CSE 110 Lecture Notes

Mark–Jason Dominus

Summer, 1992

Contents

1 A Warning **1**

2 The Blackboard **1**

2.1 Bytes 1

2.2 Bytes are Enough to Represent All Kinds of Data 2

2.3 Variables and Types 3

2.4 Other Types 4

3 Functions **5**

3.1 Arguments 6

3.2 What a Function Looks Like 7

3.3 How the Program Starts 8

4 Statements **8**

5 The Assignment Statement **8**

5.1 Examples of Assignment Statements 9

5.2 Value Contexts and Object Contexts 10

5.3 A Real Program 11

5.4 A Note About Local Variables 12

6 More Operators **12**

6.1 Arithmetic 13

6.2 Assignment 13

6.3 Increment and Decrement 13

6.4 Precedence 14

7 The Preprocessor **15**

7.1 Macros 15
7.2 Include Files 16
7.3 Comments 17

8 Conditions **17**
8.1 The `if` Statement 18
8.2 Relational Operators 18
8.3 Boolean Operators 18

9 More About `if` **20**
9.1 `else` 20
9.2 Nested if–else Statements 21

10 Example Program: Solving Quadratic Equations **22**
10.1 The Program 23
10.2 Notes About the Program 24
10.3 Pointers and the `&` Operator 25

11 I Made a Mistake **25**

12 Loops **26**
12.1 `while` 26

13 More about `scanf()` **27**
13.1 A Heinous Error 28

14 Flushing Input **29**
14.1 `getchar()` 29
14.2 Character Constants 30
14.3 A Program to Count Words 30

14.4 Flushing the Input Line 31

15 More Loops **31**

15.1 `do...while` 31

15.2 `for` 32

16 Example: A Program to Compute Prime Numbers **34**

17 A Function to Swap the Values of Two Variables **35**

17.1 A First Try 36

17.2 Why the First Try Doesn't Work 37

17.3 This One Works 37

17.4 The `&` Operator Again 38

17.5 The `*` Operator 38

17.6 `swap's` Header 39

17.7 A Note About Functions that Return `void` 39

17.8 More About Prototypes 40

18 `break` **41**

18.1 Examples of `break` 41

18.2 `break` Statement Considered Harmful? 42

19 More about Object Contexts and Value Contexts **44**

19.1 A Pointer Value Points to an Object 45

19.2 The operand of `&` is in an Object Context 46

19.3 So What? 46

20 Arrays **46**

20.1 A Better Way 46

20.2 Program to Read 100 Integers from the User and Write them out in Reverse Order 47
20.3 Notes on the Program 47
20.4 Initializing Arrays 48
20.5 Character Arrays 49

21 Miscellaneous Details About Arrays **50**

21.1 Out-of-Bounds Array References 50
21.2 The Expression `a[i]` is an Lvalue 51
21.3 `[···]` has High Precedence 51
21.4 More About Strings 52

22 Declarations **53**

23 Pointer Arithmetic **53**

23.1 The Pointer Arithmetic Rule 54
23.2 An Example 54
23.3 Consequences of the Pointer Arithmetic Rule 54

24 Examples of Pointer Arithmetic **55**

25 Too Much Work **57**

25.1 No Array Values 57
25.2 The Array Value Rule 58
25.3 Implications for Pointer Arithmetic 58

26 Functions that Operate on Strings **59**

26.1 How `strlen` Works 60

27 The `NULL` Pointer **60**

28 Input Sources and Output Sinks **61**

28.1 A File Copy Utility 61

29 Operating on a Particular File **62**

29.1 Opening a File 62

29.2 Reading from a File with `getc` 63

29.3 Stream Versions of `printf` and `scanf` 64

29.4 Closing a File 65

30 Command-Line Arguments **65**

30.1 `main`'s Header 65

30.2 A Thousand Words about `argv` and `argc` 66

30.3 `argc` and `argv` in Practice 67

30.4 The `echo` Program 67

30.5 The `type` Program 67

31 Now You Know C **69**

32 `type`, Episode 1 **69**

32.1 Notes About the Design 69

32.2 What we got Done Today 70

32.3 Some Notes About the Design Process 71

33 Miscellaneous Things We Discussed **71**

33.1 Predefined Streams and the Standard Error Output 72

33.2 `continue` 72

33.3 Don't Ring the Bell 72

33.4 Don't Worry About the Output Device 73

34 Problem 2.2 (`strrev`) from the Exam **73**

34.1 A Solution 74

34.2 A Non-Solution 75

34.3 Another Solution with `strdup` 76

34.4 `free` 77

35 Comparing Characters and Strings **77**

35.1 Comparing Single Characters 77

35.2 Comparing Strings 78

35.3 Case-Insensitive Comparison 79

36 `fgets` **80**

36.1 `gets` 81

37 Debugging Facilities **81**

37.1 Stepping 81

37.2 Inspecting Variables 81

37.3 Setting Breakpoints 82

37.4 Restarting the Program 82

37.5 Setting Command-Line Arguments 82

38 Dynamic Memory Allocation **83**

38.1 `malloc` 83

38.2 `sizeof` 83

39 `structs` **84**

39.1 Creating a New Structure Type 84

39.2 Creating `struct` Variables 84

39.3 The . Operator 85

40 The Linked List **86**

40.1 The Nodes are Structs 86

40.2 Adding a New Node to a List 87

40.3 Getting the Data Back Out of the List 87

40.4 The `->` Operator 88

40.5 Other Operations on Structures 88

41 Global Variables and Type Declarations **89**

42 A Program to Count Words **90**

42.1 Notes on the Code 94

43 Doubly-Linked Lists **97**

43.1 A Program to Print a List of Numbers Forward and Backward . 97

43.2 The Code 98

43.3 Notes on the Code 101

44 Recursion **102**

44.1 The Factorial Function 103

44.2 The Factorial Function 103

44.3 The Towers of Hanoi Problem 104

45 The Tower of Hanoi **105**

46 Left-Over Language Features **107**

46.1 The `,` Operator 108

46.2 The `?:` Operator 109

46.3 The `switch` Statement 109

46.4 Block-Scoped variables 111

46.5 **true** and **false** 111

47 The Compiler and the Linker **112**

48 Divide-and-Conquer Sorting **112**

48.1 How Long Does insertion Sort Take? 113

48.2 An Improvement to Insertion Sorting 114

48.3 Recursion 114

49 Optimization and Performance **115**

1 A Warning

I think that in order to program effectively, you have to understand how a computer really works, and you also have to understand what the compiler is really doing. Mainstream computer science pedagogues disagree with me about this. Nevertheless, because I am an employee of the university and not a graduate student, I am accountable to practically nobody and so we can learn this the right way instead of the usual way.

However, that means that before we dive into the syntax of the language, we have to spend some time discussing some of the details of how a computer works, and we don't get to dive into the syntax of the language itself for a while. I think that is all right—C is a language for talking about what goes on in computers, and if you don't know what goes on in computers you won't have anything intelligent to say in C.

So if these first sections are not what you expected, please be a little patient.

2 The Blackboard

2.1 Bytes

Computer memory is like a blackboard divided into squares. The squares are called *bytes*. Each byte can hold one piece of information. Each square has a name, which is called its *address*. Like street addresses, the names are numbers.

Bytes vary in size from computer to computer, but on most modern computers each byte contains eight *bits*. The *bit* is the smallest possible amount of information. It represents the amount of information that one can store in a physical object which can be in one of only two states. For example, one bit of information suffices to describe the state of a light switch: the switch is either on or off.[1] Since there are 8 bits in a byte, each byte is in one of 2^8 states. 2^8 is 256, so we usually imagine that the information in a byte is a number between 0 and 255.

So each byte has exactly two properties: It has a value, which we can think of as a number between 0 and 255, and it has an address which tells the computer where it is on the microchip. The address is the only thing that distinguishes

[1] Objects that can be in only one state clearly can't be used to store any information. It turns out that the amount of information you can store in an object with n possible states is no more than $log_2 n$ bits. These notions were first clarified by Claude Shannon of Bell Laboratories in the middle 1950's.

two bytes, so two bytes never have the same address. Of course two different bytes can have the same value.

2.2 Bytes are Enough to Represent All Kinds of Data

Bytes only hold numbers between 0 and 255, which is too small a range to be useful. So when we want to talk about a bigger number, we use an old trick. In our number system, we have only ten symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. When we want to write a number bigger than 9, we write two or more symbols together and say that the symbols mean different things depending on where they are relative to one another. For example, the number 674 means 6 hundreds, 7 tens, and 4 units. Because the 6 contributes most to the value of the numeral 674, we say it is the *most significant digit*. Similarly, the 4 is the *least significant digit*. This is called a *place value* system because the value of each symbol depends on the place it is on the paper. We say it is a *base 10* or *decimal*[2] system because each place is worth 10 times as much as the one to its right.

We do the same thing with the bytes on the blackboard, only we use a base 256 system. If we want to represent a number bigger than 255, we use two or more bytes and let some of them be more significant than others: each byte is worth 256 times as much as the next. For example, to represent the number 674, we put 2 in the most significant byte and 162 in the least significant byte. The 162 in the least significant byte is worth 162. The 2 in the most significant byte is worth 256 times as much as a 2 in the least significant byte would be, so it is worth $256 \cdot 2$ or 512. Thus the total value of this two-byte object is $162 + 512 = 674$.

Clearly if we're going to have multi-byte objects then it's no longer sufficient to only keep track of where a certain number is in the computer's memory: we also need to keep track of how many bytes are being used to store that number. If someone writes a two-byte number into memory and then tells us where it is so that we can look at it or change it, they had better tell us how long it is, or else we might not look at all of it (which would be like someone writing the number 674 on the blackboard and us thinking that it was three numbers, a 6, a 7, and a 4), or else we might look at too much (which would be like someone writing the numbers 674 and 523 on the blackboard very close together and us thinking that they were the single number 674523). Either would be a big disaster.

Mostly the compiler keeps track of the lengths of things and handles them for you. That's one of the big reasons for having a computer language and a

[2] *Decimal* is from *Decem*, the Latin word for 'ten'

compiler in the first place.

Also we want to store other information than just numbers. Say we want to store someone's name—how do we do that with just numbers between 0 and 255? Of course the answer is that we assign a number to each letter of the alphabet, and we store the numbers that correspond to each letter in the name. It would be a big disaster if we forgot that this sequence of numbers was actually supposed to represent a sequence of letters. Another reason we have computer languages is to keep track of which numbers on the blackboard really represent letters and which ones really represent numbers.

2.3 Variables and Types

So suppose we are going to write a program that will need to remember a number–say it's a program that thinks of a secret number between 1 and 10000, and then the user tries to guess what that number is. We need to find a blank spot on the blackboard big enough to store the number. We need at least two bytes to store the number in, because it might be bigger than 255. (Two bytes are enough to store numbers between 0 and 65,535.) We also want to remember that this is a two-byte number and not the first half of a four-byte number, and not the first two letters in someone's name, or anything like that.

Here's what we say in C to accomplish all that:

```
int my_number;
```

`int` is short for 'integer', which is a mathematical word for a whole number. The actual size of an <`int`> depends on what computer and what compiler you are using, but it is never less than two bytes. When the compiler sees this instruction, it finds enough free space on the blackboard to hold an <`int`>, reserves it, and makes a note to itself that from now on, this space is going to hold an <`int`> and that the name `my_number` refers to the space.[3] You don't need to know the details about how it finds this space or where it is; all you need to know is that the space is there and that you can call it `my_number`. The semicolon (`;`) is just punctuation; it tells the compiler where the end of the statement is.

This whole process is called *declaring* a *variable*. The *name* of the variable is `my_number` and the *type* of the variable is <`int`>. A variable has two other properties: Its value (which depends only on the values of the bytes that make

[3] Actually the compiler doesn't find the space right away; rather, it writes out machine instructions into the executable version of your program that find the space when they are executed.

it up and on its type), and its address, which is the same as the address of the first byte that makes it up. Unless you say otherwise, <int> variables always start with the value 0.

In some languages you can use variables without declaring them; what happens then is that the compiler assumes that they have a certain type. You can't do that in C. If you name a variable you haven't declared, the compiler will signal an error.

