Lecture Notes for Fundamentals of Computing, Spring 2004

Chris Monico

e-mail: cmonico@math.ttu.edu
Department of Mathematics and Statistics
Texas Tech University

Draft of April 26, 2004
Contents

1 Introduction to C

1.1 Compiled languages ............................................... 1
1.2 Structure of the C language ....................................... 2
1.3 Compiling ................................................................. 4
1.4 C syntax by example ................................................. 5
1.5 Arithmetic operators and precedence ............................... 9
1.6 Using printf and scanf for I/O .................................... 12
1.7 Recursion and the Euclidean Algorithm ............................ 16
1.8 Some math functions ................................................ 18
1.9 Built-in data types .................................................... 21
1.10 Pointers and arrays ................................................ 22
1.11 Structures and more on pointers .................................. 27
1.12 Keywords in C .......................................................... 33
1.13 Casting and dynamic memory allocation .......................... 44

2 Practicing C

2.1 The Mandelbrot set .................................................... 47
2.2 Debugging a C program .............................................. 49
      2.2.1 Example 1 ....................................................... 49
      2.2.2 Example 2 ....................................................... 51
CONTENTS

2.2.3 Exercises ................................................................. 52
2.3 Conway’s Game of Life ................................................... 55

3 Abstract data types ......................................................... 59
  3.1 Arrays ................................................................. 59
  3.2 Queues ............................................................... 62
  3.3 Stacks ................................................................. 63
  3.4 Trees ................................................................. 63

4 Elementary cryptanalysis .................................................. 65
  4.1 The Caesar shift cipher ............................................... 65
  4.2 The Vigenère cipher .................................................. 66
  4.3 Monoalphabetic homophonic substitution ......................... 71
  4.4 Permutation ciphers ................................................. 73

5 Efficient algorithms ....................................................... 79
  5.1 Quicksort ............................................................ 79

A Solutions to selected exercises ......................................... 83

B Cygwin tips & tricks ...................................................... 87
  B.1 Preface ............................................................... 87
  B.2 Configuration ........................................................ 88
  B.3 Compiling ............................................................ 88
  B.4 Quick reference ...................................................... 89
Chapter 1

Introduction to C

1.1 Compiled languages

Why do we need computer languages? Strictly speaking, we really do not need them to operate a computer. A CPU does already come equipped with a set of instructions for performing operations and they can be accessed directly. While this is sometimes necessary (not for us, thankfully), it is cumbersome to access these very low-level instructions. To add insult to injury, there are not terribly many of them - typically, there are instructions for doing arithmetic on some built-in machine types (e.g., 32-bit integers and floating-point numbers with some fixed precision), logical operations like bitwise AND and OR, instructions for moving bytes around and instructions for jumping execution from one part of a program to another. There are enough of these low-level machine instructions to get tasks done. The difficulty is simply that:

- We may need alot of them to accomplish even a small task.
- They can be difficult to read, which makes it difficult to fix a program when something is not working quite right.

Symbolically, some of this low-level code might look like:

```
movl %edx, %ebx
movl 4(%esi), %eax
mull (%edi)
addl %eax, %ebx
```

This is not the actual machine-code itself, but rather assembly language which is essentially a human-readable version of it. Here, the machine-level instructions are on the left and their operands are on the right. The point is that each of these lines can be directly translated
into a sequence of bytes which the computer can execute. Probably the only thing clear from this code is that we would prefer not to deal with it!

Compiled language alleviate these problems by allowing us to specify a program in a way that is more robust than these simple machine-level instructions. We specify our program in the high-level language, and the compiler takes care of translating it into the machine-level instructions that the computer understands and can execute.

![Diagram](image)

Figure 1.1: A compiled computer language

C is one such programming language. We write a C program as a human-readable text file, and let the C compiler worry about translating it into code that the CPU can actually execute. The C language was developed in the late 60’s and early 70’s at AT&T. It was descended from the language B, which was itself descended from the language BCPL. Developed in parallel with the UNIX® operating system, UNIX®, Linux and other entire operating systems are still written in C today. Indeed, C is probably the language in most widespread use today. And since imitation is the highest form of flattery, there other modern languages which have modeled themselves after C in one way or another, including Java, C++, C#, and Objective-C.

### 1.2 Structure of the C language

As with any language, spoken or computer, we need to know at least a very little bit about the overall structure before we dive right in.

Remember that source code is the ASCII text that we will write to specify a computer program. To make it easy to organize and re-use source code, C programs are divided into functions. C functions are similar to mathematical functions in that they may take input arguments and give an output argument. But the similarities end there - a C function may take zero or more arguments and generally has a name which is more descriptive than we are used to for mathematical functions. For example, consider the mathematical function $A(r) = \pi r^2$. In C, I would probably specify this function as follows:
1.2. STRUCTURE OF THE C LANGUAGE

Let’s examine the syntax a bit, so we can see what we have here. The first thing to observe is the keyword `float`. This is a built-in data type of the language, for using floating point numbers. Floating point numbers can be used like real numbers, but they are only a fixed-precision approximation of a real number (in fact, the internal machine representation is actually rational). Anyway, the first occurrence of `float` is specifying the return value of this function. Next, `circleArea` is the name of the function we are defining. The parenthesis that follow enclose the list of arguments that the function takes, so we can see that it will take a single variable named `r` of type `float` as input. Finally, the list of instructions themselves which define the function `circleArea` follow the name of the function, and are enclosed within braces `{ `and `}`. This function has very little work to do: it computes `3.14159*r*r` (an approximation of \( \pi r^2 \)) and returns it to the caller.

Anyway, a C program is nothing more than a collection of functions. They may be split across several files, or all in one file. The only restrictions are that:

- Function names must be unique.
- There must be a function named `main`.

The reason for the first is somewhat obvious: if we defined the same function two different ways, how would the compiler know which one we are talking about at a given time? Some languages do allow non-unique names, so long as they can decide from the context which is to be used. But C does not allow this ‘feature’. The reason for the second restriction is equally simple: Since a C program is nothing more than a collection of functions, the compiler must know at which function the program is to begin executing. The inventors of C decided to simply use a canonical name for the function where execution begins. Since it is, in some sense, the main function, they logically decided that it should be called `main`.

For example, the function `circleArea` that is defined above can compute the area of a given circle. But it is not, by itself, a program yet. If it were, what would the output be?

```c
#include <stdio.h>

float circleArea(float r)
{
    return 3.14159*r*r;
}

int main()
```

```c
```
{ float r, a;
    r = 4.2;
    a = circleArea(r);
    printf("A circle of radius %.1f has area %.1f\n", r, a);
    return 0;
}

The first and last statements probably look a bit cryptic. At least it is clear, though, what the program should do. It contains two functions:

1. circleArea which takes a float and returns a float.
2. main which takes no input arguments and returns an int.

It should start at main, where it will call the circleArea function to find out what the area is of a circle whose radius is 4.2. It will then print something about the radius and the area. Exactly what does it print? We will cover this in more detail later, but the output of this program will be:

A circle of radius 4.2000 has area 55.4176

1.3 Compiling

So we have our little program for computing the area of a circle with radius 4.2. How do you know that the output of that code is what I claimed it to be? Well, compile it and check. I will assume that the reader is using the GNU C Compiler (gcc), because it is both good and free. In that case, if the program is contained in the file circlearea.c, I would compile it with

    gcc circlearea.c

If all has gone well and there were no errors, there should be an executable program called a.out. It is called a.out for historical reasons: it is actually the assembler output. Of course, this is no real restriction since we can rename it to anything we’d like. But if we want the compiler to just give the program another name itself, we can tell it to do so:

    gcc -o circlearea circlearea.c

But beware: the -o circlearea option is telling the compiler what I want it to call the executable program that it outputs; if we accidentally used -o circlearea.c, it would overwrite our original source code file!
1.4 C syntax by example

The easiest way to learn a spoken language is probably to dive right in and start speaking it. After all, when teaching a young child to speak, we don’t begin by detailing out the syntactical rules of the language; we begin by example. Once the child can competently speak the language, they learn the specifics and details later. There are a couple of facts which we should state clearly, though, since they are not obvious from looking at examples:

- C is a case-sensitive language. This means that a function named `getarea` is different than a function named `getArea`.

- C is very tolerant of whitespace (or the lack thereof). Spaces, tabs and carriage-returns can generally be used as much or as little as the author desires to help with readability.

With that in mind, we will just start looking at some code to learn a little about the syntax. To make life a little easier, we will start putting comments in the source code. The language itself does allow us to include comments that are for human-benefit only and don’t actually do anything. However, the designers of the language did not feel it was appropriate to reserve a keyword or make a new symbol for this purpose. Instead, they chose a combination of symbols which are individually part of the language already, but which cannot appear together. Namely, /* to indicate the beginning of a comment and */ to indicate the end. So, here is an update version of the `circlearea.c` program.

```c
/* circlearea2.c
   Chris Monico, 2/2/04.
   It is generally good practice to put the name, author and date of a source code file at the top. This makes it easier to tell what is being viewed if it is sent to a printer, for example.

Look at the line immediately following this comment.
#include tells the compiler that we wish to use some functions which are not part of the language itself. Usually, they are part of the standard library of functions. In this case we want to use the printf() function which is described for the compiler in the file stdio.h = standard Input/Output. Almost every C program uses this.
*/
#include <stdio.h>

float circleArea(float r)
{
   return 3.14159*r*r;
}
```
int main()
{
    float r, a;
    int i;

    r = 4.2;
    i=0;
    while (i<3) {
        a = circleArea(r);
        printf("A circle of radius %1.4f has area %1.4lf\n", r, a);
        r = r + 0.2;
        i = i + 1;
    } /* End of the while loop. */
}

If we compile and run this program, the output will be

A circle of radius 4.2000 has area 55.4176
A circle of radius 4.4000 has area 60.8212
A circle of radius 4.6000 has area 66.4760

The circleArea function is exactly the same one as before, so let’s start by looking closely at main. The first two statements we encounter are declaring variables. In C, it is necessary to declare variables before they are used. Additionally, they should always be declared at the very beginning of a function, as we have done here. Essentially, these first two lines of code do nothing, though; they are simply telling the compiler that we intend to use two float variables r and a as well as an int called i. It’s not hard to guess that int is for integers.

The next two lines of source are simply assigning values to r and i. Next, we have something which looks interesting: it does exactly what it looks like it should do, and simply repeats some chunk of code over and over, as long as i is less than 3. A while statement generally just repeats the line immediately after it. However, by enclosing several statements in braces, they effectively behave as one. Thus, the four statements immediately following it will be repeated.

Exercise 1.1 What would happen to the program circlearea2.c if the statement
    i = i + 1;
(1) were removed, (2) were replaced by i = i - 1; Hint: (1) and (2) do not have the same answer!

Exercise 1.2 Write a program that prints Hello world! and exits.
Let’s get a little fancier. We will use the sieve of Eratosthenes to generate small prime numbers. We will not be too fancy, though (for now :)

/* sieve1.c
Chris Monico, 2/2/04.
Program to generate primes using the sieve of Eratosthenes.
*/
#include <stdio.h>

int sievePrimes(int max)
{ int array[1001], i, p=2, count=1, done=0;

    if (max > 1000) {
        printf("Error: max=%d is too big! (Maximum allowed is %d)\n", max, 1000);
        return -1;
    }
    if (max < 2) /* Special case. */
        return 0;
    /* Initialize the array to all zeros. */
    for (i=0; i<1001; i++)
        array[i] = 0;
    do {
        printf("%d\n", p);
        /* Sieve out all multiples of p. */
        i = p*2;
        while (i < 1001) {
            array[i] = 1;
            i += p;
        }
        /* Find the next entry of the array which is still zero. */
        i=p+1;
        while ((i<1001) && (array[i] != 0))
            i++;
        if ((i==1001) || (i > max))
            done=1;
        else {
            p = i; /* We found another prime! */
            count++;
        }
    } while (done == 0);
    return count;
}
CHAPTER 1. INTRODUCTION TO C

int main()
{
    int n, numP;

    printf("Enter n: ");
    scanf("%d", &n);
    numP = sievePrimes(n);
    if (numP >= 0)
    {
        printf("Found %d primes in [1, %d]\n", numP, n);
    }
    else
    {
        printf("sievePrimes() reported an error.\n");
    }
}

There are many new things going on in this file.

1. The declaration int array[1001] is telling the compiler that array will be an array of 1001 integers. Note that arrays are indexed starting from zero in C! We access individual elements of the array with the same syntax.

2. We are initializing the variables p, count and done at their declarations. This is not only acceptable, but encouraged.

3. The statement
   
   for (i=0; i<1001; i++)
   
   will initialize i to zero, repeat the line of code immediately following it as long as i is strictly less than 1001, incrementing i after each pass. We could also loop on multiple lines of code by using braces, as with the while statement.

4. i += p is simply a shorthand for i = i + p.

5. We used the logical operators && (AND), || (OR), ! (NOT). We also used the unary increment operator ++ in several places to increment variables.

6. We used the equality operator == to test if two quantities are equal (this is different than the assignment operator =). We used the operator != to test if two quantities are not equal.

7. We used the standard library function scanf() to get input from the user.

Exercise 1.3 Modify the program sieve1.c so that it can generate primes up to 5000.
1.5 Arithmetic operators and precedence

Exercise 1.4 Two prime numbers $p$ and $q$ are called twin primes if $|p - q| = 2$. For example, 11 and 13 are twin primes. Modify the program sieve1.c so that it counts the number of pairs of twin primes it finds. i.e., there are two pairs of twin primes below 10. How many pairs of twin primes are there below 4150?

Exercise 1.5 (*) Extrapolate the algorithm employed in sieve1.c, and generalize it to an algorithm for finding all primes in an arbitrary interval $[1, n]$ (i.e., modify the algorithm the same way you modified the code in Exercise 1.3). How many operations (big-oh) does it need asymptotically? How can it be improved? (There is an obvious way to improve it).

1.5 Arithmetic operators and precedence

As mathematicians, we take for granted that the expression $b + m \times x$ means “multiply $m$ times $x$, then add $b$.” But would not it also be logical to simply interpret such notation from left-to-right as “add $b$ to $m$ then multiply by $x$.” These two interpretations of the same expression will result in different answers unless $x$ happens to be equal to one. In any programming language, it is necessary that expressions be completely unambiguous, so there is a very well defined set of rules for evaluating them.

First we digress a little to discuss what an operator is in C. We have already seen some examples of array usage in C. Suppose $A$ is an array of integers, and consider the following statement:

$$x = A[8i + j] + 4*A[j];$$

We know that both $+$ and $\times$ are binary operators, taking two inputs and giving an output. You might be surprised to learn that the square brackets $[]$ are also a binary operator. They take two inputs, an array name (actually, a pointer) and an index. In this example, they return the value of the array at the specified index. It might seem an odd thing to call these brackets an operator, but consider that they need to be parsed in essentially the same way as the arithmetic operators to make sense of a statement like the one above. For example, the language needs to know that the statement does not mean “Multiply $4*A$ and then grab the $j$-th element.”

Because of examples like this, there are actually a large number of operators in the C language. Since almost any combination of them can generally appear in a single expression, there is a very well-defined set of rules for evaluating them. From such a set of rules, we must be able to determine exactly the order in which operators are to be evaluated. In particular, we need to be able to answer:

- Given two different operators in an expression, which one is to be evaluated first?
• Given two occurrences of the same operator at the same level, which one is to be evaluated first?

We answer such questions by specifying precedence and direction for each operator. For example, we would like that an expression in parenthesis is always evaluated first. This means that parenthesis should have a high precedence. In fact, they do have the highest precedence of operators in C. Some operators should naturally precede others, as between * and +. However, other combinations such as * and / do not lend themselves to such an easy prioritization. This is resolved by partitioning the operators into sets and describing the precedence among sets. Then, precedence among operators in the same set depends on the order in which they appear.

Once we have settled the question of which sets of operators should be evaluated before others, there is still a question that remains to be answered regarding multiple instances of operators from the same precedence group at the same level (i.e., when there are no parenthesis or similar overriding circumstances). For example, consider the following statement.

\[ x = 40/4/2; \]

It is valid C syntax, and so it must evaluate to something - but what? Before we answer that, let's answer the more obvious question that the reader must be wondering: Why does the C language even accept such an expression? Shouldn't it just toss it out as meaningless? The answer is twofold. Firstly, it actually makes compiler design and specification of the language easier if such expressions are allowed. That is, by allowing all combinations of operators (with the correct number of corresponding operands) to constitute a valid expression, the job of the compiler, as well as writing a compiler, actually becomes easier. But more importantly, it allows for a great amount of flexibility in writing expressions compactly in the language. Sometimes source code can actually become more readable if it is packed somewhat densely. We will have more to say about this later. For now, suffice it to say that there are fewer restrictions on operator use than one might think.

