The Art and Craft of Programming
C Edition

John C. Lusth
David W. Cordes

December 22, 2016

Contents

Chapter 1

Starting Out

A word of warning: The C programming language has a fearsome reputation as a difficult language to learn and correctly program. This reputation is rather undeserved. C is one of the easiest languages in which to program; the areas that give some hackers and newbies fits are easily mastered by following a few simple rules. The purpose of this text is to elucidate those rules.

Most mortals cannot learn how to program by reading books. The same is true for riding a bike or playing the piano. They learn how to program (or ride a bike or play the piano) by practicing. Assuming you are like most mortals, you will need to practice programming as well. Therefore, to start, you will need to install a C compiler and an editor.

We will assume that you are using the editor and C compiler on one of three basic types of systems, a Windows-based system that has Cygwin installed, a Mac system, or a Debian-based system such as Ubuntu (a Unix system). Instructions for ensuring you have an editor and C compiler installed on each of these three types of systems can be found at `http://troll.cs.ua.edu/cs100/install.html`

Once the installation process completes, type the command:

```
    gcc --help
```

and press the `<Enter>` key. You should be rewarded with lots of information on how to use gcc. This will also confirm that your system is configured properly and you are able to compile C programs.

1.1 The edit-compile-test cycle

Writing computer programs involves three tasks, identified below.

1. The developer (you) enters what is called *source code* into a file on the system. Source code is a text version of a computer program that is readable (hopefully) by humans. The program that you use to enter the source code text into a file is called an *editor*.
2. A *compiler* then turns the human readable version of a program into a form that is readable by the computer. The computer readable form is called an *executable*. The compiler is simply another program that exists on the computer. Each programming language (e.g., C, C++, Python, Java, Pascal, FORTRAN) has its own compiler (program) to convert source code into an executable.
3. Once the compiler has created the executable, you must ensure that the code that you entered actually performs the required task. You ask the computer to run the program to ensure that it is functioning properly.

At this point, you should realize that things can go wrong during the second and third steps. If you made mistakes when entering the source code, the compiler will discover these errors (during step two). When this happens, the compiler prints some ominous messages and quits. These errors are known as compilation or *syntax* errors. Misspellings, errors in spacing, and missing symbols at specific locations are just a few of the possible *syntax* errors that a compiler will identify. Upon seeing the error messages produced by a compiler, the program writer (you) must go back to editing the source code to remedy the situation.

Once the compiler is able to successfully translate your source code into an executable (no syntax errors exist), the program can be run and tested. Sometimes errors occur at this stage. For example, the program might crash, loop forever, freeze, or produce an incorrect output. These kinds of errors are known as *run-time* or *logic* errors. These errors are due to the fact that the set of instructions (statements) that you entered in your program (the source code) are not logically coherent. Think of this as giving the wrong directions to someone trying to find Bryant-Denny Stadium. Where they end up is not what you intended. Again, the program writer (you) must go back to editing the source code to fix the problems.

Often, in the course of fixing syntax errors, the program writer introduces logic errors. Just as often, attempts to fix logic errors introduce syntax errors. And so it goes. If you are not a patient person and are easily frustrated, learning how to program will have some very unpleasant moments. On the other hand, impatience and intolerance of setbacks are not widely admired traits; sticking with programming will most assuredly cure you of these personality defects.

If all is copacetic at this point, you are ready to proceed to next chapter.

Chapter 2

The C Framework

Working with C requires that you be able to edit a text file. I use *vim* as my text editor. *Vim* is an editor that was written by programmers for programmers (*emacs* is another such editor) and Serious Computer Scientists and Programmers should learn either *vim* or *emacs*.

C also requires that your program follow certain guidelines, such as your source code having a function called *main*. You may not know what a function is, so we are going to give you an entire program for you to enter into a file named *hello.c*. By convention, all C programs and modules (we will learn about modules later) should be stored as text files with the filenames ending in *.c*. Please note that case is important – you must enter a lowercase *.c*.

2.1 Your first program

Using vim (or emacs), place the following code in a file named *hello.c*:

```
int
main()
    {
    printf("hello, world!");
    return 0;
    }
```

Save your work and exit the text editor.

Now issue the following command at the system prompt to *compile* the file:

```
gcc hello.c
```

Compiling and linking (more on linking when we get to modules) is the process of turning source code (what you placed in the file *hello.c*) into executable code (code that your computer can understand and run). If you do a directory listing after running the above command, you should see a new file created named *a.out*. This new file contains the executable code. To run the code, one types the command:

```
./a.out
```

You should see the phrase:

```
hello, world!
```

displayed on your console. Note that if you are using Cygwin, the executable will be named *a.exe*, instead of *a.out*. Here's a trace of both commands:

```
lusth@warka:~$ gcc hello.c
lusth@warka:~$ ./a.out
hello, world!
lusth@warka:~$
```

where `lusth@warka:~$` is my system prompt; yours will be different.

2.2 More advanced compilation

Most advanced programmers do not compile this simple way. The command you should use from now on should look something like this:

```
gcc -Wall -g hello.c -o hello -lm
```

The term `-Wall` tells the compiler to issue all warnings. This will tell you if you have some stylistic or other non-critical errors in your code. In the Linux world, terms beginning with a dash are denoted *options*. The `-g` option tells the compiler to save information in the executable code that will allow you to debug the code should you need to. The `-o` option says to forgo *a.out* as the name of the executable file; rather the name following the `-o` should be used instead. In the example above, the name of the executable file is designated to be *hello* (no extension). By convention, the name of the executable file is the base name of the *.c* file that contains the *main* function. Finally, the `-lm` at the end of the *gcc* command adds in some important math functions.

If you ran the *gcc* command with the `-Wall` option, you would see a couple of warning messages generated:

```
lusth@warka:~$ gcc -Wall -g hello.c -o hello
hello.c: In function 'main':
hello.c:4:5: warning: implicit declaration of function 'printf'
hello.c:4:5: warning: incompatible implicit declaration of built-in function 'printf'
```

You will receive many such warnings in your C programming career. The warning states that we are calling a built-in function named `printf`, yet we haven't told the compiler how one calls the *printf* function[1]. We fix this problem by telling the compiler to look for the missing information in what is known as a *standard header* file. Header files, by conventions, end in *.h*. We use the *include* directive to indicate the name of the header file. Our program now becomes:

```
#include <stdio.h>

int
main()
    {
```

[1]It seems odd that the compiler does not know about a function it should know about, since *printf* is built in. But C was designed to be a rather flexible language, so we will need to inform the compiler properly for reasons not worth going into at this time.

```
        printf("hello, world!");
        return 0;
        }
```

With this addition, our program compiles without any warnings issued.

From now on, always use the `-Wall`, `-g`, and `-o` options when compiling unless you have good reason not to.

2.3 Breaking down *hello.c*

Let's look in more detail at the *hello.c* program. We have annotated it to show the line numbers:

```
1:    #include <stdio.h>
2:
3:    int
4:    main()
5:        {
6:        printf("hello, world!");
7:        return 0;
8:        }
```

As stated previously, line 1 is an include directive, which informs the compiler what a call to the *printf* function (among many functions) should look like. We can tell we are including a standard header file by the use of the angle brackets (the *less than* and *greater than* symbols). A standard header file is installed for you automatically and you generally don't need to know where it is located[2]. Later, we will learn how to define and include local header files.

Line 3 holds the *return type* of the *main* function. You can think of a function as a little factory. It takes in raw data and produces refined data or *information.* The data, whether raw or refined, is made up of the primitive values or collections thereof. These values all have types, which you will learn about in the next chapter. One of the primitive types is `int`, which stands for an integer. So line 3 is saying that the *main* function will produce an integer as a result of the work it does.

Line 4 holds the name of the function (in this case *main*) and a comma-separated, parenthesized list of the receptacles that will hold the incoming raw data (in this case, there are no receptacles). If *main* had required some incoming raw data to do its job, the names of the receptacles and the types of data the receptacles hold would be listed between the parentheses. As *hello.c* is such a simple program, the *main* function does not need any incoming data, leading to the empty set of parentheses. Together, lines 3 and 4 compose the *signature* of a function. Later on, you will learn more on functions, function signatures, and formal parameters (the official name of the receptacles that hold the incoming raw data).

Line 5 is an *open brace*, which indicates the beginning of a *block.* A block is a section of code which is run sequentially. A line of code that appears earlier in the block is *executed*[3] after any preceding code in the block and prior to any subsequent code in the block. Line 8 holds a *close brace*, which indicates the end of a block of code.

Lines 6 and 7 are the lines of code in the block that is associated with the *main* function. We visually show the block belongs to *main* by indenting it. Line 6 prints out the message we see when we run the program. Line 7 gives the refined data value that the *main* function produces as a result of all its work. In this case, it says that the *main* function returns a zero when it is done. Note that zero is an integer, which matches

[2]On Linux and other Unix-like systems, the standard header files are located in the `/usr/include/` directory.

[3]*Executed* is such a gruesome term, but it means *run* or *performed.* It derives from the idea of an *executive* performing the task indicated by the line of code.

the return type found on line 3. The return value of a function is almost always used in some way, often serving as the raw data to be given to some other function. In the case of *main* functions in C programs, the return value of *main* indicates whether or not the program ran successfully. By conventions, a zero return value from *main* means 'zero errors'. If an error occurred, some other integer value would be returned. We will learn how to choose between returning zero or some other integer when we learn about conditionals.

All C programs require a *main* function, so until you can write a *main* function for scratch, use the following version as a starting point:

```
//template for testing code
//download with: wget troll.cs.ua.edu/ACP-C/tester.c
#include <stdio.h>     // Provides standard I/O functions
#include <stdlib.h>    // Includes a number of basic built-in functions
#include <string.h>    // Provides functions that operate on strings
#include <math.h>      // Provides a wealth of mathematical functions

int
main(int argc,char **argv)
    {
    /* place code here */

    return 0;
    }
```

You'll notice this version has a few more include directives plus a couple of formal parameters for the *main* function. These formal parameters accept any command-line arguments (as stated earlier, command-line arguments will be covered in a later chapter). In addition, the very first line is a *comment*, meaning it is there to be read, but is otherwise ignored by the C compiler. Comments that begin with `//` are called *line comments*; the compiler ignores those characters and every character beyond on the same line. The template program exhibits another style of comment, the block comment. In a block comment, the `/*` and `*/` characters and all the characters between, are ignored by the compiler. The advantage of block comments is that they can be placed in between source code bits and can also span lines.

2.4 About C programs and whitespace

Other than line comments and include directives, there are no rules for C programs specifying what parts of the programs have to be on what lines. For example, the template program above could have been written like this:

```
//template for testing code
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(int argc,char **argv) { /* place code here */ return 0; }
```

Note that the entire *main* function is on a single line. This is because C is a *free-format* language. A free-format language says wherever you can have *whitespace* (a space or a tab or a newline), you can replace it with some other kind of whitespace or any combination of whitespace characters.

2.5 Vim and C

In order to be most efficient at programming, one needs to configure the editor to understand the language being edited. We will do that now.

Move into your home directory with the:

```
cd
```

command and list the files found there with this version of the *ls* command:

```
ls -al
```

If you see the file *.exrc*, then all is well and good. If you do not, then run the following command to retrieve it:

```
(cd ; wget troll.cs.ua.edu/ACP-C/.exrc)
```

This places a copy of the `.exrc` file in your home directory. It contains a number of commands that help customize `vim` for writing C programs, they configure `vim` to understand C syntax and to color various primitives and keywords in a pleasing manner. These commands are processed every time that you invoke `vim`.

2.6 Four Shortcuts to Improve Efficiency

You can program your entire life without knowing the following four shortcuts. However, that is the same as typing `http://` on every website that you visit. You can save time if you know a few basic tricks for the *edit-compile-test* cycle.

2.6.1 Running a Program without leaving the Editor

Since you spend a lot of your development time in the edit-compile-test cycle, being able to compile and test your program without officially leaving the editor can save significant time. We can add some macros (shortcuts) to *vim* so that compilation and testing of your program can be done while editing.

First, go to your home directory (*cd*) and check to see if you have an *.exrc* file. If you retrieved the `.exrc` file from *troll.cs.ua.edu* as described in Section 2.5, then this line already exists. If you do not have a *.exrc* file in your home directory then either copy it (as described in Section 2.5) or create the file yourself and add this line at the top of the file:

```
map @    :!clear; rm -f a.out; gcc -Wall -g % -lm; ./a.out^M
```

The `^M` part of the macro is not a two character sequence (`^` followed by `M`), but a single character made by typing `<Ctrl>-v` followed by `<Ctrl>-m`. It's just when you type `<Ctrl>-v <Ctrl>-m`, it will display as `^M`.

What this line says is that, when the `@` key is pressed, the system should:

1. clear the screen,

2. remove the old *a.out* file,
3. run the *gcc* compiler on the file you are currently editing and place the executable code in the default file named *a.out*,
4. then run the executable code.

Remember, to save your work (`:w` is one method) before running this command. Also, if you are using Cygwin, then replace the occurrences of *a.out* with *a.exe*.

How does this work? Suppose you are editing a file named *project1.c*, then the `@` command would be equivalent to you first saving your file, exiting the editor, and then running the commands:

```
lusth@warka:~$ rm a.out
lusth@warka:~$ gcc -Wall -g project1.c -lm
lusth@warka:~$ ./a.out
```

You can see that the `@` macro saves you quite a bit of typing!

2.6.2 Running a Program with Command Line Arguments from within the Editor

As you expand your programming knowledge, you will be writing programs that take *command-line arguments* when you execute them. These are simply data values that are provided when you execute the program. In these cases, you need to be able to enter them before hitting `return` on the line that invokes your program. So that `^M` in the `@` command above needs to be removed, as `^M` represents an *enter* on the command line. Thus, we create a new macro `#` that is the same as the old, except the `^M` has been removed. The `#` macro pauses to let you enter command-line arguments to your C program[4].

```
map # :!clear; rm a.out; gcc -Wall -g % -lm; ./a.out
```

As you did with the `@` macro, add the line above to the *.exrc* file in your home directory if it does not already exist.

2.6.3 Configuring VIM to Display C Programs Elegantly

The commands below help *vim* to display C programs in a pleasing format. Make sure to add these lines to your *.exrc* if you had to create one from scratch:

```
set autoindent
set shiftwidth=4
set tabstop=4
set expandtab
set softtabstop=4
set smarttab
syntax enable
autocmd FileType make setlocal noexpandtab
```

[4]More on command-line arguments in a later chapter.

2.6.4 Cut and Paste in VIM using the Mouse

You can copy and paste test code with the mouse. On a Linux system, the copy command is `Shift-Control-c`, rather than `Control-c` used in Windows. This is because `Control-c` in Linux means "kill the currently running program". You will find `Control-c` quite useful when you write programs that don't behave like you think they should. The paste command is `Control-v`, just like it is in Windows.

Because you likely have autoindent turned on in Vim, when you get ready to paste, turn off autoindent with the vim command:

```
:set noai
```

Then go into insert mode and do the paste. To turn autoindent back on, enter this vim command:

```
:set ai
```

The *.exrc* file on *troll* has a macro that takes care of the pasting issue:

```
map <F9> :set paste!
```

This macro toggles autoindent, so to paste with the mouse, press `<F9>` first. Press `<F9>` again to turn autoindent back on.

2.7 One Final Comment

In subsequent chapters, code that we wish for you to insert into the template C program, *tester.c*, given above will start with:

```
//test
```

This is simply a shorthand notation used in the text. Mentally, you should replace `//test` with:

```
//template for testing code
//download with: wget troll.cs.ua.edu/ACP-C/tester.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

int
main(int argc,char **argv)
    {
```

and follow the test code with:

```
    return 0;
    }
```

Computer Scientists are all about efficiency; we can make the process of testing code fragments in this book even easier. We do so by writing a program (no surprise there) that does this mental replacement for you; the program is called *quickc* and is discussed in the next chapter.

Chapter 3

Testing code examples

Starting with the next chapter, we look at the C programming language and and its features and usage. As you read these chapters, you *should* want to try the various constructs and operations so that you fully understand how they operate. You can do this by writing a series of small C programs and then compiling and executing each of these programs. This chapter introduces a second method to practice using the C programming language. This method provides a clean, simple interface with minimal overhead for your practice.

This second method uses a program called *quickc*. It is designed for testing and experimentation. We expect you to write your projects and lab programs and exams using the traditional *edit-compile-test* process described previously. You should use *quickc* when you want to just do some quick checks or tests on pieces of C code. While we believe that *quickc* will be beneficial to your studies, it is not required. If you prefer to use the traditional *edit-compile-test* cycle for everything that you do with the C programming language, then you can skip the remainder of this chapter.

The *quickc* program is provided to help you quickly test some of the code examples in the book. For example, suppose you see some code fragment that is preceded with the C comment `//test`, as in:

```
//test
printf("Hello, world\n");
```

With the *quickc* program, you can copy and paste the code fragment at the *quickc* prompt, press the `<Enter>` key, and see the result of the code fragment being executed.

3.1 Using *quickc*

For example, here is the *quickc* program starting up in a terminal window. To the terminal window prompt (in this case `$`), you would type `quickc`:

```
$ quickc
>
```

The *quickc* prompt is the “greater than” sign. Pasting the above code fragment would look like:

```
$ quickc
>     //test
+     printf("Hello, world\n");
+
```

At this point, press the `<Enter>` key and you will see the fragment compiled (as part of an automatically generated C program) and executed:

```
$ quickc
>     //test
+     printf("Hello, world\n");
+
compiling the program...
running the program...
Hello, world
>
```

To exit the *quickc* program, type the `q` character followed by the `<Enter>` key. The *quickc* program can do lots of other tasks; typing the `h` character brings up a menu of these additional tasks.

If you make a mistake entering some code, you can use the *up* and *down* arrow keys to recover a statement, which you can then edit, using the *backspace*, *delete*, and *left* and *right* arrow keys. Pressing `<Enter>` at this point adds the edited line to the program.

3.2 Installing *quickc*

To get the *quickc* program, first retrieve it from the ACP server:

```
wget troll.cs.ua.edu/ACP-C/quickc.c
```

Then compile it with the command:

```
gcc -Wall quickc.c -o quickc -lm -lreadline
```

If you see an error that the symbol `free_history_entry` is not defined, edit *quickc.c* and change the line near the top that reads:

```
#define FREE_HISTORY 1
```

to read:

```
#define FREE_HISTORY 0
```

and recompile. Once compiled, place the *quickc* executable in a directory that's part of your search path, say `~/bin/`, `/usr/local/bin/` or `/usr/bin/`.

Chapter 4

Literals

C works by figuring out the meaning or value of some code. This is true for the tiniest pieces of code to the largest programs. The process of finding out the meaning of code is known as *evaluation*.

The things whose values are the things themselves are known as *literals*. The literals of C can be categorized by the following types: *integers*, *real numbers*, *characters*, and *strings*.

4.1 Integers

Integers are numbers without any fractional parts. Examples of integers are:

```
3
-5
0
```

Integers must begin with a digit or a minus sign. The initial minus sign must immediately be followed by a digit.

The integer type in C is `int`. There are other integer types that we will not get to in this book. You can find them by searching the interwebs for "C integral types".

It is important to realize that there are limits to what can be stored in an integer. As mentioned above, C has several different types of integers that are designed for specific situations. For example, if a developer only ever store values in the range of 0 to 100 then that person could use a smaller version of an integer that saves space. In this class, you should always use the default `int` class unless you are told otherwise.

It is recommended that you search the interwebs to determine the maximum (and minimum) value that you can store in an integer in C.

4.2 Real Numbers

Reals are numbers that do have a fractional part (even if that fractional part is zero!). Examples of real numbers are:

```
3.2
4.000
5.
0.3
```

```
.3
3.2e-4
3e4
.000000987654321
```

Real numbers must start with a digit or a minus sign or a decimal point. An initial minus sign must immediately be followed by a digit or a decimal point. An initial decimal point must immediately be followed by a digit. C accepts real numbers in scientific notation. For example, $3.3 * 10^{-11}$ would be entered as 3.3e-11. The "e" stands for exponent and the 10 is understood, so `e-11` means multiply whatever precedes the `e` by 10^{-11}.

The real number type in C is `double`. This is short for *double precision floating point number*. There is another real number type called *float*, which is short for *single precision floating point number*, but it is not used very much.

As with integers, there are limits to how big (and small) a number can be stored in a *double*. It is recommended that you do another interwebs search to learn the limits on real numbers. You might find it useful to compare the limits of a *double* and a *float* as that will help you understand why most programmers simply use *double*.

4.3 Characters

In C, characters generally are single letters, digits, or symbols delineated by single quotes:

```
'a'
'Z'
'&'
'9'
' '
```

The last character in the above list is the space character. There are also a number of two-character characters, some of which are:

```
'\n'
'\t'
'\0'
'\''
'\\'
```

The first symbol in these two-character characters is the *backslash*; symbols preceded by a backslash are said to have been *escaped*. Escaping a symbol will often change its meaning. For example, the character *n*, in a string refers to the letter *n* while the character sequence `\n` refers to the *newline* character. A backslash also changes the meaning of the letter *t*, converting it into a tab character. An escaped zero is the null character which used to terminate strings. To specify the single quote character or a backslash, one needs to escape them with a backslash, otherwise the C compiler will get confused. Are you confused? Don't worry. It'll become natural after a while.

The character type in C is `char`. There is a great debate on how to properly pronounce "char'. Three groups have emerged, those who know how to pronounce "char" properly (these folks say "care" as in "careful") and those who don't (these folks say "char" as in "char" as in "charcoal"). The third group takes a more Southern approach to the pronunciation, (these folks say "car" as in "cartoon"). Who knows who is right?

A character is one byte (8 bits). If you want to see how the various characters (`'a'`, `'$'`, `'Z'`, *etc.*) are stored, do a search on the interwebs for ASCII tables.

4.4 Strings

Strings are sequences of characters delineated by double quotation marks:

```
"hello, world!"
"x\nx"
"\"z\""
""
```

Like characters, the symbols in a string can be *escaped* with the backslash character, You can also escape double quotes with backslashes so that you can include them in a string. A string with no characters between the double quotes is known as an empty string.

The string type in C is `char *`, pronounced “char-star”. This is because strings are actually arrays of characters and, as you will learn later, an asterisk can be used to indicate an array. So `char *` can be read as “array of `char`”. We will delve into arrays in a later chapter.

4.5 Booleans

Many languages have an additional kind of literal that C does not have, the Boolean literals `True` and `False`. Booleans are used to guide the flow of a program. The term Boolean is derived from the last name of George Boole, who, in his 1854 paper *An Investigation of the Laws of Thought, on which are founded the Mathematical Theories of Logic and Probabilities*, laid one of the cornerstones of the modern digital computer. The so-called Boolean logic or Boolean algebra is concerned with the rules of combining truth values (i.e., true or false). As we will see, knowledge of such rules will be important for making C programs behave properly. In particular, Boolean expressions will be used to control conditionals and loops.

Even though C has no explicit Booleans, the integers serve double-duty, both as numbers and as Boolean literals. In C, the integer zero serves as `False` and any other integer or integer-like value serves as `True`. Novices sometimes make the mistake that only the number 1 is *True*. It is `True`, but it is not the only `True`. For example, 2384, -12, and 42 are also `True`.

Chapter 5

Variables

Suppose you found an envelope lying on the street and on the front of the envelope was printed the name *numberOfDogsTeeth*. Suppose further that you opened the envelope and inside was a piece of paper with the number 42 written upon it. What might you conclude from such an encounter? Now suppose you kept walking and found another envelope labeled *meaningOfLifeUniverseEverything* and, again, upon opening it you found a slip of paper with the number 42 on it. Further down the road, you find two more envelopes, entitled *numberOfDotsOnPairOfDice* and *StatueOfLibertyArmLength*, both of which contain the number 42.

Finally, you find one last envelope labeled *sixTimesNine* and inside of it you, yet again, find the number 42. At this point, you're probably thinking "somebody has an odd affection for the number 42" but then the times table that is stuck somewhere in the dim recesses of your brain begins yelling at you saying "54! It's 54!". After this goes on for an embarrassingly long time, you realize that 6 * 9 is not 42, but 54. So you cross out the 42 in the last envelope and write 54 instead and put the envelope back where you found it.

This strange little story, believe it or not, has profound implications for writing programs that both humans and computers can understand. For programming languages, the envelope is a metaphor for something called a *variable*, which can be thought of as a label for a place in memory where a literal value can reside. In other words, a variable can be thought of as a convenient name for a value. In many programming languages, one can change the value at that memory location, much like replacing the contents of an envelope[1]. A variable is our first encounter with a concept known as *abstraction*, a concept that is fundamental to the whole of Computer Science[2].

5.1 Variables

Most likely, you've encountered the term *variable* before. Consider the *slope-intercept* form of an algebraic equation of a particular line:

$$y = 2x - 3$$

You probably can tell from this equation that the slope of this line is 2 and that it intercepts the y-axis at -3. But what role do the letters y and x actually play? The names x and y are placeholders that stand for the x- and y-coordinates of any conceivable point on that line. Without placeholders, the line would have to be described by listing every point on the line. Since there are an infinite number of points, clearly an exhaustive list is not feasible. As you learned in your algebra class, the common name for a place holder for a specific value is the term *variable.*

[1]Languages that do not allow changes to a variable are called *functional languages*, while those that do are called *imperative languages.* C is an imperative language.

[2]Another fundamental concept of Computer Science is *analogy* and if you understand the purpose of the envelope story after reading this section, you're well on your way to being a Computer Scientist!

One can generalize[3] the above line resulting in an equation that describes every line.

$$y = mx + b$$

Here, the variable m stands for the slope and b stands for the y-intercept. Clearly, this equation was not dreamed up by an English-speaking Computer Scientist; a cardinal rule is to choose good names or mnemonics for variables, such as s for slope and i for intercept. But alas, for historical reasons, we are stuck with m and b.

The term *variable* is also used in most programming languages, including C, and the term has roughly the equivalent meaning. The difference is programming languages use the envelope metaphor while algebraic meaning of variable is an equivalence to a value[4]. The difference is purely philosophical and not worth going into at this time.

5.2 Creating, combining, and printing variables

Suppose you found three envelopes, marked m, x, and b, and inside those three envelopes you found the numbers 6, 9, and -12 respectively. If you were asked to make a y envelope, what number should you put inside? If the number 42 in the *sixTimesNine* envelope in the previous story did not bother you (e.g., your internal times table was nowhere to be found), perhaps you might need a little help in completing your task. We can have C calculate this number with the following dialog:

```
//test - copy this code and paste it at the quickc prompt
int m = 6;
int x = 9;
int b = -12;
int y = m * x + b;
printf("y is %d\n",y);            //should print: y is 42
```

One creates variables in C by first giving the type of the value the variable is to hold. In this case, we are creating variables that hold integers; the keyword `int` tells us that. Next comes the name of the variable. If one wishes to immediately give the variable a value (which we do), we follow the variable name with an equals sign and the value we wish the variable to have. Note that this value can be a literal or some combination of literals and variables. In any case, the type of the expression (generally) should match the type of the variable.

The code that creates a variable is known as a *variable definition.* When a variable is defined, the compiler allocates some space in memory to hold the value of the variable. Sometimes, we wish to inform the compiler that a variable will be defined elsewhere. For that task, we use a *variable declaration.* To declare that a variable exists somewhere else, we use the `extern` keyword. For example, to say that the variable x, defined above exists, we would say:

```
extern int x;      //variable declaration: x exists!
```

[3] The third great fundamental concept in Computer Science is *generalization.* In particular, Computer Scientists are always trying to make things more abstract and more general (but not overly so). The reason is that software/systems/models exhibiting the proper levels of abstraction and generalization are much much easier to understand and modify. This is especially useful when you are required to make a last second change to the software/system/model.

[4] Even the envelope metaphor can be confusing since it implies that two variables having the same value must each have a copy of that value. Otherwise, how can one value be in two envelopes at the same time? For simple literals, copying is the norm. For more complex objects, the cost of copying would be prohibitive. The solution is to storing the *address* of the object, instead of the object itself, in the envelope. Two variables can now "hold" the same object since it is the address is copied.

This tells the compiler the subsequent occurrences of x in the code refer to an integer and that space for x has already been allocated. Many people, even long-time C programmers, confuse the terms *definition* and *declaration.* The reason is C does not treat declarations and definitions consistently. For example, in some contexts, C can treat:

```
int x;
```

as a declaration, but in others, C treats it as a definition[5]. Just remember, if you see a type and a variable name, it's (likely) a definition. If you see the word `extern` in front, it's a declaration.

Continuing on with the example above, we combine the values of m, x, and b using multiplication[6] and addition to determine the value of variable y. C, when asked to compute the value of an expression containing variables, as in:

```
m * x + b
```

goes to those envelopes and retrieves the values stored there and combines them as directed. The last line of code:

```
printf("y is %d\n",y);
```

shows of some of the sophistication of the *printf* function. As we discussed when we introduced the *main* function, functions can take in raw data. In this case, the *printf* function is being passed two pieces of information, a string and a variable, separated by a comma. The string guides *printf* on how to do the printing. Inside the string is the character sequence:

```
%d
```

This informs *printf* to look for an additional value beyond the string and substitute that value for the `%d`. Moreover, the `d` in the `%d` indicates that *printf* should print that value as an integer. Moreover, when we give a variable to a function to be used as data, we actually send the value of the variable. In our code, the value of y will be 42, so we should see:

```
y is 42
```

displayed when we run the program. The `%d` is known as a *format directive.* Inserting our test code into the template program *tester.c* gives us the following source code:

```
//template for testing code
//download with: wget troll.cs.ua.edu/ACP-C/tester.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
```

[5]We will cover this in more detail in the chapter on Scope.

[6]In the algebraic equation $mx + b$, the multiplication sign is elided, but most programming languages, C included, require the presence of the multiplication sign, For C, the multiplication sign is the asterisk.

```
    int main(int argc,char **argv)
        {
        //test code goes here
        int m = 6;
        int x = 9 ;
        int b = -12;
        int y = m * x + b;

        printf("y is %d\n",y);

        return 0;
        }
```

Compiling:

```
    lusth@warka:~/cs100$ gcc -Wall -g tester.c -o tester
```

and running this program produces the output:

```
    lusth@warka:~/cs100$ tester
    y is 42
```

just as expected. Pasting the test code at the *quickc* prompt gives us the same result.

Here are some more examples of variable creation:

```
    //test
    int dots = 42;
    int bones = 206;
    printf("%d\n",dots);           //should print: 42
    printf("%d\n",bones);          //should print: 206
    int CLXIV = bones - dots;
    printf("%d\n",CLXIV);          //should print: 164
```

After a variable is created, the variable or its value can be used interchangeably. For example, it is an easy matter to set up an equivalence between the variable PI and the real number 3.14159.

```
    //test
    double PI = 3.14159;
    double radius = 10.0;
    double area = PI * radius * radius;
    double circumference = 2 * PI * radius;
    printf("area is %f, while circumference is %f\n",area,circumference);
```

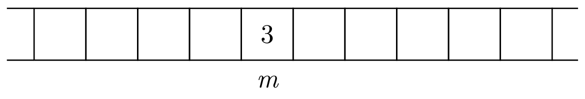
Compiling and running this code inside the template program should yield the output:

```
    area is 314.159000, while circumference is 62.831800
```

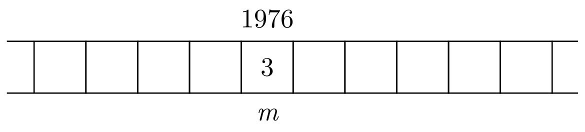
Recall that the `double` type is used for variables that hold real numbers. Notice how the expressions used to compute the values of the variables area and circumference are more readable than if 3.14159 was used instead of PI. In fact, that is one of the main uses of variables, to make code more readable. The second is if the value of PI should change (e.g., if a more accurate value of PI is desired[7]), we would only need to change the definition of PI. From the example above, we can see that to print a real number, we need to use the `%f` format directive in the guide string for *printf*.

5.3 Variables and memory

A variable in C refers to a memory location. At that memory location, the value of the variable is stored. One can imagine memory as a series of boxes. Using m as an example, the variable m identifies one of those boxes and the value of m is placed inside that box:



Just like a house on a street (the street being memory and the house being the box), every box in memory has an address. Having variables makes the actual address irrelevant, but let's just say the variable m refers to the box at address 1976 in memory[8]. If actual addresses are important, we will place them above the box:



Since the actual address of a memory location rarely matters, we will most often use a simpler notation for variables and their values:

$$m: \quad 6$$

This notation should be read as "m identifies a location in memory that holds the value 6". Often, we will shorten this to "m is 6".

5.4 Variable naming

Like many languages, C is quite restrictive in regards to legal variable names. A variable name must begin with a letter or an underscore and may be followed by any number of letters, digits, or underscores.

Variables are the next layer in a programming languages, resting on the literal expressions and combinations of expressions (which are expressions themselves). In fact, variables can be thought of as an abstraction of the literals and collections of literals. As an analogy, consider your name. Your name is not you, but it is a convenient (and abstract) way of referring to you. In the same way, variables can be considered as the names of things. A variable isn't the thing itself, but a convenient way to referring to the thing.

While C lets you name variables in wild ways:

[7]The believed value of PI has changed throughout the centuries and not always to be more accurate (see http://en.wikipedia.org/wiki/History_of_Pi)

[8]Actual addresses of memory locations in C are usually large numbers, typically displayed in octal or hexadecimal, rather than decimal.

```
int _1_2_3_iiiii__ = 7;
```

you should temper your creativity if it gets out of hand. For example, rather than use the variable *m* for the slope, we could use the name *slope* instead:

```
int slope = 6;
```

We could have also used a different name:

```
int _e_p_o_l_s_ = 6;
```

The name *_e_p_o_l_s_* is a perfectly good variable name from C's point of view. It is a particularly poor name from the point of making your C programs readable by you and others. It is important that your variable names reflect their purpose. In the example above, which is the better name: *b*, *i*, *intercept*, or *_t_p_e_c_r_e_t_n_i_* to represent the intercept of a line?

5.5 Uninitialized variables

If you create an integer variable, but don't initialize it:

```
int x;
```

the value held by that variable is some random integer[9]. Thus, if your program uses the value of an uninitialized variable, not only is it likely to compute incorrect results, it will also likely compute different results each time it is run. The same is true of all other types of variables as well.

An uninitialized variable is said to be *filled with garbage*, which leads to the old saying about computer programs:

> *garbage in, garbage out*

[9]The value is not really random. A better description would be *detritus*, the stuff left over from previous activities. The actual value found in an uninitialized variable are the bits left over from previous programs that used that memory location.

Chapter 6

Operators

Way back when, computers were used primarily as calculators. Some whiz-bang programmer would code up some formula and run the program on a bunch of data. In fact, the very first high-level programming language was called Fortran, for *for*mula *tran*slation. Fortran, and most every language that came after it, supplied a full complement of mathmatical operators. C is no exception. For example, suppose you have forgotten your times table and aren't quite sure whether 8 times 7 is 54 or 56. We can write a C program to tell us the answer:

```
//test
printf("%d\n",8 * 7);           //should print: 56
```

Of course, we can multiply variables as well as numbers:

```
//test
int x = 8;
int y = 7;
printf("%d\n",x * y);           //should print: 56
```

We can also intermix variables and numbers:

```
//test
int x = 8;
printf("%d\n",x * 7);           //should print: 56
```

Time for some terminology. The multiplication sign `*` is known as an *operator*, as it *operates* on the numbers or variables on either side of it, producing an equivalent literal value. The items preceding and following the multiplication sign are known as *operands*. It seems that the actual names of various operands are not being taught anymore, so for nostalgia's sake, here they are. The operand to the left of the multiplication sign is known as the *multiplicand*. The operand to the right is known as the *multiplier*. The result is known as the *product*.

The operands of the other basic operators have special names too. For addition, the left operand is known as the *augend* and the right operand is known as the *addend*. The result is known as the *sum*. For subtraction, the left operand is the *minuend*, the right the *subtrahend*, and the result as the *difference*. For division (and I think this is still taught), the left operand is the *dividend*, the right operand is the *divisor*, and the result is the *quotient*.

In general, we will separate operators from their operands by spaces, tabs, or newlines, collectively known as *whitespace*.[1] It's not necessary to do so, but it makes your code easier to read.

C always takes an expression and computes an equivalent literal expression (e.g., integer or real). Most of the C operators are binary, meaning they operate on exactly two operands. We first look at the numeric operators.

6.1 Numeric operators

If it makes sense to add two numbers together, you can probably do it in C using the + operator. For example:

```
//test
printf("%d\n",2 + 3);          //should print: 5
printf("%f\n",1.9 + 3.1);      //should print: 5.000000
```

One can see that if one adds two integers, the result is an integer. If one does the same with two reals, the result is a real.

Things get more interesting when you add things having different types. Adding an integer and a real (in any order) always yields a real.

```
//test
printf("%f\n",2 + 3.3);        //should print: 5.300000
printf("%f\n",3.3 + 2);        //should print: 5.300000
```

Adding a string to a real number yields an error; the types are not "close" enough, like they are with integers and reals. This statment:

```
printf("%f\n",2.2 + "123");
```

generates the following error when compiled:

```
error: invalid operands to binary + (have 'double' and 'char *')
```

In general, when adding two things, the types must match or nearly match.

Subtraction, multiplication, and division of numbers follow the same rules as addition. Of special note is the division operator with respect to integer operands. Consider evaluating the following expression:

```
15 / 2
```

If one asked the C compiler to compute the result, the value 7, not 7.5, would be produced. This is because, when given an integer dividend and an integer divisor, the quotient will be an integer. Since 7.5 is not an integer, the fractional part is lopped off to make it an integer. If we wish to compute a real result, we need to *cast* the operands:

[1]Computer Scientists, when they have to write their annual reports, often refer to the things they are reporting on as *darkspace*. It's always good to have a lot of darkspace in your annual report!

```
//test
int x = 15, y = 2;
int i_quotient;
double r_quotient;
//calculate!
i_quotient = x / y;
r_quotient = (double) x / (double) y;           //(double) is a cast
printf("%d and %f\n",i_quotient,r_quotient);    //should print: 7 and 7.500000
```

For division, dividing an integer dividend by a real divisor or vice versa, a real number results. Therefore, we could have used any of the following ways to compute *r_quotient*.

```
r_quotient = (double) x / (double) y;
r_quotient = (double) x / y;
r_quotient = x / (double) y;
```

The complement to integer division is the modulus operator %. While the result of integer division is the quotient, the result of the modulus operator is the remainder. Thus

```
14 % 5
```

evaluates to 4 since 4 is left over when 5 is divided into 14. To check if this is true, one can ask C to compute this expression:

```
(14 / 5) * 5 + (14 % 5) == 14
```

This complicated expression asks the question "is it true that the quotient times the divisor plus the remainder is equal to the original dividend?". Running the following code will show that, indeed, it is true as a 1 is printed.

```
$ quickc
> printf("%d\n",(14 / 5) * 5 + (14 % 5) == 14);
+
compiling the program...
running the program...
1
>
```

The first set of parentheses in the expression surrounds the quotient while the second pair surrounds the remainder.