2.4 Other Types

C has a few other primitive types:

- <short int> and <long int> are like <int>, but might be shorter or longer. <short int> is always at least two bytes long, and it's never longer than int. <long int> is always at least four bytes, and it's never shorter than int. How long <short int> and <long int> actually are depends on your compiler. <int> is supposed to be a size that is convenient for your computer and you should use <int> rather than <long int> or <short int> unless there's a good reason. You use <short int> to save blackboard space and <long int> when you need to handle bigger numbers.
- <char> is short for *character*; you use it when you want to store a character, which could be a letter, a digit, or any other symbol on the keyboard. It's always exactly one byte long.
- <float> is short for *floating-point number*. A floating point number is one which is represented internally in scientific notation, with a mantissa and an exponent.[4] This means that a <float> can represent a fractional number like $\frac{3}{4}$ or π. A <float> variable has a wide range, and can typically hold a number between about 10^{-38} and about 10^{38}, but it loses precision towards the ends of its range. floats on typical machines are four bytes long. There's a more precise version of a <float> called a <double>, which is twice as long and twice as accurate. Similarly, there are <long double>s which are even longer than ordinary <double>s.

That's all the simple types there are. There are other types, but nearly everything else is built up from these few.

[4] This means, for example, that a number like 1234567890 is stored as 1.234567890×10^9. 1.234567890 is the *mantissa* and 9 is the *exponent*. If you don't understand this, don't worry; we won't need it for a long time.

3 Functions

In the bad old days before high-level languages like C, people wrote in assembly language. Every part of every program could see all the bytes at once.[5] This might sound like a good idea, but what happened was that people wrote all sorts of horrible programs. You might write a program which stored some payroll information in the memory of the computer, then printed a message on the screen, and then did something else with the payroll information. You would expect that printing a message would not change the payroll information. However, it might, depending on what the person who wrote the print function thought was the best thing to do. You might find that your program had an error because the print function had twiddled the payroll information when you were not looking. Worse still, if one day someone changed the way the payroll information was stored internally, they might not rewrite the print function to inform it of that fact, and then the print function would scramble the payroll information completely. People's programs had all kinds of horrible errors because all the sub-parts of their programs were mucking around with each others' data in unpredictable ways.

In C, and most other high-level languages, you can be sure that something like this will not happen. You can know that the print function will not try to change the payroll information. The way that you know this is that you do not tell the print function where the payroll information is on the blackboard. That is, the print function can't mess with the payroll information because it simply doesn't know where that information is.

Languages these days mostly differ in the methods they provide for letting one part of a program hide its information from another. How to divide up the blackboard is a big deal. One of C's major advantages over other languages is that it is a particularly good language for talking about where things are on the blackboard.

In C, we break up programs into smaller, self-contained parts called *functions*. Each function has some instructions associated with it, and it has some variables that it gets to keep all to itself—no other functions can look at or alter those variables or even know where they are unless the function that owns them tells the other function. One fine point of programming is deciding the best way to carve up a problem into sub-parts so that the functions are as nearly self-contained as possible.

Every instruction in a C program is in one function or another. When you run a C program, the operating system takes care of starting the first function for you. This first function can start other functions, which in turn can start

[5] BASIC and FORTRAN are like this also.

other functions. When a function starts another, we say that the first function has *invoked* or *called* the second function. When a function invokes another, the operating system stops executing instructions from the first function and starts executing instructions from the second function. When the second function is finally done, control returns to the first function and it goes on with what it was doing. Thus only one function is ever executing at any time.

Of course the functions do have to communicate with each other a little, and they do that in a specific and controlled way. The way we do this in C is by *passing arguments in* to a function when it starts, and by *receiving a return value* from a function when it is finished. Starting up a function is called *calling* the function or *invoking* the function.

Only one function is ever active at any time. When a function calls another function, it pauses, and waits for the called function to finish, before it continues with what it was doing.

When a function calls another function, it decides on the *arguments*, which are some data that it will pass to the function it is calling. The called function gets to see these arguments, and the things it does might depend on them. When the called function is finished, it gets to return some value to the calling function. So the arguments are the way a function communicates to a function that it calls, and the return value is the way a called function communicates information back to its caller.

3.1 Arguments

Suppose I decide that I will have a function which computes the square of an integer. (The square of a number is what you get when you multiply the number by itself.) We'll call this function `square`. Now whatever other function calls `square` must tell it at least one thing: The number that `square` is supposed to square. The calling function will therefore communicate its number to `square` by passing in that number as an argument.

The way we call a function in C is by writing its name, an open parenthesis (`(`), the arguments, if there are any, and then a close parenthesis (`)`). For example, here's how we might call `square`:

```
square(57);
```

When the function that is executing reaches this instruction, it pauses. It doesn't go on to the next instruction until it has computed the square of 57. To do that, it saves somewhere some information about what it was doing and

transfers control to the `square` function. When `square` is finished, control returns to the calling function, which picks up where it left off. The details of how the function remembers where it left off, and how it gets back to what it was doing after the called function is finished, are the compiler's problem.

3.2 What a Function Looks Like

A function has two parts. The first part, called the *header*, says the name of the function, how many arguments it has and what they are called and what their types are, and what type of result it returns to its callers. The second part is the *body* for the function. It is enclosed between curly braces (`{` and `}`) and it contains the instructions to the compiler about what to do when the function is called.

Here's how we might write `square`:

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

This is our first real C code, and so we'll discuss it at length.

The first line is the header. A header has three parts: First, the name of the type of the return value that the function returns to its callers, in this case `int`. Then comes the name of the function, in this case `square`. Then, in parentheses (`(` and `)`) is a list of *parameter declarations*, which describe what types of values a calling function is allowed to pass in, and what to names to give to these values. In this case there's only one argument, and `int n` means that that argument will have type <`int`> and that within the body of the `square` function, the name `n` will refer to that argument.

The open curly brace ({) and close curly brace (}) *delimit* the function's body—they say where the body begins and ends. When there's more than one function in the program, the braces help the compiler understand where one function leaves off and another begins.

In between the braces is the code for the function itself. The star `*` means multiplication. `return` means to compute the value of the mathematical expression that follows, and return control to the calling function, passing it back the value of the mathematical expression. So this line computes $(n \times n)$ and immediately returns the result to the calling function, which was waiting around for that. When a function executes a `return` instruction, that means that it is done.

Normally, the instructions in a function are executed in order, from top to bottom. After an instruction is executed, control passes to the instruction on the next line. The `return` instruction is an exception to this. When a function executes a `return` instruction, it does *not* go on to the next instruction. Instead, control returns to the function that called the one that executed the `return` instruction. The next instruction that is executed is in the *calling* function, which picks up right where it left off.

3.3 How the Program Starts

The operating system arranges that if your program has a function named `main`, that function will be called first. If your program has no `main`, the linker will complain and you won't be able to run your program.[6]

When `main` executes a `return` instruction, control is returned to the operating system, and your program is over. The value you return with the `return` statement from `main` gets passed back to the operating system as a status code about whether the program succeeded or failed. Conventionally, `main` returns zero for success and nonzero for failure.

4 Statements

This is just terminology: Each instruction in a C program is called a *statement*. Statements are easy to recognize: Simple statements always end with semicolons.

You can group statements together into *blocks*, which are also called *compound statements*. A compound statement is just a bunch of other statements (which might be simple or compound), with an open brace at the beginning and a close brace at the end. To execute a compound statement, the computer just executes the statements that make it up, in order.

The body of a function is always a single compound statement.

5 The Assignment Statement

The *assignment statement* performs a fundamental operation: It copies information from one part of the blackboard to another.

[6] We'll find out why it is the linker in section 47 .

Its syntax is:

lvalue = *expression* ;

Like all simple statements, it ends with a semicolon to tell the compiler where its end is.

It has three parts: There's an *lvalue* on the left, an equals sign in the middle, and an *expression* on the right. *expression* means the same as it does in mathematics. An *lvalue* is an expression that refers to part of the blackboard—for example, the name of a variable is an lvalue; it refers to the part of the blackboard where the variable is stored.[7]

When the computer executes an assignment statement, it first *evaluates* the expression on the right; that means that it reads it and performs whatever calculations are necessary to determine its value. For example:

- If the expression is just a variable name, then the value of the expression is just the value stored in the variable.
- If the expression is a function call like `square(57)`, the computer pauses, calls the function, and the value of the expression is whatever the called function returned with its `return` statement.
- If the expression is just a numeral, like 57, then the value of the expression is just the value of that numeral.

After the computer has calculated the value of the expression, it stores that value into the part of its memory referred to by the lvalue.

5.1 Examples of Assignment Statements

We'll suppose that someone has already declared <`int`> variables called `x` and `y`.

```
x = 10 ;
```

The computer evaluates the expression on the right of the equals sign; the expression is just `10`, and so the value of the expression is the integer 10. Then the computer stores the integer 10 into the variable `x`.

```
y = x ;
```

[7] 'lvalue' is pronounced 'ell-value'.

The computer evaluates the expression on the right of the equals sign; the expression is just `x`, and so the value of the expression is the the value of the variable `x`, which is now 10. Then the computer stores the integer 10 into the variable `y`.

```
x = x + 1 ;
```

The computer evaluates the expression on the right of the equals sign. It evaluates the `x`, whose value is 10, and it evaluates the `1`, whose value is 1, and then it adds those two numbers together, because `+` means addition. The result, 11, is the value of the expression on the right of the equals sign. Then the computer stores the integer 11 into the variable `x`.

The following is *not* a legal assignment statement; the compiler will refuse to compile it:

```
4 = x ;
```

This fails because `4` is not an lvalue. That is, it does not refer to a part of the computer's memory, that way a variable name does. It is just a number. In short, you can't do this because `4` is not the name of a place where you can store a value.

5.2 Value Contexts and Object Contexts

Notice that in the statement

```
y = x ;
```

there is an asymmetry: `x` is evaluated, but `y` is not. In fact, `y`'s value is completely irrelevant—the compiler evaluates `x` because it is going to store that value in the part of memory referred to by `y`, but it never evaluates `y` at all.

`x` and `y` in this statement are canonical representatives of the two contexts that an expression can be in in C. One context is called the *value context*, and it means that the expression will be evaluated to produce a value. `x` here is in a value context. The other context is called the *object context*. It means that the expression will not be evaluated; instead, it is being used to refer to part of the computer's memory. `y` here is in an object context.

lvalue and *object context* are really two sides of the same coin.[8] Only an lvalue may appear in an object context; conversely, to decide if something is an lvalue, just see if it makes sense in an object context, such as on the left-hand side of an assignment statement.

[8] In fact, many people say *lvalue context* instead of *object context*.

For example: Is `x + y` an lvalue? No, because

```
x + y = 4 ;
```

doesn't make any sense—it says to store the value 4 into the part of the computer's memory represented by `x + y`, and that doesn't mean anything.

5.3 A Real Program

I fibbed a little about the right way to call the function `square`. Here I'll present a real program for computing the square of the number 57 and printing it out.

```
int main(void)
{
  int the_square;

  the_square = square(57);
  printf("The square of 57 is %d.\n", the_square);
  return 0;
}

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

There are two functions here: `main` and `square`. Notice that `main` is a function which returns a value of type <`int`> and which takes a value of type <`void`>. <`void`> is a special type which means "no value at all". So when the operating system first invokes `main`, it shouldn't give it any arguments. When `main` finishes, it'll return an <`int`> to the operating system to report whether it succeeded of failed.

The first line in `main` declares a variable, `the_square`, in which we will store the result of squaring the number 57. The compiler finds space for an int and arranges that if we name `the_square` again that the value in this space is used.

The second line is something we haven't seen before: An *assignment* instruction. We'll discuss it at length in section 6.2, but what it says to do is to compute the value of the mathematical expression on the right of the = sign, and that value into the part of the blackboard represented by the object on the left of the = sign. The value of a function is whatever is returned by that function with a `return` instruction.

When control reaches this point in `main`, `main` stops and waits, and control passes to `square`. The compiler has arranged that the number 57 gets copied into the part of the blackboard referred to by `n`, so that when you ask for the value of `n`, you get the number 57. `square` fetches the value of `n`, multiplies it by the value of `n`, and returns that result. The compiler arranges that the result (it happens to be 3249) gets stored into the variable `the_square`. Then control returns to `main`.

The next thing that happens is that control passes to the next line in `main`. We haven't seen the `printf` function yet, but the call we've made to it here makes it print out

