How to Think Like a Computer Scientist

C Version

Allen B. Downey

C-Version by Thomas Scheffler

Version 1.09

January 5th, 2018

The LaTeX source for this book, and more information about the Open Source Textbook project, is available from

`http://www.cs.colby.edu/~downey/ost`

or by writing to Allen B. Downey, 5850 Mayflower Hill, Waterville, ME 04901.

This book was typeset by the author using LaTeX and dvips, which are both free, open-source programs.

Contents

Chapter 1

The way of the program

The goal of this book, and this class, is to teach you to think like a computer scientist. I like the way computer scientists think because they combine some of the best features of Mathematics, Engineering, and Natural Science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem-solving**. By that I mean the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called "The way of the program."

On one level, you will be learning to program, which is a useful skill by itself. On another level you will use programming as a means to an end. As we go along, that end will become clearer.

1.1 What is a programming language?

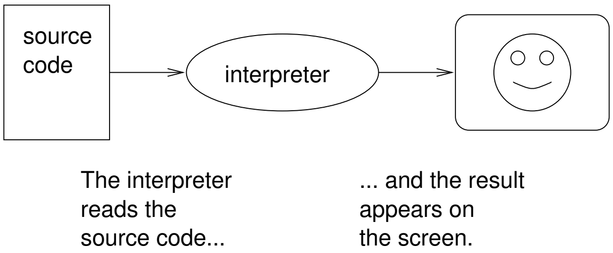
The programming language you will be learning is C, which was developed in the early 1970s by Dennis M. Ritchie at the Bell Laboratories. C is an example of a **high-level language**; other high-level languages you might have heard of are Pascal, C++ and Java.

As you might infer from the name "high-level language," there are also **low-level languages**, sometimes referred to as machine language or assembly language. Loosely-speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be translated before they can run. This translation takes some time, which is a small disadvantage of high-level languages.

But the advantages are enormous. First, it is *much* easier to program in a high-level language; by "easier" I mean that the program takes less time to write, it's shorter and easier to read, and it's more likely to be correct. Secondly, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can only run on one kind of computer, and have to be rewritten to run on another.

Due to these advantages, almost all programs are written in high-level languages. Low-level languages are only used for a few special applications.

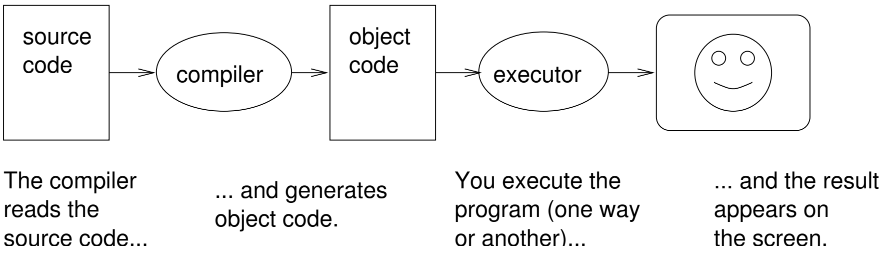
There are two ways to translate a program; **interpreting** or **compiling**. An interpreter is a program that reads a high-level program and does what it says. In effect, it translates the program line-by-line, alternately reading lines and carrying out commands.


source code
interpreter
The interpreter reads the source code...
... and the result appears on the screen.

A compiler is a program that reads a high-level program and translates it all at once, before executing any of the commands. Often you compile the program as a separate step, and then execute the compiled code later. In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**.

As an example, suppose you write a program in C. You might use a text editor to write the program (a text editor is a simple word processor). When the program is finished, you might save it in a file named `program.c`, where "program" is an arbitrary name you make up, and the suffix `.c` is a convention that indicates that the file contains C source code.

Then, depending on what your programming environment is like, you might leave the text editor and run the compiler. The compiler would read your source code, translate it, and create a new file named `program.o` to contain the object code, or `program.exe` to contain the executable.


source code
compiler
object code
executor
The compiler reads the source code...
... and generates object code.
You execute the program (one way or another)...
... and the result appears on the screen.

The next step is to run the program, which requires some kind of executor. The role of the executor is to load the program (copy it from disk into memory) and make the computer start executing the program.

Although this process may seem complicated, in most programming environments (sometimes called development environments), these steps are automated for you. Usually you will only have to write a program and press a button or type a single command to compile and run it. On the other hand, it is useful to know what the steps are that are happening in the background, so that if something goes wrong you can figure out what it is.

1.2 What is a program?

A program is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, like solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, like searching and replacing text in a document or (strangely enough) compiling a program.

The instructions, which we will call **statements**, look different in different programming languages, but there are a few basic operations most languages can perform:

input: Get data from the keyboard, or a file, or some other device.

output: Display data on the screen or send data to a file or other device.

math: Perform basic mathematical operations like addition and multiplication.

testing: Check for certain conditions and execute the appropriate sequence of statements.

repetition: Perform some action repeatedly, usually with some variation.

That's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of statements that perform these operations. Thus, one way to describe programming is the process of breaking a large, complex task up into smaller and smaller subtasks until eventually the subtasks are simple enough to be performed with one of these basic operations.

1.3 What is debugging?

Programming is a complex process, and since it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

There are a few different kinds of errors that can occur in a program, and it is useful to distinguish between them in order to track them down more quickly.

1.3.1 Compile-time errors

The compiler can only translate a program if the program is syntactically correct; otherwise, the compilation fails and you will not be able to run your program. **Syntax** refers to the structure of your program and the rules about that structure.

For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a syntax error. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e e cummings without spewing error messages.

Compilers are not so forgiving. If there is a single syntax error anywhere in your program, the compiler will print an error message and quit, and you will not be able to run your program.

To make matters worse, there are more syntax rules in C than there are in English, and the error messages you get from the compiler are often not very helpful. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.

1.3.2 Run-time errors

The second type of error is a run-time error, so-called because the error does not appear until you run the program.

C is not a **safe** language, such as Java, where run-time errors are rare. Programming in C allows you to get very close to the actual computing hardware. Most run-time errors C occur because the language provides no protection against the accessing or overwriting of data in memory.

For the simple sorts of programs we will be writing for the next few weeks, run-time errors are rare, so it might be a little while before you encounter one.

1.3.3 Logic errors and semantics

The third type of error is the **logical** or **semantic** error. If there is a logical error in your program, it will compile and run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying logical errors can be tricky, since it requires you to work backwards by looking at the output of the program and trying to figure out what it is doing.

1.3.4 Experimental debugging

One of the most important skills you will acquire in this class is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways debugging is like detective work. You are confronted with clues and you have to infer the processes and events that lead to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (from A. Conan Doyle's *The Sign of Four*).

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should always start with a working program that does *something*, and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux" (from *The Linux Users' Guide* Beta Version 1).

In later chapters I will make more suggestions about debugging and other programming practices.

1.4 Formal and natural languages

Natural languages are the languages that people speak, like English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

> **Programming languages are formal languages that have been designed to express computations.**

As I mentioned before, formal languages tend to have strict rules about syntax. For example, $3 + 3 = 6$ is a syntactically correct mathematical statement, but

$3 = +6\$$ is not. Also, H_2O is a syntactically correct chemical name, but $_2Zz$ is not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, like words and numbers and chemical elements. One of the problems with `3=+6$` is that `$` is not a legal token in mathematics (at least as far as I know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz.

The second type of syntax rule pertains to the structure of a statement; that is, the way the tokens are arranged. The statement `3=+6$` is structurally illegal, because you can't have a plus sign immediately after an equals sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this unconsciously). This process is called **parsing**.

For example, when you hear the sentence, "The other shoe fell," you understand that "the other shoe" is the subject and "fell" is the verb. Once you have parsed a sentence, you can figure out what it means, that is, the semantics of the sentence. Assuming that you know what a shoe is, and what it means to fall, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common—tokens, structure, syntax and semantics—there are many differences.

ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness: Natural languages are full of idiom and metaphor. If I say, "The other shoe fell," there is probably no shoe and nothing falling. Formal languages mean exactly what they say.

People who grow up speaking a natural language (everyone) often have a hard time adjusting to formal languages. In some ways the difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose: The literal meaning of words is more important and the structure contributes more meaning. Prose is more amenable to analysis than poetry, but still often ambiguous.

Programs: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, remember that the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.5 The first program

Traditionally the first program people write in a new language is called "Hello, World." because all it does is display the words "Hello, World." In C, this program looks like this:

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

/* main: generate some simple output */

int main(void)
{
      printf("Hello, World.\n");
      return(EXIT_SUCCESS);
}
```

Some people judge the quality of a programming language by the simplicity of the "Hello, World." program. By this standard, C does reasonably well. Even so, this simple program contains several features that are hard to explain to beginning programmers. For now, we will ignore some of them, like the first two lines.

The third line begins with `/*` and ends with `*/`, which indicates that it is a **comment**. A comment is a bit of English text that you can put in the middle of a program, usually to explain what the program does. When the compiler sees a `/*`, it ignores everything from there until it finds the corresponding `*/`.

In the forth line, you notice the word `main`. `main` is a special name that indicates the place in the program where execution begins. When the program runs, it starts by executing the first **statement** in `main()` and it continues, in order, until it gets to the last statement, and then it quits.

There is no limit to the number of statements that can be in `main()`, but the example contains only two. The first is an **output** statement, meaning that it displays or prints a message on the screen. The second statement tells the operating system that our program executed successfully.

The statement that prints things on the screen is `printf()`, and the characters between the quotation marks will get printed. Notice the `\n` after the last character. This is a special character called *newline* that is appended at the end of a line of text and causes the cursor to move to the next line of the display. The next time you output something, the new text appears on the next line. At the end of the statement there is a semicolon (`;`), which is required at the end of every statement.

There are a few other things you should notice about the syntax of this program. First, C uses curly-brackets ({ and }) to group things together. In this case, the output statement is enclosed in curly-brackets, indicating that it is *inside* the definition of `main()`. Also, notice that the statement is indented, which helps to show visually which lines are inside the definition.

At this point it would be a good idea to sit down in front of a computer and compile and run this program. The details of how to do that depend on your programming environment, this book assumes that you know how to do it.

As I mentioned, the C compiler is very pedantic with syntax. If you make any errors when you type in the program, chances are that it will not compile successfully. For example, if you misspell `stdio.h`, you might get an error message like the following:

```
hello_world.c:1:19: error: sdtio.h: No such file or directory
```

There is a lot of information on this line, but it is presented in a dense format that is not easy to interpret. A more friendly compiler might say something like:

> "On line 1 of the source code file named hello_world.c, you tried to include a header file named sdtio.h. I didn't find anything with that name, but I did find something named stdio.h. Is that what you meant, by any chance?"

Unfortunately, few compilers are so accomodating. The compiler is not really very smart, and in most cases the error message you get will be only a hint about what is wrong. It will take some time for you to learn to interpret different compiler messages.

Nevertheless, the compiler can be a useful tool for learning the syntax rules of a language. Starting with a working program (like hello_world.c), modify it in various ways and see what happens. If you get an error message, try to remember what the message says and what caused it, so if you see it again in the future you will know what it means.

1.6 Glossary

problem-solving: The process of formulating a problem, finding a solution, and expressing the solution.

high-level language: A programming language like C that is designed to be easy for humans to read and write.

low-level language: A programming language that is designed to be easy for a computer to execute. Also called "machine language" or "assembly language."

formal language: Any of the languages people have designed for specific purposes, like representing mathematical ideas or computer programs. All programming languages are formal languages.

natural language: Any of the languages people speak that have evolved naturally.

portability: A property of a program that can run on more than one kind of computer.

interpret: To execute a program in a high-level language by translating it one line at a time.

compile: To translate a program in a high-level language into a low-level language, all at once, in preparation for later execution.

source code: A program in a high-level language, before being compiled.

object code: The output of the compiler, after translating the program.

executable: Another name for object code that is ready to be executed.

statement: A part of a program that specifies an action that will be performed when the program runs. A print statement causes output to be displayed on the screen.

comment: A part of a program that contains information about the program, but that has no effect when the program runs.

algorithm: A general process for solving a category of problems.

bug: An error in a program.

syntax: The structure of a program.

semantics: The meaning of a program.

parse: To examine a program and analyze the syntactic structure.

syntax error: An error in a program that makes it impossible to parse (and therefore impossible to compile).

logical error: An error in a program that makes it do something other than what the programmer intended.

debugging: The process of finding and removing any of the three kinds of errors.

1.7 Exercises

Exercise 1.1

Computer scientists have the annoying habit of using common English words to mean something different from their common English meaning. For example, in English, a statement and a comment are pretty much the same thing, but when we are talking about a program, they are very different.

The glossary at the end of each chapter is intended to highlight words and phrases that have special meanings in computer science. When you see familiar words, don't assume that you know what they mean!

a. In computer jargon, what's the difference between a statement and a comment?

b. What does it mean to say that a program is portable?

c. What is an executable?

Exercise 1.2

Before you do anything else, find out how to compile and run a C program in your environment. Some environments provide sample programs similar to the example in Section 1.5.

a. Type in the "Hello World" program, then compile and run it.

b. Add a second print statement that prints a second message after the "Hello World.". Something witty like, "How are you?" Compile and run the program again.

c. Add a comment line to the program (anywhere) and recompile it. Run the program again. The new comment should not affect the execution of the program.

This exercise may seem trivial, but it is the starting place for many of the programs we will work with. In order to debug with confidence, you have to have confidence in your programming environment. In some environments, it is easy to lose track of which program is executing, and you might find yourself trying to debug one program while you are accidentally executing another. Adding (and changing) print statements is a simple way to establish the connection between the program you are looking at and the output when the program runs.

Exercise 1.3

It is a good idea to commit as many errors as you can think of, so that you see what error messages the compiler produces. Sometimes the compiler will tell you exactly what is wrong, and all you have to do is fix it. Sometimes, though, the compiler will produce wildly misleading messages. You will develop a sense for when you can trust the compiler and when you have to figure things out yourself.

a. Remove the closing curly-bracket (}).

b. Remove the opening curly-bracket ({).

c. Remove the `int` before `main`.

d. Instead of `main`, write `mian`.

e. Remove the closing `*/` from a comment.

f. Replace `printf` with `pintf`.

g. Delete one of the parentheses: `(` or `)`. Add an extra one.

h. Delete the semicolon after the `return` statement.

Chapter 2

Variables and types

2.1 More output

As I mentioned in the last chapter, you can put as many statements as you want in `main()`. For example, to output more than one line:

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

/* main: generate some simple output */

int main (void)
{
      printf ("Hello World.\n");     /* output one line */
      printf ("How are you?\n");     /* output another line */
      return (EXIT_SUCCESS);
}
```

As you can see, it is legal to put comments at the end of a line, as well as on a line by themselves.

The phrases that appear in quotation marks are called **strings**, because they are made up of a sequence (string) of letters. Actually, strings can contain any combination of letters, numbers, punctuation marks, and other special characters.

Often it is useful to display the output from multiple output statements all on one line. You can do this by leaving out the `\n` from the first `printf`:

```
int main (void)
{
      printf ("Goodbye, ");
      printf ("cruel world!\n");
      return (EXIT_SUCCESS);
}
```

In this case the output appears on a single line as `Goodbye, cruel world!`. Notice that there is a space between the word “Goodbye,” and the second quotation mark. This space appears in the output, so it affects the behavior of the program.

Spaces that appear outside of quotation marks generally do not affect the behavior of the program. For example, I could have written:

```
int main(void)
{
printf("Goodbye, ");
printf("cruel world!\n");
return(EXIT_SUCCESS);
}
```

This program would compile and run just as well as the original. The breaks at the ends of lines (newlines) do not affect the program’s behavior either, so I could have written:

```
int main(void){printf("Goodbye, ");printf("cruel world!\n");
return(EXIT_SUCCESS);}
```

That would work, too, although you have probably noticed that the program is getting harder and harder to read. Newlines and spaces are useful for organizing your program visually, making it easier to read the program and locate syntax errors.

2.2 Values

Computer programs operate on values stored in computer memory. A value —like a letter or a number— is one of the fundamental things that a program manipulates. The only values we have manipulated so far are the strings we have been outputting, like `"Hello, world."`. You (and the compiler) can identify these string values because they are enclosed in quotation marks.

There are different kinds of values, including integers and characters. It is important for the program to know exactly what kind of value is manipulated because not all manipulations will make sense on all values. We therefore distinguish between different **types** of values.

An integer is a whole number like 1 or 17. You can output integer values in a similar way as you output strings:

```
printf("%i\n", 17);
```

When we look at the `printf()` statement more closely, we notice that the value we are outputting no longer appears inside the quotes, but behind them separated by comma. The string is still there, but now contains a `%i` instead of any text. The `%i` a placeholder that tells the `printf()` command to print an integer value. Several such placeholders exist for different data types and formatting options of the output. We will see the next one just now.

A character value is a letter or digit or punctuation mark enclosed in single quotes, like `'a'` or `'5'`. You can output character values in a similar way:

```
printf("%c\n", '}');
```

This example outputs a single closing curly-bracket on a line by itself. It uses the `%c` placeholder to signify the output of a character value.

It is easy to confuse different types of values, like `"5"`, `'5'` and `5`, but if you pay attention to the punctuation, it should be clear that the first is a string, the second is a character and the third is an integer. The reason this distinction is important should become clear soon.

2.3 Variables

One of the most powerful features of a programming language is the ability to manipulate values through the use of **variables**. So far the values that we have used in our statements where fixed to what was written in the statement. Now we will use a variable as a named location that stores a value.

Just as there are different types of values (integer, character, etc.), there are different types of variables. When you create a new variable, you have to declare what type it is. For example, the character type in C is called `char`. The following statement creates a new variable named `fred` that has type `char`.

```
char fred;
```

This kind of statement is called a **declaration**.

The type of a variable determines what kind of values it can store. A `char` variable can contain characters, and it should come as no surprise that `int` variables can store integers.

Contrary to other programming languages, C does not have a dedicated variable type for the storage of string values. We will see in a later chapter how string values are stored in C.

To create an integer variable, the syntax is

```
int bob;
```

where `bob` is the arbitrary name you choose to identify the variable. In general, you will want to make up variable names that indicate what you plan to do with the variable. For example, if you saw these variable declarations:

```
char first_letter;
char last_letter;
int hour, minute;
```

you could probably make a good guess at what values would be stored in them. This example also demonstrates the syntax for declaring multiple variables with the same type: `hour` and `minute` are both integers (`int` type).

ATTENTION: The older C89 standard allows variable declarations only at the beginning of a block of code. It is therefore necessary to put variable declarations before any other statements, even if the variable itself is only needed much later in your program.

2.4 Assignment

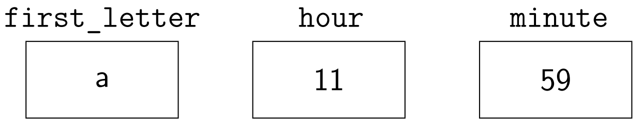
Now that we have created some variables, we would like to store values in them. We do that with an **assignment statement**.

```
first_letter = ’a’;   /* give first_letter the value ’a’ */
hour = 11;            /* assign the value 11 to hour */
minute = 59;          /* set minute to 59 */
```

This example shows three assignments, and the comments show three different ways people sometimes talk about assignment statements. The vocabulary can be confusing here, but the idea is straightforward:

- When you declare a variable, you create a named storage location.
- When you make an assignment to a variable, you give it a value.

