call sequence:

```
==2731==    at 0x4004DE: f1 (segfault2.c:11)
==2731==    by 0x400507: f2 (segfault2.c:21)
==2731==    by 0x400525: main (segfault2.c:25)
```

At line 25 of `segfault2.c`, `main` calls `f2`; at line 21 `f2` calls `f1`. It is a good habit to use `valgrind` to test your program even though it seems working.

12.4 Print Object

GDB can print the attributes of an object. We consider the `Vector` example on page 108. The following are the GDB commands we use to show the attributes:

```
> gcc -g vector.c -o vec
> gdb vec
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
(gdb) b main
Breakpoint 1 at 0x400534: file vector.c, line 17.
(gdb) r
Breakpoint 1, main (argc=1, argv=0x7fffe7d8) at vector.c:17
17  v1.x = 3;
(gdb) n
18  v1.y = 6;
(gdb)
```

```
19  v1.z = -2;
(gdb)
20  printf("The vector is (%d, %d, %d).\n", v1.x, v1.y, v1.z);
(gdb) print v1
$1 = {x = 3, y = 6, z = -2}
```

The last command `print v1` can show all attributes of a `Vector` object.

12.5 DDD

If you prefer to use a graphical user interface for debugging, DDD (data display debugger) is a good choice. DDD provides a user interface for GDB so all GDB commands are still valid. Figure 12.1 shows the start-up window of DDD.

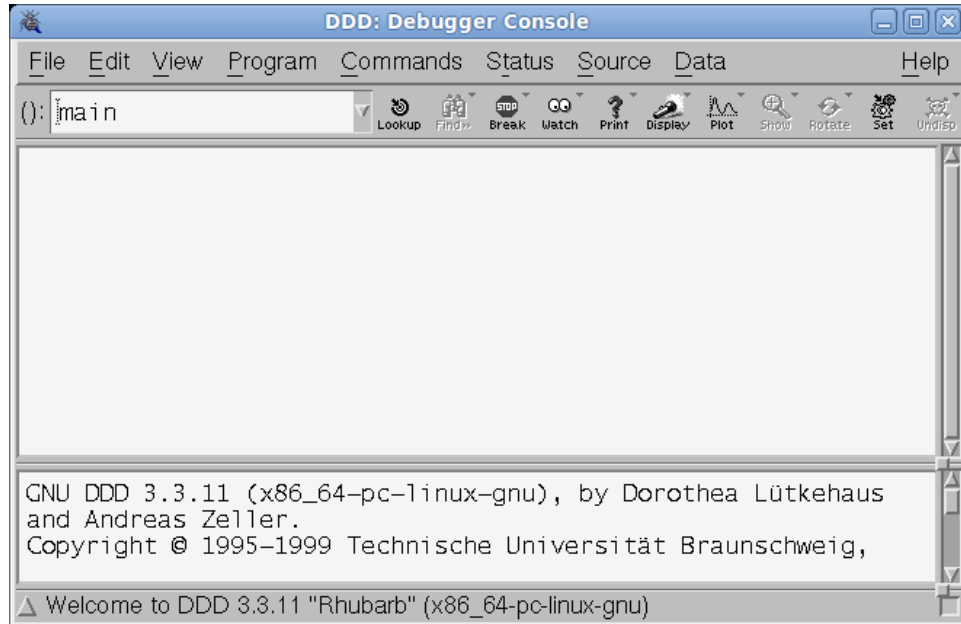


Figure 12.1: Start-up window of DDD.

Select File - Open Program and find the program you want to debug. This is the executable, not the source file. Figure 12.2 shows the window after opening the `vector.c` program.

