Program Design in C

Derek Williams

August 29, 2013

Contents

1 Introduction **4**
1.1 Designing Programs in C 4
1.2 Conventions 4
1.3 Author's Note 5

2 First Steps **6**
2.1 Environment 6
2.2 A First Program 6

3 Variables **9**
3.1 Types 9
3.1.1 `char` 10
3.1.2 `int` 10
3.1.3 `float` and `double` 11
3.1.4 Variable Type Qualifiers 11
3.2 Picking the Right Type 11
3.3 Creating Variables 12
3.3.1 Naming 13
3.4 Creating Constants 13
3.5 Arrays 14
3.6 Arithmetic 15
3.6.1 Arithmetic Operators 15
3.6.2 Assignment Operators 16
3.6.3 Increment and Decrement Operators 16

4 Standard Input and Output **17**
4.1 Output 17
4.1.1 Escape Sequences 18

4.1.2 Conversion Characters 18
4.2 Input 19

5 Flow Control 22
5.1 Logical Comparison 22
5.2 `if` and `else` 23
5.3 `switch`, `case`, and `break` 25
5.4 `while` 26
5.5 `do/while` 26
5.6 `for` 27
5.7 Ternary Operator 28
5.8 Nested Loops 28
5.9 Infinite Loops 28

6 Functions 30
6.1 Function Basics 30
6.2 Function Prototypes 32
6.3 Scope 34
6.4 Static Variables 36
6.5 Recursion 36

7 Pointers 39
7.1 Pointer Basics 39
7.2 Pointers and Functions 40
7.3 Pointer Arithmetic and Arrays 41
7.4 Dynamically Allocated Memory 42
7.5 Double Pointers 43

8 Structures 45
8.1 Structure Basics 45
8.2 Structures, Pointers, and Functions 46
8.3 Linked Lists 48
8.4 Unions 50

9 Files 51
9.1 File Modes 51
9.2 Opening and Closing a File 52
9.3 Inputting from a Text File 52

9.4 Outputting to a Text File 54
9.5 Inputting from a Binary File 55
9.6 Outputting to a Binary File 55
9.7 Text vs. Binary 56

10 `ctype.h` 57

11 `string.h` 59

11.1 String Tokenization 60

12 `math.h` 62

13 `stdlib.h` 63

14 Miscellany 64

14.1 Coding Style 64
14.2 `typedef` and `enum` 65
14.3 Typecasting 67
14.4 Preprocessor Directive `#define` 67
14.5 Command Line Arguments 70
14.6 Bit Fields 71
14.6.1 Accessing One Bit 71
14.6.2 Setting a Bit 72
14.6.3 Checking a Bit 72
14.6.4 Clearing a Bit 73
14.7 Pseudorandom Number Generation 74
14.8 Partial ASCII Table 75

Chapter 1

Introduction

1.1 Designing Programs in C

Designing and implementing computer programs is as much of an art as writing an english paper or mathematical proof. Some think that if a program works, then the actual mechanics behind it are irrelevant. One must, however, consider the potential lifetime of a program. A program may be used once and then deleted, one may refer back to the program years later to recall some long forgotten function, or in the case where design is of the utmost importance one may need to give the code to someone else to use, work with, or modify.

This document is designed to assist teaching program design in C to those who have never programmed in C before and it presumes no prior programming knowledge. This document does place emphasis on good design. In learning good design, you'll also be able to choose the most appropriate way to design and implement a program based on the specific requirements and expected lifetime of that program.

1.2 Conventions

Various conventions are used in this document to distinguish code and output from regular text. All code and output will be in a monospaced font, either in paragraph or outside of a paragraph, as such:

```
#include <stdio.h>

/* This is a simple program */

int main()
{
     printf("Hello\nworld!\n");

     return(0);
}
```

Terms of importance or interest will be put in **bold** type.

1.3 Author's Note

This document was designed for the CECS 121 course at the University of Louisville. At the completion of the course, students are prepared to take courses on basic computer science, data structures, C++, and Java.

Thanks to the following people who have helped me with this book, through editing, finding errors, or making suggestions: Tim Hardin, Daniel Irwin, Kris Kumler, Tom St. Denis, Max Stoler.

If you see any errors or have any suggestions, examples, definitions, or anything else that you'd like to bring to my attention to make this document better, please email me at derk@derk.org. If this book has helped you in any way, or you're using it for a course, feel free to let me know as well!

This book is licensed under the Creative Commons Attribution Noncommercial Share Alike license version 2.0. For more details see:
`http://creativecommons.org/licenses/by-nc-sa/2.0/`

Chapter 2

First Steps

2.1 Environment

Before starting programming, you must have an environment to program in. An environment consists of two things: a text editor and a compiler.

A **text editor** is any program that a user can enter text into such that text can be saved to a file exactly as it was typed in, without any superfluous information. For Microsoft Windows, such editors include notepad, wordpad, and Microsoft Word. On Linux/UNIX systems, and the new Mac operating systems, editors include pico, nano, vi, and emacs.

A **compiler** is a program that compiles the code in a text file and (after linking) produces an executable file. The executable file is just like any other program that runs on a computer. For Linux/UNIX/Mac systems the preferred compiler is gcc. Microsoft Windows users can use the various incarnations of Visual Studio, but gcc has also been ported to Windows as part of the DJGPP package.

Any of the software above can easily be found by doing a search on the Internet.

2.2 A First Program

We shall start with what is perhaps the most simple program – one that outputs just one line to the screen and is done. Here it is:

```
1  #include <stdio.h>
2
3  /* This is a simple program */
4
5  int main()
6  {
7      printf("Hello\nworld!\n");
8
9      return(0);
10 }
```

It should be noted that the line numbers here are not actually part of the program, but exist only as a way to refer to lines.

Line 1 contains what is known as a **preprocessor directive**. All preprocessor directives start with the hash mark `#`. When the compiler starts to compile the code, the first thing it does is look at the code and does whatever the preprocessor directives say to do first, then it compiles the rest. This particular directive tells the compiler to include a **header file** called `stdio.h`. `stdio.h` contains functions related to input and output and the reason we need it here is because we use the function `printf()` in line 7.

Lines 2, 4, and 8 are blank lines. While having these blank lines here are not necessary, they are good coding style and help improve readability.

Line 3 contains a **comment**. Anything between `/*` and `*/` is a comment. Whatever is written here is not compiled. When the compiler compiles the code, it doesn't pay attention to what is in here. Writing comments is also a component of good design. One should comment to describe any relevant information about the program as a whole, such as the date, what the program does, and the programmer's name. Also, one should write comments about particularly confusing pieces of code or those that another programmer may not readily understand. Comments are very useful for other reasons, which we will study later.

Line 5 is the declaration of the function called `main()`. It is often times called the **main body**. The main body is where all programs start executing, after they're compiled and run. Eventually we will include functions we write ourselves in the program, but for now, the main body is all there is. All programs have a main body.

Note that before `main()` we put `int`. `int` is the **return type** of the function main. This means that the main body will return an integer value to the operating system when it is finished. For now, this is not so important. When we write our own functions, return types will allow us to get a value back from the function. For instance, a function that calculates the sine of an angle, we would provide the angle as an argument, x, and the function would return sin(x).

Lines 6 and 10 comprise the beginning and ending of a **block** of code, respectively. Every { has a corresponding }. Every function contains a block, and since `main()` is a function, it contains a block. A block serves to execute multiple statements, in order, from start to finish as one statement. This is of much more importance when we start discussing loops.

Of utmost importance, with respect to coding style, is the fact that both braces are aligned, vertically, and that all code between the braces is tabbed over. This serves to make the code much more readable. If ever there is a time such that two braces are one character away from each other, horizontally or vertically, it probably means that the code was written with a poor style.

Line 7 contains a function call to `printf()`. `printf()` is a function located in `stdio.h` that allows you to write output to the output device, `stdout`. `stdout` typically means the screen. In its most simple form, `printf()` takes as an argument (an argument is anything between parentheses) a single string (a string is a collection of characters between two double quotes), which it prints out.

The character '`\n`' is what is known as an **escape sequence**. It tells the function `printf()` to go to the next line on the output. So, when run, this code will output:

```
Hello
world!
```

Line 9 contains the `return()` statement. Since the `main()` has a return type of `int`, we place an integer value (in this case, 0) as the **return value**. As stated before, this will become more important when we write our own functions, and as we use more advanced functions that we gain access to by including header files, such as in line 1.

Try entering in this code into the text editor and compiling it, you should see that the output agrees with what is above.

Chapter 3

Variables

3.1 Types

Variables exist for data to be stored in. Storing and manipulating data is the essence of programming. The good thing about C is that there are only a few basic data types (often called primitives). The basic data types are character, integer, real number, and memory address (these will be discussed in a later chapter). Perhaps even more fortunate is the fact that a character is really nothing more than an integer! Here are the basic variable types in C:

- `char`: able to store character values like 'A', 'e', '@', and '7'
- `int`: able to store integer values like 500, -20, and 16296898
- `float`: able to store real number values like 3.14, 2.718, and 6.0
- `double`: able to store real number values like 3.14, 2.718, and 6.0

Before discussing data types related to and derived from the basic ones, let us first introduce the concept of **range**. The range of a variable type is the set of possible values that can be stored in a variable of that type. For instance, the real number 6.55 is not an integer, and therefore cannot be stored in an integer data type. Here are some typical ranges on the above data types on a 32-bit system[1]:

[1]These values are not the same on every system. These values can be found by looking in the file `limits.h`.