A common way to represent variables on paper is to draw a box with the name of the variable on the outside and the value of the variable on the inside. This kind of figure is called a **state diagram** because is shows what state each variable is in (you can think of it as the variable's "state of mind"). This diagram shows the effect of the three assignment statements:



When we assign values to variables, we have to make sure that the assigned value correspondents to the type of the variable. In C a variable has to have the same type as the value you assign. For example, you cannot store a string in an `int` variable. The following statement generates a compiler warning:

```
int hour;
hour = "Hello.";       /* WRONG !! */
```

This rule is sometimes a source of confusion, because there are many ways that you can convert values from one type to another, and C sometimes converts things automatically. But for now you should remember that as a general rule variables and values have the same type, and we'll talk about special cases later.

Another source of confusion is that some strings *look* like integers, but they are not. For example, the string `"123"`, which is made up of the characters `1`, `2` and `3`, is not the same thing as the *number* `123`. This assignment is illegal:

```
minute = "59";         /* WRONG!! */
```

2.5 Outputting variables

You can output the value of a variable using the same commands we used to output simple values.

```
int hour, minute;
char colon;

hour = 11;
minute = 59;
colon = ':';

printf ("The current time is ");
printf ("%i", hour);
printf ("%c", colon);
printf ("%i", minute);
printf ("\n");
```

This program creates two integer variables named `hour` and `minute`, and a character variable named `colon`. It assigns appropriate values to each of the variables and then uses a series of output statements to generate the following:

```
The current time is 11:59
```

When we talk about "outputting a variable," we mean outputting the *value* of the variable. The name of a variable only has significance for the programmer. The compiled program no longer contains a human readable reference to the variable name in your program.

The `printf()` command is capable of outputting several variables in a single statement. To do this, we need to put placeholders in the so called *format string*, that indicate the position where the variable value will be put. The variables will be inserted in the order of their appearance in the statement. It is important to observe the right order and type for the variables.

By using a single output statement, we can make the previous program more concise:

```
int hour, minute;
char colon;

hour = 11;
minute = 59;
colon = ':';

printf ("The current time is %i%c%i\n", hour, colon, minute);
```

On one line, this program outputs a string, two integers and a character. Very impressive!

2.6 Keywords

A few sections ago, I said that you can make up any name you want for your variables, but that's not quite true. There are certain words that are reserved in C because they are used by the compiler to parse the structure of your program, and if you use them as variable names, it will get confused. These words, called **keywords**, include `int`, `char`, `void` and many more.

Reserved keywords in the C language

```
auto
break
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if
inline
int
long
register
restrict
return
short
signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile
while
_Bool
_Complex
_Imaginary
```

The complete list of keywords is included in the C Standard, which is the official language definition adopted by the the International Organization for Standardization (ISO) on September 1, 1998.

Rather than memorize the list, I would suggest that you take advantage of a feature provided in many development environments: code highlighting. As you type, different parts of your program should appear in different colors. For example, keywords might be blue, strings red, and other code black. If you type a variable name and it turns blue, watch out! You might get some strange behavior from the compiler.

2.7 Operators

Operators are special symbols that are used to represent simple computations like addition and multiplication. Most of the operators in C do exactly what you would expect them to do, because they are common mathematical symbols. For example, the operator for adding two integers is `+`.

The following are all legal C expressions whose meaning is more or less obvious:

```
1+1         hour-1         hour*60+minute     minute/60
```

Expressions can contain both variables names and values. In each case the name of the variable is replaced with its value before the computation is performed.

Addition, subtraction and multiplication all do what you expect, but you might be surprised by division. For example, the following program:

```
int hour, minute;
hour = 11;
minute = 59;
printf ("Number of minutes since midnight: %i\n", hour*60 + minute);
printf ("Fraction of the hour that has passed: %i\n", minute/60);
```

would generate the following output:

```
Number of minutes since midnight: 719
Fraction of the hour that has passed: 0
```

The first line is what we expected, but the second line is odd. The value of the variable `minute` is 59, and 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that C is performing **integer division**.

When both of the **operands** are integers (operands are the things operators operate on), the result must also be an integer, and by definition integer division always rounds *down*, even in cases like this where the next integer is so close.

A possible alternative in this case is to calculate a percentage rather than a fraction:

```
printf ("Percentage of the hour that has passed: ");
printf ("%i\n", minute*100/60);
```

The result is:

```
Percentage of the hour that has passed: 98
```

Again the result is rounded down, but at least now the answer is approximately correct. In order to get an even more accurate answer, we could use a different type of variable, called floating-point, that is capable of storing fractional values. We'll get to that in the next chapter.

2.8 Order of operations

When more than one operator appears in an expression the order of evaluation depends on the rules of **precedence**. A complete explanation of precedence can get complicated, but just to get you started:

- Multiplication and division happen before addition and subtraction. So `2*3-1` yields 5, not 4, and `2/3-1` yields -1, not 1.

- If the operators have the same precedence they are evaluated from left to right. So in the expression `minute*100/60`, the multiplication happens first, yielding `5900/60`, which in turn yields `98`. If the operations had gone from right to left, the result would be `59*1` which is `59`, which is wrong.

- Any time you want to override the rules of precedence (or you are not sure what they are) you can use parentheses. Expressions in parentheses are evaluated first, so `2*(3-1)` is 4. You can also use parentheses to make an expression easier to read, as in `(minute*100)/60`, even though it doesn't change the result.

2.9 Operators for characters

Interestingly, the same mathematical operations that work on integers also work on characters. For example,

```
char letter;
letter = 'a' + 1;
printf ("%c\n", letter);
```

outputs the letter `b`. Although it is syntactically legal to multiply characters, it is almost never useful to do it.

Earlier I said that you can only assign integer values to integer variables and character values to character variables, but that is not completely true. In some cases, C converts automatically between types. For example, the following is legal.

```
int number;
number = 'a';
printf ("%i\n", number);
```

The result is 97, which is the number that is used internally by C to represent the letter `'a'`. However, it is generally a good idea to treat characters as characters, and integers as integers, and only convert from one to the other if there is a good reason.

Automatic type conversion is an example of a common problem in designing a programming language, which is that there is a conflict between **formalism**, which is the requirement that formal languages should have simple rules with few exceptions, and **convenience**, which is the requirement that programming languages be easy to use in practice.

More often than not, convenience wins, which is usually good for expert programmers, who are spared from rigorous but unwieldy formalism, but bad for beginning programmers, who are often baffled by the complexity of the rules and the number of exceptions. In this book I have tried to simplify things by emphasizing the rules and omitting many of the exceptions.

2.10 Composition

So far we have looked at the elements of a programming language—variables, expressions, and statements—in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, we know how to multiply integers and we know how to output values; it turns out we can do both at the same time:

```
printf ("%i\n", 17 * 3);
```

Actually, I shouldn't say "at the same time," since in reality the multiplication has to happen before the output, but the point is that any expression, involving numbers, characters, and variables, can be used inside an output statement. We've already seen one example:

```
    printf ("%i\n", hour * 60 + minute);
```

You can also put arbitrary expressions on the right-hand side of an assignment statement:

```
    int percentage;
    percentage = (minute * 100) / 60;
```

This ability may not seem so impressive now, but we will see other examples where composition makes it possible to express complex computations neatly and concisely.

WARNING: There are limits on where you can use certain expressions; most notably, the left-hand side of an assignment statement has to be a *variable* name, not an expression. That's because the left side indicates the storage location where the result will go. Expressions do not represent storage locations, only values. So the following is illegal: `minute + 1 = hour;`.

2.11 Glossary

variable: A named storage location for values. All variables have a type, which determines which values it can store.

value: A letter, or number, or other thing that can be stored in a variable.

type: The meaning of values. The types we have seen so far are integers (`int` in C) and characters (`char` in C).

keyword: A reserved word that is used by the compiler to parse programs. Examples we have seen include `int`, `void` and `char`.

statement: A line of code that represents a command or action. So far, the statements we have seen are declarations, assignments, and output statements.

declaration: A statement that creates a new variable and determines its type.

assignment: A statement that assigns a value to a variable.

expression: A combination of variables, operators and values that represents a single result value. Expressions also have types, as determined by their operators and operands.

operator: A special symbol that represents a simple computation like addition or multiplication.

operand: One of the values on which an operator operates.

precedence: The order in which operations are evaluated.

composition: The ability to combine simple expressions and statements into compound statements and expressions in order to represent complex computations concisely.

2.12 Exercises

Exercise 2.1

a. Create a new program named `MyDate.c`. Copy or type in something like the "Hello, World" program and make sure you can compile and run it.

b. Following the example in Section 2.5, write a program that creates variables named `day`, `month` and `year` What type is each variable?

   Assign values to those variables that represent today's date.

c. Print the value of each variable on a line by itself. This is an intermediate step that is useful for checking that everything is working so far.

d. Modify the program so that it prints the date in standard American form: `mm/dd/yyyy`.

e. Modify the program again so that the total output is:

```
American format:
3/18/2009
European format:
18.3.2009
```

The point of this exercise is to use the output function `printf` to display values with different types, and to practice developing programs gradually by adding a few statements at a time.

Exercise 2.2

a. Create a new program called `MyTime.c`. From now on, I won't remind you to start with a small, working program, but you should.

b. Following the example in Section 2.7, create variables named `hour`, `minute` and `second`, and assign them values that are roughly the current time. Use a 24-hour clock, so that at 2pm the value of `hour` is 14.

c. Make the program calculate and print the number of seconds since midnight.

d. Make the program calculate and print the number of seconds remaining in the day.

e. Make the program calculate and print the percentage of the day that has passed.

f. Change the values of `hour`, `minute` and `second` to reflect the current time (I assume that some time has elapsed), and check to make sure that the program works correctly with different values.

The point of this exercise is to use some of the arithmetic operations, and to start thinking about compound entities like the time of day that are represented with multiple values. Also, you might run into problems computing percentages with `ints`, which is the motivation for floating point numbers in the next chapter.

HINT: you may want to use additional variables to hold values temporarily during the computation. Variables like this, that are used in a computation but never printed, are sometimes called intermediate or temporary variables.

Chapter 3

Function

3.1 Floating-point

In the last chapter we had some problems dealing with numbers that were not integers. We worked around the problem by measuring percentages instead of fractions, but a more general solution is to use floating-point numbers, which can represent fractions as well as integers. In C, there are two floating-point types, called `float` and `double`. In this book we will use `double`s exclusively.

You can create floating-point variables and assign values to them using the same syntax we used for the other types. For example:

```
double pi;
pi = 3.14159;
```

It is also legal to declare a variable and assign a value to it at the same time:

```
int x = 1;
char first_char = "a";
double pi = 3.14159;
```

In fact, this syntax is quite common. A combined declaration and assignment is sometimes called an **initialization**.

Although floating-point numbers are useful, they are often a source of confusion because there seems to be an overlap between integers and floating-point numbers. For example, if you have the value `1`, is that an integer, a floating-point number, or both?

Strictly speaking, C distinguishes the integer value `1` from the floating-point value `1.0`, even though they seem to be the same number. They belong to different types, and strictly speaking, you are not allowed to make assignments between types. For example, the following is illegal:

```
int x = 1.1;
```

Because the variable on the left is an `int` and the value on the right is a `double`. But it is easy to forget this rule, especially because there are places where C automatically converts from one type to another. For example,

```
double y = 1;
```

should technically not be legal, but C allows it by converting the `int` to a `double` automatically. This is convenient for the programmer, but it can cause problems; for example:

```
double y = 1 / 3;
```

You might expect the variable `y` to be given the value `0.333333`, which is a legal floating-point value, but in fact it will get the value `0.0`. The reason is that the expression on the right appears to be the ratio of two integers, so C does *integer* division, which yields the integer value `0`. Converted to floating-point, the result is `0.0`.

One way to solve this problem (once you figure out what it is) is to make the right-hand side a floating-point expression:

```
double y = 1.0 / 3.0;
```

This sets `y` to `0.333333`, as expected.

All the operations we have seen—addition, subtraction, multiplication, and division—work on floating-point values, although you might be interested to know that the underlying mechanism is completely different. In fact, most processors have special hardware just for performing floating-point operations.

3.2 Constants

In the previous section we have assigned the value 3.14159 to a floating point variable. An important thing to remember about variables is, that they can hold – as their name implies – different values at different points in your program. For example, we could assign the value 3.14159 to the variable `pi` now and assign some other value to it later on:

```
double pi = 3.14159;
...
pi = 10.999;  /* probably a logical error in your program */
```

The second value is probably not what you intended when you first created the named storage location `pi` . The value for π is constant and does not change over time. Using the storage location `pi` to hold arbitrary other values can cause some very hard to find bugs in your program.

C allows you to specify the static nature of storage locations through the use of the keyword `const`. It must be used in conjunction with the required type of the constant. A value will be assigned at initialisation but can never be changed again during the runtime of the program.

```
const double PI = 3.14159;
printf ("Pi: %f\n", PI);
 ...
PI = 10.999;  /* wrong, error caught by the compiler  */
```

It is no longer possible to change the value for PI once it has been initialised, but other than this we can use it just like a variable.

In order to visually separate constants from variables we will use all uppercase letters in their names.

3.3 Converting from double to int

As I mentioned, C converts ints to doubles automatically if necessary, because no information is lost in the translation. On the other hand, going from a double to an int requires rounding off. C doesn't perform this operation automatically, in order to make sure that you, as the programmer, are aware of the loss of the fractional part of the number.

The simplest way to convert a floating-point value to an integer is to use a **typecast**. Typecasting is so called because it allows you to take a value that belongs to one type and "cast" it into another type (in the sense of molding or reforming, not throwing).

The syntax for typecasting requires the explicit specification of the target type, set in parenthesis before the expression (Type). For example:

```
const double PI = 3.14159;
int x = (int) PI;
```

The (int) operator casts the value of PI into an integer, so x gets the value 3. Converting to an integer always rounds down, even if the fraction part is 0.99999999.

For every type in C, there is a corresponding operator that typecasts its argument to the appropriate type.

3.4 Math functions

In mathematics, you have probably seen functions like sin and log, and you have learned to evaluate expressions like $\sin(\pi/2)$ and $\log(1/x)$. First, you evaluate the expression in parentheses, which is called the **argument** of the function. For example, $\pi/2$ is approximately 1.571, and $1/x$ is 0.1 (if x happens to be 10).

Then you can evaluate the function itself, either by looking it up in a table or by performing various computations. The sin of 1.571 is 1, and the log of 0.1 is -1 (assuming that log indicates the logarithm base 10).

This process can be applied repeatedly to evaluate more complicated expressions like $\log(1/\sin(\pi/2))$. First we evaluate the argument of the innermost function, then evaluate the function, and so on.

C provides a set of built-in functions that includes most of the mathematical operations you can think of. The math functions are invoked using a syntax that is similar to mathematical notation:

```
    double log = log (17.0);
    double angle = 1.5;
    double height = sin (angle);
```

The first example sets `log` to the logarithm of 17, base *e*. There is also a function called `log10` that takes logarithms base 10.

The second example finds the sine of the value of the variable `angle`. C assumes that the values you use with `sin` and the other trigonometric functions (`cos`, `tan`) are in *radians*. To convert from degrees to radians, you can divide by 360 and multiply by 2π.

If you don't happen to know π to 15 digits, you can calculate it using the `acos` function. The arccosine (or inverse cosine) of -1 is π, because the cosine of π is -1.

```
  const double PI = acos(-1.0);
  double degrees = 90;
  double angle = degrees * 2 * PI / 360.0;
```

Before you can use any of the math functions, you have to include the math **header file**. Header files contain information the compiler needs about functions that are defined outside your program. For example, in the "Hello, world!" program we included a header file named `stdio.h` using an **include** statement:

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

`stdio.h` contains information about input and output (I/O) functions available in C.

Similarly, the math header file contains information about the math functions. You can include it at the beginning of your program along with `stdio.h`:

```
#include <math.h>
```

3.5 Composition

Just as with mathematical functions, C functions can be **composed**, meaning that you use one expression as part of another. For example, you can use any expression as an argument to a function:

```
   double x = cos (angle + PI/2);
```

This statement takes the value of `PI`, divides it by two and adds the result to the value of `angle`. The sum is then passed as an argument to the `cos` function.

You can also take the result of one function and pass it as an argument to another:

```
   double x = exp (log (10.0));
```

This statement finds the log base *e* of 10 and then raises *e* to that power. The result gets assigned to `x`; I hope you know what it is.

3.6 Adding new functions

So far we have only been using the functions that are built into C, but it is also possible to add new functions. Actually, we have already seen one function definition: `main()`. The function named `main()` is special because it indicates where the execution of the program begins, but the syntax for `main()` is the same as for any other function definition:

```
void NAME ( LIST OF PARAMETERS )
{
    STATEMENTS
}
```

You can make up any name you want for your function, except that you can't call it `main` or any other C keyword. The list of parameters specifies what information, if any, you have to provide in order to use (or **call**) the new function.

`main()` doesn't take any parameters, as indicated by the parentheses containing the keyword `(void)` in it's definition. The first couple of functions we are going to write also have no parameters, so the syntax looks like this:

```
void PrintNewLine (void)
{
    printf ("\n");
}
```

This function is named `PrintNewLine()`. It contains only a single statement, which outputs a newline character. Notice that we start the function name with an uppercase letter. The following words of the function name are also capitalized. We will use this convention for the naming of functions consistently throughout the book.

In `main()` we can call this new function using syntax that is similar to the way we call the built-in C commands:

```
int main (void)
{
   printf ("First Line.\n");
   PrintNewLine ();
   printf ("Second Line.\n");
   return EXIT_SUCCESS;
}
```

The output of this program is:

```
First line.

Second line.
```

Notice the extra space between the two lines. What if we wanted more space between the lines? We could call the same function repeatedly:

```
int main (void)
{
    printf ("First Line.\n");
```

```
      NewLine ();
      NewLine ();
      NewLine ();
      printf ("Second Line.\n");
  }
```

Or we could write a new function, named `PrintThreeLines()`, that prints three new lines:

```
  void PrintThreeLines (void)
  {
    PrintNewLine ();  PrintNewLine ();  PrintNewLine ();
  }

  int main (void)
  {
      printf ("First Line.\n");
      PrintThreeLines ();
      printf ("Second Line.\n");
      return EXIT_SUCCESS;
  }
```

You should notice a few things about this program:

- You can call the same procedure repeatedly. In fact, it is quite common and useful to do so.

- You can have one function call another function. In this case, `main()` calls `PrintThreeLines()` and `PrintThreeLines()` calls `PrintNewLine()`. Again, this is common and useful.

- In `PrintThreeLines()` I wrote three statements all on the same line, which is syntactically legal (remember that spaces and new lines usually don't change the meaning of a program). On the other hand, it is usually a better idea to put each statement on a line by itself, to make your program easy to read. I sometimes break that rule in this book to save space.

So far, it may not be clear why it is worth the trouble to create all these new functions. Actually, there are a lot of reasons, but this example only demonstrates two:

1. Creating a new function gives you an opportunity to give a name to a group of statements. Functions can simplify a program by hiding a complex computation behind a single command, and by using English words in place of arcane code. Which is clearer, `PrintNewLine()` or `printf("\n")`?

2. Creating a new function can make a program smaller by eliminating repetitive code. For example, a short way to print nine consecutive new lines is to call `PrintThreeLines()` three times. How would you print 27 new lines?