So back to our example: what does 40/4/2 evaluate to? Looking in Table 1.1, we find the / operator in the third group from the top. The associativity column says “left to right”, meaning that the operators in question will be evaluated left to right. Thus, \[ x = 40/4/2; \] will have the leftmost / operator evaluated, then the rightmost, giving a final answer of \[ x=5. \]

Similarly, the expression \[ y = 40/4*2 \] will evaluate to 20 instead of 5. Why? Looking again at Table 1.1 we see that the operators / and * fall into the same precedence group, so occurrences of them at the same level will be evaluated left to right.
1.5. ARITHMETIC OPERATORS AND PRECEDENCE

Table 1.1: Operator precedence (higher in the table = higher precedence)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>parentheses</td>
<td>left to right</td>
</tr>
<tr>
<td>[]</td>
<td>square brackets</td>
<td>left to right</td>
</tr>
<tr>
<td>++</td>
<td>increment</td>
<td>right to left</td>
</tr>
<tr>
<td>--</td>
<td>decrement</td>
<td>right to left</td>
</tr>
<tr>
<td>(type)</td>
<td>cast operator</td>
<td>right to left</td>
</tr>
<tr>
<td>*</td>
<td>the contents of</td>
<td>right to left</td>
</tr>
<tr>
<td>&amp;</td>
<td>the address of</td>
<td>right to left</td>
</tr>
<tr>
<td>-</td>
<td>unary minus</td>
<td>right to left</td>
</tr>
<tr>
<td>~</td>
<td>one’s complement</td>
<td>right to left</td>
</tr>
<tr>
<td>!</td>
<td>logical NOT</td>
<td>right to left</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>left to right</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>left to right</td>
</tr>
<tr>
<td>%</td>
<td>remainder (MOD)</td>
<td>left to right</td>
</tr>
<tr>
<td>+</td>
<td>add</td>
<td>left to right</td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
<td>left to right</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
<td>left to right</td>
</tr>
<tr>
<td>&gt;</td>
<td>is greater than</td>
<td>left to right</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>left to right</td>
</tr>
<tr>
<td>==</td>
<td>is equal to</td>
<td>left to right</td>
</tr>
<tr>
<td>!=</td>
<td>is not equal to</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>left to right</td>
</tr>
<tr>
<td>^</td>
<td>bitwise exclusive OR</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise inclusive OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>assign</td>
<td>right to left</td>
</tr>
<tr>
<td>+=</td>
<td>add assign</td>
<td>right to left</td>
</tr>
<tr>
<td>-=</td>
<td>subtract assign</td>
<td>right to left</td>
</tr>
<tr>
<td>*=</td>
<td>multiply assign</td>
<td>right to left</td>
</tr>
<tr>
<td>/=</td>
<td>divide assign</td>
<td>right to left</td>
</tr>
<tr>
<td>%=</td>
<td>remainder assign</td>
<td>right to left</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>right shift assign</td>
<td>right to left</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>left shift assign</td>
<td>right to left</td>
</tr>
<tr>
<td>&amp;=</td>
<td>AND assign</td>
<td>right to left</td>
</tr>
<tr>
<td>^=</td>
<td>exclusive OR assign</td>
<td>right to left</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>inclusive OR assign</td>
</tr>
</tbody>
</table>
1.6 Using printf and scanf for I/O

Possibly the part of the C language which is most intimidating to newcomers is input/output using the standard I/O functions. The standard function for performing output is `printf()`, and the standard function for getting input is `scanf()`. These functions send output to `stdout` (standard output) and get input from `stdin` (standard input), and are actually special cases of a more general collection of functions defined in `stdio.h`. Under normal circumstances, `stdout` is nothing more than a fancy name for the screen and `stdin` is a representation of the keyboard. These concepts are actually far more robust than that, but for now this description will suffice. Let’s look at some simple examples.

```c
/*stdio1.c
   Chris Monico, 2/9/04.
   Standard Input Output example.
*/
#include <stdio.h>

int main()
{
    int n=150;
    float x=123.5678;
    char c='Z';

    printf("The integer is %d. ", n);
    printf("The floating point number is %f.\n", x);
    printf("The char is %c.\n", c);

    printf("The integer is %d and the float is %f.\n", n, x);
}
```

The output of this program is:

The integer is 150. The floating point number is 123.567802.
The char is Z.
The integer is 150 and the float is 123.567802.

The general format for a `printf` function call is:

```
printf( [Format string], [variable 1], ..., [variable k] )
```

The first argument is special - it is what’s known as the formatting string. It should be a string of characters which describes the precise format of the output you want. In the first example above, we have
printf("The integer is %d. ", n);

Generally, the format string will print out exactly as it looks, with a few exceptions. The first exception is this: The % sign is special - it indicates that a variable is to be printed at that location. The character immediately following the % character indicates what type of variable it is. The real reason that we have to indicate the variable type is a little technical, but just think of it this way: We have total control over the format of the output. But total control comes at a price - we have to specify the format totally, including variable types. It should be obvious from the above example that %d then means we wish to print an integer ('d' is for decimal, as opposed to octal or hexadecimal formatting, which is also possible).

Similarly, in the next statement the %f indicates that we wish to print a float. Then, of course, the %c stands for char. So what are all the types? Well, there are a lot of them but here are examples of the commonly used items for the format string.

| %d | An int in decimal (base 10). |
| %f | A float. |
| %c | A char. |
| %s | A string. |
| %ld | A long int. |
| %lf | A double (lf ≈ long float) |
| %e | A float in scientific notation. |
| %% | Print an actual percent sign. |

Now, what about the \n part? These two characters together actually represent a single ‘newline’ character. (How else would we represent it?). It is called an escape sequence. In fact, there are several characters which we might want to print to the screen or do something else with, but we cannot type. Thus, there is an escape sequence for each of them. The only ones with which we are concerned are:

| \n | Newline character. |
| \t | Tab character. |
| \b | Backspace character. |
| \a | Audible signal. |
| \\ | A backslash character. |

As is evident from the general form of the printf function, we can supply as many arguments as we wish. Practically, this means we can print several variables at once. Consider the final statement from the earlier example:

printf("The integer is %d and the float is %f.\n", n, x);
The format string contains two variable specifiers: the \%d and \%f. Thus, we need to supply to the printf function two variables: An int and a float, in that order! This last point is very important - variables passed to printf must be given in the same order as they appear in the format string!

One last note before we move on to input. It is possible to more precisely format most types of output. We present the following example, in which we have specified the number of digits to be printed to the right of the decimal point as well as the minimum width of the entire number (left-padded with blanks, as necessary).

```c
#include <stdio.h>

int main()
{
    double x=123.0123;

    printf("The double is %1.3lf\n", x);
    printf("The double is %2.3lf\n", x);
    printf("The double is %3.3lf\n", x);
    printf("The double is %4.3lf\n", x);
    printf("The double is %5.3lf\n", x);
    printf("The double is %6.3lf\n", x);
    printf("The double is %7.3lf\n", x);
    printf("The double is %8.3lf\n", x);
    printf("The double is %9.3lf\n", x);
    printf("With more digits after the decimal point:\n");
    printf("The double is %9.7lf\n", x);
}
```

This program generates the following output:

```
The double is 123.012
The double is 123.012
The double is 123.012
The double is 123.012
The double is 123.012
The double is 123.012
The double is 123.012
The double is 123.012
With more digits after the decimal point:
The double is 123.0123000
```

Finally, let’s talk about the scanf function. It is quite similar in nature to the printf function in most respects. We give it a format string, specifying the type of input we’re
expecting, as well as the variables that should hold the input. But there’s one notable exception that deserves a brief digression.

Ordinarily, when a function call is made, like:

\[ a = \text{circleArea}(r); \]

The caller would like to know that it’s variable \( r \) has not been modified by the function it was passed to. In fact, this is the default behavior of C. The language guarantees this by not even passing the variable itself, but instead passing a copy of it to the function in question. In this instance, the function \text{circleArea} will not get our variable \( r \) - just a copy of it containing the same value.

What, then, if we wish to allow a function to modify some of it’s input? In that case, we should pass a \textit{pointer} to the variable rather than the variable itself. A pointer to a variable is simply an integer which specifies the memory location where the variable is living. Furthermore, C provides an operator which does exactly that: when applied to a variable, it produces a pointer to that variable. It is the unary \& operator, and is usually very easy to use. So let’s look again at some examples:

/* stdin1.c
   Chris Monico, 2/9/04.
   A simple example of how to use scanf(). */
#include <stdio.h>

int main()
{ int n, m;
  double x;

  printf("Enter an integer: ");
  scanf("%d", &n);
  printf("You entered %d.\n", n);

  printf("Enter an floating point number: ");
  scanf("%lf", &x);
  printf("You entered %1.5lf.\n", x);

  printf("Enter two integers: ");
  scanf("%d %d", &n, &m);
  printf("You entered %d and %d.\n", n, m);
}
Note in particular that we use the same format specifiers for `scanf` as for `printf`. The only real notable difference is that the variables which are supposed to be modified must be passed by pointer (address). In this case, that means little more than preceding the name of the variable with the unary \& operator. Writing functions that accept pointer arguments is a little trickier, but we’ll get to that later.

### 1.7 Recursion and the Euclidean Algorithm

Here we will put together a few things from earlier sections, as well as giving an example of how to do recursion in C. Recall that the greatest common divisor (gcd) of two integers \( u, v \in \mathbb{Z} \) is the largest positive integer \( g \) such that \( g|u \) and \( g|v \). The gcd of \( u \) and \( v \) is denoted by \( \text{gcd}(u, v) \) (when there is no chance of ambiguity, some authors prefer simply \((u, v)\), but we will not abuse notation so badly). For example,

- \( \text{gcd}(10, 35) = 5 \).
- \( \text{gcd}(13, 2) = 1 \).
- \( \text{gcd}(0, 41) = 41 \).

Recursion can be thought of as the antithesis of mathematical induction. Typically, an argument employing mathematical induction starts with some known base case and works its way up, ad infinitum. Recursion, on the other hand, starts at some unknown base case and works its way down to a known case. It embodies the very notion of mathematics in some sense:

1. Suppose you are given a problem you don’t know how to solve.
2. Simplify the problem.
3. If you can solve it then do so. Otherwise, goto 2.

We will show how recursion can be used to compute gcd’s. First, we recall the division algorithm for integers.

**Lemma 1.7.1 (The Division Algorithm)** Let \( u, v \) be integers with \( v \neq 0 \). Then there exist unique integers \( q, r \) with \( 0 \leq r < |v| \) so that

\[
u = qv + r.
\]

**Exercise 1.6** Prove Lemma 1.7.1. By example, show that uniqueness need not hold if we insist only that \(|r| < |v|\).
In Lemma 1.7.1, the integer \( r \) is nothing more than the remainder when \( u \) is divided by \( v \). However, to insure that the remainder is unique, we impose the condition that it should always be non-negative. In C, we have a modulo operator \( \% \) for computing the remainder of \( u \) divided by \( v \). Some care must be exercised, though, since the exact behavior of the \( \% \) operator can vary from machine to machine when one of the operands is negative. However, when both are positive there is no ambiguity and the operation \( u\%v \) will give the remainder as in Lemma 1.7.1.

Finally, we describe the recursive algorithm for computing gcd's.

**Algorithm 1.7.2 (Recursive Euclidean Algorithm)**  
*Input:* Non-negative integers \( a, b \).  
*Output:* \( \gcd(a, b) \).

1. Set \( x \leftarrow \min\{a, b\}, y \leftarrow \max\{a, b\} \).
2. If \( x = 0 \), output \( y \) as the gcd and terminate.
3. Apply Algorithm 1.7.2 to the pair \( (x, y \mod x) \) and return the result.

**Proof:** Suppose the initial input to the algorithm is \((a_0, b_0)\), and after Step 1 results in the pair \((x_0, y_0)\) with \( x_0 \leq y_0 \). The termination condition of Step 2 is certainly correct, since \( \gcd(0, y) = y \). Thus, assume that \( x_0 > 0 \).

Now, if \( d \) is any integer dividing \( x_0 \) and \( y_0 \), then it necessarily divides \( y_0 \mod x_0 \). To see this, suppose

\[
y_0 = qx_0 + r.
\]

Then \( y_0 - qx_0 = r \). Since \( d \) divides \( y_0 \) and \( x_0 \), it must divide \( y_0 - qx_0 \) which is equal to \( r \). In particular, we have that

\[
\gcd(x_0, y_0) = \gcd(x_0, y_0 \mod x_0).
\]

So that the result of Step 3 is correct. All that remains is to show that the algorithm terminates. Observe that whenever Step 3 is executed, the value of \( x \) at the next iteration will always be strictly less than it was the previous time. That is to say, \( y \mod x \) is strictly less than \( x \), so the values of \( x \) obtained at each iteration of Step 2 are strictly decreasing, \( x_0 > x_1 > x_2 > \cdots \). Yet, they are all necessarily non-negative, so one of them must eventually be zero, terminating the algorithm.

Note that the proof we gave for Algorithm 1.7.2 contains a number of subtleties which are often used in practice. For example, we never did have to directly show that the output at Step 2 is equal to the gcd of the original input. Instead, we relied on a chain of reasoning that leaves no other possible conclusion. Finally, here is a simple C implementation of Algorithm 1.7.2. Observe that there is really nothing to it: In C, it is perfectly acceptable for a function to refer to itself.
/* euclidean.c
    Chris Monico, 2/9/04.
    A simple implementation of the Euclidean algorithm
    implemented recursively.
*/
#include <stdio.h>

long gcdRecursively(long a, long b)
    /* 'a' and 'b' must be non-negative! */
{
    long x, y;
    /* Setup so that x=min{a, b} and y=max{a, b}. */
    if (a < b) {
        x = a; y = b;
    } else {
        x = b; y = a;
    }
    if (x==0)
        return y; /* gcd(0, y) = y. */
    return gcdRecursively(y%x, x);
}

int main()
{
    long u, v, g;

    printf("Enter two non-negative integers: ");
    scanf("%ld %ld", &u, &v);
    g = gcdRecursively(u, v);
    printf("gcd(%ld, %ld) = %ld.\n", u, v, g);
}

1.8 Some math functions

So far, the only standard functions we have used have been the standard input/output functions printf and scanf. These are but the tip of the iceberg for the available input/output functions that C provides. For that matter, the input/output functions are but a small part of the total collection of standard functions that C provides. In this section, we will talk about some of the math functions that are available.

As with the standard I/O functions, we must tell the compiler if we wish to use the math
functions. We do this with the statement (which is actually a *compiler directive*)

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

Here is a small list of some of the functions that are available to us from `math.h`:

- `acos`, `asin`, `atan`, `cos`, `sin`, `tan`, `cosh`, `sinh`, `tanh`, `acosh`, `asinh`, `atanh`, `exp`, `log`, `log10`, `pow`, `sqrt`, `hypot`, `fabs`, `ceil`, `floor`, `erf` (error function), `lgamma` (logarithm of the gamma function), `j0`, `j1`, `jn` (Bessel functions of the first kind), `y0`, `y1`, `yn` (Bessel functions of the second kind). These functions generally take and return `double` arguments.

As an example, we present a simple implementation of Newton’s method. Recall first the mathematical definition of Newton’s method: If $f(x)$ is a polynomial, we start with an initial guess point $x_0$ which can be chosen arbitrarily. We define a sequence by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

**Exercise 1.7** Show that, under suitable conditions, the sequence \{${x_n}$\} defined above converges to a root of $f$. (Hint: If you haven’t seen Newton’s method before, think geometrically about what it might be doing).