- `char`: 0 to 255 (1 byte)
- `int`: -2147483648 to 2147483647 (4 bytes)
- `float`: $-3.40 \cdot 10^{38}$ to $-1.40 \cdot 10^{-45}$ and $1.40 \cdot 10^{-45}$ to $3.40 \cdot 10^{38}$ with 6 decimal place accuracy. (4 bytes)
- `double`: $-1.80 \cdot 10^{308}$ to $-4.94 \cdot 10^{-324}$ and $4.94 \cdot 10^{-324}$ to $1.80 \cdot 10^{308}$ with 15 decimal place accuracy. (8 bytes)

3.1.1 `char`

To understand the ranges of the variables, one must understand exactly what bits are used for each type. The `char` data type contains 1 byte, which consists of 8 bits. Each bit can take on a value of 0 or 1, so there are $2^8 = 256$ possible values, from `00000000` $= 0$ to `11111111` $= 255$. The right most bit is the least significant bit, so `00000001` is $0 \cdot 2^7 + ... + 1 \cdot 2^0 = 1$, and `10000000` is $1 \cdot 2^7 + 0 \cdot 2^6 + ... = 128$. To represent characters, we encode each each character as one of the values in the range. For instance, the character 'A' is encoded as the value 65 (`01000001`), 'B' is encoded as the value 66 (`01000010`), etc. The ASCII (American Standard Code for Information Interchange) table shows all the characters and corresponding codes.

One immediate consequence of the way the ASCII table is structured is that we can easily add letters to numbers and get the coded value of the character we'd expect, in most cases. For example, if we add 1 to 'A', we'd get $1 + 65 = 66 =$ 'B'. However, if we were to add 1 to 'Z' we'd get '['.

3.1.2 `int`

The `int` data type is a natural extension of the `char` data type, with a couple of minor changes. The most obvious is that the data type is now 4 bytes in size, therefore there are $4 \cdot 8 = 32$ bits, or $2^{32} = 4294967296$ possible values. In this case, we need to represent positive AND negative values, so one bit is reserved for the sign (positive or negative), and the other 31 bits are used for values. When the most significant bit (leftmost bit) is 0, the values represented by the rightmost 31 bits are positive. Since

$2^{31} = 2147483648$, the range is 0 to 2147483647. Likewise, when the most significant bit (leftmost bit) is 1, the values represented by the rightmost 31 bits are negative. Note that there already is a value for 0, so the range on the negative values is -2147483648 to -1.

There are also `short int` and `long int` data types. `short int` uses only 2 bytes instead of 4, so the range is restricted to -32768 to 32767. `long int` uses 4 bytes[2].

3.1.3 `float` and `double`

The `float` data type is structured differently from the `int` and `char` data types. 1 bit in the `float` represents the **sign** of the number, 23 bits represent the **mantissa**, and the remaining 8 bits represent the **exponent**. Values are calculated using the expression $sign \cdot mantissa \cdot 2^{exponent}$.

The `double` data type is like the float, but with 8 bytes instead of 4 bytes, hence an extended range and accuracy. A `double` uses 1 bit for the sign, 52 bits for the mantissa, and 11 bits for the exponent.

3.1.4 Variable Type Qualifiers

There are a number of qualifiers that one can place before some variable types to alter the way they are structured or used. These include `unsigned`, so an `unsigned int` has no sign bit and is therefore always positive, giving it the range from 0 to 4294967295 instead of -2147483648 to 2147483647. Others are `const` and `static`, both of which are both discussed later, and `register`[3].

There is no `unsigned float` or `unsigned double`.

3.2 Picking the Right Type

Picking the appropriate variable for an application is critical, and in most cases it can be done without thought. For instance, to represent a bank

[2]Once again, these values can be may be different on different systems, so check `limits.h`.

[3]Although not mentioned later in this document, making a variable a `register` variable will attempt to keep its contents in the register of a CPU, making calculations with it considerably faster – doing this is uncommon in most programs though.

account balance, we need dollars and cents, so we need decimal places – in this case a `float` would be appropriate but an `int` would not[4]. However, even though memory and disk space aren't as much of a consideration as before, using a smaller variable can save space. For instance, the age of a person (between 0 and 121 years) could be represented by an `int` (4 bytes), a `short int` (2 bytes), or even a `char` (1 byte).

When variables exceed their values, such as adding 1 to 32767 and storing it in a `short int`, a condition called overflow occurs. Since 32768 cannot be represented in a `short int`, the system will attempt to store a value in the variable. The system will "wrap around" the numbers to the negative side and continue on. In this case, $32767 + 1 = -32768$. In similar cases, $32767 + 1001 = -31768$, $20000 + 20000 = -25534$, $32767 + 32767 = -2$.

3.3 Creating Variables

Before using a variable, you have to create the variable of a certain data type. There are various rules about creating variables, such as what they can be named and where they can be declared.

Variables can only be declared at the beginning of a block of code. Recall that a block is started with the { character. Variables are also case-sensitive, like all things in C. So, a variable named `AGE` is different from `age` which is different from `aGe`.

Variables can only be started with a letter (upper or lower case), or an underscore '_'. After the first letter, variable names can consist of underscores, numbers, or letters. Variables CANNOT be started with a number, however they may contain a number after the first character. Variables also cannot have the same name as a keyword or function name, such as `int` or `main`.

Some valid variable names are `age123`, `_aGe`, `____`, `_127773_`, `INT`, and `int123`. Some invalid variable names are `int`, `21age`, and `while` (`while` is reserved, as we shall see later).

To declare a variable, simply write the name after the type, and put a semicolon after the name. Declaring a variable is a statement, and after every statement, we must use a semicolon. To declare multiple variables of one type, just separate the names by commas. Variables can be declared over

[4]In the real world, even `float` doesn't provide the accuracy necessary to keep track of very large or very small numbers

multiple lines if necessary, just remember that a comma must go at the end of the line where it is broken. Variables can also be initialized when declared, that is a value can be stored in them. Use the '`=`' operator for this. Here is a valid set of variable declarations, illustrating all of the above:

```
int age, year=2004,
    dayofmonth;
float intrate=.05;
float balance=650.20;
```

3.3.1 Naming

Use descriptive names for variables, such that anyone else looking at the code can immediately tell what the variable is used for. Don't go overboard, however, because variable names should be relatively short so that you don't have to spend a lengthy amount of time typing out the variable name each time you want to use it. As a convention, we typically use all lowercase letters for variable names.

3.4 Creating Constants

Some values never change, but it still helps to have a name for them, for example using 'pi' when talking about the value 3.141592653589793.... To create a constant of a certain data type, there is a qualifier called `const` which places a restriction on a variable, such that it can never be changed. A `const` must be initialized, and never after that can its value be changed. If one tries to change the value of a `const` after declaration and initialization, it will cause an error at compile time. Constants are declared in the same place as variables, and as a convention we always use all uppercase letters as their names. Here is an example of declaring two constants:

```
const double PI=3.141592653589793, E=2.718281828;
```

3.5 Arrays

Arrays of variables can be created to store multiple values of the same type. Rather than declaring ten different variables (age1, age2, etc.) to store ten different ages, one can have an array called age that stores 10 values. It should be noted, however, that an array should not be used to store two unrelated values, even if they are the same type, such as age and year.

An array can be declared in multiple ways. The size of the array can be explicitly or implicity defined, but the size of the array is always determined at compile time, and can never be altered afterwards. Using `[]` after the variable name will allow us to declare an array, and the size can be defined within the brackets (explicit size definition), or the array can be initialized in which case the array is whatever size as what it was initialized to (implicit size definition), or both. Initialization is achieved by placing values in a comma-separated set, surrounded with braces. Numbering of an arrays elements starts at 0, so an array of size n will have elements from 0 to $n - 1$. All of that is pretty wordy, but can be summed up with the following four examples, all of which do the same thing:

```
int age[] = { 10, 20, 30 };

int age[3];
age[0] = 10;
age[1] = 20;
age[2] = 30;

int age[3] = { 10, 20, 30 };

int age[3] = { 10, 20 };
age[2] = 30;
```

To fill an array of characters, there is a shorter method. The following two declarations do the same thing, but it's obvious that the latter is the preferred method:

```
char name[] = { 'S', 'm', 'i', 't', 'h', '\0' };

char name[] = "Smith";
```

Character strings end with the `NULL` character, '\0', although not all arrays of characters have to contain this character.

3.6 Arithmetic

3.6.1 Arithmetic Operators

Manipulating variables is the essence of programming. As such, the most common method of doing so is using the basic arithmetic operators used in most mathematical expressions. Addition is done using the '+' operator, subtraction with '-', multiplication with '*', and division with '/'. The minus, '-', also serves as a unary negative operator. Parentheses can be introduced at any time, and precedence follows the normal mathematical precedence[5]. Variables are assigned values using the '=' operator. See this example, where the variable `i` is assigned the value 45:

```
int i, j=50;

i = (10*10 + j)/3 + -5;
```

It is very important to note that in the case where a division doesn't come out whole (e.g. `int i = 3/4;`), typecasting must be done, because setting an `int` equal to a `float` will not work. See the section on typecasting for more information.