3.7 Definitions and uses

Pulling together all the code fragments from the previous section, the whole program looks like this:

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

void PrintNewLine (void)
{
    printf ("\n");
}

void PrintThreeLines (void)
{
    PrintNewLine ();  PrintNewLine ();  PrintNewLine ();
}

int main (void)
{
   printf ("First Line.\n");
   PrintThreeLines ();
   printf ("Second Line.\n");
   return EXIT_SUCCESS;
}
```

This program contains three function definitions: `PrintNewLine()`, `PrintThreeLine()`, and `main()`.

Inside the definition of `main()`, there is a statement that uses or calls `PrintThreeLine()`. Similarly, `PrintThreeLine()` calls `PrintNewLine()` three times. Notice that the definition of each function appears above the place where it is used.

This is necessary in C; the definition of a function must appear before (above) the first use of the function. You should try compiling this program with the functions in a different order and see what error messages you get.

3.8 Programs with multiple functions

When you look at the C source code that contains several functions, it is tempting to read it from top to bottom, but that is likely to be confusing, because that is not the **order of execution** of the program.

Execution always begins at the first statement of `main()`, regardless of where it is in the program (often it is at the bottom). Statements are executed one at a time, in order, until you reach a function call. Function calls are like a detour in the flow of execution. Instead of going to the next statement, you go to the first line of the called function, execute all the statements there, and then come back and pick up again where you left off.

That sounds simple enough, except that you have to remember that one function can call another. Thus, while we are in the middle of `main()`, we might have to go off and execute the statements in `PrintThreeLines()`. But while we are executing `PrintThreeLines()`, we get interrupted three times to go off and execute `PrintNewLine()`.

Fortunately, C is adept at keeping track of where it is, so each time `PrintNewLine()` completes, the program picks up where it left off in `PrintThreeLine()`, and eventually gets back to `main()` so the program can terminate.

What's the moral of this sordid tale? When you read a program, don't read from top to bottom. Instead, follow the flow of execution.

3.9 Parameters and arguments

Some of the built-in functions we have used have **parameters**, which are values that you provide to let the function do its job. For example, if you want to find the sine of a number, you have to indicate what the number is. Thus, `sin()` takes a `double` value as a parameter.

Some functions take more than one parameter, like `pow()`, which takes two `doubles`, the base and the exponent.

Notice that in each of these cases we have to specify not only how many parameters there are, but also what type they are. So it shouldn't surprise you that when you write a function definition, the parameter list indicates the type of each parameter. For example:

```
void PrintTwice (char phil)
{
    printf("%c%c\n", phil, phil);
}
```

This function takes a single parameter, named `phil`, that has type `char`. Whatever that parameter is (and at this point we have no idea what it is), it gets printed twice, followed by a newline. I chose the name `phil` to suggest that the name you give a parameter is up to you, but in general you want to choose something more illustrative than `phil`.

In order to call this function, we have to provide a `char`. For example, we might have a `main()` function like this:

```
int main (void)
{
    PrintTwice ('a');
    return EXIT_SUCCESS;
}
```

The char value you provide is called an **argument**, and we say that the argument is **passed** to the function. In this case the value 'a' is passed as an argument to PrintTwice() where it will get printed twice.

Alternatively, if we had a char variable, we could use it as an argument instead:

```
int main ()
{
    char argument = 'b';
    PrintTwice (argument);
    return EXIT_SUCCESS;
}
```

Notice something very important here: the name of the variable we pass as an argument (argument) has nothing to do with the name of the parameter (phil). Let me say that again:

> **The name of the variable we pass as an argument has nothing to do with the name of the parameter.**

They can be the same or they can be different, but it is important to realize that they are not the same thing, except that they happen to have the same value (in this case the character 'b').

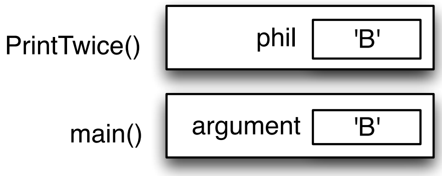
The value you provide as an argument must have the same type as the parameter of the function you call. This rule is important, but it is sometimes confusing because C sometimes converts arguments from one type to another automatically. For now you should learn the general rule, and we will deal with exceptions later.

3.10 Parameters and variables are local

Parameters and variables only exist inside their own functions. Within the confines of main(), there is no such thing as phil. If you try to use it, the compiler will complain. Similarly, inside PrintTwice() there is no such thing as argument.

Variables like this are said to be **local**. In order to keep track of parameters and local variables, it is useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but the variables are contained in larger boxes that indicate which function they belong to.

For example, the stack diagram for PrintTwice() looks like this:



Whenever a function is called, it creates a new **instance** of that function. Each instance of a function contains the parameters and local variables for that function. In the diagram an instance of a function is represented by a box with the name of the function on the outside and the variables and parameters inside.

In the example, `main()` has one local variable, `argument`, and no parameters. `PrintTwice()` has no local variables and one parameter, named `phil`.

3.11 Functions with multiple parameters

The syntax for declaring and invoking functions with multiple parameters is a common source of errors. First, remember that you have to declare the type of every parameter. For example

```
void PrintTime (int hour, int minute)
{
  printf ("%i", hour);
  printf (":");
  printf ("%i", minute);
}
```

It might be tempting to write `(int hour, minute)`, but that format is only legal for variable declarations, not for parameters.

Another common source of confusion is that you do not have to declare the types of arguments. The following is wrong!

```
    int hour = 11;
    int minute = 59;
    PrintTime (int hour, int minute);   /* WRONG! */
```

In this case, the compiler can tell the type of `hour` and `minute` by looking at their declarations. It is unnecessary and illegal to include the type when you pass them as arguments. The correct syntax is `PrintTime (hour, minute);`.

3.12 Functions with results

You might have noticed by now that some of the functions we are using, like the math functions, yield results. Other functions, like `PrintNewLine`, perform an action but don't return a value. That raises some questions:

- What happens if you call a function and you don't do anything with the result (i.e. you don't assign it to a variable or use it as part of a larger expression)?
- What happens if you use a function without a result as part of an expression, like `PrintNewLine() + 7`?
- Can we write functions that yield results, or are we stuck with things like `PrintNewLine()` and `PrintTwice()`?

The answer to the third question is “yes, you can write functions that return values,” and we’ll do it in a couple of chapters. I will leave it up to you to answer the other two questions by trying them out. Any time you have a question about what is legal or illegal in C, a good way to find out is to ask the compiler.

3.13 Glossary

constant: A named storage location similar to a variable, that can not be changed once it has been initialised.

floating-point: A type of variable (or value) that can contain fractions as well as integers. There are a few floating-point types in C; the one we use in this book is `double`.

initialization: A statement that declares a new variable and assigns a value to it at the same time.

function: A named sequence of statements that performs some useful function. Functions may or may not take parameters, and may or may not produce a result.

parameter: A piece of information you provide in order to call a function. Parameters are like variables in the sense that they contain values and have types.

argument: A value that you provide when you call a function. This value must have the same type as the corresponding parameter.

call: Cause a function to be executed.

3.14 Exercises

Exercise 3.1

The point of this exercise is to practice reading code and to make sure that you understand the flow of execution through a program with multiple functions.

a. What is the output of the following program? Be precise about where there are spaces and where there are newlines.

   HINT: Start by describing in words what `Ping()` and `Baffle()` do when they are invoked.

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

 void Ping ()
 {
    printf (".\n");
  }
```

```
void Baffle ()
{
  printf ("wug");
  Ping ();
}

void Zoop ()
{
  Baffle ();
  printf ("You wugga ");
  Baffle ();
}

int main (void)
{
  printf ("No, I ");
  Zoop ();
  printf ("I ");
  Baffle ();
  return EXIT_SUCCESS;
}
```

b. Draw a stack diagram that shows the state of the program the first time `Ping()` is invoked.

Exercise 3.2 The point of this exercise is to make sure you understand how to write and invoke functions that take parameters.

a. Write the first line of a function named `Zool()` that takes three parameters: an `int` and two `char`.

b. Write a line of code that invokes `Zool()`, passing as arguments the value `11`, the letter `a`, and the letter `z`.

Exercise 3.3

The purpose of this exercise is to take code from a previous exercise and encapsulate it in a function that takes parameters. You should start with a working solution to exercise

a. Write a function called `PrintDateAmerican()` that takes the day, month and year as parameters and that prints them in American format.

b. Test your function by invoking it from `main()` and passing appropriate arguments. The output should look something like this (except that the date might be different):

```
3/29/2009
```

c. Once you have debugged `PrintDateAmerican()`, write another function called `PrintDateEuropean()` that prints the date in European format.

Exercise 3.4

Many computations can be expressed concisely using the “multadd” operation, which takes three operands and computes `a*b + c`. Some processors even provide a hardware implementation of this operation for floating-point numbers.

a. Create a new program called `Multadd.c`.

b. Write a function called `Multadd()` that takes three `doubles` as parameters and that prints their multadditionization.

c. Write a `main()` function that tests `Multadd()` by invoking it with a few simple parameters, like `1.0, 2.0, 3.0`, and then prints the result, which should be `5.0`.

d. Also in `main()`, use `Multadd()` to compute the following value:

$$\sin\frac{\pi}{4} + \frac{\cos\frac{\pi}{4}}{2}$$

e. Write a function called `Yikes()` that takes a double as a parameter and that uses `Multadd()` to calculate and print

$$xe^{-x} + \sqrt{1 - e^{-x}}$$

HINT: the Math function for raising e to a power is `double exp(double x);`.

In the last part, you get a chance to write a function that invokes a function you wrote. Whenever you do that, it is a good idea to test the first function carefully before you start working on the second. Otherwise, you might find yourself debugging two functions at the same time, which can be very difficult.

One of the purposes of this exercise is to practice pattern-matching: the ability to recognize a specific problem as an instance of a general category of problems.

Chapter 4

Conditionals and recursion

4.1 Conditional execution

In order to write useful programs, we almost always need the ability to check certain conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the `if` statement:

```
if (x > 0)
{
    printf ("x is positive\n");
}
```

The expression in parentheses is called the condition. If it is true, then the statements in brackets get executed. If the condition is not true, nothing happens.

The condition can contain any of the **comparison operators**:

```
x == y              /* x equals y */
x != y              /* x is not equal to y */
x > y               /* x is greater than y */
x < y               /* x is less than y */
x >= y              /* x is greater than or equal to y */
x <= y              /* x is less than or equal to y */
```

Although these operations are probably familiar to you, the syntax C uses is a little different from mathematical symbols like $=$, $\neq$ and $\leq$. A common error is to use a single `=` instead of a double `==`. Remember that `=` is the assignment operator, and `==` is a comparison operator. Also, there is no such thing as `=<` or `=>`.

The two sides of a condition operator have to be the same type. You can only compare `ints` to `ints` and `doubles` to `doubles`. Unfortunately, at this point you can't compare strings at all! There is a way to compare strings, but we won't get to it for a couple of chapters.

4.2 The modulus operator

The modulus operator works on integers (and integer expressions) and yields the *remainder* when the first operand is divided by the second. In C, the modulus operator is a percent sign, `%`. The syntax is exactly the same as for other operators:

```
int quotient = 7 / 3;
int remainder = 7 % 3;
```

The first operator, integer division, yields 2. The second operator yields 1. Thus, 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if `x % y` is zero, then `x` is divisible by `y`.

Also, you can use the modulus operator to extract the rightmost digit or digits from a number. For example, `x % 10` yields the rightmost digit of `x` (in base 10). Similarly `x % 100` yields the last two digits.

4.3 Alternative execution

A second form of conditional execution is alternative execution, in which there are two possibilities, and the condition determines which one gets executed. The syntax looks like:

```
if (x%2 == 0)
{
    printf ("x is even\n");
}
else
{
    printf ("x is odd\n");
}
```

If the remainder when `x` is divided by 2 is zero, then we know that `x` is even, and this code displays a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed.

As an aside, if you think you might want to check the parity (evenness or oddness) of numbers often, you might want to “wrap” this code up in a function, as follows:

```
void PrintParity (int x)
{
    if (x%2 == 0)
    {
        printf ("x is even\n");
    }
    else
    {
        printf ("x is odd\n");
    }
}
```

Now you have a function named `PrintParity()` that will display an appropriate message for any integer you care to provide. In `main()` you would call this function as follows:

```
PrintParity (17);
```

Always remember that when you *call* a function, you do not have to declare the types of the arguments you provide. C can figure out what type they are. You should resist the temptation to write things like:

```
int number = 17;
PrintParity (int number);         /* WRONG!!! */
```

4.4 Chained conditionals

Sometimes you want to check for a number of related conditions and choose one of several actions. One way to do this is by **chaining** a series of `if`s and `else`s:

```
if (x > 0)
{
    printf ("x is positive\n");
}
else if (x < 0)
{
    printf ("x is negative\n");
}
else
{
    printf ("x is zero\n");
}
```

These chains can be as long as you want, although they can be difficult to read if they get out of hand. One way to make them easier to read is to use standard indentation, as demonstrated in these examples. If you keep all the statements and squiggly-braces lined up, you are less likely to make syntax errors and you can find them more quickly if you do.

4.5 Nested conditionals

In addition to chaining, you can also nest one conditional within another. We could have written the previous example as:

```
if (x == 0)
{
        printf ("x is zero\n");
    }
    else
    {
        if (x > 0)
        {
            printf ("x is positive\n");
        }
        else
        {
            printf ("x is negative\n");
        }
    }
```

There is now an outer conditional that contains two branches. The first branch contains a simple output statement, but the second branch contains another `if` statement, which has two branches of its own. Fortunately, those two branches are both output statements, although they could have been conditional statements as well.

Notice again that indentation helps make the structure apparent, but nevertheless, nested conditionals get difficult to read very quickly. In general, it is a good idea to avoid them when you can.

On the other hand, this kind of **nested structure** is common, and we will see it again, so you better get used to it.

4.6 The `return` statement

The `return` statement allows you to terminate the execution of a function before you reach the end. One reason to use it is if you detect an error condition:

```
#include <math.h>

void PrintLogarithm (double x)
{
    if (x <= 0.0)
    {
        printf ("Positive numbers only, please.\n");
        return;
    }

   double result = log (x);
   printf ("The log of x is %f\n", result);
}
```

This defines a function named `PrintLogarithm()` that takes a `double` named `x` as a parameter. The first thing it does is check whether `x` is less than or equal to zero, in which case it displays an error message and then uses `return` to exit the function. The flow of execution immediately returns to the caller and the remaining lines of the function are not executed.

I used a floating-point value on the right side of the condition because there is a floating-point variable on the left.

Remember that any time you want to use one a function from the math library, you have to include the header file `math.h`.

4.7 Recursion

I mentioned in the last chapter that it is legal for one function to call another, and we have seen several examples of that. I neglected to mention that it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical and interesting things a program can do.

For example, look at the following function:

```
void Countdown (int n)
{
    if (n == 0)
    {
        printf ("Blastoff!");
    }
    else
    {
        printf ("%i", n);
        Countdown (n-1);
    }
}
```

The name of the function is `Countdown()` and it takes a single integer as a parameter. If the parameter is zero, it outputs the word “Blastoff.” Otherwise, it outputs the parameter and then calls a function named `Countdown()`—itself—passing `n-1` as an argument.

What happens if we call this function like this:

```
int main (void)
{
     Countdown (3);
     return EXIT_SUCCESS;
}
```

The execution of `Countdown()` begins with `n=3`, and since `n` is not zero, it outputs the value 3, and then calls itself...

> The execution of `Countdown()` begins with `n=2`, and since `n` is not zero, it outputs the value 2, and then calls itself...

> > The execution of `Countdown()` begins with `n=1`, and since `n` is not zero, it outputs the value 1, and then calls itself...
> >
> > > The execution of `Countdown()` begins with `n=0`, and since `n` is zero, it outputs the word “Blastoff!” and then returns.
> >
> > The Countdown that got `n=1` returns.
>
> The Countdown that got `n=2` returns.

The Countdown that got `n=3` returns.

And then you’re back in `main()` (what a trip). So the total output looks like:

```
3
2
1
Blastoff!
```

As a second example, let’s look again at the functions `PrintNewLine()` and `PrintThreeLines()`.

```
void PrintNewLine ()
 {
   printf ("\n");
 }

void PrintThreeLines ()
 {
   PrintNewLine ();  PrintNewLine ();  PrintNewLine ();
 }
```

Although these work, they would not be much help if I wanted to output 2 newlines, or 106. A better alternative would be

```
void PrintLines (int n)
{
  if (n > 0)
  {
    printf ("\n");
    PrintLines (n-1);
  }
}
```

This program is similar to `Countdown`; as long as `n` is greater than zero, it outputs one newline, and then calls itself to output `n-1` additional newlines. Thus, the total number of newlines is `1 + (n-1)`, which usually comes out to roughly `n`.

The process of a function calling itself is called **recursion**, and such functions are said to be **recursive**.

4.8 Infinite recursion

In the examples in the previous section, notice that each time the functions get called recursively, the argument gets smaller by one, so eventually it gets to zero. When the argument is zero, the function returns immediately, *without making any recursive calls*. This case—when the function completes without making a recursive call—is called the **base case**.

If a recursion never reaches a base case, it will go on making recursive calls forever and the program will never terminate. This is known as **infinite recursion**, and it is generally not considered a good idea.

In most programming environments, a program with an infinite recursion will not really run forever. Eventually, something will break and the program will report an error. This is the first example we have seen of a run-time error (an error that does not appear until you run the program).

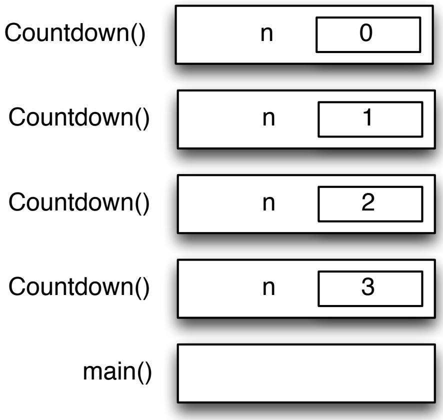
You should write a small program that recurses forever and run it to see what happens.

4.9 Stack diagrams for recursive functions

In the previous chapter we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can make it easier to interpret a recursive function.

Remember that every time a function gets called it creates a new instance that contains the function's local variables and parameters.

This figure shows a stack diagram for Countdown, called with `n = 3`:



There is one instance of `main()` and four instances of `Countdown()`, each with a different value for the parameter `n`. The bottom of the stack, `Countdown()`

with n=0 is the base case. It does not make a recursive call, so there are no more instances of Countdown().

The instance of main() is empty because main() does not have any parameters or local variables. As an exercise, draw a stack diagram for PrintLines(), invoked with the parameter n=4.

4.10 Glossary

modulus: An operator that works on integers and yields the remainder when one number is divided by another. In C it is denoted with a percent sign (%).

conditional: A block of statements that may or may not be executed depending on some condition.

chaining: A way of joining several conditional statements in sequence.

nesting: Putting a conditional statement inside one or both branches of another conditional statement.

recursion: The process of calling the same function you are currently executing.

infinite recursion: A function that calls itself recursively without every reaching the base case. Eventually an infinite recursion will cause a run-time error.

4.11 Exercises

Exercise 4.1 This exercise reviews the flow of execution through a program with multiple methods. Read the following code and answer the questions below.