```
The square of 57 is 3249.
```

We don't have to supply code for `printf`; it's already written and stored in a file called a *library*. It's the linker's job to see if we used any functions, such as `printf`, for which we didn't supply any code, and, if so, to search for them in the libraries and to include the code for them if necessary. We'll discuss this more later.

Notice that we passed two arguments to `printf`, separated by a comma (`,`): A message to print, and the value of `the_square` to fill into the message. We'll talk about the details of what is going on here soon.

Finally, `main` itself returns a value, 0, to the operating system, to indicate that it completed successfully.

5.4 A Note About Local Variables

The variables `n` and `the_square` are called *local* variables. That means they are usable only in the functions in which they're declared. If you tried to use the variable `n` in the function `main`, or if you tried to use the variable `the_square` in the function `square`, the compiler would complain and would refuse to compile your program. That way you can be sure that the functions are communicating only in the way you expect them to, by passing arguments and returning return values, and that they are not mucking around with each others' private data.

6 More Operators

C has many operators. Some of them, like `+`, are *binary*, which means that they require two operands, as in `4 + 5`. Others are *unary*, which means they require only one operand. We'll see an example of this in section 8.3.

6.1 Arithmetic

Arithmetic operators include `+`, `-`, `*`, and `/` for addition, subtraction, multiplication, and division. Division is a little odd: Its semantics change depending on the types of its operands. If both operands are <`int`>s, then `/` represents *integer division*, in which the fractional part of the result is discarded. For example, the value of the expression `13/5` is 2.

`%` denotes the *modulus* operator: If a and b are integer expressions, then a`%`b is the remainder when a is divided by b. For example, the value of the expression `13 % 5` is 3, because 3 is the remainder when you divide 13 by 5. If one of the operands is an expression whose value isn't of integer type, that's an error and the compiler won't compile the program.

6.2 Assignment

`+=` is an assignment operator. Like `=`, its left operand must be an lvalue. `x += 2` means the same thing as `x = x + 2`. It's more natural to think "Add 2 to `x`" and to write `x += 2;` than it is to think "Get `x`, add 2, and put it back." and write `x = x + 2;`. Furthermore, in an expression like

```
yyval[yypv[p3+p4] + yypv[p1+p2]] += 2
```

the assignment operator makes the code easier to understand, since the reader doesn't have to check painstakingly that two long expressions are the same, or to wonder why they're not.[9]

Similarly, there are `-=`, `*=`, `/=`, `%=` operators, all defined analogously.

6.3 Increment and Decrement

Adding one to something is such a common operation that there's a special name for it and yet another notation for it. Adding 1 to a quantity is called *incrementing* it. There's a special increment operator, `++`. If we write `x++` or `++x` in an expression, then, sometime before the computer executes the next statement, it will add 1 to the value stored at `x`.[10]

The values of `x++` and `++x` differ, however: if `x` is 12, then the value of `x++` is 12 and the value of `++x` is 13. The notion is that if you use `++x`, the

[9] I swiped this example from *The C Programming Language*, by Kernighan and Ritchie.

[10] Of course, when we say "the value stored at `x`," we are implying that `x` is an lvalue—you can't do `4++` to increment the value of 4; the compiler will complain.

compiler increments `x` before getting its value, and if you use `x++`, the compiler increments `x` after geting its value. So if the value of `x` is 119, then after

```
y = x++ ;
```

`y` would be 119 and `x` would be 120, but after

```
y = ++x ;
```

`y` would be 120 and `x` would be 120.

When the compiler actually chooses to do the increment is None of Your Business, as long as the increment happens before the next statement.[11]

Question: What happens if `x` is 119, and we do:

```
y = x++ + x++ ;
```

? Depending on when the incrementing actually happens, `y` might get 238, or 239, right? Wrong. For various technical reasons the standard says that if you try to modify the same object twice in one statement,[12] you get undefined behavior—the compiler is allowed to do whatever it likes, including nothing, printing a warning, erasing itself, or teleporting elephants into the room. Similarly

```
x = x++ ;
```

is undefined. On the other hand, if `x` is 2, then

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

is perfectly legal, (`x` and `y` are each modified only once before the end of the statement) and might assign `y` the value 4 or the value 6, depending on whether the compiler does the increment before or after it evaluates the first `x`. In general it's best to avoid such situations.

There is a `--` operator, which is just like `++`, except that it subtracts 1 instead of adding 1. This is called *decrementing*.

6.4 Precedence

Consider this expression:

[11] Actually it has to happen before the next *sequence point*. We know about two kinds of sequence points: semicolons are sequence points, and also there is a sequence point just before every function call.

[12] Really it says twice without a sequence point in between.

```
2 * 3 + 4
```

does the compiler do `2 * 3` first and then add `4`, yielding 10? Or does it do `3 + 4` first and then multiply by `2`, yielding 14?

The rules are consistent with regular mathematics: Multiplication and division (meaning `*`, `/`, and `%`) happen before `+` and `-`. So the example above evaluates to 10.

Assignment happens way late, after almost everything else, because if you write

```
x = y + 4 ;
```

you never ever mean that you want to store the value of `y` in `x` and then add 4 to that result—you always mean that you want to add 4 to the value in `y` and then store that result into `x`.

As in mathematics, expressions in parentheses (`(` and `)`) get evaluated first. So the value of

```
2 * ( 3 + 4 )
```

is 14.

`++` and `--` have higher precedence than most other things, including everything we've seen so far. That's so that the `++` in `y * x++` applies just to the `x` and not to the `y * x`.

7 The Preprocessor

Before the compiler starts in on its work in earnest, it transforms your program a little. On some systems this transformation is done by a separate program, called a *preprocessor*, but in Turbo C++, the preprocessor is built into the compiler. There are three important transformations and a few unimportant ones that we won't discuss.

7.1 Macros

If your source code contains a line like

```
#define PI 3.141592654
```

the preprocessor defines a *macro*. What this means is that from now on, every

time it sees the symbol `PI`, it will replace it with the sequence `3.141592654`. The compiler proper will never find out about `PI` at all; as far as it's concerned, you wrote out `3.141592654` in full every time.

You can use this for three things:

- You can use it to clarify the code, by writing things like `PI`, rather than `3.141592654` all over the place.
- You can use it to set up *manifest constants* that might change over time. For example, suppose you are writing accounting software for an insurance company; the medical insurance deductible is $200, and you want to compute the payment. You could write

  ```
  payment = claims - 200 ;
  ```

  but then if the deductible ever changed, you'd have to go and find all the `200`'s in your code and change every one. Worse, it might be that not every `200` is actually a deductible—you'd have to decide which ones to change. It's much better to do

  ```
  #define DEDUCTIBLE 200
  ```

  and then you can write

  ```
  payment = claims - DEDUCTIBLE ;
  ```

  and if you need to change the deductible, you just change it in the `#define` directive and nowhere else.
- You can use the macro facility to make your code into an unreadable horror. We will not see an example of how to do this.

Conventionally the names we give to macros contain only capital letters.

7.2 Include Files

If you write

```
#include <file.h>
```

the compiler immediately pauses what it was doing, seeks out a *header file* called `file.h` in some standard place,[13] and pretends that the entire contents of that file appeared in your source file in place of the `#include` directive. If the compiler doesn't find `file.h` in any of the standard places, it complaints.

[13] Just what place this is is *implementation defined*, which means that it's well-defined, and that it must be documented, but it differs from system to system.

One of the good things about C is that you can have a program whose source lives in more than one file; then if you make changes to one file you don't have to recompile all the others to make an executable. But if there's some information they need to share, you can put it in one header file and have all of the source files `#include` it; again, if the information changes, you only need to change the one copy in the header file instead of going around and changing each source file.

Similarly, when you want to use a library function like `printf`, there might be information you need to give to the compiler about that function. The people who wrote `printf` can put whatever information is necessary into a header file, and then you can include the header file in your program before you use `printf`. In fact, in order to use `printf`, you have to `#include` the header file `stdio.h`, or else you get undefined behavior.[14]

If you write `#include "file.h"` instead of `#include <file.h>`, the preprocessor looks for `file.h` in the current directory before it looks in the standard places.

Header files don't have to have a `.h` extension, but they always do.

7.3 Comments

The preprocessor provides a facility for including explanatory text, called *comments*, into your program, without confusing the compiler. The rule is this: The sequences `/*` and `*/` delimit comments. The preprocessor replaces the `/*`, the `*/`, and everything in between with blank space, which the compiler ignores.

It's too early for an enormous rant about the importance and proper style of comments, but we'll have several later.

8 Conditions

If a program did the same thing every time we ran it, it wouldn't be useful. We have to have a way to perform certain actions only when certain conditions are true.

[14] Our `square` program from section 3.2 had undefined behavior for this reason. Fortunately, it did the right thing anyway.

8.1 The `if` Statement

The `if` statement has this form:

`if (`*condition*`)` *statement*

The condition is just an expression. We say that the condition is *false* if the value of the expression is zero, and we say that the condition is *true* otherwise. To execute an `if` statement, the computer first evaluates the expression to decide if the condition is true or not. If the condition is true, the computer executes the statement. Otherwise, it doesn't.

The statement could be a compound statement, or it could even be another `if` statement.

8.2 Relational Operators

C provides operators for comparing numbers. The operator == tests two expressions for equality, for example. The expression `a == b` has the value 0 if `a` and `b` are not equal and `1` if they are equal. So we can write

```
if ( a == b )
 printf("a and b are equal\n");
```

If `a` and `b` are equal, the expression `a == b` evaluates to 1, so the condition is true, and the `printf` statement gets executed. But otherwise, `a == b` evaluates to 0, so the condition is false and the `printf` is not executed.

a `!=` *b* is true when the value of expression *a* is not equal to the value of expression *b*. Similarly, we have `<`, `>`, `<=`, and `>=`, for testing whether one expression's value is less than another's, greater than another's, less than or equal to another's, or greater than or equal to another's.

Relational operators have low precedence, just before assignments, but after arithmetic.

8.3 Boolean Operators

Suppose we want the user to enter an integer between 1 and 10, inclusive. We want to write some code to print a rude message if the user didn't do what we wanted. Suppose the user's number is stored in the variable `x` Then we could write

```
if ( x < 1 )
  <print rude message> ;
if ( x > 10 )
  <print rude message> ;
```

but then the code to print the rude message is the same both times. That's bad, because someday someone is going to change one and not the other, and then the program will have different behavior where it used to have the same behavior; or else someday both statements might break[15] and someone might fix only one of them by mistake. A principal rule of programming is to never ever have two pieces of code to do the same thing. Fortunately there's a better way to accomplish what we want:

```
if ( x<1 || x>10 )
  <print rude message> ;
```

`||` reads as 'or', so we say "if `x` is less than 1 **or** `x` is greater than 10...". `||` requires two operands, and an expression with an `||` operator is true if either of its operands are true, false if neither is true.

`||` has a special property: it *short-circuits*. In the example above, suppose the value of `x` is 0. The computer compares `x` with 1, and finds that `x < 1` is true, and so we already know that the rude message will be printed. There's no longer any reason to computer whether or not `x > 10` is true; either way we'll print the message. And in fact in C when the computer is evaluating an `||` expression, if the left-hand operand is true, then the computer never evaluates the right-hand one at all.

Similarly, there's a `&&` operator, which is pronounced 'and'. For example:

```
if ( x>=1 && x<=10 )
  <print polite message> ;
```

"If `x` is greater than or equal to 1 **and** `x` is less than or equal to 10, then print the polite message." An expression with `&&` is true if both its operands are true, false otherwise. `&&` also short-circuits, so if `x` is 0 in the example above, then the computer will only bother to evaluate the `x >= 1` part above; since that part is false, it already knows that the whole `&&` expression is false, and there's no point in evaluating the `x <= 10` part.

`||` and `&&` are called *logical operators* because they operate on logical conditions such as `x < 10`, rather than on raw numbers like `12`. There is one

[15] Maybe because the print function changed or something.

other logical operator: `!`, pronounced *not*. `!` is a unary operator; it takes only one operand. When the computer wants to evaluate something like `!`x, it first evaluates x, and then if x is true, `!`x is false, and vice-versa.

`!` has higher precedence than anything else we've seen yet. `&&` and `||` have lower precedence than anything except assignment. `&&` has higher precedence than `||`, which is consistent with conventional mathematics.[16][17]

9 More About `if`

9.1 `else`

Consider this code:

```
if ( x > 0 )
  printf("X has a positive value\n");
if ( ! (x > 0) )
  printf("X does not have a positive value\n");
```

Suppose the value of `x` is 0. We know that exactly one of the `printf` statements will always be executed, and so we know that if it isn't the first one, it must be the second. So as smart humans, we don't have to perform both tests. The computer, on the other hand, is dumber, and we have to tell it explicitly that the two `if` statements here represent an exclusive partitioning of all possible situations. We do that with an `else` clause:

```
if ( x > 0 )
  printf("X has a positive value\n");
else
  printf("X does not have a positive value\n");
```

This tells the computer to compute the value of the expression `x > 0`, and if that expression is true, then to print `X has a positive value`. Otherwise, the computer prints `X does not have a positive value`. Exactly one of the `printf` statements is ever executed each time through this code.

When we want to select one of many conditions, we use a similar construction:

[16] There is a table of operator precedence in your text, on pages 690–691.
[17] Incidentally, `&&` and `||` are both sequence points.