There is one arithmetic operator that you may not be familiar with, the modulus operator, '%'. The modulus operator returns the remainder when the first operand is divided by the second operand. So, 5%3 is equal to 2, because when 5 is divided by 3, 2 is left over. It is used in the same syntax as the above four arithmetic operators. This operation is used in higher mathematics all the time, but it also has its uses at a more basic level. For

[5]Expressions in parentheses are evaluated first, then multiplication & division, and finally addition & subtraction.

instance, if a number is equal to 0 when modded with 2, it must be even. Likewise, if a number is equal to 1 when modded with 2, it must be odd.

The modulus operator is the only one that can only be used on integers. The other four arithmetic operators can be applied to integers and any real numbers in general.

3.6.2 Assignment Operators

To save time, there are a number of assignment operators based on the arithmetic operators mentioned in the previous section. These operators are '`+=`', '`-=`', '`*=`', '`/=`', and '`%=`'. The '`+=`' operator will add a value to a variable, and as such `i += 3;` is the same as `i = i + 3;`. The other operators work in the same way.

3.6.3 Increment and Decrement Operators

There also exist operators for incrementing and decrementing a variable by 1. '`++`' is the operator for incrementing, and '`--`' is the operator for decrementing. They can be placed before or after a variable, but note that placing the operator before the variable increments or decrements the variable before the line is evaluated, and placing the operator after the variable increments or decrements the variable after the line is evaluated. See this example:

```
int i=0, j=0, k;

k = i++; /* k is set equal to 0 first,
            then i is incremented to 1 */
k = ++j; /* j is incremented to 1 first,
            then k is set equal to 1 */
```

Chapter 4

Standard Input and Output

Gathering input from the user, and giving output back is of prime importance. Now that variables have been thoroughly covered, the matter at hand is how to put user input into variables, and how to print out variables.

To accomplish input and output, we make use of functions in the `stdio.h` header file.

4.1 Output

Output can be accomplished with many different functions, but one of the simplest, and one which can accomplish everything, is `printf()`. `printf()` has the format `int printf(const char *format, ...)`. This means that `printf()` takes as its first argument a character array, or string. The remainder of the arguments are optional. If variables, or any sort of mathematical expression, needed to be printed out, they would go in the optional argument section. Before `printf()` is the data type `int`, which means that `printf()` returns an integer, and in fact this integer is the number of characters that were written to the screen[1].

For beginners, let's begin with just the first argument of `printf()`, the string. Any text placed within this string will be printed out, for instance with `printf("Welcome!");` the text printed out to the screen will be `Welcome!`. Note that the text is within double quotation marks – all strings are within double quotation marks.

[1]This is only useful in rare situations, so it's not of paramount importance now.

4.1.1 Escape Sequences

It is also necessary at times to print out special characters. The enter key on your keyboard performs two functions, carriage return and line feed. The carriage return brings the cursor back to the front of the line, and the line feed moves down to the next line. Within a `printf()` statement, we must use a special character for this, '`\n`'. So, if the code was:

```
printf("Welcome\neverybody\n");
```

The output would be:

```
Welcome
everybody
```

Characters such as '`\n`' are called **escape sequences**. Note that all escape sequences are actually just one character. In the instance of '`\n`', it is merely shorthand for the ASCII code for the enter key! Here is a list of common escape sequences, and what each does.

Escape Sequence	**Meaning**
`\b`	backspace
`\n`	newline
`\t`	horizontal tab
`\\`	backslash
`\"`	double quote

Since backslash is reserved for starting escape sequences, we need a special escape sequence to print out a backslash. Double quote needs an escape sequence because otherwise, when we placed a double quote, the compiler would think it is the end of our string!

4.1.2 Conversion Characters

Printing out variables is accomplished using special characters as well. These characters are called conversion characters, and instead of starting with a \, they start with a `%`. Conversion characters are placed within the format string and correspond with variable arguments placed after the format string in the same order. There is a conversion character for each primitive data type – if an `int` is to be printed out, the `int` conversion character should be used. Here is a list of conversion characters:

Conversion Character	**Variable Type**
`%d` or `%i`	`int`
`%c`	`char`
`%s`	string (array of `char`)
`%f` or `%e`	`float` and `double`
`%%`	no type, print `%`

Note that there are two conversion characters for `float` and `double`. The first conversion character prints a number out normally, whereas the second conversion character, '`%e`', prints a number out using scientific notation. Since `%` is reserved as the start of a conversion character, we need to use `%%` if we wanted to print out a single percent sign.

Additionally, `%X.Yf` is commonly used as a conversion character, where `X` and `Y` are integers representing horizontal offset and decimal places to be printed out, respectively. For example, if we wanted to print out a number with two decimal places, we'd use `%0.2f` or `%.2f`.

4.2 Input

Input is accomplished using a combination of two functions, `fgets()` and `sscanf()`. `fgets()` is used to read input from the keyboard, and has the format `char *fgets(char *s, int size, FILE *stream);`. The first argument is a character pointer (for now, this is the same thing as a character array) into which the input is stored. The `size` argument is the maximum amount of input that will be stored, and the third argument is a file pointer. When we read from files we will use this same function, but for now, since we want to read from the keyboard, we will use `stdin` as our file pointer[2]. As a return value, `fgets()` returns the value of `s`, the address of where the input is stored (more on this in the chapter on pointers).

`sscanf()` is used to process the data after input. Since input is stored in

[2]There are three streams, `stdin` for keyboard input, `stdout` for screen output, and `stderr` for errors. `stdout` is where `printf()` sends its output to. Streams will be covered in further detail in the chapter on files.

a character array, if we want to get something like an `int` out if it, we must do some processing. `sscanf()` has the format `int sscanf(const char *str, const char *format, ...);`. The first argument is the string that the variables will be scanned out of. The second argument is a format string much like `printf()`, using the same conversion characters. Variables constitute the rest of the arguments, corresponding with the conversion characters in the format string. In order to store the variables, we must preface them with an ampersand, '&', sending the memory address of the variable to the function (more on this in the chapter on pointers). Here is a basic example of how to read an integer in:

```
char buffer[50];
int value;

printf("Enter in a value: ");
fgets(buffer, sizeof(buffer), stdin);
sscanf(buffer, "%d", &value);
```

Note that for the size, we use the `sizeof()` command, which returns the size of the array in bytes (in this case, 50). This will fill the buffer up to the maximum size. Every string ends with the `NULL` character, so up to 49 characters can be stored in the variable `buffer`. `fgets()` continues to read input from the keyboard until the enter key on the keyboard is pressed. `sscanf()` then takes the inputted string and stores the integer value into the variable `value`.

Although it is often not a good program design practice, multiple inputs can be read on one line, as such:

```
char buffer[50];
int value[2];

printf("Enter in two values, separated by spaces: ");
fgets(buffer, sizeof(buffer), stdin);
sscanf(buffer, "%d %d", &value[0], &value[1]);
```

Using a combination of `fgets()` and `sscanf()` to read in variable inputs is fine, but in the case where we desire just a string, we don't need `sscanf()`. However, `fgets()` saves the newline character as part of the character array.

Hence, if the user types in `Smith` and hits enter, the string value in the array will be `Smith\n`. We can remove the newline character using the `strlen()` function from `string.h`. We always want to remove the next to last character in the array that was inputted, so we set that character to `NULL`, as such:

```
char buffer[50];

printf("Enter your name: ");
fgets(buffer, sizeof(buffer), stdin);

buffer[strlen(buffer)-1] = NULL;

printf("Greetings, %s!\n", buffer);
```

Chapter 5

Flow Control

Controlling the flow of a program is the majority of what programming is. The functions in the header files and operators for manipulating variables are the building blocks, but flow control brings everything together to accomplish tasks. Until now, all programs have been linear, in that there is an exact sequence of events that could be accurately anticipated ahead of time. Now we introduce two things, branching (making a decision on what to do next based on the users input), and looping (doing the same sequence of statements over and over until some condition is met).

5.1 Logical Comparison

Logical comparison with conditions will be used as our primary method of flow control. This means that we will be comparing values, to see if they're equal, not equal, greater than, less than, etc. We can also combine these, using logical operators such as AND and OR to produce conditional statements such as ”If it's raining or I have $20, then I will go to the movie”, in which it is is only necessary for one of the conditions to be true. It is of importance to know that the logical value of FALSE is 0, and TRUE is represented by any non-zero value. Here is a list of operators that can be used for logical comparison:

Condition	**Returns true if...**
`x`	`x` is true (non-zero)
`!x`	`x` is false (zero)
`x == y`	`x` is equal to `y`
`x != y`	`x` is not equal to `y`
`x > y`	`x` is greater than `y`
`x >= y`	`x` is greater than or equal to `y`
`x < y`	`x` is less than `y`
`x <= y`	`x` is less than or equal to `y`
`x || y`	`x` is true or `y` is true
`x && y`	`x` is true and `y` is true

In the above table `x` and `y` are generic placeholders for any variable or expression. Parentheses can be introduced to avoid confusion, as will be seen in future examples.