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

void Zippo (int quince, int flag);

void Baffle (int output)
 {
    printf ("%i\n",output);
    Zippo (12, -5);
  }

 void Zippo (int quince, int flag)
 {
    if (flag < 0)
    {
          printf ("%i zoop\n", quince);
    }
    else
    {
       printf ("rattle ");
       Baffle (quince);
       printf ("boo-wa-ha-ha\n");
      }
  }

  int main (void)
  {
      Zippo (5, 13);
      return EXIT_SUCCESS;
  }
```

a. Write the number `1` next to the first *statement* of this program that will be executed. Be careful to distinguish things that are statements from things that are not.

b. Write the number `2` next to the second statement, and so on until the end of the program. If a statement is executed more than once, it might end up with more than one number next to it.

c. What is the value of the parameter `blimp` when `baffle()` gets invoked?

d. What is the output of this program?

Exercise 4.2 The first verse of the song "99 Bottles of Beer" is:

> 99 bottles of beer on the wall, 99 bottles of beer, ya' take one down, ya' pass it around, 98 bottles of beer on the wall.

Subsequent verses are identical except that the number of bottles gets smaller by one in each verse, until the last verse:

> No bottles of beer on the wall, no bottles of beer, ya' can't take one down, ya' can't pass it around, 'cause there are no more bottles of beer on the wall!

And then the song (finally) ends.

Write a program that prints the entire lyrics of "99 Bottles of Beer." Your program should include a recursive method that does the hard part, but you also might want to write additional methods to separate the major functions of the program.

As you are developing your code, you will probably want to test it with a small number of verses, like "3 Bottles of Beer."

The purpose of this exercise is to take a problem and break it into smaller problems, and to solve the smaller problems by writing simple, easily-debugged methods.

Exercise 4.3 You can use the `getchar()` function in C to get character input from the user through the keyboard. This function stops the execution of the program and waits for the input from the user.

The `getchar()` function has the type `int` and does not require an argument. It returns the ASCII-Code (cf. Appendix B) of the key that has been pressed on the keyboard.

a. Write a program, that asks the user to input a digit between 0 and 9.

b. Test the input from the user and display an error message if the returned value is not a digit. The program should then be terminated. If the test is successful, the program should print the input value on the computer screen.

Exercise 4.4 Fermat's Last Theorem says that there are no integers a, b, and c such that

$$a^n + b^n = c^n$$

except in the case when $n = 2$.

Write a function named `checkFermat()` that takes four integers as parameters—`a`, `b`, `c` and `n`—and that checks to see if Fermat's theorem holds. If n is greater than 2 and it turns out to be true that $a^n + b^n = c^n$, the program should print "Holy smokes, Fermat was wrong!" Otherwise the program should print "No, that doesn't work."

You should assume that there is a function named `raiseToPow()` that takes two integers as arguments and that raises the first argument to the power of the second. For example:

```
int x = raiseToPow (2, 3);
```

would assign the value `8` to `x`, because $2^3 = 8$.

Chapter 5

Fruitful functions

5.1 Return values

Some of the built-in functions we have used, like the math functions, have produced results. That is, the effect of calling the function is to generate a new value, which we usually assign to a variable or use as part of an expression. For example:

```
double e = exp (1.0);
double height = radius * sin (angle);
```

But so far all the functions we have written have been **void** functions; that is, functions that return no value. When you call a void function, it is typically on a line by itself, with no assignment:

```
PrintLines (3);
Countdown (n-1);
```

In this chapter, we are going to write functions that return things, which I will refer to as **fruitful** functions, for want of a better name. The first example is `area`, which takes a `double` as a parameter, and returns the area of a circle with the given radius:

```
double Area (double radius)
{
    double pi = acos (-1.0);
    double area = pi * radius * radius;
    return area;
}
```

The first thing you should notice is that the beginning of the function definition is different. Instead of `void`, which indicates a void function, we see `double`, which indicates that the return value from this function will have type `double`.

Also, notice that the last line is an alternate form of the `return` statement that includes a return value. This statement means, "return immediately from this function and use the following expression as a return value." The expression you provide can be arbitrarily complicated, so we could have written this function more concisely:

```
double Area (double radius)
{
    return acos(-1.0) * radius * radius;
}
```

On the other hand, **temporary** variables like `area` often make debugging easier. In either case, the type of the expression in the `return` statement must match the return type of the function. In other words, when you declare that the return type is `double`, you are making a promise that this function will eventually produce a `double`. If you try to `return` with no expression, or an expression with the wrong type, the compiler will take you to task.

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
double AbsoluteValue (double x)
{
    if (x < 0)
    {
        return -x;
    }
    else
    {
        return x;
    }
}
```

Since these returns statements are in an alternative conditional, only one will be executed. Although it is legal to have more than one `return` statement in a function, you should keep in mind that as soon as one is executed, the function terminates without executing any subsequent statements.

Code that appears after a `return` statement, or any place else where it can never be executed, is called **dead code**. Some compilers warn you if part of your code is dead.

If you put return statements inside a conditional, then you have to guarantee that *every possible path* through the program hits a return statement. For example:

```
double AbsoluteValue (double x)
{
    if (x < 0)
    {
         return -x;
    }
    else if (x > 0)
    {
            return x;
        }                              /* WRONG!! */
}
```

This program is not correct because if `x` happens to be 0, then neither condition will be true and the function will end without hitting a return statement. Unfortunately, many C compilers do not catch this error. As a result, the program may compile and run, but the return value when `x==0` could be anything, and will probably be different in different environments.

By now you are probably sick of seeing compiler errors, but as you gain more experience, you will realize that the only thing worse than getting a compiler error is *not* getting a compiler error when your program is wrong.

Here's the kind of thing that's likely to happen: you test `AbsoluteValue()` with several values of `x` and it seems to work correctly. Then you give your program to someone else and they run it in another environment. It fails in some mysterious way, and it takes days of debugging to discover that the problem is an incorrect implementation of `AbsoluteValue()`. If only the compiler had warned you!

From now on, if the compiler points out an error in your program, you should not blame the compiler. Rather, you should thank the compiler for finding your error and sparing you days of debugging. Some compilers have an option that tells them to be extra strict and report all the errors they can find. You should turn this option on all the time.

As an aside, you should know that there is a function in the math library called `fabs()` that calculates the absolute value of a `double` – correctly.

5.2 Program development

At this point you should be able to look at complete C functions and tell what they do. But it may not be clear yet how to go about writing them. I am going to suggest one technique that I call **incremental development**.

As an example, imagine you want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2). By the usual definition,

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{5.1}$$

The first step is to consider what a `Distance` function should look like in C. In other words, what are the inputs (parameters) and what is the output (return value).

In this case, the two points are the parameters, and it is natural to represent them using four `doubles`. The return value is the distance, which will have type `double`.

Already we can write an outline of the function:

```
double Distance (double x1, double y1, double x2, double y2)
{
    return 0.0;
}
```

The return statement is a placekeeper so that the function will compile and return something, even though it is not the right answer. At this stage the function doesn't do anything useful, but it is worthwhile to try compiling it so we can identify any syntax errors before we make it more complicated.

In order to test the new function, we have to call it with sample values. Somewhere in main() I would add:

```
double dist = Distance (1.0, 2.0, 4.0, 6.0);
printf  ("%f\n" dist);
```

I chose these values so that the horizontal distance is 3 and the vertical distance is 4; that way, the result will be 5 (the hypotenuse of a 3-4-5 triangle). When you are testing a function, it is useful to know the right answer.

Once we have checked the syntax of the function definition, we can start adding lines of code one at a time. After each incremental change, we recompile and run the program. That way, at any point we know exactly where the error must be—in the last line we added.

The next step in the computation is to find the differences $x_2 - x_1$ and $y_2 - y_1$. I will store those values in temporary variables named dx and dy.

```
double Distance (double x1, double y1, double x2, double y2)
{
    double dx = x2 - x1;
    double dy = y2 - y1;
    printf  ("dx is %f\n",  dx);
    printf  ("dy is %f\n", dy;
    return 0.0;
}
```

I added output statements that will let me check the intermediate values before proceeding. As I mentioned, I already know that they should be 3.0 and 4.0.

When the function is finished I will remove the output statements. Code like that is called **scaffolding**, because it is helpful for building the program, but it is not part of the final product. Sometimes it is a good idea to keep the scaffolding around, but comment it out, just in case you need it later.

The next step in the development is to square dx and dy. We could use the pow() function, but it is simpler and faster to just multiply each term by itself.

```
double Distance (double x1, double y1, double x2, double y2)
{
    double dx = x2 - x1;
    double dy = y2 - y1;
    double dsquared = dx*dx + dy*dy;
    printf  ("d_squared is %f\n", dsquared);
    return 0.0;
}
```

Again, I would compile and run the program at this stage and check the intermediate value (which should be 25.0).

Finally, we can use the `sqrt()` function to compute and return the result.

```
double Distance (double x1, double y1, double x2, double y2)
{
    double dx = x2 - x1;
    double dy = y2 - y1;
    double dsquared = dx*dx + dy*dy;
    double result = sqrt (dsquared);
    return result;
}
```

Then in `main()`, we should output and check the value of the result.

As you gain more experience programming, you might find yourself writing and debugging more than one line at a time. Nevertheless, this incremental development process can save you a lot of debugging time.

The key aspects of the process are:

- Start with a working program and make small, incremental changes. At any point, if there is an error, you will know exactly where it is.
- Use temporary variables to hold intermediate values so you can output and check them.
- Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

5.3 Composition

As you should expect by now, once you define a new function, you can use it as part of an expression, and you can build new functions using existing functions. For example, what if someone gave you two points, the center of the circle and a point on the perimeter, and asked for the area of the circle?

Let's say the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, we have a function, `Distance()`, that does that.

```
double radius = Distance (xc, yc, xp, yp);
```

The second step is to find the area of a circle with that radius, and return it.

```
double result = Area (radius);
return result;
```

Wrapping that all up in a function, we get:

```
double AreaFromPoints (double xc, double yc, double xp, double yp)
{
    double radius = Distance (xc, yc, xp, yp);
    double result = Area (radius);
    return result;
}
```

The temporary variables `radius` and `area` are useful for development and debugging, but once the program is working we can make it more concise by composing the function calls:

```
double AreaFromPoints (double xc, double yc, double xp, double yp)
{
    return Area (Distance (xc, yc, xp, yp));
}
```

5.4 Boolean values

The types we have seen so far can hold very large values. There are a lot of integers in the world, and even more floating-point numbers. By comparison, the set of characters is pretty small. Well, many computing languages implement an even more fundamental type that is even smaller. It is called **_Bool**, and the only values in it are `true` and `false`.

Unfortunately, earlier versions of the C standard did not implement boolean as a separate type, but instead used the integer values 0 and 1 to represent truth values. By convention 0 represents `false` and 1 represents `true`. Strictly speaking C interpretes any integer value different from 0 as true. This can be a source of error if you are testing a value to be true by comparing it with `1`.

Without thinking about it, we have been using boolean values in the last of chapter. The condition inside an `if` statement is a boolean expression. Also, the result of a comparison operator is a boolean value. For example:

```
if (x == 5)
{
  /* do something*/
}
```

The operator == compares two integers and produces a boolean value.

Pre C99 has no keywords for the expression of `true` or `false`. A lot of programs instead are using C preprocessor definitions anywhere a boolean expression is called for. For example,

```
#define FALSE 0
#define TRUE 1
 ...
if (TRUE)
{
    /* will be always executed  */
  }
```

is a standard idiom for a loop that should run forever (or until it reaches a `return` or `break` statement).

5.5 Boolean variables

Since boolean values are not supported directly in C, we can not declare variables of the type boolean. Instead, programmers typically use the `short` datatype in combination with preprocessor definitions to store truth values.

```
#define FALSE 0
#define TRUE 1
 ...
short fred;
fred = TRUE;
short testResult = FALSE;
```

The first line is a simple variable declaration; the second line is an assignment, and the third line is a combination of a declaration and as assignment, called an initialization.

As I mentioned, the result of a comparison operator is a boolean, so you can store it in a variable

```
short evenFlag = (n%2 == 0);      /* true if n is even */
short positiveFlag = (x > 0);     /* true if x is positive */
```

and then use it as part of a conditional statement later

```
if (evenFlag)
{
    printf("n was even when I checked it");
}
```

A variable used in this way is called a **flag**, since it flags the presence or absence of some condition.

5.6 Logical operators

There are three **logical operators** in C: AND, OR and NOT, which are denoted by the symbols `&&`, `||` and `!`. The semantics (meaning) of these operators is similar to their meaning in English. For example `x > 0 && x < 10` is true only if `x` is greater than zero AND less than 10.

`evenFlag || n%3 == 0` is true if *either* of the conditions is true, that is, if `evenFlag` is true OR the number is divisible by 3.

Finally, the NOT operator has the effect of negating or inverting a bool expression, so `!evenFlag` is true if `evenFlag` is false; that is, if the number is odd.

Logical operators often provide a way to simplify nested conditional statements. For example, how would you write the following code using a single conditional?

```
if (x > 0)
{
    if (x < 10)
    {
        printf ("x is a positive single digit.\n");
    }
}
```

5.7 Bool functions

It is sometimes appropriate for functions to return boolean values just like any other return type. This is is especially convenient for hiding complicated tests inside functions. For example:

```
int IsSingleDigit (int x)
{
    if (x >= 0 && x < 10)
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}
```

The name of this function is IsSingleDigit(). It is common to give such test functions names that sound like yes/no questions. The return type is int, which means that again we need to follow the agreement that 0 represents false and 1 represents true. Every return statement has to follow this convention, again, we are using preprocessor definitions.

The code itself is straightforward, although it is a bit longer than it needs to be. Remember that the expression x >= 0 && x < 10 is evaluated to a boolean value, so there is nothing wrong with returning it directly, and avoiding the if statement altogether:

```
int IsSingleDigit (int x)
{
    return (x >= 0 && x < 10);
}
```

In main() you can call this function in the usual ways:

```
printf("%i\n", IsSingleDigit (2));
short bigFlag = !IsSingleDigit (17);
```

The first line outputs the value `true` because 2 is a single-digit number. Unfortunately, when C outputs `boolean` values, it does not display the words `TRUE` and `FALSE`, but rather the integers `1` and `0`.

The second line assigns the value `true` to `bigFlag` only if 17 is *not* a positive single-digit number.

The most common use of boolean functions is inside conditional statements

```
if (IsSingleDigit (x))
{
    printf("x is little\n");
}
else
{
    printf("x is big\n");
}
```

5.8 Returning from `main()`

Now that we know functions that return values, we can look more closely at the return value of the `main()` function. It's supposed to return an integer:

```
int main (void)
```

The usual return value from `main()` is 0, which indicates that the program succeeded at whatever it was supposed to to. If something goes wrong, it is common to return -1, or some other value that indicates what kind of error occurred.

C provides two predefined constants `EXIT_SUCCESS` and `EXIT_FAILURE`

```
int main (void)
{
    return EXIT_SUCCESS;   /*program terminated successfully*/
}
```

Of course, you might wonder who this value gets returned to, since we never call `main()` ourselves. It turns out that when the operating system executes a program, it starts by calling `main()` in pretty much the same way it calls all the other functions.

There are even some parameters that can be passed to `main()` by the system, but we are not going to deal with them for a little while, so we define `main()` as having no parameters: `int main (void)`.

5.9 Glossary

return type: The type of value a function returns.

return value: The value provided as the result of a function call.

dead code: Part of a program that can never be executed, often because it appears after a `return` statement.

scaffolding: Code that is used during program development but is not part of the final version.

void: A special return type that indicates a void function; that is, one that does not return a value.

boolean: A value or variable that can take on one of two states, often called *true* and *false*. In C, boolean values are mainly stored in variables of type `short` and preprocessor statements are used to define the states.

flag: A variable that records a condition or status information.

comparison operator: An operator that compares two values and produces a boolean that indicates the relationship between the operands.

logical operator: An operator that combines boolean values in order to test compound conditions.

5.10 Exercises

Exercise 5.1 If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, it is clear that you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle:

> "If any of the three lengths is greater than the sum of the other two, then you cannot form a triangle. Otherwise, you can."

Write a function named `IsTriangle()` that it takes three integers as arguments, and that returns either `TRUE` or `FALSE`, depending on whether you can or cannot form a triangle from sticks with the given lengths.

The point of this exercise is to use conditional statements to write a function that returns a value.

Exercise 5.2 What is the output of the following program?

The purpose of this exercise is to make sure you understand logical operators and the flow of execution through fruitful methods.

```
#define TRUE 1
#define FALSE 0


  short IsHoopy (int x)
   {
      short hoopyFlag;
      if (x%2 == 0)
      {
          hoopyFlag = TRUE;
      }
      else
      {
          hoopyFlag = FALSE;
      }
      return hoopyFlag;
  }

  short IsFrabjuous (int x)
  {
      short frabjuousFlag;
      if (x > 0)
      {
          frabjuousFlag = TRUE;
      }
      else
      {
          frabjuousFlag = FALSE;
      }
      return frabjuousFlag;
  }

  int main (void)
  {
      short flag1 = IsHoopy (202);
      short flag2 = IsFrabjuous (202);
      printf ("%i\n", flag1);
      printf ("%i\n", flag2);
      if (flag1 && flag2)
      {
          printf ("ping!\n");
      }
      if (flag1 || flag2)
      {
          printf ("pong!\n");
      }
      return EXIT_SUCCESS;
  }
```

Exercise 5.3

a. Create a new program called `Sum.c`, and type in the following two functions.

```
int FunctionOne (int m, int n)
{
    if (m == n)
    {
        return n;
    }
```

```
        else
        {
            return m + FunctionOne (m+1, n);
        }
    }

    int FunctionTwo (int m, int n)
    {
        if (m == n)
        {
            return n;
        }
        else
        {
            return n * FunctionTwo (m, n-1);
        }
    }
```

b. Write a few lines in `main()` to test these functions. Invoke them a couple of times, with a few different values, and see what you get. By some combination of testing and examination of the code, figure out what these functions do, and give them more meaningful names. Add comments that describe their function abstractly.

c. Add a `prinf` statement to the beginning of both functions so that they print their arguments each time they are invoked. This is a useful technique for debugging recursive programs, since it demonstrates the flow of execution.

Exercise 5.4 Write a recursive function called `Power()` that takes a double `x` and an integer `n` and that returns x^n.

Hint: a recursive definition of this operation is `Power (x, n) = x * Power (x, n-1)`. Also, remember that anything raised to the zeroeth power is 1.

Exercise 5.5

(This exercise is based on page 44 of Ableson and Sussman's *Structure and Interpretation of Computer Programs.*)

The following algorithm is known as Euclid's Algorithm because it appears in Euclid's *Elements* (Book 7, ca. 300 B.C.). It may be the oldest nontrivial algorithm.

The algorithm is based on the observation that, if r is the remainder when a is divided by b, then the common divisors of a and b are the same as the common divisors of b and r. Thus we can use the equation

$$gcd(a, b) = gcd(b, r)$$

to successively reduce the problem of computing a GCD to the problem of computing the GCD of smaller and smaller pairs of integers. For example,

$$gcd(36, 20) = gcd(20, 16) = gcd(16, 4) = gcd(4, 0) = 4$$

implies that the GCD of 36 and 20 is 4. It can be shown that for any two starting numbers, this repeated reduction eventually produces a pair where the second number is 0. Then the GCD is the other number in the pair.