Assuming the sequence \{${x_n}$\} converges to a root, we hope that it will also converge to a number representable by the computer.

```c
/* newton1.c
   Chris Monico, 2/9/04.
   A simple implementation of Newton’s method for
   finding a zero of a polynomial.
*/
#include <stdio.h>
#include <math.h>

double newton(double *coef, int degree, double x)
{
    double f, df;
    int i;

    /* Set f <-- f(x). */
    f = coef[0];
    for (i=1; i<=degree; i++)
        f += coef[i]*pow(x, i);
```

/ Set df <-- f'(x). */

df = 0.0;
for (i=0; i<degree; i++)
    df += (i+1)*coef[i+1]*pow(x, (double)i);
return x - f/df;

} /********************************

double getZero(double *coef, int degree)
{
    double x0, x1, x2;

    x0 = 1.0; /* Arbitrary. */
    x1 = newton(coef, degree, x0);
    x2 = newton(coef, degree, x1);
    do {
        x0 = x1;
        x1 = x2;
        x2 = newton(coef, degree, x1);
    } while (fabs(x2-x1) < fabs(x1-x0));
    return x1;

} /********************************

int main()
{
    double coef[20], root, evalRoot;
    int degree, i;

    printf("Enter the degree of the polynomial f: ");
    scanf("%d", &degree);
    printf("Enter the %d coefficients from low to high: ", degree+1);
    for (i=0; i<=degree; i++)
        scanf("%lf", &coef[i]);
    root = getZero(coef, degree);
    printf("getZero() reported %.1lf as a root.\n", root);

    evalRoot = coef[0];
    for (i=1; i<=degree; i++)
        evalRoot += coef[i]*pow(root, i);
    printf("f(%.1lf) = %.1lf.\n", root, evalRoot);
    if (evalRoot > 0.00001)
        printf("getZero() probably failed. Does f even have a real root?\n");
}
1.9 Built-in data types

Before we begin diving into interesting algorithms, there are a few last major topics to cover. We have already seen examples of the most fundamental data types. For completeness, we now list all of the built-in data types of the C language.

<table>
<thead>
<tr>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>characters</td>
</tr>
<tr>
<td>int</td>
<td>positive and negative integers</td>
</tr>
<tr>
<td>long</td>
<td>positive and negative integers (usually using max machine precision)</td>
</tr>
<tr>
<td>float</td>
<td>floating point numbers</td>
</tr>
<tr>
<td>double</td>
<td>floating point numbers (usually using max machine precision)</td>
</tr>
</tbody>
</table>

Strictly speaking, *long* is actually shorthand for the type *long int*, but the difference does not concern us. Also, note that the type-modifier *unsigned* may be applied to the first three data types to obtain *unsigned char*, *unsigned int*, and *unsigned long*. Obviously, the last two are used to represent non-negative integers. The idea is that a *long*, for example, may be represented by 32 bits on a particular machine (i.e., a typical PC). This means that it can take on one of \(2^{32} = 4,294,967,296\) values. Since it should be able to represent both positive and negative values, such a *long* would generally hold numbers in the range \([-2^{31}, 2^{31} - 1] = [-2147483648, 2147483647]\) (it can’t be symmetric because zero must be stored as well! The reason it is generally chosen to be skewed in this way has to do with how the machine itself represents negative numbers. See, for example, *two’s complement arithmetic*.) So in this case, we could use the type *unsigned long* to represent integers in the range \([0, 2^{32} - 1] = [0, 4294967296]\). This would be useful, for example, if we knew we would only need non-negative integers and we wanted to represent the largest one we could.

The *unsigned char* may seem a bit bizarre to the reader. After all, a character is a character - it does not have a sign, right? This is one of the subtle points of the C language - it does very closely match computer architecture. In this case, we must realize that ultimately a *char* will be represented as a collection of zeros and ones. The only substantial difference between a *char* and an *int* is that we need fewer bits to represent a *char*. For all intents and purposes, it needs only to be able to hold the 128 characters defined by the ASCII standard (ASCII = American Standard Code for Information Interchange). Typically, a *char* is represented by a single byte (8-bits) at the machine level. But 8 bits are 8 bits; regardless of what they represent, they can still be interpreted in a natural way as an integer. The difference is simply that it will have to be a small integer. This may seem confusing at first, but it is very useful to have this capability. We illustrate with a simple example.

/* chararith.c
   Chris Monico, 2/16/04.
   A simple demo of the usefulness of having arithmetic
   on the ‘char’ data type.
*/
/*

int main()
{ char c;

    printf("Enter a letter: ");
    scanf("%c", &c);
    if ((c>='a') && (c<='z'))
        c = (c-'a')+'A';
    printf("The capitalized version is: %c.\n", c);
    c = (c-'A'+1)%26 + 'A';
    printf("The next letter in the alphabet is: %c.\n", c);
}

Note: We have not yet talked about logical operators, but the double ampersand && is the logical AND operator. It tests if condition on the left AND the condition on the right are both true.

So, with 8 bits, a char data type can also be used to represent integers in the range \([-128,127]\). Thus, the unsigned char type would be able to represent integers in the range \([0,255]\). Table 1.2 gives the table defining the ASCII standard. The first 32 entries in that table correspond to various control characters which are not displayable. In the table, the ‘Dec’ column is the decimal representation of the entry, ‘Hex’ is the hexadecimal representation of the entry and ‘Char’ is the character representation of the entry (for those entries which have displayable characters).

Remark 1.9.1 Some compilers have defined various extensions to the C language where they appear to have more built-in data types than we have listed here. These are compiler features, however, and not to be confused with the language itself. The gcc compiler, for example, supports a long long type, which will give a 64 bit integer on a PC. But any code written using this data type will not compile with many other compilers. Such ‘features’ of the gcc compiler are intended to be used by people writing very low-level code for a very specific platform (for example, code for an operating system which is inherently platform specific anyway).

1.10 Pointers and arrays

After reading the last section, the reader might be troubled by how few built-in data types the C language has. What about strings? What about arrays?... We should not be troubled by this at all. The point is that the language really was designed with machine architecture in mind. Essentially, the built-in data types of the language mimic exactly the data types that a typical modern CPU can operate on. But fear not - the designers of the language have
### Table 1.2: ASCII characters

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>NUL</td>
<td>32</td>
<td>20</td>
<td>SPACE</td>
<td>64</td>
<td>40</td>
<td>@</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SOH</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>STX</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>ETX</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>EOT</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>ENQ</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>ACK</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>BEL</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>BS</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>TAB</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>a</td>
<td>LF</td>
<td>42</td>
<td>2a</td>
<td>*</td>
<td>74</td>
<td>4a</td>
<td>J</td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>VT</td>
<td>43</td>
<td>2b</td>
<td>+</td>
<td>75</td>
<td>4b</td>
<td>K</td>
</tr>
<tr>
<td>12</td>
<td>c</td>
<td>FF</td>
<td>44</td>
<td>2c</td>
<td>,</td>
<td>76</td>
<td>4c</td>
<td>L</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
<td>CR</td>
<td>45</td>
<td>2d</td>
<td>-</td>
<td>77</td>
<td>4d</td>
<td>M</td>
</tr>
<tr>
<td>14</td>
<td>e</td>
<td>SO</td>
<td>46</td>
<td>2e</td>
<td>.</td>
<td>78</td>
<td>4e</td>
<td>N</td>
</tr>
<tr>
<td>15</td>
<td>f</td>
<td>SI</td>
<td>47</td>
<td>2f</td>
<td>/</td>
<td>79</td>
<td>4f</td>
<td>O</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>DLE</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>DC1</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>DC2</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>DC3</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>DC4</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>NAK</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>SYN</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>ETB</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>CAN</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>EM</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>26</td>
<td>1a</td>
<td>SUB</td>
<td>58</td>
<td>3a</td>
<td>:</td>
<td>90</td>
<td>5a</td>
<td>Z</td>
</tr>
<tr>
<td>27</td>
<td>1b</td>
<td>ESC</td>
<td>59</td>
<td>3b</td>
<td>;</td>
<td>91</td>
<td>5b</td>
<td>]</td>
</tr>
<tr>
<td>28</td>
<td>1c</td>
<td>FS</td>
<td>60</td>
<td>3c</td>
<td>&lt;</td>
<td>92</td>
<td>5c</td>
<td>\</td>
</tr>
<tr>
<td>29</td>
<td>1d</td>
<td>GS</td>
<td>61</td>
<td>3d</td>
<td>=</td>
<td>93</td>
<td>5d</td>
<td>]</td>
</tr>
<tr>
<td>30</td>
<td>1e</td>
<td>RS</td>
<td>62</td>
<td>3e</td>
<td>&gt;</td>
<td>94</td>
<td>5e</td>
<td>^</td>
</tr>
<tr>
<td>31</td>
<td>1f</td>
<td>US</td>
<td>63</td>
<td>3f</td>
<td>?</td>
<td>95</td>
<td>5f</td>
<td>_</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>96</td>
<td>60</td>
<td>'</td>
<td>127</td>
<td>7f</td>
<td>DEL</td>
</tr>
</tbody>
</table>
left us plenty of mechanisms for designing and using more complicated structures. We will have much more to say about this in the next section. For now, let’s discuss what appears to be a glaring omission from the language - strings.

When we talk about *strings* what we actually mean is *strings of characters*.

This sentence, for example, can be considered as a string of 75 characters.

It is clear that we can equivalently think about a string as being an array (or list) of characters. As it happens, computer memory is arranged as a large (but finite) contiguous array, with entries indexed by integers.

So, an array of characters can be mapped very easily onto computer memory. The string

This sentence, for example, can be considered as a string of 75 characters.

can be stored in memory as follows:

Terrific! We know how to store it in memory, but how do we use it in the language? We have already seen some examples of pointers in the C language, when we looked at the *scanf* function. A *pointer* is literally something that *points* to a memory location. This is exactly how we handle strings in C, and more generally, we will see that this is how arrays of all types are handled. A *string is represented by a pointer to its first element*.

But there is one caveat to using strings in C which has caused many a headache to newcomers and experts alike:

**Important:** Text strings in C should always be null-terminated!
That is, we should always mark the end of a string with a null character. This is the escaped character ‘\0’, which generally corresponds to the numeric value zero (though the specification for the language does not require this). Why do we need to do this? There are many standard library functions that we can use to operate on and manipulate strings. Not the least of which is `printf`, which can be used to print a string. If we ask `printf` to print a string for us, we will literally give it the memory location where the string begins (via a pointer), but it also needs to know where the string ends. Rather than passing around an extra parameter with the length of the string, this is simply the convention that the developers of the language adopted. Thus, we should actually store it in memory like this:

```
<table>
<thead>
<tr>
<th>Byte n</th>
<th>Byte n+1</th>
<th>Byte n+2</th>
<th>Byte n+3</th>
<th>Byte n+74</th>
<th>Byte n+74</th>
</tr>
</thead>
<tbody>
<tr>
<td>'T'</td>
<td>'h'</td>
<td>'i'</td>
<td>'s'</td>
<td>'0'</td>
<td>'0'</td>
</tr>
</tbody>
</table>
```

Literally, the string would be stored in memory in the following way (compare with the ASCII table given).

```
<table>
<thead>
<tr>
<th>Byte n</th>
<th>Byte n+1</th>
<th>Byte n+2</th>
<th>Byte n+3</th>
<th>Byte n+74</th>
<th>Byte n+74</th>
</tr>
</thead>
<tbody>
<tr>
<td>84</td>
<td>104</td>
<td>105</td>
<td>115</td>
<td>46</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Here is a small example of string usage in C.

```c
/* stringex1.c
   Chris Monico, 2/16/04.
   A small example of string usage.
*/
#include <stdio.h>

/*******************************************/
void capitalize(char *str)
/* Convert all alphabetic characters in the
given string to uppercase. */
{
    int i;

    i=0;
```
while (str[i] != '\0') {
    if ((str[i] >= 'a') && (str[i] <= 'z'))
        str[i] += ('A'-'a');
    i++;
}

/*******************************************/
int main()
{
    char str[256];

    printf("Enter a string: ");
    fgets(str, 256, stdin);
    capitalize(str);
    printf("The capitalized string is:\n%s\n", str);
}

With the following input:

Hello there! How are you?

the program will generate the following output:

HELLO THERE! HOW ARE YOU?

There are several important things going on in this example.

1. In main(), we declared our str variable to be simply an array of characters.

2. We used the standard I/O function fgets to input the string. The function scanf
   does support string input, but it always interprets space as argument delimiting. That
   is, scanf would view the input "Hello there!" as two separate strings. The function
   fgets considers everything upto an EOL (End Of Line) as a single string.

3. The function capitalize is declared to take an argument of type 'char *'. This is
   the notation for 'pointer to a character'. The reason is that an array is not actually a
   data type in C. Although we can declare an array, as we did in this example with char
   str[256], this is merely a shorthand notation for saying "give me memory to hold 256
   characters, and make str a pointer to it." This is very literally what happens: the
   variable str is actually a pointer to a character. In fact, it is pointing to the first of
   256 characters that we have had allocated to us.
There are a number of useful functions declared in the file `string.h`. Some of these are:

- **strlen**
  Get the length of a string.
- **strcat**
  Concatenate two strings.
- **strcmp**
  Compare two strings (lexicographically).
- **strcasecmp**
  As above, but case-insensitive.
- **strncmp**
  Compare up to \( n \) characters of two strings.
- **strcpy**
  Copy a string.
- **strlen**
  Get the length of a string.

There are others as well, but these are the most often used ones.

The point that we made in 3 above is actually true for all arrays in C.

As an example, look back at the example `newton1.c` in Section 1.8. In particular, we declared an array of doubles to hold the coefficients. But when we passed it to the function `getZero`, we actually passed a pointer!

### 1.11 Structures and more on pointers

What do we do when we need more data types other than the built-in ones? We construct them from the basic data types. We will introduce the reader here to the two important keywords **typedef** and **struct**. The first, **typedef**, gives a way to tell the compiler “Hey - I don’t want to type this long data-type name over and over, I’m going to use this name instead”. For example, if we are using the data type **unsigned long** in many many places, we may quickly tire of typing it all out. If we add the following statement to the top of our program:

```c
typedef unsigned long ulong;
```

then we can simply start typing `ulong` instead of `unsigned long`.

The **struct** keyword is a bit more useful. It allows us to combine several data items into a single structure with named fields.

```c
struct {
```
We could use this to represent polynomials. Of course, it is a lot to type each time we need to declare a polynomial, so we would also use the `typedef` keyword to make our lives a little easier.

```c
/* structex1.c
   Chris Monico, 2/16/04.
*/
#include <stdio.h>

typedef struct {
    long coef[20];
    int degree;
} poly_t;

/*******************************************/
void printPoly(poly_t f)
{
    int i;

    for (i=0; i<f.degree; i++) {
        if (i>1)
            printf("%ld*x^%d ", f.coef[i], i);
        else if (i==1)
            printf("%ld*x ", f.coef[i]);
        else
            printf("%ld ", f.coef[i]);

        if (f.coef[i+1] >= 0)
            printf("+ ");
    }
    printf("%ld*x^%d\n", f.coef[f.degree], f.degree);
}

/*******************************************/
int main()
{
    poly_t f;

    f.coef[0] = 7;
    f.coef[1] = 2;
    f.coef[2] = 1;
```
Recall: when we pass a variable to a function, what actually happens at runtime is that a copy of it is made, and the copy is passed to the function. This serves two purposes.

1. From the function’s point of view, it’s input is a variable just like any other. That is, it behaves exactly the same as if it were a variable declared by the function itself. The only difference is that the initial value of the variable may be meaningful (e.g., it may contain input).

2. From the caller’s point of view, it is nice to know that functions cannot simply modify whatever they want whenever they want. How rude would it be if I asked `printf` to print an integer for me and it arbitrarily decided to modify it?

But with the example we have above, this creates two potential problems (as well as some other much more subtle problems that we will ignore for now).

1. What if we have a very large structure and call alot of functions? We could find ourselves wasting alot of time copying variables.

2. What if we actually want a function to modify it’s input?

These problems are both solved by using pointers to the structures. The notation for getting the pointer to a given structure is exactly the same as any other variable: we simply prepend it with an ampersand `&`. However, we now hit a problem we haven’t directly addressed: dereferencing. To dereference a pointer simply means to follow it. For example, a pointer may literally be an integer like 19123. Dereferencing this pointer means looking at the contents of memory location 19123. The difference is pretty subtle, so let’s say it again: The value of this pointer is 19123. The dereferenced pointer is the contents of memory location 19123. In the same way that we can get a pointer to a variable by prepending with an ampersand, we can dereference a pointer by prepending with an asterisk `*`.

/* structex2.c
   Chris Monico, 2/16/04.
*/
#include <stdio.h>
typedef struct {
    long coef[20];
    int    degree;
} poly_t;

/******************************************/
void printPoly(poly_t *f)
/* I have modified this function from the one in structex1.c
    just to illustrate one way to dereference a pointer.
*/
{
    int i;
    for (i=0; i<(*f).degree; i++) {
        if (i>1)
            printf("%ld*x^%d ", (*f).coef[i], i);
        else if (i==1)
            printf("%ld*x ", (*f).coef[i]);
        else
            printf("%ld ", (*f).coef[i]);
        if ((*f).coef[i+1] >= 0)
            printf(" + ");
    }
    printf("%ld*x^%d\n", (*f).coef[(*f).degree], (*f).degree);
}

/******************************************/
int squarePoly(poly_t *r, poly_t f)
/* Do r <-- f*f.
   This function will use a different mechanism to
dereference the pointer. We could use an ampersand
as in the previous function. But when we deal with
a pointer to a structure, we have another option:
we can use the ‘arrow’ ->
*/
{
    int i, j;
    for (i=0; i<=2*f.degree; i++)
        r->coef[i] = 0; /* Same as *r.coef[i] */
    for (i=0; i<f.degree; i++)
        for (j=0; j<=f.degree; j++)
            r->coef[i+j] += f.coef[i]*f.coef[j];
    r->degree = 2*f.degree;
1.11. *STRUCTURES AND MORE ON POINTERS*

```c
int main()
{
    poly_t f, g;

    f.coef[0] = 7;
    f.coef[1] = 2;
    f.coef[2] = 1;
    f.degree = 2;
    printPoly(&f);
    squarePoly(&g, f);
    printf("The polynomial squared is:
    ");
    printPoly(&g);
}
```

Running this program results in the following output.