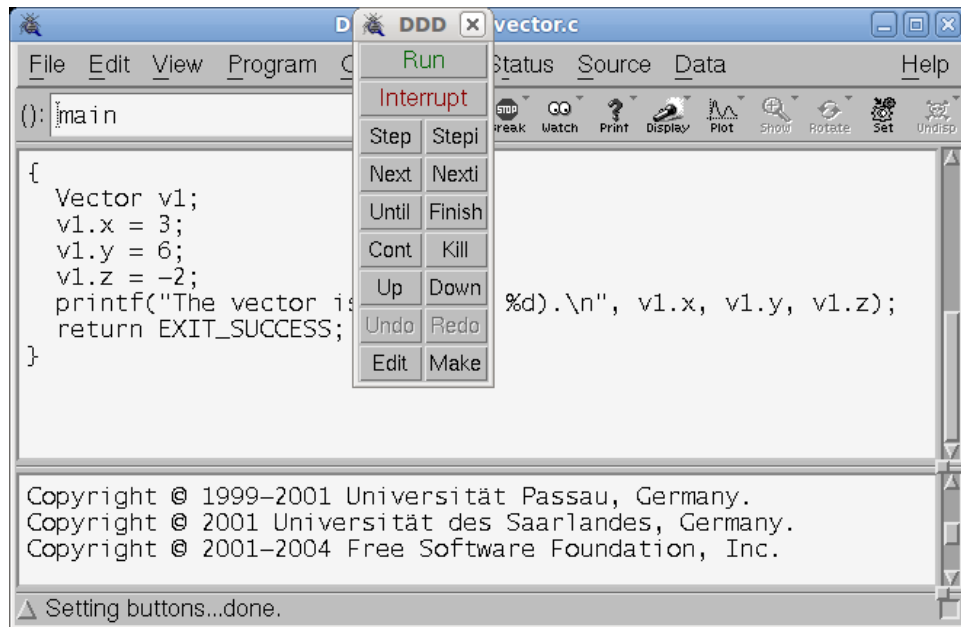


Figure 12.2: DDD opens the `vector.c` program.

Move the mouse cursor next to the line with `printf`. Click the right button and select Set Breakpoint. As you can see in Figure 12.3, a STOP symbol is added in front of the line.

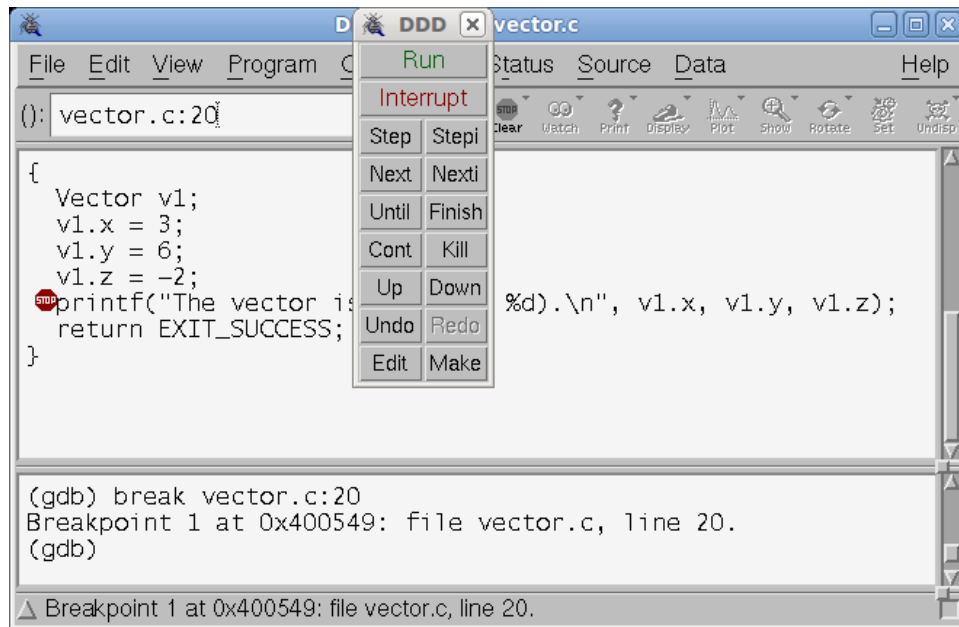


Figure 12.3: A breakpoint is set. A STOP symbol is shown at the line.

Click Run and the program stops at the breakpoint. An arrow points to the breakpoint.