Write a function called `gcd` that takes two integer parameters and that uses Euclid's algorithm to compute and return the greatest common divisor of the two numbers.

Exercise 5.6 The distance between two points (x_1, y_1) and (x_2, y_2) is

$$Distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Please write a function named `Distance()` that takes four doubles as parameters—`x1`, `y1`, `x2` and `y2`—and that prints the distance between the points.

You should assume that there is a function named `SumSquares()` that calculates and returns the sum of the squares of its arguments. For example:

```
    double x = SumSquares (3.0, 4.0);
```

would assign the value `25.0` to `x`.

The point of this exercise is to write a new function that uses an existing one. You should write only one function: `Distance()`. You should not write `SumSquares()` or `main()` and you should not invoke `Distance()`.

Exercise 5.7 The point of this exercise is to practice the syntax of fruitful functions.

a. Use your existing solution to Exercise 3.4 and make sure you can still compile and run it.

b. Transform `Multadd()` into a fruitful function, so that instead of printing a result, it returns it.

c. Everywhere in the program that `Multadd()` gets invoked, change the invocation so that it stores the result in a variable and/or prints the result.

d. Transform `Yikes()` in the same way.

Chapter 6

Iteration

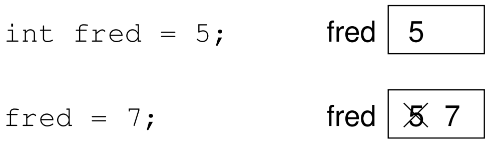
6.1 Multiple assignment

I haven't said much about it, but it is legal in C to make more than one assignment to the same variable. The effect of the second assignment is to replace the old value of the variable with a new value.

```
int fred = 5;
printf ("%i", fred);
fred = 7;
printf ("%i", fred);
```

The output of this program is `57`, because the first time we print `fred` his value is 5, and the second time his value is 7.

This kind of **multiple assignment** is the reason I described variables as a *container* for values. When you assign a value to a variable, you change the contents of the container, as shown in the figure:



When there are multiple assignments to a variable, it is especially important to distinguish between an assignment statement and a statement of equality. Because C uses the `=` symbol for assignment, it is tempting to interpret a statement like `a = b` as a statement of equality. It is not!

First of all, equality is commutative, and assignment is not. For example, in mathematics if $a = 7$ then $7 = a$. But in C the statement `a = 7;` is legal, and `7 = a;` is not.

Furthermore, in mathematics, a statement of equality is true for all time. If $a = b$ now, then a will always equal b. In C, an assignment statement can make two variables equal, but they don't have to stay that way!

```
int a = 5;
int b = a;     /* a and b are now equal */
a = 3;         /* a and b are no longer equal */
```

The third line changes the value of `a` but it does not change the value of `b`, and so they are no longer equal. In many programming languages an alternate symbol is used for assignment, such as `<-` or `:=`, in order to avoid confusion.

Although multiple assignment is frequently useful, you should use it with caution. If the values of variables are changing constantly in different parts of the program, it can make the code difficult to read and debug.

6.2 Iteration

One of the things computers are often used for is the automation of repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

In section 4.7 we have seen programs that use **recursion** to perform repetition, such as `PrintLines()` and `Countdown()`. I now want to introduce a new type of repetition, that is called **iteration**, and C provides several language features that make it easier to write repetetive programs.

The two features we are going to look at are the `while` statement and the `for` statement.

6.3 The `while` statement

Using a `while` statement, we can rewrite `Countdown()`:

```
void Countdown (int n)
{
    while (n > 0)
    {
        printf ("%i\n", n);
        n = n-1;
    }
    printf ("Blastoff!\n");
}
```

You can almost read a `while` statement as if it were English. What this means is, "While `n` is greater than zero, continue displaying the value of `n` and then reducing the value of `n` by 1. When you get to zero, output the word 'Blastoff!"'

More formally, the flow of execution for a `while` statement is as follows:

1. Evaluate the condition in parentheses, yielding `true` or `false`.
2. If the condition is false, exit the `while` statement and continue execution at the next statement.
3. If the condition is true, execute each of the statements between the curly-brackets, and then go back to step 1.

This type of flow is called a **loop** because the third step loops back around to the top. Notice that if the condition is false the first time through the loop, the statements inside the loop are never executed. The statements inside the loop are called the **body** of the loop.

The body of the loop should change the value of one or more variables so that, eventually, the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop.

In the case of `Countdown()`, we can prove that the loop will terminate because we know that the value of `n` is finite, and we can see that the value of `n` gets smaller each time through the loop (each **iteration**), so eventually we have to get to zero. In other cases it is not so easy to tell:

```
void Sequence (int n)
{
    while (n != 1)
    {
        printf ("%i\n", n);
        if (n%2 == 0)        /* n is even */
        {
            n = n / 2;
        }
        else                 /* n is odd */
        {
            n = n*3 + 1;
        }
    }
}
```

The condition for this loop is `n != 1`, so the loop will continue until `n` is 1, which will make the condition false.

At each iteration, the program outputs the value of `n` and then checks whether it is even or odd. If it is even, the value of `n` is divided by two. If it is odd, the value is replaced by $3n + 1$. For example, if the starting value (the argument passed to `Sequence`) is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1, or that the program will terminate. For some particular values of `n`, we can prove termination. For example, if the starting value is a power of two, then the value of n will be even every time through the loop, until we get to 1. The previous example ends with such a sequence, starting with 16.

Particular values aside, the interesting question is whether we can prove that this program terminates for *all* values of n. So far, no one has been able to prove it *or* disprove it!

6.4 Tables

One of the things loops are good for is generating tabular data. For example, before computers were readily available, people had to calculate logarithms, sines and cosines, and other common mathematical functions by hand. To make that easier, there were books containing long tables where you could find the values of various functions. Creating these tables was slow and boring, and the result tended to be full of errors.

When computers appeared on the scene, one of the initial reactions was, "This is great! We can use the computers to generate the tables, so there will be no errors." That turned out to be true (mostly), but shortsighted. Soon thereafter computers and calculators were so pervasive that the tables became obsolete.

Well, almost. It turns out that for some operations, computers use tables of values to get an approximate answer, and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the original Intel Pentium used to perform floating-point division.

Although a "log table" is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and their logarithms in the right column:

```
double x = 1.0;
while (x < 10.0)
{
    printf ("%.0f\t%f\n", x ,log(x));
    x = x + 1.0;
}
```

The sequence \t represents a **tab** character. The sequence \n represents a newline character. They are so called *escape sequences* which are used to encode non-printable ASCII-characters. Escape sequences can be included anywhere in a string, although in these examples the sequence is the whole string.

A tab character causes the cursor to shift to the right until it reaches one of the **tab stops**, which are normally every eight characters. As we will see in a minute, tabs are useful for making columns of text line up. A newline character causes the cursor to move on to the next line.

The output of this program is:

```
1       0.000000
2       0.693147
3       1.098612
4       1.386294
5       1.609438
6       1.791759
7       1.945910
8       2.079442
9       2.197225
```

If these values seem odd, remember that the `log()` function uses base e. Since powers of two are so important in computer science, we often want to find logarithms with respect to base 2. To do that, we can use the following formula:

$$\log_2 x = \frac{log_e x}{log_e 2}$$

Changing the output statement to

```
    printf ("%.0f\t%f\n", x, log(x) / log(2.0));
```

yields:

```
1       0.000000
2       1.000000
3       1.584963
4       2.000000
5       2.321928
6       2.584963
7       2.807355
8       3.000000
9       3.169925
```

We can see that 1, 2, 4 and 8 are powers of two, because their logarithms base 2 are round numbers. If we wanted to find the logarithms of other powers of two, we could modify the program like this:

```
double x = 1.0;
while (x < 100.0)
{
    printf ("%.0f\t%.0f\n", x, log(x) / log(2.0));
    x = x * 2.0;
}
```

Now instead of adding something to `x` each time through the loop, which yields an arithmetic sequence, we multiply `x` by something, yielding a **geometric** sequence. The result is:

```
1       0
2       1
4       2
8       3
16      4
32      5
64      6
```

Because we are using tab characters between the columns, the position of the second column does not depend on the number of digits in the first column.

Log tables may not be useful any more, but for computer scientists, knowing the powers of two is! As an exercise, modify this program so that it outputs the powers of two up to 65536 (that's 2^{16}). Print it out and memorize it.

6.5 Two-dimensional tables

A two-dimensional table is a table where you choose a row and a column and read the value at the intersection. A multiplication table is a good example. Let's say you wanted to print a multiplication table for the values from 1 to 6.

A good way to start is to write a simple loop that prints the multiples of 2, all on one line.

```
int i = 1;
while (i <= 6)
{
    printf("%i   ", i*2);
    i = i + 1;
}
printf("\n");
```

The first line initializes a variable named `i`, which is going to act as a counter, or **loop variable**. As the loop executes, the value of `i` increases from 1 to 6, and then when `i` is 7, the loop terminates. Each time through the loop, we print the value `2*i` followed by three spaces. By omitting the `\n` from the first output statement, we get all the output on a single line.

The output of this program is:

```
2   4   6   8   10   12
```

So far, so good. The next step is to **encapsulate** and **generalize**.

6.6 Encapsulation and generalization

Encapsulation usually means taking a piece of code and wrapping it up in a function, allowing you to take advantage of all the things functions are good for. We have seen two examples of encapsulation, when we wrote `PrintParity()` in Section 4.3 and `IsSingleDigit()` in Section 5.7.

Generalization means taking something specific, like printing multiples of 2, and making it more general, like printing the multiples of any integer.

Here's a function that encapsulates the loop from the previous section and generalizes it to print multiples of `n`.

```
void PrintMultiples (int n)
{
    int i = 1;
    while (i <= 6)
    {
        printf("%i   ", i*n);
        i = i + 1;
    }
    printf("\n");
}
```

To encapsulate, all I had to do was add the first line, which declares the name, parameter, and return type. To generalize, all I had to do was replace the value 2 with the parameter `n`.

If we call this function with the argument 2, we get the same output as before. With argument 3, the output is:

```
3   6   9   12   15   18
```

and with argument 4, the output is

```
4   8   12   16   20   24
```

By now you can probably guess how we are going to print a multiplication table: we'll call `PrintMultiples()` repeatedly with different arguments. In fact, we are going to use another loop to iterate through the rows.

```
int i = 1;
while (i <= 6)
{
    PrintMultiples (i);
    i = i + 1;
}
```

First of all, notice how similar this loop is to the one inside `PrintMultiples()`. I only replaced the call of the `printf()` function with the call of the `PrintMultiples()` function.

The output of this program is

```
1   2   3   4   5   6
2   4   6   8   10   12
3   6   9   12   15   18
4   8   12   16   20   24
5   10   15   20   25   30
6   12   18   24   30   36
```

which is a (slightly sloppy) multiplication table. If the sloppiness bothers you, try replacing the spaces between columns with tab characters and see what you get.

6.7 Functions

In the last section I mentioned “all the things functions are good for.” About this time, you might be wondering what exactly those things are. Here are some of the reasons functions are useful:

- By giving a name to a sequence of statements, you make your program easier to read and debug.
- Dividing a long program into functions allows you to separate parts of the program, debug them in isolation, and then compose them into a whole.
- Functions facilitate both recursion and iteration.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

6.8 More encapsulation

To demonstrate encapsulation again, I’ll take the code from the previous section and wrap it up in a function:

```
void PrintMultTable ()
{
    int i = 1;
    while (i <= 6)
    {
        PrintMultiples (i);
        i = i + 1;
    }
}
```

The process I am demonstrating is a common development plan. You develop code gradually by adding lines to `main()` or someplace else, and then when you get it working, you extract it and wrap it up in a function.

The reason this is useful is that you sometimes don’t know when you start writing exactly how to divide the program into functions. This approach lets you design as you go along.

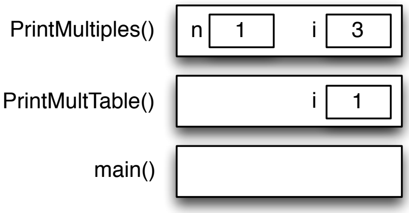
6.9 Local variables

About this time, you might be wondering how we can use the same variable `i` in both `PrintMultiples()` and `PrintMultTable()`. Didn’t I say that you can only declare a variable once? And doesn’t it cause problems when one of the functions changes the value of the variable?

The answer to both questions is “no,” because the `i` in `PrintMultiples()` and the `i` in `PrintMultTable()` are *not the same variable*. They have the same name, but they do not refer to the same storage location, and changing the value of one of them has no effect on the other.

Remember that variables that are declared inside a function definition are local. You cannot access a local variable from outside its "home" function, and you are free to have multiple variables with the same name, as long as they are not in the same function.

The stack diagram for this program shows clearly that the two variables named `i` are not in the same storage location. They can have different values, and changing one does not affect the other.



Notice that the value of the parameter `n` in `PrintMultiples()` has to be the same as the value of `i` in `PrintMultTable()`. On the other hand, the value of `i` in `PrintMultiples()` goes from 1 up to `6`. In the diagram, it happens to be 3. The next time through the loop it will be 4.

It is often a good idea to use different variable names in different functions, to avoid confusion, but there are good reasons to reuse names. For example, it is common to use the names `i`, `j` and `k` as loop variables. If you avoid using them in one function just because you used them somewhere else, you will probably make the program harder to read.

6.10 More generalization

As another example of generalization, imagine you wanted a program that would print a multiplication table of any size, not just the 6x6 table. You could add a parameter to `PrintMultTable()`:

```
void PrintMultTable (int high)
{
    int i = 1;
    while (i <= high)
    {
        PrintMultiples (i);
        i = i + 1;
    }
}
```

I replaced the value 6 with the parameter `high`. If I call `PrintMultTable()` with the argument 7, I get:

```
1   2   3   4   5   6
2   4   6   8   10   12
3   6   9   12   15   18
4   8   12   16   20   24
5   10   15   20   25   30
6   12   18   24   30   36
7   14   21   28   35   42
```

which is fine, except that I probably want the table to be square (same number of rows and columns), which means I have to add another parameter to `PrintMultiples()`, to specify how many columns the table should have.

Just to be annoying, I will also call this parameter `high`, demonstrating that different functions can have parameters with the same name (just like local variables):

```
void PrintMultiples (int n, int high)
{
    int i = 1;
    while (i <= high)
    {
        printf ("%i    ", n*i);
        i = i + 1;
    }
    printf ("\n");
}

void PrintMultTable (int high)
{
    int i = 1;
    while (i <= high)
    {
        PrintMultiples (i, high);
        i = i + 1;
    }
}
```

Notice that when I added a new parameter, I had to change the first line of the function, and I also had to change the place where the function is called in `PrintMultTable()`. As expected, this program generates a square 7x7 table:

```
1   2   3   4   5   6   7
2   4   6   8   10   12   14
3   6   9   12   15   18   21
4   8   12   16   20   24   28
5   10   15   20   25   30   35
6   12   18   24   30   36   42
7   14   21   28   35   42   49
```

When you generalize a function appropriately, you often find that the resulting program has capabilities you did not intend. For example, you might notice that the multiplication table is symmetric, because $ab = ba$, so all the entries in the table appear twice. You could save ink by printing only half the table. To do that, you only have to change one line of `PrintMultTable()`. Change

```
      PrintMultiples (i, high);
```

to

```
      PrintMultiples (i, i);
```

and you get:

```
1
2   4
3   6   9
4   8   12   16
5   10   15   20   25
6   12   18   24   30   36
7   14   21   28   35   42   49
```

I'll leave it up to you to figure out how it works.

6.11 Glossary

loop: A statement that executes repeatedly while a condition is true or until some condition is satisfied.

infinite loop: A loop whose condition is always true.

body: The statements inside the loop.

iteration: One pass through (execution of) the body of the loop, including the evaluation of the condition.

tab: A special character, written as `\t` in C, that causes the cursor to move to the next tab stop on the current line.

encapsulate: To divide a large complex program into components (like functions) and isolate the components from each other (for example, by using local variables).

local variable: A variable that is declared inside a function and that exists only within that function. Local variables cannot be accessed from outside their home function, and do not interfere with any other functions.

generalize: To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

development plan: A process for developing a program. In this chapter, I demonstrated a style of development based on developing code to do simple, specific things, and then encapsulating and generalizing.

6.12 Exercises

Exercise 6.1

```
void Loop(int n)
{
    int i = n;
    while (i > 1)
    {
        printf ("%i\n",i);
        if (i%2 == 0)
        {
            i = i/2;
        }
        else
        {
            i = i+1;
        }
    }
}

int main (void)
{
    Loop(10);
    return EXIT_SUCCESS;
}
```

a. Draw a table that shows the value of the variables `i` and `n` during the execution of the program. The table should contain one column for each variable and one line for each iteration.

b. What is the output of this program?

Exercise 6.2 In Exercise 5.4 we wrote a recursive version of `Power()`, which takes a double `x` and an integer `n` and returns x^n. Now write an iterative function to perform the same calculation.

Exercise 6.3 Let's say you are given a number, a, and you want to find its square root. One way to do that is to start with a very rough guess about the answer, x_0, and then improve the guess using the following formula:

$$x_1 = (x_0 + a/x_0)/2$$

For example, if we want to find the square root of 9, and we start with $x_0 = 6$, then $x_1 = (6 + 9/6)/2 = 15/4 = 3.75$, which is closer.

We can repeat the procedure, using x_1 to calculate x_2, and so on. In this case, $x_2 = 3.075$ and $x_3 = 3.00091$. So that is converging very quickly on the right answer (which is 3).

Write a function called `SquareRoot` that takes a `double` as a parameter and that returns an approximation of the square root of the parameter, using this algorithm. You may not use the `sqrt()` function from the `math.h` library.

As your initial guess, you should use $a/2$. Your function should iterate until it gets two consecutive estimates that differ by less than 0.0001; in other words, until the absolute value of $x_n - x_{n-1}$ is less than 0.0001. You can use the built-in `fabs()` function from the `math.h` library to calculate the absolute value.

Exercise 6.4 One way to evaluate e^{-x^2} is to use the infinite series expansion

$$e^{-x^2} = 1 - 2x + 3x^2/2! - 4x^3/3! + 5x^4/4! - ...$$

In other words, we need to add up a series of terms where the ith term is equal to $(-1)^i(i+1)x^i/i!$. Write a function named `Gauss()` that takes `x` and `n` as arguments and that returns the sum of the first `n` terms of the series. You should not use `factorial()` or `pow()`.

Chapter 7

Arrays

A **array** is a set of values where each value is identified and referenced by a number (called an index). The nice thing about arrays is that they can be made up of any type of element, including basic types like `int`s and `double`s, but all the values in an array have to have the same type.

When you declare an array, you have to determine the number of elements in the array. Otherwise the declaration looks similar to other variable types:

```
    int c[4];
    double values[10];
```

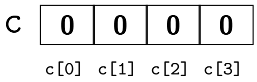
Syntactically, array variables look like other C variables except that they are followed by `[NUMBER_OF_ELEMENTS]`, the number of elements in the array enclosed in square brackets. The first line in our example, `int c[4];` is of the type "array of integers" and creates a array of four integers named `c`. The second line, `double values[10];` has the type "array of doubles" and creates an array of 10 `double`s.

C allows you to to initialize the element values of an array immediately after you have declared it. The values for the individual elements must be enclosed in curly brakets `{}` and separated by comma, as in the following example:

```
    int c[4] = {0, 0, 0, 0};
```

This statement creates an array of four elements and initializes all of them to zero. This syntax is only legal at initialisation time. Later in your program you can only assign values for the array element by element.