```
if ( profits < 0  )
  printf("The company is losing money\n");
else if ( profits == 0 )
  printf("The company exactly broke even\n");
else
  printf("The company turned a profit\n");
```

The computer evaluates the conditions, one at a time, until it finds one that is true. Then it executes the associated statement, and then skips all the rest of the clauses. If none of the conditions are true, the computer executes the statement associated with the `else` clause, if there is one.

9.2 Nested if–else Statements

Here's some code for printing out information about average test grades: `total` holds the sum of all the grades in the class, and `count` holds the number of students in the class.

```
if (count != 0)
  if ( total/count < 80 )
    printf("This class is doing badly\n");
  else
    printf("This class is not doing too badly\n");
else
  printf("There are no students in this class\n");
```

An `if` construction or an `if–else` construction is a statement, and can go anywhere a simple or a compound statement can. Here, we first check to see if `count` is not 0; if it is 0, we print `There are no students in this class`. Otherwise, we compute whether or not the average grade is below 80 and print a message depending on whether it is or not. This second `if–else` statement is said to be *nested within* the first.

We indent the nested statement to make our meaning clearer.

No suppose the final `else` and the last `printf` weren't in the example above. How does the compiler know that we meant what we did, rather than

```
if (count != 0)
  if ( total/count < 80 )
    printf("This class is doing badly\n");
else
  printf("This class is not doing too badly\n");
```

, which would print `This class is not doing too badly` whenever `count` was equal to 0.

If you said the compiler knows the difference because of the indentation, you're mistaken: the compiler ignores all white space completely. The indentation is only there for the benefit of human readers.

The C language resolves this ambiguaity by fiat: The rule is that if there's more than one `if` that an `else` could go with, it goes with the nearest one. So the `else` above matches with the second `if`, not the first. If for some reason you really wanted the meaning suggested by the second indentation, you could write this:

```
if (count != 0) {
  if ( total/count < 80 )
    printf("This class is doing badly\n");
} else
  printf("This class is not doing too badly\n");
```

The curly braces here delimit the statement that follows the first `if`. That statement doesn't include the `else` clause, so the compiler can't construe the `else` as being under control of the first `if`—it must be parallel to it.

10 Example Program: Solving Quadratic Equations

Now we'll write a program which lets the user enter a quadratic equation. If the roots of the equation are real, the computer will find them and print them out, and otherwise the computer will print a message saying that the roots are complex. We did this in class, so here's the code:

10.1 The Program

```
/* Program to solve quadratic equations ax^2 + bx + c = 0.
   Known bugs:  Fails when a=0.
 */

#include <stdio.h>
#include <math.h>

int main(void) {
   double a, b, c;
   double x1, x2;
   double discriminant;

   /* get inputs a, b, and c from user. */
   printf("Please input a, b, and c, one per line.\n");
   scanf("%lf", &a);
   scanf("%lf", &b);
   scanf("%lf", &c);

   discriminant = b*b - 4*a*c;

   if ( discriminant < 0 ) {
     printf("The roots are imaginary.  I can't find them.\n");
     return 1;
   } else {
     x1 = ( -b + sqrt(discriminant) )  / (2*a);
     x2 = ( -b - sqrt(discriminant) )  / (2*a);

     printf("The roots are %f and %f.\n", x1, x2);

     return 0;
   }

}
```

10.2 Notes About the Program

The program's small enough that we didn't bother with any functions other than `main`.

9 We are going to need the `sqrt()` function, which takes one <`double`> argument and returns the <`double`> which is its square root. Someone else wrote the `sqrt()` function and put it in the library for us, so we don't have to supply it. But here's a problem: How does the compiler know that `sqrt()` takes a <`double`> argument and returns a <`double`> result? It needs to know, or else it won't be able to pass the argument or interpret the return value properly. The answer is that we must tell it. We `#include` the file `<math.h>`, which contains a line that tells the compiler what types the arguments and return value of `sqrt()` will have. Such a line is called a *prototype*; it looks just like a function header with no body:

```
double sqrt(double x);
```

This gives the compiler the information it needs to compile our calls to `sqrt()` corectly. Otherwise it would assume that `sqrt()` returned an <`int`> value, and all sorts of nastiness would ensue. We didn't have to write this, because the authors of `sqrt()` put it in `<math.h>` for us, so all we have to do is `#include` that file.

The `#include <stdio.h>` is there for a similar reason, to tell the compiler what kinds of arguments and return types `printf()` and `scanf()` have.

13 The declaration

```
double discriminant;
```

is analogous to the `int foo;` declarations we've seen, but it allocates enough space for a variable of type <`double`> instead of one of type <`int`>. A <`double`> is a double-precision floating-point number and can represent a fraction or a number much larger than an <`int`> can. We'll need that for this program.

18–20 We haven't seen `scanf()` yet, and we'll talk about it in detail later. But it's the reverse of `printf()`: `printf()` prints data out, and `scanf()` reads data in. `scanf()`'s first argument, like `printf()`'s first argument, describes the format of input to be read, and the remaining arguments, rather than getting printed out, describe where to store the input values entered by the user. The `"%lf"` says that the input will be in the form of a **l**ong **f**loat, which is an obsolete synonym for a <`double`>. The `&a` is more interesting and important, and so it gets a section to itself.

10.3 Pointers and the & Operator

We need to tell `scanf()` where to store the value that the user enters. Say we want to store that value in the variable `a`. Now, `scanf()` can't normally do that because it doesn't know where `a` is. If we wrote something like

```
scanf("\%lf", a);
```

that would be no good at all, because `scanf()` would get the *value* of `a` and would never find out what it really needs to know, which is where `a` actually is.

The `&` operator, called the *address-of* operator, says where a variable is. The expression `&a` yields a value of a special type, called a <pointer to `double`>, because it 'points to' a <`double`>—that is, it says where a certain <`double`> object can be found. So, in short, we are telling `scanf()` where to find the variable in which we want it to store its result. Later on, we'll see more of `&`, and of its complementary operation, `*`, which takes a pointer and finds to which it points. The main point here is that since we explicitly told `scanf()` where `a` is stored, `scanf()` can change the value of `a`.

The call

```
scanf("\%lf", &a);
```

instructs `scanf()` to read input from the keyboard, parsing and interpreting it as a floating-point value, and to store the value that that data represents into the space pointed to by `&a`.

Note the parentheses in the assignment expressions, to preserve clarity and also to get the compiler to do what we want.

Note that we use `%f` to get `printf()` to print out a <`double`> value, whereas we used `%d` to print out an <`int`>.

11 I Made a Mistake

When I said that the declaration `int foo;` initialized `foo` to be 0, I lied. The compiler is a allowed to initialize `foo` to anything it wants, including 0 or a random garbage value. To initialize `foo` to 0, write `int foo = 0;`.

12 Loops

Frequently we want a program to keep doing the same thing over and over again until something happens. That's called *looping*.

12.1 `while`

For example, we might play a game and then ask the user if he or she wants to play again. We'd want to do that over and over as long as the user kept saying 'yes', until finally the user said 'no'.

We use a `while` statement for that sort of control flow. The form of a `while` statement is:

`while (`*condition*`)` *statement*

To execute this kind of statement, the computer first evaluates the condition, which is just an expression. If the condition is true, it executes the statement. If the condition is false, it skips the whole thing and goes on to the next statement.

So far this is just like `if`. The difference is this: When the computer is done executing the statement in the body of an `if`, it goes on to the next statement. When the computer is done executing the statement in the body of a `while`, it goes back and repeats the whole statement. The computer will test the condition and execute the statement over and over, until finally when the condition is false, it gets to go on to the next statement.

Here's code to print out the numbers between 1 and 50, inclusive:

```
int x = 1;

while (x <= 50)
  printf("%d ", x++);
```

Control is stuck in the `while` loop until the value of `x` exceeds 50. Each time through the `while` loop, we call `printf()` to print the value of `x`, and we increment `x`. When we've incremented `x` so many times that its value exceeds 50, the test in the `while` statement fails and we drop down to the next statement.

Here's another example. This time the body of the `while` statement is a compound statement:

```
int n=0;

while ( n<1 || n>10 ) {
  printf("Please enter an integer between 1 and 10.\n");
  scanf("%d", &n);
}
```

We'll keep prompting the user and reading an integer until we get one between 1 and 10. Note that `n` is initialized to 0; if it were initialized to 7 instead for some reason, we'd never prompt at all, because the condition would be false the first time we got to the `while` statement.

13 More about `scanf()`

`scanf()` is a function, and so it has a return value. It returns a value to its caller to say whether or not it succeeded in doing what the caller asked. It returns an <`int`>.

In the quadratic equation program we did

```
/* get inputs a, b, and c from user. */
printf("Please input a, b, and c, one per line.\n");
scanf("%lf", &a);
scanf("%lf", &b);
scanf("%lf", &c);
```

but there's a somewhat simpler way:

```
/* get inputs a, b, and c from user. */
printf("Please input a, b, and c, separated by white space.\n");
scanf("%lf %lf %lf", &a, &b, &c);
```

`scanf()`'s first argument, the *format string*, can contain more than one conversion specifier (The `%lf`s are *conversion specifiers* because they specify how to convert the input to values), and it can contain white space. The format string is like a pattern that tells `scanf()` what things to look for in the input, in what order, and what to do with them if it finds them.

A `%lf` tells `scanf()` to start looking for input in the form of a decimal numeral, and that there will be an extra argument of type <pointer to `double`>

which will say where to store the value that the numeral represents. White space in the format string tells `scanf()` to expect some white space characters in the input and to read them and throw them away.

The format string we gave to `scanf()` above tells it to read a decimal numeral into the `<double>` variable pointed to by `&a`, then read and discard some white space characters in the input, then to read another decimal numeral into the space pointed to by `&b`, discard more white characters, and finally read a last decimal numeral into the space pointed to by `&c`. When `scanf()` reaches the end of the format string, it stops reading the input, and any characters still unread stay there until the next time a function asks for input.

Now, we already know what happens if the `scanf()` call above gets the input `12 39.07 -.00003`, but what if you gave it `12 foo -.00003` instead? The `12` gets read and put into the space pointed to by `&a`, and then the blank character is all right because it's white space, but as soon as `scanf` sees the character `f`, it knows something's wrong—it's expecting something it can interpret as a `<double>`, and no `<double>` starts with `f`. What does it do?

It leaves the `f` and all the following characters unread, does not store anything into the spaces pointed to by `&b` or `&c`, and returns immediately. It returns the value 1, because it was only able to read, interpret, and store one of the conversions. If we had given `scanf()` the correct input with three numerals in it, it would have been able to read, interpret and store all three of the conversions, and so it would have returned 3. `scanf()` always returns the number of conversions it was able to read, interpret, and store.

13.1 A Heinous Error

Beginners often make this mistake:

```
int num = 0;

while ( num < 1 || n > 50 ) {
  printf("Enter an integer between 1 and 50.\n");
  scanf("%d", &num);
}
```

What's wrong here is that if the user enters a bogus input, such as `foo`, instead of a decimal numeral, the `scanf()` will fail to convert the input, will return 0, will leave the value of `num` unchanged, and *will leave the bogus input unread*. Then the next time through the loop, the bogus input will still be there,

and it must be read before anything else is,[18] so `scanf()` will fail again in the same way. This code loops forever if the user enters a bogus input, printing the prompt over and over again.

Because of situations like this, `scanf()` is rarely used to read input from users—it is much better for reading input from other computers, which are more predictable. But we can use it, if we are careful.

14 Flushing Input

To recover from bogus[19] input to `scanf()` above, we first need to detect the bogusness of the input, and then we need to discard the bogus input.

We already know how to do the former: We check the return code from `scanf()` and see if it's what we expected. This section is about how to do the latter.

14.1 `getchar()`

`getchar()` accepts no arguments. It reads one character from the input and returns that character as its return value. The character we read with`getchar()` is no longer available to be read again; next time we call `getchar()` or `scanf()` or any other input function, reading will commence with the character after the one we just read.

If there is no more input, or if `getchar()` can't read a character for some reason, it returns `EOF`. `EOF` stands for "End Of File".

Whatever `EOF` is `#defined` as, it can't have type <`char`>. Why not? Because if it did, then that <`char`> value might actually appear in the input, and then you wouldn't be able to distinguish between when `getchar()` returned `EOF` because it had happened to read that character in the input and when `getchar()` returned `EOF` because there was an error.

The return type of `getchar()` therefore can't be <`char`>, because `getchar()` needs to be able to return `EOF`, which is not a <`char`>. Accordingly, `getchar()`'s return type is <`int`>, and the characters it returns are converted to <`int`>s when they're returned.

[18] It would be a terrible disaster for the program to receive the input characters in any order other than the order the user typed them in. If you want your program to look at the characters in the input in some other order, you must read them into memory, and then you can look at the memory any way you want.

[19] I am not being cute here. 'Bogus' is an English word meaning 'counterfeit'.

You have to `#include <stdio.h>` to use `getchar()`, because you need the definition of `EOF`, and also because on most systems `getchar()` is actually a macro, and the definition of that macro is in `<stdio.h>`.

14.2 Character Constants

If you write the sequence `'f'`, the compiler generates a value of type `<int>` which represents the value of the character `f`. If `c` is an `<int>` variable in which a character is stored, you can see whether the character stored in `c` is the letter `f` by doing

```
if (c == 'f') { ... }
```

`'f'` is called a *character constant*. Some characters, such as newlines, are hard to type or would confuse the compiler if you actually put them into the source code. The compiler lets you write `'\n'` to represent the newline character; similarly, `'\t'` is a TAB character, `'\\'` is a backslash (\) character, and `' '` is a blank character. Then

```
if (c == '\n') { ... }
```

checks to see if the character stored in `c` is the newline character, and executes the statements in the body of the `if` if it is.

14.3 A Program to Count Words

This program counts the number of characters, words, and lines in its input.