5.2 `if` and `else`

The most common form of flow control is that of the `if` command. If a condition is true, the next statement is executed, otherwise it isn't. For instance, presume the user inputted something into the variable `money`. Here is an example that will print out a message, presuming that `money` is greater than 10000:

```
if (money > 10000)
    printf("You sure are rich!\n");
```

If `money` is less than or equal to 10000, no message is printed out. Notice the indentation of the `printf()` statement. Remember, C is whitespace ignorant, but tabbing this over helps emphasize that this code is only executed if a certain condition is met. Now, what if we want to execute multiple statements if a condition is true? We can declare a block which will allow us to place multiple statements together and treat them as one. Remember, a block is denoted by the braces, { and }, for the beginning and end of the block, respectively. See this example in which multiple statements are executed when the condition is true:

```
if (money > 10000)
{
    printf("Let me borrow some!\n");
    money-=1000;
}
```

Once again, notice the coding style: braces are aligned with each other and with the `if` command, and the statements inside are tabbed over. If we wish to execute an alternate piece of code, if the condition is not true, we can do that using `else`, as such:

```
if (money > 10000)
{
    printf("Let me borrow some!\n");
    money-=1000;
}
else
{
    printf("Here's a donation!\n");
    money+=500;
}
```

Furthermore, we can chain `if` and `else` together, for multiple conditions, of which only one is executed:

```
if (money > 10000)
{
    printf("Let me borrow some!\n");
    money-=1000;
}
else if (money < 0)
{
    printf("Here's a donation!\n");
    money+=500;
}
else
    printf("You get nothing from me!\n");
```

5.3 switch, case, and break

`switch` and `case` can be used in the same way as `if` and `else`, however we can only test for equality, and due to that nature it is useful primarily when we have a finite set of pre-determined inputs such as a menu, where we know the input will be a selection from the menu. `switch` and `case` only work with `int` and `char` data types. See this example of `switch` and `case`:

```
switch(varname)
{
    case 1 :  printf("The variable was 1\n");
              break;

    case 5 :  printf("The variable was 5\n");
              break;

    default : printf("The variable was something else\n");
}
```

In the above example, if `varname` is equal to 1, the corresponding set of `case` statements for 1 is executed. If `varname` is equal to 5, the corresponding set of `case` statements for 5 is executed. If `varname` is not equal to 1 or 5, then the code by `default` set is executed. This works exactly like a sequence of `if` and `else` commands, however note some syntactical differences: we need braces after `switch` but not after each `case`. So, how does one know where code for one `case` command ends and the next begins? The `default` set of statements will always execute, which is why we place `break` commands at the end of each set of `case` statements – the `break` statement will immediately stop the `switch` block. `case` statements can also be chained, as such:

```
switch(varname)
{
    case 'a' : case 'A' : printf("You typed a or A\n");
                          break;

    default:              printf("You didn't enter in a or A\n");
}
```

5.4 `while`

The `while` command will keep executing a statement or sequence of statements while the condition is true, as such:

```
while (money > 0)
{
    printf("You have $%d, I'm taking $100!\n", money);
    money-=100;
}
```

As soon as the condition is no longer true, the `while` loop stops executing. If the condition isn't true to begin with, the code inside the loop never executes.

5.5 `do/while`

The `do/while` loop is similar to the `while` loop, except that the code inside is always executed at least once, because the condition is checked at the end. Here's an example:

```
do
{
    printf("You have $%d, I'm taking $100!\n", money);
    money-=100;
} while (money > 0);
```

Now note the difference between this and the example in the previous section: if `money` is negative to begin with, the code is executed, but only once.

5.6 `for`

Although all loops are equivalent (any code written with one loop construct can be written with any other), the `for` loop is meant to be used for iteration – that is, doing a loop a specific number of times. The format of the loop is `for (initialization; condition; iterator)`. The initialization is done once, at the start of the loop, then the condition is checked. If the condition is true, the loop executes, then the iterator is executed, and the condition checked again, etc. This is best illustrated with an example:

```
int i;

for(i=0; i<20; i++)
    printf("%d\n",i);
```

The initialization sets `i` equal to 0. Then the condition is checked – is `i` less than 20? Yes, it's 0. The `printf()` statement executes, and then `i` is incremented to 1. Then the condition is checked again. Is `i` less than 20? Yes, it's 1. And so on.

The loop variable is typically called `i`, `j`, etc., although it can be anything. The iterator can be anything. In the above example if we only wanted to print out even numbers, we could change the iterator from `i++` to `i+=2`.

5.7 Ternary Operator

The ternary operator, '`?:`', functions in much the same way as the `if` and `else` commands. The format is `condition ? statement1 : statement2`. If the condition is true, then `statement1` is returned, otherwise `statement2` is. The ternary operator isn't frequently used, but here is an example of an appropriate use:

```
printf("I have %d child%s\n", kids,
       (kids > 1 || kids == 0) ? "ren." : ".");
```

In the above example, if `kids` is greater than 1 or equal to 0, then "child" is pluralized, however if `kids` is equal to 1, then it is not pluralized.

5.8 Nested Loops

A loop is nested when it is contained within another loop. Nesting loops is common in programs, and nesting loops can be very powerful. Here is an example piece of code that will list, in order, all 3 letter combinations of the letters A through Z:

```
int i, j, k;

for(i=0; i<26; i++)
    for(j=0; j<26; j++)
        for(k=0; k<26; k++)
            printf("%c%c%c\n", 'A'+i, 'A'+j, 'A'+k);
```

5.9 Infinite Loops

Infinite loops are loops that run forever. An undesired infinite loop is the bane of programmers, but some infinite loops are intended. Here is a piece of code that will not stop until the user types in the number 65 and hits enter.

```
int i;
char buffer[20];

while (1)
{
    printf("Enter in a number: ");
    fgets(buffer, sizeof(buffer), stdin);
    sscanf(buffer, "%d", &i);

    if (i==65)
        break;
}
```

Note that it would be easy to write the loop without the infinite part. We use `break` here to exit the loop to illustrate that the command can be used to exit a loop. If one desires to stop the current iteration of the loop, and go to the next iteration, one can use the command `continue;`. In a `for` loop, `continue;` will cause the iterator to be run and the condition to be checked. Infinite loops of each type are as follows:

```
while (1)
{
    ...
}

do
{
    ...
} while(1);

for(;;)
{
    ...
}
```

Chapter 6

Functions

Creating callable functions is a large part of C programming. Being able to reuse the same piece of code over and over, and being able to encapsulate these pieces is important. Also, once the behavior of a function is known, that code can be passed to another programmer and they are able to use it without caring about its implementation (e.g. just like you use the `printf()` function without knowning exactly what code is contained inside of it).

6.1 Function Basics

At the very basic level, a function takes some number of arguments and returns exactly one value. See this example where a function returns the sum of two numbers, and the main body in which it is called:

```
/* This function returns the sum of two numbers */
int sum(int x, int y)
{
    return(x+y);
}

int main()
{
    int a, b;

    b = 5;

    a = sum(3,b);

    return(0);
}
```

The function `sum()` takes as arguments two integers, which are called `x` and `y`, locally (inside that function only), and it returns an integer. See in the main body how 3 and `b` are passed to the function. 3 is passed into `x` and the value of `b` is passed into `y`. This is called **passing by value**. The function then returns a value back to the main body. Since we set `a` equal to the returned value, `a` is equal to 8.

Functions can be called from other functions, and can have no return values at all. Function names have the same restriction as variable names, and function names cannot be the same as any variable name. In the case where there are no return values, we use the keyword `void`, and no return value, as such:

```
/* This function prints out a number */
void printstuff(int a)
{
    printf("Hey, check out the value of a: %d\n", a);

    return;
}

int main()
{
    printstuff(6);
    printstuff(99);

    return(0);
}
```

Note that the functions went before the main body. This is a matter of coding style, but the ramifications of placing functions after the main body will be addressed in the next section.

6.2 Function Prototypes

Function prototypes are syntactical definitions of functions. They serve two purposes, the first of which is related to good design, the second of which is the fact that they are absolutely necessary in some cases.

Function prototypes are exactly like the first line of the function implementation, but do not contain variable names[1]. Function prototypes typically go at the top of the code, before `typedef` and `enum` statements, and `struct` definitions. See this example:

[1]Technically, variable names can be put in function prototypes but this is not a good coding style, because if a variable name is changed, it would have to be changed in the function prototype as well.

```
int sum(int, int);
void printstuff(int);

/* This function returns the sum of two numbers */
int sum(int x, int y)
{
    return(x+y);
}

/* This function prints out a number */
void printstuff(int a)
{
    printf("Hey, check out the value of a: %d\n", a);

    return;
}
```

Note that in the case of more than one function prototype, function prototype order corresponds to the same order as the functions are laid out in the source code. Also, semicolons go after each function prototype. In code with many dozens of functions, it helps to have a list of function prototypes, as sort of a table of contents.

There are some situations where functions are required to have prototypes. Any time a function is called in the code before it is defined, there must be a prototype. For instance, if we place all function bodies after the main body. Because the compiler compiles the code in a sequential manner (from top to bottom), it must have a definition of a function before it is called. This is typically accomplished by placing all function bodies before the main body, however, consider this example, where one function calls another, and vice versa:

```
void a(int);
void b(int);

void a(int x)
{
    if (x>0)
        b();

    return;
}

void b(int x)
{
    a(x-10);

    return;
}
```

Although this is somewhat of a silly example, and in actuality one function calling another function that calls the original function rarely happens, it does illustrate the point that there are some situations where function prototypes are absolutely necessary.

6.3 Scope

The **scope** of a variable describes when and where it is active and can be used. When using functions it is important to note that variables in the function, whether they were defined in the function format, or declared within the function itself, those variables can only be used in that function. When that function is over, the variable contents are erased. When inside of a function, variables from other functions or the main body can not be used. If a variable is needed, it should be passed by value into the function when the function is called.