```
7 + 2*x + 1*x^2
The polynomial squared is:
49 + 28*x + 18*x^2 + 4*x^3 + 1*x^4
```

Look at the function `printPoly`. Dereferencing the pointer is a bit difficult in that case, because the compiler wants to resolve the field name first and then dereference. In this case, however, it would not make sense - we need to dereference and then resolve the field name. To make the dereferencing happen first, we had to add parenthesis. This can become rather cumbersome, so the language gives us another choice. We could replace the awkward

```
for (i=0; i< (*f).degree ; i++) {
```

with the much more succinct

```
for (i=0; i< f->degree ; i++) {
```

That is, this gives us a mechanism for dereferencing and selecting a field, in that order.

Finally, we note that it is also possible to create an array of structures. If `poly_t` is defined as in the previous example, the following code would make perfect sense.

```
int main()
```
{ poly_t f_i[5]; /* 5 polynomials. */

    /* Set the polynomial f_i[0] to be 2 + x^2. */
    f_i[0].degree = 1;
    f_i[0].coef[0] = 2;
    f_i[0].coef[1] = 1;
    printPoly(&f_i[0]);
}

Exercise 1.8 Using the following structure,

typedef struct {
    double real;
    double im;
} comp_t;

create the four functions compAdd, compSub, compMul, compDiv for doing complex arithmetic.

Exercise 1.9 This problem is motivated by a recurring problem in computer graphics, as well as other areas. Write a program that takes, as input, an integer \( n \) followed by a list of \( n + 1 \) real-valued points \( P_i \) such that the first \( n \) of them define a simple polygon (not necessarily convex). For example, a sample input would be

\[
\begin{align*}
4 \\
0.0, 0.0 \\
2.0, 2.0 \\
2.5, 0.5 \\
3.0, 0.0 \\
0.2, 0.3
\end{align*}
\]

The program should determine whether or not the \( n + 1 \)-th point is in the interior of the polygon defined by the line segments \( \overline{P_1P_2}, \overline{P_2P_3}, \ldots, \overline{P_{n-1}P_n}, \overline{P_nP_1} \). It should output either interior or exterior accordingly. For this example, the program should output interior. However, if the last point were changed to 2.4, 1.6, the program would have to output exterior.

Exercise 1.10 Write a program to play the game of tic-tac-toe. The program should allow the user to be 'O' and give the user the first move. The program must play the turn of 'X'.
Exercise 1.11 Using the data structures and functions from Exercise 1.8, modify the program newton1.c to work with complex numbers, and to choose a non-real starting point. Does the algorithm/program generally still find a root? Can it find a complex root?

1.12 Keywords in C

This section can (indeed, should) be skipped by the reader, since we are encouraging the reader to learn the language by example. It is provided only for reference.

The reserved keywords in the language (depending on the particular standard you look at, there may be more or less) are:

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

Also note that some compilers ‘extend’ the language to accept other keywords, such as `asm`. We are concerned here with only the C language itself, though, and not the various extensions.

**auto**
The `auto` keyword is used to specify that the storage class of a variable should be local. That is, the variable is only recognized within the code block where it was defined. Since this is the default behavior for variable definitions anyway, this keyword is almost never used.

*Usage:* `auto variable-type variable-name [ = value];`

*Example:* `auto int x=2;`

**break**
The `break` statement is used to ‘break’ out of a loop. Specifically, it causes program execution to immediately leave the innermost enclosing `do`, `for`, `switch`, or `while` loop.

*Usage:* `break;`

*Example:*

```c
for (x=i=0; i<100; i++) {
    x = x + i;
    if (x > 21)
```
break;
}
printf("When control arrives here, x will be 28 and i will be 7.\n");

case
See switch.
char
See int.
const
The const keyword is used to designate a variable or pointer parameter as a constant (i.e., it cannot be modified).

Usage: const variable-type variable-name [ = value];
Example 1: const int arraySize=100;
Example 2: int printf(const char *format, ...); Example 1 shows a constant declaration of a variable. Any attempt to modify the value of arraySize will generate a compiler error. In Example 2, we have given the function declaration for the standard library printf function. The declaration specifies that the function should not be allowed to modify the memory pointed to by format (i.e., it cannot modify the format string.)

continue
The continue statement is similar to the break statement, except that it causes program execution to immediately jump to the end of the innermost enclosing do, for, switch, or while loop, at which point the loop condition is re-evaluated.

Usage: continue;
Example:

for (x=i=0; i<10; i++) {
    x += i;
    if (i < 4)
        continue;
    printf("1 + 2 + ... + %d = %d.\n", i, x);
}

In this example, the first line of output will be 1 + 2 + ... + 4 = 10.

default
See switch.
do

`do` is used together with `while` to create another looping construct. It causes a statement (or block of statements) to be executed once, then repeated as long as some condition remains true.

Usage: `do statement while (expression);`

Example 1:

```c
x=i=0;
do x += i++; while (i<10);
```

Example 2:

```c
x=i=0;
do {
    x = x + i;
    i = i + 1;
} while (i<10);
```

These two examples are actually equivalent; both will end with the values `i=10` and `x=45`.

double

One of the built-in data types for representing floating point numbers. It has at least as much precision (generally more than) a `float`.

double

else

See `if`.

de

enum

The `enum` keyword is used to define a set of constants of type `int`. Perhaps more importantly, the set itself can be defined as a type.

Usage: `enum [tag] { name1 [=value], ... };`

Example:

```c
enum Suits { Clubs, Hearts, Diamonds, Spades};
```

We can then declare variables of type `enum Suits`:

```c
{ enum Suits mySuit;
```
mySuit = getCard();
    if ((mySuit == Hearts) || (mySuit == Diamonds))
        printf("The card is red.\n");
}

In this example, Clubs is actually a constant int equal to zero. Hearts is equal to one, and so on. If we wish, we could have overridden this default behavior:

    enum Suits { Clubs=-10, Hearts=14, Diamonds, Spades }; In this case, Diamonds will default to 15 (the next subsequent integer), and so on.

extern
This is another storage class specifier. It indicates that an identifier (either a function definition or the actual storage and initial value of some variable) is defined somewhere else (usually outside of the current file). This is useful if you have a project with multiple files, and one file wants to access a global variable defined in another file.

Usage: extern variable-type variable-name;
or: extern function-prototype;
Example: extern int numberFieldDegree;
or: extern double circleArea(double r);

float
One of the built-in data types for representing floating point numbers. Exactly how much precision is uses may vary from platform to platform, as well as certain other behaviors (default rounding modes,...). For that matter, the same is true of all the built-in data types to some degree.

for
The for statement is another looping construct. It will perform a specified initialization and repeat a set of instructions as long as some given expression is nonzero. Formally, the syntax is:

Usage: for (expr1 ; expr2; expr3;) statement;
The easiest way to see the flow control is to look at the equivalent code:

    expr1;
    while (expr2) {
        statement;
        expr3;
    }
This code is exactly equivalent to the for loop as described by the usage. Example 1:

    for (i=0; i<10; i++)
        printf("i = %d\n", i);
1.12. KEYWORDS IN C

Example 2:

```
for (i=x=0; i<10; i++) {
    x = x + i;
    printf("i=%d, x=%d\n", i, x);
}
```

Example 3:

```
for (i=x=0; (i<10) && (x < 30) ; i++, x+=2)
    printf("i=%d, x=%d\n", i, x);
```

Also note that all three expressions may be the empty expression. If `expr2` is empty, its value is taken to be 1.

goto

This often misunderstood keyword is used to immediately cause program execution to jump to a specified location. If abused, `goto` statements can easily cause infinite loops and debugging nightmares. However, used sparingly, they can be useful.

Usage: `goto label;`

Example:

```
... if (x > 10) 
    goto ALG_2_10_STEP5;
    printf("x is less than or equal to ten.\n");
... ALG_2_10_STEP5:
    printf("I suppose this is where the Step 5 code begins.\n");
...```

The `goto` statement is not allowed to jump into another scope (for example, it cannot jump from one function into another). However, it can jump out of scope (for example, out of a `while` loop). It is an exercise for the reader to think about why this is the case.

if

The `if` statement conditionally executes a specified statement.

Usage: `if (expr) statement; [else statement2;]`

Example 1:
if (x>10)
    printf("x is greater than ten.\n");
else
    printf("x is not greater than ten.\n");

Example 2:

if (x>10)
    printf("x is greater than ten.\n");
else if (x > 5)
    printf("x is greater than five.\n");
else
    printf("x is not greater than five.\n");

int
One of the built in data types. char and int are the built in data types for representing integers. The exact precision they carry can vary from platform to platform, but a char is generally just a single byte (8 bits). An int has at least 16 bits of precision, often 32. The type-modifiers signed, unsigned, short and long can be applied to these. In declaring a long int, however, the int is optional.

long
See int.

register
The register keyword instructs the compiler to attempt to store a variable being declared in a CPU register. This is occasionally done to optimize the compiler output.
Usage: register variable-type variable-name;
Example: register int ii;

return
A return statement causes an immediate exit from the currently executing function, and control returns to caller. Optionally, it specifies a value to be returned.
Usage: return [expression];

short
See int.

signed
See int.
1.12. KEYWORDS IN C

The keyword `sizeof` is actually an operator for determining the size (in bytes) of a given type or expression.

Usage: `sizeof expression`

or: `sizeof(type)`

Example:

```c
printf("A double needs %d bytes of storage.\n", sizeof(double));
printf("A poly_t needs %d bytes of storage.\n", sizeof(poly_t));
```

The `static` keyword specifies a static storage class for the variable being declared. That is, a variable declared as `static` will retain its value even after it’s gone out of scope. It may be applied to both variable and function definitions:

Usage: `static variable-type variable-name [=initial-value];`

or: `static function-definition`

Example:

```c
int myFunction()
{
    static int timesCalled=0;

    timesCalled++;
    printf("myFunction() has been called %d times.\n", timesCalled);
}
int main()
{
    int i;

    for (i=0; i<5; i++)
    {
        myFunction();
    }
}
```

The output of this code snippet would be:

```
myFunction() has been called 1 times.
myFunction() has been called 2 times.
myFunction() has been called 3 times.
myFunction() has been called 4 times.
myFunction() has been called 5 times.
```

The `struct` keyword is used to create records with one or more fields.
Usage: `struct [struct-type-name] {`  
`[variable-type variable-name];`  
`...`  
`} [structure-variables];`  

In the usage, `struct-type-name` and `structure-variables` are both shown to be optional, but at least one of them must be used. Example:

```c
struct poly_s {  
  int    degree;  
  double coef[20], root[20];  
} f1;
```

This declares both a particular type of structure, `struct poly_s`, as well as one variable `f1` of that type. Having declared the type, we are free to later refer to it again, and even declare variables of that type later:

```c
...  
{ struct poly_s g1, g2, h[10];  
  int i;  
  g1.degree = g2.degree = 0;  
  for (i=0; i<10; i++)  
    h[i].degree = 0;
}
```

### switch

The `switch` statement together with the `case` keyword provide a useful mechanism for conditional branching when there are several possibilities. The common usage is as follows:

**Usage:**
```c
switch (expression) {  
  case const-expr_1: statement_1  
  ...  
  case const-expr_k: statement_k  
  [ default : statement ]
}
```

**Example:**

```c
switch (polyDegree) {  
  case 1 :  
    printf("Linear polynomial: easily solved.\n");  
    break;
  case 2 :
```
Notice how the break statements (or lack of, in the last case) affect flow control. Once a matching case statement has been found, all code below it will be executed, unless there is a break to escape out.

**typedef**

The typedef keyword assigns a type-name to a type-definition. That is, it allows one to ‘define’ a new type (in some sense).

*Usage:* typedef type-definition type-name;

*Example:*

```c
typedef unsigned long ulong;
typedef struct { int degree; double coef[20]; } poly_t;
```

Of course, this last one would usually be spread across several lines for readability:

```c
typedef struct {
    int    degree;
    double coef[20];
} poly_t;
```

And we could subsequently declare variables of these types:

```c
int main()
{ ulong   a, b, c;
  poly_t  f, g, h;
  ...
```
The `union` keyword is used to group several variables together into shared memory space. It’s usage is similar to that of `struct`.

**Usage:**

```c
union [union-type-name] {
  [variable-type variable-name] ;
  ...
} [union-variables] ;
```

The point is that the fields may actually overlap so that writing to one field will overwrite another field. This can be useful when a record could contain one of several different types of entries, but only one of them per record. The compiler will allocate enough memory to hold the largest field.

**Example:**

```c
union varint {
  char   c;
  short int  s;
  long int  l;
} num1, num2;
```

Then `num1` and `num2` are variables of this type. Each requires only `sizeof(long int)` bytes to store since, for example, the fields `num1.c`, `num1.s`, and `num1.l` will overlap in memory.

`unsigned`

See `int`.

`void`

This is the ‘empty data type’. It is often used to indicate that a function returns no value or that it accepts no arguments.

**Usage:**

```c
void function-name( [function-args] )
or : return-type function-name(void)
```

Pointers can also be declared to be of type `void`. This is done to allow some functions to be able to operate on data of arbitrary type. The caveat is that `void` pointers cannot be dereferenced without an explicit cast to some data type (the cast is needed so that the compiler knows the size of the object being dereferenced). As an example of the usefulness, we note briefly that there is a standard library function with prototype

```c
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

This function will write out to file the data pointed to by `ptr`. To do this, all it really needs to know is the size of each data element and the number of elements. For example, we could use it to store some polynomials to disk as follows:
int main()
{ poly_t h[10];
  int i;
  FILE *fp;

  ...
  fp = fopen("somepolys.out", "w");
  fwrite(h, sizeof(poly_t), 10, fp);
  fclose(fp);
}

Of course, there is also an \texttt{fread} function which would happily read them back in for us.

\textbf{volatile}

This keyword tells the compiler that a variable may be modified in a way that is not obvious (for example, by another process or as a result of some external interrupt event). As a result, the compiler should not try to optimize occurrences of it (sometimes, for example, if a variable is still in a register after an earlier use, the compiler may decide to simply re-use it in that register rather than reload it from memory). Instead, the compiler should reload the variable from memory each time it is referenced.

\textit{Usage:} \texttt{volatile variable-type variable-name};

\textbf{while}

This is the simplest of all loop constructs in C. It indicates simply that some statement should be repeated as long as some expression is nonzero.

\textit{Usage:} \texttt{while (expression) statement}

First, \texttt{expression} will be evaluated. If it is nonzero, \texttt{statement} will be executed. This will be repeated as long as \texttt{expression} is nonzero. Of course, \texttt{statement} may also be a compound statement.

\textit{Example:}

\begin{verbatim}
x=1;
i=1;
while (i<10) {
  x *= i;
  i++;
}
\end{verbatim}
1.13 Casting and dynamic memory allocation

Casting (or typecasting) is the mechanism by which we may convert from one data type to another. It’s not always clear what the result means, and we shouldn’t do it unless we know what the result will be. Nevertheless, it is sometimes convenient and sometimes necessary (in C) to have this ability. For example, suppose we have a value contained in an int. Does it not make perfect sense to want to set a long variable equal to the value of the int? Or, perhaps we might want to set a double to the same value? We can accomplish this via casting. The syntax for casting is illustrated in the following example.