We can also use the modulus operator to check whether a value is even or odd. The expression

```
x % 2 == 0
```

evaluates to true if the value of x is even, false otherwise.

6.2 Precedence

Precedence (partially) describes the order in which operators, in an expression involving different operators, are evaluated. In C, the expression

```
3 + 4 < 10 - 2
```

evaluates to true. In particular, `3 + 4` and `10 - 2` are evaluated before the `<`, yielding `7 < 8`, which is indeed true. This implies that `+` and `-` have higher precedence than `<`. If `<` had higher precedence, then `4 < 10` would be evaluated first, yielding `3 + True - 2`, which is nonsensical.

Note that precedence is only a partial ordering. We cannot tell, for example whether `3 + 4` is evaluated before the `10 - 2`, or vice versa. Upon close examination, we see that it does not matter which is performed first as long as both are performed before the expression involving `<` is evaluated.

It is common to assume that the left operand is evaluated before the right operand. For the Boolean connectives `And` and `Or`, this is indeed true. But for other operators, such an assumption can lead you into trouble. You will learn why later. For now, remember never, never, never depend on the order in which operands are evaluated!

The C language has a complete precedence chart that illustrates operator precedence for all possible operators. This chart includes 15 levels of precedence. However, since it includes a number of operators that we have not yet discussed, we will simply provide you with a subset of this chart for now.

- The lowest precedence operator in C is the assignment operator (which is described later)
- Next, the Boolean connectives `And` and `Or`
- At the next higher level are the Boolean comparatives, `<`, `<=`, `>`, `>=`, `==`, and `!=`
- After that come the additive arithmetic operators `+` and `-`
- Next comes the multiplicative operators `*`, `/` and `%`.
- Finally, at the highest level of precedence is the selection, or *dot*, operator (the dot operator is a period or full-stop).

Higher precedence operations are performed before lower precedence operations. Functions which are called with operator syntax have the same precedence level as the mathematical operators.

Parentheses can also be used to change the *precedence* of operators; in a complicated expression composed of differing operators, subexpressions with parentheses are done first, then dot operations, then division and mulitplication, then addition and subtraction, and so on.

General comment on precedence: unless you have the C Operator Precedence Chart memorized, it is always a good idea to use lots of parentheses whenever you are using multiple operators in an expression or statement. That way, you know exactly how things will be handled.

Finally, you are encouraged to find a complete C Operator Precedence Chart on the web and examine it. It contains lots of useful information.

6.3 Associativity

Associativity describes how multiple expressions connected by operators at the same precedence level are evaluated. All the operators, with the exception of the assignment operator, are left associative. For example, the expression $5 - 4 - 3 - 2 - 1$ is equivalent to $((((5 - 4) - 3) - 2) - 1)$. For a left-associative structure, the equivalent, fully parenthesized, structure has open parentheses piling up on the left. If the minus operator was right associative, the equivalent expression would be $(5 - (4 - (3 - (2 - 1))))$, with the close parentheses piling up on the right. For a commutative operator, it does not matter whether it is left associative or right associative. Subtraction, however, is not commutative, so associativity does matter. For the given expression, the left associative evaluation is -5. If minus were right associative, the evaluation would be 3.

Like subtraction, division, which is non-commutative, exhibits left associativity. In the expression:

```
a / b / c / d
```

the divisions are done from left to right. The above expression is equivalent to:

```
(((a / b) / c) / d)
```

The assignement operator, covered in the next chapter, exhibits right associativity.

6.4 Comparing things

Remember the Boolean interpretation of integers? We can use the Boolean comparison operators to generate such values. For example, we can ask if 3 is less than 4:

```
//test
int b = 3 < 4;
printf("%d\n",b);           //should print: 1
```

The output says that, indeed, 3 is less than 4, since 1 is considered true. If the expression had resolved to false, the output would be 0, since 0 represents falsity. Besides `<` (less than), there are other Boolean comparison operators: `<=` (less than or equal to), `>}` (greater than), `>=` (greater than or equal to), `==` (equal to), and `!=` (not equal to).

Note that two equal signs are used to see if to things are the same. A single equals sign is reserved for the assignment operator, which you will learn about in the next chapter.

Besides integers, we can compare reals with reals and integers and reals using the comparison operators. We cannot compare strings with strings using the Boolean operators, however. We need a special operator to do that. This operator, *strcmp*, is covered in the chapter on strings.

6.5 Combining comparisons

We can combine comparisons with the Boolean logical connectives `&&` and `||`, logical AND and logical OR, respectively.

```
//test
int b = 3 < 4 && 4 < 5; //true AND true
printf("%d\n",b);           //should print: true (which is 1)
b = 3 < 4 || 5 < 4; //true OR false
printf("%d\n",b);           //should print: true (which is 1)
b = 3 < 4 && 5 < 4; //true AND false
printf("%d\n",b);           //should print: false (which is 0)
```

The first bit of code asks if both the expression `3 < 4` and the expression `4 < 5` are true. Since both are, *C* calculates a `1`. The second bit of code asks if at least one of the expressions is true. Again, *C* calculates a `1`. The difference between `&&` and `||` is illustrated by the last two interactions. Since only one expression is true (the latter expression being false) only the `||` operator yields a true value.

There is one more Boolean logic operation, called *not*. It simply reverses the value of the expression to which it is attached. The *not* operator is represented by an exclamation point (Computer Scientists call it 'bang'):

```
int b = !(3 < 4 && 4 < 5);
printf("%d\n",b);           //should print false (which is 0)
b = !(3 < 4 || 5 < 4);
printf("%d\n",b);           //should print false (which is 0)
b = !(3 < 4 && 5 < 4);
printf("%d\n",b);           //should print true  (which is 1)
```

Note that we attached *not* to each of the previous expressions involving the logical connectives. Note also that the calculated result is reversed from before in each case.

In terms of precedence, `&&` and `||` are lower than the comparison operators, which in turn are lower than the mathmatical operators.

Chapter 7

Assignment

Once a variable has been created, it is possible to change its value, or *binding*, using the assignment operator. Consider the following sequence of statements:

```
//test
int BLACK = 1;                    //BLACK created, initialized to 1
int BROWN = 2;                    //BROWN created, initialized to 2
int GREEN = 3;                    //GREEN created, initialized to 3
int eyeColor = BLACK;             //eyeColor created, initialized to 1
//calculate!
printf("%d\n",eyeColor);          //should print 1
eyeColor = GREEN;                 //assignment! eyeColor becomes 3
printf("%d\n",eyeColor == BLACK); //equality, should false: 0
printf("%d\n",eyeColor == BROWN); //equality, should false: 0
printf("%d\n",eyeColor == GREEN); //equality, should true:  1
```

Recall that the `//` character sequence starts a comment in C; those two characters and any following characters on the line are ignored.

The operator `=` (equals sign) is the *assignment* operator. The assignment operator, however, is not like the operators `+` and `*`. If one wants to add the variables:

```
a + b
```

one would take the value of *a* and the value of *b* and add those two values together. In general, for the mathematical operators, both sides of the operator are evaluated. For `=`, the variable on the left is not evaluated. If it were, the assignment:

```
eyeColor = GREEN;
```

would attempt to assign the value of 3 (the current value of *GREEN*) to the value 1 (the current value of *eyeColor*). The last three expressions given in the code fragment above refer to the equality operator. The equality operator, `==` (often pronounced "double equals"), returns true if its operands refer to the same thing and false otherwise.

In the above snippet of *C* code, we use the integers 1, 2, and 3 to represent the colors black, brown, and green. By abstracting 1, 2, and 3 and giving them meaningful names (i.e., BLACK, BROWN, and GREEN) we find it easy to read code that assigns and tests eye color. We do this because it is difficult to remember which integer is assigned to which color. Without the variables BLACK, BROWN, and GREEN, we have to keep little notes somewhere to remind ourselves what's what. Here is an equivalent sequence of statements without the use of the variables BLACK, GREEN, and BROWN.

```
//test
int eyeColor = 1;
//calculate!
printf("%d\n",eyeColor);              //should print: 1
eyeColor = 3;
printf("%d\n",eyeColor == 2);         //should print false: 0
printf("%d\n",eyeColor == 3);         //likely print true:  1
```

In this interaction, the meaning of *eyeColor* is not so obvious. We know it's a 3 at the end, but what eye color does 3 represent? When numbers appear directly in code, they are referred to as *magic numbers* because they obviously mean something and serve some purpose, but how they make the code work correctly is not always readily apparent, much like a magic trick. Magic numbers are to be avoided. Using well-named variables to hold these values is considered stylistically superior.

7.1 Lvalues vs. rvalues

To distinguish when the value of a variable is extracted and when it is updated, Computer Scientists use the terms *rvalue* and *lvalue*. An *rvalue* refers to the value of a variable while an *lvalue* refers to a location in memory. In an assignment statement such as:

```
x = y;          //update the value of x with the value of y
```

the variable y appears to the *right* of the assignment operator and is therefore an rvalue. We extract the value of y in this case. The variable x, on the other hand, appears to the *left* of the assignment operator and is therefore an lvalue. We update the value of x.

7.2 Variables versus constants

Many programming languages have constructs similar to variables known as *constants*. A constant can be thought of as a variable that, once it gets a value, cannot be reassigned. Constants are used for values that never change, the value of π, the square root of 2, and so on. The C programming language does not have these kinds of constants[1]. So C programmers use variables as constants instead. In the previous section, you probably noticed that some variables were named using all capital letters: BLACK, GREEN, and BROWN. By convention, a variable named in (mostly) all-caps is not meant to change from its initial value. The use of caps emphasizes the constant nature of the variable.

There is one type of constant C does have, the name of a statically allocated array. You will learn about statically allocated arrays in the next chapter.

7.3 Precedence and Associativity of Assignment

The assignment operator is right associative. The right associativity allows for statements like

[1]As you get more sophisticated in your C programming, you will likely take advantage of cpp, the C pre-processor. Using cpp, you can add constants to your C program.

```
a = b = c = d = 0;
```

which conveniently assigns a zero to four variables at once and, because of the right associative nature of the operator, is equivalent to:

```
(a = (b = (c = (d = 0))));
```

The resulting value of an assignment operation is the value assigned, so the assignment `d = 0` returns 0, which is, in turned, assigned to c and so on.

Assignment has the lowest precedence among the binary operators. Thus, assignment is always performed last in any expression. For example:

```
a = b = c * d
```

is equivalent to the fully parenthesized:

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

Note that expressions like:

```
a = b = c * d = e
```

are nonsensical; the multiplication has to happen first and therefore the value of e would have to be assigned to a number, which is illegal. On the other hand:

```
a = b = c * (d = e)
```

is perfectly legal and has the effect of assigning to d the value of e and to a the value of $c * d$.

7.4 Assignment Patterns

The art of writing programs lies in the ability to recognize and use patterns that have appeared since the very first programs were written. In this text, we take a pattern-based approach to teaching how to program. For the topic at hand, we will give a number of patterns that you should be able to recognize to use or avoid as the case may be.

7.4.1 The Transfer Pattern

The *transfer* pattern is used to change the value of a variable based upon the value of another variable. Suppose we have a variable named *alpha* which is initialized to 3 and a variable *beta* which is initialized to 10:

```
alpha = 3;
beta = 10;
```

Now consider the statement:

```
alpha = beta;
```

This statement is read like this: make the new value of *alpha* equal to the value of *beta*, throwing away the old value of *alpha*. What is the value of *alpha* after that statement is executed?

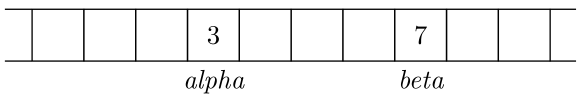
The new value of *alpha* is 10.

The *transfer* pattern tells us that value of *beta* is imprinted on *alpha* at the moment of assignment but in no case are *alpha* and *beta* conjoined in any way in the future. Think of it this way. Suppose your friend spray paints her bike neon green. You like the color so much you spray paint your bike neon green as well. This is like assignment: you made the value (color) of your bike the same value (color) as your friend's bike. Does this mean your bike and your friend's bike will always have the same color forever? Suppose your friend repaints her bike. Will your bike automatically become the new color as well? Or suppose you repaint your bike. Will your friend's bike automatically assume the color of your bike?

Let's look at the transfer pattern graphically. For the assignments:

```
alpha = 3;
beta = 7;
```

we would diagram the situation as:



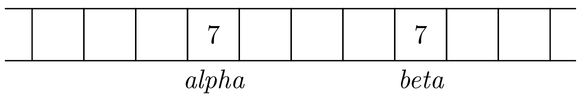
or more conveniently:

alpha: 3
beta: 7

Now, when we assign *alpha* the value of *beta*:

```
alpha = beta;
```

we replace the current value stored at the memory location identified by *alpha* with the value of *beta*:



or:

alpha: 7
beta: 7

We can see from the diagram that if we give a new value to *beta*, causing us to replace the value stored at *beta*, the value stored at *alpha* will be unaffected.

To test your understanding, consider what happens if the following code is executed:

```
alpha = 4;
beta = 13;
alpha = beta;
beta = 5;
```

What are the final values of *alpha* and *beta*? Here, the variable diagram would look like:

$$\begin{array}{rl} \textit{alpha}: & 4 \\ \textit{beta}: & 13 \end{array} \quad\rightarrow\quad \begin{array}{rl} \textit{alpha}: & 13 \\ \textit{beta}: & 13 \end{array} \quad\rightarrow\quad \begin{array}{rl} \textit{alpha}: & 13 \\ \textit{beta}: & 5 \end{array}$$

where the diagram on the left illustrates what things look like after *alpha* and *beta* get their original values, the diagram in the middle after *alpha* gets *beta*'s value, and the diagram on the right after *beta* gets the value of 5. So we can see that the final value of *alpha* is 13 and *beta* is 5.

To further test your understanding, what happens if the following code is executed:

```
alpha = 4;
beta = 13;
alpha = beta;
alpha = 42;
```

What are the final values of *alpha* and *beta*? Here, the variable diagram would look like:

$$\begin{array}{rl} \textit{alpha}: & 4 \\ \textit{beta}: & 13 \end{array} \quad\rightarrow\quad \begin{array}{rl} \textit{alpha}: & 13 \\ \textit{beta}: & 13 \end{array} \quad\rightarrow\quad \begin{array}{rl} \textit{alpha}: & 42 \\ \textit{beta}: & 13 \end{array}$$

where the diagram on the left illustrates what things look like after *alpha* and *beta* get their original values, the diagram in the middle after *alpha* gets *beta*'s value, and the diagram on the right after *alpha* gets the value of 42. So we can see that the final value of *alpha* is 42 and *beta* is 13.

7.4.2 The Update Pattern

The *update* pattern is used to change the value of a variable based upon the original value of the variable. Suppose we have a variable named *counter* which is initialized to zero:

```
int counter = 0;
```

Now consider the statement:

```
counter = counter + 1;
```

This statement is read like this: make the new value of counter equal to the old value of counter plus one. Since the old value is zero, the new value is one. Consider this sequence:

```
//test
int counter = 0;
counter = counter + 1;
counter = counter + 1;
counter = counter + 1;
counter = counter + 1;
counter = counter + 1;
printf("%d\n",counter);
```

What is the value of *counter* after the following code is executed?

The value of *counter* is 5.

There is another form of this update:

```
counter = 0;
counter += 5;
```

The operator `+=` says to update the variable on the left by adding in the value on the right to the current value of the variable.

The *update* pattern can be used to sum a number of variables. Suppose we wish to compute the sum of the variables *a*, *b*, *c*, *d*, and *e*. The obvious way to do this is with one statement:

```
int sum = a + b + c + d + e;
```

However, we can use the *update* pattern as well:

```
//test
int a = 1,b = 2,c = 3,d = 4,e = 5;
int sum = 0;
sum = sum + a;
sum = sum + b;
sum = sum + c;
sum = sum + d;
sum = sum + e;
printf("sum is %d\n",sum);
```

If *a* is 1, *b* is 2, *c* is 3, *d* is 4, and *e* is 5, then the value of *sum* in both cases is 15. Why would we ever want to use the *update* pattern for computing a sum when the first version is so much more compact and readable? The answer is...you'll have to wait until we cover a programming concept called a *loop*. With loops, the *update* pattern is almost always used to compute sums, products, etc.

7.4.3 The Throw-away Pattern

The *throw-away* pattern is a mistaken attempt to use the *update* pattern. In the *update* pattern, we use the original value of the variable to compute the new value of the variable. Here again is the classic example of incrementing a counter:

```
counter = counter + 1;
```

In the *throw-away* pattern, the new value is computed but it the variable is not reassigned, nor is the new value stored anywhere. Many novice programmers attempt to update a counter simply by computing the new value:

```
counter + 1;       // the value counter + 1 is thrown away!
```

C does all the work to compute the new value, but since the new value is not assigned to any variable; the new value is thrown away.

7.4.4 The Throw-away Pattern and Functions

The *throw-away* pattern applies to function calls as well. We haven't discussed functions much, but the following example is easy enough to understand. First we define a function that computes some value:

```
int
increment(int x)
    {
    return x + 1;
    }
```

The function *increment*, when given an integer (stored in *x*), returns a value one greater than the value of *x*. What the function actually does, however, is irrelevant to this discussion. but we want to start indoctrinating you on the use of functions. Repeat this ten times:

> We always do four things with functions: *define them*, *call them*, *return something*, and *save the return value.*

To call the function, we use the function name followed by a set of parentheses. Inside the parentheses, we place the value we wish to send to the function. Consider this code, which includes a call to the function *increment*:

```
int y = 4;
y = increment(y);
printf("y is %d\n",y);
```

If we were to run this code, we would see the following output:

```
y is 5
```

The value of *y*, 4, is sent to the function which adds one to the given value and returns this new value. This new value, 5, is assigned back to *y*. Thus we see that *y* has a new value of 5.

Suppose, we run the following code instead:

```
int y = 4;
increment(y);              // NOT y = increment(x); as before
printf("y is %d\n",y);
```

Note that the return value of the function *increment* is not assigned to any variable. Therefore, the return value is thrown away and the output becomes:

```
    y is 4
```

The variable *y* is unchanged because it was never reassigned.

7.5 About Patterns

As you can see from above, not all patterns are good ones. However, we often mistakenly use bad patterns when programming. If we can recognize those bad patterns more readily, our job of producing a correctly working program is greatly simplified.

Chapter 8

Arrays

In this chapter, we will study *arrays*, a device used for bringing related bits of data together underneath one roof, so to speak. Such a device is known as a data structure; an array is one of the most basic of data structures.

An array is composed of *slots*. Each slot can hold one piece of data. If the array has ten slots, then it can hold 10 pieces of data. Each piece of data is known as an *element*.

Depending upon the programming language, arrays can be *heterogeneous* or *homogeneous*. Orthogonally, they can be *fixed-size* or *dynamic*. The word *heterogeneous* means that a single array can hold multiple types of data. That is to say, one array could hold a string in one slot and an integer in another. Conversely, if an array is *homogeneous*, then each piece of data stored in the array must be the same type. A *fixed-size* array is given a specified number of slots at creation time; the number of slots cannot change while the program is running. On the other hand, a *dynamic* array can grow larger, if necessary, to accommodate additional data items. In C, arrays are homogeneous and fixed-sized. Note that this fixed size can be determined while the program is running, but once you make the allocation then you can't change that size at a later date.

8.1 Arrays as data structures

As stated previously, a *data structure* is simply a collection of bits of information[1] that are somehow glued together into a single whole, along with a set of operations for manipulating said bits. Usually, the bits are somehow related, so the data structure is a convenient way to keep all these related bits nicely packaged together.

At its most basic, a data structure supports the following actions:

- creating the structure
- putting data into the structure
- taking data out of the structure

We will study these actions with an eye to how long each action is expected to take.

An array is a data structure with the property that each individual piece of data can be accessed just as quickly as any of the others. In other words, the process of putting data in and taking data out takes a constant amount of time regardless of the size of the array and which slot is being used to store a data item[2].

[1]Bits in the informal sense, not zeros and ones.

[2]Not all data structures have this property of quick access to data. However, these other data structures, which you will learn about later, may have advantages over arrays in certain situations. Choosing the correct data structure for solving a problem is most of the work in crafting a solution.

8.1.1 Array creation

There are two basic ways to create an array, statically and dynamically. Note that a dynamically created array is not the same thing as a dynamic array; a dynamically created array can only be fixed-size in C.

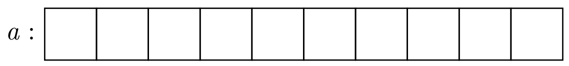
A static array lasts as long as the function in which the array was created lasts (runs). If you create a static array in `main` then it lasts for the duration of your program. Since creation of an array entails the allocation of memory somewhere to serve as slots, we refer to array creation as *array allocation*.

To allocate a static array, one uses *bracket notation*:

```
    int a[10];
```

This statement creates a *constant*, named a, that refers to 10 slots in memory. Each of the slots can hold an integer. The size of the array (*i.e.* the number of slots) is placed between the brackets that follow the variable name. The reason a is a constant and not a variable is that the value of a cannot be changed. The value of a is a particular piece of memory where the ten slots are located; that value is forever fixed while a exists.

Pictorially, we represent the constant created when an array is statically allocated like this:



We read this image as "a holds a location in memory where one can find the first slot of an array". This is a bit wordy, so we often use the less precise shorthand, "the array a".

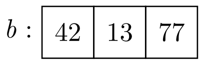
Arrays created in this manner have what might be considered random values in each of the slots. Such an array is said to be *uninitialized*. We can also *initialize* arrays at allocation time:

```
    int b[3] = { 42, 13, 77 };
```

The initialization values are comma-separated and placed between braces. It is considered "bad style" to have a mismatch between the number of slots and the number of initializers[3]. If you are bad at counting, you can have C count for you. The following statement is equivalent to the previous one.

```
    int b[] = { 42, 13, 77 };
```

In either case, we end up with a picture like this:



In the latter version, the number of slots is omitted. One must specify the number of slots, either explicitly using non-empty brackets, or implicitly, using initializers. It is an error to do neither. Thus, the statement:

[3]There is one exception. Sometimes you will see an array created like this:
`int x[512] = { 0 };`
C, if given at least one initializer, will set all slots that are missing initializers to zero. This quirk of C makes it quite easy to initialize an array to all zeros.

```
int c[];
```

will generate an error.

Remember, if you allocate an array and do not give any initializers, the slots are filled with whatever random bits are lying around in the memory allocated. Such uninitialized slots are said to be filled with garbage, just like when a variable is created but not initialized.

8.2 Indexing into Arrays

You can retrieve a value stored in an array by using *bracket notation*. With bracket notation, you specify exactly which element you wish to extract from the array. This specification is called an *index*. The first element of the array has index 0, the second index 1, and so on. The last element in an array with n slots has an index of $n-1$. This concept of having the first element having an index of zero is known as *zero-based counting*, a common concept in Computer Science. Here is some code that extracts the first element of an array. Note that the first interaction creates a *variable* named *items* that points to an array of three elements:

```
//test
int sum;
int items[3] = { 7, 11, 711 };
printf("the first slot holds %d\n",items[0]);
printf("the second slot holds %d\n",items[1]);
printf("the third slot holds %d\n",items[2]);
sum = items[0] + items[1] + items[2];
printf("the sum of the three elements is %d\n",sum);
```

The first three print statements should produce, respectively:

```
the first slot holds 7
the second slot holds 11
the third slot holds 711
```

Accessing an element in a slot does not remove the element. Thus, the last print statement yields:

```
the sum of the three elements is 729
```

In the statement:

```
sum = items[0] + items[1] + items[2];
```

the appearance of `items[0]`, `items[1]`, and `items[2]` to the right of the assignment operator means that they are rvalues. Thus, we use their values to compute the sum.

We can also use an expression involving variables as an index. The following code gives exactly the print outs as previously:

```
//test
int items[3] = { 7, 11, 711 };
int i = 0;
printf("the first slot holds %d\n",items[i]);
printf("the second slot holds %d\n",items[i+1]);
printf("the third slot holds %d\n",items[i+2]);
int sum = items[i] + items[i+1] + items[i+2];
printf("the sum of the three elements is %d\n",sum);
```

In addition to retrieving elements of an array, we can change existing elements as well, by using the assignment operator:

```
//test
int a[3] = { 7, 11, 711 };
a[1] = 13;                                  // replace 11 with 13
printf("the second element is %d\n",a[1]);  //should print: the second element is 13
```

In the statement:

```
a[1] = 13;
```

the expression `a[1]` is considered an rvalue and thus refers to a location in memory that is to be updated. As with retrieving values, we can use any arithmetic expression as an index when replacing elements.

8.3 Array indices that are out of bounds

When we use an index that is too small (i.e. a negative index) or too large (i.e. an index greater than or equal to the size of the array), we say the array index is *out of bounds*. What actually happens if we use an out-of-bounds index?

```
//test
int items[3] = { 7, 11, 711 };                   //only three slots!
printf("the fourth slot holds %d\n",items[3]);
```

We get a surprising result:

```
the fourth slot holds 134513712
```

You may get a different number printed, but you will likely not get an error message, even though an error has clearly occurred. Why is this?

C is designed to be a very fast language. Generally, a program written in C will run faster (by a stopwatch) than the equivalent program written in some other programming language. One reason for its speed is due to the fact that it does very little error checking. When an array is allocated, the slots are allocated consecutively in a block of memory. When we ask for a particular slot, C calculates how far into the block to look for the value in that slot. An index of zero refers to a slot at the beginning of the block, an index of one refers to the next slot in, and so on. Since no error checking is performed, asking for a slot beyond the last slot simply tells C to reference memory beyond the block allocated to the array. What is found there can be anything.

Behavior of a program that retrieves or changes memory that is out of bounds is undefined. When one is lucky, the program crashes immediately upon the out-of-bounds access. When one is less lucky, the program runs for a while before crashing, making the determination of what caused the crash much more difficult. If one is extremely unlucky, the program never crashes. In this last case, only thorough testing along with a very careful screening of the output reveals the existence of the error.

8.4 Arrays and aliases to arrays

In programming, an array *alias* is a second variable referring to the same array as the first. In the following code, we want to configure *b* as an alias for *a*. To do this, we will have to properly define *b*. However, for now, assume that `...` contains a definition of *b* (we'll formally define it in a few minutes). Let's see what should happen if we can establish *b* as an alias to *a*.

```
double a[3] = { 2.7, 3.1, 1.4 };
...
b = a;
```

If we were to change an element using *b*, we would see the new element using *a*:

```
double a[3] = { 2.7, 3.1, 1.4 };
...
b = a;
b[0] = 1.61;
printf("does a[0] == b[0]? %d\n",a[0] == b[0]);
```

The answer we would get is:

```
does a[0] == b[0]? 1
```

Recall that a non-zero integer means true, so the 1 following the question mark means the equality test evaluated to true: the first element of *a* and the first element of *b* are the same.

OK, now that you see how aliasing works, we need to figure out the proper technique for defining *b*. What is missing from the above code fragments is how to create the variable *b*. One would think we would have to create *b* as the same kind of array as *a*:

```
double a[3] = { 2.7, 3.1, 1.4 };
double b[3];
b = a;
```

After all, it is only logical that *a* and *b* should have the same type if we are going to have them point to the same thing. Unexpectedly, we get the following error message when we compile:

```
error: incompatible types when assigning to type 'double[3]' from type 'double *'
```

Why this error occurs will become clear in time, but for now, remember this rule:

For a statically allocated array, you cannot reassign its name.

If you declared a statically allocated array, it has a fixed location. You can't move it to a new location. The statement `b = a` was trying to move a statically allocated array.

If we look at the error message carefully we see that we are `assigning to type 'double[3]'`. This makes sense because clearly the type of *b* is an array of three doubles. What is strange is the next part, `from type 'double *'`. Somehow, when we use the name of an array on the right-hand side of an assignment, its type morphs. Let's try again, changing the type of *b* to `double *`, the type that *a* appears to have become:

```
//test
double a[3] = { 2.7, 3.1, 1.4 };
double *b;
b = a;
b[0] = 1.61;
printf("does a[0] == b[0]? %d\n",a[0] == b[0]); //should print true: 1
```

We see now that the program compiles correctly and, indeed, *b* appears to be an alias of *a*. Especially note that we use the same syntax to index a slot an array, whether we use the array name or a variable that has been assigned the array name.

So why the `double *`? In C, when an asterisk appears in a type, it means a variable of that type will hold the address of a memory location. So a variable of type `double *` will hold the address of a memory location that stores a real number. Recall that, in our example, *a* is a memory location where the first slot of an array of doubles can be found. So if we want to store this memory location in another variable, the type of the other variable must be `double *`. Of course, if *a* instead referred to a statically allocated array of integers, then the type of *b* would have to be `int *`.

A variable that holds a memory location *and* is of a type containing an asterisk is known as a *pointer*. In our example, both the constant *a* and variable *b* hold memory locations. We call the constant *a* a *pseudopointer* since its type is `double [3]`, not `double *`. The variable *b*, on the other hand, has type `double *`, so it is a true pointer. In all but a very few situations, pseudopointers are converted to pointers, so it is convenient to think of the constant *a* as a pointer. Assignment is one situation where the difference between pointers and pseudopointers becomes apparent . If we try to assign to *a*, the compiler rejects our attempt:

```
//test
double a[3] = { 2.7, 3.1, 1.4 };
double *b;
b = a;                  // OK, can assign a pseudopointer to a pointer
a = b;                  // rejected, cannot assign to a pseudopointer
```

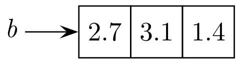
The last line generates the error:

```
error: incompatible types when assigning to type 'double[3]' from type 'double *'
```

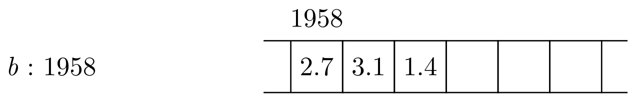
just like we saw before. When we refer to a pseudopointer as a value, the compiler converts it to a pointer (as in the next to the last line). When we refer to a pseudopointer as a location, in order to reassign it (as in the last line), the compiler says no.

Other than assignment and one other difference, discussed in the next section, it is generally OK to think of a pseudopointer as a pointer. But don't be surprised by what happens if you don't keep these two exceptions in mind.

Graphically, we represent a pointer with an arrow. So *b*, from our example, looks like this:



Suppose the array to which *a* refers starts at memory location 1958. Then, an alternate view of *b* would be:



In fact, you can see where *a* and *b* point by using the `%p` directive in the control string for *printf*:

```
//test
double a[3] = { 2.7, 3.1, 1.4 };
double *b;
b = a;
printf("a points to %p\n",a);
printf("b points to %p\n",b);
```

Running this code will yield something like:

```
a points to 0x96b0570
b points to 0x96b0570
```

Whatever memory locations are printed, they should match. We can also see that the `%p` directive causes the pointed-to location to be printed out in hexadecimal (hexadecimal numbers in C begin with `0x`). Note that from now on, we will use the terms *memory location* and *address* interchangeably. Thus, we will often say things like "variables *a* and *b* hold the address of the array".

8.5 Determining the length of an array

One can compute the length of a statically allocated array using the *sizeof* operator:

```
//test
double a[3] = { 2.7, 3.1, 1.4 };
printf("the size of a is %zu\n",sizeof(a));
```

Depending on your system, it yields something like:

```
the size of a is 24
```

This appears to be incorrect; shouldn't the size of *a* be three? There are three elements in *a*, but that is not the same thing as asking how much total size the array *a* takes in memory.

The *sizeof* operator is working correctly, but it is not returning the number of slots in the array. Instead, it is returning the number of bytes[4] allocated for the array. If one knew how many bytes made up a double, we could compute the number of slots. Luckily, the *sizeof* operator can tell us the number of bytes each double slot requires. By dividing the size of the array (in bytes) by the size of a slot (in bytes), we get the number of slots:

[4]A byte is eight bits. An integer is usually 4 or 8 bytes, depending on the system, while a double is usually 8 bytes.

```
//test
double a[3] = { 2.7, 3.1, 1.4 };
printf("the size of a is %zu\n",sizeof(a)/sizeof(double));
```

Now we get the desired result:

```
the size of a is 3
```

You likely have noticed a new format specifier in the guide string in the above example: `%zu`. The *sizeof* operator returns a 64-bit integer on 64-bit computers and a 32-bit integer on 32-bit computers. The specifier for 32-bit integers is `%d`, as we have seen before, but the specifier for 64-bit integers is `%ld`. In order to write portable code that runs on both kinds of computers, we use the generic `%zu` specifier that can hand both kinds of integers.

Two quick questions for you at this point. First, looking at the example above, it should be easy to modify that code so that you could figure out how much space an array of 50 integers required. Can you do that? Second, is this trick (sizeof the array divided by sizeof each element) required for strings (arrays of characters)? Why or why not?

We can simplify the process of finding the number of slots by using a *macro*:

```
//test
#define SLOTS(t) (sizeof(t)/sizeof(*t))
double a[3] = { 2.7, 3.1, 1.4 };
printf("the size of a is %zu\n",SLOTS(a));
```

You can read more about C macros on the web. When we run this code, as before, we get the desired result:

```
the size of a is 3
```

If we try to use *sizeof* to find the size of an array through an alias:

```
//test
#define SLOTS(t) (sizeof(t)/sizeof(*t))
double a[3] = { 2.7, 3.1, 1.4 };
double *b = a;
printf("the size of a is %zu\n",SLOTS(b));
```

On a 32-bit machine, we get an incorrect answer:

```
the size of a is 0
```

The reason is because *sizeof*, when applied to a true pointer variable, returns the size of the variable, not the size of memory allocated. In this case, `sizeof(double *)` returns how many bytes it takes to store an address of a double. We can deduce that the size of a `double *` is less than the size of a `double`, on a 32-bit machine. Using integer division, we get a zero result.

8.6 Pointers and indexing

Luckily, one can use the same notation to index into an array, whether using the name of the array or a pointer to the array. In fact a pseudopointer to an array is immediately converted into a pointer nearly every time we reference an array name.

When we refer to a particular slot in an array, either through the array name or a pointer, as in:

```
a[5]
```

a number of computations are made in finding the desired slot. First, if the array name is used, it is converted into a pointer. Second, the address stored in the pointer is extracted. This address is known as the *base address* and is the address of the first slot. Then, the value of the index, which refers to a slot number, is converted to an offset. The offset reflects how many bytes past the base address the desired slot resides. Suppose a, in the example above, points to an integer array. Then slot five is located `5 * sizeof(int)` bytes past the base address. If the array of integers starts at memory location 1000 and an integer is 4 bytes, then slot 5 is located at memory location $1000 + 5 * 4$ or 1020.

You can also use pointer offsets directly to access a slot in an array. These two statements are equivalent:

```
array[5] = 0;
*(array + 5) = 0;
```

The latter form is known as *star-notation* or *dereferencing*. In fact, C converts array access using brackets to star notation in computing offsets to the base address.

The following four statements are also equivalent:

```
array[0] = 5;
*(array + 0) = 5;
*(array) = 5;
*array = 5;
```

In fact, `*array` is often used as a short hand to the first element of an array, especially in dealing with arrays of characters. This form is known as *pointer dereferencing*.

8.7 The evils of statically allocated arrays

Statically allocated arrays suffer from two problems. The first is that they cannot be returned from the function that created them. We haven't covered functions yet, but the following function definition is easy enough to understand:

```
double *
newArrayOfThreeDoubles()
    {
    double a[3] = {0.0};
    return a;
    }
```

The function allocates a static array that can hold three doubles; each slot has been initialized to zero. As its final act, the function returns the address of this statically allocated array.

If we try to compile this function, we get the following warning:

```
    warning: function returns address of local variable [-Wreturn-local-addr]
```

The problem is the memory allocated to the array is expected to be of use while the function is executing; after the function returns, that memory is free to be reused. Once the memory was reused, the caller of the function now has an array whose slots appear to change at random. Back in the day, such warnings were only pipe dreams, so a program that contained such code would be happily compiled, so consider yourself lucky you live in an age where the compiler looks after you so well.

The second problem with static arrays is that their misuse accounts for a vast majority of the security holes found in the past and many of the security holes currently being uncovered. The reason is the C programming language is quite popular for writing the guts of software systems, since it is so fast. The reason it is so fast is that C does not check for out-of-bounds errors with regard to accessing array elements. It turns out that a large amount of the C code extant has been written by people not properly trained in Modern Computer Science theory and practice. The result is a perfect storm of vulnerable code, illustrated by the following, rather simple, code fragment:

```
    char buffer[512];                //create a static array of char
    printf("What is your name? ");   //print a prompt for the user
    scanf("%s",buffer);              //read the response into the array
```

What could possibly be wrong with this code? Surely, no one has a name longer than 511 characters? This issue, known as the *buffer overflow problem*, will be discussed further in the chapter on input and output.

8.8 Dynamically allocated arrays

We have discussed statically allocated arrays at length because they are the easiest version of arrays for beginners to learn. Yet, as a programmer gets more and more sophisticated, statically allocated arrays appear less and less in the programs he or she writes. This is mainly due to the two dangers listed in the previous section. Dynamically allocated arrays solve the first problem, returning a local array from a function.

Dynamically allocated arrays exist for as long as the program runs, regardless of when or where they are allocated. Here is a template for dynamically allocated array. In this code, we are allocating an array of type `XXXX` with `YYYY` slots. The system call that finds a block of space and allocates that space is *malloc*. Note that *malloc* is configured to return a zero when it is unable to allocate the requested memory. You should always check to make sure that any *malloc* you perform actually succeeds.

```
    XXXX *a;

    a = malloc(sizeof(XXXX) * YYYY);
    if (a == 0)
        {
        fprintf(stderr, "memory allocation failed\n");
        exit(1);
        }
```

In the template, `XXXX` is replaced by a type, say `int` or `double` or `char *`. `YYYY` is replaced by the number of slots desired.

If we wish for an integer array with eight slots, we fill out the template like this:

```
int *a;

a = malloc(sizeof(int) * 8);
if (a == 0)
    {
    fprintf(stderr, "memory allocation failed\n");
    exit(1);
    }
```

The function *malloc* does the actual allocation and it needs to know the number of bytes required. We compute that quantity with the *sizeof* operator, which we use to give us the number of bytes needed by one slot. We multiply that by the number of slots to get the total number of bytes. Note that arrays allocated in this manner are uninitialized; their slots are filled with garbage.