Variable names can be reused in different functions, however even if the variables are the same name, the values they hold aren't. See this example:

```
void fname(int x)
{
    x = x + 5;

    printf("x in fname = %d\n", x);

    return;
}

void fname2()
{
    int x=8;

    printf("x in fname2 = %d\n", x);

    return;
}

int main()
{
    int x = 15;

    printf("x in main = %d\n", x);
    fname(x);
    printf("x in main = %d\n", x);
    fname2();
    printf("x in main = %d\n", x);

    return(0);
}
```

This code will output the following:

```
x in main = 15
x in fname = 20
x in main = 15
x in fname2 = 8
x in main = 15
```

6.4 Static Variables

If we desire to keep a value in a function resident in memory after the function is compled, we can use the `static` variable type qualifier. A `static` variable can be initialized once and only once. Here is an example where a function keeps track of how many times it has been called:

```
void f()
{
    static int called=1;

    printf("This function has been called %d times\n", called++);

    return;
}
```

The expected result happens when the function is called over and over again. The variable is incremented, and the value stays in memory. However, this variable can only be accessed from within that function – the fact that the variable stays in memory does not mean that any other function or the main body can access it.

6.5 Recursion

Recursion is a powerful way to use functions to tackle certain kinds of problems that refer to themselves. For instance, the mathematical factorial, denoted by the exclamation point, !. $n!$ is defined as $n \cdot (n-1)!$, and $1!$ is defined as 1. So, $4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$. See how the factorial is defined in terms of itself? This indicates that it is recursive in nature.

A recursive function is one that calls itself. Obviously if it calls itself, it will never stop, so we have to code in a way to make it stop. In the case of the factorial function, it stops when we reach 1. A function to calculate a factorial would look like this:

```
int factorial(int n)
{
    if (n==1)
        return(1);
    else
        return(n*factorial(n-1));
}
```

The disadvantage to recursion is that it can take a lot of memory, and the management of that memory can take a lot of time. See that in order for `factorial(4)` to be finished, we must first calculate `factorial(3)`, and for that to be calculated, we must first calculate `factorial(2)`, and for that to be calculated, we must first calculate `factorial(1)` (which evaluates to 1). So, at one point in this calculation, four functions are active in memory, and only when the last one finishes, can the rest finish. For a larger calculation, like 40!, this can be cumbersome and take a lot of time and memory.

It is very often that we seek an iterative solution to recursive problems. This can be very difficult to do at times, however in the case of the factorial problem, it is simple, although it doesn't look as ”elegant” as the original recursive form:

```
int factorial(int n)
{
    int value=1;

    do
    {
        value*=n;
    } while (--n>0);

    return(value);
}
```

Chapter 7

Pointers

Pointers are a method of direct memory access, and are perhaps the most difficult part of C to master. However, pointers are very powerful, and are the reason why C is the most popular programming language in the world.

7.1 Pointer Basics

Pointers are created by placing an asterisk before the variable name in a variable declaration, as such:

```
int *ptr;
```

Now, once we have created this integer pointer, we can have it point at an integer. What this means is that the pointer will contain the memory address of an integer variable. When discussing variables, we can deal with the value of the variable, and the memory address of the variable. When discussing pointers, we can deal with the content of the pointer (the memory address of some variable), and the value of what the pointer is pointing to.

To access the value of what the pointer is pointing at, we **dereference** it using the asterisk operator, '`*`'. To get the value of the memory address of a variable, we use the the ampersand operator, '`&`'. See this example, where an integer pointer is pointed to an integer, and values are modified and printed out:

```
    int *ptr, x=5;

    ptr = &x;

    printf("*ptr = %d, x = %d\n", *ptr, x);

    *ptr = 8;

    printf("*ptr = %d, x = %d\n", *ptr, x);

    x = 15;

    printf("*ptr = %d, x = %d\n", *ptr, x);
```

This code will produce the output:

```
*ptr = 5, x = 5
*ptr = 8, x = 8
*ptr = 15, x = 15
```

Notice how the pointer points to `x` and that dereferencing the pointer produces the value of `x`. `x` and the dereferenced value of the pointer are the same thing.

7.2 Pointers and Functions

The use of pointers is most easily seen when using them with functions. Remember that one of the major limitations of functions is that you can only get one value back from them. When using pointers, we can directly access memory, and if we change it in the function, it stays changed throughout the entire program. Here is a simple example of a function that swaps the contents of two variables, with function prototype, and how it would be called from the main body:

```
void swap(int *, int *);

void swap(int *ptrx, int *ptry)
{
    int temp = *ptrx;

    *ptrx = *ptry;
    *ptry = temp;

    return;
}

int main()
{
    int x=5, y=6;

    swap(&x, &y);

    return(0);
}
```

Notice how the memory addresses are passed to the function, and then in the function they are dereferenced to access the values at those memory addresses. Using pointers with functions, passing memory addresses, is called **passing by reference**.

7.3 Pointer Arithmetic and Arrays

When a pointer points to an element in an array, there are four different ways to access that element. See this example:

```
    int a[2] = { 10, 20 }, *ptra = &a[0];
```

Now that `ptra` has the memory address of the variable `a[0]`, we can access that element in four different ways: `*ptra`, `ptra[0]`, `a[0]`, and `*a`. If we want to access the next element in the array, we can use four methods to do that: `*(pa+1)`, `pa[1]`, `a[1]`, and `*(a+1)`. Note that we have to put the values in parentheses, otherwise it would dereference the value immediately and add one to it, giving the value $10 + 1 = 11$ instead of 20. This also introduces another important fact, by adding 1 to the memory address, C is smart enough to realize that since these are integers, we're going to increment by 4 bytes. If we were using a `double` data type, adding 1 would increase the memory address by 8 bytes. All of this just goes to show that arrays are nothing more than pointers!

7.4 Dynamically Allocated Memory

Although arrays are nothing but pointers, as stated in the last section, there is one huge difference between the two. Array sizes are declared at compile time, and memory for pointers is declared at run time. Up until now, we've only pointed pointers at variables that already exist in memory. A powerful use of pointers is to declare a section of memory during run time and have the pointer point to that.

Suppose we want the user to enter an unknown number of integers. One strategy would be to declare an arbitrarily large size array, suppose an array of 10000 integers. If the user only entered in one integer, this would certainly be wasteful of space, and if the user entered over 10000 we wouldn't be able to store them all. Another strategy is to use dynamically allocated memory. Let's start off by allocating space for one integer. As the user enters more in, we will double the size each time[1].

The method of initially allocating memory for a pointer is via a function in `stdlib.h` called `malloc()`. The format for `malloc()`is `void *malloc(int size)`, where `size` is the number of bytes of memory required. The return value is a `void *`, so it should be cast to whatever pointer type is being used (in our case, `int *`). To get the increase in size needed, we use a function called `realloc()`, also in `stdlib.h`, which has the format `void *realloc(void *ptr, int size)`, where `size` is the desired size of the new memory. Contents of the old memory are preserved. Here is the example of our dynamically resizing pointer to a sequence of numbers:

[1]We do not need to double the size, we could simply allocate 4 more bytes at a time, but the overhead associated with doing this is wasteful.

```
int i, val, cur=0, max=1, *ptr=(int *)malloc(1*sizeof(int));
char buffer[10];

do
{
    if (cur == max) /* Need to resize */
    {
        max*=2;
        ptr = realloc(ptr, max*sizeof(int));
    }

    printf("Enter a number in (0 to quit): ");
    fgets(buffer, sizeof(buffer), stdin);
    sscanf(buffer, "%d", &val);

    *(ptr+cur) = val; /* Place value in memory */

    cur++; /* Value added, increase counter */
} while (val != 0);

printf("cur = %d, max = %d.\n", cur, max);

for(i=0; i<cur; i++)
    printf("Value %d: %d\n", i, *(ptr+i));
```

When finished with memory, one should also use the `free()` function to give the memory back to the operating system. Simply pass the pointer to function, as such: `free(ptr)`.

7.5 Double Pointers

Double pointers can be created, and are of much use particularly with respect to arrays of strings, such as with command line arguments. With double pointers, two sets of memory allocation must occur. Let us repeat the example from the previous section, except with strings instead of integers:

```
int i, cur=0, max=1;
char buffer[1000], **strings=(char **)malloc(1*sizeof(char *));

do
{
    if (cur == max)
    {
        max*=2;
        strings = (char **)realloc(strings, max*sizeof(char *));
    }

    printf("Enter a string (no string to quit): ");
    fgets(buffer, sizeof(buffer), stdin);

    *(strings+cur) = (char *)malloc((strlen(buffer)+1)*sizeof(char));
    sprintf(*(strings+cur), "%s", buffer);

    cur++;
} while (buffer[0] != '\n');

printf("cur = %d, max = %d.\n", cur, max);

for(i=0; i<cur; i++)
    printf("String %d: %s", i, *(strings+i));
```

Chapter 8

Structures

Structures exist in C to allow related data types to be combined into a user-defined data type. Variables of that data type can be declared, and the individual elements of that variable can be accessed and modified the same as any basic data type.