```c
int main()
{
    int n = 1010;
    long m;
    double x;
    char c;

    m = (long)n;
    x = (double)n;
    /* In fact, we could do the following, and the
    compiler will understand that we want to
    (implicitly) typecast.
    */
    m = n;
    x = n;
    /* However, the following is not completely unambiguous,
    so the compiler will generally insist that we explicitly
    do the cast:
    */
    c = (char)n;
}
```

In the very last example from this code, we have converted from an int (which can generally hold integer values at least in the range $[-32768, 32767]$) to a char (which can generally hold integer values in the range $[-128, 127]$). The language and the compiler will let us do this - it simply assumes that we know what we’re doing. For example, if the number n happened to have a value of, say 100, then we would get exactly what we expect. In this case with n=1010, however, it is not totally clear what the result will be. Nevertheless, the compiler assumes that we know what we’re doing and it will let us get away with it.

The idea behind casting is essentially the following: Remember that all data types are eventually living in a sequence of bytes in memory. To cast from one data type to another, all that needs to be done is to interpret the memory locations differently. For example,
perhaps an int is represented by 16 bits. Then it will be stored in memory as two bytes. If we cast this to a char which has, say 1 byte, the compiler will accomplish this feat by taking only one of the bytes from the specified memory location (whichever one is responsible for holding the 'low' bits).

Now an important special case: casting pointers. Internally, a pointer to a long and a pointer to a char are exactly the same thing - they are simply pointers to some memory location (i.e., they are integers containing some memory address). What makes them different is that the compiler knows how to interpret them. For example, if we have the declarations

```c
int main()
{
    long n[10];
    char str[10];
    ...
}
```

then whenever the compiler sees something like n[0], it knows that it should go to the first memory location pointed to by n, and grab a certain number of bytes (say 4), because it knows how big an long is. Similarly, when it sees str[0] it knows to go to the memory location pointed to by str and grab a single byte (because that’s how big a char generally is).

So, when I ask for n[1] what happens? The compiler knows that n is a pointer to longs, and it knows that longs need 4 bytes each. Thus, it looks at the pointer n, adds 4 to it (to skip over the 4 bytes comprising n[0]), and goes to that memory location. The point here is that all the compiler really needs to know is the size of the data type in the corresponding memory. Similarly, when it sees str[1], it looks at the pointer str, recognizes that each element of this array needs only one byte. So it simply adds 1 to the pointer and goes to that memory location.

There are some functions which need to operate on arbitrary data types, and so there is a special notion of a void type pointer. For example, there is a qsort standard library function which will sort any data type you have, so long as you tell it how to compare two items. This function is declared to accept a pointer to arbitrary data, or a void pointer.

Another such function is the memory allocation function, malloc. What do you do if you don’t know in advance how much memory you’ll need to perform a particular task? Imagine you are writing a word processing program. Certainly you need to be able to hold as much text as the machine can handle. But you don’t know in advance how much memory your users machines will have. The point is, there is really now way for you to know exactly how many bytes you need while you’re writing the program. This will only become known at runtime. This sort of problem (and other more subtle ones) occur often enough that there is a very precise way to handle it. It’s called dynamic memory allocation. It does exactly what it sounds like - allocate memory dynamically, as needed.
So what do you do when you need more memory? You ask the memory allocation function `malloc` for some. Of course, you need to tell it how much you need. It will then find a chunk of memory of the right size, and pass it back to you as a pointer. To receive it, you’ll need a pointer to store the result. So a typical call to `malloc` might look like this:

```c
int main()
{
    int dataSize, i;
    long *data;

    /* Right now, data is just a pointer. It does not point to anything in particular. It certainly does not point to any memory owned by our program. So we absolutely can not dereference it until it points to something meaningful. Let’s ask malloc for some memory that data can point to. */
    printf("How much data do you want to enter? ");
    scanf("%d", &dataSize);
    data = (long *)malloc(dataSize*sizeof(long));

    /* Now we can do things like this: */
    for (i=0; i<dataSize; i++)
        data[i] = 0;

    /* Now we’re done with the memory, so give it back: */
    free(data);

    /* Again – data is now just a pointer, but it does not point to anything meaningful, so don’t dereference it! */
}
```

Note that `malloc` is just a function like any other. It is not a special keyword in the language, or anything like that. So it really has no way to know exactly what kind of data we want to store. This is why we must tell it how much we need, in bytes, using `dataSize*sizeof(long)`. Since it does not know what kind of data we need, it will just allocate the needed number of bytes and return what? It doesn’t know the specific type of pointer to return, so it will return it as a `void` pointer. This is no problem, though - we need only cast it to a pointer of the proper type. That’s what the `(long *)` part of that statement is doing.

This section is not done!
Chapter 2

Practicing C

This chapter is a collection of some interesting mathematical problems and exercises to help us become more familiar with programming in C.

2.1 The Mandelbrot set

This is a classic and spectacular example of how complex an apparently simple algorithm can behave. Consider a recurrence relation defined in the field of complex numbers by

\[
\begin{align*}
    z_0 &= 0 \\
    z_{n+1} &= z_n^2 + c.
\end{align*}
\]

where \( c \in \mathbb{C} \) is some constant. This is an apparently simple recurrence relation, n’est-ce pas? In fact, a careful look shows that it is actually a family of recurrence relations, since we haven’t specified what \( c \) is.

Certainly one may ask: for what values of \( c \) does the sequence defined by Equation 2.1 converge? It turns out that this is not such an easy question. However, it can be shown that if, for some particular \( c \) and \( n, |z_n| > 2 \) then the sequence \( \{z_n\} \) diverges.

We can (and you will!) attempt to give this some geometric meaning as follows: Let \( X \subset \mathbb{C} \) denote the subset of values of \( c \) for which the sequence defined by 2.1 converges. What does it look like? We can even wonder about the following: when it does diverge, how fast does it do so? We can even gain a visual representation of this by simply plotting points in the complex plane a different color depending on how fast they diverge (in cases where we can determine this). This leads to the following two exercises.

Exercise 2.1 Write a function called \texttt{mval(double x, double y)} to compute the iterations defined by the system 2.1 (with \( z = x + iy \)). It should compute \( z_1, z_2, \ldots \) until either
some $|z_k| > 2$ or $z_{100}$ is reached. It should return the number of iterations it computed (i.e., the smallest $k$ so that $|z_k| > 2$ or 100 if no such $k$ was found.)

**Exercise 2.2** You will use the program `draw.c` as a template for this exercise. It has all the code for creating a window and plotting pixels of a specified color. In this exercise, you will draw the following picture.

Map the set of pixels on a 640x480 screen into the region $-3 \leq Re(z) \leq 1$ and $-1.5 \leq Im(z) \leq 1.5$ of the complex plane. For each pixel on the 640x480 screen, do the following:

- Compute the function `mval()` you wrote in Exercise 2.1 on the corresponding complex value.
- If the value returned by `mval()` is 100, assume the sequence is bounded at that point and paint it black. Otherwise, paint it some other color that depends on the value
2.2. DEBUGGING A C PROGRAM

returned by mval(). The set of values for which the sequence is bounded is called the Mandelbrot set (i.e., it is more-or-less the set of points which you will paint black).

If you modify your program to zoom in on some of the apparently interesting areas of this picture, then things will become interesting. Try zooming in on some interesting areas until machine precision starts to break down the computations.

2.2 Debugging a C program

By now, the reader has almost certainly had the painful experience of trying to “figure out what went wrong”. This process is generally referred to as debugging. The etymology of this term is rather interesting; It goes back to some computer operators who, in 1945, found a moth in a Mark II computer which was causing it to malfunction. They naturally used the term ‘debugging’ for what they had done to fix the computer, and it stuck. See Figure 2.1.

There are any number of problems which can cause a computer program to not do what we expect. We first discuss the process of identifying and correcting syntax errors which cause the compilation process to fail. By now, the reader has almost certainly experienced this.

2.2.1 Example 1

/* debug1.c */
#include <stdio.h>

int main()
{
    int j;

    for (j=0, j<10, j++) {
        printf("Hello world! i=%d\n", j);
    }
}

Trying to compile this code with gcc will result in something like the following.

gcc -o debug1 debug1.c
depth1.c:2:18: stio.h: No such file or directory
depth1.c: In function ‘main’:
depth1.c:7: parse error before ‘)’ token

In this instance, the compiler is telling us several things about each error. It tells us the filename where the error occurred, debug1.c, followed by the line number at which the error occurred, as well as what caused the error.
Figure 2.1: U.S. Naval Historical Center Photograph (The moth taped to the page!)
2.2. DEBUGGING A C PROGRAM

The first error is at line number 2: `stdio.h: No such file or directory`. What caused this? It is a simple typographical error. We were supposed to be including the standard input/output functions using `#include <stdio.h>` and we typed the wrong filename. The compiler tried to find a file called `stdio.h` somewhere on the system but failed, so it complained.

Although there are three lines of compiler complaints, there were actually only two errors generated. The next line which says `debug1.c: In function ‘main’:` is just a simple courtesy telling us in which function the error was found. This is sometimes useful for another reason we’ll discuss in a moment. For now, though, consider the third line:

```
default.c:7: parse error before ‘)’ token
```

The occurred at line 7 in the file debug1.c. As this is a short program, we can locate line 7 by simply counting down from the top. If the file were larger, though, we would rather try something like this from the shell:

```
less -N debug1.c
```

This command will display the contents of the file `debug1.c` one screen at a time, showing the line numbers as it goes. We soon discover that the error was caused by the following line of code:

```
for (j=0, j<10, j++) {
```

The error message is telling us that it discovered something wrong before the closing parenthesis. What went wrong? Ah yes - the `for` loop must have three semicolon-separated expressions in the parenthesis and we don’t have that. Why didn’t it just tell us something more useful like “Hey - you used commas instead of semicolons!”? Because the comma is part of the C language, and `(j=0, j<10, j++)` does actually form a single expression. So, the compiler didn’t know whether we messed up on the punctuation or simply forgot the last two expressions needed to define the loop. Now that we know what the error is, though, it is easily fixed by replacing the commas with semicolons.

What about debugging large programs or finding less obvious errors? There are a large number of tools that are specifically designed to help programmers debug code. Some will even let you step through the program one instruction or statement at a time, watching the values of variables as you go, to help pinpoint specific errors. You can run a program upto a certain point, where you stop it and examine and/or change the values of variables. The possibilities are nearly limitless. However, for us it will suffice to simply debug our programs manually.

2.2.2 Example 2

Consider another small example.
/* debug2.c */
#include <stdio.h>

long factorial(int n)
{ int i;
  long result=1;
  for (i=1; i<=n; i++) {
    result *= i;
    return result;
  }
}

int main()
{ int k;
  long kFact;

  printf("Enter a number: ");
  scanf("%d", &k);
  kFact = factorial(k);
  printf("%d! = %ld.\n", k, kFact);
}

Trying to compile this program results in the following.

gcc -o debug2 debug2.c
debug2.c: In function ‘factorial’:  
debug2.c:22: parse error at end of input

Here we have an apparent contradiction. On the one hand, the compiler says that the error is in the function factorial. On the other hand, it claims the error is at line 22, which is the end of the file. Well, the end of the file is clearly not supposed to be in the function factorial, so what happened? We forgot a closing brace } somewhere in the function factorial, so the compiler never realized we were done defining that function!

2.2.3 Exercises

Here we give a list of small exercises designed to help you get more familiar with the C language, and help you spot common mistakes.

Exercise 2.3 Why won’t the following code correctly compute n!?  Hint: Look carefully at the structure of a proper for loop, bearing in mind that the empty statement is a valid statement in C.
2.2. DEBUGGING A C PROGRAM

```c
long factorial(int k)
{ int i;
  long result=1;
  for (i=1; i<=k; i++)
    result *= i;
}
```

Will it compile?

**Exercise 2.4** What is wrong with the following piece of code? (There are two mistakes.)

```c
#include <stdio.h>

int main()
{ int n;
  long nFact;

  printf("Enter an integer n: ");
  scanf("%d", n);
  nFact = factorial(n);
  printf("%d! = %ld.\n", n);
}
```

**Exercise 2.5** Fix the syntax errors in the following program (there are five of them).

```c
#include <stdio.h>

int main
{ int i; j;
  for (i=0, i<5; i++) {
    for (j=0; j<3; j++) {
      printf("(i,j) = (%d, %d).\n", i, j)
    }
  }
}
```

**Exercise 2.6** The following program is supposed to print out the factorization of small integers. What is wrong with it? (*Hint:* The underlying algorithm is sound, but the program is flawed).
/* debug4.c */
#include <stdio.h>

int main()
{ long n, p,

    printf("Enter a positive integer: ");
    scanf("%ld", &n);
    printf("%ld = ", n);
    if (n==1) {
        printf("(1)\n");
        return 0;
    }

    p=2;
    while (n > 1) {
        if (n%p==0)
            printf("(%ld)", p);
        n=n/p;
        else
            p++;
    }
}

Exercise 2.7 This program should decide if a given number is prime. However, it is loaded
with seven errors (possibly eight, depending on how you count them). Find and fix them all.

/* debug5.c */
#include <stdio,h>

main()
{ long n, d

    printf('Enter a positive integer: ');
    scanf("%ld", &n);

    d=2;
    while (d*d<= n) {
        if (n%d = 0) {
            printf("%ld is composite.\n", n)
            return 0;
        } else
            d++;
}
Exercise 2.8 Let $k$ be any positive integer and consider the sequence of integers defined by

\[
\begin{align*}
    a_1 &= k \\
    a_{n+1} &= \begin{cases} 
        1, & \text{if } a_n = 1 \\
        a_n/2, & \text{if } a_n \text{ is even} \\
        3a_n + 1, & \text{if } a_n > 1 \text{ is odd.}
    \end{cases}
\end{align*}
\]

So, for example, with $k = 12$ we obtain the sequence $12, 6, 3, 10, 5, 16, 8, 4, 2, 1, 1, 1, \ldots$ Write a program that takes as input an integer $k$ and computes this sequence. Run it on as many inputs as you can, and make a conjecture about this sequence. (This is a well-known unsolved problem).

2.3 Conway’s Game of Life

In Section 2.1 we saw that a relatively simple mathematical expression could give rise to some extraordinarily complex behavior. In this section we will examine a similar phenomenon on a discrete scale.

The Game of Life was invented by the well-known mathematician John H. Conway, now at Princeton. He was working to simplify a complicated model of John von Neumann of a theoretical machine which could self-replicate. He eventually did succeed, and showed this “game” to the mathematical recreations author Martin Gardner. Gardner subsequently described The Game of Life in his October 1970 column for Scientific American. Since then it has become so popular that many have claimed it to be the most often written computer program and even the most often run computer program. It seems unlikely that the former is true, though it may place a close second to the “Hello world” program.

The Game of Life is not a game in the usual sense. It is a game in the sense that it has a board and pieces whose arrangement will change in discrete steps. However, there are no players in this game. The arrangement of the pieces follows entirely from the initial arrangement, subject to a very precise set of simple rules which mimic living organisms in a very simplistic way.

1. Pieces are placed in any arrangement on a grid of squares (the board). Theoretically, the board is infinite, but we may assume it to be simply a finite $n \times n$ grid.

2. A square (or cell) is said to be alive if it has a piece in it. Otherwise, it is dead. The neighbors of a cell are the nine cells immediately surrounding it.
3. **Simultaneously** update all cells according to the following rules:

   (a) If the cell is dead and has exactly three live neighbors, it will become alive (*birth*).
   
   (b) If the cell is alive and has exactly two or three live neighbors, it will remain alive.
   
   (c) If the cell is alive and has zero or one live neighbors it will die (*loneliness*).
   
   (d) If the cell is alive and has four or more live neighbors it will die (*overcrowding*).
   
   (e) In any other case, the cell remains dead.

4. If there are any cells which are alive, goto Step 3.

Figure 2.2 shows an example of one iteration (time step) for the Game of Life.

![Figure 2.2: Example of one time step in the Game of Life](image)

In Exercises 2.9 through 2.11 you will construct a program to run the game of life.

**Exercise 2.9** Write a program which declares a two-dimensional array of integers, `int board[64][64];`
and reads in from `stdin` (i.e., using `scanf`)

1. Two integers `0 < r, c ≤ 64`.

2. A sequence of `r × c` zeros and ones (not necessarily white-space separated).

Use the zeros and ones to initialize `board[i][j]` to zero or one respectively for `0 ≤ i < r` and `0 ≤ j < c`. 
Exercise 2.10 Add a function called `printBoard` to the program you wrote in Exercise 2.9. The definition for the function you should write begins:

```c
void printBoard(int B[][64], int r, int c)
```

This function should print the board out to the screen, printing blanks for zero entries and some other character for entries equal to one (perhaps `@` or `X`).

Exercise 2.11 Add a function called `evolveBoard` to the program you’ve written in the previous exercise. The definition for the function you should write begins:

```c
void evolveBoard(int B[][64], int r, int c)
```

This function should apply one iteration of Conway’s Game of Life to the given board. You will almost certainly find it helpful to declare a temporary array

```
int tempBoard[64][64];
```

and use this to find the new board (or the neighbor count). Perhaps something like the following might be helpful:

```c
for (i=0; i<r; i++) {
    for (j=0; j<c; j++) {
        count=0;
        for (deltaI=-1; deltaI<=1; deltaI++) {
            for (deltaJ=-1; deltaJ<=1; deltaJ++) {
                row = i+deltaI;
                col = j+deltaJ;
                if ((row >=0) && (row < r) && (col >= 0) && (col < c))
                    count += board[row][col];
            }
        }
        tempBoard[i][j] = count - board[i][j]; /* Don’t count a cell as its own neighbor! */
    }
}
```

Finally, construct a loop in the `main` function which will repeatedly call the functions `evolveBoard` and `printBoard`.

*Tip:* You can take advantage of I/O redirection to run or test this program without having to type the same input over and over. If `input.txt` is a text file containing the input you wish to supply, you can make this file behave like the input stream by redirection:

```
./myprogram.exe < input.txt
```

A sample input file called `golinput1.txt` is provided.
Exercise 2.12 (*) Modify your program to run on a torus instead of a square (i.e., identify opposite edges with matching orientation). With the input file golinput5.txt what is the maximum population size reached? After how many iterations is that maximum reached? Do the same for a Klein bottle and projective plane by identifying edges as shown in Figure 2.3.

![diagram](image)

Figure 2.3: Construction of two of the fundamental surfaces
Chapter 3

Abstract data types

This chapter describes some of the data structures which are fundamental to computer programming, regardless of the language. Loosely speaking, a data structure is simply an organizational system for data. However, different data sets and different problems are best represented in different ways. The four types we will describe here are arrays, stacks, queues and trees. These are certainly not all-inclusive, but these are the most fundamental types.

3.1 Arrays

An array is simply a finite, sequentially indexed list of elements. It can be thought of as a row of boxes with sequentially indexed addresses, where each box can hold one item. This object is so useful that nearly all computer languages implement it as a built-in type. We have already seen how to declare and use arrays in C. The declaration syntax is quite simple:

\[
data-type \text{ identifier}[\text{array-size}];
\]

This declares an array which holds array-size objects of type data-type. So, for example, the declaration

\[
\text{int A[20];}
\]

declares an array which holds 20 variables of type int, indexed from zero to 19. The uses for arrays are almost endless, and the reader can no doubt think of several already (if not, look back through the notes for some examples).

There are certain operations which are very natural to perform on arrays. These include
• Sorting.
• Searching.
• Concatenation.

Sorting is a very lengthy topic that we don’t wish to dive too much into here, but we will give a simple method for sorting an array. Suppose we have an array \(a_0, a_1, \ldots, a_{n-1}\) of \(n\) elements together with a total ordering, and we wish to re-arrange them so they are sorted in ascending order. One way to accomplish this is as follows:

• If \(a_0 > a_1\), then swap \(a_0\) and \(a_1\).

• If \(a_0 > a_2\), then swap \(a_0\) and \(a_2\).

• ...

• If \(a_0 > a_{n-1}\), then swap \(a_0\) and \(a_{n-1}\).

At the end of this process, we will be guaranteed that \(a_0\) is the smallest item in the list. We then repeat for the next element:

• If \(a_1 > a_2\), then swap \(a_1\) and \(a_2\).

• If \(a_1 > a_3\), then swap \(a_1\) and \(a_3\).

• ...

• If \(a_1 > a_{n-1}\), then swap \(a_1\) and \(a_{n-1}\).

Following this, \(a_1\) will be the second smallest item in the list. It is clear that if we repeat this process, we will finish with a list which is completely sorted in ascending order. Equally clear is the fact that this will require \(O(n^2)\) operations to complete. But wait - is that to say that if we have a list of 100,000 elements that we will need around 10,000,000,000 operations to sort it? Well, if we use this very naive sorting technique, then yes. However, there are sorting techniques which are much better than this one. The quicksort, for example, needs only about \(O(n \log n)\) operations in the average case. Furthermore, there is an implementation of quicksort in the standard C library so that we need not implement it ourselves! There are a few caveats to using it, but it is still much easier than implementing it ourselves. The nice thing about the C implementation of quicksort is that it will sort any data that we can put in an array (which is basically anything, if we are clever about it). But we pay a penalty for this: We need to tell it how to compare elements in the array as well as how large each array element is (in bytes). Here is a simple example of sorting a list of integers.
/* qsortex1.c */
#include <stdio.h>
#include <stdlib.h> /* This is needed to get the function qsort(). */

/* This is the style of declaration that qsort() expects.
   It will give us two constant void pointers, and expect
   an integer: negative if X < Y,
   positive if X > Y,
   zero if X=Y.
   Since it gives is a 'void' pointer, though, it is our responsibility
   to know how to interpret the pointer. In our case, we know that
   the elements of the array are long's, so we will treat them as such. */
int cmpLongs(const void *X, const void *Y)
{ long *x=(long *)X; /* Cast X from a 'const void *' to a 'long *' */
  long *y=(long *)Y;
  if (*x < *y) return -1;
  if (*x > *y) return 1;
  return 0;
}

int main()
{ long A[100];
  int i, numMemb;
  printf("How many integers would you like to enter? ");
  scanf("%d", &numMemb);
  if (numMemb > 100) numMemb=100;
  printf("Ok. Enter %d integers:
", numMemb);
  for (i=0; i<numMemb; i++)
    scanf("%ld", &A[i]);
  printf("Sorting the data...
");
  qsort(A, numMemb, sizeof(long), cmpLongs);
  printf("Here is the sorted result:
");
  for (i=0; i<numMemb; i++)
    printf("%ld ", A[i]);
  printf("\n");
}

Searching an array for a specific element is very easy when the array is sorted. Consider,
for example, that a typical phone book may have several hundred thousand entries. Yet,
you have no trouble searching it for a specific phone number. Why? Precisely because it is
sorted. The same technique you use to look through the phone book can be formalized to something a computer can do. The basic idea is to do as follows (this is the so-called binary search).

- Suppose the list to be searched has \( n \) things. Start in the middle and set \( I \leftarrow n/2 \) (this is the maximum distance between where we are looking and where the desired item may be in the array).

- Look in the current spot - if it’s too low, move \( I/2 \) to the right. If it’s too high, move \( I/2 \) to the left. Otherwise, we’re done (we found the item, or we found where it should be).

- Do \( I \leftarrow I/2 \). If \( I \geq 2 \), goto Step 2.

This is hardly precise - what does \( I/2 \) mean if \( I \) is not divisible by 2? We leave the details for the reader. But again - we note that C does provide us with an implementation of the binary search in the standard library. The function is called \texttt{bsearch()}\, and its usage is very similar to that of \texttt{qsort()}\. It should be clear that, using this technique, we can search for an item in an array of size \( n \) using \( O(\log n) \) operations.

### 3.2 Queues

A queue is a finite, ordered list of elements together with two primitive operations \texttt{add} and \texttt{retrieve} for adding and retrieving elements respectively. Elements are retrieved from a queue in a “First-In First-Out”, or FIFO, fashion.

A queue is so named because it models a real queue, like a line of customers at the bank. Elements are added to the rear of a queue (customers arriving) and are retrieved from the front of the queue (customers being serviced).

The C language does not have a built-in data type for representing a queue, but the standard library does have two queue manipulation functions \texttt{insque} and \texttt{remque}. Nevertheless, the data type is so simple that it is probably more efficient for the user to create his or her own from scratch as needed.

**Exercise 3.1** Suppose that elements are being added to a queue with a time between arrivals which is exponentially distributed with a mean of 1.5 seconds between arrivals. Further suppose that a elements are removed from the queue (when it’s nonempty) at a constant rate of 1 element per second. What is the expected time that an element will have to wait in the queue?
3.3 Stacks

We will have little or no use for stacks in the sequel; this section is provided for the sake of completeness since this is a fundamentally important data type in computer science, and can be considered the dual-notion of a queue.

A stack is a finite, ordered set of elements together with two primitive operations push and pop for adding and retrieving elements respectively. The name is actually very descriptive: think of a stack of papers on a desk - you can add a new page to the top or take a page from the top. But these are the only two operations that are permitted. For this reason, a stack is also known as a “Last-In First-Out”, or LIFO, type; the last object added will be the first object retrieved.

There are several obvious uses for a stack: to reverse the order of a string, to remember intermediate results during a computation. There are many less obvious uses as well, the most important of which is the following: Any computation which can be performed using recursion can be performed without recursion, using a stack.

3.4 Trees

A tree is a finite connected graph with no cycles. Their usefulness in computer programming is demonstrated by the following example. Suppose we have a large amount of data which will be repeatedly searched and modified, by adding and removing elements. Since it is to be searched, we need to keep the data sorted somehow. Yet, if we are to add and remove elements, we certainly do not want to keep it in an array (otherwise, we will have to constantly move half of the elements around just to insert or remove a single element). The solution is to maintain the data in a sorted binary tree. A sorted binary tree is one which satisfies

1. Each node has at most two children and they are distinguished as Left and Right children of the given node.

2. At any given node, the Left child and its descendants are less than the node itself. Similarly, the Right child and its descendants are greater than the node itself.

It is then clear how to search such a tree efficiently, if the tree is organized efficiently (the tree is not too deep, relative to the number of nodes it has). It is also clear that the operations of adding and removing nodes can be carried out very easily, if the tree is implemented wisely. That is, adding a node amounts to adding the data somewhere and adding a pointer two it (representing an edge of the graph). Removing a node amounts to changing a small number of edges so that the resulting graph is still a sorted binary tree. The only things that take a little work is figuring out how to keep the tree ‘balanced’ (i.e., of depth not too much bigger than \( \log_2 n \) for \( n \) nodes). But this can be accomplished without too much difficulty.
Figure 3.1: Example of a sorted binary tree
Chapter 4

Elementary cryptanalysis

Loosely speaking, cryptanalysis is the discipline of breaking ciphers. The inclusion of this subject here is justified by the fact that some of the earliest programmable computers were developed for this specific purpose. In fact, the world’s very first programmable computer, Colossus, was developed specifically to help the codebreakers at Britain’s Bletchley Park in their task. For more on this amazing machine and its inventors and significance, see [1] and [2]. The Colossus computer helped the Allies break message encrypted with the German Enigma cipher on a regular basis – a task which would be nontrivial even on a modern digital computer.

4.1 The Caesar shift cipher

The most well-known cipher is the so-called Caesar shift cipher. It is a very simple method, well-known to many school children today. However, at the time of its inception when its basic method was unknown, it was possibly quite effective.

The idea is as follows: Suppose Caesar wants to relay a message to a general located some distance away. He will write the message on a piece of paper and dispatch a courier to take the message to the general. However, should the messenger be captured, the message falls into enemy hands which could lead to disaster. However, having planned ahead for such a catastrophe, Caesar and his general agreed on a secret number $1 \leq k \leq 25$ the last time they met. When they need to communicate with each other, they do as follows:

- Break the message down into a sequence of numbers, $m_1, m_2, \ldots, m_\ell \in \{0, 1, \ldots, 25\}$ corresponding to each letter.
- For each letter $m_j$, compute $\hat{m}_j = m_j + k \pmod{26}$.
- The sequence $\hat{m}_1, \hat{m}_2, \ldots, \hat{m}_\ell$ is the encrypted message to be sent. It can be mapped back onto alphabetic characters for transmission.
The receiver recovers the original message by simply reversing the process, and computing \( m_j = \hat{m}_j - k \pmod{26} \) for each \( j \).

An example with \( k = 3 \) is presented in Figure 4.1.

As noted earlier, this was probably sufficient to secure a message before the technique was known. But once the technique is known, it suffers from the obvious limitation that there are only 26 possibilities. So it is easily possible for the cryptanalyst to simply try all 26 possibilities to find the message.

Exercise 4.1 Decipher the following message which was encrypted using a Caesar shift cipher: PHPMATMPTLXR

4.2 The Vigenère cipher

When the Caesar cipher is described as in Figure 4.1, several obvious generalizations present themselves. The one which is known as a Vigenère cipher simply suggests using a longer key. That is, instead of adding the same value to each letter of the message, what about cycling through some set of values to be added? For example, we may let \( k \) be the ordered pair \((3, 5)\) and obtain:

\[
\begin{array}{c|ccccc}
\text{Input:} & H & E & L & L & O \\
+3 & +5 & +3 & +5 & +3 \\
\text{Output:} & K & J & O & Q & R \\
\end{array}
\]

Figure 4.2: Vigenère Cipher, \( k = (3, 5) \)

There is obviously nothing special about using ordered pairs as keys - we may use a key of any length we wish. Thus, there are infinitely many possible keys, and so the cryptanalyst cannot simply enumerate all the possibilities to break this cipher.

However, for this to be a practical pencil-and-paper cipher, the sender and receiver should be able to remember the key (after all, if the key is written down and someone finds it out, the messages are again compromised). A simple technique for doing this is to let the key be given by a word or phrase. For example, we may let the key be given by the word MATH.
4.2. THE VIGENÈRE CIPHER

After a numerical translation, this becomes \( k = (13, 1, 20, 8) \), and we may use this key for encryption and decryption:

<table>
<thead>
<tr>
<th>Input:</th>
<th>C</th>
<th>R</th>
<th>Y</th>
<th>P</th>
<th>T</th>
<th>O</th>
<th>I</th>
<th>S</th>
<th>F</th>
<th>U</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+M</td>
<td>+A</td>
<td>+T</td>
<td>+H</td>
<td>+M</td>
<td>+A</td>
<td>+T</td>
<td>+H</td>
<td>+M</td>
<td>+A</td>
<td>+T</td>
</tr>
</tbody>
</table>

| Output: | P  | S  | S  | X  | G  | P  | C  | A  | S  | V  | H |

Figure 4.3: Vigenère Cipher, \( k = 'MATH' \)

Let us now play the role of the cryptanalyst. Suppose we have seen the ciphertext, PSSXGPCASVH, and wish to decipher it without knowing the key. Further suppose that we have managed to some deduce (or guess) that they key has length four.

At first glance, it would appear that there are \( 26^4 = 456976 \) possibilities for the key. Even with several friends helping, it could take quite a long time to try all of those possibilities. However, if we were to guess that the key is actually a word in the English language, the number of possibilities is reduced to a manageable number of about 2200 or so (based on the number of four letter words in a particular 45,000 word dictionary). If we were to further guess at likely keys based on the context (for example, as a mathematician I might, perhaps, choose a ‘mathy’ word), we could possibly find the key without trying 2200 of them. (For a more dramatic example, consider that \( 26^5 \approx 11.8 \) million, yet there are only 4200 five letter words in the same dictionary!).

![Frequency distribution (in percents) of single letters in English text](image)

Figure 4.4: Frequency distribution (in percents) of single letters in English text

Figures 4.4, 4.5 and 4.6 were generated using a sample text consisting of “Moby Dick” (H. Melville), “The Adventures of Sherlock Holmes” (A. Doyle), “Up From Slavery” (B. Washington), “Frankenstein” (A. Shelly) and several other literary works. In total, the combined text consisted of 1,086,286 words and about 6 million letters. The chosen texts are perhaps not representative of randomly chosen English text, but they are a reasonable
Figure 4.5: Freq. dist. (in percents) of common digrams in English text with spaces approximation. These tables show clearly that English text does not consist of randomly chosen symbols (for that matter, the same is true of almost any written language). This opens a wide avenue of attack on ciphertext which is known (or believed) to decode to English text.

For example, with this frequency distribution information in hand, we are in position to present a fairly straightforward method of attacking the Vigenère cipher when a sufficient amount of ciphertext is given. Loosely, the idea is the following: Assume that there are only a small number of possible key lengths, say 10. For $K = 1 \ldots 10$, assume the key length to be $K$ and find the most likely plaintext in that case. We may then examine each of the 10 possibilities and decide which, if any, is correct.

Now, assuming the key length is $K$, what is the “most likely plaintext?” We want to find the shifts that, when applied to the ciphertext, give letters which look most like English. We can make this well-defined in the following way: given a string of letters let $f_A, f_B, \ldots, f_Z$ denote the frequency with which A,B,\ldots, Z occur, respectively. We define the \textit{absolute variation} of this string from English text to be the quantity

$$v = |f_A - F_A| + |f_B - F_B| + \cdots + |f_Z - F_Z|,$$

where $F_*$ is the known relative frequency of * in English text (i.e., $F_A = 0.081$). The quantity $v$ is meant to measure roughly how far the distribution of letters in the string varies from the distribution of letters in English text. If the plaintext is sufficiently long, we would expect $v$ to be very small if the string is English, and not very small otherwise. So for each of the $K$
4.2. THE VIGENÈRE CIPHER

positions, we should find the shift which minimizes this variation. We declare the resulting
plaintext to be the most likely plaintext for the given key length.

Here we present the above description more formally.

Algorithm 4.2.1 VigenereBreak

Input: Ciphertext, \( c_0, c_1, \ldots, c_{N-1} \in \mathbb{Z}/26\mathbb{Z} \), and a supposed key length \( K \).

Output: The most likely plaintext (with respect to a particular probability measure).

1. Set \( i \leftarrow 0 \) (\( i \) is the key position we will examine).

2. (Prepare to examine positions \( i, i + K, i + 2K, \ldots \))
   
   Set \( \delta \leftarrow 10000 \) (an arbitrary, but large number. \( \delta \) will be the score of the best candidate so far). Set \( s \leftarrow 0 \) (\( s \) is the shift we will try).

3. (Shift positions \( i, i + K, i + 2K, \ldots \) by \( s \)).
   
   Set \( j \leftarrow i \) and \( \ell \leftarrow 0 \). While \( j < N \) do the following:
   
   \( \bullet \) \( p_\ell \leftarrow c_j + s \mod 26. \)
   
   \( \bullet \) \( \ell \leftarrow \ell + 1. \)
   
   \( \bullet \) \( j \leftarrow j + K. \)

4. Compute the relative frequencies \( f_0, f_1, \ldots, f_25 \) of \( 0, 1, \ldots, 25 \) respectively in the \( \{p_0, p_1, \ldots, p_{[N/K]}\} \)
   
   just obtained. (\( f_0 \) is the frequency of A’s in the derived text, \( f_1 \) is the frequency of B’s, and so on).
5. (Compute the variation of $p_1, p_2, \ldots$ from English text).
   Set $v \leftarrow |f_0 - F_A| + |f_1 - F_B| + \cdots + |f_{25} - F_Z|$, where $F_*$ is the relative frequency of the letter * occurring in English text.

6. (Is this a new ‘best candidate’?)
   If $v < \delta$ set $\delta \leftarrow v$ and $\sigma_i \leftarrow s$.

7. (Try next shift in this position)
   Set $s \leftarrow s + 1$. If $s \leq 25$ goto Step 3.

8. (This position is done. Look at the next position)
   Set $i \leftarrow i + 1$. If $i < K$, goto Step 2.

9. (Output most likely text)
   The most likely (inverse) key is $(\sigma_0, \sigma_1, \cdots, \sigma_{K-1})$. For $j = 0 \ldots N - 1$, set $p_j \leftarrow c_j + \sigma_j \mod K \mod 26$. Output $p_0, p_1, \ldots, p_{N-1}$ as the most likely plaintext.

Consider why this technique is interesting. The brute force approach could be used is to simply enumerate all possible keys, and one could manually look at each of the resulting strings to find one which is English. If the key length were 1, we would have to look at 26 possibilities which is easy. If the key length were 2, we would have to look at $26^2 = 676$ possibilities, which is starting to look unattractive. If the key length were 3, though, we might have to look at $26^3 = 17576$ possibilities, which is horrible to even think about. At that point, we would want some technique to weed out unlikely possibilities and have the computer look at them for us. This is the first benefit of the algorithm we’ve described - it does weed most of them out for us. But if the key length were 7, the computer would then have to examine about 8 billion possibilities by the brute force approach! That is, the brute force approach of enumerating all possible keys needs about $O(26^K)$ operations, and so it grows exponentially. (Here, we are using the standard big-Oh notation that $f(x) = O(g(x))$ if there is a constant $c$ so that $f(x) \leq cg(x)$ for all sufficiently large $x$.)

Compare the $O(26^K)$ brute force approach with the method described above, which examines each key position separately. It needs about 26 operations to find the first key position, 26 to find the second, and so on. Asymptotically, it needs about $O(K)$ operations which is much much better than $O(26^K)$. Even if the plaintext is short, we can modify the technique to look at all combinations of the two most likely shifts in each position, which would require only $O(2^K)$ operations.

**Exercise 4.2** Using the technique outlined above (or any reasonable technique of your own design), decipher the following Vigenère encrypted ciphertext:

```
CNIVZSODVOCZNVBKRULQARHGGAEQRDLXSMVPVDHOCXBFAVKYTCCYXCUFRUMKNTKZPKLVNKBCYQ
```

This ciphertext is on the course web page, along with the following test case:
4.3 Monoalphabetic homophonic substitution

This string was encrypted with the key ARC.

| Plaintext: | T | H | I | S | I | S | A | S | A | M | P | L | E | M | E | S | S | S | A |
| Key:       | A | R | C | A | R | C | A | R | C | A | R | C | A | R | C | A | R | C | A |
| Ciphertext:| U | Z | L | T | A | V | B | K | D | N | H | O | F | E | H | T | K | D |

4.3 Monoalphabetic homophonic substitution

The attack on the Vigenère cipher demonstrated in the previous section effectively kills it. What made the attack so successful is the knowledge of the underlying structure of the plaintext. That is, since E occurs much more frequently than other letters in English text, we may suppose that the ciphertext letter occurring most frequently in a given position corresponds to E. More generally, what makes the attack work is that the distribution of letters in English text is very peculiar, containing many peaks and valleys. If the letters in the English language happened to be uniformly distributed, the attack described would no longer work. This is the goal of monoalphabetic homophonic substitution - to make the distribution of letters in the ciphertext look closer to uniform, by expanding the alphabet and using the extra symbols to disguise the more common letters.

The idea is as follows: Suppose $S = \{A, B, \ldots, Z\}$ is the alphabet of the plaintext and $F_s$ is the frequency with which the letter * occurs. Choose a larger superset $S' \supset S$ and identify each $s \in S$ with a subset $C_s \subset S'$ so that

1. The $C_s$ are nonempty and pairwise disjoint.
2. The number of elements in each $C_s$ is as close to $F_s \cdot |S'|$ as possible.
3. $\bigcup_{s \in S} C_s = S'$.

To encrypt a message, $m_1, m_2, \ldots, m_N$, replace each $m_i$ with a randomly chosen $c_i \in C_{m_i}$. This is a many-to-one process, but it is clear that the message can be uniquely recovered by property 1 above. That is, when the receiver sees $c_i$, he or she can determine uniquely which $C_m$ it is in, and so recover the original message. To encrypt a message, $m_1, m_2, \ldots, m_N$, replace each $m_i$ with a randomly chosen $c_i \in C_{m_i}$. This is a many-to-one process, but it is clear that the message can be uniquely recovered by property 1 above. That is, when the receiver sees $c_i$, he or she can determine uniquely which $C_m$ it is in, and so recover the original message.

Example 4.3.1 Let $S = \{A, B, \ldots, Z\}$ and $S' = S \cup \{\#\}$.
Then we may encrypt the message

| Plaintext | H | E | R | E | I | S | A | N | O | T | H | E | R | M | E | S | S | A | G | E |
| Ciphertext | C | Z | X | A | K | E | N | # | D | H | C | A | X | Y | Z | E | E | N | B | Z |

The important thing to notice in the above example is that we have ‘disguised’ the letter E somewhat. If the plaintext were sufficiently long, we would expect that the letters A and Z would each occur with a relative frequency of about 0.06 in the ciphertext, and that no letter would occur with a frequency of about 0.12. This would already be enough to greatly complicate a simple frequency-based attack. But it is clear that if we would enlarge the alphabet by say another 10 or 12 symbols, we could make the resulting distribution even closer to uniform and completely defeat the simple attack.

However, even if we enlarged the alphabet quite a bit, there are still several possible attacks. In the same way the distribution of single letters in the English language has distinguishing characteristics, the distribution of pairs of letters or digrams is even more so. If we looked at pairs of letters from a random string, we would expect each digram to occur with a relative frequency of about $1/26^2$. However, this is not what happens in English at all! For example, some digrams like QM, YJ, JG never appear at all. Others, like TH, HE and AN occur with frequency much more than $1/26^2$. This fact, together with the distributions in Figures 4.5 and 4.6 can be used to create an attack on this type of substitution.

Exercise 4.3 The following message was encrypted with a monoalphabetic homophonic substitution. Clearly describe and justify an algorithm for breaking this cipher in general (assuming English plaintext), and use it to recover the plaintext for this message (you can also copy-paste it from the web).

```
PDXB#GTCTUGJKPQ*FPW*FCKXB#ECSGKGQUJYBIBSVNC*U*FPW UZCTIPWCGKIIBKQTGY#NGYPBKWTUPKBTJUTRUIG#WCPYLWKG B*IBSVNCYUNXYTPEPNGVCTPBFlBCECTLP*FGNPYYNSG*FGK JWBSFCNVDTSBIFBGSVYCTPYTCGNNXCECTW*BBJIFGKIUVZ TPBJXB#IGKKBBLH#PYTUGJKPQ*FPWGKJQU*WYG#UJLTPYPKQP *#VWXB#IGKY#TKPYPKBKYPSC```
4.4 Permutation ciphers

Of the ciphers we have seen so far, all shared the property that the $n^{\text{th}}$ letter of ciphertext somehow corresponds to the $n^{\text{th}}$ letter of plaintext. What if, in addition, a permutation is added? For example, a Sunday jumble problem might read:

OLEHL HETER

This is easily determined to correspond to the original message HELLO THERE.

There are many ways to make systematic use of permutations in ciphers. We consider here the most straightforward way to do it: A fixed permutation is agreed upon in advance (i.e., as part of the cipher key, or determined from the key), and applied to the plaintext in blocks. For example, consider the permutation $(1 4 2 3 5) \in S_5$. Here we are using the standard notation to describe the permutation

\[
(1 4 2 3 5) = \begin{cases} 
1 &\mapsto 4 \\
4 &\mapsto 2 \\
2 &\mapsto 3 \\
3 &\mapsto 5 \\
5 &\mapsto 1
\end{cases}
\]

To apply this permutation to some plaintext, we could first pad it with random letters so that the total length is a multiple of 5, and then apply the permutation block-by-block.

<table>
<thead>
<tr>
<th>Original message:</th>
<th>M A T H I S F U N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Padded message:</td>
<td>M A T H I S F U N W</td>
</tr>
<tr>
<td>Permutation:</td>
<td>1 4 2 3 5 1 4 2 3 5</td>
</tr>
<tr>
<td>Permuted message:</td>
<td>I H A M T W N F S U</td>
</tr>
</tbody>
</table>

How is a cipher like this to be cryptanalyzed? As before, we may follow the generic rules:

1. Choose an upper bound $N$ on the length of the permutation.
2. For $\ell = 1 \ldots N$, assume the permutation length to be $\ell$ and find the most likely plaintext.
3. Inspect the $N$ candidates and determine which is correct.

We may then attempt the brute force approach, to simply try all possible permutations. This is an exceedingly bad idea, however, since the number of permutations of length $n$ is $n!$ which grows fast, to say the least. If the permutation length happens to be 25, for
example, there are $15,511,210,043,330,985,984,000,000$ permutations with that length. As cryptanalysts, we would very much prefer to have a better method.

Notice that if we do brute force through permutations, we will waste much time. For example, if a given permutation of length 25 is clearly wrong in the first 5 positions, there are $20!$ other permutations which will give the same result in the first 5 positions, and so they are all wrong - we don’t even need to try them. This observation gives rise to a dictionary attack.

1. Make a dictionary so that any word in the plaintext is most likely in the dictionary (i.e., a regular English dictionary, perhaps with some technical words or abbreviations or such things added).

2. As per the earlier observation, assume the length of the permutation is $\ell$.

3. (Try to guess the first word) For each word (or partial word of length $\ell$) in the dictionary, see if all of its letters occur in the first block of $\ell$ letters. If not, try the next word, and continue until such a word is found.

4. Find the restrictions on a permutation that will put the given word in the given position (i.e., make an array of size $\ell$ to describe the inverse permutation, and fill in the first $\ell$ coordinates). Repeat this process, attempting to guess the second word (or partial word), until all $\ell$ coordinates of the permutation are filled in, or until they cannot be filled in. If an entire permutation was found in this way, report it as a candidate.

5. If there are still words in the dictionary which have not been tried in the first position, goto Step 3.

6. Examine all of the candidates to see if any are correct (all will decode to a list of words from the dictionary, so there won’t be many, but some may be gibberish - for example, a permutation of the words themselves within a sentence).

A similar attack could be derived to use simply the distribution of digrams in English as a scoring mechanism. In fact, such an attack would likely be preferable to the one we’ve described here as it would be tolerant of misspelled or unknown words not in the dictionary. It is, however, not so straightforward to implement.

Exercise 4.4 The following was encoded by a permutation cipher. Use any method you can to recover the original text.

```
METURPTACOPINIEHRSNEARTOYEAHVDRROAEBTKEOHNVGUEETUN
HHBFROPOESSEBEPLIMRTTOIAUSNHIBTGMONRNSIMNALWLCBR
```

*Hint:* You can find the possible permutation lengths by looking at the total length of the message.
Exercise 4.5 Suppose a message is encrypted by first applying a Vigenère cipher, then following that with a permutation cipher. Is this secure? What if the order is reversed: permutation cipher first, Vigenère second? Describe a method of attack for both.
References


Chapter 5

Efficient algorithms

5.1 Quicksort

In Section 3.1, we looked at an obvious sorting algorithm which sorts \( n \) items using \( O(n^2) \) operations. In that same section, however, we mentioned the library function `qsort` which is an implementation of the Quicksort algorithm. The Quicksort algorithm will sort \( n \) items with an expected \( O(n \log n) \) operations, in the average case.

The overall idea is easy:

- Choose a pivot position.
- Reorder the elements so that everything to the left of the pivot is less than the pivot and everything to the right is greater (or equal) to the pivot. The pivot element now partitions the array into two pieces, LEFT and RIGHT (possibly with one of them being empty).
- Recursively apply this same procedure to LEFT and RIGHT.

The ‘reordering’ is done in a clever way, as to keep the overall number of operations in that step as small as possible (in the average case). Suppose the items to be sorted are living in an array, \( A[0], \ldots, A[n-1] \). Then here is how we proceed.

1. If \( n \leq 1 \), there is nothing to do, so stop.
2. Set \( P \leftarrow A[0] \) and \( p \leftarrow 0 \). \( P \) will be the pivot element and \( p \) the pivot position. Set \( L \leftarrow 1, R \leftarrow n - 1 \).
3. Find the largest index \( i \in [L, R] \) so that \( A[i] < P \). If such an \( i \) is found, set \( A[p] \leftarrow A[i] \), then \( p \leftarrow i \) and \( R \leftarrow i - 1 \). Otherwise set \( R \leftarrow L - 1 \).
4. Find the smallest index \( i \in [L, R] \) so that \( A[i] \geq P \). If such an \( i \) is found, set \( A[p] \leftarrow A[i] \), then \( p \leftarrow i \) and \( L \leftarrow i + 1 \). Otherwise, set \( L \leftarrow R + 1 \).

5. If \( R \geq L \), goto Step 2.

6. Set \( A[p] \leftarrow P \). Recursively apply the same procedure to \( A[0], \ldots, A[p - 1] \) and \( A[p + 1], \ldots, A[n - 1] \).