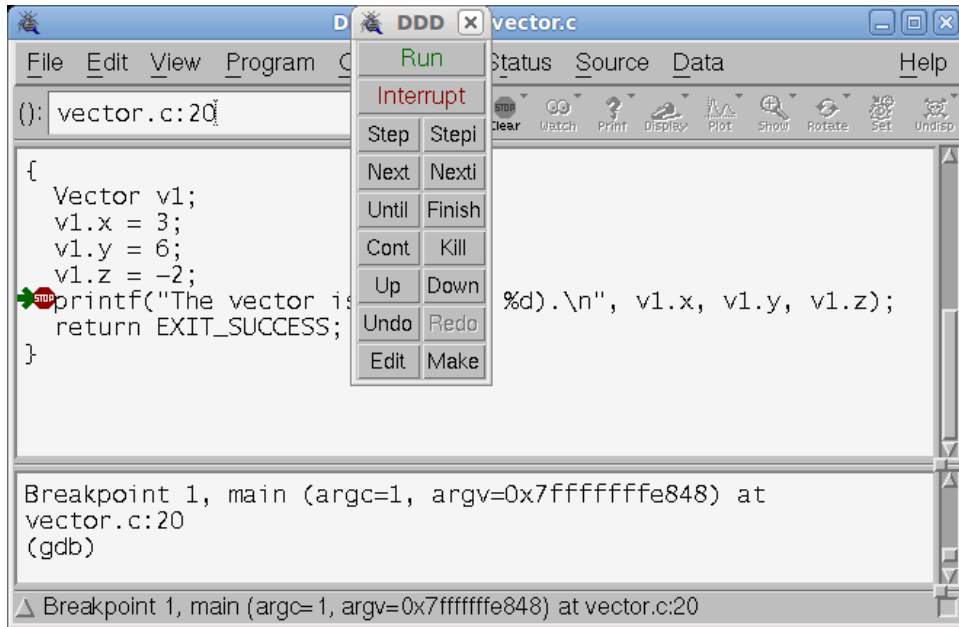


Figure 12.4: The program stops at the breakpoint.

Use the mouse to highlight `v1` and select `Display v1`. Figure 12.5 shows the attributes of `v1`.

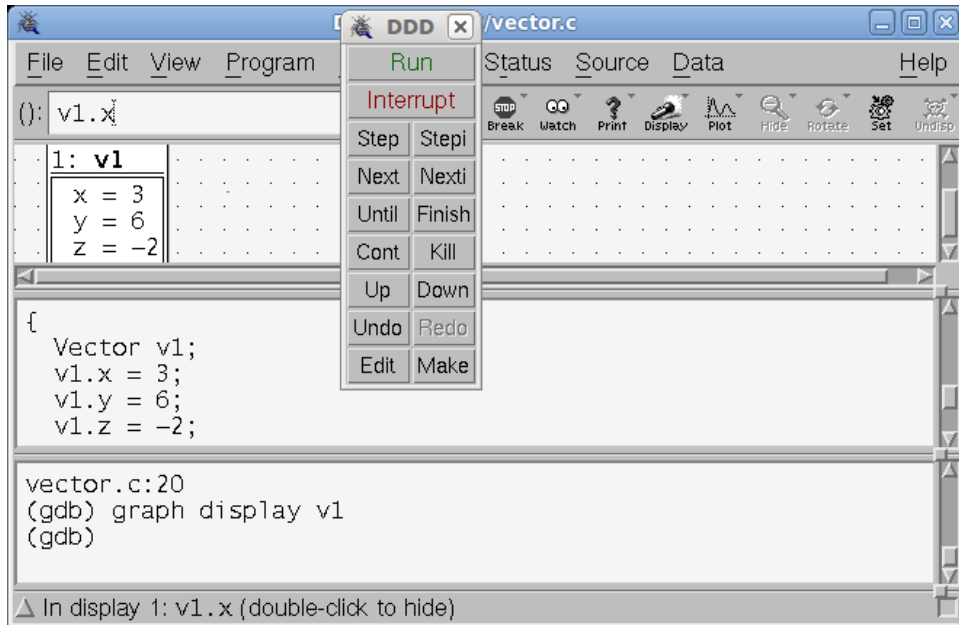


Figure 12.5: DDD shows the attributes of `v1`.