```
#include <stdio.h>

int main(void)
{
  int c = 0, w = 0, l = 0;       /* Count of characters, words, lines */
  int ch;                        /* Current character */

  while ((ch = getchar()) != EOF) {
    c++;
    if ( ch == ' ' || ch == '\t' || ch == '\n') w++;
    if ( ch == '\n' ) l++;
  }
```

```
  printf("Characters: %d.  Words: %d.  Lines: %d.\n", c, w, l);
  return 0;
}
```

There aren't any notes for this program because there's nothing new here.

14.4 Flushing the Input Line

Now we know enough to recover from the heinous error of section 13.1. If `scanf()` doesn't return what we expect, we'll use `getchar()` to discard all the input up to the end of the line, to give `scanf()` a fresh start next time we call it. Here's the code:

```
int num = 0;

while ( num < 1 || num > 50 ) {
  printf("Enter an integer between 1 and 50.\n");

  if (scanf("%d", &num) < 1) {
    /* Gobble up rest of input line */
    while (getchar() != '\n')
      /* do nothing */ ;
  }
}
```

There's only one new feature here: a `;` by itself is a *null statement*: it behaves syntactically like a statement, but it doesn't actually do anything. The gobbling up of characters happens in `getchar()`, which is in the condition part of the `while` loop, so the `while` doesn't have to do anything in its body. But the syntax rules say that `while` must be followed by a statement. So we follow it with a null statement, which doesn't do anything.

15 More Loops

Looping is so important that there are three ways to do it.

15.1 `do...while`

Consider this example:

```
int n = 0;

while ( n<1 || n>10 ) {
  printf("Please enter an integer between 1 and 10.\n");
  scanf("%d", &n);
}
```

Note that `n` is initialized to 0; if it were initialized to 7 instead for some reason, we'd never prompt at all, because the condition would be false the first time we got to the `while` statement. We had to be careful to fix up `n` in advance to make sure that the loop gets executed at least once.

There's another loop construct like `while`, which isn't as often used, but which might have been better for the example above. It looks like this:

`do` *statement* `while (`*condition*`) ;`

The difference is that the condition is tested *after* the statement is executed, instead of before. That means that the statement is always executed at least once. For example:

```
int n = 7;

do {
  printf("Please enter an integer between 1 and 10.\n");
  scanf("%d", &n);
} while ( n<1 || n>10 ) ;
```

Now it doesn't matter that `n` is initialized to 7, because we prompt and ask for a number from the user at least once anyway, and the 7 is obliterated by the call to `scanf()` before we ever look at the value of `n`.

Use `while` when it might be appropriate for the actions in the body to never happen at all; use `do...while` when you always want the actions to be executed at least once.

15.2 `for`

This code exemplifies the notion that every loop has four logical parts:

```
int x = 1;

while (x <= 50) {
  printf("%d ", x);
  x++;
}
```

There's an *initialization* to set up the variable or variables that form the main control of the loop; that's the `x = 1`. There's a *control expression* that says how long to run the loop and when to stop; that's `x <= 50` here. There's a *control update*, which updates the values the main control variables; that's `x++` here. And there's a *loop body*, which contains the statements that the control expression actually controls; that's the `printf()`.

There's a loop construct that makes all four parts explicit. It looks like this:

```
for ( initialization ; control ; update )
     loop body
```

This is functionally identical[20] to this code:

```
initialization
while ( control ) {
     loop body
     update
}
```

For example, here's our example that prints the numbers from 1 to 50 again:

```
int x;

for ( x=1; x<=50; x++ )
    printf("%d ", x);
```

When to use `for` and when to use `while` is a matter of style. Typically, we like to use `for` when the initialization, control, and update expressions are all closely related, because it keeps them together, and `while` in other cases, because it doesn't emphasize a connection that isn't there.

[20] Not quite identical, but the only difference is in the behavior of `continue` statements, which we'll see later.

Sometimes for some reason we don't want to have an initialization in a `for` statement. In that case we just omit it and leave a blank space between the open parenthesis and the first semicolon after the `for`. Similarly, we can omit the update expression. We can even omit the control condition, in which case the computer will take it as being always true. So as a special case,

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

performs the statement delimited by the braces over and over, forever.[21] This is functionally identical with

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

16 Example: A Program to Compute Prime Numbers

A *prime number* is a positive integer, n, which is evenly divisible only by n and by 1. For example, 2, 3, 5, 7, 11, and 13 are prime numbers. 4 is not prime, because it is divisible by 2; 6 and 9 are not prime because they are divisible by 3.

This program reads number inputs from the user and says whether they are prime or not, until the user enters 0, when it exits. It checks a number n to see if it is prime by dividing it by each number j between 2 and $\frac{n}{2}$; if the remainder, computed with the `%` operator, is zero, then n is evenly divisible by j and so is not prime.

```
#include <stdio.h>
#include <math.h>                /* For sqrt() */

int main(void)
{
  int n;
```

[21] There are ways to get out of this statement: You could use `return`, for example. Other ways we haven't seen yet include `goto` and `break`.

```
  for (;;) {
    printf("Enter a number, 0 to quit.  ");
    if (scanf("%d\n", &n) < 1)  /* Bogus input */
      while (getchar() != '\n') /* Discard input characters to end of line */
        /* nothing */ ;
    else {
      if (n == 0)
        return 0;
      else if (is_prime(n))
        printf("That number is prime.\n");
      else
        printf("That number is not prime.\n");
    }
  }
}


int is_prime(int n)
{
  int divisor;


  for (divisor=2; divisor < sqrt(n); divisor+=1)
    if (n%divisor == 0)
      return 0;


  return 1;
}
```

17 A Function to Swap the Values of Two Variables

Consider a program which takes a long list of numbers and arranges them in numerical order. This operation is called *sorting*. By some estimates, 50% of all computer use in the world is devoted to sorting things. Many typical sorting algorithms work by finding two elements in the list that are out of order, and then swapping their positions. In C that would mean we would have a bunch of variables, one for each element in the list, and we would swap the values in the two variables that contained out-of-order elements.

If we were writing such a program, we would do a lot of swapping. We can swap the values of two variables, say `x` and `y`, as follows:

```
temp = x;   x = y;   y = temp;
```

where `temp` is a variable of the same type as that of `x` and `y`. Of course, that'll get ugly and cluttered if we have to write it a lot; it would be better if we could put it into a function and simply write something like

```
swap(x,y);
```

to swap the values of variables `x` and `y`.

17.1 A First Try

Here's our first cut at a program which sets up two variables and the calls a `swap` function to swap their values:

```
#include <stdio.h>

void swap(int n1, int n2);

int main(void)
{
  int x = 5, y = 119;


  printf("The values of x and y before the swap are %d and %d.", x, y);
  swap(x, y);
  printf("The values of x and y after the swap are %d and %d.", x, y);
  return 0;
}

void swap(int n1, int n2)
{
  int temp;

  temp = n1;
  n1 = n2;
  n2 = temp;
}
```

17.2 Why the First Try Doesn't Work

If you key this in and compile it, you'll discover it doesn't work. It compiles okay, but the values of `x` and `y` never get swapped. What's wrong?

The problem is this: `swap` only gets the *values* of its arguments `x` and `y`; it never finds out where `x` and `y` actually are so that it can change them. Put another way, the values `swap` receives are *copies* of the values stored in `x` and `y`. `swap`'s variables `n1` and `n2` are private variables; they get created when `swap` is called and destroyed again when `swap` returns. All the shuffling around of values that `swap` does is in its own private variables which get destroyed when it returns.

This is actually a feature and not a bug—normally, of course, we don't want functions that we call from `main` to be able to mess around with `main`'s variables. It would be very bad, for example, if `printf` surreptitiously changed the computer's secret number in the middle of your guessing game program.

The way out is the same as with `scanf`: instead of passing the values of the variables whose values we want to swap, we'll pass their addresses. Then `swap` will know where `x` and `y` are actually written on the blackboard, and it will be able to change their values all it wants. Note that even if we tell `swap` where `x` and `y` are on the blackboard, it still doesn't know where any of `main`'s other variables are, so it can only change the ones we wanted it to.

This would be a good time to go back and review the material from section 10.3

17.3 This One Works

```
#include <stdio.h>

void swap(int *n1, int *n2);

int main(void)
{
  int x = 5, y = 119;
```

```
  printf("The values of x and y before the swap are %d and %d.", x, y);
  swap(&x, &y);
  printf("The values of x and y after the swap are %d and %d.", x, y);
  return 0;
}

void swap(int *n1, int *n2)
{
  int temp;

  temp = *n1;
  *n1 = *n2;
  *n2 = temp;
}
```

17.4 The & Operator Again

First, note that the call to `swap` in `main` has changed from

```
swap(x, y);
```

to

```
swap(&x, &y);
```

. Now, recall what the `&` operator does: if `x` is some variable, then `&x` describes where that variable can be found on the blackboard. The value of `&x` is said to *point to* `x`; since the type of `x` is <`int`>, the type of `&x` is <pointer to `int`>.

17.5 The * Operator

Now we know how to make a pointer: we use the `&` operator. Once we have a pointer, how to we find out or change what it is pointing to? The complement of the `&` operator is the `*` operator, sometimes called the *dereferencing operator*. If p is a pointer value, then `*`p is the object to which p points. So, for example, the value of `*&x` is the same as the value of `x`, because `*&x` means to get whatever `&x` points to, and `&x` points to `x`.

Let's suppose that `swap` was called from `main` as in the example above, and let's look at the individual statements in `swap` and see what they do. First,

when `swap` gets called, `n1` gets assigned the value of the first argument, which is a pointer to the variable `x`. (That is, `&x`.) `n2` gets assigned the value of the second argument, which is a pointer to the variable `y`. (That is, `&y`.) Then, the line

```
temp = *n1;
```

in `swap` says to find the variable that `n1` points to (`x` in this case), get its value (5), and assign that value to the variable on the left, `temp`.

The line

```
*n1 = *n2;
```

says to find the variable that `n2` points to (`y`), get its value (119), and assign that value to the variable that `n1` points to (`x`).

The line

```
*n2 = temp;
```

says to get the value of `temp` (5), and assign that value to the variable that `n2` points to. (`y` in this case.)

17.6 `swap`'s Header

As usual, in `swap`'s header we have to tell the compiler what types of arguments `swap` will take and what type of return value it will return. We say that `swap` returns type `void`. That means that `swap` really doesn't return any value.

The parameter declaration `int *n1` declares a variable, `n1`, of type <pointer to `int`>. Here is how to remember what this means: If the declaration had said `int n1`, it would mean that `n1` is an <`int`>. But instead it says `int *n1`, so instead it means that `*n1` is an <`int`>. Since `*n1`, the thing `n1` points to, is an <`int`>, `n1` itself must be a <pointer to `int`>.

17.7 A Note About Functions that Return `void`

If a function's return type is `void`, that means it doesn't return a value at all. To return from such a function, we just write