Here is the file `qsort.c` containing an implementation of this algorithm to sort integers.

```c
/* qsort.c
   Chris Monico, 4/26/04.
   Sample implementation of the Quicksort algorithm.
   Note that this is also implemented by the standard library
   as qsort.
*/
#include <stdio.h>

void quickSortInt(int *A, int n)
{
    int i, p, P, L, R;

    if (n<=1) return;

    p=0; P=A[p]; L=1; R=n-1;

    while (R >= L ) {
        /* Scan \([L,R]\) right-to-left, looking for an element smaller than the pivot. */
        i=R;
        while ((i>=L) && (A[i] >= P))
            i--;
        if (i>=L) {
        }
        R=i-1;

        /* Scan \([L,R]\) left-to-right, looking for an element greater than the pivot. */
        i=L;
        while ((i<=R) && (A[i] <= P))
            i++;
        if ((i <= R) && (A[i] > P)) {
        }
        L=i+1;
    }
}
```
5.1. QUICKSORT

A[p]=P;
quickSortInt(A, p); /* Sort the left. */
quickSortInt(&A[p+1], n-1-p); /* Sort the right. */
}

int main(int argC, char *args[])
{
    int array[512], size, i;

    printf("How many elements to be sorted? ");
    scanf("%d", &size);
    printf("Enter the elements: ");
    for (i=0; i<size; i++)
        scanf("%d", &array[i]);

    quickSortInt(array, size);
    printf("Result: ");
    for (i=0; i<size; i++)
        printf("%d ", array[i]);
    printf("\n");
}

What makes the Quicksort algorithm interesting is the number of operations it needs, in
the average case. Suppose an array of $N$ elements is to be sorted and let $f(N)$ denote the
expected number of operations needed by Quicksort for random input. The first reordering
process will compare each of $N-1$ elements to the pivot once, for about $N$ operations. In
the average case, the expected location of the pivot is in the middle. Thus, each of the two
recursive calls will require $f(N/2)$ operations. Thus,

$$f(N) = N + 2 \cdot f(N/2).$$ (5.1)

Exercise 5.1 Show that Equation 5.1 together with the initial conditions $f(x) = 1$ for
$x \in [0, 1]$ gives $f(N) = O(N \log N)$. 
Appendix A

Solutions to selected exercises

Exercise 1.6: we will give two solutions for this problem.

First, the classical solution. Observe, however, that our statement is slightly different than the typical statement of this lemma, in that we don’t insist that the denominator $v$ be positive. Thus, a modification of the standard proof is necessary. One way to do this is as follows: Let $u, v \in \mathbb{Z}$ with $v \neq 0$ and consider the set $S = \{u - nv|n \in \mathbb{Z} \text{ and } u - nv \geq 0\}$. If $v > 0$, then $u - nv > 0$ for sufficiently small $n$ and if $v < 0$ then $u - nv > 0$ for sufficiently large $n$. Thus, $S$ is nonempty, and so it contains a least element $r$. Let $q \in \mathbb{Z}$ so that $u = qv + r$, with $0 \leq r < |v|$. We must show that $r < |v|$. Suppose to the contrary that $r \geq |v|$. Then $r - |v| \geq 0$ and so $0 \leq r - |v| = u - v(q + |v|/v)$. Since $|v|/v = \pm 1 \in \mathbb{Z}$ it follows that $0 \leq r - |v| = u - v(q + |v|/v) = \tilde{r} \in S$. But $\tilde{r} < r$, contradicting the assumption that $r$ was minimal. Therefore $r < |v|$.

For uniqueness, suppose

\[ u = q_1v + r_1, \text{ with } 0 \leq r_1 < |v| \]

and $u = q_2v + r_2$ with $0 \leq r_2 < |v|$. Then $q_1v + r_1 = q_2v + r_2 \Rightarrow r_1 - r_2 = v(q_2 - q_1)$, so $|v|$ divides $|r_1 - r_2|$. But $|r_1 - r_2| < |v|$, so $|r_1 - r_2| = 0$, and hence $r_1 = r_2$. Then $q_2v = u - r_2 = u - r_1 = q_1v$, so $q_1 = q_2$.

To see that uniqueness need not hold with the relaxed condition that $|r| < |v|$ instead of $0 \leq r < |v|$, it suffices to produce a single example where uniqueness fails. To that end, let $u = 20, v = 3$. Then $u = 6v + 2$ and $u = 7v - 1$, yet both remainders satisfy the relaxed condition.
We give now a second argument, whose purpose is to show that there is often a completely
different way to attack even an elementary problem. For \( u, v \in \mathbb{Z} \) with \( v \neq 0 \), we define
\[
Q(u, v) = \{ q \in \mathbb{Z} | u = qv + r \text{ for some } 0 \leq r < |v| \}.
\]

Fix some \( 0 \neq v \in \mathbb{Z} \). We will show by induction that \( Q(n, v) \) is nonempty for all \( n \in \mathbb{Z} \).
Certainly \( 0 \in Q(0, v) \).

(Positive direction) Suppose now that \( Q(n, v) \) is nonempty for all \( 0 \leq n \leq N \) and let \( q \in Q(N, v), r \in \mathbb{Z} \) so that \( u = qv + r \) and \( 0 \leq r < |v| \). If \( r + 1 < |v| \), then
\[
N + 1 = qv + (r + 1) \quad \text{and} \quad 0 \leq r < |v| \Rightarrow q \in Q(N + 1, v).
\]

On the other hand, if \( r + 1 \geq |v| \), then \( r + 1 = |v| \) and it follows that
\[
N + 1 = qv + (r + 1) = qv + |v| = \left(q + \frac{|v|}{v}\right)v + 0 \Rightarrow \left(q + \frac{|v|}{v}\right) \in Q(N + 1, v).
\]

By induction, it follows that \( Q(n, v) \) is nonempty for all \( n \geq 0 \).

(Negative direction) Suppose now that \( Q(n, v) \) is nonempty for all \( n \geq N \). We must show that \( Q(N - 1, v) \) is nonempty. Let \( q \in Q(N, v) \) and \( r \in \mathbb{Z} \) so that \( u = qv + r \) and \( 0 \leq r < |v| \). If \( r - 1 \geq 0 \), then
\[
N - 1 = qv + (r - 1) \quad \text{and} \quad 0 \leq r < |v| \Rightarrow q \in Q(N - 1, v).
\]

On the other hand, if \( r - 1 \geq 0 \) it must be the case that \( r = 1 \), so
\[
N - 1 = qv - 1 = qv - |v| + |v| - 1 = \left(q - \frac{|v|}{v}\right)v + (|v| - 1),
\]
and certainly \( |v| - 1 < |v| \), so \( \left(q - \frac{|v|}{v}\right) \in Q(N - 1, v) \).

It follows that \( Q(n, v) \) is nonempty for all \( n \in \mathbb{Z} \). Since \( v \) was an arbitrary nonzero integer, it in fact follows that \( Q(u, v) \) is nonempty for all \( u, v \) with \( v \neq 0 \).

Uniqueness can then be shown exactly as in the first proof.

**Exercise 1.9** Let \( P_1, \ldots, P_n \) be points in the plane so that the edges \( P_1P_2, \ldots, P_{n-1}P_n, P_nP_1 \) form a simple \( n \)-gon \( \mathcal{P} \). Let \( Q \) be another point in the plane not lying on \( \mathcal{P} \). The question asks for an algorithm for determining if \( Q \) is interior or exterior to \( \mathcal{P} \). We will sketch two solutions.

**Solution 1:** Project an infinite ray of slope 0 in the positive \( x \)-direction from \( Q \), and count the number of edges of \( \mathcal{P} \) this ray intersects. Then \( Q \) is an interior point if and only if the number of such intersections is even. To prove this, invoke the Jordan Curve Theorem as follows: Choose a point on sufficiently far along this ray to that it is exterior and the infinite segment to the right is entirely exterior to \( \mathcal{P} \). Follow the ray from this point toward \( Q \),
counting the number of intersections with edges of \( P \). The first time an intersection occurs, you will have moved from the exterior to the interior. The second time, you will have moved from the interior to the exterior, and so on. This argument can be made completely rigorous with little difficulty.

Solution 2: Construct the unit vectors

\[
v_j = \frac{P_j - Q}{\|P_j - Q\|},
\]

for \( j = 1 \ldots n + 1 \) where \( n + 1 \) is taken to be \( P_1 \). Compute \( \theta_j = 2 \sin^{-1} \left( \frac{\|v_{j+1} - v_j\|}{2} \right) \), the signed angle between the vectors for \( j = 1 \ldots n \). Then \( Q \) is interior if and only if \( \theta_1 + \cdots + \theta_n = \pm 2\pi \).

To prove this, identify points in \( \mathbb{R}^2 \) with points in \( \mathbb{C} \) in the natural way and consider the meromorphic function \( f(z) = \frac{1}{z - Q} \). Then \( Q \) is interior to \( P \) if and only if

\[
\int_P \frac{dz}{z - Q} = \pm 2\pi i.
\]

Then show that \( \theta_1 + \cdots + \theta_n = \pm 2\pi \) iff \( P \) is path homotopic to a circle enclosing \( Q \) in the topological space \( \mathbb{C} \setminus \{Q\} \). Note that this will actually show that \( \theta_1 + \cdots + \theta_n = 0 \) if \( Q \) is exterior. Also note that the necessary path homotopy can be obtained as follows: choose a circle about \( Q \) with sufficiently large radius as to enclose \( P \). Then project a simple closed path along \( P \) to a path on the circle and the result will follow after some details.

Although it appears at first that the first method might always be superior, there is one important difference between the two which justifies our giving the second method. In the first method, there will be a large number of statements which are executed conditionally. That is, we have to do some sort of if...then operation with each edge. While coding this in software is no problem, it does present difficulties for modern processors which try to predict the direction such branches will take. Worse yet, this can make the actual amount of CPU time required to perform such a test difficult to predict. The latter method has the advantage that it simply performs a series of calculations (which could be easily hardwired into hardware for example), and tests a single condition at the end. It could be the case, in some instances, that this is preferable. But it does illustrate some subtle facts which can have an important impact when deciding between algorithms.

Exercise 2.8 The sequence defined in this problem is called the Collatz sequence, after Lothar Collatz who, as a student, asked the still open question: Does this sequence eventually terminate at 1 for all initial values \( a_1 \)?

There are a few things which are trivially proven about this sequence. For example, if \( a_1 \not\in \{1, 2, 4, 8\} \) and \( a_n = 1 \) for some \( n \), then the tail of the sequence is 16,8,4,2,1. To see this, let \( f(a) \) be the map:

\[
f(a) = \begin{cases} 
1, & \text{if } a = 1, \\
\frac{a}{2}, & \text{if } a \equiv 0 \pmod{2} \\
3a + 1, & \text{otherwise}.
\end{cases}
\]
While $f$ is not invertible, it is partially invertible on these last few terms. That is, $f(a) = 1$ if and only if $a = 1$ or $a = 2$. Similarly, $f(a) = 2 \iff a = 4$, $f(a) = 4 \iff a = 8$ and $f(a) = 8 \iff a = 16$. This is where the phenomenon stops, as there are two possibilities if we continue: we have both $f(32) = 16$ and $f(5) = 16$. This phenomenon is easy to describe in general, since $2^k$ will have two points in the pre-image of $f$ iff $2^k \equiv 1 \pmod{3}$.

A natural way to attempt a proof of the convergence of this sequence is to try to show that for any $m$ there exists a $k$ so that $f^k(m) < m$, where $f^k(m) = f(f^{k-1}(m))$. For example, suppose $m = 4u + 1 > 1$. Then $m$ is odd, so $f(m) = f(4u + 1) = 12u + 4$. This is even, so it follows that: $f^2(m) = f(12u + 4) = 6u + 2$, and $f^3(m) = f(6u + 2) = 3u + 1 < 4u + 1$. Hence, for any $m \equiv 1 \pmod{4}$ there does exist a $k$ so that $f^k(m) < m$. Certainly the same is true if $m \equiv 0, 2 \pmod{4}$, leaving only numbers of the form $4u + 3$. This same technique fails to resolve the situation in this case, however. We may thus attempt to break it into the two cases $m = 8u + 3$ and $m = 8u + 7$. These will fail, so we lift to 4 cases modulo 16; there we may eliminate 1 of 4 cases, leaving three residue classes modulo 16. We may continue this indefinitely, and indeed, we will eliminate residue classes at many steps along the way. Nevertheless, this technique seems to be doomed to failure as well. (If this direct approach would succeed, it would mean that not only do all these sequences converge, but that they stay above the initial term for no more than $c$ terms for some fixed constant $c$; it seems extremely unlikely that this could be the case. It seems more likely that there are sequences staying above the initial for term arbitrarily long).
Appendix B

Cygwin tips & tricks

This appendix is a brief reference to using Cygwin on a PC with Windows. Cygwin is free software that essentially emulates a UNIX or Linux environment. The benefit, for us, of doing this is that writing programs in a UNIX-like style is easier to learn than many others, and it is (in theory) more platform independent, so that more of what is learned carries over to other programming environments. Another important benefit is that the GNU C Compiler (gcc) which we use is free, but it works best in UNIX-like environments. Additionally, there is a large amount of mathematical software and libraries available on the internet for free, but the majority of these are written for a *NIX environment and are generally not so easy to compile in other settings.

B.1 Preface

The first thing you will absolutely need to know before proceeding is exactly where your Cygwin home directory is located. For most installations, this will be something like:

\texttt{C:\cygwin\home\user-name}

However, it may be different on networked installations. In our lab, for example, it will be mapped to our network file space and look something like:

\texttt{U:\user-name}

The point here is that when you run Cygwin, this is the subdirectory where you will start. So, for example, if you obtain a file that you want to compile under Cygwin, you should save it in this directory.
B.2 Configuration

There is at least one configuration option you will want to change to make life a little easier. Under Cygwin, the command `ls (ell - ess)` will list the files in the current directory so you can see what you have. However, it is not so easy to tell which files are executable programs, which are directories,... This change will make that easier.

First, exit Cygwin if it is already running. Using your favorite text-editor, open the file:

```
C:\cygwin\home\user-name\bashrc
```

*Note:* You may have to change the “Files of type:” setting to coax it into showing you this file as one of the choices. In this file, you will see a line that says:

```
# alias ls='ls -F --color=tty'
```

Remove the pound sign (#) from this line and save the file. Now whenever you use the `ls` command, files of different types will be shown in different colors. Additionally, directories will be shown with a trailing slash (/) character and executables will be shown with a trailing asterisk (*).

B.3 Compiling

All C programs should have a filename ending in “.c”. If not, there are complications that can arise, so make sure! Sometimes Windows will try to hide filename extensions from you when you use the file explorer. You can override the default settings for that program so it will show you the extensions, and we recommend doing this. To do so under Windows XP (other versions are similar, though you may have to poke around through the menus to find the location of the option you will change).

- Bring up the file explorer by Right-clicking on the Start button and choose “Explore”.
- From the program menu, select “Tools-->Folder Options”. (Older versions of Windows may have a “Preferences” choice somewhere).
- Select the “View” tab.
- One of the choices says “Hide extensions for known file types.” Make sure this box is not selected or checked.
- Select “Apply to All Folders” to apply this change to all directories.
Finally, here’s how to compile and run a program.

1. Make sure you and the source code file are in the same directory. Type `ls` and you should see the file you want to compile. It’s name should end in “.c”. Let’s suppose the name of the file is “foo.c”.

2. Type

   `gcc -o foo.exe foo.c`

   to produce an executable program called “foo.exe”. If there were errors, look carefully at the error messages to see what caused them. A typical error message might look like:

   `foo.c:6: error: syntax error before ‘}’ token`

   The ’6’ is the line number where the error occurred. So even if you can’t quite tell what the error is, at least you’ll know where to look.

3. If there were no errors while compiling, you should be able to do `ls` and see the executable file (in this example, foo.exe). Note that “warnings” are different from errors; if the compiler gives only warnings, it will still produce the executable program.

4. Finally, run the program. In this case, I would type:

   `./foo.exe`

   You will always need to preface the command with dot-slash; this tells the shell that you want to execute a program which is in the current directory (as opposed to the standard binary directories where `ls` and the other system commands are living.

B.4 Quick reference

There are several little tricks that can save you time when typing on the command line. We present them here along with a handful of basic commands so you have them for a quick reference.

1. If you hit the up-arrow key, you can recall commands that you typed earlier, to issue them again or modify them and issue them again. Experiment with this - it is a real time saver!
2. If you type a partial filename and hit the TAB key, the shell will attempt to complete the filename for you. If you have typed enough to uniquely identify the file, it will complete the name. If you’ve only typed enough to narrow it down to several files, it will complete as much as it can. Hit the TAB key once or twice more in this case and it’ll show you all the files whose names begin with what you’ve typed. Note: If you are running a program from the current directory, you still need to preface it with a dot-slash (./) combination.

3. The `pwd` command is available, but works a little different under Cygwin than you might expect. It tells you what directory you are in relative to the installation path, not relative to the entire filesystem.

4. To create a directory, use:

   `mkdir directory-name`

5. To delete a file use:

   `rm file-name`

6. To rename a file (or move it into a subdirectory) use:

   `mv old-name new-name`

7. To compile a program, use:

   `gcc -o output-name input-name`

   or if you need to link in the math library:

   `gcc -o output-name input-name -lm`

8. To display the contents of a file along with line numbers:

   `less -N file-name`