This is a brief introduction of DDD. DDD has many other functions, you should spend a few minutes exploring the functions.

12.6 Prevent Bugs

The best method to debug a program is to prevent bugs from happening. Here are some suggestions about how to prevent bugs in your programs.

- *Design before coding.* When you write a program to solve a complex problem, you have to develop a solution before writing code. A design has two parts: algorithms and program organization. This book focuses more on the latter part. Which functions do you need? What are their arguments? If you have a good design, coding is much easier. The sooner you start coding, the longer it takes to finish the program because your program will be “messy”.
- *Assume your programs will change many times.* A common mistake among students is to assume that programs will be thrown away quickly. As a result, the students write “dirty” code—disorganized and hard to understand. Their attitude is to write programs for assignments and do not want to see their own programs again. Unfortunately, this approach usually hurts them *before* submissions. They will quickly discover that their programs are too difficult to debug.

Why will you change your programs many times? Obviously, if your programs have problems (i.e. bugs), you have to change the programs. You change programs for many other reasons: You misunderstand the requirements. You need better algorithms. Your competitors add new features and you have to provide the same or better features to keep customers. Non-trivial programs—programs that are worth your effort writing—will probably be rewritten many times. Why? Because technologies are changing. Because the world is changing. Your customers are changing. Your competitors are changing. You need to design your programs so that they can be changed more easily.

If you assume that your programs will change many times, you will be more careful thinking about the structure. Your programs will be more understandable. Some people believe that a well-designed program needs to be written only once. I think this is too ideal and impractical. Believing a well-designed program will not change is like believing a well-designed airplane will never crash. Airplanes should be designed carefully but there are many protections (such as oxygen masks and evacuation slides) in case problems do arise. Similarly, a well-designed program will change. The question is whether you can make changes easy.

- *Take one step each time.* Write a small piece of code, make sure it works before you do something else. This usually means that you have to write additional code to test that small piece of code. Often, you have to create test cases for each small piece of code. Sometimes, you create the test cases manually; sometimes, you write “test case generators”.
- *Make every line solid.* Many people write messy code because they will “clean up” later. Unfortunately, they never have time to clean up. Why? Their messy code is so buggy that they are too busy debugging. If they write “clean” code, they don’t have to clean up and they probably don’t have many bugs.

- *Initialize variables and pointers.* This is a common problem. A program's behavior is undefined if there are uninitialized variables or pointers. Unfortunately, this type of bugs is hard to detect by testing. You should assign *invalid* values if you do not know what values to use for initialization. For example, initialize pointers to `NULL`. Why? This guarantees the programs to fail if the values are not corrected later. Will initialization make a program slower? Yes, by a fraction of a nanosecond. How long does it take you to debug a program with uninitialized variables? Probably several hours or longer. Why would you save a nanosecond at the risk of spending a few hours?
- *Allocate and release memory in the same function.* Memory allocation and release should be done inside the same function. For a structure, the constructor and the destructor should be called in the same function. Why? It is less likely that you forget the other.
- *Create functions and do not copy-paste code.* Very often your programs need to do something similar but slightly different. An "easy" solution is to copy-paste code and make necessary changes. However, this easy solution can make your program buggy. Why? If you change something in one place and forget to change other places, you have bugs. When you want to copy-paste code, try to create a function and use the arguments to handle the differences.
- *Treat gcc compiler warnings seriously.* Some bugs are detected by `gcc`. On page 210, the warning message is indeed a serious problem. If we pay attention to `gcc` warnings, we can fix many problems before running the program.
- *Understand types.* What are types? Examples are `int`, `double`, and structures explained in Chapter 7. Types define the sizes of memory for storing data; this the reason we need to use `sizeof` when calling `malloc`. Types define what operations are allowed. If `p` is a pointer to a structure object, we can use `p -> attr` to access an attribute. If `i` is an integer, we cannot use `i -> attr`; `gcc` will tell us this is illegal.
- *Use arguments to control functions' behavior.* Each function's behavior should depends only on the arguments. Do not use global variables. Why? Global variables make it difficult to understand a function's behavior. You need to understand where and how the global variables are changed. If you want to understand *any* function where one global variable is used (read or write), you have to understand *all* functions where this global variable is changed.