Alternatively, we can ask the *sizeof* operator to compute the size of what a pointer points to:

```
int *a;
a = malloc(sizeof(*a) * 8); //equivalent to malloc(sizeof(int) * 8);
```

If the memory cannot be allocated by *malloc* (maybe you're asking for too many slots), the function returns zero.

We haven't talked about *if* statements yet, but the code following the allocation is easy enough to read. If the *malloc* function returns a zero rather than an address, we print an error message and exit the program. Recall that if *main* returns a non-zero value, an error is assumed. The *exit* function mimics *main* returning immediately with the given value.

One other note on the above example. The `fprintf` statement prints to *stderr*, which is by default associated with the console. There are lots of cases where you might direct regular output (*stdout*) to a file rather than the console. By writing to *stderr*, we can see error messages on the console even if the regular output from a program is being captured elsewhere.

Once the array has been successfully allocated, you can use indexing just like with statically allocated arrays, both for retrieving and changing elements in the array. Moreover, the array will last until the program ends or until you free up the memory allocated to be used again. To free the array, you simply call the *free* function:

```
free(a);
```

Be careful when you free things. You can only free things that have been allocated with `malloc` (or similar functions), you may not free the same thing more than once, and you may not refer to slots in an array you have previously freed.

You have probably heard of *memory leaks*. An example of a memory leak is a situation where a program that runs for a long time (when is the last time you turned your smartphone off?) consumes new memory as it processes requests but never releases those dynamically allocated resources. Eventually, the system runs out of space to allocate additional resources (*malloc* returns a zero). The issue of memory leaks is critical for applications that run for any amount of time. While you don't have to worry about memory leaks for `project0`, this is a major issue for the writers of any modern operating system.

Finally, for completeness, we note that the second problem with statically allocated arrays, the buffer overflow problem, can be solved with dynamically allocated arrays. You can use dynamically allocated arrays to simulate dynamic arrays (those arrays that grow as needed). We will cover this technique in a later chapter.

There is a lot going on in this section that you likely don't understand at this point and that's OK. You should, however, be able to use this template if you are required to use dynamically allocated arrays in a program.

8.9 Pointers to static arrays

We have already discussed pointers to arrays of one dimension. Here is an example:

```
int a[10];    //a is a one-dimensional array
int *p = a;   //p points to the first slot of a
```

Although, we will not discuss them in detail until later, it is possible to have multi-dimensional static arrays:

```
int b[10][12];          //a two-dimensional array, 10 rows and 12 columns
int c[50][60][40];      //a 3D array, 50 sheets, 60 rows, and 40 columns
int d[11][13][17][19];  //a 4D array, 11 ledgers, 13 sheets, 17 rows, and 19 columns
```

Based upon the array a and the pointer p above, one might think that a pointer to b would look like this:

```
int b[10][12];
int **q = b;        //compiler warning, types do not match
```

Instead, we get a compiler warning. This is because when the pseudopointer b devolves into a pointer, only the first dimension becomes a pointer; the remaining dimensions remain arrays. The correct type for q would be:

```
int b[10][12];
int (*q)[12] = b;
```

The parentheses are required since the brackets have a higher precedence than than the asterisk. Without the parentheses, q would be a one-dimensional array of integer pointers. With them, it is a pointer to a one-dimensional array of integers with length 12. Thus, the variable q points to the first set of 12 integers, while the expression $q + 1$ points to the second set of 12 integers. The expression $q + 2$ points to the third set of twelve integers, and so on.

Pointers r and s, which are to point to arrays c and d above, respectively, would be defined as follows:

```
int c[50][60][40];
int (*r)[60][40] = c;

int d[11][13][17][19];
int (*s)[13][17][19] = d;
```

Remember, when an array name devolves into a pointer, the type of that pointer can be generated by removing the first set of brackets and adding the asterisk and parentheses in its stead.

The reason for this behavior is as follows. Consider our example of a two dimensional array:

```
    int b[10][12];          //10 rows, 12 columns
    int (*q)[12] = b;
    int **z = b;            //not really valid, but likely will compile
```

The expression `q[0]` refers to the first set of 12 integers, while `q[1]` refers to the second set of 12, and so on. Thus `q[1][2]` refers to the third integer in the second set of 12, as expected. C needs to know how many integers it needs to skip over to move from row 0 to row 1. The answer is twelve and can be found in the types of both b and q. On the other hand, that information is missing in z and computations invoving elements of z invariably go wrong.

Chapter 9

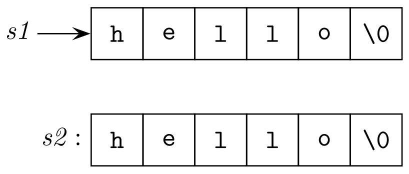
Strings

As stated in a previous chapter, strings in C are really arrays of characters. For example, the two variables *s1* and *s2* are very similar:

```
//test
char *s1 = "hello";
char s2[6] = { 'h', 'e', 'l', 'l', 'o', '\0' };
printf("%s\n",s1);
printf("%s\n",s2);
```

Both print statements produce the word `hello`. The difference is that *s1* is a true pointer while *s2* is a pseudopointer. We can tell this by the types of *s1* and *s2*. The variable *s1* has a type `char *`, which means it holds an address of a memory location that stores a character. The variable *s2* has the type `char[6]`. Since *s1* is a true pointer, it can point to any number of different characters. The program above points it to the first character in the literal `hello` but that can be updated and point to different locations as the program executes. On the other hand, *s2* will always refer to (first of) the six characters that were allocated to it. While the characters might change, the location that *s2* references will never change.

A strange thing in the code above is that *s2* is an array of six characters while the counting the characters between the quotes in the initializer for *s1*, `"hello"`, yields five. It appears the sixth slot of *s2* (at index 5, remember your zero-based counting!) is initialized to a rather funny looking character `'\0'`. This character is sometimes pronounced *backslash-zero* or, more often, the *null character* or the *null byte.* In C, strings are terminated by the null character, so both *s1* and *s2* in our example point to arrays of six slots:



For *s1*, the C compiler adds on terminating null characters to literal strings such as `"hello"`. When we build a string with an array ourselves, we are in charge of getting things right. We must be mindful not to forget to both make room for the null character and place it into the array.

Note: you can safely allocate a static array of characters, making sure there is room for the null character with initializations like:

```
char s3[] = "hello";
```

When you do this, the C compiler will determine the length of the string and allocate space for all the characters in the string plus the *null* character.

9.1 Printing strings

When we wish to print a string, we use the `%s` format directive in the guide string:

```
    //test
    char *s = "The integer directive is %d";
    printf("%s\n",s);              //should print: The integer directive is %d
```

Often, we wish to build a guide string and use it in a print statement:

```
    //test
    int z = 5;
    char *s = "The result is %d\n";
    printf(s,z*z);                 //should print: The result is 25
```

From this example, we can see that *printf* (indeed, all functions) are quite flexible in terms of what kinds of data we can send it. For the guide string, we can send a literal string, a character pointer, or an array of characters. Of course, with the latter two, the array of characters has to be terminated with the null character.

One difference between literal strings and arrays of characters is that one can change a character in an array of characters using bracket notation. For literal strings, attempting to change a character will cause undefined behavior. For example:

```
    char *r = "rat";                    //r points to a literal string
    char s[] = { 'r', 'a', 't', '\0'};  //s pseudopoints to an array of char
    char *t = r;                        //t points to a literal string
    char *u = s;                        //u points to an array of char

    r[0] = 'c';     //BAD
    s[0] = 'c';     //OK
    t[0] = 'c';     //BAD
    u[0] = 'c';     //OK
```

From the code above, we can see it doesn't matter if we use the name of an array or a pointer to the same array; we are allowed to change the contents of the array. Likewise, we are *not* allowed to change the contents of a literal string, regardless of how we access it.

9.2 Determining string length

C has a built-in function for determining the length of a string, *strlen*. The *strlen* function returns the number of characters in the given string, *not counting the null character.* For example,

```
    //test
    char s[6] = { 'w', 'o', 'r', 'l', 'd', '\0' };
    char *t = "world";
    //calculate!
    printf("the length of s is %zu\n",strlen(s));
    printf("the length of t is %zu\n",strlen(t));
```

The output of the above code is:

```
    the length of s is 5
    the length of t is 5
```

There are actually six characters in *s* and *t*, so when using *strlen* to compute the length of the string, remember to add in one more to account for the null character.

A quick question for you regarding *strelen*. In the example above, what character does `s[strlen(s)]` reference? Is it the last character in the array ('d') or the `null` character? Since arrays start counting at zero, it is always important that you account for this when using *strlen*.

You likely noticed the use of the `%zu` format specifier in the guide string to look at the return value of *strlen*. Like *sizeof*, *strlen* returns an integer on a 32-bit machine and a long(er) integer on a 64-bit machine.

In order for *strlen* to work properly on character arrays, they must be terminated by the null character. Consider this code:

```
    char s[5] = { 'w', 'o', 'r', 'l', 'd' };
    printf("the length of s is %zu\n",strlen(s));
```

The result of running the *strlen* function in this case is undefined. This is because *strlen* will count characters until it encounters the null character. Since there is no null character specified, *strlen* will cruise through memory until it finds one accidentally. If it finds one, it will report an erroneous length, If it doesn't find one, your program will crash.

9.3 Copying strings

Sometimes it is useful to copy a string. The following code does not copy. Rather, it sets up an alias:

```
    //test
    char *s = "rat";
    char *t = s;                //t is an alias for s
    printf("%s\n",t);           //should print: rat
```

To make a copy, we need to allocate an array and then use the *strncpy* function to copy the string into the array. The *strncpy* function takes an array that receives the characters being copied, the string to be copied, and an integer that limits the number of characters to be copied. Here is an example:

```
    //test
    char *s = "rat";
    char t[4];
    //calculate!
    strncpy(t,s,4);         //magic number, s has four characters!
    t[0] = 'c';             //OK, t is an array
    printf("%s\n",s);       //should print rat
    printf("%s\n",t);       //should print cat
```

Here the output of the first print statement is `rat`, showing that changing the first character of *t* had no effect on *s*. The second print statement produces `cat`, indicating that the copy performed by *strncpy* worked.

One common mistake made with this kind of code is to specifiy one less character than necessary in specifying the number of characters to be copied, i.e., forgetting to count the null character. A better version uses *strlen*:

```
strncpy(t,s,strlen(s)+1);   //add 1 for the null byte
```

A second common mistake is to allocate the destination array with too few slots. A safer version of this code copies as many characters as allowed, then writes a null byte in the last slot of array:

```
strncpy(t,s,SLOTS(t));
t[SLOTS(t)-1] = '\0';
```

Even this code is problematic, in that if *t* is too small, the string in *t* will be a prefix of the string in *s*:

```
//test
#define SLOTS(a) (sizeof(a)/sizeof(*a))
char *s = "rat";
char t[3];              //t is too small!
//calculate!
strncpy(t,s,SLOTS(t));
t[SLOTS(t)-1] = '\0';
printf("%s\n",s);       //should print rat
printf("%s\n",t);       //should print ra
```

Instead of using a statically allocated array to hold a string, we can use a dynamically allocated array:

```
//test
char *s = "rat";
char *t = malloc(strlen(s) + 1); //remember the null character!
// check to see that malloc was successful
if (t == 0)
    {
    fprintf(stderr,"allocation of t failed\n");
    exit(1);
    }
//calculate!
strncpy(t,s,strlen(s) + 1);
t[0] = 'c';             //OK, t points to an array
printf("%s\n",s);       //should print rat
printf("%s\n",t);       //should print cat
```

The use of dynamic allocation solves all our problems!

One final note on copying strings. We used the `strncpy` routine and specifically watched how many characters were being copied. There is also a `strcpy` routine that simply copies a string without doing any checking to make sure you have the space necessary to make the copy. Since one of the easiest ways to get in trouble with C programming is to overwrite something, we recommend you always make sure you have room to store whatever it is that you are copying.

9.4 Comparing strings

Can one string be "less than" another, in the same way that one number can be less than another? It turns out the answer is yes. In fact, the ability to say one string is less than another allows us to quickly look up words in a dictionary[1]. In a dictionary, the word *aardvark* comes before *zebra*, because the letter *a*, which starts aardvark, comes before *z*, which starts zebra, in the alphabet.

If we were to ask:

```
//test
char *w1 = "zebra";
char *w2 = "aardvark";
printf("%d\n",w2 < w1);     //may print true: 1, but likely false: 0
```

the output may surprise us. This is because we did not ask whether *w2* comes before *w1* in the dictionary. Recall that *w1* and *w2* are pointers. Thus, the question that we asked was is the address held in *w2* less than the address held in *w1*.

We have a special purpose operator/function for answering dictionary questions. It is called *strcmp*. When given two strings, *strcmp* returns 0 if they are composed of the same characters in the same order, a negative integer if the first string precedes the second in a dictionary, and a positive integer if the first string follows the second:

```
//test
char *w1 = "zebra";
char *w2 = "aardvark";
printf("%d\n",strcmp(w2,w1) < 0);       //should print true: 1
```

We have thrown around the term *dictionary* rather cavalierly. C's dictionary ordering is a bit different than what we are used to. For example, in C's alphabet, a capital *Z* comes before a lower-case *a*. Therefore, when we capitalize *Zebra*:

```
//test
char *w1 = "Zebra";
char *w2 = "aardvark";
printf("%d\n",strcmp(w2,w1) < 0);       //should print false: 0
```

we get the opposite result from before.

Several chapters ago, we mentioned an `ASCII` table for characters. This table is what C uses for its ordering. If you find this table online, you can easily see why a capital-Z (`ASCII` value 90) is considered less than a lowercase-a (`ASCII` value 97).

9.5 Converting strings to numbers

The function *atoi* can convert a string representing an integer to an actual integer:

```
//test
```

[1]Back in the day, when we wanted to look up the meaning of a word, we grabbed a dictionary that had been stored in the form of a *book*. A book was a device made of something called *paper*, which, more often than not, was made of pine trees.

```
int i;
char *s;
//calculate!
s = "123";
i = atoi(s);
printf("s+1 is %s\n",s+1);   //should print 23
printf("i+1 is %d\n",i+1);   //should print 124
```

The first print statements prints 23 because `s+1` skips over the first character. The second print 124 because *atoi* converts the string `"123"` into the integer 123. Adding one to the integer increments the value to 124. If you give *atoi* a string that doesn't look like an integer, it will return 0.

The *atof* function can be used to convert a string into a real number.

```
double r = atof("3.14159");
```

Like *atoi*, if you give *atof* a string that does not look like a number, it will return 0.0;

Chapter 10

Functions

Recall from Chapter **??** the series of expressions we evaluated to find the *y*-value of a point on the line:

```
y = 5x - 3
```

First, we assigned values to the slope, the *x*-value, and the *y*-intercept:

```
int m = 5;
int x = 9;
int b = -3;
```

Once those variables have been assigned, we can compute the value of *y*:

```
int y = m * x + b;
```

Now, suppose we wished to find the *y*-value corresponding to a different *x*-value or, worse yet, for a different *x*-value on a different line. All the work we did would have to be repeated. A *function* is a way to encapsulate all these operations so we can repeat them with a minimum of effort.

10.1 Encapsulating a series of operations

First, we will define a not-too-useful function that calculates *y*, given a slope of 5, a *y*-intercept of -3, and an *x*-value of 9 (exactly as above). We do this by wrapping a function around the sequence of operations above. The return value of this function is the computed *y* value:

```
int
getY(void)
    {
    int m = 5;
    int x = 9;
    int b = -3;
    return m * x + b;
    }
```

There are a few things to note. A function definition begins with a type such as `int`, `double`, or `char *`. This type is known as the *return type*. In our example, the return type is *int*, meaning the function we are defining will return an integer. Following the return type is the name of the function; the name of this particular function is *getY*. The names of the things being sent to the function are given between the parentheses. If the function does not need any thing sent to it, we use the keyword `void` to signify this fact. Together, the return type, the name of the function, and the information about what needs to be sent to the function is known as the *function signature*.

The stuff indented from the function signature is called the *function body* and is the code that will be evaluated (or executed) when the function is used. You must remember this: *the function body is not evaluated until the function is actually used.*

While in many languages the name of a function is a variable, in C the name of a function is a constant. It is, in fact, a pseudopointer, much like the name of a statically allocated array. Thus we cannot reassign *getY*, but we can create a pointer that points to what *getY* points to. This thing that *getY* points to is known as a *function object* or a *closure*. A true pointer that points to a function object is known as a *function pointer*. We will discuss function pointers in a later chapter.

Here is a program that uses our function:

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

int getY(void);                     //function declarations go here

int
main(int argc,char **argv)
    {
    int result;

    result = getY();                //function call

    printf("getY returned %d\n",result);

    return 0;
    }

int                                 //function definitions go here
getY(void)
    {
    int m = 5;
    int x = 9;
    int b = -3;
    return m * x + b;
    }
```

There are a number of points to go over with respect to this program.

1. The first is we place all the signatures of the functions we write near the top of the file. The signatures must precede any calls (or uses) of the function. A function signature by itself is known as a *declaration*, as it tells the compiler that the function exists and is defined elsewhere. Another name for a function declaration is *prototype*.

2. Second, to call a function, we give the name of the function followed by a parenthesized list of the things we want to send to it. Since the function *getY* needs nothing, we send nothing, signified by the empty parentheses.

3. Third, we place our function definitions after the definition of *main*.

This placement of function definitions is not a hard and fast rule. We could have written our program like this:

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

int                                 //function definitions go here
getY(void)
    {
    int m = 5;
    int x = 9;
    int b = -3;
    return m * x + b;
    }

int
main(int argc,char **argv)
    {
    int result;

    result = getY();                //function call

    printf("getY returned %d\n",result);

    return 0;
    }
```

In this version, there is no need for separate function declarations since definitions include function signatures. Which style you use is a matter of choice.

One final comment on the function declaration lines. The use of these declarations is strongly encouraged. Among other things, you will eventually encounter situations where they are mandatory. For example, consider a *main* routine that calls functions *foo* and *bar*. During execution, *foo* will occasionally make calls to *bar* for computations, and `bar` might also have to make a call to *foo*. It is impossible to identify an order to define *main* and *foo* and *bar* that the C compiler will accept without the use of function declaration lines.

10.2 Passing arguments

The *getY* function, as written above, is not too useful in that we cannot use it to compute similar things, such as the y-value for a different value of x. This is because we "hard-wired" the values of b, x, and m in the definition of the function.

A hallmark of a good function is that it lets you compute more than one thing. We can modify our *getY* function to *take in* the value of x in which we are interested. In this way, we can compute more than one value of y. We do this by *passing* in some information. This information that is passed to a function in a function call is known as an *argument*[1], in this case, the value of x:

```
int
getY(int x)
    {
    int slope = 5;
    int intercept = -3;
    return slope * x + intercept;
    }
```

Note that we have moved the variable x from the body of the function to between the parentheses. We have also refrained from giving it a value since its value is to be sent to the function when the function is called. What we have done is to *parameterize* the function to make it more general and more useful. The variable x is now called a *formal parameter* since it sits between the parentheses in the first line of the function definition.

Now we can compute y for an infinite number of x's:

```
int
main(int argc,char **argv)
    {
    int result;

    result = getY(9);
    printf("getY(9) returned %d\n",result);     //should be: 42

    result = getY(0);
    printf("getY(0) returned %d\n",result);     //should be: -3

    result = getY(-2);
    printf("getY(-2) returned %d\n",result);    //should be: -13

    return 0;
    }
```

What if we wish to compute a y-value for a given x for a different line? One approach would be to pass in the *slope* and *intercept* as well as x:

```
int
getY(int x,int m,int b)
    {
    int ans = m * x + b;
    return ans;
    }
```

If we wish to calculate using a different line, we just pass in the new *slope* and *intercept* along with our value of x. This certainly works as intended, but is not the best way. One problem is that we keep on having

[1]The information that is passed into a function is collectively known as *arguments*. The arguments are then bound to the variables or *formal parameters* that are found after the function name in the function definition.

to type in the slope and intercept even if we are computing y-values on the same line. Anytime you find yourself doing the same tedious thing over and over, be assured that someone has thought of a way to avoid that particular tedium. If so, how do we customize our function so that we only have to enter the slope and intercept once per particular line? We'll have to postpone these thoughts until we learn about another data structure called *objects*.

You should note that, in the last version of our `getY` function, we introduced a local variable *ans*. We calculated an answer in *ans* and then that value was returned to the calling routine. The variable *ans* 'disappears' when the function quits. That is, the space that was allocated for it is freed and made available for other uses.

In any case, the things you should take away, so far, about functions are:

- functions encapsulate calculations
- functions can be parameterized
- functions are defined so that they may be called
- any statically declared variables in the function are destroyed/released when the function completes
- functions return values

This last point is very important. Whoever calls a function needs to handle the return value either by assigning it to a variable or by passing it immediately to another function (nested function calls). Here is an example of the former:

```
y = square(x);
z = square(y);
```

and here is an example of both the former and the latter:

```
z = square(square(x));
```

Both approaches yield identical results.

10.3 The Function and Procedure Patterns

When a function calculates (or obtains) a value and returns it, we say that it implements the *function* pattern. If a function does not have a return value, we say it implements the *procedure* pattern.

The *square* function mentioned in the previous section is an example of the *function* pattern:

```
int
square(x)
    {
    return x * x;
    }
```

This function takes a value, stores it in x, computes the square of x and returns the result of the computation.

In contrast, here is an example of the *procedure* pattern:

```
void
displayLine(int m,int b)
    {
    printf("y = %dx + %d",m,b);
    return;
    }
```

We use the `void` return type to indicate procedures. Also, we give no value for the return.

Almost always, a function that implements the *function* pattern does not print anything, while a function that implements the procedure pattern often does[2]. A mistake often made by beginning programmers is to print a calculated value rather than to return it. So, when defining a function, you should ask yourself, should I implement the function pattern or the procedure pattern?

Most of the functions you will implement in this class follow the function pattern.

10.4 The dangers of functions and statically allocated arrays

A common mistake mistake made by beginners to have a function define a statically allocated array and then attempt to return the array:

```
int *
bundle3(int a,int b,int c)
   {
   int bundle[3];
   bundle[0] = a;
   bundle[1] = b;
   bundle[2] = c;
   return bundle;
   }

...
int *p = bundle3(x,y,z); //the returned array's memory may be reused!
```

Modern C compilers will flag this attempt to return *bundle*. The reason this is a bad idea is, once the function returns, the memory reserved for the statically allocated array is reclaimed by the system. Remember that we had mentioned above that any statically declared variables are released/destroyed when a function completes.

While the "array" *p*, in the example above, may appear normal for a while, once that memory allocated to *bundle* is reused, the elements of *p* will be corrupted. Note that this memory reuse only affects arrays statically allocated within a function; arrays statically allocated outside of functions last the lifetime of the program.

The proper way to implement the *bundle3* function is to dynamically allocate the array:

```
int *
bundle3(int a,int b,int c)
   {
   int *bundle = malloc(sizeof(int) * 3);
   //check for malloc failure omitted
```

[2]Many times, the printing is done to a file, rather than the console.

```
    bundle[0] = a;
    bundle[1] = b;
    bundle[2] = c;
    return bundle;
    }

...
int *p = bundle3(x,y,z); //the returned array's memory will NOT be reused!
```

Our function now returns a pointer to a location that contains three integers. Since that location was allocated dynamically, it remains available after the function has completed. The user can now reference the three values that were placed in *bundle*. However, the user should also remember to `free` that memory when it is no longer needed. Otherwise, repeated calls to this function will result in a memory leak that could cause issues over time.

Chapter 11

More about Functions

We have already seen some examples of functions, some user-defined and some built-in. For example, we have used the built-in functions, such as `printf` and defined our own functions, such as *square*. C has many built-in, or *predefined*, functions. No one, however, can anticipate all possible tasks that someone might want to perform, so most programming languages allow the user to define new functions. C is no exception and provides for the creation of new and novel functions. Of course, to be useful, these functions should be able to call built-in functions as well as other programmer created functions.

For example, a function that determines whether a given number is odd or even is not built into C but can be quite useful in certain situations. Here is a definition of a function named *isEven* which returns true (integer 1) if the given number is even, false (integer 0) otherwise:

```
int
isEven(int n)
    {
    return n % 2 == 0;
    }
```

Even though this function definition is very short, there is a lot going on. If you ever take a course on compilers, you will learn how this code is turned into language your computer understands. We, however, will stay on a higher plane. First, we will talk about the purpose of a function definition. Later, we'll talk about the syntax of a function definition. Finally, will talk about the mechanics of a function definition and a function call.

11.1 The purpose of functions

C programming is all about functions. We define a function to do some task. This function calls other functions, some that are built-in and some that are not. For those that are not, we define those as well. These newly defined, lower-level functions, in turn, call more functions, some that are built-in and some that are not. And so it goes, until our lowest-level functions call only built-in functions or functions we have previously defined.

Even in higher-level languages, such as Java, the same process holds as well. So learning to be comfortable with defining and calling functions is paramount to being a good programmer and to being a good Computer Scientist. Consider functions the language of Computer Scientists.

11.2 Function syntax

Recall that the words of a programming language include its primitives, keywords and variables. A function definition corresponds to a sentence in the language in that it is built up from the words of the language. And like human languages, the sentences must follow a certain form. This specification of the form of a sentence is known as its *syntax*. Computer Scientists often use a special way of describing syntax of a programming language called the Backus-Naur form (or Bnf). Here is a simplfied description of the syntax of a C function definition using Bnf:

```
functionDefinition : signature block

signature : type VARIABLE OPEN_PARENTHESIS parameterList CLOSE_PARENTHESIS

parameterList : VOID
              | type variable [COMMA type variable]*

block : OPEN_BRACE statement+ CLOSE_BRACE
```

In Bnf, the items in all capital letters represent the words and punctuation in the program. A plus sign means "one or more" while an asterisk means "zero or more". A question mark, not shown in this example, means "zero or one". The first Bnf *rule* says that a function definition is composed of two pieces, a *signature* followed by a *block*, the more formal name for a function body. The *signature*, discussed previously, starts with a type of some kind, followed by a variable, followed by an open parenthesis, followed by a parameter list, and finally followed by a close parenthesis. The *parameterList* rule tells us that the formal parameters in the function signature consist of either the keyword `void` (which signifies no formal parameters) or a *type* followed by a *variable*. If there is more than one formal paramter, they are separated by a comma. A *block* is composed of an open brace, followed by one or more *statements*, and ending with a close brace.

As we can see from the Bnf rules, formal parameters are variables that will be bound to the values supplied in the function call. In the particular case of *isEven*, from the previous section, the variable x will be bound to the number whose evenness is to be determined. As noted earlier, it is customary to call x a *formal parameter* of the function *isEven*.

11.3 The syntax of function calls

Once a function is defined, it is used by *calling* the function with *arguments*. A function is called by supplying the name of the function followed by a parenthesized, comma separated, list of expressions. Those expressions may be literals, variables, array accesses, function calls, combinations thereof, and more.

In Bnf, a description of a function call would look like:

```
functionCall : ID OPAREN argumentList? CPAREN

argumentList : expression [COMMA expression]*
```

Here are some calls to the *isEven* function defined above:

```
x = isEven(y);
process(isEven(q+r),s);
a = isEven(isEven(b));
```

Note that the formal paramaters include a type, but the arguments do not. In Computer Science speak, we say that the values of the arguments are *bound* to the formal parameters during the processing of a function call.

In general, if there are n formal parameters, there should be n arguments.[1] Furthermore, the value of the first argument is bound to the first formal parameter, the second argument is bound to the second formal parameter, and so on. Moreover, all the arguments are evaluated before being bound to any of the parameters. If an argument is a literal, the corresponding formal parameter is bound to that literal value. If an argument is more complicated, C determines the value of that more complicated expression before the formal parameter receives it.

Once the evaluated arguments are bound to the parameters, then the body of the function is evaluated. Most times, the expressions in the body of the function will reference various variables, some of which may be the formal parameters, while some may not. For those that are not formal parameters, how does C find the values of those variables? That question is answered in Chapter **??**.

11.4 Returning from functions

Usually, the evaluation of a function body proceeds, statement by statement, until there are no more statements or when a return statement is encountered. If the return statement has an expression, the value of that expression is determined and becomes the return value of the function. The return value should match the return type of the function.

A function body may have more than one return statement; the first one *reached* wins. Look at this example:

```
int
safeDivide(int x,int y)
    {
    if (y == 0)             //conditional
        return 0;
    else
        return x / y;
    }
```

We haven't covered conditional statements yet, but the code is easy enough to read. When this function is called, if the value of the second argument, which will be bound to *y*, is equal to zero, then the function returns zero. If not, then the function returns the value of the first argument, bound to *x*, divided by the value of the second, bound do *y*.

When a return statement is processed, execution of the function body terminates. Thus, it is possible that some statements in the function can never be processed. For example, consider this version of *safeDivide*:

```
int
safeDivide(int x,int y)
    {
    if (y == 0)             //conditional
        return 0;
    else
        return x / y;
```

[1] For *variadic* functions, which C allows for, the number of arguments may be more than the number of formal parameters. The built-in function *printf* is a variadic function.

```
    return 1;               //not reached
    }
```

Since either y is zero or it is not, one of the two returns in the `if` statement must be reached. Once either is reached, the processing of the function body terminates. Therefore, the return statement at the end of the function can never be reached.

11.5 Changing arguments using the procedure pattern

Consider writing a function to swap two values:

```
void
swap(int x,int y)
    {
    int temp = x;   //line 1
    x = y;          //line 2
    y = temp;       //line 3
    }
```

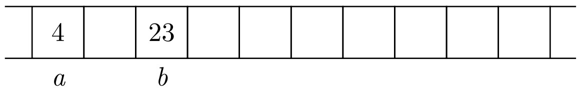
The function, as written, correctly swaps the values of the formal parameters x and y. But can we use the function to swap the values of variables passed to the function? For example, what does the following code fragment produce?

```
//test (with swap defined)
int a = 4;
int b = 23;
printf("a was %d, b was %d\n",a,b);
swap(a,b);
printf("a now is %d, b now is %d\n",a,b);
```

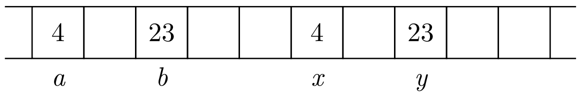
Surprisingly, we get the following output:

```
a was 4, b was 23
a now is 4, b now is 23
```

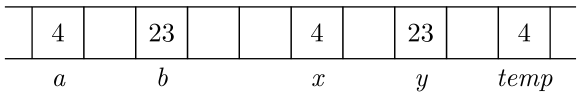
Even though *swap* correctly swaps the values of the formal parameters, the arguments to *swap* remained unchanged. If we look at memory, we can see why. Just before *swap* is called, both a and b refer to memory initialized to 4 and 23, respectively:



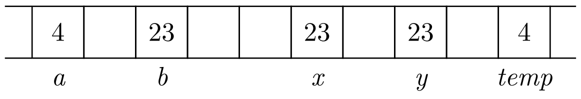
When *swap* is called, the space for the formal parameters is allocated and the formal parameters are given their initial values:



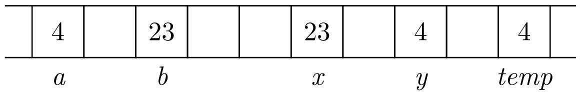
As the body of *swap* runs, the variable *temp* is created (line 1) and initialized to the value of *x*:



After *x* gets the value of *y* (line 2), we have:



After *y* gets the value of *temp* (line 3), we have:



We see that *swap* has indeed swapped the values of the formal parameters. At this point, swap returns and the space allocated for *x*, *y*, and *temp* is reclaimed by the system. Note that through this whole process, the variables *a* and *b* remain unchanged.

So how do we get a function to swap the values of *a* and *b*? We do so by passing the addresses of *a* and *b* to the *swap* function. Here is the new version of *swap*:

```
void
swap2(int *x,int *y)
    {
    int temp = *x;
    *x = *y;
    *y = temp;
    }
```

Taking the address of a variable creates a pointer. Hence, the formal parameters need to be declared `int *` (pointer to `int`). Note the formals *x* and *y* look the same as if we passed in an array of integers to each. In fact, C does not know the difference, which is why you must pass in the size of an array to a function when you pass an array to that function. If you wanted to, you could consider *x* and *y* to be arrays of a single element each, so an alternative, but equivalent, version of *swap* would be:

```
void
swap3(int *x,int *y)
    {
    int temp = x[0];
    x[0] = y[0];
    y[0] = temp;
    }
```

We will stick with the *swap2* version, since that is how 99.999% of the world would write *swap*.

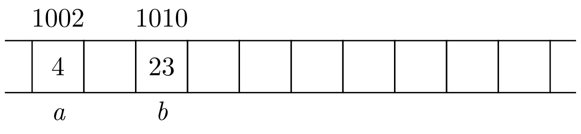
To call *swap*, we send the addresses of the variables whose values are to be swapped:

```
//test (with swap2 defined)
int a = 4;
int b = 23;
printf("a was %d, b was %d\n",a,b);
swap2(&a,&b);
printf("a now is %d, b now is %d\n",a,b);
```

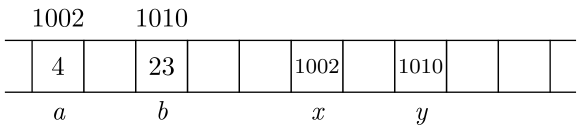
Now, we get the results we desire:

```
a was 4, b was 23
a now is 23, b now is 4
```

Let's look again at memory while our code is running. This time, since we are manipulating addresses, we will pay attention to the addresses of *a* and *b*. For illustration, we will assume the value of *a* is stored at memory location 1002, while *b*'s value is stored at location 1010. Here is the state of memory just before *swap2* is called:



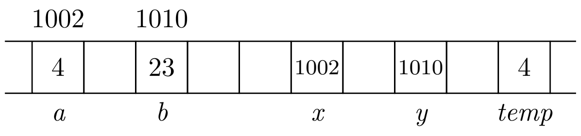
When *swap2* is called, formal parameters *x* and *y* are initialized to the address of *a* and the address of *b*, respectively:



Next, the body of *swap2* is executed, causing the variable *temp* to be created. The line:

```
int temp = *x;
```

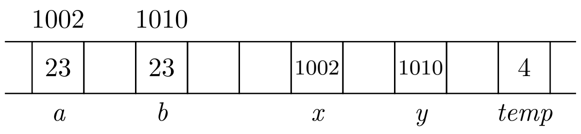
says, "Don't initialize *temp* with the value of *x*. Instead, intialize *temp* to the value at the *address* found in *x*." This leaves memory looking like:



Now, the line:

```
*x = *y;
```

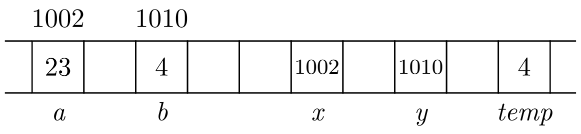
is executed. This line says, "Don't update *x* with the value of *y*. Instead, update the *address* found in *x* with the value at the address found in *y*." Now, memory looks like:



Finally, the line:

```
*y = temp;
```

is executed. This line says, “Don’t update *y* with the value of *temp*. Instead, update the *address* found in *y* with the value of *temp*.” After this update, memory looks like: Now, memory looks like:



At this point, *swap2* returns and we see that the values of *a* and *b* have been exchanged, as desired.

11.6 Functions that “return” multiple items

The C language is limited somewhat because a function can return only one thing. However, sometimes you need more than one piece of information from a function. Consider a function which returns a dynamically allocated array of a random length:

```
int *
randomlySizedIntegerArray(void)
    {
    int size;
    char **array;

    size = random() % 100 + 1; //size will be between 1 and 100
    array = malloc(sizeof(int) * size); //check for malloc failure omitted

    return ???
    }
```

What should this function return? It can return the array, but without knowing the size, the caller of this function won’t know how many slots are in the array it is receiving. The function can return the size, but then the caller will not have access to the array. One solution is to return the array, but require the caller to pass the address of a variable; that variable will be updated by the function, much like the *swap2* function in the previous section updated variables supplied by the caller. Here is a possible implementation:

```
int *
randomlySizedIntegerArray(int *size)
    {
    int *array;

    *size = random() % 100 + 1; //size will be between 1 and 100
    array = malloc(sizeof(int) * *size); //check for malloc failure omitted

    return array;
    }
```

A caller of this function would pass in the address of the variable that is to hold the randomly generated size:

```
int arraySize;
int *array;

array = randomlySizedIntegerArray(&arraySize); //pass in address of arraySize
//arraySize now holds the size of the array
```

This is an important rule to remember: *you cannot change the value of a variable by passing it to a function.* If you wish to change its value, you must pass the address of the variable. An easy way to write such functions is to first assume a value is being sent, as in the original, non-working, *swap* function. Then, for every formal parameter that is to receive an address, place an asterisk before all occurences in the function signature and the function body. Then, place an ampersand before the variable in the function call. If you do this for *swap*, you end up with the correctly working *swap2*. By following this advice, you will ensure proper updates to any variable whose address is to be sent to the function.

11.7 Function pointers

In the same way we can have a variable point to an array, we can have a variable that points to a function. Suppose with have a function whose signature is:

```
char *getString(int,double);
```

The function *getString* is a function that, when passed an integer and a double in that order, returns a string. To create a variable named *g* that can point to this function, we start with the function signature, wrapping the name of the function in the signature with parentheses:

```
char *(getString)(int,double);
```

Then, we place an star in front of the function name (inside the parentheses we just added):

```
char *(*getString)(int,double);
```

Finally, we substitute the variable name *g* for the function name *getString*:

```
char *(*g)(int,double);
```

Given the original signature and this definition of g (plus a definition of *getString* somewhere), we can do this:

```
char *getString(int,double);  //function prototype for getString
char *(*g)(int,double);       //creating a function pointer named g
g = getString;                //getString is NOT called here
char *s = g(2,3.3);           //getString IS called here
```

The type of variable *g* is `char *(*)(int,double)`, and we will refer to this as the *long form* of a function pointer. This long form is rather unwieldy to type over and over, so we can use the *typedef* mechanism to create a shorthand for this type. If we add the keyword `typedef` in front of the type and change the variable name *g* to a name that represents the new type:

```
typedef char *(*FPtr)(int,double);
```

we can make a new type called `FPtr`; this new type can be used to define variables that point to functions like *getString*:

```
typedef char *(*FPtr)(int,double);  //creating a type named FPtr
char *getString(int,double);        //function prototype for getString
FPtr g;                             //creating a function pointer named g
g = getString;                      //getString is NOT called here
char *s = g(2,3.3);                 //getString IS called here
```

We will refer to the typedef'ed version of the function pointer as the *short form*.