The following figure shows how arrays are represented in state diagrams:



The large numbers inside the boxes are the values of the **elements** in the array. The small numbers outside the boxes are the indices used to identify each box. When you allocate a new array, without initializing, the arrays elements typically contain arbitrary values and you must initialise them to a meaningful value before using them.

7.1 Increment and decrement operators

Incrementing and decrementing are such common operations that C provides special operators for them. The `++` operator adds one to the current value of an `int`, `char` or `double`, and `--` subtracts one.

Technically, it is legal to increment a variable and use it in an expression at the same time. For example, you might see something like:

```
printf ("%i\n ", i++);
```

Looking at this, it is not clear whether the increment will take effect before or after the value is displayed. Because expressions like this tend to be confusing, I would discourage you from using them. In fact, to discourage you even more, I'm not going to tell you what the result is. If you really want to know, you can try it.

Using the increment operators, we can rewrite the `PrintMultTable()` from Section 6.10:

```
void PrintMultTable(int high)
{
    int i = 1;
    while (i <= high)
    {
        PrintMultiples(i);
        i++;
    }
}
```

It is a common error to write something like:

```
index = index++;             /* WRONG!! */
```

Unfortunately, this is syntactically legal, so the compiler will not warn you. The effect of this statement is to leave the value of `index` unchanged. This is often a difficult bug to track down.

Remember, you can write `index = index + 1;`, or you can write `index++;`, but you shouldn't mix them.

7.2 Accessing elements

The `[]` operator allows us to read and write the individual elements of an array. The indices start at zero, so `c[0]` refers to the first element of the array, and `c[1]` refers to the second element. You can use the `[]` operator anywhere in an expression:

```
c[0] = 7;
c[1] = c[0] * 2;
c[2]++;
c[3] -= 60;
```

All of these are legal assignment statements. Here is the effect of this code fragment:

c	7	14	1	-60
	c[0]	c[1]	c[2]	c[3]

By now you should have noticed that the four elements of this array are numbered from 0 to 3, which means that there is no element with the index 4.

Nevertheless, it is a common error to go beyond the bounds of an array. In safer languages such as Java, this will cause an error and most likely the program quits. C does not check array boundaries, so your program can go on accessing memory locations beyond the array itself, as if they where part of the array. This is most likely wrong and can cause very severe bugs in your program.

It is necessary that you, as a programmer, make sure that your code correctly observes array boundaries!

You can use any expression as an index, as long as it has type `int`. One of the most common ways to index an array is with a loop variable. For example:

```
int i = 0;
while (i < 4)
{
    printf ("%i\n", c[i]);
    i++;
}
```

This is a standard `while` loop that counts from 0 up to 4, and when the loop variable `i` is 4, the condition fails and the loop terminates. Thus, the body of the loop is only executed when `i` is 0, 1, 2 and 3.

Each time through the loop we use `i` as an index into the array, printing the `i`th element. This type of array traversal is very common. Arrays and loops go together like fava beans and a nice Chianti.

7.3 Copying arrays

Arrays can be a very convenient solution for a number of problems, like storing and processing large sets of data.

However, there is very little that C does automatically for you. For example you can not set all the elements of an array at the same time and you can not assign one array to the other, even if they are identical in type and number of elements.

```
double a[3] = {1.0, 1.0, 1.0};
double b[3];

a = 0.0;      /* Wrong! */
b = a;        /* Wrong! */
```

In order to set all of the elements of an array to some value, you must do so element by element. To copy the contents of one array to another, you must again do so, by copying each element from one array to the other.

```
int i = 0;
while (i < 3)
{
    b[i] = a[i];
    i++;
}
```

7.4 `for` loops

The loops we have written so far have a number of elements in common. All of them start by initializing a variable; they have a test, or condition, that depends on that variable; and inside the loop they do something to that variable, like increment it.

This type of loop is so common that there is an alternate loop statement, called `for`, that expresses it more concisely. The general syntax looks like this:

```
for (INITIALIZER; CONDITION; INCREMENTOR)
{
    BODY
}
```

This statement is exactly equivalent to

```
INITIALIZER;
while (CONDITION)
{
    BODY
    INCREMENTOR
}
```

except that it is more concise and, since it puts all the loop-related statements in one place, it is easier to read. For example:

```
int i;
for (i = 0; i < 4; i++)
{
    printf("%i\n", c[i]);
}
```

is equivalent to

```
int i = 0;
while (i < 4)
{
    printf("%i\n", c[i]);
    i++;
}
```

7.5 Array length

C does not provide us with a convenient way to determine the actual length of an array. Knowing the size of an array would be convenient when we are looping through all elements of the array and need to stop with the last element.

In order to determine the array length we could use the `sizeof()` operator, that calculates the size of data types in bytes. Most data types in C use more than one byte to store their values, therefore it becomes necessary to divide the byte-count for the array by the byte-count for a single element to establish the number of elements in the array.

```
sizeof(ARRAY)/sizeof(ARRAY_ELEMENT)
```

It is a good idea to use this value as the upper bound of a loop, rather than a constant. That way, if the size of the array changes, you won't have to go through the program changing all the loops; they will work correctly for any size array.

```
int i, length;
length = sizeof (c) / sizeof (c[0]);

for (i = 0; i < length; i++)
{
    printf("%i\n", c[i]);
}
```

The last time the body of the loop gets executed, the value of `i` is `length - 1`, which is the index of the last element. When `i` is equal to `length`, the condition fails and the body is not executed, which is a good thing, since it would access a memory location that is not part of the array.

7.6 Random numbers

Most computer programs do the same thing every time they are executed, so they are said to be **deterministic**. Usually, determinism is a good thing, since we expect the same calculation to yield the same result. For some applications, though, we would like the computer to be unpredictable. Games are an obvious example.

Making a program truly **nondeterministic** turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to generate pseudorandom numbers and use them to determine the outcome of the program. Pseudorandom numbers are not truly random in the mathematical sense, but for our purposes, they will do.

C provides a function called `rand()` that generates pseudorandom numbers. It is declared in the header file `stdlib.h`, which contains a variety of "standard library" functions, hence the name.

The return value from `rand()` is an integer between 0 and `RAND_MAX`, where `RAND_MAX` is a large number (about 2 billion on my computer) also defined in the header file. Each time you call `rand()` you get a different randomly-generated number. To see a sample, run this loop:

```
    for (i = 0; i < 4; i++)
    {
        int x = rand();
        printf("%i\n", x);
    }
```

On my machine I got the following output:

```
1804289383
846930886
1681692777
1714636915
```

You will probably get something similar, but different, on yours.

Of course, we don't always want to work with gigantic integers. More often we want to generate integers between 0 and some upper bound. A simple way to do that is with the modulus operator. For example:

```
    int x = rand ();
    int y = x % upperBound;
```

Since `y` is the remainder when `x` is divided by `upperBound`, the only possible values for `y` are between 0 and `upperBound - 1`, including both end points. Keep in mind, though, that `y` will never be equal to `upperBound`.

It is also frequently useful to generate random floating-point values. A common way to do that is by dividing by `RAND_MAX`. For example:

```
    int x = rand ();
    double y = (double) x / RAND_MAX;
```

This code sets `y` to a random value between 0.0 and 1.0, including both end points. As an exercise, you might want to think about how to generate a random floating-point value in a given range; for example, between 100.0 and 200.0.

7.7 Statistics

The numbers generated by `rand()` are supposed to be distributed uniformly. That means that each value in the range should be equally likely. If we count the number of times each value appears, it should be roughly the same for all values, provided that we generate a large number of values.

In the next few sections, we will write programs that generate a sequence of random numbers and check whether this property holds true.

7.8 Array of random numbers

The first step is to generate a large number of random values and store them in a array. By “large number,” of course, I mean 20. It’s always a good idea to start with a manageable number, to help with debugging, and then increase it later.

The following function takes three arguments, an array of integers, the size of the array and an upper bound for the random values. It fills the array of `ints` with random values between 0 and `upperBound-1`.

```
void RandomizeArray (int array[], int length, int upperBound)
{
    int i;
    for (i = 0; i < length; i++)
    {
        array[i] = rand() % upperBound;
    }
}
```

The return type is `void`, which means that this function does not return any value to the calling function. To test this function, it is convenient to have a function that outputs the contents of a array.

```
void PrintArray (int array[], int length)
{
    int i;
    for (i = 0; i < length; i++)
    {
        printf ("%i ",  array[i]);
    }
}
```

The following code generates an array filled with random values and outputs it:

```
int r_array[20];
int upperBound = 10;
int length = sizeof(r_array) / sizeof(r_array[0]);

RandomizeArray (r_array, length, upperBound);
PrintArray (r_array, length);
```

On my machine the output is:

```
3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6
```

which is pretty random-looking. Your results may differ.

If these numbers are really random, we expect each digit to appear the same number of times—twice each. In fact, the number 6 appears five times, and the numbers 4 and 8 never appear at all.

Do these results mean the values are not really uniform? It's hard to tell. With so few values, the chances are slim that we would get exactly what we expect. But as the number of values increases, the outcome should be more predictable.

To test this theory, we'll write some programs that count the number of times each value appears, and then see what happens when we increase the number of elements in our array.

7.9 Passing an array to a function

You probably have noticed that our `RandomizeArray()` function looked a bit unusual. We pass an array to this function and expect to get a a randomized array back. Nevertheless, we have declared it to be a `void` function, and miraculously the function appears to have altered the array.

This behaviour goes against everything what I have said about the use of variables in functions so far. C typically uses the so called **call-by-value** evaluation of expressions. If you pass a value to a function it gets copied from the calling function to a variable in the called function. The same is true if the function returns a value. Changes to the internal variable in the called function do not affect the external values of the calling function.

When we pass an array to a function this behaviour changes to something called **call-by-reference** evaluation. C does not copy the array to an internal array – it rather generates a reference to the original array and any operation in the called function directly affects the original array. This is also the reason why we do not have to return anything from our function. The changes have already taken place.

Call by reference also makes it necessary to supply the length of the array to the called function, since invoking the `sizeof` operator in the called function would determine the size of the reference and not the original array.

We will further discuss call by reference and call by value in Section 8.7, Section 9.6 and 9.7.

7.10 Counting

A good approach to problems like this is to think of simple functions that are easy to write, and that might turn out to be useful. Then you can combine them into a solution. This approach is sometimes called **bottom-up design**.

Of course, it is not easy to know ahead of time which functions are likely to be useful, but as you gain experience you will have a better idea. Also, it is not always obvious what sort of things are easy to write, but a good approach is to look for subproblems that fit a pattern you have seen before.

In our current example we want to examine a potentially large set of elements and count the number of times a certain value appears. You can think of this program as an example of a pattern called "traverse and count." The elements of this pattern are:

- A set or container that can be traversed, like a string or a array.
- A test that you can apply to each element in the container.
- A counter that keeps track of how many elements pass the test.

In this case, I have a function in mind called `HowMany()` that counts the number of elements in a array that are equal to a given value. The parameters are the array, the length of the array and the integer value we are looking for. The return value is the number of times the value appears.

```
int HowMany (int array[], int length, int value)
{
    int i;
    int count = 0;

    for (i=0; i < length; i++)
        {
            if (array[i] == value) count++;
        }
    return count;
}
```

7.11 Checking the other values

`HowMany()` only counts the occurrences of a particular value, and we are interested in seeing how many times each value appears. We can solve that problem with a loop:

```
int i;
int r_array[20];
int upperBound = 10;
int length = sizeof(r_array) / sizeof(r_array[0]);

RandomizeArray(r_array, length, upperBound);

printf ("value\tHowMany\n");
for (i = 0; i < upperBound; i++)
{
    printf("%i\t%i\n", i, HowMany(r_array, length, i));
}
```

This code uses the loop variable as an argument to `HowMany()`, in order to check each value between 0 and 9, in order. The result is:

```
value   HowMany
0       2
1       1
2       3
3       3
4       0
5       2
6       5
7       2
8       0
9       2
```

Again, it is hard to tell if the digits are really appearing equally often. If we increase the size of the array to 100,000 we get the following:

```
value   HowMany
0       10130
1       10072
2       9990
3       9842
4       10174
5       9930
6       10059
7       9954
8       9891
9       9958
```

In each case, the number of appearances is within about 1% of the expected value (10,000), so we conclude that the random numbers are probably uniform.

7.12 A histogram

It is often useful to take the data from the previous tables and store them for later access, rather than just print them. What we need is a way to store 10 integers. We could create 10 integer variables with names like `howManyOnes`, `howManyTwos`, etc. But that would require a lot of typing, and it would be a real pain later if we decided to change the range of values.

A better solution is to use a array with length 10. That way we can create all ten storage locations at once and we can access them using indices, rather than ten different names. Here's how:

```
    int i;
    int upperBound = 10;
    int r_array[100000];
    int histogram[upperBound];
    int r_array_length = sizeof(r_array) / sizeof(r_array[0]);

    RandomizeArray(r_array, r_array_length, upperBound);

    for (i = 0; i < upperBound; i++)
    {
        int count = HowMany(r_array, length, i);
        histogram[i] = count;
    }
```

I called the array **histogram** because that's a statistical term for a array of numbers that counts the number of appearances of a range of values.

The tricky thing here is that I am using the loop variable in two different ways. First, it is an argument to `HowMany()`, specifying which value I am interested in. Second, it is an index into the histogram, specifying which location I should store the result in.

7.13 A single-pass solution

Although this code works, it is not as efficient as it could be. Every time it calls `HowMany()`, it traverses the entire array. In this example we have to traverse the array ten times!

It would be better to make a single pass through the array. For each value in the array we could find the corresponding counter and increment it. In other words, we can use the value from the array as an index into the histogram. Here's what that looks like:

```
    int upperBound = 10;
    int histogram[upperBound] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

    for (i = 0; i < r_array_length; i++)
    {
        int index = r_array[i];
        histogram[index]++;
    }
```

The second line initializes the elements of the histogram to zeroes. That way, when we use the increment operator (`++`) inside the loop, we know we are starting from zero. Forgetting to initialize counters is a common error.

As an exercise, encapsulate this code in a function called `Histogram()` that takes an array and the range of values in the array (in this case 0 through 10) as two parameters `min` and `max`. You should pass a second array to the function where a histogram of the values in the array can be stored.

7.14 Random seeds

If you have run the code in this chapter a few times, you might have noticed that you are getting the same "random" values every time. That's not very random!

One of the properties of pseudorandom number generators is that if they start from the same place they will generate the same sequence of values. The starting place is called a **seed**; by default, C uses the same seed every time you run the program.

While you are debugging, it is often helpful to see the same sequence over and over. That way, when you make a change to the program you can compare the output before and after the change.

If you want to choose a different seed for the random number generator, you can use the `srand()` function. It takes a single argument, which is an integer between 0 and `RAND_MAX`.

For many applications, like games, you want to see a different random sequence every time the program runs. A common way to do that is to use a library function like `time()` to generate something reasonably unpredictable and unrepeatable, like the number of seconds since January 1970, and use that number as a seed. The details of how to do that depend on your development environment.

7.15 Glossary

array: A named collection of values, where all the values have the same type, and each value is identified by an index.

element: One of the values in a array. The `[]` operator selects elements of a array.

index: An integer variable or value used to indicate an element of a array.

increment: Increase the value of a variable by one. The increment operator in C is `++`.

decrement: Decrease the value of a variable by one. The decrement operator in C is `--`.

deterministic: A program that does the same thing every time it is run.

pseudorandom: A sequence of numbers that appear to be random, but which are actually the product of a deterministic computation.

seed: A value used to initialize a random number sequence. Using the same seed should yield the same sequence of values.

bottom-up design: A method of program development that starts by writing small, useful functions and then assembling them into larger solutions.

histogram: A array of integers where each integer counts the number of values that fall into a certain range.

7.16 Exercises

Exercise 7.1

A friend of yours shows you the following method and explains that if `number` is any two-digit number, the program will output the number backwards. He claims that if `number` is 17, the method will output `71`.

Is he right? If not, explain what the program actually does and modify it so that it does the right thing.

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

 int main (void)
 {
   int number = 71;
   int lastDigit = number%10;
   int firstDigit = number/10;
   printf("%i",lastDigit + firstDigit);
   return EXIT_SUCCESS;
}
```

Exercise 7.2 Write a function that takes an array of integers, the length of the array and an integer named `target` as arguments.

The function should search through the provided array and should return the first index where `target` appears in the array, if it does. If `target` is not in the array the function should return an invalid index value to indicate an error condition (e.g. -1).

Exercise 7.3

One not-very-efficient way to sort the elements of an array is to find the largest element and swap it with the first element, then find the second-largest element and swap it with the second, and so on.

a. Write a function called `IndexOfMaxInRange()` that takes an array of integers, finds the largest element in the given range, and returns *its index.*

b. Write a function called `SwapElement()` that takes an array of integers and two indices, and that swaps the elements at the given indices.

c. Write a function called `SortArray()` that takes an array of integers and that uses `IndexOfMaxInRange()` and `SwapElement()` to sort the array from largest to smallest.

Chapter 8

Strings and things

8.1 Containers for strings

We have seen four types of values—characters, integers, floating-point numbers and strings—but only three types of variables—`char`, `int` and `double`. So far we have no way to store a string in a variable or perform operations on strings.

This chapter is going to rectify this situation and I can now tell you that strings in C are stored as an array of characters terminated by the character `\0`.

By now this explanation should make sense to you and you probably understand why we had to learn quite a bit about the working of the language before we could turn our attention towards string variables.

In the previous chapter we have seen that operations on arrays have only minimal support from the C language itself and we had to program extra functions by ourselves. Fortunately things are a little bit easier when we manipulate these special types of arrays - called strings. There exist a number of library functions in `string.h` that make string handling a bit easier than operations on pure arrays.

Nevertheless string operations in C are still a lot more cumbersome than their equivalence in other programing languages and can be a potential source of errors in your programs, if not handled carefully.

8.2 String variables

You can create a string variable as an array of characters in the following way:

```
char first[] = "Hello, ";
char second[] = "world.";
```

The first line creates an `string` and assigns it the string value `"Hello."` In the second line we declare a second string variable. Remember, the combined declaration and assignment is called initialization.

Initialisation time is the only time you can assign a value to a string directly (just as with arrays in general). The initialisation parameters are passed in the form of a string constant enclosed in quotation marks ("...").

Notice the difference in syntax for the initialisation of arrays and strings. If you like you can also initialize the string in the normal array syntax, although this looks a little odd and is not very convenient to type.