```
return ;
```

, omitting the expression that usually follows the word `return`.

The other way to return from a `void` function is to just let control flow off the bottom of the function; this is the same as doing `return ;`.

If you do `return ;` in a non-`void` function, or let control flow off the bottom of a non-`void` function, the function may return a random garbage value to its caller, or your program may fail completely.

17.8 More About Prototypes

Back in section 10.2, there was a brief note about *prototypes*. You should go and reread that now if you've forgotten it.

Here's the problem we must solve: At the time the compiler compiles the line `swap(&x, &y);` , it hasn't seen the definition of `swap` yet.[22] But the compiler needs to write the machine instructions for the call to and return from `swap`, and to write those correctly it needs to know the types and number of the arguments and the type of the return value.[23] We need a way to give the compiler this information even in the absence, temporary or otherwise, of the `swap` function itself.

The way we do that is with a *prototype*. To write a prototype for a certain function, we write an ordinary function header for that function, but we follow it with a semicolon instead of a function body. The function header contains all the information the compiler needs to translate the call and return properly.

That's what the line `void swap(int *n1, int *n2);` is doing at the top of the program of section 17.3. It provides argument and return value type information for `swap` to the compiler in advance of the actual definition of `swap`.

If the compiler hasn't seen a prototype for a certain function at the time it compiles a call to that function, it guesses and does the best that it can. It assumes that the function's return value is <`int`>, which can lead to disaster if it isn't,[24] and it does the best it can to handle the arguments, which sometimes works out and sometimes doesn't.

[22] In fact, it's quite possible that `swap`'s source code resided in a different file entirely, which was compiled eight months ago and then thrown away. I am not being facetious.

[23] I'm afraid you won't be able to appreciate why this is until you've studied the inner workings of the compiler in detail. We won't do that in this course.

[24] Try writing a program that uses `sqrt` without including `<math.h>` and you'll see what kind of horrors can occur—you get completely bogus answers back from `sqrt` because the compiler thinks that `sqrt` is returning an <`int`> when it's really returning a <`double`>.

You should have a prototype in your program for every function except `main`. (`main` always has the same return type and argument types anyway.)

Library functions like `printf` and `sqrt` must have prototypes too, so that the compiler can compile the calls to them correctly,[25] but the prototypes almost always appear in some header file, and so you include the prototype into your program by including the appropriate header file. For example, if you hunt up the `math.h` header file and look in it, you'll see, along with a lot of other nonsense, something like

```
double sqrt(double arg);
```

.

The actual definition of a function, the one that supplies the instructions about how to execute the function, includes a header for the function, and so it counts as a prototype—after the compiler has seen an entire function, it certainly has enough information to compile calls to that function. If we moved the `swap` function so that it appeared before `main` in the file, we could omit the prototype, because by the time the compiler had to compile the call to `swap`, it would have seen the entire definition of the `swap` function and would have known all about it.

18 `break`

The `break` statement interrupts a loop prematurely. It's easy to use: You just write `break;`, and if the computer reaches the `break` statement, control immediately passes the the statement following the end of the smallest enclosing `while`, `do...while`, `for`, or `switch` statement.[26]

18.1 Examples of `break`

In each of these ghostly examples, the ⊗ symbol shows the place in the program to which the computer skips if it happens to execute the indicated `break` statement.

[25] In this case the source code for the called function *really* isn't available—it's locked in a vault somewhere at Borland.

[26] We haven't seen `switch` yet, but it's coming up soon.

```
while ( . . . ) {
    . . .
    if ( . . . )
      break ;
    . . .
}
⊗
```

```
while ( . . . ) {
    for ( . . . ) {
        . . .
        if ( . . . )
            break ;
        . . .
    }
    ⊗
    . . .
}
```

```
do {
    . . .
    if ( . . . )
        if ( . . . )
            break ;
    . . .
} while ( . . . ) ;
⊗
```

Note that `break` does *not* care about `if` (or `else`) when it breaks; if it did, it would be useless. (Why?)

If you write a `break` statement that isn't enclosed by a `while`, `do...while`, `for`, or `switch` statement, the compiler will grouse and refuse to compile your program.

18.2 `break` Statement Considered Harmful?

Many early programming languages had only two control structures: They had an `if–else`, and they had the infamous `goto` statement, which unconditionally transferred control to a certain other statement.[27]

[27] This is an oversimplification. Many early programming languages, notably LISP, had an utterly different way of managing control flow in the first place, and this whole debate is moot

Around 1968, imperative languages such as ALGOL (a distant ancestor of C), were just beginning to have what are known as *logical control structures*, which let you express your algorithms in terms of blocks of code which were executed when certain conditions held, rather than in terms of a flow of control which jumped around the program from numbered statement to numbered statement in response to certain conditions.

In 1968 a gentleman named Edsger Dijkstra wrote a note in *Communications of the ACM*, a well-known computer science journal, called *Goto Statement Considered Harmful*. He had discovered that when programmers in his organization were forbidden from using `goto`, and required to use only logical control structures, the programs they wrote had fewer errors and the errors the programs did have were easier to fix.

This is a reasonable thing to notice, and, because the logical control structure proponents were mostly right, most modern imperative languages, including C, stress logical control structures such as `while`, and have `goto` only as an afterthought, if at all.

Logical control structures are a little closer to the way we think than `goto` is. Rather than giving someone a laundry list of instructions like "14. If you're not at the store, `goto` step 11." we're more likely to say, "Keep walking north `until` you get to the store." `goto` is considered bad form in most cases, and although it occasionally has its uses, it really is better to avoid it whenever possible.

That warning extends to `break` (and to `continue`, although we haven't seen that yet): `break` interrupts the logical flow of control and causes an unconditional jump to somewhere else, and so it can be confusing; overuse of `break` can obscure what your program is really doing. When you see a `while` loop, you normally know what's going on; you can say, "Oh, this part of the program tries to do such-and-so `while` there is data left in the file." But if there's a `break` in the loop, you have to add on a qualifier: "...unless we hit the `break`."

I'm tolerant of `break`. Sometimes it's much easier to express something with a `break` than it would be without, and the code is shorter or clearer with the `break` than without. Nevertheless, there are a lot of logical-control fanatics in the world, and they'll tell you never to use `break`, `continue`, or `goto`, and to `return` only from the bottom of a function, and never from the middle.

In the CSE110 handbook, it says never to use `break` or `continue` in this class, and that you'll lose at least one point on any assignment you turn in that has a `break` or a `continue`. That's the logical-control fanatics talking. Since I

for them. On the other hand, FORTRAN (which is much more like C than LISP is) had a whole mob of conditional and computed test statements, all of which ended by transferring control to the statement with a particular line number. The point stands anyway.

am not a logical-control fanatic, that rule goes out the window. I am, however, a simplicity and clarity fanatic, because I spend a substantial part of my life reading other people's C code. So the rule I'd rather have you follow is this: When in doubt, write the code both ways, and pick the one that seems clearest and simplest.

19 More about Object Contexts and Value Contexts

If you look back at the notes for section 5.2, you'll recall the following facts:

- The word *object* is short for the phrase "a particular part of the computer's memory."
- Some expressions, such as `x`, refer to objects. (For example, `x` refers to the variable `x`, which is a particular part of the computer's memory and is therefore an object.) Most expressions, such as `4` or `3*x + y`, do not refer to objects.
- An expression which refers to an object is called an *lvalue expression*.[28]
- A *context* is a place in your program where you are allowed to put an expression.
- When the computer encounters a certain expression in one context, it may behave differently than when it encounters the same expression in another context.
- There are only two kinds of contexts in C: *object contexts* and *value contexts*. What the computer does with an expression depends only on the expression itself and on whether that expression is in an object context or a value context.
- Most contexts are value contexts.
- The only contexts that are not value contexts are:

  1. The left-hand-side of an assignment expression, and
  2. The operand of the `&` operator.

  These two contexts are object contexts, not value contexts.

[28] The 'l' in 'lvalue' stands for 'left', because the thing on the **left** side of an assignment must be an lvalue expression.

- Any expression may appear in a value context.
- An expression in a value context is evaluated to produce a value.
- Only an lvalue expression may appear in an object context.
- An lvalue expression in an object context is not evaluated to produce a value. Instead, the computer figures out what object it refers to, and then
  1. If the lvalue expression was on the left-hand-side of an assignment, the computer uses the object it represents as a place to store the value it computed on the right-hand-side of the assignment, or
  2. If the lvalue expression was the operand of the `&` operator, then the computer gets the address of the object that the lvalue expression represents.

19.1 A Pointer Value Points to an Object

If p is a valid expression whose type is <pointer to something>, then p points to an object of type <something>. Then $*p$ is an expression which represents the object that p is pointing to. Since $*p$ therefore refers to an object, $*p$ is an lvalue expression, and may appear on the left-hand side of an assignment statement, as in

```
*n1 = temp ;
```

. `temp` is in a value context, so it's evaluated, and the computer gets the value stored in the variable `temp`. On the other hand, `*n1` is in an object context, and so once the computer finds out what part of the blackboard `*n1` represents, it does *not* go get the value stored there. Rather, it uses it as a place to store the value of `temp`. On the other hand, in

```
temp = *n1 ;
```

the `*n1` is in a value context; once the computer figures out what region of the blackboard `*n1` represents, it goes and gets the value stored there, and then it stores that value into the region of the blackboard represented by `temp`. `temp` is in an object context, not a value context, so it is not evaluated and the computer does never retrieves the value stored in `temp`.

19.2 The operand of & is in an Object Context

We have seen exactly two examples of object contexts so far. One is the left-hand side of an assignment.

The other is that in the expression `&`*foo*, *foo* is in an object context: It is not evaluated to produce a value, and the value of the entire `&`*foo* expression has nothing whatever to do with the value actually stored in *foo* itself. Also, *foo* must be an lvalue expression, because `&` asks where in memory its operand is stored, so therefore its operand must be an object, and only lvalue expression, by definition, represent objects.

19.3 So What?

This is going to become frightfully important in section 25.3. In the meantime, we'll do something else.

20 Arrays

Suppose we want to write a program which reads in a list of 100 numbers from the user, and then prints them out in reverse order. To do that, we must remember all 100 numbers until the end; there's no way around it. We could write a program with a very long declaration:

```
int main(void)
{
  int n0, n1, n2, n3, n4, n5, n6, n7, n8, n9;
  . . .
  int n90, n91, n92, n93, n94, n95, n96, n97, n98, n99;

  . . .
}
```

and then use 100 `scanf`'s to read them in and 100 `printf`'s to write them out backwards, but of course there has to be a better way.

20.1 A Better Way

If we write

```
int n[100];
```

instead, the compiler creates an *array* of 100 <int>s for us. That means that it finds enough space for 100 <int> variables, reserves the space, and arranges that `n` refers to that space. The space is all contiguous, meaning that if an <int> is 2 bytes long, the compiler will find 200 bytes of space all in the same place—it won't find us 37 bytes here and 22 bytes there and 41 bytes somewhere else.

We can ask for these 100 <int>s individually: The expression `n[0]` refers to the first one, and `n[1]` to the second one, and soforth, up to the last one which is `n[99]`. These 100 <int> variables are called the *elements* of the array; each element has an *index* which says where it is in the array. The index of the first element is 0, the index of the second is 1, and the index of the last one is 99. So here's our program:

20.2 Program to Read 100 Integers from the User and Write them out in Reverse Order

```
#include <stdio.h>

int main(void)
{
  int n[100], i;

  for (i=0; i<100; i++)
    scanf("%d", &(n[i]));

  for (i=99; i>=0; i--)
    printf("%d\n", n[i]);

  return 0;
}
```

20.3 Notes on the Program

In the first loop, `i` starts at 0 and goes up by ones until it gets to 99, the index number of the last element in the array `n`. Each time through the loop, the computer figures out what `n[i]` is (it's the `i`'th element of the array `n`), but,

because the `n[i]` is in an object context, not a value context (it's the operand of the `&` operator), the computer does not go and get the value of `n[i]`. Instead, it computes the address of `n[i]`, and passes that address to `scanf`, which reads an <int> value from the user and stores the value in the variable `n[i]`. `scanf` could change the value of `n[i]` because it knew where on the blackboard `n[i]` was. `scanf` knew where on the blackboard `n[i]` was because we passed it the address of `n[i]`.

The first number the user enters gets stored in `n[0]`, and the second gets stored in `n[1]`, and soforth, until finally the user enters the hundredth number, which is stored in `n[99]`.

Then the computer executes the second loop. `i` starts at 99, the index number of the last element of the array `n`, and counts backwards until it gets to 0. After 0, the `for` loop is done. Each time through the loop, the computer figures out what `n[i]` is (it's the `i`'th element of the array `n`), and then, because the `n[i]` is in a value context, the computer gets the value of `n[i]` and passes that value to `printf` as an argument. `printf` then prints out the value of `n[i]`.

Since `i` starts at 99, the first value the computer prints out is that of `n[99]`, which was the last number the user entered. Then the computer decrements `i`, runs through the loop again, and prints out the value of `n[98]`, whose value is the next-to-last number the user entered. The computer keeps running backwards through the array, until finally `i` is 0; then the computer prints out the value of `n[0]`, which is the first number the user entered. Then the computer decrements `i` to -1, exits the loop, and quits the program.

20.4 Initializing Arrays

When you declare an ordinary variable, you can initialize it thus:

```
int n = 57;
```

This means that `n` starts out with the value 57; the 57 here is called an *initializer*. We can do a similar thing with an array, only we have to specify more than one initializer: If we write

```
int n[5] = { 1, 3, 9, 27, 81} ;
```

then `n[0]` starts out with the value 1, `n[1]` starts out with the value 3, `n[2]` starts out with the value 9, `n[3]` starts out with the value 27, and `n[4]` starts out with the value 81.

If you make the initializer list too short for the array, as in

```
int n[5] = { 1, 3 };
```

the computer initializes the elements of the array until it runs out of initializers, and initializes the rest of the elements of the array with zero values. So the example above is the same as if we had written `int n[5] = { 1, 3, 0, 0, 0};`.

If you make the initializer list too long, the compiler will complain.

If you include an initializer, you can omit the array size:

```
int n[] = { 1, 3, 5};
```

The compiler counts the number of initializers and makes an array that is just big enough to hold all of them. In this case it makes an array with 3 elements because you specified 3 initializers.

20.5 Character Arrays

Of course you can have an array of variables of any type, including <char>s. Here's a program to print someone's name out backwards:

```
#include <stdio.h>

int main(void)
{
  char name[] = { 'J', 'e', 'a', 'n', ' ', '0', 'g', 'r', 'i', 'n', 'z' };
  int i;

  for (i=10; i>=0; i--)
    printf("%c", name[i]);

  printf("\n");
  return 0;
}
```

Note that we use `%c` to print out a <char> with `printf`. Also note we had to hard-wire the length of the character array into the loop. We'll learn how to get around that tomorrow.

It's a pain having to type all those character constants, and C gives us a shortcut: We're allowed to initialize a character array this way: Instead of writing

```
char name[] = { 'J', 'e', 'a', 'n', ' ', 'O', 'g', 'r', 'i', 'n', 'z' };
```

we can do this:

```
char name[] = "Jean Ogrinz";
```

. The shorthand `"Jean Ogrinz"` denotes a special array of characters, called a *string of characters* or usually just a *string*. Strings are how we handle things like peoples' names in C. We'll see a lot of them in the rest of the course.

In case you hadn't made the connection yet, the first argument to `printf` is always a string.

21 Miscellaneous Details About Arrays

Here are some fine points of array use that we didn't talk about before.

21.1 Out-of-Bounds Array References

Question: `arr` has only 23 elements. In this code we're trying to store the value 5 into the 120th element. What happens?

```
. . .
int arr[23];

arr[119] = 5;
```

Answer: You get undefined behavior, so the compiler might choose to teleport elephants into the room, or to generate an executable program, which, when run, will teleport elephants into the room. Most likely, however, is that the compiler will say nothing at all, and will generate an executable program that will assign the value 5 to the place in memory where `arr[119]` *would* have been if `arr` had had that many elements. However, `arr` doesn't extend that far,

and chances are that the compiler put some other variable at that spot instead, so that the `5` will obliterate some other piece of data.

You might ask why the compiler doesn't catch this for you, and the answer is that catching out-of-bounds array references at the time the program is compiled is impossible.[29]

You can catch out-of-bounds array references at the time the program is run, so that when you run the program, it aborts in the middle rather than doing a bogus array reference. Some languages, such as Pascal, do this automatically. But to detect the bogus array reference the computer must test the index value, every time it looks at any element of any array, to make sure it is not too big or too small, and that will make your programs slow. The C compiler takes the point of view that if you wanted to check to index value, and accept the corresponding slowness, then you'd write code yourself that tests the index value, and that if you didn't you wouldn't, and so washes its hands of the problem.

An out-of-bounds array reference can create a bug that is very difficult to find. If you store a value into an array element that isn't there, you'll clobber some other variable's value, and you might not find out about that until much later when you try to use the variable and discover it is full of garbage. So be careful.

21.2 The Expression `a[i]` is an Lvalue

The expression `a[i]` represents an object, namely the `i`'th variable in the array `a`, and is therefore an lvalue, and may appear on the left-hand-side of an assignment statement, or as the operand of the `&` operator. We've already seen array references in both contexts.

21.3 `[···]` has High Precedence

The `[···]` operator has higher precedence than *everything* else, so that `&a[i]` means `&(a[i])` and not `(&a)[i]`.

[29] One can prove mathematically that there is *no* algorithm which, given the source code for a C program, can determine whether or not that C program will generate an out-of-bounds array reference when it is run.

21.4 More About Strings

Suppose we have an array, and we want to perform some operation on every element of the array. We'd write a loop, of course; and loop over the elements in the array, performing the operation on the first one, and then on the second one, and soforth.

The question: How do we know when to stop?

Either, we have to remember how long the array was, or, we have to arrange that the last element of the array contains a special *sentinel* value that we can look for so that when we see the sentinel, we know we're done. Both methods are widely used. For strings, however, we always use a the sentinel value method, and the C language provides a little support for this.

When you write something like

```
char name[] = "Jean Ogrinz";
```

the compiler creates an array of *twelve* characters, not eleven. It initializes the first element of the array with the letter `J`, the second with `e`, and soforth, and it initializes the last element with a special sentinel value: the *NUL character*.

The C library provides many functions that operate on strings, and every one of them assumes that the strings you pass to it as arguments will end with the NUL character; that's how these functions know where to stop. For example, there's a function `strlen` which takes a string as an argument and returns the length of the string, and it does this by counting the characters in the string one at a time until it sees the NUL character.

So when I said a string was an array of characters, I fibbed a little. A string is an array of characters that is terminated with a NUL character.

Every character is represented internally by some integer between 0 and 255; for example, on typical machines, the character `A` is represented with the number 65. The details about what number represents what character are None of Your Business, with one exception: The NUL character is always, always, always the character which is represented by the number 0.

If you need to write the NUL character in your C program, you can write `'\0'`.

22 Declarations

The C declaration syntax was designed to be intuitive, but isn't.

The declaration

```
int i, *pi, ai[5], ii;
```

declares four things: `i`, an <`int`>; `pi`, a <pointer to `int`>; `ai`, an <array of 5 `int`s>; and `ii`, another <`int`>.

The way to remember this is that if you were to take any one of the expressions from the declaration and actually write it in your program, you'd have an <`int`>. So:

- `i` is an <`int`>.
- `*pi` is an <`int`>, and therefore `pi` itself is a <pointer to `int`>.
- `ai[5]` is an <`int`> (or would be if it weren't out-of-bounds; in any case, `ai`[something] is an <`int`>.), and therefore `ai` itself is an <array of `int`>)
- `ii` is an <`int`>.

This interpretation may come in handy when you're trying to understand a more complicated declaration, such as

```
char *argv[];
```

which says that `*argv[]` is a <`char`>, so therefore `argv`[something] must be a <pointer to `char`>, and `argv` itself must be an <array of pointer to `char`>.

C's declaration syntax has been widely criticized already, so let's try to live with it. We won't be seeing declarations much more complicated than this in any case.

23 Pointer Arithmetic

Having just had a section on C's declaration syntax, surely one of its worst features, we should try to recoup a little prestige and have a section on one of C's very best features. Here it is; it gets big type because it is so important:

23.1 The Pointer Arithmetic Rule

If *p* is a pointer which points to a certain element of an array, then the value of the expression `p + 1` is, by definition, a pointer which points to the *next* element of the array.

23.2 An Example

```
float arr[53], *p *q;

p = &arr[4];
```

`p` is now pointing to the fifth element of the array `arr`. (Remember that `arr[4]` is the fifth element, because `arr[0]` is first and `arr[1]` is second.)

```
q = p + 1;
```

`q` is now pointing to the sixth element of the array `arr`. (Remember that `arr[5]` is the sixth element.) Then

```
*q = 119;
 printf("%d\n", arr[5]);
```

will print `119`, regardless of what `arr[5]` was holding before.

23.3 Consequences of the Pointer Arithmetic Rule

Similarly, if *p* is a pointer which points to a certain element of an array, then the value of the expression `p + 2` is, by definition, a pointer which points to the element after the next one in the array.

Similarly, if *p* is a pointer which points to a certain element of an array, then the value of the expression `p - 1` is, by definition, a pointer which points to the previous element in the array.

Similarly, if `p` points to the element `a[0]` of the array `a`, and if `n` is an <`int`>, then the value of the expression `p + n` is a pointer to the element `a[n]` of the same array `a`. Note what happens when the value of `n` is 0: `p + 0` points to the same place as `p`, as it should.

Similarly, if `p` points to an element of an array, and you do `p++;`, then afterwards, `p` points to the next element of the array.

This simple convention is at least partly responsible for C's vast popularity. C makes pointer manipulations easy and flexible by using this simple notation for talking about pointers to different parts of the same array.

We'll see how to use this in the next few sections and in section 25.3, everything will finally fall into place.

24 Examples of Pointer Arithmetic

In each of the following examples, suppose that `array` is an <array of 3 `int`s>, whose elements contain the values 5, 23, and 119, respectively. Suppose `p` and `q` are pointer variables of type <pointer to `int`>, which point initially to the first element of the array `array`, and that `d` is an <`int`> variable. Then:

- This statement

  ```
  q = p + 1;
  ```

  computes the value of `p + 1`, which is a pointer to `array`'s second element (by the pointer arithmetic rule), and assigns that pointer value to the variable `q`. `q` now points to `array`'s second element. `p` has not changed.

- The statement

  ```
  q = p++;
  ```

  gets the value of `p`, assigns that value to `q`, and makes a note to bump up the value of `p` by one before the statement is over. So `q` ends up pointing to the first element of `array`, and `p`, which got bumped up, points to the the second element of `array`.

- This statement is illegal:

  ```
  q = *(p + 1);
  ```

  because `p` is a <pointer to `int`>, and so `p + 1` is also a <pointer to `int`>—it points to the element after the one that `p` points to. Thus `*(p + 1)` is the thing that `p + 1` points to, and is therefore an <`int`>. `q`, on the other hand, is a pointer variable. You can't store an <`int`> in a pointer variable, because pointers are not <`int`>s. The compiler will refuse to compile this statement.

- Perhaps what we meant to write in the last example was this:

```
*q = *(p + 1);
```

  This gets the value of the element after the one `p` is pointing to, and assigns that value to the element that `q` is pointing to. `q` is pointing to the first element of `array`, so that first element gets assigned the value 23. Neither `p` nor `q` changes; both are still pointing to the first element of `array`. But `array` itself now contains the values {`23, 23, 119`}.

- Or, perhaps, what we meant to say in the incorrect example was this:

```
d = *(p + 1);
```

  `d` is an <`int`>, `*(p + 1)` is an <`int`>, so things work out. `p + 1` points to the second element of `array`, and `*(p + 1)` is this second element itself. Since the expression `*(p + 1)` is in a value context, it's evaluated to produce a value, and the result is the contents of the second element of `array`, which is 23. This value is assigned to the <`int`> variable `d`. The value of `p` does not change.

- On the other hand, this is something else again:

```
d = *p + 1;
```

  `p` points to the first element of `array`, so `*p` gets the value of that element, 5. Then the computer adds 5 and 1 and assigns the result, 6, to the variable `d`. Neither `p` nor `*p` change.

- Unary operators like `++` and `*` take precedence from right to left. That means that if you see `*p++`, it means `*(p++)` and not `(*p)++`. What's the difference between these two expressions?