With function pointers, we can now pass functions as arguments and return functions as return values. When passing a function like *getString* as an argument, the accepting formal parameter can be declared with the long form or the short form:

```
int f(char *(*u)(int,double));  //long form formal parameter
int g(FPtr v);                  //short form formal parameter
...
int x = f(getString);           //call to function f, passing getString
int y = g(getString);           //call to function g, passing getString
```

Presumably, functions *f* and *g* call *getString* via the formal parameters *u* and *v*, respectively.

When defining a function *h* that returns a function like *getString*, we must use the short form to specify the return type:

```
FPtr g(void) { return getString; }
...
FPtr h = g();
```

We will make use of function pointers when we explore the *map pattern* in the Chapter **??**.

Chapter 12

Conditionals

Conditionals implement decision points in a computer program. Suppose you have a program that performs some task on an image. You may well have a point in the program where you do one thing if the image is a JPEG and quite another thing if the image is a GIF file. Likely, at this point, your program will include a conditional expression to make this decision.

Before learning about conditionals, it is important to learn about logical expressions. Such expressions are the core of conditionals and loops.[1]

12.1 Logical expressions

A logical expression evaluates to a truth value, in essence true or false. For example, the expression $x > 0$ resolves to true if x is positive and false if x is negative or zero. In C, falsehood is represented by the integer 0 and truth by any other integer, usually 1. Recall that, together, these two values are known as Boolean values.

One can assign truth values to integer variables:

```
//test
int c = -1;
int z = (c > 0);
printf("z is %d\n",z);     //should print z is 0
```

If c were positive, the variable z would be assigned a value of 1, which is interpreted as true. Since c is negative, however, it is assigned a value of zero, which is interpreted as false. The parentheses are added to the comparison to improve readability.

12.2 Logical operators

Let's review C's logical operators:

[1]We will learn about loops in the next chapter.

`==`	equal to
`!=`	not equal to
`>=`	greater than or equal to
`>`	greater than
`<`	less than
`<=`	less than or equal to
`&&`	AND, both must be true
`||`	OR, one or both must be true
`!`	a unary operator that reverses a truth value

The first six operators are used for comparing two things, while the next two operators are the glue that joins up simpler logical expressions into more complex ones. The last operator is used to reverse a logical value from true to false and vice versa.

Beginning programmers often confuse the = operator, which is the assignment operator, with the == operator, which is the equality operator. Remember, assignment is used to change the value of a variable while equality can be used to test the value of a variable (without changing its value). Fortunately, the GNU C compiler detects many attempts to use assignment when equality is intended and will complain appropriately.

12.3 Short circuiting

When evaluating a logical expression, C evaluates the subexpressions from left to right and stops evaluating as soon as it finds out that the entire expression is definitely true or definitely false. For example, when encountering the expression:

```
x != 0 && y / x > 2
```

if x has a value of 0, the subexpression on the left side of the AND connective resolves to false. At this point, there is no way for the entire expression to be true (since both the left hand side and the right hand side must be true for an AND expression to be true), so the right hand side of the expression is not evaluated. Note that this particular expression protects against a divide-by-zero error.

Likewise, if the first subexpression in an OR expression is true, then the entire OR expression is true and there is no need to evaluate the second expression in the disjunction. For example, if the call to function f returns true, then the call to function g is never made:

```
f(x) || g(x,y)
```

12.4 If expressions

C's *if* expressions are used to conditionally execute code, depending on the truth value of what is known as the *test* expression. One version of *if* has a single statement following the test expression:

Here is an example:

```
if (strcmp(name,"John") == 0)                //0 means the same
    printf("I like that name!");
```

In this version, when the test expression is true (*i.e.*, the statement following it is executed, causing the compliment. If the test expression is false, however the following statement is not evaluated. You can also use blocks. The following code is equivalent to the previous:

```
    if (strcmp(name,"John") == 0)               //0 means the same
        {
        printf("I like that name!");
        }
```

The advantage of blocks is that you can have more the one statement evaluated when the test expression is true. Novice programmers sometimes think that indentation is sufficient, as in:

```
    if (strcmp(name,"John") == 0)               //0 means the same
        printf("I like that name!");
        goodName = 1;
```

Since C doesn't care about indentation[2], it assumes the assignment statement does not belong to the *if* at all and *goodName* is set to 1 regardless of whether *name* is `"John"` or not. Some programmers always use blocks even if there is a single statement to be executed. We will use that convention in this text, unless we are trying to save space.

Here is another form of *if*:

```
    if (strcmp(major,"Computer Science") == 0)
        {
        print("Smart choice!")
        }
    else
        {
        print("Ever think about changing your major?")
        }
```

In this version, *if* has two blocks, one following the test expression and one following the *else* keyword. As before, the first block is evaluated if the test expression is true. If the test expression is false, however, the second block is evaluated instead. Also as before, if the blocks have a single statement, the braces can be omitted.

12.5 if-elseif-else chains

You can chain *if* statements together using the *else* keyword, followed by an *if* statement, as in:

```
    if (bases == 4)
        {
        printf("HOME RUN!!!\n");
        }
    else if (bases == 3)
        {
        printf("TrIpLe!!\n");
        }
    else if (bases == 2)
        {
        printf("Double!\n");
        }
```

[2]C doesn't care, but your instructor certainly does!

```
else if (bases == 1)
    {
    printf("single\n");
    }
else
    {
    printf("out\n");
    }
```

The block that is eventually evaluated is directly underneath the first test expression that is true, reading from top to bottom. If no test expression is true, the block associated with the *else* is evaluated. Note that the *else* is not required; if no test expressions evaluate to true, then no blocks are executed.

What is the difference between *if-elseif-else* chains and a sequence of unchained *if*s? Consider this rewrite of the above code:

```
if (bases == 4)
    {
    printf("HOME RUN!!!\n");
    }
if (bases == 3)
    {
    printf("TrIpLe!!\n");
    }
if (bases == 2)
    {
    printf("double!\n");
    }
if (bases == 1)
    {
    printf("single\n");
    }
else
    {
    printf("out\n");
    }
```

In the second version, there are four *if* statements and the *else* belongs to the last *if*. Does this behave exactly the same? The answer is, it depends. Suppose the value of the variable *bases* is 0. Then both versions print:

```
out
```

However, if the value of *bases* is 3, for example, the first version prints:

```
TrIpLe!!
```

while the second version prints:

```
TrIpLe!!
out
```

Why the difference? In the first version, a subsequent test expression is evaluated *only* if all previous test expressions evaluated to false. Once a test expression evaluates to true in an *if-elseif-else* chain, the associated block is evaluated and no more processing of the chain is performed. Like the `AND` and `OR` Boolean connectives, an *if-elseif-else* chain short-circuits.

In contrast, the sequences of *if*s in the rewrite are independent; there is no short-circuiting. When the test expression of the first *if* fails, the test expression of the second *if* succeeds and `TrIpLe!!` is printed. Now the test expression of the third *if* is tested and fails as well as the test expression of the fourth *if*. But since the fourth *if* has an else, the *else* block is evaluated and `out` is printed.

It is important to know when to use an *if-elseif-else* chain and when to use a sequence of independent *if*s. If there should be only one outcome, then use an *if-elseif-else* chain. Otherwise, use a sequence of *if*s.

12.6 Explore further

Other constructs in the same vein as the *if* statement are *switch* or *case* statement and the embedded *if* expression (sometimes known as the *ternary operator*). Search the web to find out more.

Practice with Boolean expressions

Here is some practice on writing complex Boolean expressions. Your task is to convert the English description into a C Boolean expression. The variables to be used in the expressions are:

- *animal* with string values `"dog"` and `"cat"`
- *size* with string values `"small"`, `"medium"`, and `"large"`
- *hasSpots* with Boolean values 1 or 0
- *age* with positive integer values
- *coatColor* with string values `"white"`, `"black"`, `"red"`, and `"brown"`
- *eyeColor* with string values `"green"`, `"brown"`, `"blue"`

An old dog is one with an age greater than seven years while an old cat has an age greater than or equal to nine years. A cat and a dog are young if they are younger than three years old.

Write a boolean expression that captures:

Dogs that are large and have spots or cats that are white but do not have blue eyes.

Answer: `(animal == "dog" && size == "large" && hasSpots) || (animal == "cat" && coatColor == "white" && eyeColor != "blue")`

Chapter 13

Input and Output

Generally all computational entities, be it variables, functions, or programs, require some sort of input and output. Input roughly corresponds to the process of supplying data and output roughly corresponds to the process of retrieving data. For a variable, assignment is the way values (data) are supplied to the variable, while referencing the variable (as part of an expression) is the way that data is retrieved. For functions, the input is the set of arguments that are supplied in a function call, while the output of a function is its return value. For programs, input refers to pulling in data and output refers to pushing out results. In this chapter, we focus on managing the input and output of programs.

13.1 Input

Input is the process of obtaining data that a program needs to perform its task. There are generally three ways for a program to obtain data:

- interactively, by querying the user for information
- via the command line, by accessing an array of command line arguments
- via a file, by opening a file and reading its contents

We examine these three approaches in turn.

13.1.1 Reading from the keyboard

To read from the keyboard, one uses the *scanf* function.

Reading integers

To read an integer, an integer directive is supplied to *scanf*:

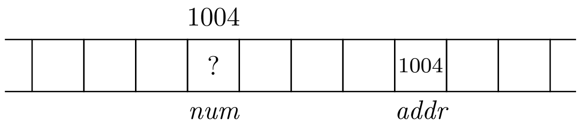
```
//test
int num;
printf("Please enter an integer: ");
scanf("%d",&num);
printf("the number you entered was %d\n",num);
```

The *scanf* function takes a guide string and a pointer to a memory location, Moreover, *scanf* returns when it has seen an integer followed by a newline on the input. At a level lower than *scanf*, C waits until an entire line has been input before *scanf* can start its task.

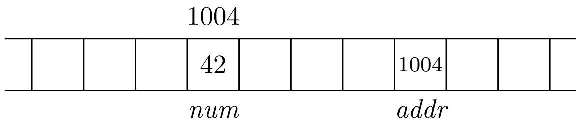
In the example, we give the address of *num*; the `&` operator returns the address of the *num* variable. We have seen pointers to arrays. Pointers to variables are really the same thing. The difference is a pointer to a single variable can be thought of as an array with a single slot. Let's rewrite the above example a little bit:

```
//test
int num;
int *addr;              //an 'array' pointer
addr = &num;            //&num has type 'int *'
printf("Please enter an integer: ");
scanf("%d",addr);
printf("num is %d\n",num);
printf("addr[0] is %d\n",addr[0]);
printf("*addr is %d\n",*addr);
```

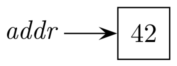
Suppose *num* is placed at memory location 1004. Immediately prior to the first *printf*, the situation looks like:



The *addr* variable, as desired, holds the address of *num*. Now the value of *addr* is passed as the second argument to *scanf*. The *scanf* function reads in a number and places it in the memory location specified by the second argument. In this case, it is placed at memory location 1004. So, if the user enters the number 42, then 42 is stored as the value of *num*:



The final three print statements all output the value 42. The first does so since the value read in by *scanf* was stored in *num*. The second does so since *addr* can be considered a pointer to an array of length 1:



The final print statement prints 42 since `*addr` is an alternate form of `addr[0]`. When we take the address of a variable, we will rarely use the array indexing form used by the second print statement, preferring instead the *pointer dereferencing* form of the third print statement.

Reading real numbers

Reading a real number is similar to reading an integer, except the `"%lf"` directive and the address of a double variable is passed to *scanf*:

```
//test
double realNumber;
printf("Please enter a real number: ");
scanf("%lf",&realNumber);
printf("the number you entered was %f\n",realNumber);
```

Reading single characters

For a character, the `"%c"` directive and the address of a character variable is passed to *scanf*.

```
//test
char ch;
printf("Please enter a character: ");
scanf("%c",&ch);
printf("the character you entered was <%c>\n",ch);
```

With the `"%c"` directive, the next character on the input stream is read, regardless of whether or not it is a whitespace. If one places a space before the percent sign in the directive, then *scanf* will skip over any whitespace and read the first non-whitespace character pending on the input:

```
//test
char ch;
printf("Please enter some whitespace, then a character: ");
scanf(" %c",&ch); //note the space in the directive
printf("the first darkspace character you entered was <%c>\n",ch);
```

Reading strings

One can use the *scanf* function to read multiple characters at a time from the keyboard with the `"%s"` directive:

```
//test of VERY DANGEROUS CODE
char buffer[512]; //room for 511 chars plus the null char
printf("Please enter a token: ");
scanf("%s",buffer);
printf("the token you entered was <%s>\n",buffer);
```

where a `token` is a contiguous sequence of non-whitespace characters. Why is this code dangerous? The reason is *scanf* does not ensure that no more than 511 characters (in this example) are read into array *buffer*. So if the token entered is, say, 600 characters in length, an additional 90 characters will be read into memory, trashing the memory beyond the extent of the array.

A very clever and devious person at some point realized that if the token was long enough, the memory trashing would extend beyond the memory space reserved for the local variables and into the region of memory that stores the location of the caller of the current function. When the current function returns, program control jumps to the caller of the function. By carefully choosing the additional characters, the return location is overwritten with the address of the buffer itself. If the buffer is filled with characters that look like machine code, then this new program would run with the same privileges as the original program. If a program with high privileges is exploited in this way, then the exploiter could potentially take over the system.

This is not just a theoretical problem. The first widespread internet virus, the Morris worm, used this approach and brought large portions of the internet to a halt[1].

[1]see http://en.wikipedia.org/wiki/Morris_worm.

In fact, many of the vulnerabilities being discovered or exploited today are due to this use of *scanf* and related functions in system-level C code. So, NEVER USE SCANF TO READ IN A STRING! Instead of *scanf*[2], one should read a line one character at a time and checking to see if too many characters are encountered before the newline is seen. This is a rather onerous task, so it has been done for you. To get this code, download the following files:

```
wget troll.cs.ua.edu/ACP-C/scanner.c
wget troll.cs.ua.edu/ACP-C/scanner.h
```

The *scanner.c* file contains the definitions of the following functions:

function	*purpose*
readInt	returns the next integer
readReal	returns the next double
readChar	returns the next darkspace character
readRawChar	returns the next character, whitespace or darkspace
readToken	returns the next token
readString	returns the next double quoted string
readLine	returns the remainder of current line

You can read more about these functions in the documentation of *scanner.c*. You can find the prototypes of the functions in *scanner.h*.

The *scanf* code for reading a token can be replaced with a safe call to *readToken*. To use the scanner, you need to add the following line after the system includes:

```
#include "scanner.h"
```

Then you can make calls to the scanner functions:

```
//test
char *s;  //note change of s to a char pointer
printf("Please enter a token: ");
s = readToken(stdin);     //stdin is the keyboard
printf("the token you entered was <%s>\n",s);
free(s); //s points to a malloc'd array so free it when done
```

To compile code that calls scanner functions, you will need to compile *scanner.c* with the rest of your program. Suppose your *main* function is in *tester.c* and it calls *readToken*. To compile the program, you would enter the command:

```
gcc -Wall -g -o tester tester.c scanner.c
```

The scanner functions require the passing in of a file pointer. This allows them to read from the keyboard (using *stdin*) or from a file (more on that later on in the chapter). The first four functions listed are wrappers to *scanf*. The last three read characters into a statically allocated array. If there is not enough room, an error message is printed and the program is terminated. If there is room, the array is copied into malloc'd memory and a pointer this memory is returned.

[2]In more modern versions of C, one can use the `"%ms"` directive and pass in the address of an uninitialized string pointer. In this case, *scanf* would dynamically allocate a sufficiently large character array, fill it, and set the pointer to point to the array. The caller of *scanf* would be responsible for freeing the array.

Reading directly into arrays

One can read directly into arrays by using pointer offsets:

```
//test
int a[3];
printf("give me the first number: ");
scanf("%d",a+0);
printf("give me the second number: ");
scanf("%d",a+1);
printf("give me the third number: ");
scanf("%d",a+2);

printf("[%d]",a[0]);
printf("[%d]",a[1]);
printf("[%d]",a[2]);
printf("\n");
```

If the three numbers read were 2, 13, and 42, in that order, then the output would be:

```
[2][13][42]
```

Sometimes you will see:

```
scanf("%d",&a[1]);
```

instead of:

```
scanf("%d",a+1);
```

The former is usually written by a programmer who is a little unsure of the relationships between statically allocated arrays and pointers. One can also use the scanner functions:

```
...
printf("give me the first number: ");
a[0] = readInt(stdin);
printf("give me the second number: ");
a[1] = readInt(stdin);
printf("give me the third number: ");
a[2] = readInt(stdin);
...
```

13.1.2 Reading from the command line

The second way to pass information to a program is through *command-line arguments*. The command line is the line typed in a terminal window that runs a C program (or any other program). Here is a typical command line on a Linux system:

```
lusth@warka:~/l1/activities$ prog3
```

Everything up to and including the dollar sign is the system prompt. As with all prompts, it is used to signify that the system is waiting for input. The user of the system (me) has typed in the command:

```
prog3
```

in response to the prompt. Suppose *prog3.c* is a file with the following code:

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

int
main(int argc,char **argv)
    {
    int i;
    printf("Number of command-line arguments: %d\n",argc);
    printf("Command-line arguments:\n");
    i = 0;
    while (i < argc)
        {
        printf("    %s\n",argv[i]);
        ++i;
        }
    return 0;
    }
```

We haven't covered loops yet, but all the program does is print out all of the command line arguments. In this case, the output of this program would be:

```
lusth@warka:~/l1/activities$ gcc -Wall -g -o prog3 prog3.c
lusth@warka:~/l1/activities$ prog3
Number of command-line arguments: 1
Command-line arguments:
    prog3
```

From the output, we can see that there was one command-line argument, the name of the executable. Looking more closely at the code:

```
printf("Number of command-line arguments: %d\n",argc);
```

we see that the number of command line arguments is stored in the formal parameter *argc*. We see also:

```
printf("    %s\n",argv[i]);
```

that the command line arguments themselves are stored in the formal parameter *argv*. This tells us that *argv* points to an array of strings. A pointer has a star, and a string has a *, hence the typing of *argv*:

```
    char **argv;
```

with two stars.

Any whitespace-delimited tokens following the program file name are stored in *argv* along with the name of the program being run. For example, suppose we run *prog3* with the this command:

```
    lusth@warka:~/l1/activities$ prog3.py 123 123.4 True hello, world
```

Then the output would be:

```
    Number of command-line arguments: 6
    Command-line arguments:
        prog3
        123
        123.4
        True
        hello,
        world
```

From this result, we can see that all of the tokens are stored in *argv* and that they are stored as strings, *regardless* of whether they look like some other entity, such as integer or real number.

If we wish for `"hello, world"` to be a single token, we would need to enclose the tokens in quotes:

```
    prog3 123 123.4 True "hello, world"
```

In this case, the output is:

```
Number of command-line arguments: 5
Command-line arguments:
    prog3
    123
    123.4
    True
    hello, world
```

There are certain characters that have special meaning to the system. A couple of these are '*' and ';'. To include these characters in a command-line argument, they need to be *escaped* by inserting a backslash prior to the character. Here is an example:

```
    prog3 \; \* \\
```

To insert a backslash, one escapes it with a backslash. The output from this command is:

```
Number of command-line arguments: 4
Command-line arguments:
    prog3
    ;
    *
    \
```

Although it looks as if there are two backslashes in the last token, there is but a single backslash. Most Linux programs, including the terminal window shell, uses two backslashes to indicate a single backslash.

What command-line arguments are

The command line arguments are stored as strings. Therefore, you must use *atoi* or *atof* if you wish to use any of the command line arguments as integers or real numbers, respectively. Here is code that tests that there are two command-line arguments beyond the program name, the first representing an integer and the second representing a real:

```
//test
int x;
double y;
if (argc != 3)
   {
   fprintf(stderr,"there should be two args beyond the program name\n");
   exit(1);
   }
x = atoi(argv[1]);
y = atof(argv[2]);

printf("1st additional arg is %d\n",x);
printf("2nd additional arg is %f\n",y);
```

Given the additional arguments 23 and 4.5, the output should be:

```
1st additional arg is 23
2nd additional arg is 4.500000
```

13.1.3 Reading from files

The third way to get data to a program is to read the data that has been previously stored in a file.

C uses a *file pointer* system in reading from a file. To read from a file, the first step is to obtain a pointer to the file. This is known as *opening* a file. The file pointer will always point to the first unread character in a file. When a file is first opened, the file pointer points to the first character in the file.

Reading files using *fscanf*

Suppose we wish to read from a file named *data*. We first obtain a file pointer by opening the file like this:

```
FILE *fp = fopen("data","r")
if (fp == 0)
```

```
        {
        fprintf(stderr,"file data could not be opened for reading\n");
        exit(1);
        }
```

The *fopen* function takes two arguments, the name of the file and the kind of file pointer to return. We store the file pointer in a variable named *fp* (a variable name commonly used to hold a file pointer). In this case, we wish for a *reading* file pointer, so we pass the string `"r"`. We can also open a file for writing; more on that in the next section. In any case, you should always test the return value of *fopen*; a return value of zero means a problem has occurred.

Once we have the file pointer, we can use *fscanf* to read various kinds of items. For example, to read an integer, we would use the `"%d"` directive:

```
//test
int x;
FILE *fp = fopen("data","r");
//test of fp omitted
fscanf(fp,"%d",&x);
printf("the number read was %d\n",x);
fclose(fp);
```

Note how similar the call to *fscanf* is to *scanf*. The following two calls are equivalent:

```
scanf("%d",&x);
fscanf(stdin,"%d",&x);
```

In fact, *scanf* and *fscanf* are identical except that *scanf* hard-wires the file pointer to *stdin*, the keyboard.

When we are done reading a file, we *close* it:

```
fclose(fp);
```

Always remember to close your files!

Reading files using the scanner

We can also use the scanner functions to read from a file, by passing a file pointer instead of *stdin*:

```
//test
int x;
FILE *fp = fopen("data","r");
//test of fp omitted
x = readInt(fp);
printf("the number read was %d\n",x);
fclose(fp);
```

As always, remember to close your files when finished.

13.2 Output

Once a program has processed its input, it needs to make its output known, either by displaying results to the user or by storing the results in a file.

13.2.1 Writing to the console

We have been writing to the console using *printf* for some time now. The *printf* function is *variadic*, which means it can take a variable number of arguments:

```
//test
int x = 3;
double y = 14.4;
char *z = "hello";
printf("an integer, %d, a real number, %f, and a string, %s\n",x,y,z);
```

The arguments after the guide string are match to the directives in the guide string, in order given. If you have a mismatch between the directive and the corresponding argument, you will get a warning message when you compile.

Printing quote characters

Suppose I have the string:

```
char *str = "Hello";
```

If I print my string:

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

The output looks like this:

```
Hello
```

Notice the double quotes are not printed. But suppose I wish to print quotes around my string, so that the output looks like:

```
"Hello"
```

To do this, the print statement becomes:

```
printf("\"%s\"",str);
```

If I want the double quotes to be in the string itself, str would assigned thusly:

```
str = "\"Hello\"";
```

If you need a refresher on what the string `"\""` means, please see Chapters **??**.

13.3 Writing to a file

C also requires a file pointer to write to a file. The *fopen* function is again used to obtain a file pointer, but this time we desire a *writing* file pointer, so we send the string `"w"` as the second argument to *fopen*:

```
FILE *fp = fopen("data.save","w")
```

Now the variable *fp* points to a writing file object; we write to the file using *fprintf*. The correspondence between *fprintf* and *printf* is the same as that between *fscanf* and *scanf*. The following two calls are equivalent:

```
printf("hello, world!\n");
fprintf(stdout,"hello, world!\n");
```

The *printf* function simply hardwires the file pointer to *stdout*, which represents the console. As with reading, a file opened for writing should be checked for a successful open and should be closed with *fclose* when reading is complete.

Opening a file in order to write it has the effect of emptying the file of its contents soon as it is opened. The following code deletes the *contents* of a file (which is different than deleting the file):

```
// delete the contents
FILE *fp = fopen(fileName,"w")
//check that fopen did not encounter a problem
fclose(fp);
```

If you wish to start writing to a file, but save what was there previously, call the *open* function to obtain an *appending* file pointer:

```
FILE *fp = fopen(fileName,"a")
```

Subsequent writes to *fp* will append text to what is already there.

Chapter 14

Scope

A *scope* holds the current set of variables and their values. In C, there is something called the *global scope*. The global scope holds all the values of the built-in variables and function names (remember, a function name is a pseudopointer, similar to a variable).

When you analyze a C program, either by compiling and running it or by reading it, you start out in the global scope. As you come across definitions of variables[1] and functions, their names and their values are added to the global scope.

Both variables and functions can be defined in the global scope. Processing of the following code adds the names *x* and *main* to the global scope:

```
#include <stdio.h>

//this is the global scope

int x = 3;                  //x is bound to 3

int
main(int argc,char **argv)  //main is bound to a function object
    {
    ...
    return 0;
    }
```

This interaction adds two names to the global scope:

```
//this is the global scope
int y = 4;

int
negate(int z)
    {
    return -z;
    }
```

What are the two names? The two names added to the global scope are *y* and the *negate*.

[1]You will often see the term *variable declaration* instead of variable definition.

Indeed, since functions are defined in the global scope, and the name *negate* is being bound to a function object, it becomes clear that this binding is occurring in the global scope, just like *y* being bound to 4.

Scopes in C can be identified by *block nesting level*. The global scope holds all variables defined with an nesting level of zero or, in other words, not within a block. Recall that when functions are defined, the body of the function is contained in a block. Thus, the function body is code that is nested within the global scope and constitutes a new scope, with a nesting level of one. We can identify to which scope a variable belongs by looking at the pattern of nesting. In particular, we label variables as either *local* or *non-local* with respect to a particular scope. Moreover, we can label non-local variables as *in scope* or *out of scope*.

14.1 In Scope or Out

The nesting pattern of a program can tells us where variables are visible (in scope) and where they are not (out of scope). We begin by first learning to recognizing the scopes in which variables are defined.

14.1.1 The Local Variable Pattern

All variables *defined* at a particular nesting level or scope are considered *local* to that nesting level or scope. In C, if one defines a variable, that variable must be local to the current scope. An exception are the formal parameters of a function definition; these belong to the scope that is identified with the function body. So within a function body, the local variables are the formal parameters plus any variables defined immediately within the function body.

Let's look at an example. Note, you do not need to completely understand the examples presented in the rest of the chapter in order to identify whether names are local and non-local.

```
//this is the global scope
int
theta(int a,int b)
    {
    //this is a non-global scope
    int c,d;
    c = a + b;
    c = kappa(c) + X;
    d = c * c + a;
    return d * b;
    }
```

In this example, we can immediately say the formal parameters, *a* and *b*, are local with respect to the scope of the body of function *theta*. Furthermore, variables *c* and *d* are defined in the function body so they are local as well, with respect to the scope of the body of function *theta*. It is rather wordy to say "local with respect to the scope of the body of the function *theta*", so Computer Scientists will almost always shorten this to "local with respect to *theta*" or just "local" if it is clear the discussion is about a particular function or scope. We will use this shortened phrasing from here on out. Thus *a*, *b*, *c*, and *d* are local with respect to *theta*. The function name *theta* is local to the global scope since the function *theta* is defined in the global scope.

14.1.2 The Non-local Variable Pattern

In the previous section, we determined the local variables of the function. By the process of elimination, that means the names *kappa*, and *X* are non-local. The name of function itself is non-local with respect to its body. The name *theta*, if it is referenced with the body of *theta* is non-local as well.

In a function body, any variable that is not a formal parameter and is also not defined immediately within the function body must be non-local.

14.1.3 The Visible Variable Pattern

A variable is accessible or *visible* with respect to a particular scope if it is *in scope*. A variable is in scope if it is local or was defined in a scope that *encloses* the particular scope. Some scope *A* encloses some other scope *B* if, by moving (perhaps repeatedly) leftward from scope B, scope A can be reached. Here is example:

```
int Z = 5;

int
iota(int x)
   {
   return x + Z;
   }
```

The variable *Z* is local with respect to the global scope and is non-local with respect to *iota*. However, we can move leftward from the scope of *iota* one nesting level and reach the global scope where *Z* is defined. Therefore, the global scope encloses the scope of *iota* and thus *Z* is visible from *iota* and its value can be accessed within *iota*.. Indeed, the global scope encloses all other scopes and this is why the built-in functions are accessible at any nesting level.

Here is another example that has two enclosing scopes:

```
int X = 3;

int
phi(int a)
    {
    int Y = a - 1;
    if (isEven(a))
        {
        int b = X + 1;
        return a * b;
        }
    else
        {
        return X * Y;
        }
    }
```

The global scope locally defines two names, *X* and *phi*. If we look at function *phi*, we see that it has two local variables, the formal parameter *a* and the locally defined variable *Y*. In the *if* statement, we see that the true branch is a block. This block is a new nesting level and therefore a new scope. This new scope has the locally defined variable *b*, but references the non-local variable *X*. The false branch is also a block and a new scope, but it has no local variables. However, it references the non-local variables *X* and *Y*.

Are all these non-local variables accessible? Consider the first non-local reference to *X*. Moving leftward from the true branch of the *if*, we reach the body of *phi*, so the scope of *phi* encloses the scope of the true branch. Moving leftward again, we reach the global scope, where *X* is defined. Therefore, the global scope encloses (transitively) the scope of the true branch, so X is visible and accessible within the true branch. For the same reasons, both *X* and *Y* are visible and accessible within the false branch of the *if*.

In the next section, we explore how a variable can be inaccessible.

14.1.4 The Tinted Windows Pattern

The scope of local variables is like a car with tinted windows, with the variables defined within riding in the back seat. If you are outside the scope, you cannot peer through the car windows and see those variables. You might try and buy some x-ray glasses, but they probably wouldn't work. Here is an example:

```
int
alpha(int a)
    {
    if (isEven(a))
        {
        int b = a + 1;
        printf("%d\n",a * b);
        }
    printf("%d\n",b * 2);
    }
```

The print statement at the end of the function causes an error:

```
error: 'b' undeclared (first use in this function)
     printf("%d\n",b * 2);
                   ^
```

The rule for figuring out which variables are in scope and which are not is: *you* **cannot** *see into an enclosed scope*. In the example, the scope of the *if* is enclosed by the scope of *alpha*. Therefore, in *alpha*, any references to variables local to the *if* block are invalid. Contrast this with the non-local pattern: In the *if* block, any references to variables local to *alpha* (and the global scope, by transitivity) are valid.

14.1.5 Tinted Windows with Parallel Scopes

The tinted windows pattern also applies to parallel scopes. Consider this code:

```
int
gamma(int a)
    {
    return a + delta(a);
    }

int
delta(int x)
    {
    // starting point 1
    printf("the value of a is %d\n",a); //x-ray!
    return x + 1;
    }
```

Note that the global scope encloses both the scope of *gamma* and the scope of *delta*. However, the scope of *gamma* does not enclose the scope of *delta*. Neither does the scope of *delta* enclose the scope of *gamma*.

One of these functions references a variable that is not in scope. Can you guess which one? The function *delta* references a variable not in scope.

Let's see why by first examining *gamma* to see whether or not its non-local references are in scope. The only local variable of function *gamma* is *a*. The only referenced non-local is *delta*. Moving leftward from the body of *gamma*, we reach the global scope where where both *gamma* and *delta* are defined. Therefore, *delta* is visible with respect to *gamma* since it is defined in a scope (the global scope) that encloses *gamma*.

Now to investigate *delta*. The only local variable of *delta* is *x* and the only non-local that *delta* references is *a*. Moving outward to the global scope, we see that there is no variable *a* defined there, therefore the variable *a* is not in scope with respect to *delta*.

When we actually run the code, we get an error similar to the following when running this program:

```
error: 'a' undeclared (first use in this function)
    printf("the value of a is %d\n",a); //x-ray!
                                    ^
```

' The lesson to be learned here is that we cannot see into the local scope of the body of function *gamma*, *even if we are at a similar nesting level*. Nesting level doesn't matter. We can only see variables in our own scope and those in *enclosing* scopes. All other variables cannot be seen.

Therefore, if you ever see a variable-not-defined error, you either have spelled the variable name wrong or you are trying to use x-ray vision to see somewhere you can't.

14.2 Alternate terminology

Sometimes, enclosed scopes are referred to as *inner* scopes while enclosing scopes are referred to as *outer* scopes. In addition, both locals and any non-locals found in enclosing scopes are considered *accessible* or *visible* or *in scope*, while non-locals that are not found in an enclosing scope are considered *inaccessible*, or *not visible* or *out of scope*. We will use all these terms in the remainder of the text book.

14.3 Three Scope Rules

Here are three simple rules you can use to help you figure out the scope of a particular variable:

- Formal parameters go in
- The function name goes out
- You can see out but you can't see in (tinted windows).

The first rule is shorthand for the fact that formal parameters belong to the scope of the function body. Since the function body is "inside" the function definition, we can say the formal parameters go in.

The second rule reminds us that function names are defined in the global scope, so they go out with respect to the function body.

The third rule tells us all the variables that belong to ever-enclosing scopes are accessible and therefore can be referenced by the innermost scope. The opposite is not true. A variable in an enclosed scope can not be referenced by an enclosing scope. If you forget the directions of this rule, think of tinted windows. You can see out of a tinted window, but you can't see in.

14.4 Shadowing

Consider the rule that says that references to variables or names in an outer scope are visible. There is one exception to that rule; consider the following code fragment:

```
int a = 4;

void
epsilon(int a)
    {
    printf("a is %d\n",a);
    return;
    }
...
epsilon(13);
...
```

In this example, the global scope defines two names, *a* and *epsilon*. The scope of the body of *epsilon* defines the name *a*. When *epsilon* is called with some argument (in the example, 13), what is printed for the value of *a*, the value passed in and bound to the formal parameter *a* or the value of the global variable *a*?

If we write a program around this code fragment, we see the following output:

```
a is 13
```

Clearly, the C compiler preferred the formal parameter over the global version. The reason is, when resolving the value of a variable, the most local scope is searched for a binding for that name. Should the local scope not hold a binding, the immediate enclosing scope is searched. If no binding is found there, the enclosing scope of *that* scope is search and so on, until the global scope is reached. Should no binding be found in the global scope, an undefined variable error is generated by the compiler.

In the example, a binding for variable *a* is found in the local scope and the value found there is retrieved for the print statement. As a consequence, it is impossible to retrieve the value of the global binding for *a*. It is said that the formal parameter *shadows* the global variable. Being in the shadow of the formal parameter means it cannot be seen. In general, when a more local variable has the same name as a less local variable that is also in scope, the more local variable shadows the less local version.

In the example, if you wished to reference both the global *a* and the formal parameter *a*, you would need to rename the formal parameter.

14.5 Multiple definitions and declarations

C does not allow multiple definitions having the same name. Thus, one cannot have two (or more) functions named the same or two variables in the same scope with the same name. On the other hand, one can have multiple declarations. For example, it is perfectly legal to repeat a function prototype/signature/declaration:

```
char *f(int,double);
char *f(int,double); //duplicative, but OK
```

In contrast, the compiler complains about:

```
int square(int x) { return x * x; }
int square(int x) { return x * x; } //duplicative and illegal
```

Variables work the same way, except there is a slight wrinkle in C. In the global scope, the following is legal in C:

```
int x;
int x; //duplicative, but OK in the global scope
```

The C compiler interprets one of the lines as a declaration and the other as a definition. However, if the above lines were found in a non-global scope, say a function body, the compiler would complain about multiple definitions:

```
int almostSquare(int y)
    {
    int x;
    int x;        //duplicative and illegal
    x = y - 1;
    return x * y;
    }
```

Within a non-global scope, both lines are considered definitions, thus violating the multiple definition rule for C. One can avoid the multiple definition error in a non global scope by explicitly stating that one of the lines is a declaration. This is accomplished with the `extern` keyword:

```
int almostSquare(int y)
    {
    int x;
    extern int x;        //OK
    x = y - 1;
    return x * y;
    }
```

Initializing a variable forces the compiler to treat a potential declaration as a definition. In the global scope:

```
int x = 13;   //definition
int x;        //declaration
```

the first line must be interpreted as a definition as the variable is initialized. To avoid a multiply defined variable error, the compiler interprets the second line as a declaration. However, the following is illegal in all scopes:

```
int x = 13; //definition
int x = 13; //also a definition, therefore illegal
```

Do not tag a variable as `extern` and initialize it at the same time:

```
extern int x = 100; //don't ever do this
```

If you do this, you are telling the compiler this is only a declaration, not a definition (via `extern`), but at the same time, you are telling the compiler this is a definition (via the initialization).

You might be thinking, this is really a non-issue since nobody in his or her right mind would define/declare the same variable in the same scope. Such a situation, it turns out, happens quite commonly. Suppose a file named *c.h* contains the following line:

```
int x;
```

and both module *a.c* and *b.c* include *c.h*. Then *a.c* and *b.c* are compiled together to make the executable:

```
gcc -Wall a.c b.c
```

To the compiler, definitions in both a.c and b.c are all in the global scope, so the compiler sees two declarations of *x*, since *c.h* is included in both. The compiler turns one of those declarations into a definition and everything works. However, if the line in *c.h* is changed to:

```
int x = 100;
```

then the compiler, when compiling *a.c* and *b.c* together, sees two definitions of *x* and emits a compiler error.

To sum all this up in two simple rules, we have:

1. Never initialize a variable in a *.h* file
2. If a global variable is used across modules, only initialize the variable in one module.

Chapter 15

Loops

You can download the functions defined in this chapter with the following commands:

```
wget troll.cs.ua.edu/ACP-C/loops.c
wget troll.cs.ua.edu/ACP-C/loops.h
wget troll.cs.ua.edu/ACP-C/scanner.c
wget troll.cs.ua.edu/ACP-C/scanner.h
```

These files will help you run the test code listed in this chapter.

15.1 While loops

Loops are used to repeatedly execute some code. The most basic loop structure in C is the *while* loop, an example of which is:

```
int i = 0;
while (i < 10)
    {
    printf("%d\n",i);
    i = i + 1;
    }
```

We see a `while` loop looks much like an `if` statement. The difference is that blocks belonging to `if`s are evaluated at most once whereas block associated with a loop may be evaluated many many times. Another difference in nomenclature is that the block of a loop is known as the *body* (like blocks associated with function definitions). Furthermore, the loop test expression is known as the *loop condition.*

As Computer Scientists hate to type extra characters if they can help it, you will often see:

```
i = i + 1;
```

written three different ways:

```
i += 1;
++i;
i++;
```

The latter two versions are read as "increment *i*" and saves at least a whopping two characters of typing, more as the variable name gets longer. The difference between the last and the second-to-last version is when the expression is used within a larger expression. Stylewise, this is generally a bad idea, so until you know better, always use `++i` or `i++` as stand-alone statements.

A *while* loop tests its condition before the body of the loop is executed. If the initial test fails, the body is not executed at all. For example:

```
//test
int i = 10;
while (i < 10)
    {
    printf("%d\n",i);
    ++i;
    }
printf("[nothing should have printed]\n");
```

never prints out anything since the test immediately fails. In this example, however:

```
//test
int i = 0;
while (i < 10)
    {
    printf("%d",i);
    ++i;
    }
printf("\n");
printf("0123456789 should have printed\n");
```

the loop prints out the digits 0 through 9:

```
0123456789
```

A `while` loop repeatedly evaluates its body as long as the loop condition remains true.

To write an infinite loop, use true as the condition:

```
while (1)
    {
    i = getInput();
    printf("input is %d\n",i);
    process(i);
    }
```

15.2 The *for* loop

The other loop in C is the *for* loop:

```
//test
```

```
    int i;
    for (i = 0; i < 10; ++i)
        {
        printf("%d\n",i);
        }
```

This loop is exactly equivalent to:

```
    //test
    int i = 0;
    while (i < 10)
        {
        printf("%d\n",i);
        ++i;
        }
```

In fact, a while loop of the general form:

```
    i = INIT;
    while (i < LIMIT)
        {
        // body
        ...
        i += STEP;
        }
```

can be written as `for` loop:

```
    for (i = INIT; i < LIMIT; i += STEP)
        {
        // body
        ...
        }
```

For loops are commonly used to sweep through each element of an array:

```
    //test
    int i;
    int items[] = { 0,1,1,2,3,5,8,13 };
    int itemCount = sizeof(items)/sizeof(int);

    for (i = 0; i < itemCount; ++i)
        {
        printf("%d\n",items[i]);
        }
```

Here, we sweep through an array of integers. Since we cannot tell, in general, the size of the array given the name of the array, we must make available the size of the size of the array to the loop. In the example above, the size of the array is stored in the variable *itemCount*.

Recall the items in an array of n elements are located at indices 0 through $n - 1$. These are exactly the values produced by the initialization, test, and step of the *for* loop. The above loop accesses each element by its index in turn, and thus prints out each element, in turn. Since using an index of n in an array of n items produces an error, the test of the index i against the size of the array uses a strict "less than" comparison.

A common use for loops is to initialize an array. Recall that when an array is allocated statically without initializers or is dynamically allocated, the slots are filled with garbage. A loop can be used to set the elements of an array to a known value:

```
array = malloc(sizeof(int) * size);
//check for malloc failure omitted
for (i = 0; i < size; ++i)
    {
    array[i] = 0;
    }
```

Loops and arrays go very well together; the next sections detail some common loop patterns involving arrays.

15.3 The *counting* pattern

The counting pattern is used to count the number of items in a collection. Note that we require the size of the array to be available, so counting the elements in an array is a bit of a waste of time. Still, it is about the easiest array processing loop to write, so let's forge ahead[1]

```
int i,count = 0;
for (i = 0; i < itemCount; ++i)
     {
     ++count;
     }
//count now holds the number of items
```

When the loop finishes, the variable *count* holds the number of items in the array. In general, the counting pattern increments a counter every time the loop body is evaluated.

The counting pattern is more useful for *while* loops. Suppose we are trying to count the number of characters in a file. Suppose further that the function *getNextCharacter* reads and returns the next character in a file or it returns -1 if there are no more characters to be read. A counting loop would look like this:

```
count = 0;
ch = getNextCharacter();
while (ch != -1)
    {
    ++count;
    ch = getNextCharacter();
    }
//count now holds the number of characters
```

At the end of the loop, *count* holds the number of characters in the file. Note how we did not need to know the number of characters in advance. The reason is we had a way to detect when the counting loop should stop running.

[1]We will see the utility of the counting pattern for *lists*, for which we do not need the size, *a priori*.

15.4 The *filtered-count* pattern

A variation on the counting pattern involves filtering. When *filtering*, we use an `if` statement to decide whether we should count an item or not. Suppose we wish to count the number of even items in an array:

```
int i,count = 0;
for (i = 0; i < itemCount; ++i)
    {
    if (items[i] % 2 == 0)  //true if items[i] is even
        {
        ++count;
        }
    }
//count now holds the number of even items
```

When this loop terminates, the variable *count* will hold the number of even integers in the array of items. The *if* makes sure the count is incremented only when the item of interest is even.

15.5 The *accumulate* pattern

Similar to the counting pattern, the *accumulate* pattern updates a variable, not by increasing its value by one, but by the value of an item. This loop, sums all the values in an array:

```
int i,total = 0;
for (i = 0; i < itemCount; ++i)
    {
    total += items[i];      //equivalent to total = total + items[i];
    }
//total now holds the sum of all items
```

By convention, the variable *total* is used to accumulate the item values. When accumulating a sum, the total is initialized to zero. When accumulating a product, the total is initialized to one[2].

15.6 The *filtered-accumulate* pattern

Similar to both the *filtered-count* pattern, the *filtered-accumulate* pattern updating the total only if some test is passed. This function sums all the even values in a given array, returning the final sum:

```
int
sumEvens(int *items,int itemCount)
    {
    int i,total;
    total = 0;
    for (i = 0; i < itemCount; ++i)
        {
        if (items[i] % 2 == 0)
            {
            total += items[i];
```

[2]The superior student will ascertain why this is so.