8.1 Structure Basics

Structure definitions should go near the top of the code, after preprocessor directives and any necessary enumerations. A structure has a data type name, and some elements, which are declared like normal variables. Here is an example structure definition:

```
struct movie
{
    char title[20];
    short int year_released;
}
```

Now, there exists a data type called `struct movie`. We can declare variables of that type, as such[1]:

```
    struct movie varname;
```

Then the individual elements of the variable can be accessed as such:

[1]To speed up programming, and make code less confusing, use a `typedef`

```
    printf("Enter a title in: ");
    fgets(varname.title, sizeof(varname.title), stdin);
    varname.title[strlen(varname.title)-1] = NULL;

    varname.year_released = 2004;
```

To access elements of the `struct` we use a decimal point. `varname` is just like any other variable, and `struct movie` is just like any other data type. We can even copy one variable to another by using the equal operator, '`=`'. However, we cannot compare two structures using any of the flow control operators discussed previously. We can use them on the individual elements, but not on the `struct` as a whole.

8.2 Structures, Pointers, and Functions

Since structures can become pretty sizeable, we do not want to pass a `struct` by value to a function – doing so would create a copy of it in memory, which is wasteful. Therefore, we should use passing by reference. Here is an example where two elements of a `struct` are swapped:

```
struct point
{
    int x, y;
}

void swap(struct point *pt)
{
    int temp = (*pt).x;
    (*pt).x = (*pt).y;
    (*pt).y = temp;

    return;
}

int main()
{
    struct point p;

    p.x = 6;
    p.y = 10;

    swap(&p);

    return(0);
}
```

This example illustrates the cumbersomeness of using both the asterisk and the decimal point, and having to introduce parentheses to reduce ambiguity. When we use any sort of pointer to a `struct`, we can use instead of the asterisk and the decimal point, the arrow, '`->`'. See this revised version of the `swap()` function:

```
void swap(struct point *pt)
{
    int temp = pt->x;
    pt->x = pt->y;
    pt->y = temp;

    return;
}
```

The arrow operator does the job of both dereferencing the pointer and accessing the element.

8.3 Linked Lists

Linked lists are a data structure built of structures. A linked list contains some number of nodes, such that there is a beginning node, and each node points to another distinct node – think of it as a chain, where each link in the chain is connected to the next one. In order to do this, we need to have a `struct` pointer inside of a `struct`. As necessary, we can allocate memory for a new structure, and link it to a current one. This is very similar to the previous examples of reallocating memory, however the big difference is that `realloc()` will attempt to find a contiguous block of memory for use, and with linked lists we allocate enough memory for each `struct` one at a time, so they can be spread out in memory. Here is an example of a linked list of integers:

```
typedef struct stype
{
    int x;
    struct stype *next;
} node;

int main()
{
    node *current=(node *)malloc(sizeof(node)), *front=current;
    char buffer[10];

    do
    {
        printf("Enter a number in (0 to quit): ");
        fgets(buffer, sizeof(buffer), stdin);
        sscanf(buffer, "%d", &(current->x));

        if (current->x==0) /* User entered 0 */
        {
            current->next = NULL;
            current = front; /* Stopping condition */
        }
        else
        {
            current->next = (node *)malloc(sizeof(node));
            current = current->next; /* Move to new element */
        }
    } while (current != front);

    do
    {
        printf("Value entered: %d\n", current->x);
        current = current->next;
    } while (current != NULL);

    return(0);
}
```

Note that we keep setting the `current` pointer to the newly allocated memory, so we need some method of getting back to the very first element, hence the use of the `first` pointer.

8.4 Unions

Unions are not commonly used in programs, but they exist for specialized cases when memory is limited in some way or another. They function much like a `struct`, except that a `union` is only as large as the largest element contained within, and can only hold one element. Here is an example:

```
union utype
{
    int x;
    char c1, c2;
}
```

Now, if a variable is declared, such as `utype u`, then the elements of that `union` can be accessed as `u.x`, `u.c1`, and `u.c2`. The size of the `union` is only 4 bytes, because the largest element of it is an `int`. If a value is stored in `u.x`, the values of `u.c1` and `u.c2` are erased. Likewise, if a value is stored in either of `u.c1` and `u.c2`, the value of the other character and the value of the integer will be erased. One can attempt to read the value of the `int`, but it will not be the previously stored value.

Chapter 9

Files

File access allows us a more permanent storage of data, insomuch as we can put data on to the disk and when our program is finished executing, the data is not lost forever, like when only using main memory. File access also allows us to read in information from files and manipulate it. In many cases we need specific pieces of data from the file, and must write our programs to retrieve and manipulate that data. All file functions are in the header file `stdio.h`.

9.1 File Modes

There are 12 different modes for manipulating files:

Mode	**Meaning**
`r`	text read
`w`	text write (erase previous contents, if any)
`a`	text append (open for writing at end of file)
`r+`	text update (read/write, file must already exist)
`w+`	text update (read/write, file is created)
`a+`	text update (read/write, start at end of file)
`rb`	binary read
`wb`	binary write (erase previous contents, if any)
`ab`	binary append (open for writing at end of file)
`r+b`	binary update (read/write, file must already exist)
`w+b`	binary update (read/write, file is created)
`a+b`	binary update (read/write, start at end of file)

9.2 Opening and Closing a File

To open a file, we use the function `fopen()`. The format for `fopen()` is `FILE *fopen(char *name, char *mode);`. The first argument is a string containing the name of the file to be opened, and the mode is one of the above modes. It returns a file pointer, `FILE *`, which is used to refer to the file in other functions like `fgets()`, `fprintf()`, `fread()`, and `fwrite()`.

Closing a file is accomplished by using `fclose()`. The only argument to `fclose()` is the file pointer of a file that was opened. It is very important to close files for two reasons. The first reason is that if a file is not closed, all of the data may not be written to it[1]. The second reason is that operating systems have limits on how many files can be open at one time. Although this is not usually a concern, opening a few hundred files without closing them may cause a program crash.

9.3 Inputting from a Text File

After opening the file in the text file read mode, "r", we can read the data in the file using the `fgets()` function. When using `fgets()` before, the

[1]This is a consequence of the way different operating systems write data to files

third argument was stdin, which indicated that we were reading from the keyboard. Now instead of stdin we will use our file pointer returned by fopen(). fgets() will read a number of bytes specified by the second argument given to it, or will read until a newline character is encountered – just like reading input from the keyboard. After being read, anything can be done to the data just as normal, including using sscanf() to retrieve variables from the string. Here is a piece of code that reads the contents of a text file named "text" and prints it to the screen:

```
FILE *in;
char buffer[50];

in = fopen("text","r");

while (fgets(buffer,sizeof(buffer),in))
{
    printf("%s",buffer);
}

fclose(in);
```

Note that when there is no more text to read to the file, fgets() will return 0, at which point the while loop will stop. Also, the size of buffer could be decreased to 2, and the code would still function exactly the same. The only difference is that instead of reading 49 characters at a time, the loop would only read 1 character at a time. If the size was decreased to 1, this would never read anything from a file, since the last character of a string is always the NULL character. Now, here's an example where a file with one number on each line is read, and the numbers are added together and printed out:

```
FILE *in;
char buffer[50];
int total=0, current;

in = fopen("input","r");

while (fgets(buffer,sizeof(buffer),in))
{
    sscanf(buffer, "%d", &current);
    total+=current;
}

printf("Total is: %d\n", total);

fclose(in);
```

9.4 Outputting to a Text File

Outputting text to a file is easily accomplished with a function called `fprintf()`. It is only different from `printf()` in that before the format string, we now put the file pointer of a file that was opened in text mode write format, ”w”. Here is an example where the user types something in and it is directly outputted to a file:

```
FILE *out=fopen("output","w");
char buffer[50];

printf("Type something in: ");
fgets(buffer, sizeof(buffer), stdin);

fprintf(out, "%s", buffer);

fclose(out);
```

9.5 Inputting from a Binary File

Inputting from a binary file is accomplished through use of the `fread()` command. The format of `fread()` is `int fread(void *ptr, int size, int nobj, FILE *stream);`. The first argument is the memory address of where the read data should go. The next two arguments define how much data should be read – `nobj` objects each of size `size` – or `nobj`·`size` bytes. The last argument is a file pointer of a file opened in the binary read mode "rb". See this example of reading in an array of 20 integers from a file:

```
FILE *in=fopen("input","rb");
int ary[20];

fread(ary, sizeof(int), 20, in); /* &ary[0] is a valid
                                      substitute for ary */
fclose(in);
```

9.6 Outputting to a Binary File

Outputting to a binary file is done using `fwrite()`. The format of the command is exactly the same as `fread()`, the only difference being the function name change. This time, instead of reading the data from a file, it writes it to the file. See this example, in which an array of 20 integers is written to a file:

```
FILE *out=fopen("output","wb");
int ary[20];

fwrite(ary, sizeof(int), 20, out);

fclose(out);
```

9.7 Text vs. Binary

The choice of text versus binary file format is dependant upon several things, including the file's interaction with various programs, memory constraints, and how much hassle it is to write the code to save and load.

When writing an array, be it a small array of integers, or a large array of structures, binary is almost always preferred. Not only is file I/O easily coded, but the size of the file will be smaller than if outputted using text mode and programmer-set delimiters (e.g. writing out each element of a `struct` and placing a comma between each one).

When trying to write programs that will produce files that are compatible with different programs, it is almost always necessary to use text mode. Producing a comma separated value (CSV) file to import into Microsoft Excel is very easy using the powerful formatting abilities of `fprintf()`. It is often that we write programs to gather certain data from one file and prepare that data for use in a different program. The downside to text mode is that it is almost always larger than storing the file in a raw binary mode.

Also, one must be mindful of how they create things such as a `struct` when using file I/O, because pointers cannot be stored to a binary file. Therefore, any strings inside of a `struct` must be character arrays, and not character pointers.