```
char first[] = {’H’,’e’,’l’,’l’,’o’,’,’,’ ’,’\0’};
```

There is no need to supply an array size when you are initialising the string variable at declaration time. The compiler compute the necessary array size to store the supplied string.

Remember what we said about the nature of a string variable. It is an array of characters **plus** a marker that shows where our string ends: the termination character `\0`.

Normally you do not have to supply this termination character. The compiler understands our code and insertes it automatically. However, in the example above, we treated our string exactly like an array and in this case we have to insert the termination character ourselves.

When we are using a string variable to store different sting values during the lifetime of our program we have to declare a size big enough for the largest sequence of characters that we are going to store. We also have to make our string variable exactly one character longer than the text we are going to store, because of the necessary termination character.

We can output strings in the usual way using the `printf()` function:

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

8.3 Extracting characters from a string

Strings are called “strings” because they are made up of a sequence, or string, of characters. The first operation we are going to perform on a string is to extract one of the characters. C uses an index in square brackets (`[` and `]`) for this operation:

```
char fruit[] = "banana";
char letter = fruit[1];
printf ("%c\n", letter);
```

The expression `fruit[1]` indicates that I want character number 1 from the string named `fruit`. The result is stored in a `char` named `letter`. When I output the value of `letter`, I get a surprise:

```
a
```

a is not the first letter of "banana". Unless you are a computer scientist. For perverse reasons, computer scientists always start counting from zero. The 0th letter ("zeroeth") of "banana" is b. The 1th letter ("oneth") is a and the 2th ("twoeth") letter is n.

If you want the zereoth letter of a string, you have to put zero in the square brackets:

```
char letter = fruit[0];
```

8.4 Length

To find the length of a string (the number of characters this string contains), we can use the strlen() function. The function is called using the string variable as an argument:

```
#include <string.h>
int main(void)
{
   int length;
   char fruit[] = "banana";

   length = strlen(fruit);
   return EXIT_SUCCESS;
}
```

The return value of strlen() in this case is 6. We assign this value to the integer length for further use.

In order to compile this code, you need to include the header file for the string.h library. This library provides a number of useful functions for operations on strings. You should familiarize yourself with these functions because they can help you to solve your programming problems faster.

To find the last letter of a string, you might be tempted to try something like

```
int length = strlen(fruit);
char last = fruit[length];       /* WRONG!! */
```

That won't work. The reason is that fruit is still an array and there is no letter at the array index fruit[6] in "banana". Since we started counting at 0, the 6 letters are numbered from 0 to 5. To get the last character, you have to subtract 1 from length.

```
int length = strlen(fruit);
char last = fruit[length-1];
```

8.5 Traversal

A common thing to do with a string is start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. A natural way to encode a traversal is with a `while` statement:

```
int index = 0;
while (index < strlen(fruit))
{
    char letter = fruit[index];
    printf("%c\n" , letter);
    index = index + 1;
}
```

This loop traverses the string and outputs each letter on a line by itself. Notice that the condition is `index < strlen(fruit)`, which means that when `index` is equal to the length of the string, the condition is false and the body of the loop is not executed. The last character we access is the one with the index `strlen(fruit)-1`.

The name of the loop variable is `index`. An **index** is a variable or value used to specify one member of an ordered set, in this case the set of characters in the string. The index indicates (hence the name) which one you want. The set has to be ordered so that each letter has an index and each index refers to a single character.

As an exercise, write a function that takes a `string` as an argument and that outputs the letters backwards, all on one line.

8.6 Finding a `character` in a `string`

If we are looking for a letter in a `string`, we have to search through the string and detect the position where this letter occurs in the string. Here is an implementation of this function:

```
int LocateCharacter(char *s, char c)
{
    int i = 0;
    while (i < strlen(s))
    {
        if (s[i] == c) return i;
        i = i + 1;
    }
    return -1;
}
```

We have to pass the `string` as the first argument, the other argument is the character we are looking for. Our function returns the index of the first occurrence of the letter, or `-1` if the letter is not contained in the string.

8.7 Pointers and Addresses

When we look at the definition of the `LocateCharacter()` function you may notice the following construct `char *s` which looks unfamiliar.

Remember, when we discussed how we had to pass an array to a function, back in Section 7.9, we said that instead of copying the array, we only pass a reference to the function. Back then, we did not say exactly what this reference was.

C is one of the very few high-level programming languages that let you directly manipulate objects in the computer memory. In order to do this direct manipulation, we need to know the location of the object in memory: it's address. Adresses can be stored in variables of a special type. These variables that point to other objects in memory (such as variables, arrays and strings) are therefore called **pointer** variables.

A pointer references the memory location of an object and can be defined like this:

```
int *i_p;
```

This declaration looks similar to our earlier declarations, with one difference: the asterisk in front of the name. We have given this pointer the type `int`. The type specification has nothing to do with the pointer itself, but rather defines which object this pointer is supposed to reference (in this case an `integer`). This allows the compiler to do some type checking on, what would otherwise be, an anonymous reference.

A pointer all by itself is rather meaningless, we also need an object that this pointer is referencing:

```
int number = 5;
int *i_p;
```

This code-fragment defines an `int` variable and a pointer. We can use the "address-of" operator `&` to assign the memory location or **address** of our variable to the pointer.

```
i_p = &number;
```

Pointer `i_p` now references integer variable `number`. We can verify this using the "content-of" operator `*`.

```
printf("%i\n", *i_p);
```

This prints `5`, which happens to be the content of the memory location at our pointer reference.

With pointers we can directly manipulate memory locations:

```
*i_p = *i_p + 2;
printf("%i\n", number);
```

Our variable `number` now has the value `7` and we begin to understand how our `LocateCharacter()` function can directly access the values of string variables through the use of a `char` pointer.

Pointers are widely used in many C programs and we have only touched the surface of the topic. They can be immensely useful and efficient, however they can also be a potential source of problems when not used appropriately. For this reason not many programming languages support direct memory manipulation.

8.8 String concatenation

In Section 8.6 we have seen how we could implement a search function that finds a `character` in a `string`.

One useful operation on strings is string **concatenation**. To concatenate means to join the two operands end to end. For example: `shoe` and `maker` becomes `shoemaker`.

Fortunately, we do not have to program all the necessary functions in C ourselves. The `string.h` library already provides several functions that we can invoke on strings.

We can use the library function `strncat()` to concatenate strings in C.

```
    char fruit[20] = "banana";
    char bakedGood[] = " nut bread";
    strncat(fruit, bakedGood, 10);
    printf ("%s\n", fruit);
```

The output of this program is `banana nut bread`.

When we are using library functions it is important to completely understand all the necessary arguments and to have a complete understanding of the working of the function.

The `strncat()` does not take the two strings, joins them together and produces a new combined string. It rather copies the content from the second argument into the first.

We therefore have to make sure that our first string is long enough to also hold the second string. We do this by defining the maximum capacity for string `fruit` to be 19 characters + 1 termination character (`char fruit[20]`). The third argument of `strncat()` specifies the number of characters that will be copied from the second into the first string.

8.9 Assigning new values to `string` variables

So far we have seen how to initialise a string variable at declaration time. As with arrays in general, it is not legal to assign values directly to strings, because it is not possible to assign a value to an entire array.

```
    fruit = "orange";  /* Wrong: Cannot assign directly! */
```

In order to assign a new value to an existing string variable we have to use the `strncpy()` function. For example,

```
char greeting[15];
strncpy (greeting, "Hello, world!", 13);
```

copies 13 characters from the of the second argument string to the first argument string.

This works, but not quite as expected. The `strncpy()` function copies exactly 13 characters from the second argument string into the first argument string. And what happens to our string termination character `\0`?

It is **not** copied automatically. We need to change our copy statement to copy also the invisible 14th character at the end of the string:

```
strncpy (greeting, "Hello, world!", 14);
```

However, if we only copy parts of the second string into the first we need to explicitly set the n+1th character in the `greeting[15]` string to `\0` afterwards.

```
strncpy (greeting, "Hello, world!", 5); /*only Hello is copied*/
greeting[5] = '\0';
```

Attention! In the last two sections we have used the `strncpy()` and the `strncat()` function that require you to explicitly supply the number of characters that will get copied or attached to the first argument string.

The `string.h` library also defines the `strcpy()` and the `strcat()` functions that have no explicit bound on the number of characters that are copied.

The usage of these functions is strongly discouraged! Their use has lead to a vast number of security problems with C programs. Remember, C does not check array boundaries and will continue copying characters into computer memory even past the length of the variable.

8.10 strings are not comparable

All the comparison operators that work on `ints` and `doubles` do work on `strings`. For example, if you write the following code to determine if two strings are equal:

```
if (word == "banana")  /* Wrong! */
```

This test will always fail.

You have to use the `strcmp()` function to compare two strings with each other. The function returns `0` if the two strings are identical, a negative value if the first string is 'alphabetically less' than the second (would be listed first in a dictionary) or a positive value if the second string is 'greater'.

Please notice, this return value is not the standard true/false result, where the return value `0` is interpreted as 'false'.

The `strcmp()` function is useful for putting words in alphabetical order.

```
if (strcmp(word, "banana") < 0)
{
    printf( "Your word, %s, comes before banana.\n", word);
}
else if (strcmp(word, "banana") > 0)
{
    printf( "Your word, %s, comes after banana.\n", word);
}
else
{
    printf ("Yes, we have no bananas!\n");
}
```

You should be aware, though, that the `strcmp()` function does not handle upper and lower case letters the same way that people do. All the upper case letters come before all the lower case letters. As a result,

```
Your word, Zebra, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, like all lower-case, before performing the comparison. The next sections explains how.

8.11 Character classification

It is often useful to examine a character and test whether it is upper or lower case, or whether it is a character or a digit. C provides a library of functions that perform this kind of character classification. In order to use these functions, you have to include the header file `ctype.h`.

```
char letter = 'a';
if (isalpha(letter))
{
    printf("The character %c is a letter.", letter);
}
```

The return value from `isalpha()` is an integer that is 0 if the argument is not a letter, and some non-zero value if it is.

It is legal to use this kind of integer in a conditional, as shown in the example. The value `0` is treated as `false`, and all non-zero values are treated as `true`.

Other character classification functions include `isdigit()`, which identifies the digits 0 through 9, and `isspace()`, which identifies all kinds of “white” space, including spaces, tabs, newlines, and a few others. There are also `isupper()` and `islower()`, which distinguish upper and lower case letters.

Finally, there are two functions that convert letters from one case to the other, called `toupper()` and `tolower()`. Both take a single character as an argument and return a (possibly converted) character.

```
char letter = 'a';
letter = toupper (letter);
printf("%c\n", letter);
```

The output of this code is A.

As an exercise, use the character classification and conversion library to write functions named StringToUpper() and StringToLower() that take a single string as a parameter, and that modify the string by converting all the letters to upper or lower case. The return type should be void.

8.12 Getting user input

The programs we have written so far are pretty predictable; they do the same thing every time they run. Most of the time, though, we want programs that take input from the user and respond accordingly.

There are many ways to get input, including keyboard input, mouse movements and button clicks, as well as more exotic mechanisms like voice control and retinal scanning. In this text we will consider only keyboard input.

In the header file stdio.h, C defines a function named scanf() that handles input in much the same way that printf() handles output. We can use the following code to get an integer value from the user:

```
int x;
scanf("%i", &x);
```

The scanf() function causes the program to stop executing and wait for the user to type something. If the user types a valid integer, the program converts it into an integer value and stores it in x.

If the user types something other than an integer, C doesn't report an error, or anything sensible like that. Instead, the scanf() function returns and leaves the value in x unchanged.

Fortunately, there is a way to check and see if an input statement succeeds. The scanf() function returns the number of items that have been successfully read. This number will be 1 when the last input statement succeeded. If not, we know that some previous operation failed, and also that the next operation will fail.

Getting input from the user might look like this:

```
int main (void)
{
    int success, x;

    /* prompt the user for input */
    printf ("Enter an integer: \n");

    /* get input */
    success = scanf("%i", &x);

    /* check and see if the input statement succeeded */
    if (success == 1)
    {
        /* print the value we got from the user */
        printf ("Your input: %i\n", x);
        return EXIT_SUCCESS;
    }
    printf("That was not an integer.\n");
    return EXIT_FAILURE;
}
```

There is another potential pitfall connected with the `scanf()` function. Your program code might want to insist that the user types a valid integer, because this value is needed later on. In this case you might want to repeat the input statement in order to get a valid user input:

```
if (success != 1)
{
      while (success != 1)
      {
           printf("That was not a number. Please try again:\n");
           success = scanf("%i", &x);
      }
 }
```

Unfortunately this code leads into an endless loop. You probably ask yourself, why? The input from the keyboard is delivered to your program by the operating system, in something called an input buffer. A successful read operation automatically empties this buffer. However, if the `scanf()` function fails, like in our example, the buffer does not get emptied and the next `scanf()` operation re-reads the old value - you see the problem?

We need to empty the input buffer, before we can attempt to read the next input from the user. Since there is no standard way to do this, we will introduce our own code that reads and empties the buffer using the `getchar()` function. It run through a `while`-loop until there are no more characters left in the buffer (notice the construction of this loop, where all the operations are executed in the test condition):

```
char ch;    /* helper variable stores discarded chars*/
while (success != 1)
{
    printf("That isn’t a number. Please try again:\n");
     /* now we empty the input buffer*/
     while ((ch = getchar()) != ’\n’ && ch != EOF);
     success = scanf("%i", &x);
}
```

The `scanf()` function can also be used to input a `string`:

```
char name[80] ;

printf ("What is your name?");
scanf ("%s", name);
printf ("%s", name);
```

Again, we have to make sure our string variable is large enough to contain the complete user input. Notice the difference in the argument of the `scanf()` function when we are reading an `integer` or a `string`. The function requires a pointer to the variable where the input value will be stored. If we are reading an `integer` we need to use the address operator `&` with the variable name. In the case of a `string` we simply provide the variable name.

Also notice, that the `scanf()` function only takes the first word of the input, and leaves the rest for the next input statement. So, if you run this program and type your full name, it will only output your first name.

8.13 Glossary

index: A variable or value used to select one of the members of an ordered set, like a character from a string.

traverse: To iterate through all the elements of a set performing a similar operation on each.

counter: A variable used to count something, usually initialized to zero and then incremented.

concatenate: To join two operands end-to-end.

pointer: A reference to an object in computer memory.

address: The exact storage location of objects in memory.

8.14 Exercises

Exercise 8.1 A word is said to be "abecedarian" if the letters in the word appear in alphabetical order. For example, the following are all 6-letter English abecedarian words.

> abdest, acknow, acorsy, adempt, adipsy, agnosy, befist, behint, beknow, bijoux, biopsy, cestuy, chintz, deflux, dehors, dehort, deinos, diluvy, dimpsy

a. Describe an algorithm for checking whether a given word (String) is abecedarian, assuming that the word contains only lower-case letters. Your algorithm can be iterative or recursive.

b. Implement your algorithm in a function called `IsAbecedarian()`.

Exercise 8.2 Write a function called `LetterHist()` that takes a String as a parameter and that returns a histogram of the letters in the String. The zeroeth element of the histogram should contain the number of a's in the String (upper and lower case); the 25th element should contain the number of z's. Your solution should only traverse the String once.

Exercise 8.3 A word is said to be a "doubloon" if every letter that appears in the word appears exactly twice. For example, the following are all the doubloons I found in my dictionary.

> Abba, Anna, appall, appearer, appeases, arraigning, beriberi, bilabial, boob, Caucasus, coco, Dada, deed, Emmett, Hannah, horseshoer, intestines, Isis, mama, Mimi, murmur, noon, Otto, papa, peep, reappear, redder, sees, Shanghaiings, Toto

Write a function called `IsDoubloon()` that returns `TRUE` if the given word is a doubloon and `FALSE` otherwise.

Exercise 8.4

The Captain Crunch decoder ring works by taking each letter in a string and adding 13 to it. For example, 'a' becomes 'n' and 'b' becomes 'o'. The letters "wrap around" at the end, so 'z' becomes 'm'.

a. Write a function that takes a String and that returns a new String containing the encoded version. You should assume that the String contains upper and lower case letters, and spaces, but no other punctuation. Lower case letters should be tranformed into other lower case letters; upper into upper. You should not encode the spaces.

b. Generalize the Captain Crunch method so that instead of adding 13 to the letters, it adds any given amount. Now you should be able to encode things by adding 13 and decode them by adding -13. Try it.

Exercise 8.5 In Scrabble each player has a set of tiles with letters on them, and the object of the game is to use those letters to spell words. The scoring system is complicated, but as a rough guide longer words are often worth more than shorter words.

Imagine you are given your set of tiles as a String, like `"qijibo"` and you are given another String to test, like `"jib"`. Write a function called `TestWord()` that takes these two Strings and returns true if the set of tiles can be used to spell the word. You might have more than one tile with the same letter, but you can only use each tile once.

Exercise 8.6 In real Scrabble, there are some blank tiles that can be used as wild cards; that is, a blank tile can be used to represent any letter.

Think of an algorithm for `TestWord()` that deals with wild cards. Don't get bogged down in details of implementation like how to represent wild cards. Just describe the algorithm, using English, pseudocode, or C.

Chapter 9

Structures

9.1 Compound values

Most of the data types we have been working with represent a single value—an integer, a floating-point number, a character value. Strings are different in the sense that they are made up of smaller pieces, the characters. Thus, strings are an example of a **compound** type.

Depending on what we are doing, we may want to treat a compound type as a single thing (or object), or we may want to access its parts (or member variables). This ambiguity is useful.

It is also useful to be able to create your own compound values. C provides a mechanism for doing that: **structures**.

9.2 `Point` objects

As a simple example of a compound structure, consider the concept of a mathematical point. At one level, a point is two numbers (coordinates) that we treat collectively as a single object. In mathematical notation, points are often written in parentheses, with a comma separating the coordinates. For example, $(0, 0)$ indicates the origin, and (x, y) indicates the point x units to the right and y units up from the origin.

A natural way to represent a point in C is with two `double`s. The question, then, is how to group these two values into a compound object, or structure. The answer is a `struct` definition:

```
typedef struct
{
    double x;
    double y;
} Point_t;
```

`struct` definitions appear outside of any function definition, usually at the beginning of the program (after the `include` statements).

This definition indicates that there are two elements in this structure, named `x` and `y`. These elements are called the **members** or **fields** of a structure.

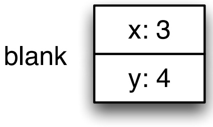
It is a common error to leave off the semi-colon at the end of a structure definition. It might seem odd to put a semi-colon after curly-brackets, but you'll get used to it.

Once you have defined the new structure, you can create variables with that type:

```
Point_t blank;
blank.x = 3.0;
blank.y = 4.0;
```

The first line is a conventional variable declaration: `blank` has type `Point_t`. The next two lines initialize the fields of the structure. The "dot notation" used here is called the **field selection operator** and allows to access the structure fields.

The result of these assignments is shown in the following state diagram:



As usual, the name of the variable `blank` appears outside the box and its value appears inside the box. In this case, that value is a compound object with two named member variables.

9.3 Accessing member variables

You can read the values of an member variable using the same syntax we used to write them:

```
double x = blank.x;
```

The expression `blank.x` means "go to the object named `blank` and get the value of `x`." In this case we assign that value to a local variable named `x`. Notice that there is no conflict between the local variable named `x` and the member variable named `x`. The purpose of dot notation is to identify *which* variable you are referring to unambiguously.

You can use dot notation as part of any C expression, so the following are legal.

```
printf ("%0.1f, %0.1f\n", blank.x, blank.y);
double distance = blank.x * blank.x + blank.y * blank.y;
```

The first line outputs `3, 4`; the second line calculates the value 25.

9.4 Operations on structures

Most of the operators we have been using on other types, like mathematical operators (+, %, etc.) and comparison operators (==, >, etc.), do not work on structures.

On the other hand, the assignment operator *does* work for structures. It can be used in two ways: to initialize the member variables of a structure or to copy the member variables from one structure to another. An initialization looks like this:

```
Point_t blank = { 3.0, 4.0 };
```

The values in squiggly braces get assigned to the member variables of the structure one by one, in order. So in this case, x gets the first value and y gets the second.