```
                }
            }
        return total;
        }
```

As before, the variable *total* is used to accumulate the item values. As with a regular accumulating, *total* is initialized to zero when accumulating a sum. The initialization value is one when accumulating a product and the initialization value is the empty array when accumulating an array (see *filtering* below).

15.7 The *search* pattern

The *search* pattern is a slight variation of *filtered-counting*. Suppose we wish to see if a value is present in an array. We can use a filtered-counting approach; if the count is greater than zero, we know that the item was indeed in the array.

```
    int
    occurrences(int target,int *items,int itemCount)
        {
        int i,count;
        count = 0;
        for (i = 0; i < itemCount; ++i)
            {
            if (items[i] == target)
                {
                ++count;
                }
            }
        return count;
        }

    int
    find(int target,int *items,int itemCount) // is target in items?
        {
        return occurrences(target,items,itemCount) > 0;
        }
```

In this case, *occurrences* implements the filtered-count pattern and helps *find* do its job. We designate such functions as *occurrences* and the like, *helper functions*. However, we can improve the efficiency of *find* by having it partially implement the filtered-count, but short-circuiting the search once the target item is found. We do this by returning from the body of the loop:

```
    int
    find(int target,int *items,int itemCount)
        {
        int i;
        for (i = 0; i < itemCount; ++i)
            {
            if (items[i] == target)
                {
                return 1;               // short-circuit! return true
                }
            }
        return 0;   //must not be in the array, return false
        }
```

When the array is empty, we return false, as indicated by the last line of the function, because the loop never runs and thus the return inside the loop can never be reached. In the same vein, if no item matches the target, we cannot return true, and eventually the loop exits and false is returned.

As a beginning programmer, however, you should be wary of returning from the body of a loop. The reason is most beginners end up defining the function this way instead:

```
    int
    find(int target,int *items,int itemCount)
        {
        int i;
        for (i = 0; i < itemCount; ++i)
            {
            if (items[i] == target)
                return 1;
            else
                return 0;
            }
        }
```

The first problem with this definition is that the compiler generates a warning indicating that it is possible for the function to complete without explicitly returning a value. The second problems is that the function fails to find the target most times. Unfortunately, it works correctly under some conditions, which is why thorough testing of code is always a good thing. If you cannot figure out why this version generates the warning and why it fails under some conditions but succeeds under others, you most definitely should stay away from placing returns in loop bodies.

15.8 The *extreme* pattern

Often, we wish to find the largest or smallest value in an array. Here is one approach, which assumes that the first item is the largest and then corrects that assumption if need be:

```
    int i,largest = items[0];
    for (i = 0; i < itemCount; ++i)
        {
        if (items[i] > largest)
            {
            largest = items[i];
            }
        }
    //largest now holds the largest item
```

When this loop terminates, the variable *largest* holds the largest value. We can improve the loop slightly by noting that the first time the loop body evaluates, we compare the putative largest value against itself, which is a worthless endeavor. To fix this, we can start the index variable *i* at 1 instead:

```
int i,largest = items[0];
for (i = 1; i < itemCount; ++i) //start comparing at index 1
    {
    if (items[i] > largest)
        {
        largest = items[i];
        }
    }
//largest now holds the largest item
```

Novice programmers often make the mistake of initialing setting *largest* to zero and then comparing all values against *largest*, as in:

```
int i,largest = 0;
for (i = 0; i < itemCount; ++i)
    {
    if (items[i] > largest)
        {
        largest = items[i];
        }
    }
//largest may hold an incorrect value!
```

This code appears to work in some cases, namely if the largest value in the array is greater than or equal to zero. If not, as is the case when all values in the array are negative, the code produces an spurious result of zero as the largest value.

15.9 The *extreme-index* pattern

Sometimes, we wish to find the index of the most extreme value in an array rather than the actual extreme value. In such cases, we assume index zero holds the extreme value:

```
int i,ilargest = 0;
for (i = 0; i < itemCount; ++i)
    {
    if (items[i] > items[ilargest])
        {
        ilargest = i;
        }
    }
```

Here, we successively store the index of the largest value seen so far in the variable *ilargest*.

15.10 The *filter* pattern

A special case of a filtered-accumulation is called *filter*. Instead of summing the filtered items (for example), we keep the filtered items in the array. The modified array is said to be a *reduction* of the original array.

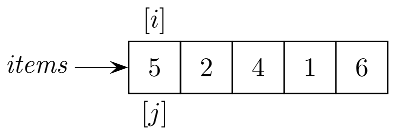
Suppose we wish to remove the odd numbers from an array. The code looks very much like the *sumEvens* function, but instead of adding in the desired item, we move the even numbers to the front of the array, overwriting any odd numbers or any previously moved even items.

```
int
removeOdds(int *items,int itemCount)
    {
    int i,j;
    j = 0;
    for (i = 0; i < itemCount; ++i)
        if (isEven(items[i]))
            {
            items[j] = items[i];
            ++j;
            }
    return j;
    }
```

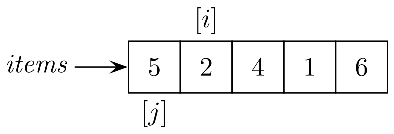
The *removeOdds* function needs to implement the function pattern, because we need to return the number of items that were kept in the array. This number becomes the new size of the array. The caller of the *removeOdds* function is responsible for updating the size of the array:

```
int data[] = {1,2,3,4,5,6};
int size = sizeof(data) / sizeof(int);
size = removeOdds(data,size);
```

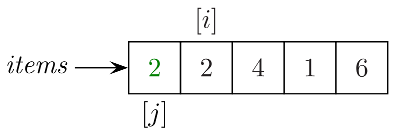
Here is a pictorial view of the workings of *removeOdds*. Suppose we start with items pointing to this array:



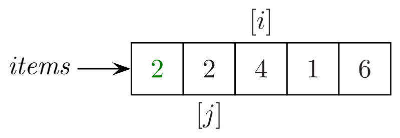
Both *i* and *j* start at zero, so they have been placed above and below slot 0. The brackets around the variables are to remind us these variables are used as indices into the array. When the first element is checked, we see that it is odd. We drop to the bottom of the loop and increment *i*, leaving us at:



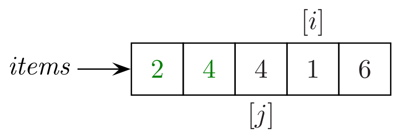
Now, we check the element at index *i* and we see it is even. We copy it to the slot at index *j*, overwriting the 5 that was there originally:



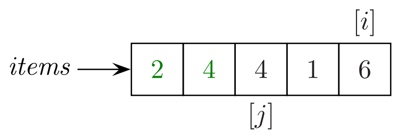
We then increment *j* and then, at the bottom of the loop, increment *i*. Now the situation is:



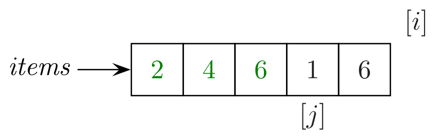
Again, the element at index i is even, so we copy it to slot j. We then increment both i and j:



The element at i is odd, so we simply increment i for this iteration of the loop:



Now the element at i is even, so we copy it to slot j. After incrementing both j and i, we have:



At this point, i is equal to *itemCounts*, the loop ends, and we return j as the new size of the array. In the example, j has a value of 3. This is as desired, as there were three even numbers in the original array.

Of course, we are being a bit wasteful here, in that there may be unused slots at the end of the array. You will note that 1 and 6 remain in the array, but they have no effect because the caller of *removeOdds* will believe the array has shrunk, assuming the caller properly resets the size of the array:

```
//test (compile with loops.c)
#include "loops.h"

int i;
int a[] = {5,2,4,1,6};
int aSize  = sizeof(a) / sizeof(int);

aSize = removeOdds(a,aSize);

for (i = 0; i < aSize; ++i)
    {
    printf("[%d]",a[i]);
    }
printf("\n");
```

This code should print out:

```
[2][4][6]
```

as desired. Since the original array is modified, we say that *removeOdds* is a *destructive* procedure. We might also say the *removeOdds* implements a *destructive filtering* pattern.

If *removeOdds* instead allocated a new array to store the even numbers, filled that array with the even numbers found in items, and returned the newly allocated and filled array, then *removeOdds* would implement a non-destructive filtering pattern:

```
int
removeOdds2(int *items,int itemCount,int *evens) //non-destructive
    {
    int i,j;
    j = 0;
    for (i = 0; i < itemCount; ++i)
        if (isEven(items[i]))
            {
            evens[j] = items[i];
            ++j;
            }
    return j;
    }
```

In this version, we are required to pass in an array to hold the even values:

```
//test (compile with loops.c)
#include "loops.h"
int i;
int a[] = { 5, 2, 4, 1, 6 };
int aSize = sizeof(a) / sizeof(int);
int b[] = { 0, 0, 0, 0, 0 };
int bSize = sizeof(b) / sizeof(int);

bSize = removeOdds2(a,aSize,b);

for (i = 0; i < bSize; ++i)
    printf("[%d]",b[i]);
printf("\n");
```

Of course, the caller must ensure that the array *b* is large enough to hold all the even values in *a*. Most often, the array *b* would be allocated dynamically, rather than statically:

```
...
int *b = (int *) malloc(sizeof(int) * aSize);
//the test that malloc succeeded omitted
int bSize = aSize;

bSize = removeOdds2(a,aSize,b);
...
free(b);
```

Rather than call *malloc* and then check if the allocation succeeded, we can write a wrapper function that does the allocation *and* checking:

```
char *
allocate(int size)
    {
    char *s = malloc(size); //size given in bytes
    if (s == 0)
        {
        fprintf(stdin,"allocate: out of memory\n");
        exit(1);
        }
    return s;
    }
```

Now we can use *allocate* instead of *malloc*, obviating the need for us to write the allocation checking code. You can find a version of *allocate* in the *scanner.c* file. The code fragment above becomes:

```
...
int *b = (int *) allocate(sizeof(int) * aSize);
int bSize = aSize;

bSize = removeOdds(a,aSize,b);
...
free(b);
```

From here on out, we will use *allocate* instead of *malloc*. You would need to include the definition of *allocate* in your code if you wish to use it in your programming.

15.11 The *map* pattern

Mapping is a task closely coupled with that of filtering, but rather than collecting certain items, as with the *filter* pattern, we collect all the items. As we collect, however, we transform each item as we collect it, using a passed in function. The basic destructive pattern looks like this, with the formal parameter *f* holding the function used to transform the original items:

```
void
map(int *items,int itemCount,int (*f)(int)) //destructive
    {
    int i;
    for (i = 0; i < itemCount; ++i)
        items[i] = f(items[i]);      //pass items[i] through function f
    }
```

The type of the formal parameter *f* looks rather odd. Recall, from the last section of the chapter Chapter **??**, how we derive the type of a variable that can point to a function. Let's review with the *square* function, whose definition is:

```
int square(int x) { return x * x; } //definition
```

and whose signature is:

```
int square(int);                    //signature
```

First, we wrap the name of the function in the function signature with parentheses:

```
int (square)(int)
```

Then we place an star in front of the function name (inside the parentheses we just added):

```
int (*square)(int)
```

Finally, substitute the formal parameter name for *square*:

```
int (*f)(int)
```

The formal paramter *f* has type `int (*)(int)`.

So any time you wish to pass a function to another function, you can follow this technique to derive the type of the formal parameter that receives the passed-in function.

Suppose we wish to subtract one from each element in an array. First we need a transforming function that reduces its argument by one:

```
int decrement(int);                      //signature
...
int decrement(int x) { return x - 1; }  //definition
```

Now we can "map" the *decrement* function over an array of numbers:

```
//test (compile with loops.c)
#include "loops.h"
int i;
int a[] = { 5, 2, 4, 1, 6 };
int aSize = sizeof(a) / sizeof(int);

map(a,aSize,decrement);

for (i = 0; i < aSize; ++i)
    printf("[%d]",a[i]);
printf("\n");
```

For output, we get:

```
[4][1][3][0][5]
```

as expected.

A non-destructive version of *map* is similar to the non-destructive version of *filter*; we need to pass in the accepting array:

```
void
map2(int *items,int itemCount,int (*f)(int),int *mapped) //non-destructive
    {
    int i;
    for (i = 0; i < itemCount; ++i)
        mapped[i] = f(items[i]);      //pass items[i] through function f
    }
```

The array *mapped* should be the same size as the array *items*.

15.12 The *append* pattern

Sometimes, we wish to create a third array that contains the elements of two arrays. The *append pattern*, places the elements of second array after the elements of the first array with all elements placed in the third array:

```
...
//a and b are integer arrays with sizes aSize and bSize
cSize = aSize + bSize;
c = (int *) allocate(cSize * sizeof(int));

append(a,aSize,b,bSize,c);
```

The definition of *append* might look like this:

```
void
append(int *a,int aSize,int *b,int bSize,int *c)
    {
    int i,spot;
    spot = 0;
    for (i = 0; i < aSize; ++i)
        {
        c[spot] = a[i];
        ++spot;
        }
    for (i = 0; i < bSize; ++i)
        {
        c[spot] = b[i];
        ++spot;
        }
    //c now holds all of a and b
    }
```

We use index *i* to walk through the source arrays *a* and *b* and the index *spot* to keep track of where we are in the destination array c.

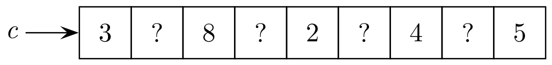
15.13 The *shuffle* pattern

When we wish to combine two arrays into a third array, sometimes we wish to shuffle the items in the source arrays, much like shuffling a deck of cards, rather than appending them.

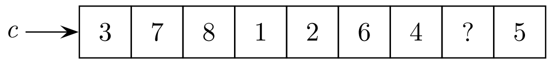
Here is one attempt. The *shuffle* function places the items from the first source array in the even locations of the destination array. Likewise, it places items from the second source into the odd locations:

```
int *
shuffle(int *a,int aSize,int *b,int bSize)
    {
    int *c; //a pointer to the destination array
    int i,spot;

    c = malloc(sizeof(int) * (aSize+bSize)); //malloc failure test omitted

    spot = 0;                           //the first even slot
    for (i = 0; i < aSize; ++i)
        {
        c[spot] = a[i];
        spot += 2;                      //skip to the next even slot
        }
    spot = 1;
    for (i = 0; i < bSize; ++i)     //the first odd slot
        {
        c[spot] = b[i];
        spot += 2;                      //skip to the next odd slot
        }
    //c now holds all of a and b
    return c;
    }
```

Note that this first version only works if the source arrays are the same size. Suppose the first source has the elements 3, 8, 2, 4, and 5, while the second has the elements 7, 1, and 6. After placing the first source, the destination arrays looks like this:



The question marks signify holes in the array. These holes are slots that have not been explicitly filled and thus contain garbage, due to the way C dynamically allocates arrays. After adding in the second source, we have:



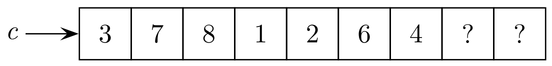
Note that holes remain at the end of the destination array. There is a worse problem, however. Presumably, the destination array was allocated to be just big enough to hold the all the elements in the source arrays. In our example, that would be a size of 8. However, the last element placed, the 5, was placed at index 8, which would be beyond the extent of the destination array. It is clear, for mismatched source arrays, we need a better solution.

Here is one approach: we shuffle the two arrays, as before, but stop when we run out of items in one of the source arrays. We then fill out the remainder of the destination array with the unused elements in the other source array.

To implement this second approach, we will combine the two loops from the first attempt at *shuffle*, but limit the loop's run to the smaller of the two source array sizes:

```
    spot = 0;
    for (i = 0; i < aSize && i < bSize; ++i)
        {
        c[spot] = a[i];     //spot must be even here
        c[spot+1] = b[i];   //spot+1 therefore must be odd
        spot += 2;
        }
```

After this loop runs, our destination array looks like:



with *spot* referencing the slot of the first question mark. The variable *i*, on the other hand, holds the index of the next element that should go into the destination array. We don't know which, so we will attempt to copy elements from *both* sources:

```
    while (i < aSize)
        {
        c[spot] = a[i];
        ++spot;
        ++i;
        }

    while (i < bSize)
        {
        c[spot] = b[i];
        ++spot;
        ++i;
        }
```

How can this possibly work when one of the sources is all used up? Suppose the first source is exhausted. We know that *i* must be equal to *aSize* in that case. Therefore, the first *while* loop test immediately fails and the loop body is never executed. We continue on to the next loop, where *i* must hold the index of first unused element in the second source. We continue to copy elements over into the destination array until the source is exhausted.

Conversely, if the *for* loop ends with the second source exhausted, we copy over elements from the first source. At this point, the variable *i* is greater than the size of the second source, so the second *while* loop immediately fails. In either case, the destination array holds all the elements of the source arrays with the elements shuffled as much as possible. Putting it all together yields the following implementation for shuffle:

```
    int *
```

```
shuffle(int *a,int aSize,int *b,int bSize)
    {
    int *c; //a pointer to the destination array
    int i,spot;

    c = malloc(sizeof(int) * (aSize+bSize)); //malloc failure test omitted

    //shuffle until one source is exhausted
    spot = 0;
    for (i = 0; i < aSize && i < bSize; ++i)
        {
        c[spot] = a[i];     //spot must be even here
        c[spot+1] = b[i];   //spot+1 therefore must be odd
        spot += 2;
        }

    //i is either equal to aSize or bSize
    //spot holds the index of the 1st available slot in c

    // copy over leftover elements in a, if any
    while (i < aSize)
        {
        c[spot] = a[i];
        ++spot;
        ++i;
        }

    // copy over leftover elements in b, if any
    while (i < bSize)
        {
        c[spot] = b[i];
        ++spot;
        ++i;
        }

    //c now holds the shuffle of a and b
    return c;
    }
```

With this new version, what happens when the sizes of the source arrays are equal? The superior student will quickly ascertain the validity of the new version in this case.

One last note: the three pieces of a *for* loop, INIT, LIMIT, and STEP, are all optional. We could replace the first *while* loop with this *for* loop:

```
    for ( ; i < aSize; ++i) //i starts at its previous value
        {
        c[spot] = a[i];
        ++spot;
        }
```

The second *while* loop could be replaced in a similar fashion.

15.14 The *merge* pattern

With the *shuffle* pattern, we always took the head elements from both arrays at each step in the shuffling process. Sometimes, we wish to place a constraint of the choice of elements. For example, suppose the two arrays to be combined are sorted and we wish the resulting array to be sorted as well. The following example shows that shuffling does not always work:

```
//test (compile with scanner.c and loops.c)
#include "scanner.h"
#include "loops.h"

int i;
int a[] = {1,4,6,7,8};   //a is sorted
int b[] = {2,3,9};       //b is sorted
int *c;
int aSize,bSize,cSize;

aSize = sizeof(a) / sizeof(int);
bSize = sizeof(b) / sizeof(int);

c = shuffle(a,aSize,b,bSize);
cSize = aSize + bSize;

for (i = 0; i < cSize; ++i)
    printf("[%d]",c[i]);
printf("\n");
```

Running this code gives us the following output:

```
[1][2][4][3][6][9][7][8]
```

Clearly, the destination array c is not sorted.

The *merge* pattern is used to ensure the resulting array is sorted and is based upon the *filtered-accumulate* pattern. The twist is we only accumulate an item *if* it is the smallest item in the two arrays that has not already been accumulated. We start by keeping two index variables, one pointing to the smallest element in the first source and one pointing to the smallest element in second source. Now, we loop, similar to *shuffle*. The difference is we will only advance the index variable of a source array *if* we use the smallest of its unused elements:

```
i = 0;          //index of the smallest unused element in a
j = 0;          //index of the smallest unused element in b
spot = 0;
while (i < aSize && j < bSize)
    {
    if (a[i] < b[j])    //a's element is smaller, use it
        {
        c[spot] = a[i];
        spot++;
        i++;
        }
    else                //b's element is smaller, use it
        {
        c[spot] = b[j];
        spot++;
        j++;
        }
    }
```

Inside the loop, we test to see if the smallest unused element in *a* is smaller than the smallest unused element in *b*. If it is, we copy the element from *a* to *c* and increment *a*'s index variable, "using up" the smallest unused value. Likewise, we do the same for *b* if *b*'s element is smaller.

The *while* loop ends when one of the source arrays is exhausted. At this point, we just copy over the elements in the unexhausted source, just the same as we did for *shuffle*. Putting it all together yields:

```
    int *
    merge(int *a,int aSize,int *b,int bSize)
        {
        int *c; //a pointer to the destination array
        int i,j,spot;

        c = malloc(sizeof(int) * (aSize+bSize)); //malloc failure test omitted

        //merge until one source is exhausted
        spot = 0;
        for (i = 0,j = 0; i < aSize && j < bSize; ++spot)
            {
            if (a[i] < b[j])    //a's element is smaller, use it
                {
                c[spot] = a[i];
                i++;
                }
            else                //b's element is smaller, use it
                {
                c[spot] = b[j];
                j++;
                }
            }

        //i is either equal to aSize or j is equal to bSize
        //spot holds the index of the 1st available slot in c

        // copy over leftover elements in a, if any
        for ( ; i < aSize; ++i)
            {
            c[spot] = a[i];
            ++spot;
            }

        // copy over leftover elements in b, if any
        for ( ; j < bSize; ++j)
            {
            c[spot] = b[j];
            ++spot;
            }

        //c now holds the shuffle of a and b
        }
```

The main *while* loop has been rewritten as a *for* loop by taking advantage of *for* loop flexibility. Note we can initialize two (or more) variables in the INIT portion of the loop by separating the initializations with a comma. We also moved the increment of spot to the STEP portion since we increment spot regardless of the which array held the smaller value. We do *not* want to increment i and j in the STEP portion since those indices only get incremented when their arrays hold the smallest unused elements.

15.15 The *fossilized* pattern

Sometimes, a loop is so ill-specified that it never ends. This is known as an *infinite loop.* Of the two loops we are investigating, the *while* loop is the most susceptible to infinite loop errors. One common mistake is the *fossilized* pattern, in which the index variable never changes so that the loop condition never becomes false:

```
    i = 0;
    while (i < n)
        {
        printf("%d\n",i);
        }
```

This loop keeps printing until you terminate the program with prejudice. The reason is because i never changes; presumably a statement to increment i at the bottom of the loop body has been omitted.

15.16 The *missed-condition* pattern

With the missed condition pattern, the index variable is updated, but it is updated in such a way that the loop condition never evaluates to false.

```
    i = getAnInteger();
    while (i != 0)
        {
        printf("%d\n",i);
        i -= 2;
        }
```

If the *getAnInteger* returns a positive, even number, then the loop terminates as expected. However, if *getAnInteger* returns an odd or negative number, the loop never ends. Suppose i starts out with the value 5. The loop test keeps succeeding and the value of i keeps being printed:

```
    5
    3
    1
    -1
    -3
    -5
    -7
    ...
```

Here, the variable i has skipped over the value used for terminating the loop; i never becomes zero. Thus, the loop condition never fails and the loop becomes infinite.

Chapter 16

Input and Loops

You can download the functions defined in this chapter with the following commands:

```
wget troll.cs.ua.edu/ACP-C/moreInput.c
wget troll.cs.ua.edu/ACP-C/moreInput.h
```

These files will help you run the test code listed in this chapter.

16.1 Converting command line arguments en mass

Now that we have learned how to loop, we can perform more sophisticated types of input. Suppose all the command-line arguments are integers that need to be converted from their string versions stored in *sys.argv*. We can use a loop and the accumulate pattern to accumulate the converted string elements:

```
void
convertArgsToNumbers(int argc,char **argv,int *numbers)
    {
    int i;
    // start at 1 to skip over program file name
    for (i = 1; i < argc; ++i)
        numbers[i-1] = atoi(argv[i]);
    }
```

Note the use of `i-1` as the index for the numbers array. This makes sure the second element of *argv* corresponds to the first element of numbers, and so on. A careful analysis of this function yields the following insight: the function implements the *map pattern*. The function being mapped is *atoi* and we map over the *argv* array of strings.

We can test our conversion function:

```
//test (compile with scanner.c and moreInput.c)
#include "scanner.h"
#include "moreInput.h"
int i;
int *nums;

nums = allocate(sizeof(int) * (argc-1));
```

```
    convertArgsToNumbers(argc,argv,nums);

    printf("original: ");
    for (i = 0; i < argc; ++i)
        printf("[\"%s\"]",argv[i]);
    printf("\n");

    printf("converted: ");
    for (i = 0; i < argc-1; ++i) //one fewer item in nums
        printf("[%d]",nums[i]);
    printf("\n");
```

Note the use of the string directive in the first loop and the integer directive in the second. Let's look at our test code's behavior, assuming we place the code in *convert.c*:

```
$ gcc -Wall -g -o convert convert.c scanner.c
$ convert 1 34 -2
original:  ["convert"]["1"]["34"]["-2"]
converted: [1][34][-2]
```

We see that the elements of *argv* are indeed strings. Otherwise, the `"%s"` directive would not have worked.

16.2 Reading individual items from files

we can read a unknown number of items from a file by using functions found in the scanner module. When doing so, we always follow this pattern:

```
open the file
read the first item
while the read was good
    process the item
    read the next item
close the file
```

In C, we tell if the read was good by checking the the file pointer for the *end-of-file* condition. We do this with the *feof* function, which returns true if the read failed and false if the read succeeded. With that detail, the while loop is refined to:

```
FILE *fp = fopen(fileName,"r");
//read the first item
...
while (!feof(fp))
    {
    //process the item
    ...
    //read the next item
    ...
    }
```

Processing files a line at a time

Here is a function that version reads and writes all the lines in a file, one line at a time. In addition, the function returns the number of lines processed. It makes use of the *readLine* function in the scanner module.

```
int
copyFile(char *inFile,char *outFile)
    {
    FILE *in = fopen(inFile,"r");       //check for failed open omitted
    FILE *out = fopen(outFile,"w");     //check for failed open omitted
    int count = 0;
    char *line;

    line = readLine(in);        //readLine is a scanner module function
    while (!feof(in))
        {
        ++count;
        fprintf(out,"%s\n",line);
        free(line);             //we're done with line, so free it
        line = readLine(in);
        }

    //always close your files
    fclose(in);
    fclose(out);

    return count;
    }
```

Notice we used the counting pattern, augmented by printing out the current line to the output file every time the count was incremented. Since the *readLine* function allocates space for the line it reads, we must free it when we are done with it. Note that you only need to free items read with the *readLine*, *readToken*, and *readString* functions. You should only free these items when you are absolutely done with them.

Using functions in the scanner module

Recall that the scanner module contains a number of useful function for reading input. To use these functions, you must first download the module (see the first section of this chapter). Then, you need tell the compiler what those functions look like before you call them. You do that by including *scanner.h* in any source code file that calls scanner functions:

```
#include "scanner.h"
```

Finally, you need to compile the scanner source in with your main program:

```
gcc -Wall -g mainProgram.c scanner.c
```

Here is a program that utilizes the *copyFile* function defined above:

```
//test (compile with scanner.h and moreInput.h)
```

```
#include "scanner.h"
#include "moreInput.h" //contains prototype for copyFile

int count;

count = copyFile("copycat.c","junk.c");
printf("%d lines copied.\n",count);
```

If this program is placed in a file *copycat.c*, compiled, and then run:

```
$ gcc -Wall -g -o copycat cat.c scanner.c
$ copycat
```

Then you should find a copy of *copycat.c* in the file *junk.c*.

16.3 Patterns for reading input

The same patterns that are commonly used for processing arrays can be used for reading input. For example, here is a function that implements a filtered count:

```
int
countSmallNumbers(char *fileName)
    {
    FILE *fp = fopen(fileName,"r");    //check for failed open omitted
    int count;
    int number;

    count = 0;
    number = readInt(fp);              //read the first integer
    while (!feof(fp))                  //check if the read was good
        {
        if (number < SIZE_LIMIT)       //smaller than SIZE_LIMIT, then small!
            ++count;
        number = readInt(fp);          //read the next integer
        }
    fclose(fp);                        //always close files when done
    return count;
    }
```

Note that the use of the standard reading pattern: opening the file, making the first read, testing if the read was good, processing the item read (by counting it), reading the next item, and finally closing the file after the loop terminates. Using the scanner functions always means performing the five steps as given in the comments.

16.4 Reading Items into an Array

Note that the *countSmallNumbers* function is doing two things, reading the tokens and also counting the number of short tokens. It is said that this function has two *concerns*, reading and counting. A fundamental principle of Computer Science is the *separation of concerns*. To separate the concerns, we have one function read the tokens, storing them into an array (reading and storing is considered to be a single concern). We then have another function count the small numbers in the array. Thus, we will have separated the two concerns into separate functions, each with its own concern.

Reading in an undetermined number of items from a file is problematic. We don't know how big to allocate our array that will hold the items that we read. We have three choices:

1. allocate a really big array and stop reading if it fills up
2. read the file twice, the first time to count, the second to store
3. grow the array if it fills up

The first choice is a very poor one. The second works well, but is inefficient. Still, it is a viable strategy, so let's explore that approach.

16.4.1 Reading a file to size an array

For this example, we will count the number of short tokens in a file. To count tokens in a file, we modify the *countSmallNumbers* function above to read in tokens instead of integer and also removing the *if* that protects the increment of the count. Call this function *countTokens*.

```
int
countTokens(char *fileName)
    {
    FILE *fp = fopen(fileName,"r");    //check for failed open omitted
    int count;
    char *token;

    count = 0;
    token = readToken(fp);             //read the first integer
    while (!feof(fp))                  //check if the read was good
        {
        ++count;
        free(token);                   //we've counted it, so free it
        token = readToken(fp);         //read the next integer
        }
    fclose(fp);                        //always close files when done
    return count;
    }
```

Once we have counted the token, then we can free it. We call *countTokens* to get the number of tokens in the file:

```
int count = countTokens(fileName);
```

Next, we allocate the array with exactly the right number of slots:

```
char **tokens = malloc(sizeof(char *) * count);
```

Finally, we fill the array with a call to a procedure named *fillArrayWithTokens*:

```
    fillArrayWithTokens(fileName,tokens);
```

The *fillArrayWithTokens* function is left as an exercise for the reader.

16.4.2 Resizing an array as needed

More sophisticated C programs use the third choice. The scanner module contains a function that will help us do that, *reallocate*. The *reallocate* function is merely a wrapper for the *realloc* function, much in the same way *allocate* is a wrapper for *malloc*. What *realloc* and *reallocate* do, when given a previously allocated array and a new size, is to allocate a new array with the given size and copy over elements from the old array to the new array. These routines return the new array, so the old array is often reassigned with the return value. For example, to reallocate an integer array *items* with an original size of *size* to a a new size of *newSize*, we would do something like this:

```
    size = newSize;                                //reset the size
    items = reallocate(items,sizeof(int) * size);  //reset the array
```

The *realloc* and *reallocate* functions can also be used to shrink an array as well as grow it.

Here is the reading (and storing) function, which grows an array as needed using *reallocate*:

```
    char **
    readTokens(char *fileName,int *finalSize)
        {
        FILE *fp = fopen(fileName,"r");     //check for failed open omitted
        int size = 10;                      //initial size of destination array
        char *token;
        int count;

        //allocate the destination array
        //char ** is a pointer to an array of strings (tokens are strings)
        char **items = allocate(sizeof(char *) * size);

        count = 0;
        token = readToken(fp);
        while (!feof(fp))
            {
            if (count == size)              //array is full!
                {
                // grow the array by doubling its size
                size = size * 2;
                items = reallocate(items,sizeof(char *) * size);
                //now there is enough room
                }
            items[count] = token;           //DO NOT FREE THE TOKEN!
            ++count;
            token = readToken(fp);
            }
        fclose(fp);

        //shrink the array to 'count' number of elements
        items = reallocate(items,sizeof(char *) * count);
        //count holds the number of items, store it in *finalSize
        *finalSize = count;
        return items;
        }
```

The *readTokens* function is complicated by the fact that it needs to return two things, the array of tokens that were found in the file, plus the size of the array. Unfortunately, C only allows a function to return one thing. So, we have chosen to return the array and to update one of the arguments to the function, which we do on the second to the last line of the function. To use *readTokens*, we need to define a variable to hold the size of the array and a variable to point to that array:

```
    int size;
    char **tokenArray;

    tokenArray = readTokens(fileName,&size); //send the address of size
    //size now holds the number of elements in the tokenArray
    //tokenArray points to the array of tokens (each token is a string)
```

Another thing that is different about the *readTokens* function is we do not free the token after it has been read. Instead, we store it in the destination array and leave it to the caller of the function to free the tokens and the array itself when it is done with them.

Finally, note that *readToken* shrinks its array to its final size after filling and growing. This is because as much as 50% of the array might not be used[1].

Next, we implement the function to perform the filtered counting. Instead of passing the file name, as before, we pass the array of tokens that were read, plus the size of that array:

```
    int
    countShortTokensInArray(char **items,int size)
        {
        int i,count;

        count = 0;
        for (i = 0; i < size; ++i)
            {
            if (strlen(items[i]) < SHORT_LIMIT)
                count += 1;
            }
        return count;
        }
```

The *countTokens* function is now simpler than the original function[2]. This makes it easier to fix any errors in the function since you can concentrate on the single concern implemented by that function.

We can test our two functions with the following code:

```
    //test (compile with scanner.c and moreInput.c)
```

[1]This happens if adding the last element causes the array to grow.

[2]The *readTokens* function is more complex, but its complexity arises from the desire to not know the number of tokens in advance.