Chapter 10

ctype.h

The `ctype.h` header file contains functions that are used to gather information about, and manipulate, characters. Here is a listing of those functions that assist in logical evaluation of a character – all functions return an `int`:

Format	**Returns true if `c` is a(n)...**
`isalnum(char c)`	alphanumeric (0-9, 'a'-'z', 'A'-'Z')
`isalpha(char c)`	alphabetic ('a'-'z', 'A'-'Z')
`isascii(char c)`	7 bit ASCII character (0 to 127 in ASCII table)
`isblank(char c)`	blank character (space or tab)
`iscntrl(char c)`	control character
`isdigit(char c)`	digit
`isgraph(char c)`	printable character (except space)
`islower(char c)`	lower-case alphabetic
`isprint(char c)`	printable character (including space)
`ispunct(char c)`	printable character (not space or alphanumeric)
`isspace(char c)`	white-space character (space, tab, newline)
`isupper(char c)`	upper-case alphabetic ('a'-'z')
`isxdigit(char c)`	hexadecimal digit (0-9, 'a'-'f', 'A'-'F')

Remember that all the above functions use the ASCII values. This is not typically a concern, but it must be considered in the case of `isdigit()`, as `isdigit(3);` returns false (3 is an ASCII control character), and `isdigit('3');` will return true.

In addition to the above functions that are used in control flow situations (e.g. for `if` conditions, for `while` conditions, etc.), there are two functions in `ctype.h` that allow for letter transformations between upper and lower-case:

Format	**What the function does**
`char tolower(char c)`	if `c` is an upper-case letter ('A'-'Z') it returns the lower-case equivalent of the character, otherwise it returns the original character.
`char toupper(char c)`	If `c` is a lower-case letter ('a'-'z') it returns the upper-case equivalent of the character, otherwise it returns the original character.

Chapter 11

`string.h`

The functions in `string.h` are used to gather information about and manpulate strings. Here is a list of the functions:

Format	**What the function does**
`char *strncpy(char *s, const char *t, int n)`	Copies `n` characters from `t` to `s` and returns a pointer to `s`.
`int strcmp(const char *s, const char *t)`	If `s>t`, alphabetically, it returns a positive integer, if they're equal, it returns 0, otherwise it returns a negative.
`char *strchr(const char *s, char c)`	Returns a pointer to the first occurrence of `c` in `s`, or `NULL` if `c` is not in `s`.
`char *strrchr(const char *s, char c)`	Returns a pointer to the last occurrence of `c` in `s` or `NULL` if `c` is not in `s`.
`char *strstr(const char *s, const char *t)`	Returns a pointer to the first occurrence of `t` in `s`, or `NULL` if `t` is not in `s`.
`int strlen(const char *s)`	Returns the length of string `s`.
`char *strtok(const char *s, const char *t)`	Finds tokens in `s` delimited by characters in `t`. See next section.

It should be noted that the difference between `char *` and `const char *` is important. Remember that for most cases, arrays are nothing more than pointers. A `const char *` indicates that both a character array and a character pointer. `char *` on the other hand indicates that the value must be a character pointer. Only in the case of `strncpy()` is this a concern.

The functions that return pointers to strings may return pointers inside of that string (such as `strchr()` does) or it may return newly allocated pointers outside of that string (such as `strtok()` does).

11.1 String Tokenization

String tokenization is frequently used to parse strings. When certain information is desired from a string, we can use `strtok()` to pick apart the string. Strings can be broken up using delimiters. Delimiters are used to separate text all the time. The space acts as a delimiter between words.

`strtok()` takes as its first argument a string which is to be broken into tokens (groups of characters separated by delimiters), and as its second argument a set of delimiters. `strtok()` then looks through the string and copies out the first token, and returns a pointer to it. Note that part of the string is copied out and put in another location in memory, so the original string is not altered. When seeking future tokens, the first argument to `strtok()` should be `NULL`. The reason for this is that if the same arguments were passed to `strtok()`, the first token would keep getting returned over and over. `strtok()` contains a `static` variable, and when `NULL` passed to the function, it keeps searching through the original string because it stored the stopping character of the first token in the `static` variable. When there are no more tokens in the string, `strtok()` returns a `NULL` value. Here is an example of breaking a string up by a space and a comma:

```
    char *token;
    char string[] = "This is a te,st!";

    token = strtok(string, " ,"); /* First token */

    while (token) /* NULL if no more tokens */
    {
        printf("Token: %s\n", token);
        token = strtok(NULL, " ,"); /* Get next token */
    }
```

The output of this program will be:

```
Token: This
Token: is
Token: a
Token: te
Token: st!
```

Chapter 12

math.h

The functions in `math.h` are common mathematical functions. It may be necessary to link libraries by compiling using `gcc source.c -lm`. Here is a list of major functions in `math.h` – all functions return a `double`[1]:

Format	**What it does**
`sin(double x)`	Returns the sine of `x`
`cos(double x)`	Returns the cosine of `x`
`tan(double x)`	Returns the tangent of `x`
`asin(double x)`	Returns the arcsine of `x`
`acos(double x)`	Returns the arccosine of `x`
`atan(double x)`	Returns the arctangent of `x`
`sinh(double x)`	Returns the hyperbolic sine of `x`
`cosh(double x)`	Returns the hyperbolic cosine of `x`
`tanh(double x)`	Returns the hyperbolic tangent of `x`
`exp(double x)`	Returns the value $e^{\mathtt{x}}$ ($e = 2.71828\ldots$)
`log(double x)`	Returns the natural logarithm of `x`
`log10(double x)`	Returns the logarithm (base 10) of `x`
`pow(double x, double y)`	Returns the value $\mathtt{x}^{\mathtt{y}}$
`sqrt(double x)`	Returns the value $\sqrt{\mathtt{x}}$
`ceil(double x)`	Returns the lowest integer higher than `x`
`floor(double x)`	Returns the highest integer lower than `x`
`fabs(double x)`	Returns the absolute value of `x`

[1]All trigonometric functions utilize radians, not degrees.

Chapter 13

stdlib.h

The header file `stdlib.h` contains many miscellaneous functions:

Format	**What it does**
`int atoi(const char *s)`	Returns the `int` value of the string
`double atof(const char *s)`	Returns the `double` value of the string
`long int atol(const char *s)`	Returns the `long int` value of the string
`int rand()`	Returns a pseudorandom number between 0 and `RAND_MAX`
`void srand(int seed)`	Seeds the pseudorandom number generator
`void *malloc(int size)`	Returns a pointer to `size` bytes of free space
`void *calloc(int nobj, int size)`	Returns a pointer to `size`·`nobj` bytes of free space
`void *realloc(void *ptr, int size)`	Returns a pointer to `size` bytes of memory if it is larger than the previously allocated memory only. Contents are preserved.
`void free(void *ptr)`	Frees up previously allocated memory
`int abs(int n)`	Returns the absolute value of `n`

Chapter 14

Miscellany

This chapter contains miscellaneous topics which aren't large enough to, or contain information important to warrant their own chapter.

14.1 Coding Style

Coding style is of the utmost importance in programming, for many reasons. The first is that getting in the habit of writing in a good coding style will ultimately make you a faster coder that produces more error-free code. The second is that if anyone else looks at the code, they want to be able to easily understand what the code does, both in a general level and on a line-by-line basis. Coding style consists of, but is not limited to, good spacing and good commenting.

Good spacing means tabbing over at the beginning of a block, tabbing over inside of a loop, and tabbing back to complete a block or a loop. In general, all open and close braces should line up vertically, and no brace should be within one character (in any direction) of another.

Good commenting means that there should be comments that describe the program, each function, and each "difficult" piece of code (perhaps something more complicated such as string tokenizing). Comments should also include identifying information about the author, when it was programmed, and documenting any code that was taken from a source[1].

Every coder develops their own coding style and no two are exactly alike. Be sure to use a coding style that is both comfortable and familiar to yourself,

[1]In this case, permission should be given by the original author to reproduce that code.

but also be considerate of others who may be using the code in any capacity in the future.

14.2 `typedef` and `enum`

`typedef` can be used to rename a data type, be it a primitive such as `int` or `char`, or something more complicated such as a `struct`. A `typedef` should ideally occur near the top of the program, after preprocessor directives and `struct` definitions. The format of `typedef` is `typedef oldname newname;`. See the following example:

```
typedef int dimension;

int main()
{
    dimension length, width, height;

    ...
```

In the above example, `length`, `width`, and `height` are declared as type `dimension`, which is really an `int`. In the following example, we `typedef` a `struct`:

```
struct point
{
    int x, y;
}

typedef struct point pt;
```

Now, instead of declaring variables using `struct point`, we can use `pt` to declare them. We can also combine the `struct` definition with the `typedef`, as such:

```
typedef struct point
{
    int x, y;
} pt;
```

We can also remove the `point` from the above code, however if we do so we will no longer be able to use `struct point` to declare or use the data type, and will be forced to use `pt`. This is not a bad thing, because in general we would only use one method of talking about the `struct` data type. So, the ideal way to declare a `struct` is as follows:

```
typedef struct
{
    int x, y;
} pt;
```

The `enum` command can be used to enumerate a set of strings. We often enumerate the months (e.g. January = 1, February = 2, etc.). An `enum` should go at the top of the code, in the same place as `struct` definitions and `typedef` definitions. `enum` has the format `enum setname { name1, name2, ...}`. By default `name1` will have an integer value of 0, `name2` will have an integer value of 1, etc. Values can be set at any time, at which point future names will keep increasing from the previous value. See this example, which enumerates the first three months as 1, 2, and 3:

```
enum month { JAN=1, FEB, MAR };
```

The set name can be used as a defined type of `int`, so functions can return them, and variables can be declared with them, as such:

```
month nextmonth(month m)
{
    if (m == JAN)
        return(FEB);
    else if (m == FEB)
        return(MAR);
    else
        return(JAN);
}
```

14.3 Typecasting

Typecasting can be used to force a value into a different data type. This is mostly necessary to convert or compare real numbers (`float` or `double`) to strictly integer values (`int` or `char`), or vice versa, because of how the bits are stored. Typecasting is done by putting the data type in parentheses prior to the value (or variable). Typecasting will truncate a variable, for instance `2000` is equal to `(int)2000.99`.