Unfortunately, this syntax can be used only in an initialization, not in an assignment statement. So the following is illegal:

```
Point_t blank;
blank = { 3.0, 4.0 };       /* WRONG !! */
```

You might wonder why this perfectly reasonable statement should be illegal; I'm not sure, but I think the problem is that the compiler doesn't know what type the right hand side should be. You must specify the type of the assignment by adding a typecast:

```
Point_t blank;
blank = (Point_t){ 3.0, 4.0 };
```

That works.

It is legal to assign one structure to another. For example:

```
Point_t p1 = { 3.0, 4.0 };
Point_t p2 = p1;
printf ("%f, %f\n", p2.x, p2.y);
```

The output of this program is 3, 4.

9.5 Structures as parameters

You can pass structures as parameters in the usual way. For example,

```
void PrintPoint (Point_t point)
{
    printf ("(%0.1f, %0.1f)\n", point.x, point.y);
}
```

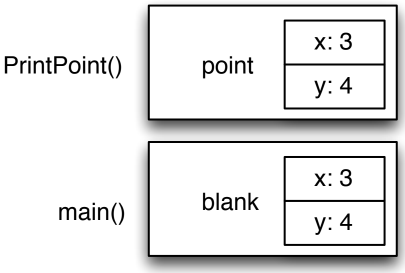
PrintPoint() takes a point as an argument and outputs it in the standard format. If you call PrintPoint(blank), it will output (3.0, 4.0).

As a second example, we can rewrite the ComputeDistance() function from Section 5.2 so that it takes two Points as parameters instead of four doubles.

```
double ComputeDistance (Point_t p1, Point_t p2)
{
    double dx = p2.x - p1.x;
    double dy = p2.y - p1.y;
    return sqrt (dx*dx + dy*dy);
}
```

9.6 Call by value

When you pass a structure as an argument, remember that the argument and the parameter are not the same variable. Instead, there are two variables (one in the caller and one in the callee) that have the same value, at least initially. For example, when we call `PrintPoint()`, the stack diagram looks like this:



If `PrintPoint()` happened to change one of the member variables of `point`, it would have no effect on `blank`. Of course, there is no reason for `PrintPoint()` to modify its parameter, so this isolation between the two functions is appropriate.

This kind of parameter-passing is called "pass by value" because it is the value of the structure (or other type) that gets passed to the function.

9.7 Call by reference

An alternative parameter-passing mechanism that is available in C is called "pass by reference." By now we already know that C uses pointers as references. This mechanism makes it possible to pass a structure to a procedure and modify it directly.

For example, you can reflect a point around the 45-degree line by swapping the two coordinates. The most obvious (but incorrect) way to write a `ReflectPoint()` function is something like this:

```
void ReflectPoint (Point_t point)      /* Does not work! */
{
    double temp = point.x;
```

```
        point.x = point.y;
        point.y = temp;
    }
```

This won't work, because the changes we make in `ReflectPoint()` will have no effect on the caller.

Instead, we have to specify that we want to pass the parameter by reference. Our function now has a struct pointer argument `Point_t *ptr`.

```
    void ReflectPoint (Point_t *ptr)
    {
        double temp = ptr->x;
        ptr->x = ptr->y;
        ptr->y = temp;
    }
```

When we are accessing the struct member variables through a pointer we can no longer use the "field-selection-operator" (`.`). Instead we need to use the "pointing-to" operator (`->`).

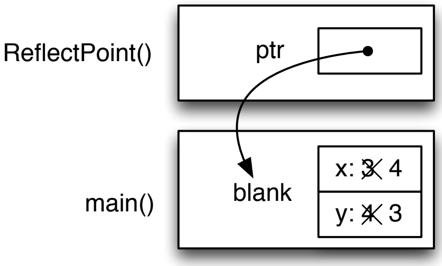
We pass a reference of our struct parameter by adding the "address-of" operator (`&`) to the structure variable when we call the function:

```
    PrintPoint (blank);
    ReflectPoint (&blank);
    PrintPoint (blank);
```

The output of this program is as expected:

```
    (3.0, 4.0)
    (4.0, 3.0)
```

Here's how we would draw a stack diagram for this program:



The parameter `ptr` is a reference to the structure named `blank`. The usual representation for a reference is a dot with an arrow that points to whatever the reference refers to.

The important thing to see in this diagram is that any changes that `ReflectPoint()` makes through `ptr` will also affect `blank`.

Passing structures by reference is more versatile than passing by value, because the callee can modify the structure. It is also faster, because the system does not have to copy the whole structure. On the other hand, it is less safe, since it is harder to keep track of what gets modified where. Nevertheless, in C programs, almost all structures are passed by reference almost all the time. In this book I will follow that convention.

9.8 Rectangles

Now let's say that we want to create a structure to represent a rectangle. The question is, what information do I have to provide in order to specify a rectangle? To keep things simple let's assume that the rectangle will be oriented vertically or horizontally, never at an angle.

There are a few possibilities: I could specify the center of the rectangle (two coordinates) and its size (width and height), or I could specify one of the corners and the size, or I could specify two opposing corners.

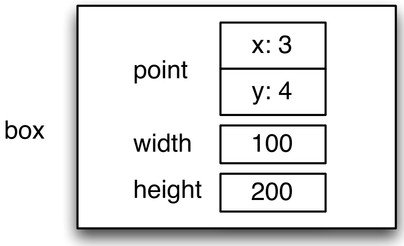
The most common choice in existing programs is to specify the upper left corner of the rectangle and the size. To do that in C, we will define a structure that contains a `Point_t` and two doubles.

```
typedef struct
{
    Point_t corner;
    double width, height;
} Rectangle_t;
```

Notice that one structure can contain another. In fact, this sort of thing is quite common. Of course, this means that in order to create a `Rectangle_t`, we have to create a `Point_t` first:

```
Point_t corner = { 0.0, 0.0 };
Rectangle_t box = { corner, 100.0, 200.0 };
```

This code creates a new `Rectangle_t` structure and initializes the member variables. The figure shows the effect of this assignment.



We can access the `width` and `height` in the usual way:

```
box.width += 50.0;
printf("%f\n", box.width);
```

In order to access the member variables of corner, we can use a temporary variable:

```
Point_t temp = box.corner;
double x = temp.x;
```

Alternatively, we can compose the two statements:

```
double x = box.corner.x;
```

It makes the most sense to read this statement from right to left: "Extract x from the corner of the box, and assign it to the local variable x."

While we are on the subject of composition, I should point out that you can, in fact, create the Point and the Rectangle at the same time:

```
Rectangle_t box = { { 0.0, 0.0 }, 100.0, 200.0 };
```

The innermost squiggly braces are the coordinates of the corner point; together they make up the first of the three values that go into the new Rectangle. This statement is an example of **nested structure**.

9.9 Structures as return types

You can write functions that return structures. For example, FindCenter() has a Rectangle_t parameter and returns a Point_t that contains the coordinates of the center of the rectangle:

```
Point_t FindCenter (Rectangle_t box)
{
    double x = box.corner.x + box.width/2;
    double y = box.corner.y + box.height/2;
    Point_t result = {x, y};
    return result;
}
```

To call this function, we have to pass a Rectangle_t as an argument (notice that it is being passed by value), and assign the return value to a Point_t variable:

```
Rectangle_t box = { {0.0, 0.0}, 100, 200 };
Point_t center = FindCenter (box);
PrintPoint (center);
```

The output of this program is (50, 100).

We could have passed the structure as a reference to the function. In this case our function would look like this:

```
Point_t FindCenter (Rectangle_t *box)
{
    double x = box->corner.x + box->width/2;
    double y = box->corner.y + box->height/2;
    Point_t result = {x, y};
    return result;
}
```

Notice, how we had to change the access to the members of the structure, since `box` is now a pointer. We would also have to change the function call for `FindCenter()`:

```
Point_t center = FindCenter (&box);
```

9.10 Passing other types by reference

It's not just structures that can be passed by reference. All the other types we've seen can, too. For example, to swap two integers, we could write something like:

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

We would call this function in the usual way:

```
int i = 7;
int j = 9;
printf (" i=%i, j=%i\n", i, j);
Swap (&i, &j);
printf (" i=%i, j=%i\n", i, j);
```

The output of this program shows that the variable values have been swapped. Draw a stack diagram for this program to convince yourself this is true. If the parameters `x` and `y` were declared as regular integer variables (without the s), `Swap()` would not work. It would modify `x` and `y` and have no effect on `i` and `j`.

When people start passing things like integers by reference, they often try to use an expression as a reference argument. For example:

```
int i = 7;
int j = 9;
Swap (&i, &j+1);          /* WRONG!! */
```

Presumably the programmer wanted to increase the value of `j` by `1` before it is passed to the function. This does not work as expected, because the expression `j+1` now is interpreted a pointer value and in now pointing to a memory location beyond the variable `j`. It is a little tricky to figure out exactly what kinds of expressions make sense to be passed by reference. For now a good rule of thumb is that reference arguments have to be variables.

9.11 Glossary

structure: A collection of data grouped together and treated as a single object.

member variable: One of the named pieces of data that make up a structure.

reference: A value that indicates or refers to a variable or structure. In a state diagram, a reference appears as an arrow.

pass by value: A method of parameter-passing in which the value provided as an argument is copied into the corresponding parameter, but the parameter and the argument occupy distinct locations.

pass by reference: A method of parameter-passing in which the parameter is a reference to the argument variable. Changes to the parameter also affect the argument variable.

9.12 Exercises

Exercise 9.1

Section 9.5 defines the function `PrintPoint()`. The argument of this function is passed along as a value (call-by-value).

a. Change the definition of this function, so that it only passes a reference of the structure to the function for printing (call-by-reference).

b. Test the new function with different values and document the results.

Exercise 9.2

Most computer games can capture our interest only when their actions are non-predictable, otherwise they become boring quickly. Section 7.6 tells us how to generate random numbers in in C.

Write a simple game, where the computer chooses an arbitrary number in the range between 1 and 20. You will then be asked to guess the number chosen by the Computer.

To give you a hint the computer should answer your guess in the following way: in case your guess was lower than the number of the computer, the output should be:
My number is larger!
If you guess was higher than the number of the computer, the output should read:
My number is smaller!

It is necessary to seed the random number generator when you start your program (cf. Section 7.14. You could use the `time()` function for this. It returns the actual time measured in seconds since 1971.

```
srand(time(NULL));   /*Initialisation of the random number generator*/
```

When you found the right answer, the computer should congratulate you. The program should also display the number of tries that where needed in order to guess the number.

The program should also keep the a ‘High-Score’, that gets updated once our number of trials is lower than any previous try. The High-Score (the number of minimal guesses) should be stored in a `struct`, together with the name of the player.

The `High-Score()` function should ask for your name and store it, when your current number of tries is lower than the previous High-Score value.

The program then gives you the chance to play again or stop the game by pressing ’q’ on the keyboard.

Appendix A

Coding Style

A.1 A short guide on style

In the last few sections, I used the phrase “by convention” several times to indicate design decisions that are arbitrary in the sense that there are no significant reasons to do things one way or another, but dictated by convention.

In these cases, it is to your advantage to be familiar with convention and use it, since it will make your programs easier for others to understand. At the same time, it is important to distinguish between (at least) three kinds of rules:

Divine law: This is my phrase to indicate a rule that is true because of some underlying principle of logic or mathematics, and that is true in any programming language (or other formal system). For example, there is no way to specify the location and size of a bounding box using fewer than four pieces of information. Another example is that adding integers is commutative. That’s part of the definition of addition and has nothing to do with C.

Rules of C: These are the syntactic and semantic rules of C that you cannot violate, because the resulting program will not compile or run. Some are arbitrary; for example, the fact that the = symbol represents assignment and *not* equality. Others reflect underlying limitations of the compilation or execution process. For example, you have to specify the types of parameters, but not arguments.

Style and convention: There are a lot of rules that are not enforced by the compiler, but that are essential for writing programs that are correct, that you can debug and modify, and that others can read. Examples include indentation and the placement of squiggly braces, as well as conventions for naming variables, functions and types.

In this section I will briefly summarize the coding style used within this book. It follows loosely the "Nasa C Style Guide" [1] and its main intent is on readability rather than saving space or typing effort.

Since C has such a long history of usage, many different coding styles have been developed and used. It is important that you can read them and follow one particular scheme in all your code. This makes it much more accessible should you find yourself in a position where you have to share your work with other people or have to access code written by your younger self - many years ago...

A.2 Naming conventions and capitalization rules

As a general rule, you should always choose meaningful names for your identifiers. Ideally the name of a variable or function already explains its behaviour or use.

It may be more typing effort to use a function named `FindSubString()` rather than `FndSStr()`. However, the former is almost self describing and might save you a lot in debugging-time.

Don't use single letter variable names!

Similarly to functions, you should give your variables names that speak for themselves and make clear what values will be stored by this variable. There are few noticeable exceptions to this rule: People use `i`, `j` and `k` as counter variables in loops and for spacial coordinates people use `x`, `y` and `z`. Use these conventions if they suit you. Don't try to invent new conventions all by yourself.

The following capitalization style shold be used for the different elements in your program. The consistent use of one style gives the programmer and the reader of the source code a quick way to determine the meaning of different items in your program:

variableNames: variable names always start with lower-case, multiple words are separated by capitalizing the first letter.

CONSTANTS: use all upper case letters. In order to avoid name space collisions it might be necessary to use a prefix such as `MY_CONSTANT`.

FunctionNames: start always with upper case and should possibly contain a verb describing the function. Names for functions that test values should start with '`Is`' or '`Are`'.

UserDefinedTypes_t: always end in '`_t`'. Type names names must be capitalised in order to avoid conflict with POSIX names.

pointerNames_p: in order to visually separate pointer variables from ordinary variables you should consider ending pointers with '`_p`'.

[1] www.scribd.com/doc/6878959/NASA-C-programming-guide

A.3 Bracing style

There exist different bracing or indent styles that serve the goal to make your code more readable through the use of a consistent indentation for control block structures. The styles differ in the way the braces are indented with the rest of the control block. This book uses the BSD/Allman Style because its is the most readable of the four. It needs more horizontal space than the K&R Style but it makes it very easy to track opening and closing braces.

When you are writing programs, make sure that you are using one style consistently. In larger projects all contributors should agree on the style they are using. Modern programming environments like Eclipse support you through the automatic enforcement of a single style.

```
/*Whitesmiths Style*/
   if (condition)
       {
       statement1;
       statement2;
       }
```

Is named after Whitesmiths C, an early commercial C compiler that used this style in its examples. Some people refer to it as the One True Brace Style.

```
/*GNU Style*/
   if (condition)
     {
       statement1;
       statement2;
     }
```

Indents are always four spaces per level, with the braces halfway between the outer and inner indent levels.

```
/*K&R/Kernel Style*/
   if (condition) {
       statement1;
       statement2;
   }
```

This style is named after the programming examples in the book *The C Programming Language* by Brian W. Kernighan and Dennis Ritchie (the C inventors).

The K&R style is the style that is hardest to read. The opening brace happens to be at the far right side of the control statement and can be hard to find. The braces therefore have different indentation levels. Nevertheless, many C programs use this style. So you should be able to read it.

```
/*BSD/Allman Style*/
   if (condition)
   {
       statement1;
       statement2;
   }
```

This style is used for all the examples in this book.

A.4 Layout

Block comments should be used at the top of your file, before all function declarations, to explain the purpose of the program and give additional information.

You should also use a similar documentation style before every relevant function in your program.

```
/*
 * File:     test.c
 * Author:   Peter Programmer
 * Date:     May, 29th, 2009
 *
 * Purpose: to demonstrate good programming
 *          practise
 * /

#include <stdlib.h>

/*
 * main function, does not use arguments
 */

int main (void)
{
    return EXIT_SUCCESS;
}
```

Appendix B

ASCII-Table

Dec	Hex	Oct	Character	Dec	Hex	Oct	Character
0	0x00	000	NUL	32	0x20	040	SP
1	0x01	001	SOH	33	0x21	041	!
2	0x02	002	STX	34	0x22	042	"’
3	0x03	003	ETX	35	0x23	043	#
4	0x04	004	EOT	36	0x24	044	$
5	0x05	005	ENQ	37	0x25	045	%
6	0x06	006	ACK	38	0x26	046	&
7	0x07	007	BEL	39	0x27	047	’
8	0x08	010	BS	40	0x28	050	(
9	0x09	011	TAB	41	0x29	051	)
10	0x0A	012	LF	42	0x2A	052	*
11	0x0B	013	VT	43	0x2B	053	+
12	0x0C	014	FF	44	0x2C	054	,
13	0x0D	015	CR	45	0x2D	055	-
14	0x0E	016	SO	46	0x2E	056	.
15	0x0F	017	SI	47	0x2F	057	/
16	0x10	020	DLE	48	0x30	060	0
17	0x11	021	DC1	49	0x31	061	1
18	0x12	022	DC2	50	0x32	062	2
19	0x13	023	DC3	51	0x33	063	3
20	0x14	024	DC4	52	0x34	064	4
21	0x15	025	NAK	53	0x35	065	5
22	0x16	026	SYN	54	0x36	066	6
23	0x17	027	ETB	55	0x37	067	7
24	0x18	030	CAN	56	0x38	070	8
25	0x19	031	EM	57	0x39	071	9
26	0x1A	032	SUB	58	0x3A	072	:
27	0x1B	033	ESC	59	0x3B	073	;
28	0x1C	034	FS	60	0x3C	074	"<
29	0x1D	035	GS	61	0x3D	075	=
30	0x1E	036	RS	62	0x3E	076	">
31	0x1F	037	US	63	0x3F	077	?

Dec	Hex	Oct	Character	Dec	Hex	Oct	Character
64	0x40	100	@	96	0x60	140	‘
65	0x41	101	A	97	0x61	141	a
66	0x42	102	B	98	0x62	142	b
67	0x43	103	C	99	0x63	143	c
68	0x44	104	D	100	0x64	144	d
69	0x45	105	E	101	0x65	145	e
70	0x46	106	F	102	0x66	146	f
71	0x47	107	G	103	0x67	147	g
72	0x48	110	H	104	0x68	150	h
73	0x49	111	I	105	0x69	151	i
74	0x4A	112	J	106	0x6A	152	j
75	0x4B	113	K	107	0x6B	153	k
76	0x4C	114	L	108	0x6C	154	l
77	0x4D	115	M	109	0x6D	155	m
78	0x4E	116	N	110	0x6E	156	n
79	0x4F	117	O	111	0x6F	157	o
80	0x50	120	P	112	0x70	160	p
81	0x51	121	Q	113	0x71	161	q
82	0x52	122	R	114	0x72	162	r
83	0x53	123	S	115	0x73	163	s
84	0x54	124	T	116	0x74	164	t
85	0x55	125	U	117	0x75	165	u
86	0x56	126	V	118	0x76	166	v
87	0x57	127	W	119	0x77	167	w
88	0x58	130	X	120	0x78	170	x
89	0x59	131	Y	121	0x79	171	y
90	0x5A	132	Z	122	0x7A	172	z
91	0x5B	133	[	123	0x7B	173	{
92	0x5C	134	\	124	0x7C	174	|
93	0x5D	135	]	125	0x7D	175	}
94	0x5E	136	ˆ	126	0x7E	176	"
95	0x5F	137	_	127	0x7F	177	DEL

Index