```
#include "scanner.h"
#include "moreInput.h"

int count;
int size;
char **tokens;

count = countShortTokens("tester.c");
printf("counting tokens directly: %d tokens\n",count);

tokens = readTokens("tester.c",&size);
count = countShortTokensInArray(tokens,size);
printf("counting tokens in two stages: %d tokens\n",count);
```

16.5 Reading Records into an Array

Often, data in a file is organized as *records*, where a record is just a collection of consecutive and related tokens. Each token in a record is known as a *field*. Suppose every four tokens in a file comprises a record:

```
"Amber Smith"       President   32   97000.05
"Thad Jones"        Assistant   15   89000.42
"Ellen Thompson"    Hacker       2  147000.99
```

Typically, we define a function to read one collection of tokens at a time. Here is a function that reads a single record:

```
char **
readRecord(FILE *fp)                // we pass the file pointer in
    {
    char *name,*title,*years,*salary;
    char **record;

    name = readString(fp);           //name is a string, not a token

    if (feof(fp)) { return 0; }      // no record, return null

    title = readToken(fp);
    years = readToken(fp);
    salary = readToken(fp);

    //make an empty record
    record = allocate(sizeof(char *) * 4);  //there are four fields

    //fill the record
    record[0] = name;
    record[1] = title;
    record[2] = years;
    record[3] = salary;

    return record;
    }
```

Note that we return either a record as an array or the null pointer (zero) if no record was read. Note that we store every field as a string, even though the last two fields are numbers. This is because arrays are homogeneous. In a later chapter when we learn about structures, we will be able to create records that store heterogeneous data, such as years as an integer and salary as a real number.

To total up all the salaries, for example, we can use an accumulation loop (assuming the salary data resides in a file represented by the string *fileName*).

We do so by repeatedly calling *readRecord*, remembering to convert the string representing the salary in the record to a real number using *atof*:

```
double
totalPay(char *fileName)
    {
    FILE *fp;
    char **record;
    double total;

    fp = fopen(fileName,"r");       //check for failure to open omitted
    total = 0;
    record = readRecord(fp);
    while (!feof(fp))               //see if the read was good
        {
        total += atof(record[3]);
        freeRecord(record);         //done with record, free it
        record = readRecord(fp);
        }
    fclose(fp);                     //always close file
    return total;
    }
```

Note that it is the job of the caller of *readRecord* to open the file, repeatedly send the file pointer to *readRecord*, and close the file pointer when done. As, before we always check if the read was good using *feof*, although we could equivalently test if *readRecord* returns zero.

Note also the use of a function named *freeRecord*. If we just return the space occupied by the record with free:

```
free(record);
```

we will release the record array, but not the individual elements, which also need to be freed. So here is an implementation of *freeRecord*:

```
void
freeRecord(char **r)
    {
    //free the fields first
    free(r[0]); //name
    free(r[1]); //title
    free(r[2]); //years
    free(r[3]); //salary
    //now free the record array
```

```
        free(r);
        }
```

Both *totalPay* and *freeRecord* suffer from a stylistic flaw. It uses those magic numbers we read about in Chapter **??**. It is not clear from the code that the field at index three is the salary. To make the code more readable, we can set up some constants in the global scope (so that they will be visible everywhere): A second issue for *totalPay* is that that the function has two concerns (reading and accumulating). We will fix the magic number problem first, by placing constants representing the indices of the various fields of a record at the top of the file (usually after the includes but before the function prototypes):

```
    #define NAME 0
    #define TITLE 1
    #define SERVICE 2
    #define SALARY 3
```

In C, we can create constants by using the `#define` preprocessor directive. Our accumulation loop now becomes:

```
    total = 0;
    record = readRecord(fp);
    while (!feof(fp))
        {
        total += atof(record[SALARY]);
        record = readRecord(fp);
        }
```

The body of *freeRecord* now becomes:

```
    free(r[NAME]);
    free(r[TITLE]);
    free(r[YEARS]);
    free(r[SALARY]);
    ...
```

We can also rewrite our *readRecord* function to use the field index constants:

```
    ...
    //fill the record
    result[NAME] = name;
    result[TITLE] = title;
    result[SERVICE] = service;
    result[SALARY] = salary;
```

Even if someone changes the constants to:

```
    #define NAME 3
    #define TITLE 2
    #define SERVICE 1
    #define SALARY 0
```

The code still works correctly. Now, however, the salary resides at index 0, but the accumulation loop is still accumulating the salary due to its use of the constant to access the salary.

16.6 Creating an Array of Records

We can separate the two concerns of the *totalPay* function by having one function read the records into an array and having another total up the salaries. An array of records is known as a *table*. Creating the table is just creating the record, but instead of storing tokens in a record array, we store records in a table array. Because token is an array of characters and characters have type `char`, a token has type `char *`. Every time we make an array to hold some type *X*, the type of the array is *X* `*`. A record, which is an array of tokens, has type `char **`. We simply add a star for each level of array. Since a table is an array of records, its type is `char ***`. If we wanted an array of tables, what would that array's type be?

Recall the *readTokens* function that stored tokens in a growing array. Here is a *readTable* function, which takes the same approach:

```
char ***
readTable(char *fileName,int *finalSize)
    {
    FILE *fp;
    int count;
    int size = 10;                  //initial size of destination array
    char **record;
    char ***table;

    fp = fopen(fileName,"r");     //check for failed open omitted

    //allocate the destination array
    table = allocate(sizeof(char **) * size);

    count = 0;
    record = readRecord(fp);
    while (!feof(fp))
        {
        if (count == size)              //array is full!
            {
            // grow the array by doubling its size
            size = size * 2;
            table = reallocate(table,sizeof(char **) * size);
            //now there is enough room
            }
        table[count] = record;           //DO NOT FREE THE RECORD!
        ++count;
        record = readRecord(fp);
        }
    fclose(fp);

    //shrink the array to 'count' number of elements
    table = reallocate(table,sizeof(char **) * count);

    //count holds the number of items, store it in *finalSize
    *finalSize = count;
    return table;
```

```
        }
```

The accumulation function is straightforward, now that records have been stored in an array:

```
    double
    totalPay2(char ***table,int size)
        {
        int i;
        double total = 0;

        for (i = 0; i < size; ++i)
            {
            char **record = table[i];
            total += atof(record[SALARY]);
            }

        return total;
        }
```

We can simply this function by removing the temporary variable *record*:

```
    double
    totalPay2(char ***table,int size)
        {
        int i;
        int total = 0;

        for (i = 0; i < size; ++i)
            {
            total += atof(table[i][SALARY]);
            }
        return total;
        }
```

Since a table is just an array, so we can walk it, accumulate items in each record (as we just did with salary), filter it and so on.

When we are done with the table, we need to free it. As with *freeRecord*, we cannot just free the table array, we also need to free what the array holds. Fortunately, the table holds records and we have already written *freeRecord*:

```
    void
    freeTable(char ***table,int size)
        {
        int i;
        //free the records first
        for (i = 0; i < size; ++i)
            {
            freeRecord(table[i]);
            }
        //now free the table array
        free(table);
        }
```

Tying it all together into a program, and putting some employment records into a file named *employees.dat* gives us:

```
    //test (compile with scanner.c and moreInput.c)
    #include "scanner.h"
    #include "moreInput.h"

    int size;
    char ***table = readTable("employees.dat",&size);
    double total = totalPay2(table,size);

    printf("total salary: $%.2f\n",total);
```

If *employees.dat* just contains the records of employees Smith, Jones, and Thompson above, then the output from this program should be:

```
    total salary: $333001.46
```

Chapter 17

Structures

You can download the functions defined or used in this chapter with the following commands:

```
wget troll.cs.ua.edu/ACP-C/employee.c
wget troll.cs.ua.edu/ACP-C/employee.h
```

These files will help you run the test code listed in this chapter.

17.1 Structures

Arrays in C are limited in that they are homogeneous and cannot grow as needed. To tackle the homogeneity problem, we will use something called a *structure*; a structure will allow us to collect heterogeneous data together. To solve the expandability problem, we will use structures to implement linked lists. We'll discuss linked-lists in subsequent chapters; for now, we will focus on the structures themselves.

Just like we can collect literal values together in an array and can collect actions together in a function, we can collect variables together in something called a *structure*. Suppose we wished to collect an integer, a string, and a double variable together. To do so, we would define a structure like this:

```
struct stuff
    {
    int x;
    char *s;
    double r;
    };
```

A structure in C is introduced with the keyword `struct`, followed by the name of the structure (in this case, *stuff*) followed by a collection of variable definitions enclosed in braces. Our *stuff* structure is composed of three variables, *x*, *s*, and *r*.

Like arrays, we can allocate a structure both statically and dynamically. A static allocation looks like this:

```
struct node n;
```

To access the individual variables in a statically allocated structure, we use the *dot* operator:

```
//test
struct stuff { int x; char *s; double r; }; //define the struct
struct stuff n;                             //allocate the struct
n.x = 42;
n.s = "hello, world!";
n.r = 3.14159;
printf("n.x is %d\n",n.x);
printf("n.s is %s\n",n.s);
printf("n.r is %f\n",n.r);
```

Running this code yields:

```
n.x is 42
n.s is hello, world!
n.r is 3.141590
```

Unlike statically allocated arrays, statically allocated structures can be returned from functions under certain circumstances. Such manipulations of structures can be inefficient, so we will focus on dynamically allocated structures. Here is the dynamically allocated version of the above code fragment:

```
//test
struct stuff { int x; char *s; double r; }; //define the struct
struct stuff *n;                            //create the pointer
n = malloc(sizeof(struct stuff));           //malloc failure test omitted
n->x = 42;
n->s = "hello, world!";
n->r = 3.14159;
printf("n->x is %d\n",n->x);
printf("n->s is %s\n",n->s);
printf("n->r is %f\n",n->r);
```

Note that the "dot" operator is replace by the "arrow" operator; the arrow is composed of a hyphen followed by a greater-than symbol.

17.2 Records as structures

Recall, from the Chapter **??**, the concept of a record as a collection of related data. A structure, because of its ability to bundle heterogeneous data together, makes an ideal repository for the data comprising a record. When we used an array to store a record, we needed to keep each field of the record as a string, even when some fields were clearly numbers. With structures, we are freed from that constraint.

Suppose, as before, our record is composed of name, title, years of service, and salary:

```
"Amber Smith"       President   32   97000.05
"Thad Jones"        Assistant   15   89000.42
"Ellen Thompson"    Hacker       2  147000.99
```

A structure to reflect such a record could be defined as:

```
struct employee
    {
    char *name;
    char *title;
    int years;
    double salary;
    };
```

Since typing `struct employee` is such an onerous task, many Computer Scientists will use the *typedef* mechanism to make an alias:

```
typedef struct employee
    {
    char *name;
    char *title;
    int years;
    double salary;
    } Employee;
```

Given this *typedef*, we can now use the terms `struct employee` and *Employee* interchangeably:

```
struct employee *e = malloc(sizeof(struct employee));
Employee *e = malloc(sizeof(Employee));
```

These dynamic allocations are equivalent. Once you have an employee record, it is extremely useful to define a function for displaying said record:

```
void
displayEmployee(Employee *e)
    {
    printf("Employee: %s\n",e->name);
    printf("    Title: %s\n",e->title);
    printf("    Years of Service: %d\n",e->years);
    printf("    Salary: $%.2f\n",e->salary);
    }
```

Given the *Employee* structure definition, a (renamed) version of the *readRecord* function from the Chapter **??**, becomes:

```
Employee *
readEmployeeRecord(FILE *fp)         // we pass the file pointer in
    {
    char *name;

    name = readString(fp);           //name is a string, not a token

    if (feof(fp)) { return 0; }      // no record, return null

    //make an empty record
```

```
        Employee *e = malloc(sizeof(Employee)); //malloc failure test omitted

        e->name = name;
        e->title = readToken(fp);
        e->years = readInt(fp);
        e->salary = readReal(fp);

        return e;
        }
```

To test our *readEmployeeRecord* function, we will successively read employee records from a file and print out the individual variable values for each record:

```
    void
    displayEmployeesInFile(FILE *fp)
        {
        Employee *e;

        e = readEmployeeRecord(fp);
        while (!feof(fp))
            {
            displayEmployee(e);
            e = readEmployeeRecord(fp);
            }

        fclose(fp);
        }
```

We use the standard reading pattern: read the first item before the loop – test if the read was successful – process the item read – read again at the bottom of the loop. In this case, processing the item means displaying the record.

Assuming the employee data can be found in the file *employees.dat*, we can implement our main:

```
    //test (compile with employee.c and scanner.c)
    #include "scanner.h"
    #include "employee.h"
    FILE *fp;
    if ((fp = fopen("employees.dat","r")) == 0)
        fprintf(stderr,"employees.dat could not be opened for reading\n");
    else
        displayEmployeesInFile(fp);
    fclose(fp);
```

17.3 Constructors, accessors, and mutators

You may have heard of the term *object-oriented* when it comes to programming languages. Object-oriented languages were created because organizing code along object lines can be a very good way to write programs. While the C language is not object-oriented, we can write our code in much the same way.

The fundamental unit in an object-oriented language is the *class*, which is roughly equivalent to structures in C. Given a class, *objects* are created using something called the *new* operator, which causes a new object to come into being. We will simulate the `new` operator by defining a function to dynamically allocate a structure and return it:

```
Employee *
newEmptyEmployee(void)
    {
    Employee *p = malloc(sizeof(Employee)); //malloc failure test omitted
    p->name = "";                           //set to the empty string
    p->title = "";                          //set to the empty string
    p->years = 0;                           //set to zero
    p->salary = 0.0;                        //set to zero
    return p;
    }
```

A good implementation initializes the variables within the structure. Given *p*, the pointer to the structure, recall that we access the individual variables in the structure by using the arrow operator, which is a hyphen followed by a greater-than symbol. We initialize all the variables stored in the structure to "empty" values.

A function such as *newEmptyEmployee* is known as a *constructor*, as it specifies how a structure should be constructed and initialized. We can make an alternate constructor whereby we pass in all the values stored in the structure:

```
Employee *
newEmployee(char *n,char *t,int y,double s)
    {
    Employee *p = malloc(sizeof(Employee)); //malloc failure test omitted
    p->name = n;
    p->title = t;
    p->years = y;
    p->salary = s;
    return p;
    }
```

Object-orientation also introduces the terms *accessors* and *mutators*. An accessor (sometimes the term *getter* is used) retrieves a component of an object or, in our case, the value of a variable inside the structure. Since we have four variables in our *Employee* structure, we will have four accessors:

```
char *getEmployeeName(Employee *e) { return e->name; }
char *getEmployeeTitle(Employee *e) { return e->title; }
int getEmployeeYears(Employee *e) { return e->years; }
double getEmployeeSalary(Employee *e) { return e->salary; }
```

A mutator (sometimes *setter*) replaces a component in an object:

```
void setEmployeeName(Employee *e,char *n) { e->name = n; }
void setEmployeeTitle(Employee *e,char *t) { e->title = t; }
void setEmployeeYears(Employee *e,int y) { e->years = y; }
void setEmployeeSalary(Employee *e,double s) { e->salary = s; }
```

A convention we will follow is that the structure will always be the first argument to a function that manipulates the structure.

Given these definitions, we can rewrite *readEmployeeRecord*:

```
Employee *
readEmployeeRecord(FILE *fp)         // we pass the file pointer in
    {
    char *name,*title;
    int years;
    double salary;

    name = readString(fp);           //name is a string, not a token

    if (feof(fp)) { return 0; }      // no record, return the null pointer

    name = name;
    title = readToken(fp);
    years = readInt(fp);
    salary = readReal(fp);

    return newEmployee(name,title,years,salary);
    }
```

We can also rewrite *displayEmployee* to use accessors:

```
void
displayEmployee(Employee *e)
    {
    printf("Employee: %s\n",getEmployeeName(e));
    printf("    Title: %s\n",getEmployeeTitle(e));
    printf("    Years of Service: %d\n",getEmployeeYears(e));
    printf("    Salary: $%.2f\n",getEmployeeSalary(e));
    }
```

Note how *readEmployeeRecord* and *displayEmployee* now give no clues as to how an employee record is stored. Indeed, with an alternate definition of the typedef *Employee*, with new versions of constructors and accessors, we could store an employee records as an array of four strings (as in the previous chapter) and both *readEmployeeRecord* as well as *displayEmployee* would still work as before, *without any changes*.

This idea of abstracting the internal details of how data is stored is a powerful tool in writing code that is easy to understand, modify, and maintain.

17.4 Storing collections of records

We have omitted from our discussions how we might store a collection of records. One possibility is to store them in an array.

17.4.1 Storing records in a dynamic array

The code for storing records in a dynamic array will be almost identical to the *readTable* function shown in the previous chapter. Recall, that we will need to return two values from the function that performs this task, the number of records found and the array that stores them:

```
Employee **
readAllEmployeeRecords(FILE *fp,int *finalSize)
    {
    int count;
    int size = 10;                //initial size of destination array
    Employee *record;
    Employee **table;

    //allocate the destination array
    table = allocate(sizeof(Employee *) * size);

    count = 0;
    record = readEmployeeRecord(fp);
    while (!feof(fp))
        {
        if (count == size)              //array is full!
            {
            // grow the array by doubling its size
            size = size * 2;
            table = reallocate(table,sizeof(Employee *) * size);
            //now there is enough room
            }
        table[count] = record;           //DO NOT FREE THE RECORD!
        ++count;
        record = readEmployeeRecord(fp);
        }
    fclose(fp);

    //shrink the array to 'count' number of elements
    table = reallocate(table,sizeof(Employee *) * count);

    //count holds the number of items, store it in *finalSize
    *finalSize = count;
    return table;
    }
```

A call to this function might look like:

```
int size;
Employee **table = readAllEmployeeRecords(fp,&size);
```

At this point, *table* points to an array of Employee record pointers and *size* holds the final size of that array.

One can see by comparing *readTable* from the previous chapter and *readAllEmployeeRecords* in this chapter, we see that the only changes are the return value, the type array of records (the table), the type of the individual record, and the name of the function to read a record. This is why the pattern approach to programming is so powerful. Once you understand the pattern, it is relatively simple to implement it for a variety of types.

17.5 Another way to store records

There is another way to store a collection of records: using a *linked list*. We investigate linked lists in the next chapter.

Chapter 18

Nodes

You can download the functions defined or used in this chapter with the following commands:

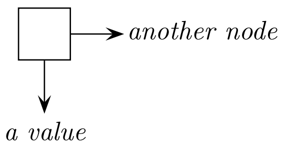
```
wget troll.cs.ua.edu/ACP-C/nodes.c
wget troll.cs.ua.edu/ACP-C/nodes.h
```

These files will help you run the test code listed in this chapter.

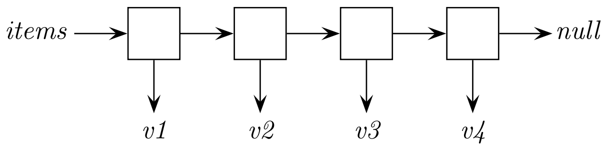
18.1 The Node Structure

A *list* is a data structure that has some benefits over arrays, the main one being it grows with hardly any effort. As such, it is useful for storing data that can grow without bound. Lists, at their very core, are composed of chains of *nodes*, so we will start our investigation into lists by first exploring nodes.

A node is a simple structure that bundles together two variables. The first variable holds a value; this value may be an integer, a real number, a pointer to a string or a record, or a myriad of other types. The second variable points to the next node in the list. So, a list is merely a chain of nodes, each node holding a value. Graphically, nodes are represented as a box with two arrows. The downward pointing arrow typically leads to the value, while the rightward pointing arrow points to another node. While the right arrow is truly a pointer, as it points to another node, the down arrow may or may not be a pointer. Typically, if the value of a node is a pointer, the down arrow will point to a box of some kind (or zero). Regardless, an arrow is used to point to a value; one uses context to decide whether the value is a pointer or a basic value.



When we chain a series of nodes together, we get a list:



where *items* is a variable that points to the first node, *v1* through *v4* are the values held in the list, and *null* represents the null pointer signifying the end of the list. In C, zero is used to signify the null pointer.

Nodes are described with the C structure mechanism. Here is one possible description of a node.

```
typedef struct node
    {
    int value;
    struct node *next;
    } Node;
```

The *next* variable is a pointer to a node structure, as expected[1]. The *value* variable, on the other hand, is given the type of values we wish to place in the list. In this case, we will use nodes to hold integer values.

Like all structures, we can allocate a node both statically and dynamically. As stated earlier, we will focus on dynamic allocation. To make our code easier to read, we will define a constructor whose task it is to dynamically allocate a node.

18.2 Node constructors, accessors, and mutators

In the previous chapter, we learned that using constructors, accessors, and mutators allow us to abstract away the internal details of structures. We can do the same for nodes. Node constructors could be defined as:

```
Node *
newEmptyNode(void)
    {
    Node *p = malloc(sizeof(Node));     //check for malloc failure omitted
    p->value = 0;                       //set to empty
    p->next = 0;                        //set to the null pointer
    return p;
    }
```

and:

```
Node *
newNode(int v,Node *n)
    {
    Node *p = malloc(sizeof(Node));     //check for malloc failure omitted
    p->value = v;
    p->next = n;
    return p;
    }
```

with accessors and mutators:

```
int getNodeValue(Node *n) { return n->value; }
Node *getNodeNext(Node *n) { return n->next; }
void setNodeValue(Node *n,int v) { n->value = v; return; }
void setNodeNext(Node *n,Node *p) { n->next = p; return; }
```

Given these constructors, accessors, and mutators, we can now assemble a chain of nodes.

[1]We can't use the alias Node yet, since the alias is set up once the entire node structure is processed. So inside the structure, we must use `struct node`.

18.3 Assembling nodes into a list

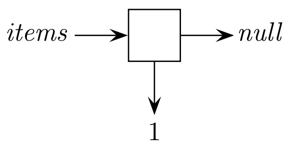
Let us now define a function that reads integer values from a file and stores them into a list:

```
Node *
readIntegers(FILE *fp)
    {
    int x;
    Node *items;

    items = 0; //items starts out as null

    //employ the standard reading pattern
    x = readInt(fp);
    while (!feof(fp))
        {
        items = newNode(x,items);
        x = readInt(fp);
        }
    return items;
    }
```

Assume the file *data.txt* exists with the following values:

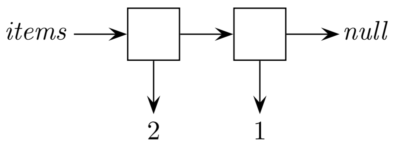
```
1 2 3
4 5 6
7 8
```

The first value read is a 1. With this value, a new node is constructed. Since *items* has an initial value of zero, this new node's *next* pointer is null. When this newly constructed node is assigned back to *items*, we have this situation:

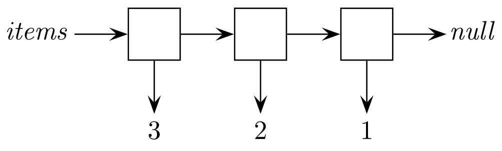


As stated earlier, the value field is not an integer pointer in this case, but the down arrow points to the value for visual convenience.

Next, the value 2 is read, the *while* test succeeds, a new node is constructed, and that node is assigned to *items*. This time, the newly constructed node's next pointer points to the old value of *items*, which was the node holding the 1. The situation now looks like:



When new nodes are added to the *items* list, they are placed on the front of the growing list. So after the next integer is read and node created, we have:



Once we have collected all the integers in the file into a list, we see that the values in the list are in the opposite order as found in the file! That is, the last value read is the first value in the list[2]. The strange order of the values in the list may bother you, but there are many situations where the order of the values in the list do not matter. For example, if we were to sum the values in the list, does the order the values appear matter? The answer is no.

Now that we have all our values in a list, how might we display them? Here is a function that moves through the list, using the *getNodeValue* and *getNodeNext* functions to access each of the values of the list in turn.

```
void
displayList(Node *items)
    {
    while (items != 0)
        {
        printf("{%d}",getNodeValue(items));
        items = getNodeNext(items);
        }
    printf("\n");
    }
```

The *while* loop runs as long as *items* isn't null. Inside the loop, the value of the first node in the list is printed. Then, *items* is reset to point to the next node in the list and the process repeats.

Now, to test our code:

```
//test
#include "scanner.h"
#include "nodes.h"
FILE *fp;
if ((fp = fopen("data.txt","r")) == 0)
    {
    fprintf(stderr,"data.txt missing\n");
    exit(-1);
    }
Node *items = readIntegers(fp);
displayList(items);
```

Assuming *data.txt* exists as described above, we get the following output:

```
{8}{7}{6}{5}{4}{3}{2}{1}
```

[2]This curious behavior is the basis for a somehting called a *stack*. You will learn about stacks in your next CS course.

Note the line:

```
    if ((fp = fopen("data.txt","r")) == 0)
```

This is a common idiom for assigning a value and then testing it. The variable *fp* gets assigned the return value of *fopen* within the parentheses; after it is assigned, it is checked to see if its newly assigned value is zero (signifying that the open failed).

Our next step is to define some functions that will make it easier to manipulate lists. We do that in the next chapter.

Chapter 19

Lists

You can download the functions defined or used in this chapter with the following commands:

```
wget troll.cs.ua.edu/ACP-C/ilist.c    //integer-valued nodes
wget troll.cs.ua.edu/ACP-C/ilist.h
wget troll.cs.ua.edu/ACP-C/rlist.c    //real-valued nodes
wget troll.cs.ua.edu/ACP-C/rlist.h
wget troll.cs.ua.edu/ACP-C/slist.c    //string-valued nodes
wget troll.cs.ua.edu/ACP-C/slist.h
wget troll.cs.ua.edu/ACP-C/olist.c    //'object'-valued nodes
wget troll.cs.ua.edu/ACP-C/olist.h
wget troll.cs.ua.edu/ACP-C/employeeList.c    //employee linked list demo
```

These files will help you run the test code listed in this chapter. You will also need to download the *list* module.

19.1 The List data structure

We could stop at this point and just use nodes and their operations to make true lists. But just as constructors, accessors, and mutators free us from being concerned how nodes and records are structured, we can design some list (sometimes called *linked list*) operators that free us from some of the details of the nodes themselves. If we do our design and implementation properly, a reader of our code will have few clues that lists are build from nodes.

The first task is to decide what will represent an empty list. The null pointer will work well for this role.

```
Node *
newEmptyList(void)
    {
    return 0;
    }
```

Next, we define *join*, a function that will be used to add a value to a list, returning the new list. We will use our integer-valued nodes from the previous chapter:

```
Node *
join(int value,Node *items)
```

```
        {
        return newNode(value,items);
        }
```

The *join* function takes a value and a list as arguments and returns a new node that glues the two arguments together. The result is a list one value larger than the list that is passed in. Note that join is non-destructive with regards to the original list; if we want to modify the original list, we must reassign the original list with the return value of *join*:

```
    items = join(value,items);
```

We can see from the definition of an empty list and from *join* that a list is either 0 (the null pointer) or a node. Two accessor functions are often defined for lists, *head* and *tail*. The *head* function returns the value stored in the first node in the list, while the *tail* returns the chain of nodes that follows the first node:

```
    int head(Node *items) { return getNodeValue(items); }
    Node *tail(Node *items) { return getNodeNext(items); }
```

Two mutator functions are also commonly defined:

```
    void setHead(Node *items,int v) { setNodeValue(items,v); }
    void setTail(Node *items,Node *n) { setNodeNext(items,n); }
```

You may be wondering at this point why we bother to distinguish lists and nodes. One reason is we can improve lists by keeping some more information around; we will save that improvement for later. Another reason is it is vitally important to practice the concept of abstraction. Modern software systems exhibit a large number of abstraction layers. With our lists, we can see for layers of abstractions: *variables*, which abstract locations in memory, *structures*, which abstract collections of variables, *nodes*, which abstract structures with two components, and *lists*, which abstracts a chain of `nodes`. Each level of abstraction frees us from thinking about the details of the underlying layers. We emphasize abstraction over and over, because it is so very important.

19.1.1 The *join* operator

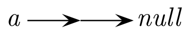
Let us now look at the *join* operation more closely. Here is a rewrite of *join* that uses a temporary variable, *n*, to hold the new node that will join the given value and list together:

```
    Node *
    join(int v,Node *items)
        {
        Node *n = newEmptyNode();     //step 1
        setNodeValue(n,v);            //step 2
        setNodeNext(n,items);         //step 3
        return n;
        }
```

First, let us create an empty list to which we will join a value:

```
a = newEmptyList();
```

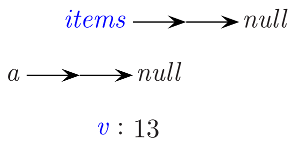
Graphically, the situation looks like this:



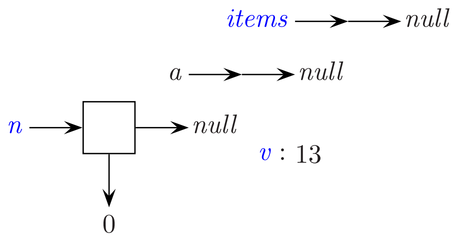
Now, we call *join* to join the value of 13 to the list *a*.

```
a = join(13,a);
```

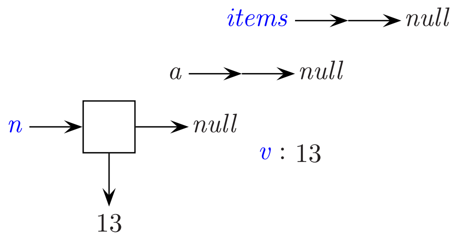
At the very start of the *join* function, the formal parameters *v* and *items* have been set to the value 13 and *a*, respectively. The situation looks like this:



The variables *v* and *items*, since they are local to the function *join*, are shown in blue[1]. After step 1, `n = newEmptyNode()`, we have:

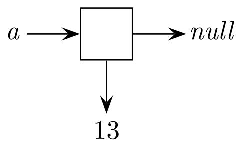


As with *items* and *v*, the variable *n* is shown in blue as it is a local variable. After step 2, `setNodeValue(v,None)`, the node *n* has its *value* set to *v*. the variable *n*. The situation changes to:



After step 3, `setNodeNext(n,items)`, the *next* pointer of *n*, which is null, is changed to the value of *items*, which is also null. So the situation remains the same as before. Finally, *n* is returned and the variable *a* is reassigned to have the value of *n*. The variables *v*, *items*, and *n* go out of scope and disappear, leaving:

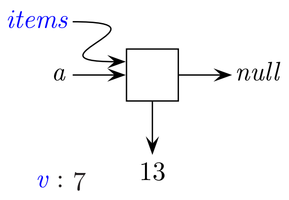
[1]When two variables, in this case *a* and *items*, point to the same thing, they are said to be *aliases* of one another.



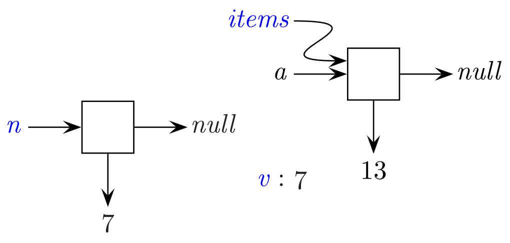
If we add a second value to the list *a*, as in:

```
    a = join(7,a);
```

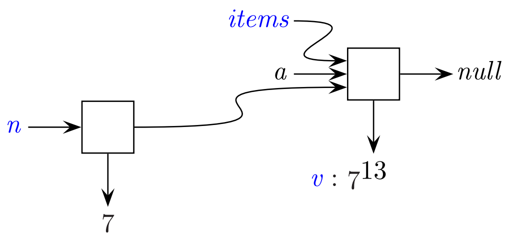
at the start of the *join* function, we have:



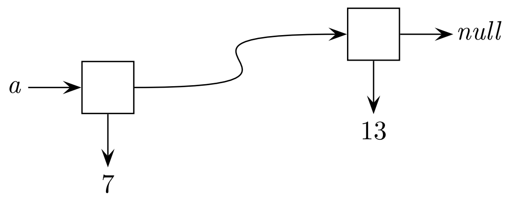
After creating node *n* (step 1) and setting its *value* pointer (step 2), we have:



Setting *n*'s next pointer to *items* yields:



At this point, we return the value of *n* and assign it to *a*:



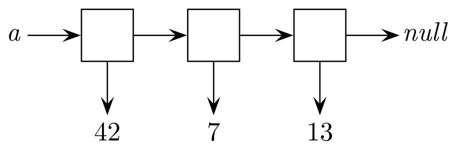
From this last drawing, we can see that list *a* now includes the value 7 at the front of the list, as intended. As before, at this point the variables local to *join*, are no longer in scope.

19.1.2 Chaining calls to *join*

As seen in the previous section, the *join* function is used to build a list:

```
a = 0;
a = join(13,a);
a = join(7,a);
a = join(42,a);
```

The above code builds the following list:



An alternative way to build the exact same list is to chain together the series of calls to *join*:

```
a = join(42,join(7,join(13,0)));
```

Both methods are rather tedious for larger arrays, so we will make use of a list constructor that takes an array of integers and returns a list of those integers. This code:

```
int numbers[] = {42,7,13};
Node *a = arrayToList(numbers,sizeof(numbers)/sizeof(int));
```

produces the same list. A first attempt at defining *arrayToList* might look like this:

```
Node *
arrayToList(int *array,int size)
    {
    int i;
    Node *items = 0; //start items as the empty list
    for (i = 0; i < size; ++i)
        items = join(array[i],items);
    return items;
    }
```

Unfortunately, this produces a list whose values are found in the reverse order from that of the array. Why this is so is answered by the question, "What is the last value joined to the growing list and where did it end up in the list?". Obviously, the last value joined was the last value in the array and it ended up as the first value in the list, since *join* places its first argument at the head of the list. By a similar logic, the next-to-the-last value in the array ended up in the second position in the list, and so on.

We can fix this deficiency by defining a function that creates a new list in the reverse order of a given list. The code mimics our *arrayToList* function:

```
Node *
reverse(Node *a)
    {
    Node *items = 0;
    while (a != 0)
        {
        items = join(head(a),items);
        a = tail(a);
        }
    return items;
    }
```

Again, the last value in the incoming list *a* is placed in the first position of the new list *items*.

The *arrayToList* function becomes:

```
Node *
arrayToList(int *array,int size)
    {
    int i;
    Node *items = 0; //start items as the empty list
    for (i = 0; i < size; ++i)
        {
        items = join(array[i],items);
        }
    Node *rev = reverse(items);
    freeList(items);
    return rev;
    }
```

This revised version of *arrayToList* is a bit inefficient in that the values to be placed in the list are traversed twice, once by accessing every value in the array (the *for* loop) and once by accessing every value in *reverse*'s incoming list. The *arrayToList* functions found in the list modules mentioned in the start of this chapter cleverly make a single traversal. See if you can figure out how they work.

19.1.3 Reading a list from a file

Consider a file of integers that we wish to read into a list. As always, we use the *read* pattern:

```
do the initial read
while (the read was good)
    {
    process the read
    read again
    }
```

Concretely, we have:

```
Node *
readIntegers(FILE *fp)
```

```
        {
        int v;
        Node *items;

        items = 0;              //items points to an empty list
        fscanf(fp,"%d",&v);
        while (!feof(fp))
            {
            items = join(v,items);  //update items
            fscanf(fp,"%d",&v);
            }
        return items;
        }
```

Like the *arrayToList* function in the previous subsection, the *readIntegers* function also reverses the order of the integers as found in the file. If order must be maintained, one can reverse *items* before returning or use the same trick the version of *arrayToList* in the ilist module uses.

19.1.4 Displaying a list

To display a list, we need to *walk* the list. A walk of a list is comprised of visiting every node in the list from front to back. Typically, a loop is used to implement a walk. Here is a display function for a list of integer-valued nodes:

```
    void
    displayList(Node *items)
        {
        while (items != 0)
            {
            int v = head(items)
            printf("{%d}",v);
            items = tail(items); //take a step
            }
        printf("\n");
        }
```

With this function in place, we can see if *join* is really working. Here is some code that tests both *join* and *display*:

```
    //test (compile with ilist.c)
    #include "ilist.h"
    int numbers[] = {42,7,13};
    Node *a = arrayToList(numbers,sizeof(numbers)/sizeof(int));
    displayList(a);
```

Running this code yields the following output, as expected:

```
    {42}{7}{13}
```

19.2 Lists of other values

We can make lists of any kind of value. The modules *rlist.c* and *slist.c* contain analogs to the functions in *ilist.c*, except they work with real numbers (`double`) and strings (`char *`), respectively.

Another module, the *olist.c* module, allows for lists of pointers to arbitrary structures (or any pointers, other than function pointers). Consider the employee record structure from the Chapter **??**. One might read a file of employee records, storing them into a linked list, this way:

```
#include "employee.h"
#include "olist.h"
...
Node *
readEmployeeList(FILE *fp)
    {
    Node *employees = 0; //zero is the empty list
    Employee *e = readEmployeeRecord(fp);
    while (!feof(fp))
        {
        employees = join(e,Employees);
        e = readEmployeeRecord(fp);
        }
    return employees;
    }
```

A careful inspection of the *olist* module reveals that the value in a node has type `void *`. A variable of this type can store any pointer (except a function pointer) safely. However, once stored, the pointer loses any knowledge of the kind of thing to which it points (i.e. the kind of thing found at the address stored in the pointer). Another way to describe this loss of information is to say the pointer becomes *generic*. Once the pointer is genericized, it is up to the programmer to convert it back to the kind of pointer it once was. For example, this code fails:

```
Node *emps = readEmployeeList(fp);
char *name = head(emps)->name;     //head(emps) is a generic pointer
```

because the code writer has not converted the generic pointer returned by *olist*'s *head* function into an *Employee* pointer. This code, on the other hand, succeeds:

```
Node *emps = readEmployeeList(fp);
Employee *e = head(emps);  //generic pointer converted to Employee pointer
char *name = e->name;
```

The latter two lines can be combined into a single line with a cast:

```
Node *emps = readEmployeeList(fp);
char *name = ((Employee *) head(emps))->name;
```

One can also use the accessor function for the *name* field in the *Employee* structure:

```
    Node *emps = readEmployeeList(fp);
    char *name = getEmployeeName(head(emps));
```

This gives the compiler the information that the generic pointer returned by `head(emps)` is really a pointer to an *Employee* structures, since *getEmployeeName* accepts *Employee* pointers.

The file *employeeList.c* demonstrates using the *Employee* structure with the *olist* module.

Chapter 20

Lists and Loops

You can download the functions defined or used in this chapter with the following commands:

```
    wget troll.cs.ua.edu/ACP-C/listLoop.c     #functions from this chapter
    wget troll.cs.ua.edu/ACP-C/listLoop.h
```

These files will help you run the test code listed in this chapter. You will also need to download the *ilist* module.

20.1 List/Loop patterns

For every array pattern, there is a corresponding list pattern. Sometimes, the pattern is easier to implement with an array than a loop. Sometimes, the opposite is true. The next sections will detail patterns for lists.

The *append* pattern

Adding values to the front of a list is known as *prepending*, so *join*[1] prepends a value to a list. Instead of adding new values to the front, we can also add values to the back of the list. Such a procedure is termed *appending*; Since lists at this point are nodes, we can append a list with either a single node or a list. In this section, we will focus an appending a list with another list:

```
    void
    append(Node *listA,Node *listB) //listA length must be > 0
        {
        while (tail(listA) != 0)
            {
            listA = tail(listA); //move to the next node in the list
            }
        setTail(listA,listB);
        return;
        }
```

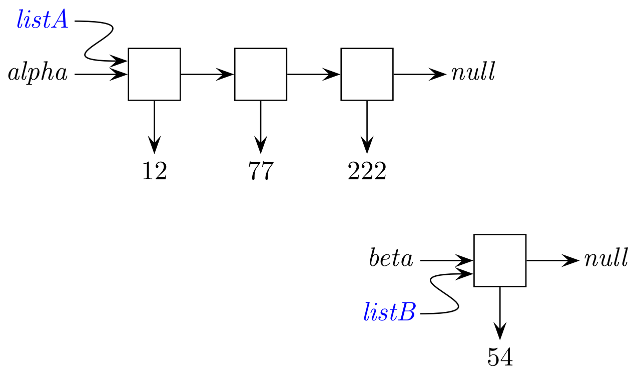
Here, we start out with the formal parameter *listA* pointing to the first node in the list passed as *listA*. Then, as long as *listA*'s *next* pointer points to a node and not null[2], we set *listA* to the next node in the list

[1] *Prepend* is rather an awkward term, so that is why we will use *join* as the function name, rather than *prepend*.
[2] The test condition of the while loop is the reason for the comment that the length of *listA* must be greater than zero.

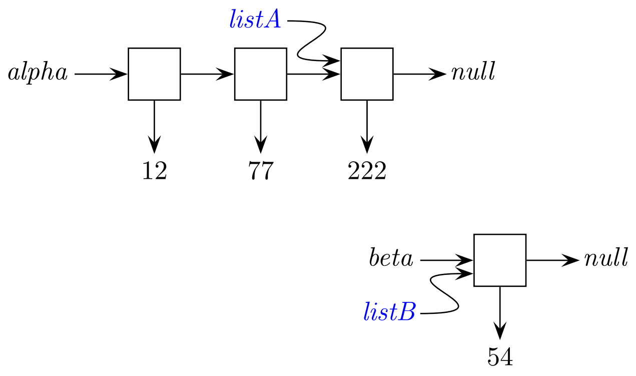
by taking the tail. We repeat this process until we reach the last node in the list, whose *next* pointer does indeed point to null. At this point, we drop out of the loop and set the next pointer of this last node to *listB* via *setTail*. Suppose we call *append* with two lists, *alpha* and *beta*:

```
append(alpha,beta);
```

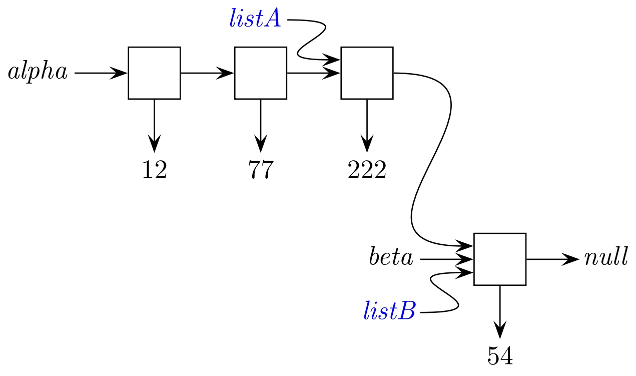
Here is the view just before *append* walks the first list:



Immediately after dropping out of the loop, the local variable *listA* points to the last node in the list:



After the reassingment of the tail pointer of the adjusted *listA*, we can see that *alpha* now incorporates the nodes of *beta*:



Note that while *head* and *tail* are non-destructive operations, *append* is a destructive operation, since one pointer in an original list is actually changed. Generally, you can quickly tell if a function is destructive or non-destructive by looking at the return value. If the function implements the *procedure* pattern, it is likely destructive. If it implements the *function* pattern, it is likely non-destructive[3].

To test *append*, we need to create some lists:

```
//test (compile with ilist.c and listLoop.c)
#include "listLoop.h"
Node *a,*b;
a = join(2,join(4,join(6,0)));
b = join(1,join(3,join(5,0)));
displayList(a);
displayList(b);
append(a,b);
displayList(a);
```

The output for this program is:

```
{2}{4}{6}
{1}{3}{5}
{2}{4}{6}{1}{3}{5}
```

The last line in the output comes from the last displayIntegerList. This clearly shows that *append* is doing its job.

If we wish to append a single value to a list, we can turn the single value into a list first:

```
//test (compile with ilist.c and listLoop.c)
#include "listLoop.h"
Node *a,*b;
a = join(2,join(4,join(6,0)));
b = newNode(1,0);
append(a,join(b,0)); //incorporate b into a list
displayList(a);
```

The process of moving from one node to the next in a list is known as *walking* the list. Obviously, the longer the list, the longer it takes to walk it. Thus, appending to a list can take much longer than prepending to the same list, since prepending takes the same amount of time regardless of how long the list is. However, as you will learn in a subsequent data structures class, there is a much faster way to append to a list. If you cannot wait to find out this clever implementation of *append*, search for the phrase ”linked list tail pointer” on the interwebs.

We can use the idea of walking a list to implement some more useful list functions. The *getListIndex* function returns the value at the i^{th} node in a given list:

```
int
getListIndex(Node *items,int index)
    {
```

[3]Sometimes, destructive functions return a value for convenience to the caller, in which case this generalization fails.

```
        while (index > 0)
            {
            items = tail(items);
            --index;
            }
        return head(items);
        }
```

Note that taking the tail of a list does not alter the original list in any way; the list that is passed in is unchanged by this function[4]. Finally, consider if there are fewer nodes in the list than is specified by the index. This implementation, like many C built-in functions, does no error checking. What happens if the list is too short for the given index is undefined (but it will likely be very bad). The analogous *setListIndex* is left as an exercise.

20.2 The *search* pattern

This idea of walking lists is an extremely important concept; so important that we will review the two "walks" that are commonly needed when manipulating lists. The first is walking to a certain location in list. Usually, one walks to index (covered previously) or to a value. Of course, walking to a value is simply the search pattern, implemented for lists. Here is a search function that works on *Integer* lists:

```
    Node *
    findInteger(int value,Node *items)
        {
        while (items != 0 && head(items) != value)
            {
            items = tail(items);
            }
        return items;
        }
```

Note that this version returns the node containing the value if the value is found and the null pointer otherwise. Because of the possibility that a value may not be in the list, we must add the condition `items != 0` to the while loop test. If we didn't, then *items* would eventually reach 0 and the *head* operation on the null pointer would likely generate a crash. Sometimes, *find* would be written with a simpler loop test, but complicated by a return from the body of the loop:

```
    Node *
    findInteger2(int value,Node *items)
        {
        while (items != 0)
            {
            if (head(items) == value)
                 {
                 return items;
                 }
            items = tail(items);
            }
        return 0;
        }
```

[4]Again, when an operation does not affect its operands, it is said to be non-destructive. The *getListIndex* function, since it never sets a pointer in the given list, is non-destructive

These two implementations are semantically equivalent; the former is usually preferred stylistically, but the latter is likely more common.

20.3 The *trailing walk* pattern

The second walk one is likely to encounter is a walk to the next-to-the-last node in a list, which is useful in many situations: removing the last item, adding an item just prior to the last item, and so on.

One must walk the list to get to the penultimate node. Here is one attempt; it keeps two pointers when walking the list, a leading pointer and a trailing pointer that stays one node behind the leader. The walk ends when the *next* pointer of the leading node becomes null. We call this pattern the *trailing walk pattern*:

```
Node *
getPenultimateNode(Node *items)
    {
    Node *trailing = 0;
    Node *leading = items;
    while (tail(leading) != 0) //when to stop walking
        {
        trailing = leading;
        leading = tail(leading);
        }
    return trailing;
    }
```

If we walked until the lead node became null, then the trailing node would be the last node in the list, not the next-to-the-last node. However, checking the next pointer of a node is a dangerous proposition. What happens in the case of an empty list? How many nodes must be in a list for this function to work properly?

The above approach can be simplified by checking if the trailing pointer's next pointer points to a node whose next pointer is null. Although the check is a bit more complicated, it obviates the need for the leading node:

```
Node *
getPenultimateNode2(Node *items)
    {
    Node *trailing = items;
    while (tail(tail(trailing)) != 0)
        {
        trailing = tail(trailing);
        }
    return trailing;
    }
```

The two versions of *getPenultimateNode* are similar, but not exactly equivalent. How many nodes must be in a list for the second version to work properly?

To test these implementations, we will create a list and then print the value of the penultimate node:

```
//test (compile with ilist.c and listLoop.c)
#include "listLoop.h"
```

```
Node *a,*n,*m;
a = join(newInteger(2),join(newInteger(4),join(newInteger(6),0)));
n = getPenultimateNode(a);
printf("n is %d\n",getIntegerValue(head(n)));
m = getPenultimateNode2(a);
printf("m is %d\n",getIntegerValue(head(m)));
```

Both version produce 4, as expected.

An application of the trailing walk pattern is to insert a value into an ordered list so that the list remains ordered[5]. For example, suppose we have an ordered list consisting of the numbers 1, 3, 8, 14, and 17:

If we wish to insert 10 into the list, we must place the 10 between the 8 and the 13 so that the list values remain in increasing order (from left to right). To simplify the making of an integer Using *getPenultimateNode* as a starting point, we have:

```
void
insertIntegerInOrder(int value,Node *items)
    {
    Node *trailing = 0;
    Node *leading = items;
    while (...) //when to keep walking
        {
        trailing = leading;
        leading = tail(leading);
        }
    //insert new value in between trailing and leading
    ...
    }
```

The ellipses mark where changes to the trailing value pattern need to be made. The first ellipsis (the while test) should force the walk to stop when the correct insertion point is found. Clearly, that is when leading value is greater than the value to be inserted. The second ellipsis concerns how the actual insertion is to be performed. We know we will need to:

- create a new node
- set the new node's value pointer to the value to be inserted
- set the new node's next pointer to the leading node
- set the trailing node's next pointer to the new node

All that's left then is to fill in the blanks. Here is the result:

```
void
insertInOrder(int value,Node *items)
    {
    Node *trailing = 0;
```

[5]This idea of repeatedly inserting items in a list with the list remaining ordered is the basis for an important data structure, the *priority queue*. Priority queues are used in network connectivity algorithms and discrete-event simulations. You will learn about priority queues in a subsequent class

```
        Node *leading = items;
        while (head(leading) < value) //when to keep walking
            {
            trailing = leading;
            leading = tail(leading);
            }
        //insert new value in between trailing and leading
        Node *n = newNode(value,0);
        setTail(n,leading);     //place n before leading
        setTail(trailing,n);    //place n after trailing
        }
```

Note that we want to *stop* when the leading value is *greater* than the value to be inserted. Since we are working with a while loop, we need to reverse the logic so that the walk *continues* as long as the leading value is *less than* the new value[6].

Let's test our code:

```
    //test (compile with ilist.c and listLoop.c)
    #include "listLoop.h"
    int numbers[] = {1,3,8,13,14,17};
    Node *a = arrayToList(numbers,sizeof(numbers)/sizeof(int));
    insertInOrder(10,a);
    displayList(a);
```

Compiling and running this code yields:

```
    {1}{3}{8}{10}{13}{14}{17}
```

It appears our code works correctly! The 10 is nestled nicely between the 8 and the 13. Very exciting! Unfortunately, the excitement of getting code to work seduces both novice and experienced Computer Scientists alike into thinking the task is finished. Many times, including this case in particular, initial success only means the basic idea is correct, not that the code is correct for all possible inputs. Indeed, for our implementation of *insertOrder*, there are *edge cases* for which this code fails. An edge case is a set of inputs that force the code to do as much (or as little) work as possible. For *insertInOrder*, what inputs causes the function to do as much work as possible? The only place where a variable amount of work is performed is the loop that walks the list. In order to make the loop go as many times as possible, it is clear that we would need to insert a value at the very end of the list. Changing the line:

```
    insertInOrder(20,a);    //was: insertInOrder(10,a);
```

yields the following result when the program is run:

```
    Segmentation fault      (core dumped) ./a.out
```

Likewise, making the loop do as little work as possible yields errors in the code. What inputs would cause the loop run very little or not at all? Obviously, if the value to be inserted is smaller than any value in the list, the loop does not need to be run. Less obviously, what if the given list is empty? Fixing *insertInOrder* to handle these edge cases is left as an exercise.

[6]Actually, reversing the logic of greater than yields less than *or equal*, but we are ignoring situations when values match exactly. What happens in those cases is left as an exercise.

20.4 The *counting* pattern

The counting pattern is very important for lists, since we can use that pattern to deduce the number of items in a list:

```
int
length(Node *items)
    {
    int count = 0;
    while (items != 0)                //typical list walking condition
        {
        ++count;
        items = tail(items);          //typical list walking step
        }
    }
```

Traditionally, the function that counts the number of items in a list is called *length*. As with the counting pattern for arrays, each "step" taken results in a counter being incremented.

20.5 The *filtered-accumulation* pattern

A filtered accumulation looks similar to the counting pattern. Here is a function that totals up all the even values in a list:

```
int
sumEvenIntegers(Node *items)
    {
    int total = 0;
    while (items != 0)
        {
        int v = head(items);
        if (v % 2 == 0)
            total += v;
        items = tail(items);
        }
    return total;
    }
```

20.6 The *extreme* pattern

Once you have the hang of walking a list, most patterns become trivial to implement. Here is an instance of the extreme pattern:

```
int
getLargest(Node *items)
    {
    int largest = head(items); //assume first is largest
    while (items != 0)
        {
        int v = head(items);
```

```
            if (v > largest)
                largest = v;
            items = tail(items);
            }
        return largest;
        }
```

This implementation looks very much like the extreme pattern for arrays, only translated to list walking.

Surprisingly or, maybe, not surprisingly, the *extreme index pattern* is much more difficult to implement for lists than for arrays. This is because the basic walk of a list ignores the index of a particular node. In general, any processing that requires the manipulation of indices will be more complicated with a list as compared to an array. The implementation of the extreme index pattern is left as an exercise.

20.7 The *copy*, *filter* and *map* patterns

Lists lend themselves to non-destructive copying, filtering, and mapping. As the original list is walked and values processed, a new list holding the original, filtered, or mapped values is built up. One slight problem arises when loops are used to walk the list. Since lists are grown using *join*, the elements in the new list are found in the opposite order as compared to their associated elements in the original list. If the order of the result doesn't matter, then lists and loops go well together for this kind of processing. If the order does matter, then another way of walking list, using recursion, can be used to preserve order. We will cover recursion in the next chapter.

Here is code which copies a list.

```
    Node *
    copyList(Node *items)
        {
        Node *result = 0;   //resulting list starts out empty
        Node *spot = items;
        while (spot != 0)           //walk the list
            {
            result = join(head(spot),result); //add current value to result
            spot = tail(spot);                //take a step
            }
        return result;
        }
```

In almost all list processing functions, we see the basic walk of the list. Indeed, should you be tasked to define a list processing function, you would be well advised to start with a basic walk and then modify as needed.

Testing the *copyList* function reveals the reordering of the original values:

```
    //test (compile with listLoop.c)
    #include "listLoop.h"
    int numbers[] = {1,3,8,13,14,17};
    Node *a = arrayToList(numbers,sizeof(numbers)/sizeof(int));
    Node *b = copyList(a);
    displayList(a);
    displayList(b);
```

Compiling and running this code yields:

```
{1}{3}{8}{13}{14}{17}
{17}{14}{13}{8}{3}{1}
```

Filtering and mapping are based off of the copy pattern. Here is a function the filters out odd-valued integers:

```
Node *
extractEvens(Node *items)
    {
    Node *result = 0;
    while (items != 0)
        {
        int v = head(items);
        if (isEven(v))
            result = join(head(items),result);
        items = tail(items);
        }
    return result;
    }
```

Note how the resulting list is grown every time an even valued item is found in the original list. Testing the function:

```
//test (compile with ilist.c and listLoop.c)
#include "listLoop.h"
int numbers[] = {1,3,8,13,14,17};
Node *a = arrayToList(numbers,sizeof(numbers)/sizeof(int));
Node *b = extractEvenIntegers(a);
displayIntegerList(b);
displayIntegerList(b);
```

yields:

```
{1}{3}{8}{13}{14}{17}
{14}{8}
```

The even numbers from the original list are found in the resulting list in the opposite order.

20.8 The *shuffle* and *merge* patterns

As with arrays, shuffling and merging lists using loops is rather complicated. So much so that we will not even bother discussing loop-with-list versions. However, using recursion with lists greatly simplifies this type of processing, as we shall see in the next chapter.

20.9 The *fossilized* pattern

The primary reason for accidentally implementing the fossilized pattern is to forget the while loop update. For example, while this loop:

```
while (items != 0)
    {
    //loop body here
    ...
    items = tail(items);      //update, often accidentally omitted
    }
```

may have been intended, what sometimes happens is the update to *items* is omitted. Since items is never updated, it never advances to the null pointer (unless, of course, it was an empty list to begin with). This leads to an infinite loop.

20.10 Freeing lists

Since our lists are dynamically allocated, we need to free them when we are done with them:

```
void
freeList(Node *items)
    {
    Node *spot;
    while(items != 0)
        {
        spot = items;
        items = tail(items);
        free(spot);                //free the current node
        }
    }
```

The only complication to this procedure is the need to save the lead node of the list before advancing to the next node. This is because, once we free something, we cannot reference any part of it. Thus, the sequence:

```
free(items);            //free the current node
items = tail(items);    //WRONG!
```

will end in grief. The second statements asks for a part of the node (the value of the next pointer), *after* the node has been freed. Note that this routine does not free the values of a node. For integer nodes, this is not an issue. For string-valued nodes, where the strings were *malloc*-ed, the *freeList* function would be modified to free those strings:

```
void
freeList(Node *items)
    {
    Node *spot;
    while(items != 0)
        {
        spot = items;
        items = tail(items);
        free(spot->value);
        free(spot);                //free the current node
        }
    }
```

20.11 Why Lists?

You may be asking, why go through all this trouble to implement lists when arrays seem to do every thing lists can do. The reason comes down to the fact that no data structure does all things well. For example, a data structure may be very fast when putting values in, but very slow in taking values out. Another kind of data structure may be very slow putting values in but very fast taking values out. This is why there are quite a number of different data structures, each with their own strengths.

Arrays versus Lists:

- Accessing any element in an array is very fast, regardless of the size of the array. In contrast, the larger the list, the more time it takes to access values near the end of the list.
- Dynamically allocated arrays can be allocated and freed in a single step. Freeing a list means freeing every node in the list in turn.
- Adding a value to the front of a list is very fast; adding a value to the end of a list can be made very fast. In contrast, adding an element to a (full) array means a new larger array has to be allocated and elements of the old array copied over to the new array.
- Lists and recursion go very well together; arrays, not so much. We will use lists heavily in the next chapter on recursion.

20.12 Problems

1. Assume a list holds the values 3, 8 and 10. Draw a box and pointer diagram of this list.
2. Why does the for loop in the *intsToList* function (in *integers.c*):

   ```
   for (i = size-1; i >= 0; --i)
   ```

   decrement the loop index?
3. Assume a list holds the values 3 and 8. Consider appending the value 5. Draw a set of box and pointer diagrams, similar to those used in the discussion of *join*, that illustrate the action of *append*. You should have a drawing of the situation before the while loop begins, one for each update of the variable node while the loop is running, one after the new node is created, and one showing the list just before the function returns. Label your drawings appropriately.
4. Define a function named *appendValue*, which takes a list and an `int` value and appends the value to a list of *Integer* values. Your function should look very similar to *append*.
5. The module *lists.c* exhibits an important principle in Computer Science, *abstraction*. The list operations treat nodes as abstract objects; that is to say, lists do not know nor care about how nodes are constructed. Verify this is true by renaming the variables in the Node structure and updating the accessors and mutators of nodes accordingly. Now test the list operations to see that they work exactly as before.
6. The list operation *getListIndex* as found in the *lists.c* module exhibits another important concept in Computer Science, *generalization*. Since both *getListIndex* and *setListIndex* need to walk the list, that list-waking code was generalized into the function *getListIndexedNode*. Modify the *lists.c* module so that *getListIndex* and *setListIndex* each walk the list. Both operations should function as before, but you should notice how similar they look. When you see such similarity, the similar code is a candidate for generalization.

7. Define a function named *reverse*, which takes a list as an argument and returns a new list with the same items as the given list, but in the opposite order.

8. Define a function named *copy2*, which takes a list as an argument and returns a copy of the list. The new list should have the same ordering as the original list.

9. Define a function named *chop*, which takes a list as an argument and destructively removes the last element of a list. Your function should look similar to *getPenultimateNode*.

10. Define a function named *equals*, which takes two lists as arguments and returns whether or not the two lists have the same elements in the same order.

11. The linked lists developed in this chapter are known as *singly*-linked lists, a class of lists with a single pointer links a node to the next node in the list. Another class of lists, *doubly*-linked lists, provide an additional pointer from a node to its predecessor. Define a constructor for nodes upon which doubly-linked lists could be based.

12. Fix the given version of *insertIntegerInOrder* so that it correctly handles edge cases.

13. Implement a function that finds the index of the largest element in the list. Do not use *getListIndex* or anything like it.

14. Consider the version of *insertIntegerInOrder* given in this chapter. If a value to be inserted matches a value already in the list, does the new value end up before or after the old value? How would you change the code to get the opposite behavior?

Chapter 21

Recursion

You can download the functions defined or used in this chapter with the following commands:

```
wget troll.cs.ua.edu/ACP-C/recursion.c
wget troll.cs.ua.edu/ACP-C/recursion.h
```

These files will help you run the test code listed in this chapter.

21.1 Recursion, just another way to loop

In Chapter **??**, we learned about *if* statements. When we combine *if* statements with functions that call themselves, we obtain a powerful programming paradigm called *recursion*.

Recursion is a form of looping; when we loop, we evaluate code over and over again. We use a conditional to decide whether to continue the loop or to stop. Recursive loops are often easier to write and understand, as compared to the iterative loops such as *while*s and *for*s, which you learned about in a previous chapter. In some programming languages, iterative loops are preferred as they use much less computer memory and are slightly faster as compared to recursive loops. In other languages, this is not the case at all. In general, there is no reason to prefer iterative loops over recursive ones, other than this memory issue (for some languages) and slight loss of performance. Any iterative loop can be written as a recursion and any recursion can be written as an iterative loop. Use a recursion if that makes the implementation more clear, otherwise, use an iterative loop.

Many mathematical functions are easy to implement recursively, so we will start there. Recall that the factorial of a number n is:

$$n! = n * (n-1) * (n-2) * ... * 2 * 1 \tag{21.1}$$

Consider writing a function which computes the factorial of a positive integer. For example, if the function were passed the value of 4, it should return the value of 24 since 4! is 4*3*2*1 or 24. To apply recursion to solve this problem or any problem, for that matter, it must be possible to state the solution of a problem so that it references a simpler version of the problem. For factorial, the factorial of a number can be stated in terms of a simpler factorial. Consider the two equations below:

$$0! = 1 \tag{21.2}$$

$$n! = n * (n-1)! \quad \text{otherwise} \tag{21.3}$$

Equation **??** says that the factorial of zero is one[1]. Equation **??** states that the factorial of any other (positive) number is obtained by multiplying the number by the factorial of one less than that number. Together, these two equations form a *recurrence equation* that describes the computation of factorial.

After some study, you should be able to see that this new way of describing factorial is equivalent to Equation **??**, the one that that used ellipses[2]. Equation **??** gets the basic idea of factorial across but is not very precise. For example, how would you compute the factorial of three using Equation **??**?

The second form with the two equations is particularly well suited for implementation as a function in a computer program:

```
int
factorial(int n)
    {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
    }
```

Note how the *factorial* function precisely implements the recurrence equation. Convince yourself that the function really works by tracing the function call:

```
factorial(3)
```

Decomposing the call, we find that:

```
factorial(3) is 3 * factorial(2)
```

since *n*, having a value of 3, is not equal to 0. Thus the second block of the *if* is evaluated and we can replace *n* with 3 and *n-1* with 2. Likewise, we can replace `factorial(2)` by `2 * factorial(1)`, yielding:

```
factorial(3) is 3 * 2 * factorial(1)
```

since *n*, now having a value of 2, is still not zero. Continuing along this vein, we can replace `factorial(1)` by `1 * factorial(0)`, yielding:

```
factorial(3) is 3 * 2 * 1 * factorial(0)
```

Now in this last call to factorial, *n* does have a value of zero, so we can replace `factorial(0)` with its immediate return value of one:

```
factorial(3) is 3 * 2 * 1 * 1
```

[1]Mathematicians, being an inclusive bunch, like to invite zero to the factorial party.
[2]Ellipses are the three dots in a row and are stand-ins for stuff that has been omitted.

Thus, `factorial(3)` has a value of six. The program:

```
//test (compile with recursion.c)
#include "recursion.h"
printf("factorial(3) is %d\n",factorial(3));
```

yields:

```
factorial(3) is 6
```

as expected.

In contrast, here is a function which computes factorial using a *for* loop:

```
int
factorial2(int n)
    {
    int i;
    int total = 1;
    for (i = 1; i < n + 1; ++i)
       total = total * i;
    return total;
    }
```

We can see from this version that factorial is an accumulation. This version also more closely follows Equation **??**. Note that we have to extend the upper end of the range by one in order to get n included in the accumulation. By habit, we like the upper limit of *for* loops always to be exclusive.

21.2 The parts of a recursive function

Recursive approaches rely on the fact that it is usually simpler to solve a smaller problem than a larger one. In the factorial problem, trying to find the factorial of $n-1$ is a little bit simpler[3] than finding the factorial of n. If finding the factorial of $n-1$ is still too hard to solve easily, then find the factorial of $n-2$ and so on until we find a case where the solution is dead easy. With regards to factorial, this is when n is equal to zero. The *easy-to-solve* code (and the values that get you there) is known as the *base* case. The *find-the-solution-using-a-simpler-problem* code (and the values that get you there) is known as the *recursive* case. The recursive case usually contains a call to the very function being executed. This call is known as a *recursive* call.

Most well-formed recursive functions are composed of at least one *base* case and at least one *recursive* case.

21.3 The greatest common divisor

Consider finding the greatest common divisor, the *gcd*, of two numbers. For example, the *gcd* of 30 and 70 is 10, since 10 is the largest number that divides both 30 and 70 evenly. The ancient Greek philosopher Euclid devised a solution for this problem that involves repeated division. The first division divides the two numbers in question, saving the remainder. Now the divisor becomes the dividend and the remainder becomes the

[3]If one views more multiplications as more complex, then, clearly, computing the factorial of $n-1$ is simpler than computing the factorial of n.

divisor. This process is repeated until the remainder becomes zero. At that point, the current divisor is the *gcd*. We can specify this as a recurrence equation, with this last bit about the remainder becoming zero as our base case:

gcd(a,b)	is	b	if a divided by b has a remainder of zero
gcd(a,b)	is	*gcd*(b,a % b)	otherwise

In this recurrence equation, a and b are the dividend and the divisor, respectively. Recall that the modulus operator `%` returns the remainder. Using the recurrence equation as a guide, we can easily implement a function for computing the *gcd* of two numbers.

```
int
gcd(int dividend,int divisor)
    {
    if (dividend % divisor == 0)
        return divisor;
    else
        return gcd(divisor,dividend % divisor);
    }
```

Note that in our implementation of *gcd*, we used more descriptive variables than a and b. We can improve this function further, by noting that the remainder is computed twice, once for the base case and once again to make the recursive call. Rather than compute it twice, we compute it straight off and save it in an aptly named variable:

```
int
gcd2(int dividend,int divisor)
    {
    int remainder = dividend % divisor;
    if (remainder == 0)
        return divisor;
    else
        return gcd2(divisor,remainder);
    }
```

Look at how the recursive version turns the *divisor* into the *dividend* by passing *divisor* as the first argument in the recursive call. By the same token, *remainder* becomes *divisor* by nature of being the second argument in the recursive call. To convince one's self that the routine really works, one can modify *gcd* to "visualize" the arguments. On simple way of visualizing the action of a function is to add a print statement:

```
int
gcd2(int dividend,int divisor)
    {
    int remainder = dividend % divisor;
    printf("gcd: %2d, %2d, %2d\n",dividend,divisor,remainder);
    if (remainder == 0)
        return divisor;
    else
        return gcd2(divisor,remainder);
    }
```

With the instrumented definition and this code:

```
//test (compile with recursion.c)
#include "recursion.h"
printf("%d\n",gcd2(66,42));
```

we get the following output:

```
gcd: 66, 42, 24
gcd: 42, 24, 18
gcd: 24, 18,  6
gcd: 18,  6,  0
6
```

Note, how the first remainder, 24, keeps shifting to the left. In the first recursive call, the remainder becomes *divisor*, so the 24 shifts one spot to the left. On the second recursive call, the current *divisor*, which is 24, becomes the *dividend*, so the 24 shifts once again to the left.

We can also write a iterative (and instrumented) version of *gcd*:

```
int
gcd3(int dividend,int divisor)
    {
    while (divisor != 0)
        {
        int temp = dividend % divisor;
        printf("gcd: %2d, %2d\n",divisor,dividend);
        dividend = divisor;
        divisor = temp;
        }
    return dividend;
    }
```

While the iterative version of factorial was only slightly more complicated than the recursive version, with *gcd*, we start to see more of an advantage using the recursive formulation. For instance, where did the *temp* variable come from and why is it necessary[4]?

21.4 The Fibonacci sequence

A third example of recursion is the computation of the n^{th} Fibonacci number. The Fibonacci series looks like this:

```
n            :  0   1   2   3   4   5   6   7   8    ...
Fibonacci(n) :  0   1   1   2   3   5   8  13  21    ...
```

and is found in nature again and again[5]. From this table, we can see that the 7^{th} Fibonacci number is 13. In general, a Fibonacci number is equal to the sum of the previous two Fibonacci numbers. The exceptions are

[4]We'll see why the variable *temp* is needed in the next chapter.

[5]A pineapple, the golden ratio, a chambered nautilus, etc.

the zeroth and the first Fibonacci numbers which are equal to 0 and 1 respectively. Voila! The recurrence case and the two base cases have jumped right out at us! Here, then, is a recurrence equation which describes the computation of the n^{th} Fibonacci number.

fib(n)	is	0	if n is zero
fib(n)	is	1	if n is one
fib(n)	is	*fib*(n - 1) + *fib*(n - 2)	otherwise

Again, it is straightforward to convert the recurrence equation into a working function:

```
//compute the nth Fibonacci number
//n must be non-negative!

int
fibonacci(int n)
    {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
    }
```

Our implementation is straightforward and elegant. Unfortunately, it's horribly inefficient in C. Unlike our recursive version of *factorial* which recurred about as many times as the size of the number sent to the function, our recursive version of Fibonacci will recur many, many more times than the size of its input. Here's why.

Consider the call to `fib(6)`. Tracing all the recursive calls to *fib*, we get:

```
fib(6) is fib(5) + fib(4)
```

Replacing `fib(5)` with `fib(4) + fib(3)`, we get:

```
fib(6) is fib(4) + fib(3) + fib(4)
```

We can already see a problem, we will compute `fib(4)` twice, once from the original call to `fib(6)` and again when we try to find `fib(5)`. If we write down all the recursive calls generated by `fib(6)` with each recursive call indented from the previous, we get a structure that looks like this:

```
fib(6)
    fib(5)
        fib(4)
            fib(3)
                fib(2)
                    fib(1)
                    fib(0)
                fib(1)
            fib(2)
                fib(1)
                fib(0)
        fib(3)
            fib(2)
                fib(1)
                fib(0)
            fib(1)
    fib(4)
        fib(3)
            fib(2)
                fib(1)
                fib(0)
            fib(1)
        fib(2)
            fib(1)
            fib(0)
```

We would expect, based on how the Fibonacci sequence is generated, to take about six "steps" to calculate `fib(6)`. Instead, ultimately there were 13 calls[6] to either `fib(1)` or `fib(0)`. There was a tremendous amount of duplicated and, therefore, wasted effort. An important part of Computer Science is figuring out how to reduce the wasted effort. One way to keep the recursive nature without the penalty of redundant computations is to *cache* previously computed values. Another way is to use an iterative loop:

```
int
fib(int n)
    {
    int i;
    int previous = 0;
    int current = 1;
    for (i = 0; i < n; ++i)
        {
        int temp = previous;
        previous = current;
        current = previous + temp;
        }
    return previous;
    }
```

Here, the recursive version, for all its faults, is much, much easier to understand. Complications of the iterative version include, why is *previous* returned and not *current*? And where did the variable *temp* come from? In the Chapter **??**, we will see a way to combine the clarity of recursion with the efficieny of loops.

[6] 13 is 7^{th} Fibonacci number and seven is one more than six. A coincidence? Maybe...or maybe not!

Chapter 22

Recursion Patterns

You can download the functions defined or used in this chapter with the following commands:

```
wget troll.cs.ua.edu/ACP-C/recursionPatterns.c
wget troll.cs.ua.edu/ACP-C/recursionPatterns.h
```

These files will help you run the test code listed in this chapter. You will also need to download the integer *list* module.

22.1 Manipulating arrays and lists with recursion

Recursion, arrays, and lists go hand-in-hand. What follows are a number of recursive patterns involving arrays and lists that you should be able to recognize and implement. Recursion for lists makes heavy use of the *head* and *tail* functions from the previous chapter. As all the examples in this chapter use lists of integer values. We will also make use of the following analogs for arrays:

list expression	*equivalent array expression*
`head(items)`	`items[0]`
`tail(items)`	`items+1`
`items == 0`	`size == 0`

The last expression tests whether the list (or array) is "empty". Note that an array name points to the first slot in the array. Suppose the name of the array is *items*. Then *items* + 1 points to the second slot in the array. If the size of the array *items* is n, the the size of the array pointed to by *items* + 1 must be $n - 1$. This is why adding one to the array name is equivalent to taking the tail of a list.

22.1.1 The *counting* pattern

The *counting* pattern is used to count the number of items in a collection. If the collection is empty, then its count of items is zero. The following function counts and ultimately returns the number of items in an array:

```
int
countList(Node *items)
    {
    if (items == 0) //base case
        return 0;
    else
        return 1 + countList(tail(items));
    }
```

The functions works on the observation that if you count the number of items in the tail of a list, then the number of items in the entire array is one plus that number. The extra one added in accounts for the head item that omitted when the tail was counted.

The array version is similar:

```
    int
    countArray(int *items,int size)
        {
        if (size == 0) //base case
            return 0;
        else
            return 1 + countArray(items+1,size-1);
        }
```

As with using a loop, the counting pattern for arrays is a bit useless as we need to the the size of the array before we can determine its size. However, the pattern is a basis for all other recursive patterns for arrays:

- recursive functions on an array need the size of the array
- the base case tests that the size has reached zero
- taking the “tail” of the array means the size needs to be reduced by one

22.1.2 The *accumulate* pattern

The *accumulate* pattern is used to combine all the values in a collection. The following function performs a summation of the list values:

```
    int
    sumList(Node *items)
        {
        if (items == 0) //base case
            return 0;
        else
            return head(items) + sumList(tail(items));
        }
```

Recall that the *head* function calls *getNodeValue* on the node at the front of a list of items. Note that the only difference between the *count* function and the *sum* function is the recursive case adds in the value of the head item, rather than just counting the head item. That the functions *countList* and *sumList* look similar is no coincidence. In fact, most recursive functions, especially those working on collections, look very similar to one another.

The equivalent *sumArray* is left as an exercise.

22.1.3 The *filtered-count* and *filtered-accumulate* patterns

A variation on the *counting* and *accumulate* patterns involves *filtering*. When filtering, we use an additional `if` statement to decide whether or not we should count the item, or in the case of accumulating, whether or not the item ends up in the accumulation.

Suppose we wish to count the number of even items in a list:

```
int
countListEvens(Node *items)
    {
    if (items == 0) //base case
        return 0;
    else if (head(items) % 2 == 0)
        return 1 + countListEvens(tail(items));
    else
        return 0 + countListEvens(tail(items));
    }
```

The base case states that there are zero even numbers in an empty list. The first recursive case simply counts the head item if it is even and so adds 1 to the count of even items in the remainder of the list. The second recursive case does not count the head item (because it is not even) and so adds in a 0 to the count of the remaining items. Of course, the last return would almost always be written as:

```
return countListEvens(tail(items));
```

The array version is similar:

```
int
countArrayEvens(int *items,int size)
    {
    if (size == 0) //base case
        return 0;
    else if (items[0] % 2 == 0) //head item always at index 0
        return 1 + countArrayEvens(items+1,size-1);
    else
        return countArrayEvens(items+1,size-1);
    }
```

As another example of *filtered counting*, we can pass in a value and count how many times that value occurs:

```
int
occurrencesInList(int target,Node *items)
    {
    if (items == 0)
        return 0;
    else if (head(items) == target)
        return 1 + occurrencesInList(target,tail(items));
    else
        return occurrencesInList(target,tail(items));
    }
```

An example of a *filtered-accumulation* would be to sum the even-numbered integers in an array:

```
int
sumArrayEvens(int *items,int size)
    {
    if (size == 0)
        return 0;
    else if (isEven(items[0]))
        return items[0] + sumArrayEvens(items+1,size-1);
    else
        return sumArrayEvens(items+1,size-1);
    }
```

22.1.4 The *extreme* and *extreme index* patterns

The extreme and extreme index patterns are a bit more difficult to implement recursively, due to the need to carry more information around while making recursive calls. For example, here is an example of the extreme pattern for arrays. We use a helper function that carries the extra information around:

```
int
getMax(int *a,int size)
    {
    return getMaxHelper(a,size,a[0],1); //start looking at index 1
    }
```

The third argument of the helper function is the assumed largest value. The fourth argument is the index at which one starts looking for a better max. Here is the helper function:

```
int
getMaxHelper(int *a,int size,int best,int i)
    {
    if (i == s) //no more indices to check
        return best;
    else if (a[i] > best) //found a bigger number!
        return getMaxHelper(a,s,a[i],i+1)
    else //best is still best
        return getMaxHelper(a,s,best,i+1)
    }
```

The first recursive case finds a better "best" number at the current index *i*, so it replaces the old *best* with the better candidate in the recursive call. The second recursive case doesn't replace *best*. In both recursive cases, the index is incremented so that subsequent elements can be checked.

The extreme index patter for arrays is similar:

```
int
getMaxIndex(int *a,int size)
    {
    return getMaxIndexHelper(a,size,0,1); //start looking at index 1
    }
```

In the case of *getMaxIndex*, the index of the best candidate so far is passed around, rather than its value. The implementation of *getMaxIndexHelper* is left as an exercise.

A similar strategy is used to find extreme values in a linked list.

22.1.5 The *filter* pattern

A special case of a filtered-accumulation is called *filter*. Instead of summing the filtered items (for example), we collect the filtered items into a collection. The new collection is said to be a *reduction* of the original collection.

Suppose we wish to extract the even numbers from a list. The structure of the code looks very much like the *sumEvens* function in the previous section, but instead of adding in the desired item, we join the item to the reduction of the tail of the array:

```
Node *
extractListEvens(Node *items)
    {
    if (items == 0)
        return 0;
    else if (isEven(head(items)))
        return join(head(items),extractListEvens(tail(items)));
    else
        return extractListEvens(tail(items));
    }
```

Given a list of integers, *extractEvens* returns a (possibly empty) list of the even numbers:

```
//test (compile with ilist.c and recursionPatterns.c)
#include "ilist.h"
#include "recursionPatternsPatterns.h"
int numbers[] = {4,2,5,2,7,0,8,3,7};
Node *a;
a = arrayToList(numbers,sizeof(numbers)/sizeof(int));
displayList(extractEvens(a));
```

Running this code yields:

```
{4}{2}{2}{0}{8}
```

Recall that, in the previous chapter, our attempt at filtering with a loop left the collected items in the opposite order. A recursive implementation preserves the order of values found in the original list.

With *extractEvens* defined, then *sumListEvens* could be written very simply as:

```
int
sumListEvens(Node *items)
    {
    return sumList(extractListEvens(items));
    }
```

Unlike many patterns, recursive filtering is best suited for lists; recursive filtering of arrays is very problematic and will not be dealt with here.

22.1.6 The *map* pattern

Mapping is a task closely coupled with that of reduction, but rather than collecting certain items, as with the *filter* pattern, we collect all the items. As we collect, however, we transform each item as we collect it. The basic pattern looks like this:

```
Node *
mapList(int (*f)(int),Node *items)
    {
    if (items == 0)
        return 0;
    else
        {
        int v = f(head(items));
        return join(v,mapList(f,tail(items)));
        }
    }
```

Here, *f* is used to transform each item in the recursive step. As with mapping using a loop, we use a function pointer as the type of the formal parameter *f*.

Suppose we wish to subtract one from each element in an array. First we need a transforming function that reduces its argument by one:

```
int decrement(int x) { return x - 1; }
```

Now we can "map" the *decrement* function over an array of numbers:

```
//test (compile with ilist.c and recursionPatterns.c)
#include "ilist.h"
#include "recursionPatterns.h"
int numbers[] = {4,3,7,2,4,3,1};
Node *a = arrayToList(numbers,sizeof(numbers) / sizeof(int));
Node *b = mapList(decrement,a);
displayList(b);
```

The code yields the following output:

```
{3}{2}{6}{1}{3}{2}{0}
```

We see that each value in the resulting list is one less than the corresponding number in the original *numbers* array.

Both map and filtering (reduce) are used heavily by Google in programming their search strategies.

22.1.7 The *search* pattern

The *search* pattern is a slight variation of *filtered-counting*. Suppose we wish to see if a value is present in a list. We can use a filtered-counting approach and if the count is greater than zero, we know that the item was indeed in the list:

```
int
findInList(int target,Node *items)
    {
    return occurrencesInList(target,items) > 0;
    }
```

In this case, *occurrencesInList* helps *find* do its job. We call such functions, naturally, *helper functions*. We can improve the efficiency of *find* by having it perform the search, but short-circuiting the search once the target item is found. We do this by turning the first recursive case into a second base case:

```
int
findInList2(int target,Node *items)
    {
    if (items == 0)                     //empty list, not found
        return 0;
    else if (head(items) == target)     //short-circuit!
        return 1;
    else
        return findInList2(target,tail(items));
    }
```

When the list is empty, we return false because if the item had been list, we would have hit the second base case (and returned true) before hitting the first. If neither base case hits, we simple search the remainder of the list (the recursive case). If the second base case never hits, the first base case eventually will.

22.1.8 The *shuffle* pattern

Sometimes, we wish to combine two lists. This is easy to do with the *append* function:

```
append(list1,list2)
```

This places the first element in the second list after the last element in the first list. However, many times we wish to intersperse the elements from the first list with the elements in the second list. This is known as a *shuffle*, so named since it is similar to shuffling a deck of cards. When a deck of cards is shuffled, the deck is divided in two halves (one half is akin to the first list and the other half is akin to the second list). Next the two halves are interleaved back into a single deck (akin to the resulting third list). Note that while appending is destructive, in that it changes the first list, shuffling is non-destructive, in the neither of the shuffled lists are modified.

We can use recursion to shuffle two lists. If both lists are exactly the same length, the recursive function is easy to implement using the *accumulate* pattern:

```
Node *
```

```
shuffle(Node *list1,Node *list2)
    {
    if (list1 == 0)
        return 0;
    else
        {
        Node *rest = shuffle(tail(list1),tail(list2));
        return join(head(list1),join(head(list2),rest));
        }
    }
```

If *list1* is empty (which means *list2* is empty since they have the same number of elements), the function returns the empty, since shuffling nothing together yields nothing. Otherwise, we shuffle the tails of the lists (the result is stored in the temporary variable *rest*), then join the the first elements of each list to the shuffled remainders.

If you have ever shuffled a deck of cards, you will know that it is rare for the deck to be split exactly in half prior to the shuffle. Can we amend our shuffle function to deal with this problem? Indeed, we can, by simply placing the extra cards (list items) at the end of the shuffle. We don't know which list (*list1* or *list2*) will go empty first, so we test for each list becoming empty in turn:

```
Node *
shuffle2(Node *list1,Node *list2)
    {
    if (list1 == 0)
        return list2;
    else if (list2 == 0)
        return list1;
    else
        {
        Node *rest = shuffle2(tail(list1),tail(list2));
        return join(head(list1),join(head(list2),rest));
        }
    }
```

If either list is empty, we return the other. Only if both are not empty do we execute the recursive case.

One can make shuffling even simpler by manipulating the recursive call that shuffles the remainder of the lists. If we flip the order of the lists in the recursive call, we only need to deal with the first list becoming empty:

```
Node *
shuffle3(Node *list1,Node *list2)
    {
    if (list1 == 0)
        return list2;
    else
        return join(head(list1),shuffle3(list2,tail(list1)));
    }
```

Note that in the recursive call, we take the tail of *list1* since we joined the head of *list1* to the resulting shuffle. The base case returns *list2* because if *list1* is empty, the "shuffle" of the remaining elements is just *list2*. Also, even if *list1* and *list2* are both empty, the base case test returns the correct value of zero, the null pointer or empty list.

Finally, note how much simpler it is to shuffle two lists using recursion as compared to shuffling two arrays with loops.

22.1.9 The *merge* pattern

With the *shuffle* pattern, we always took the head elements from both lists at each step in the shuffling process. Sometimes, we wish to place a constraint of the choice of elements. For example, suppose the two lists to be combined are sorted and we wish the resulting list to be sorted as well. The following example shows that shuffling does not always work:

```
    //test (compile with ilist.c and recursionPatterns.c)
    #include "ilist.h"
    #include "recursionPatterns.h"
    int n1[6] = {1,4,6,7,8,11};
    int n2[4] = {2,3,5,9};
    Node *c = shuffle3(arrayToList(n1,6),arrayToList(n2,4));
    displayList(c);
```

The result is shuffled, but not sorted:

```
    {1}{2}{4}{3}{6}{5}{7}{9}{8}{11}
```

The *merge* pattern is used to ensure the resulting list is sorted and is based upon the *filtered-accumulate* pattern. We only accumulate an item *if* it is the smallest item in the two lists:

```
    Node *
    mergeList(Node *list1,Node *list2)
        {
        if (list1 == 0)         //list1 is empty, return list2
            return list2;
        else if (list2 == 0)    //list2 is empty, return list1
            return list1;
        else if (head(list1) < head(list2))
            return join(head(list1),mergeList(tail(list1),list2));
        else //the head of list2 is smaller
            return join(head(list2),mergeList(list1,tail(list2)));
        }
```

As with *shuffle2*, we don't know which list will become empty first, so we check both in turn.

In the first recursive case, the first element of the first list is smaller than the first element of the second list. So we accumulate the first element of the first list and recur, sending the tail of the first list because we have used/accumulated the head of that list. The second list we pass unmodified, since we did not use/accumulate an element from the second list.

In the second recursive case, we implement the symmetric version of the first recursive case, focusing on the second list rather than the first.

Testing our function:

```
//test (compile with ilist.c and recursionPatterns.c)
#include "ilist.h"
#include "recursionPatterns.h"
int n1[6] = {1,4,6,7,8,11};
int n2[4] = {2,3,5,9};
Node *c = mergeList(arrayToList(n1,6),arrayToList(n2,4));
displayList(c);
```

yields:

```
{1}{2}{3}{4}{5}{6}{7}{8}{9}{11}
```

22.1.10 The *fossilized* pattern

If a recursive function mistakenly never makes the problem smaller, the problems is said to be *fossilized*. Without ever smaller problems, the base case is never reached and the function recurs[1] forever. This condition is known as an *infinite recursive loop*. Here is an example:

```
int
countList(Node *items)
   {
   if (n == 0)
       return 0;
   else
       return 1 + countList(items); //should be tail(items)
   }
```

Since *countList* recurs on the same list it was given, *items* never gets smaller so it never becomes empty. Fossilizing the problem is a common error made by both novice and expert programmers alike.

22.1.11 The *bottomless* pattern

Related to the *fossilized* pattern is the *bottomless* pattern. With the *bottomless* pattern, the problem gets smaller, but the base case is never reached. Here is a function that attempts to divide a positive number by two, by seeing how many times you can subtract two from the number:[2]

```
int
div2(int n)
    {
    if (n == 0)
        return 0;
    else
        return 1 + div2(n - 2);
    }
```

Things work great for a while:

[1]The word is *recurs*, not *recurses*!

[2]Yes, division is just repeated subtraction, just like multiplication is repeated division.

```
//test (compile with ilist.c and recursionPatterns.c)
#include "ilist.h"
#include "recursionPatterns.h"
printf("div2(16) is %d\n",div2(16));
```

yields:

```
div2(16) is 8
```

and:

```
printf("div2(134) is %d\n",div2(134));
```

yields:

```
div2(134) is 67
```

But then, something goes terribly wrong:

```
printf("div2(7) is %d\n",div2(7));
```

results in the program crashing:

```
Segmentation fault      (core dumped)
```

What happened? To see, let's *visualize* our function, as we did with the *gcd* function previously, by adding a *print* statement:

```
int
div2(int n)
    {
    printf("div2(%d)...\n",n);
    if (n == 0)
        return 0;
    else
        return 1 + div2(n - 2);
    }
```

Now every time the function is called, both originally and recursively, we can see how the value of n is changing:

```
div2(7)...
div2(5)...
div2(3)...
div2(1)...
div2(-1)...
div2(-3)...
...
Segmentation fault      (core dumped)
```

Now we can see why things went wrong, the value of n skipped over the value of zero and just kept on going. The solution is to change the base case to catch odd (and even) numbers:

```
int
div2(int n)
    {
    if (n <= 1)
        return 0;
    else
        return 1 + div2(n - 2);
    }
```

Remember, when you see a recursion depth exceeded error, you likely have implemented either the fossilized or the bottomless pattern.

Chapter 23

Comparing Recursion and Looping

You can download the functions defined or used in this chapter with the following commands:

```
wget troll.cs.ua.edu/ACP-C/recursionLoop.c
wget troll.cs.ua.edu/ACP-C/recursionLoop.h
```

These files will help you run the test code listed in this chapter.

23.1 Converting Recursions to Iterative loops

In previous chapters, we learned about repeatedly evaluating the same code using both recursion and loops. Now we compare and contrast the two techniques by implementing the three mathematical functions from Chapter **??**: *factorial*, *fibonacci*, and *gcd*, with loops.

23.1.1 Factorial

Recall that the factorial function, written recursively, looks like this:

```
int
factorial(int n)
    {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
    }
```

We see that is a form of the *accumulate* pattern. So our factorial function using a loop should look something like this:

```
int
factorialLoop(int n)
    {
    int i;
    int total = ???;
    for (i = ???; i < ???; i += ???)
```

```
            {
            total *= ???;
            }
        return total;
        }
```

Since we are accumulating a product, it makes sense that *total* should be initialized to 1. Also, since we need to multiply all the values from from 1 to n to compute the factorial, it makes sense to have the loop variable i take on all those values:

```
    int
    factorialLoop(int n)
        {
        int i;
        int total = 1;
        for (i = 1; i < n+1; ++i)
            {
            total *= ???;
            }
        return total;
        }
```

Finally, we accumulate i into the total:

```
    int
    factorialLoop(int n)
        {
        int i;
        int total = 1;
        for (i = 1; i < n+1; ++i)
            {
            total *= i;
            }
        return total;
        }
```

The limit on the for loop is set to $n + 1$ instead of n because we want n to be included in the total.

Now, compare the loop version to the recursive version. Both contain about the same amount of code, but the recursive version is easier to ascertain as correct.

23.1.2 The greatest common divisor

Here is a slightly different version of the *gcd* function, built using the following recurrence:

gcd(*a*,*b*)	is	*a*	if *b* is zero
gcd(*a*,*b*)	is	*gcd*(*b*,*a* % *b*)	otherwise

The function allows one more recursive call than the **??**. By doing so, we eliminate the need for the local variable *remainder*. Here is the implementation:

```
int
gcd(int a,int b)
    {
    if (b == 0)
        return a;
    else
        return gcd(b,a % b);
    }
```

Let's turn it into a looping function. This style of recursion doesn't fit any of the patterns we know, so we'll have to start from scratch. We do know that b becomes the new value of a and a % b becomes the new value of b on every recursive call, so the same thing must happen on every evaluation of the loop body. We stop when b is equal to zero so we should continue looping while b is *not* equal to zero. These observations lead us to this implementation:

```
int
gcdLoop(int a,int b)
    {
    while (b != 0)
        {
        a = b;
        b = a % b;
        }
    return a;
    }
```

Unfortunately, this implementation is faulty, since we've lost the original value of a by the time we perform the modulus operation. Reversing the two statements in the body of the loop:

```
{
b = a % b;
a = b;
}
```

is no better; we lose the original value of b by the time we assign it to a. What we need to do is temporarily save the original value of b before we assign a's value. Then we can assign the saved value to a after b has been reassigned:

```
int
gcdLoop(int a,int b)
    {
    while (b != 0)
        {
        int temp = b;
        b = a % b;
        a = temp;
        }
    return a;
    }
```

Now the function is working correctly. But why did we temporarily need to save a value in the loop version and not in the recursive version? The answer is that the recursive call does not perform any assignments so no values were lost. On the recursive call, new versions of the formal parameters a and b received the computations performed for the function call. The old versions were left untouched.

23.1.3 The Fibonacci sequence

Recall the recursive implementation for finding the n^{th} Fibonacci number:

```
int
fibonacci(int n)
    {
    if (n < 2)
        return n;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
```

For brevity, we have collapsed the two base cases into a single base case. If n is zero, zero is returned and if n is one, one is returned, as before.

Let's try to implement *fib* using an iterative loop. As before, this doesn't seem to fit a pattern, so we start by reasoning about this. If we let a be the first Fibonacci number, zero, and b be the second Fibonacci number, one, then the third fibonacci number would be $a + b$, which we can save in a variable named c. At this point, the fourth Fibonacci number would be $b + c$, but since we are using a loop, we need to have the code be the same for each iteration of the loop. If we let a have the value of b and b have the value of c, then the fourth Fibonacci number would be $a + b$ again. This leads to our implementation:

```
int
fibonacciLoop(int n)
    {
    int i;          //the loop variable
    int a = 0;      //the first Fibonacci number
    int b = 1;      //the second Fibonacci number
    for (i = 0; i < n; ++i) //n steps
        {
        int c = a + b;
        a = b;
        b = c;
        }
    return a;
    }
```

In the loop body, we see that *fibonacciLoop* is much like *gcdLoop*; the second number becomes the first number and some combination of the first and second number becomes the second number. In the case of *gcdLoop*, the combination was the remainder and, in the case of *fib*, the combination is sum. A rather large question remains, why does the function return a instead of b or c? The reason is, suppose *fib* was called with a value of zero, which is supposed to generate the first Fibonacci number. The loop does not run in this case and the value of a is returned, which is zero, as required. If a value of one is passed to *fib*, then the loop runs exactly once and a gets the original value of b, which is one. The loop exits and this time, one is returned, as required. So, empirically, it appears that the value of a is the correct choice of return value. As with factorial, hitting on the right way to proceed iteratively is not exactly straightforward, while the recursive version practically wrote itself.

23.2 Transforming loops into recursions

Transforming a recursive function to a loop sometimes takes a good deal of thought, but going the other way is somewhat easier. To transform an iterative loop into a recursive loop, one first identifies those variables that exist outside the loop but are referenced by the loop; these variable will become formal parameters in the recursive function. One then builds a helper function that has these "outside" variables as formal parameters. Finally, one builds a wrapper function that calls the helper with the initial values of the "outer" variables.

23.2.1 Fibonacci

For an example conversion, the *fibonacciLoop* function defined previously has a loop; that loop has four "outside" variables that are referenced by the *for* loop: a, b, i [1], and n. The variable c is used only inside the loop and thus is ignored.

Given this, we start out our recursive function like so:

```
int
fibonacciHelper(int a,int b,int i,int n)
    {
    ...
    }
```

If our original function was named *XXX*, we will name this new function *XXXHelper*. The original loop test becomes an *if* test in the body of the *fibonacciHelper* function:

```
int
fibonacciHelper(int a,int b,int i,int n)
    {
    if (i < n)
        ...
    else
        ...
    }
```

The *if-true* block becomes the recursive call. The arguments to the recursive call encode the updates to the loop variables. On the other hand, the *if-false* block becomes the value the loop attempted to calculate:

```
int
fibonacciHelper(int a,int b,int i,int n)
    {
    if (i < n)
        return fibonacciHelper(b,a + b,i + 1,n);
    else
        return a;
    }
```

Remember, a gets the value of b and b gets the value of c which is $a + b$. Since we are performing recursion with no assignments, we don't need the variable c anymore. The loop variable i is incremented by one each time. Because n is unchanged by the original loop, is is unchanged in the recursive call.

[1]The loop variable is considered a outside variable changed by the loop.

Next, we define a function with the same signature as the original function. We change the name, affixing the number 2, to remind us that this version works via a different sort of recursion compared to the original recursive *fibonacci* function. The body of this new function returns a call to the helper function:

```
int
fibonacci2(int n)
    {
    return fibonacciHelper(???,???,???,n);
    }
```

To complete our task, we fill out the arguments to the helper function with the initial values of the variables referenced by the original loop:

```
int
fibonacci2(int n)
    {
    return fibonacciHelper(0,1,0,n);
    }
```

Since a starts at 0, b starts at 1, and i starts at zero for the original loop, we pass those values in the initial call to the helper function. As before, since n never changes, we simply pass n along to the helper.

Note that this recursive function looks nothing like our original *fibonacci*. However, it suffers from none of the inefficiencies of the original version and yet it performs no assignments.[2] The reason for its efficiency is that it performs the exact calculations and number of calculations as the iterative loop-based function.

23.2.2 Factorial

For more practice, let's convert the iterative version of *factorial* into a recursive function using this method. We'll again end up with a different recursive function than before. For convenience, here is the loop version:

```
int
factorialLoop(int n)
    {
    int i;
    int total = 1;
    for (i = 1; i < n+1; ++i)
        {
        total *= i;
        }
    return total;
    }
```

We start, as before, by working on the helper function. In this case, only three outside variables are referenced by the loop: *total*, *i*, and *n*:

```
int
```

[2]A style of programming that uses no assignments is called *functional* programming and is very important in theorizing about the nature of computation.

```
factorialHelper(int total,int i,int n)
    {
    ...
    }
```

Next, we write the *if* statement:

```
int
factorialHelper(int total,int i,int n)
    {
    if (i < n + 1)
        return factorialHelper(total * i,i + 1,n);
    else
        return total;
    }
```

Next, we define the recursive *factorial* function, which calls the helper:

```
int
factorial2(int n)
    {
    return factorialHelper(1,1,n);
    }
```

23.3 Which way to go?

The moral of this story is that any iterative loop can be rewritten as a recursion and any recursion can be rewritten as an iterative loop. Moreover, in *good* languages,[3] there is no reason to prefer one way over the other, either in terms of the time it takes or the space used in execution. To reiterate, use a recursion if that makes the implementation more clear, otherwise, use an iterative loop.

[3]Unfortunately, C is not a good language in this regard, but the language *Scheme* is. When the value of a recursive call is immediately returned (i.e., not combined with some other value), a function is said to be *tail recursive*. The Scheme programming language optimizes tail recursive functions so that they are just as efficient as loops both in time and in space utilization.

Chapter 24

Matrices

You can download the functions defined or used in this chapter with the following commands:

```
wget troll.cs.ua.edu/ACP-C/matrix.c
wget troll.cs.ua.edu/ACP-C/matrix.h
```

These files will help you run the test code listed in this chapter. You will also need to download the *scanner* module.

24.1 Matrices

Matrices, commonly used in mathematics, are easily represented by two-dimensional arrays in C. A two-dimensional (2D) array is an array whose elements are arrays themselves[1]. Matrices can be divided into rows and columns. A matrix with p rows and q columns is said to be an pxq matrix. The same is true for 2D arrays, so we will use the term matrix and 2D array interchangeably. We will also focus in matrices of integers in this chapter. Matrices of other types would be processed similarly.

24.2 Creating a matrix

Matrices can be allocated statically or dynamically. A static allocation representing a 3x4 matrix m would be very similar to that of a simple array:

```
int m[3][4];
```

Note the extra pair of square brackets; the first set of brackets contains the number of rows while the second pair contains the number of columns. As with simple arrays, multi-dimensional arrays allocated this way are uninitialized and, therefore, filled with garbage. One can initialize multi-dimensional arrays when statically allocated; To initialize the matrix:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

we would structure the initializers by row:

[1]A three-dimensional array would be an array whose elements are arrays whose elements are arrays. Arrays of two dimensions and higher are known as multi-dimensional arrays.

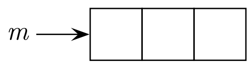
```
    int m[3][4] =
        {
        { 1, 2, 3, 4 },
        { 5, 6, 7, 8 },
        { 9,10,11,12 },
        };
```

Passing static multi-dimensional arrays to functions is not straightforward (see the last section in the chapter on Arrays). Therefore, we will focus on multi-dimensional arrays that are allocated dynamically.

Matrices need to be allocated in stages. The first stage allocates the the *backbone* of the matrix. The backbone consists of an array of one-dimensional array pointers, with the number of slots in the backbone equal to the number of rows[2]:

```
    int **m = allocate(sizeof(int *) * rows);
```

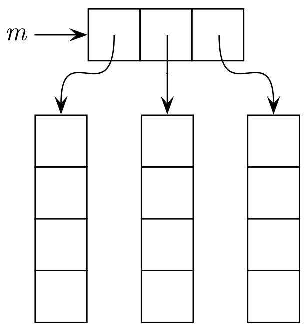
Recall that the *allocate* function is a wrapper for *malloc*. Pictorially, *m* looks like this:



The next step is to assign each slot in the backbone to point to an array with the number of slots equal to the number of columns:

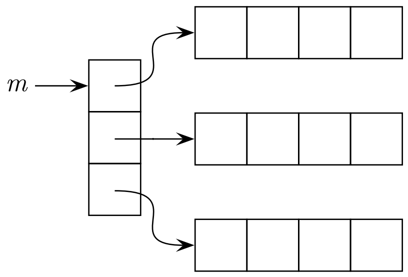
```
    for (r = 0; r < rows; ++r)
        {
        m[r] = allocate(sizeof(int) * cols);
        }
```

Now *m* looks like this:



Let's rotate the image 90 degrees so that the rows of the matrix are oriented horizontally, the natural way of viewing the matrix:

[2]This choice of the backbone representing the rows is purely arbitrary, but when this choice is made, the matrix is said to be *row-ordered*. If the opposite choice is made, that the backbone represents the columns, then the matrix is said to be *column-ordered*. Most programming languages provide row-ordered multi-dimensional arrays.



Putting it all together, we can define a constructor that creates a matrix:

```
int **
newMatrix(int rows,int cols)
    {
    int r;
    int **m = allocate(sizeof(int *) * rows);   //allocate the backbone
    for (r = 0; r < rows; ++r)
        {
        m[r] = allocate(sizeof(int) * cols);    //allocate a row
        }
    return m;
    }
```

One way to improve this function is to pass in an initializer for the slots of the constructed array. After the row is allocated, one would loop over the columns in the row, setting each element to the initializer.

Another interesting enhancement is to fill an array with random values:

```
int **
newRandomMatrix(int rows,int cols,int max)
    {
    int r,c;
    int **m = allocate(sizeof(int *) * rows);   //allocate the backbone
    for (r = 0; r < rows; ++r)
        {
        m[r] = allocate(sizeof(int) * cols);    //allocate a row
        for (c = 0; c < cols; ++c)              //fill the row
            m[r][c] = random() % max;
        }
    return m;
    }
```

Note the addition of the inner loop which fills the newly allocated row with random values. Those random values are limited in size by *max*.

24.3 Retrieving and modifying values in a matrix

To retrieve a value at row r and column c from a matrix m, one uses an expression similar to:

```
value = m[r][c];
```

One can set the value of the element at row r and column c with an expression similar to:

```
m[r][c] = value;
```

Because matrices are built from simple arrays, the first row has index 0 and the first column has index 0 as well. Thus the first value in the first row in matrix m can be found at:

```
m[0][0]
```

Where is the last element in the last row found in a matrix m with x rows and y columns?

24.4 Working with matrices

Matrices are typically processed with with two nested *for* loops. The outer loop runs over the rows, while the inner loop runs over the columns of each row. Here is a generic function for working with a matrix:

```
void
processMatrix(int **m,int rows,int cols)
    {
    int r,c;
    for (r = 0; r < rows; ++r)
        for (c = 0; c < cols; ++c)
            {
            //do something with m[r][c]
            ...
            }
    }
```

As with all functions that process arrays, we need to pass in the dimensions of the array.

One useful thing to do with a matrix is visualize it. In other words, we might wish to print out its contents. Using the general purpose template *processMatrix* as a guide, we simply substitute a print statement for the processing comment:

```
void
displayMatrix(int *m,int rows,int cols)
    {
    int r,c;
    for (r = 0; r < rows; ++r)
        for (c = 0; c < cols; ++c)
            {
            printf("%d\n",m[r][c]);
            }
    }
```

If we were to print this matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

we would see this output:

```
1
2
3
4
5
6
```

The good news is we printed every element as desired. The bad news is our output looks nothing like a matrix; every element is printed on a line by itself. We can fix this problem by replacing the `\n` in *printf*'s guide string with a space:

```
printf("%d ",m[r][c]);
```

Our output becomes:

```
1 2 3 4 5 6
```

Better, but we need a newline after each row is printed. The easiest way to do this is to print the newline after the inner loop completes:

```
void
displayMatrix(int **m,int rows,int cols)
    {
    int r,c;
    for (r = 0; r < rows; ++r)
        {
        for (c = 0; c < cols; ++c)
            {
            printf("%3d ",m[r][c]); //%3d to pad values with spaces
            }
        printf("\n");
        }
    }
```

Note how we had to add braces to the outer loop since now that loop consists of two actions, the inner loop and the printing of the newline. Now our output becomes:

```
  1   2   3
  4   5   6
```

which is a reasonable approximation to the way a matrix is supposed to look. We can test both our constructor and display functions:

```
//test (compile with matrix.c)
#include "matrix.h"
int **m = newRandomMatrix(3,4,100); //100 is the limit on random values
displayMatrix(m,3,4);
```

This code yields:

```
83  86  77  15
93  35  86  92
49  21  62  27
```

Note that no value reaches the limit (the limit is exclusive).

24.5 Reading matrix data from a file

Reading a 2-dimensional array from a file is similar to reading a simple array. As a reminder, here is a function for doing just that:

```
void
readMatrix(char *fileName,int **m,int rows,int cols)
    {
    int r,c;
    FILE *fp = fopen(fileName,"r"); //fopen failure check omitted

    for (r = 0; r < rows; ++r)
        for (c = 0; c < cols; ++c)
            m[r][c] = readInt(fp);

    fclose(fp);
    }
```

This function assumes the file being read has enough values in it to fill the matrix.

As a final note, use of the scanner means that the data in the input file does not need to be organized in matrix form; all the matrix elements could be on a single line, for example.

24.6 Matrix patterns

Matrix patterns are very similar to patterns that loop over simple arrays.

24.6.1 The filtered-count pattern

For our first example of a matrix pattern consider a filtered-count implementation for a simple array:

```
int
countArrayEvens(int *items,int size)
    {
    int i;
    int count = 0;
    for (i = 0; i < size; ++i)
        {
        if (isEven(items[i]))
            ++count;
        }
    return count;
    }
```

The analogous function for matrices is similar:

```
int
countMatrixEvens(int **items,int rows,int cols)
    {
    int r,c;
    int count = 0;
    for (r = 0; r < rows; ++r)
        for (c = 0; c < cols; ++c)
            {
            if (isEven(items[r][c]))
                ++count;
            }
    return count;
    }
```

Indeed, most of the patterns for matrices follow directly from the simple array versions. The only change is having two formal parameters specifying the size and two loops for walking the matrix.

24.6.2 The *extreme* and *extreme index* patterns

Here is an instance of the *extreme* pattern:

```
int
matrixMin(int **m,int rows,int cols)
    {
    int r,c,min;
    min = m[0][0];              //assume first value is the minimum
    for (r = 0; r < rows; ++r)
        for (c = 0; c < cols; ++c)
            {
            if (m[r][c] < min)
                min = m[r][c];
            }
    return min;
    }
```

Like the implementation for simple arrays, we start out by assuming the "first" value is the minimum. In the case of a matrix, the first value is at row 0, column 0.

Finding the extreme index is a bit trickier since one needs to save two indices, the row *and* the column of the extreme value. We also need to return two values, so we choose to return nothing. Instead, we update the addresses of the variables that are to hold the row and column of the minimum value:

```
void
matrixMinIndex(int **m,int rows,int cols,int *minr,int *minc)
    {
    int r,c;
    *minr = 0;
    *minc = 0;
    for (r = 0; r < rows; ++r)
```

```
        for (c = 0; c < cols; ++c)
            {
            if (m[r][c] < m[*minr][*minc])
                {
                *minr = r;
                *minc = c;
                }
            }
    }
```

To call this function, one needs to remember to send addresses for the last two arguments:

```
//test (compile with matrix.c)
#include "matrix.h"
int minRow,minCol;
int **m = newRandomMatrix(3,4,100); //3 rows,4 columns,values limited by 100
matrixMinIndex(m,3,4,&minRow,&minCol);
displayMatrix(m,3,4);
printf("the minimum values is at m[%d][%d]\n",minRow,minCol);
```

The output of this code is:

```
 83  86  77  15
 93  35  86  92
 49  21  62  27
the minimum values is at m[0][3]
```

By inspection, we see that the smallest value is 15 and it is located in the first row, row zero, and the last column, column three.

Things can get even more complicated with matrices. Suppose you wished to find the index of the row with the largest sum. We can simplify the complexity of this task by taking advantage of the fact that a row in a matrix is a simple array and by using a helper function:

```
int
largestRow(int **m,int rows,int cols)
    {
    int r,index,largestSum;
    index = 0;
    largestSum = sum(m[0],cols);        //sum the 1st row with a helper
    for (r = 1; r < rows; ++r)          //skip first row
        {
        int s = sum(m[r],cols);         //sum this row and see if larger
        if (s > largestSum)
            {
            index = r;
            largestSum = s;
            }
        }
    return index;
    }
```

```
int
sum(int *items,int size)
    {
    int i,total;
    total = 0;
    for (i = 0; i < size; ++i)
        {
        total += items[i];
        }
    return total;
    }
```

Curiously, the *largestRow* function does not appear to follow the pattern of other matrix functions; it only has one loop over the rows. However, it calls a helper function from within its loop and that helper function loops over the columns, so we still have our nested loops.

The helper function implements, of course, an accumulation.

24.6.3 The *search* pattern

One can return from the innermost loop, if one knows what one is doing. This makes implementing a search rather easy:

```
int
findMatrix(int target,int **m,int rows,int cols)
    {
    int r,c;
    for (r = 0; r < rows; ++r)
        {
        for (c = 0; c < cols; ++c)
            {
            if (m[r][c] == target)
                return 1; //true
            }
        }
    return 0; //false
    }
```

Be careful where you place the `return 0`, though. It has to be outside the outer loop.

24.7 Passing static arrays to functions

When you pass a static array to a function, the compiler converts the array name to a pointer. Therefore, the formal parameter that receives the array must be a pointer type. For example, given this call to *f*:

```
int a[10];
int x = f(a);
```

The definition of function *f* might start out something like:

```
int f(int *z)
    {
    ...
```

When we have a two-dimensional static array, things get more complicated. You might expect that a pointer to such an array would be defined with two asterisks, since one interpretation of:

```
int **p;
```

is that *p* can point to a two-dimensional array of integers. This is true *if* the array was dynamically allocated. What happens if we set this pointer to a static array?

```
int w[10][12];
int **p = w;
```

Compiling this fragment yields the following warning:

```
int **p = w;
      ^ initialization from incompatible pointer type
```

Upon the assignment, the compiler converts *w* into a pointer and then assigns that address to pointer *p*. What is wrong? The variable *p* holds the address of a pointer to an integer while the pointer generated from *w* holds the address of an array of 12 integers. These are not the same types, hence the warning.

Here is how one specifies a pointer to an array (of 12 integers) in C:

```
int w[10][12];
int (*q)[12] = w;
```

Now *w* and *q* can be used interchangeably, at least with respect to setting and getting values of the slots. The parentheses in the definition of q are necessary since square brackets have a higher precedence than asterisk. The definition:

```
int *q[12];
```

means *q* is an array of twelve integer pointers, while

```
int (*q)[12];
```

means *q* is a pointer to an array of twelve integers.

To pass a two-dimensional array, the receiving formal parameter must have a type similar to *q* above. Given this call to *g*:

```
double z[3][8];
double y = g(z);
```

the definition of function g would start out something like:

```
double g(double (*m)[8])
    {
    ...
```

For pointers to multi-dimensional static arrays, one elides the first set of brackets, replacing them with the asterisk, using parentheses to override the precedence of the remaining brackets. Here is an example using a four-dimensional array:

```
char alpha[3][7][5][8];
char (*beta)[7][5][8] = alpha;
```

24.8 Simulating 2D arrays with simple arrays

Although C and most other programming languages allow for two-dimensional arrays, some restricted languages do not. If you are programming in such a language and need a two-dimensional array, not to worry; two-dimensional arrays can be faked with simple arrays. The trick is to convert a two-dimensional address (row index and column index) into a one dimensional address (simple index).

Consider this matrix:

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix}$$

Using our second attempt at displaying this matrix, we would have obtained the following output:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

which looks remarkably like a simple, one-dimensional array! In fact, we can think of this output representing a simple array in which the rows of our two-dimensional array have been stored sequentially.

Now consider extracting the 6 from the 2D version. If our matrix was bound to the variable m, the location of the 6 would be:

```
m[1][2]
```

If our simple array version were bound to the variable n, then the 6 can be found at:

```
n[6]
```

since the numbers in the array reflect the indices of the array. It turns out that there is a very simple relationship between the 2D address and the 1D address. Since the rows are laid out sequentially in the 1D case and since the row index in the 2D case is 1, that means there is an entire row preceding the row of interest in the 1D case. Since the rows are 4 columns long, then four elements precede the row of interest. The column of interest has two columns preceding it, so summing the preceding row elements and the preceding column elements yields 6, exactly the address in the the simple array!

In general, the index of a 2D location in a 1D array can be computed by the following function:

```
int
simple2DIndex(int r,int c,int cols)
    {
    return r * cols + c;
    }
```

Note that this function must know the number of columns in the matrix in order to make its computation. If we wished to display a 1D simulation of a 2D array, our display function might look like this:

```
void
display1DMatrix(int *n,int size,int cols)
    {
    int r,c,rows;
    rows = size / cols;
    for (r = 0; r < rows; ++r)
        {
        for (c = 0; c < cols; ++c)
            printf("%3d ",n[simple2DIndex(r,c)]);
        printf("\n");
        }
    }
```

Testing our simulation:

```
//test (compile with matrix.c)
#include "matrix.h"
int n[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
display1DMatrix(n,sizeof(n)/sizeof(int),3); //three column format
```

yields:

```
  1   2   3
  4   5   6
  7   8   9
 10  11  12
```

In general, when simulating a 2D array, one replaces all indexing of the form:

```
m[j][k]
```

with

```
m[j * cols + k]
```

24.9 Problems

- Modify the *displayMatrix* function to print out the vertical bars that delineate a matrix. That is to say, the display function should output something like:

```
| 1 2 3 |
| 4 5 6 |
```

- Modify the *displayMatrix* function to make the displayed matrix even fancier:

```
+       +
| 1 2 3 |
| 4 5 6 |
+       +
```

- It is relatively easy to transpose a square matrix (a square matrix has the number of rows and the number of columns equal). Here is a such a routine:

```
void
transposeSquare(int **m,int size)
    {
    int r,c;
    for (r = ???; r < ???; r += ???)
        for (c = ???; c < ???; c += ???)
             {
             int old = m[r][c];
             m[r][c] = ???;
             m[c][r] = ???;
             }
    }
```

  Complete and test this procedure.