14.4 Preprocessor Directive `#define`

The preprocessor directive `#define` can be used for multiple things. In its most simple form, it can be used to create direct text substitions in code. Remember that preprocessor directives are carried out before code is compiled. Anything defined with `#define` will be replaced in the code before the code is compiled. `#define` directives should be placed in the code right after the `#include` directives. Here is an example that uses `#define` to define a string array size:

```
#define BUFFERSIZE 20

int main()
{
    char name[BUFFERSIZE], address[BUFFERSIZE];

    ...
```

This code, after the preprocessor directives are carried out, becomes:

```
int main()
{
    char name[20], address[20];

    ...
```

The use of `#define` in this instance is obvious – we can easily define a value and change it at one place later, thereby affecting all of the instances of it. The use here could easily be done using `const`. We also maintain the use of all capital letters. `#define` can be used to create macros – almost like miniature functions – with arguments. Suppose there are multiple conditions upon which an error occurs, and when one occurs we desire to close all files, print out some message, and quit the program. A macro could easily be created, as such:

```
#include <stdio.h>

#define ERROR(s) printf(s);fclose(out);return(0)

int main()
{
    FILE *out;

    ...

    if(somecondition)
    {
        ERROR("File error!\n");
    }
    else if (someothercondition)
    {
        ERROR("Invalid input!\n");
    }

    ...
```

Which, after processing, the main body becomes:

```
int main()
{
    FILE *out;

    ...

    if(somecondition)
    {
        printf("File error!\n");fclose(out);return(0);
    }
    else if (someothercondition)
    {
        printf("Invalid input!\n");fclose(out);return(0);
    }

    ...
```

This example illustrates two important points. The first is that even though `ERROR()` in the first code would seem as if it didn't need the braces surrounding it, when expanded we see that since it is three commands we do need the braces. Also note that we do not put a semicolon after the `return(0)` in the definition because the semicolon is placed in the code[2]. Much like functions can use multiple variables, so can macros – just separate arguments with a space.

14.5 Command Line Arguments

Command line arguments allow your program to take input at the command line, rather than prompting the user for data within the program. To use command line arguments, an alternate definition of the main body must be used. Rather than using the `int main()` definition, we use this one:

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

[2]Whether or not the semicolon is placed in the macro definition or in the code is irrelevant, as long as one semicolon is there. Usually we place it in the code to preserve the look and feel of the code.

`argc` is an integer representing the total number of command line arguments given when executing the program, and `argv` is an array of strings, one string for each argument. The arguments are delimited by spaces on the command line, and the executing command for the program is always stored in `argv[0]`. For example, if this was the way the the program was executed:

```
./a.out Hello world, welcome!
```

Then `argc` would be equal to 4, `argv[0]` would contain `./a.out`, `argv[1]` would contain `Hello`, `argv[2]` would contain `world,`, and `argv[3]` would contain `welcome!`.

Iterating through a list of command line arguments is simple, just loop from 0 to `argc`-1, inclusive. In all cases, the strings of the command line arguments are from `argc[0]` to `argv[argc-1]`. `argv[argc]` never exists, and `argc` is always greater than or equal to 1.

14.6 Bit Fields

Variables based on primitive data types can be used for other purposes than holding values of that data type. As we have seen, data types consist of some number of bits. We can individually toggle bits to store data. Let us say we need some way to represent whether or not it is raining outside or not – we only need one bit to store this data. If we need to store other information like this, we can easily do so in a `char`. The most relevant operations associated with using a bit to store a piece of data is how to set, check, and check a bit.

14.6.1 Accessing One Bit

Before learning of the operations associated with bit fiddling, let us create a quick method of referring to a single bit. In a `char` there are 8 bits, so we can use any one of them. We will use the method of left shifting a single bit into the proper position. With the number 1, the `char` representation is `00000001`. If we then shift this bit left two times we get `00000100`. The operator in C for doing this is '<<'. So, `1<<2` is the shifting of a bit left twice. Typically to keep from writing this over and over, and to give a name to the bit, indicating what it represents, we will use a `#define` preprocessor directive, as such:

```
#define RAINING 1<<2
```

14.6.2 Setting a Bit

Setting a bit is accomplished through use of the bitwise OR operation, '`|`'. The truth table for bitwise OR is as follows:

a	b	a|b
0	0	0
0	1	1
1	0	1
1	1	1

This table illustrates two things. The first is that if we bitwise OR a bit with 1, the resulting bit is 1. The second is that if we bitwise OR a bit with 0, the resulting bit is whatever it was originally. So, in bitwise OR we have the ability to set a bit without disturbing the contents of other bits. We can set a bit in the following manner:

```
char bitfield=0;

bitfield = bitfield | RAINING;
```

Or, we can compress it using an assignment operator, as such:

```
char bitfield=0;

bitfield |= RAINING;
```

14.6.3 Checking a Bit

Checking a bit is accomplished through use of the bitwise AND operation, '`&`'. The truth table for bitwise AND is as follows:

a	b	a&b
0	0	0
0	1	0
1	0	0
1	1	1

With the bitwise AND, we are able to check a bit because if we bitwise AND a bit with 1, the resulting bit is whatever it was originally. Why couldn't we just use bitwise OR with 0 for this? Because we also need to make all the other bits go to 0! Consider a bit field with bits `10111010`. To check the 5th bit, we AND with `00010000`, thereby making the result `00010000`. Now, if that bit had not been set originally, the value after ANDing would have been 0. A non-zero value indicates true, and a zero indicates false – perfect for our methods of flow control! See this example:

```
if (bitfield & RAINING)
    printf("It's raining!\n");
else
    printf("It's not raining!\n");
```

14.6.4 Clearing a Bit

Clearing a bit is accomplished through the use of the bitwise AND operator in conjunction with a bitwise NOT operator. Note that in the AND truth table, if anything is ANDed with 0, the result is 0, otherwise the previous contents are preserved. Therefore, if we wanted to reset bit 4 of a bit field, we would need only to bitwise AND with `11110111`. An easy way to create such a group of bits is to invert all the bits in `00001000`. Inverting is accomplished through the bitwise NOT operator, '~'. See this example:

```
bitfield &= ~RAINING;
```

14.7 Pseudorandom Number Generation

Pseudorandom number generation can be accomplished using a few functions from different header files. `srand()` and `rand()` are used from the `stdlib.h` header file, and `time()`is used from the `time.h` header file.

Calling `rand()` will just return a pseudorandom number between 0 and `RAND_MAX`[3], calling it again will return a different number, and so on. However, if a program is written that will call `rand()` five times, and that program itself is executed multiple times, then the same five numbers will be returned by `rand()` each time! This may be the desired behavior – the same set of ”random” numbers each time – but in most situations it is not.

So, we need a way to randomize the pool of random numbers. Calling `srand()` with a single integer as the argument will **seed** the numbers differently. However, if we seed it with the same number time after time, we will still get the same sequence of numbers.

So, we need only to generate a different number for the seed each time. This is easily accomplished by making a call to `time()` from the `time.h` header file. `time(0);` or `time(NULL);` will return the current number of seconds since the epoch[4]. Giving that value to `srand()` will guarantee a different seed for each execution of the program, presuming the program is not executed twice in the same second. See this example:

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

int main()
{
    srand(time(0));

    for(;;)
        printf("Random number: %d\n", rand());

    return(0);
}
```

[3]Look in `limits.h`
[4]January 1, 1970

14.8 Partial ASCII Table

Decimal	**Char**	**Decimal**	**Char**	**Decimal**	**Char**
0	NULL	60	<	95	_
8	backspace	61	=	96	‘
9	tab	62	>	97	a
10	new line	63	?	98	b
12	new page	64	@	99	c
13	carriage return	65	A	100	d
27	escape	66	B	101	e
32	space	67	C	102	f
33	!	68	D	103	g
34	"	69	E	104	h
35	#	70	F	105	i
36	$	71	G	106	j
37	%	72	H	107	k
38	&	73	I	108	l
39	’	74	J	109	m
40	(	75	K	110	n
41	)	76	L	111	o
42	*	77	M	112	p
43	+	78	N	113	q
44	,	79	O	114	r
45	-	80	P	115	s
46	.	81	Q	116	t
47	/	82	R	117	u
48	0	83	S	118	v
49	1	84	T	119	w
50	2	85	U	120	x
51	3	86	V	121	y
52	4	87	W	122	z
53	5	88	X	123	{
54	6	89	Y	124	|
55	7	90	Z	125	}
56	8	91	[	126	˜
57	9	92	\	127	delete
58	:	93	]		
59	;	94	ˆ		