You should not use static variables, either. What is a static variable? It is created by adding `static` in front of a variable's type. For example,

```
int f(int a, char b)
{
    static int var; /* a static variable */
    ...
}
```

A static variable is visible inside a function only. This property makes a static variable similar to a local variable without the word `static`. However, a static variable's value is kept when the same function is called again. Consider this example,

```

#include <stdio.h>
#include <stdlib.h>
void f(int a)
{
    static int b = 1;
    printf("b = %d\n", b);
    b = a;
}
int main(int argc, char * argv[])
{
    int i;
    for (i = 9; i > 0; i --)
    {
        f(i);
    }
    return EXIT_SUCCESS;
}

```

What is the output of this program? Without the word `static`, this program prints only 1 because `b`'s value is always initialized to 1. With the word `static`, however, the program prints

```

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

```

Why? The word `static` keeps `b`'s value from the previous call of function `f`. Why is it a problem? Imagine the function `f` has a condition based on `b`:

```

if (b > 5)
{
    ....
}

```

You have to know `b`'s value from the previous call of function `f` if you want to know whether this condition is true or not. To make things worse, the condition modifies `b` in this way:

```

void f(int a)
{
    static int b = 1;
    printf("b = %d\n", b);
    if (b > 5)
    {
        ...
        b = a;
    }
    else
    {
        ...
        b = a - 3;
    }
    printf("b = %d\n", b);
}

```

It is really hard to understand what the function does (the ... parts) if you do not know b 's value. You need to keep track of b 's value every time function f is called. This is difficult if f is called a dozen times or more.

Why does C allow global variables and static variables if they should not be used? C was designed more than 40 years ago. Programs are much more complex now and some rules are needed to help write and understand programs.

- *Use global constants and avoid "mysterious" values.* Global variables are bad but global constants are all right. For example, if your program needs the value of π , you can define it as

```
#define PI 3.1415926
```

When you need the value, you simply use `PI`.

You should define a global constant instead of using a mysterious value. For example,

```

int f(int a)
{
    if (a > 3)
    {
        ...
    }
}

```

What is the meaning of 3? Where does it come from? Should it actually be 3.14159? Do you use 3 because you know `a` is an integer? A program can become difficult to understand and debug if you have a few mysterious constants and their values are somehow related. You will easily lose track of them. If you change one, you will likely forget to change the others.

- *Examine, describe, and remove assumptions.* When you write a program, you probably make some assumptions. For example, you may assume a variable is always positive. Your program will not work if the variable becomes zero or negative. Maybe, you assume that a user has only five options. It is inconceivable that anyone would ask for any other option. If these restrictions are true, describe the assumptions clearly as comments or using code to check validity.

In many cases, however, you want to give yourself some flexibility. Why? The world is changing. Remember I told you earlier that programs will probably change many times? If an assumption is not really necessary, remove it. Allow the program to give users more options. As for restricting a number to be positive, have you played a computer game which allows you to teleport when you are about to penetrate a wall? Maybe you can use a negative value to indicate that you are in a different environment. You can have assumptions but you must ensure that the assumptions are necessary and clearly described.

- *Use static functions as needed.* A static function can be called by only the functions in the same file. This prevents you from accidentally calling the function outside the same file. Why is this helpful? For example, in Section 10.2.6 `IteratorInternal_construct` can be called by only function in the file `treeiterator.c` because it is supposed to be “internal” to the iterator. By making `IteratorInternal_construct` a static function, you prevent accidentally calling it outside the file.

12.7 Controversy about `assert`

Some books encourage programmers to use `assert`. Assertion means something must be true; *otherwise, the program stops*. For example,

```
assert(t > 0);
```

You ensure that `t` must be positive at that point, even though `t` may be negative earlier or later. If the assertion fails, i.e. `t` is negative or zero, the program stops. This prevents `t`'s incorrect value from causing problems later. This is a controversial concept. Why? Because you, as the programmer, give up your responsibility. You are saying, “I am *sure* `t` is positive. However, if I am wrong, I don't know what to do.” If the program is safety-critical, stopping the program can trouble. Instead stopping the program, you should handle the situation and recover from the problem. As more and more programs are deployed in safety-critical systems, using `assert` becomes less and less acceptable. Instead, you should

Handle the problem. Don't stop the program.

12.8 Practice Problems

1. The following program counts the occurrences of uppercase letters in a file. The name of the file is given at the command line. The program has multiple problems, even though gcc reports no error or warning. Please identify the errors and correct them.

```
#include <stdio.h>
#include <stdlib.h>
#define TRUE 1
#define FALSE 0
int isUpper(int ch)
{
    if (ch > 65)
        if (ch < 96)
            printf("%c is an uppercase letter.\n", ch);
            return TRUE;
    return FALSE;
}
int main(int argc, char * argv[])
{
    int total;
    FILE * fptr = fopen(argv[1], "r");
    while (! feof(fptr))
    {
        int onechar = fgetc(fptr);
        if (isUpper(onechar) == TRUE)
            printf("There are %d uppercase characters so far.\n", total);
            total ++;
    }
    printf("There are %d uppercase characters.\n", total);
    return EXIT_FAILURE;
}
```

This is the input

```
ABCDEFGXYZ
abcdpqrstu
1234567890
$#^&*(@{\>
```

This is part of the output

```
> gcc -Wall -Wshadow buggy.c -o buggy
> ./buggy testinput
There are 0 uppercase characters so far.
B is an uppercase letter.
There are 1 uppercase characters so far.
C is an uppercase letter.
There are 2 uppercase characters so far.
D is an uppercase letter.
There are 3 uppercase characters so far.
E is an uppercase letter.
There are 4 uppercase characters so far.
...
...
\ is an uppercase letter.
There are 41 uppercase characters so far.
There are 42 uppercase characters so far.
There are 43 uppercase characters so far.
There are 44 uppercase characters so far.
There are 45 uppercase characters.
```


Chapter 13

Test Coverage

When you write code, you probably expect that every line you write will do something in some conditions. If some lines in your program never does anything, your effort is wasted. Why would you write code that does nothing? Thus, if a line never does anything, it probably means the program has a bug. Consider this code I saw in a student's program:

```
if ((x < 0) && (x > 400))
{
    vx = -vx;
}
```

This is part of a computer game of a bouncing ball in a court whose width is 400. The intent of this code is to change the horizontal velocity v_x when the ball hits the left wall $x < 0$ or hits the right wall $x > 400$. What is wrong with this code? The student intends to type

```
if ((x < 0) || (x > 400))
{
    vx = -vx;
}
```

However, the student mistakenly typed `&&` instead of `||`. Since it is impossible for x to be smaller than zero and at the same time greater than 400,

```
vx = -vx;
```

is never executed.

As a programmer, reading your code carefully is extremely important. If you read carefully enough, you should be able to detect this type of errors. Just in case you do not find this problem, a tool may help you. This tool is test coverage. It tells you whether a particular line of code has been executed *for a particular test input*. Let's consider the following program.


```

/*
  file: coverage.c
  purpose: a condition that can never be true
*/
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
  int x;
  int vx = 10;
  for (x = -100; x < 1000; x ++)
  {
    if ((x < 0) && (x > 400))
    {
      vx = -vx;
      printf("change direction\n");
    }
  }
  return EXIT_SUCCESS;
}

```

We can use a tool called `gcov` to find that the two lines

```

vx = -vx;
printf("change direction\n");

```

are never executed.

To run `gcov`, we have to add additional flags after `gcc`:

```

> gcc -g -Wall -Wshadow -fprofile-arcs -ftest-coverage coverage.c -o cov
> ./cov

```

Two files are generated: `coverage.gcd` and `coverage.gcno`. Now, we can run the `gcov` command.

```

> gcov coverage.c
File 'coverage.c'
Lines executed:71.43% of 7
coverage.c:creating 'coverage.c.gcov'

```

Another new file called `coverage.c.gcov` is generated. The following shows the content of the file:

```

-:      0:Source:coverage.c
-:      0:Graph:coverage.gcno
-:      0:Data:coverage.gcda
-:      0:Runs:1
-:      0:Programs:1
-:      1:/*
-:      2:  file: coverage.c
-:      3:  purpose: a condition that can never be true
-:      4:*/
-:      5:#include <stdio.h>
-:      6:#include <stdlib.h>
1:      7:int main(int argc, char * argv[])
-:      8:{
-:      9:  int x;
1:     10:  int vx = 10;
1101:    11:  for (x = -100; x < 1000; x ++)
-:     12:    {
1100:    13:        if ((x < 0) && (x > 400))
-:     14:        {
#####:    15:            vx = -vx;
#####:    16:            printf("change direction\n");
-:     17:        }
-:     18:    }
1:     19:  return EXIT_SUCCESS;
-:     20:}

```

As you can see, lines 15 and 16 are marked by #####. This means these two lines are never executed. When you have a complex program, use `gcov` to detect whether any line is never executed.

If you find typing these commands too much work, write `Makefile` so that you need to type only `make` every time you modify the program.

```

GCC = gcc -g -Wall -Wshadow -fprofile-arcs -ftest-coverage
cov: coverage.c
    $(GCC) coverage.c -o cov
    ./cov
    gcov coverage.c
    grep "#" coverage.c.gcov

clean:
    rm -f *.gcov *.gcno cov

```

If you type `make`, this is the output

```
gcc -g -Wall -Wshadow -fprofile-arcs -ftest-coverage coverage.c -o cov
```

```

./cov
gcov coverage.c
File 'coverage.c'
Lines executed:71.43% of 7
coverage.c:creating 'coverage.c.gcov'

grep "#" coverage.c.gcov
-:      5:#include <stdio.h>
-:      6:#include <stdlib.h>
#####: 15:    vx = -vx;
#####: 16:    printf("change direction\n");

```

This Makefile also has an option called `clean`. If you type `make clean`, the files generated by `gcov` are deleted.

If `gcov` reports that some lines are never executed, the problem may come from the program, as shown in this case. Sometimes, the problem comes from the test inputs. Designing good test inputs is not trivial and some books discuss in details how to design test inputs. Here are some suggestions. Suppose you are writing a program searching whether a value is an element of a sorted array. You should design test inputs to cover the following scenarios:

- The value to search is an element of the array, somewhere in the middle of the array.
- The value is not an element of the array, between some elements in the array.
- The value is the same as the first element.
- The value is the same as the last element.
- The value is smaller than all elements.
- The value is larger than all elements.

As you can see, creating good test inputs is not trivial. A different approach is called *formal verification* by proving a program is correct regardless of inputs. This is an advanced topic and will not be discussed here.

It is important to understand the limitation of test coverage. Low coverage means the test inputs need improvement. However, high coverage is not necessarily better. A good test input is one that can detect problems in your programs. A simple program like the following can get 100% coverage.

```

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
    return EXIT_SUCCESS;
}

```

This program does not do anything. Pursuing high coverage per se should not be your goal. Your goal should be detecting and fixing problems in your programs.

Index

argc, 10
argv, 10
attribute, 108

binary search tree, 171

call stack, 23, 25
command line, 9
compile, 144, 147

deep copy, 118, 124
divide and conquer, 37

function return location, 23
function: declaration, 15
function:definition, 16
function:implementation, 16

heap, 77
heap memory, 21

integer partition, 40
iterator, 182

leaf, 171
link, 3, 144, 147
linked list, 156
Linux Accessories, 5
Linux Terminal, 6

Makefile, 149
memory leak, 79

object, 108
object file, 148

pointer, 67

recursion, 37

scope, 28

separate compilation, 148
shadow variable, 10
shallow copy, 124
short circuit, 168
siblings, 171
stack memory, 19
statck memory, 21
string, 95

terminating condition, 44
tower of Hanoi, 41

valgrind, 79, 211