```
d = *(p++);
```

  says to get the value of `p`, which is a pointer to the first element of `array`, and to make a note to bump up `p` by the end of the statement. Then, the computer finds the thing that `p` is pointing to, which has value 5, and assigns that value to `d`. After the statement, `d` got the value 5 and `p` was bumped up to point to the second element of `array`. None of the values in `array` changed. On the other hand,

```
d = (*p)++;
```

says to get the thing that `p` points to, namely the first element of `array`, get its value, and make a note to bump that value up by 1 before the end of the statement. The value of the first element of `array` gets assigned to `d`, and then the value gets bumped up by 1. So `d` gets the value 5, and `p` hasn't changed—it still points to the first element of `array`, whose value is now 6 instead of 5.

In short: `p++` means to bump up `p` so that it points to the next element in the array. `(*p)++` means to bump up the value of the thing `p` points to, but to leave `p` itself unchanged so that it points to the same place.

25 Too Much Work

C has a philosophy that it won't perform any operation unless that operation is completely trivial. For example, there is no way to say "Add the value 3 to every element of the array `a`." You can do it, but you have to code the loop yourself. That's because there is usually more than one way to perform any nontrivial operation, and C does not want to be in the position of having to figure out what way is best for your particular application. C lets the programmer decide what method is best.[30]

Accordingly, there is no operation in C that operates on all the elements of an array at once. You always have to use a loop. Operating on all the elements of an array at once is "Too Much Work." You can get C to do a little bit of work at a time, working on one array element, and then the next, and then the next, but that is the way you have to say it.

In fact, C goes even farther than that. Even manipulating an entire array at once is "Too Much Work". In C, there are no array values.

25.1 No Array Values

This is a little surprising. If `i` is an <`int`> variable, `i`'s value is an <`int`> value. If `pc` is a <pointer to `char`> variable, `pc`'s value is a <pointer to `char`> value.

[30]This philosophy is why there is no automatic run-time array bounds checking in C: If it were there and your program needed to run quickly, you would be out of luck, because there would be no way to take the bounds checking out. On the other hand, if you really did want it there, and you were willing to pay the speed penalty, you could write the code for the array bounds checking yourself. There is a trade-off between speed and safety, and C does not presume to know which one you value more highly.

But: If `af` is an <array of `float`> variable, `af`'s value is *not* an <array of `float`> value, because there are no array values.

So suppose `af` is an <array of `float`>s; what is the value of the expression `a`?

25.2 The Array Value Rule

No expression has a value which is an array. If `a` is an <array of thing>, then the value of `a` is a pointer to the first element of the array `a`, and has type <pointer to thing>.

25.3 Implications for Pointer Arithmetic

The array value rule says that the value of an array `a` is a pointer to `a`'s first element. That means that in a value context, where `a` is being evaluated to produce a value, the expressions `a` and `&a[0]` are interchangeable. The first denotes the array itself, and the second is a pointer (`&`) to `a`'s first element (`a[0]`), but the values of the two expressions are the same. The two expressions therefore behave differently from one another only in an object context.

Now, the pointer arithmetic rule tells us that if `p` points to the first element of an array, say to `a[0]`, then the expression `p + 1`, by definition, is a pointer which points to the second element of that array, namely `a[1]`, and that `p + n`, in general, points to `a[n]`.

The expression `a` is just such a pointer—the array value rule says that it points to the first element of the array `a`. Therefore, `a + n` points to the element `a[n]`.

Since `a + n` is a pointer to the element `a[n]`, the expression `*(a + n)` denotes the element `a[n]` itself. The expression `*(a + n)` is an lvalue expression, because it represents an object: the element of array `a` with index `n`.

Here's the punch line: The expression `a[n]` is in all ways completely identical with the expression `*(a + n)`. When the compiler needs to compile an array reference such as `a[n]`, it compiles it as though you had written `*(a + n)`. Array subscripting is only a notation convenience, and everything we want to do with array can be done with pointers instead, because of C's powerful pointer arithmetic convention.

26 Functions that Operate on Strings

We'll apply our powerful theory of arrays and pointers, and write some functions that operate on strings. First we'll write `strlen`, a function which takes a string as an argument and returns the number of characters in the string.

We'll be calling `strlen` this way:

```
char a[] = "Jean Ogrinz";
int length;

length = strlen(a);
```

Now, if we want to write `strlen`, we need to decide what its header looks like. Clearly it'll return an <`int`>. What's its argument? A string, which is an <array of `char`>, but that's not quite right. The argument to a function is in a value context, and so the `a` in `strlen(a)` is in a value context and is being evaluated to produce a value. The array value rule says that the value of an array in a value context is a pointer to the array's first element. That means that when `strlen` is really being passed is a pointer to `a`'s first element, which is a <pointer to `char`>. So `strlen` will begin like this:

```
int strlen(char *s)
```

here's the rest of `strlen`:

```
int strlen(char *s)
{
  int i = 0;

  while (*s != '\0') {
    i++;
    s++;      /* Make s point to the next character */
  }

  return i;
}
```

26.1 How `strlen` Works

When `strlen` starts up, `s` is initialized to point to the first character of the string we want to examine, as we've said. Since `s` is a <pointer to `char`>, `*s` is the <`char`>that `s` is pointing to.

We run through the `while` loop as long as `*s` is not the NUL character. Each time through, we increment `i`, which is a counter of how many characters we've seen so far, and we increment `s`, which makes it point to the next character in the string. When that character is the NUL character, we quit the loop and return the count.

27 The `NULL` Pointer

For each pointer type, there is one special value of that type which represents a pointer which does not point anywhere at all. This value is called the `NULL`*pointer*. The `NULL` pointer has one and only one interesting property: If x is an object of type <foo>, and therefore `&`x is pointer to the object x, with type <pointer to foo>, then the value of `&`x is *not* the `NULL` pointer.

The way you represent the `NULL` pointer in your code is by writing `NULL`. `NULL` is actually a macro, but we won't discuss what it's a macro for until later, because it's confusing.

Caution: To not confuse the `NULL` pointer with the `NUL` character. The `NULL` pointer is a pointer, which happens not to point anywhere. The `NUL` character is a character, which is conventionally used to mark the end of a string. It's unfortunate that the names are so similar, but the two things have nothing at all to do with one another.

Many functions which return pointer values return the `NULL` pointer to indicate that something went wrong. For example, you might have a function, `malloc`, which takes an argument `n`, somehow finds `n` contiguous bytes of space on the backboard, reserves them, and returns a pointer to the first byte. `malloc` could return the `NULL` pointer to indicate that there weren't `n` free bytes of space left on the blackboard. You could check the return value from `malloc` to see if you had run out of space:

```
char *buf;
buf = malloc(1000);
if (buf == NULL) {
  printf("Out of memory.\n");
  abort();
}
. . .
```

You know that if `malloc` succeeded in finding the memory it was looking for and returned a pointer to that memory, the value of that pointer would be different from `NULL`. If the return value compares equal to `NULL`, we know that `malloc` didn't succeed.

28 Input Sources and Output Sinks

When you call an input function such as `scanf` or `getchar`, the input comes from a place called the *standard input*. Normally, the standard input it attached to the keyboard so that `scanf` and `getchar` read from the keyboard.

If you run your program under MS-DOS, you can arrange to have the standard input connected somewhere else, such as to a file. For example, to run the command 'foo' with the standard input connected to a file, you enter the command line `foo < input_file`. When `foo` calls `scanf` or `getchar`, the data that `scanf` or `getchar` reads comes from the file `input_file` instead of from the keyboard. This is awfully handy—it means your program doesn't have to know anything about files in order to operate on files.

Similarly, `printf` sends its output to the *standard output*, which is normally attached to the screen so that `printf`'s output appears on the screen. But you can redirect the standard output to a file also: `foo > output_file` attaches `foo`'s standard output to the file `output_file`, and everything that `printf` writes will go into the file instead of to the screen.

Your program is completely unaware that anything different is happening when it gets run with its standard input or output redirected, and that's good, because it means you don't have to write any extra code to handle it.

28.1 A File Copy Utility

This trivial program copies data from the standard input to the standard output: (`putchar`'s argument is a character, which it writes out on the standard output; `putchar(c)` is identical to `printf("%c", c)`.)

```
#include <stdio.h>

int main(void)
{
  int c;

  while ((c = getchar()) != EOF)
    putchar(c);
  return 0;
}
```

Let's call it `scopy`, for 'simple copy'. If you run `scopy`, it echoes everything you type back at you, and perhaps that doesn't seem too useful. But you can use it to copy files: `scopy < source_file > destination_file` copies the data in `source_file` into `destination_file`. It's also a program that types the contents of a file on the screen: `scopy < file` reads from `file` but writes to the screen.

29 Operating on a Particular File

Sometimes, though, you want to read data from or write data to a particular file. For example, the compiler needs to write its output into a file with a particular name. How do we do that?

29.1 Opening a File

First you have to ask the operating system to *open* the file for you. Opening a file means that you notify the operating system that you want to use the file. The operating system checks to make sure that the file you've named exists, and that you have permission from the file's owner to read or write the file. It sets up variables in its own blackboard space that keep track of how much data you've read from the file and what part of the file the next byte is supposed to come from. Sometimes it does other things too—on UNIX the operating system arranges that if someone erases a file that you're reading, the file doesn't actually go away until you're done.

The way you open a file is with the `fopen` function. `fopen` accepts two arguments: The first a string containing the name of the file you want to open, and the second is a string that says whether you want to read the file or write it. For example:

```
fopen("cse110.log", "r");
```

says to open the file `cse110.log` for reading. (`"r"` means 'read'; if it were `"w"` it would open the file for writing.)

`fopen`'s return value is a peculiar type: It's a pointer to an object called a `FILE`, which contains some data from the file, information about whether you've reached `EOF` in the file, and other things that the standard I/O functions need to know to work properly. You need this `FILE *` value later on to tell the computer what file you want to read or write from. In some sense, the `FILE *` 'represents' the file.

`FILE *` values are often called *streams*.

The prototype for `fopen` looks like this:

```
FILE * fopen(char *filename, char *type);
```

If `fopen` can't open the file for some reason, it returns the `NULL` pointer.

29.2 Reading from a File with `getc`

Once you've got the file open, you can operate on it with functions that are very much like the ones you're used to. For example, `getc` is a function which takes one argument, a `FILE *`. It is exactly like `getchar` except that it reads its character from the source represented by the `FILE *` that is its argument, instead of from the standard input. Here's code to print out the contents of the file `mydata.txt` onto the standard output:

```
#include <stdio.h>
#define FILENAME "mydata.txt"

int main(void)
{
  FILE * the_file;
  int c;

  the_file = fopen(FILENAME, "r");
  if (the_file == NULL) {
    printf("Couldn't open the file %s.\n", FILENAME);
    return 1;
  }

  while ((c = getc(the_file)) != EOF)
    putchar(c);

  return 0;
}
```

Actually we left out a detail here: `getc` and `getchar` return `EOF` for two reasons: because there was no more data for them to read, or because there was some kind of error in reading the data. (For example, the disk failed in the middle.) We really should have checked whether the `getc` above was returning `EOF` for end-of-file or for an error. We'll see how to fix this soon.

29.3 Stream Versions of `printf` and `scanf`

The function `fprintf` is just like `printf`, except it has an extra argument: The first argument to `fprintf` is a stream to write its output to. The second argument is a format string just like `printf`'s first argument, and the remaining arguments are values to fill into the conversions in the format string, just like `printf`'s remaining arguments. So, for example,

```
fprintf(the_file, "The value of %s is %d.\n", "fooey", fooey);
```

is exactly like

```
printf("The value of %s is %d.\n", "fooey", fooey);
```

except that the `fprintf` writes the output to the file represented by `the_file`, while the `printf` writes it to the standard output.

Similarly, there is an `fscanf` function, which is just like `scanf`. Similarly, `getchar` has `getc` and `putchar` has `putc`.

29.4 Closing a File

When you're done with a file, you must *close* it. This tells the operating system that you are done using the file. That way the operating system knows that it can forget all the details about the file that it was keeping track of for you, such as how much data you'd read out of it.

To close a stream, you call `fclose`. `fclose`'s argument is the stream you want to close. Once a stream is closed, you can't read from it or write to it any more. `fclose` returns 0 if it succeeds and `EOF` if it fails. One reason `fclose` might fail is that you tried to close a stream that you never opened.

30 Command-Line Arguments

When you run a program from MS-DOS, you can give it arguments. For example, when you run the `copy` command, you give it arguments that say what files you want copied and where you want them copied to. These command-line arguments get passed in to `main` when the operating system calls `main`.

If we were writing `copy` in C, we'd need some way to examine the command-line arguments so that we'd know which files to copy.

30.1 `main`'s Header

There are exactly two legal ways to write `main`'s header.

```
int main(void)
```

we already know about—it means we're going to ignore the command-line arguments. The other header `main` can have is

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

When the program gets run, the operating system collects the arguments together. Each argument gets put into a `NUL`-terminated array of characters.

The operating system gets the address of the first element each of these arrays. These addresses have type <pointer to `char`>. It takes the addresses and assembles them into another array, an <array of pointer to `char`>. This array is called the *argument vector*.

The operating system then takes the address of the first element in the argument vector. The first element is a <pointer to `char`>, and so the address of the first element has type <pointer to pointer to `char`>. The operating system arranges that this value, the address of the argument vector, gets passed to `main` as its second argument, which is conventionally called `argv` for 'argument vector', even though it's really a pointer to the argument vector itself.

`main` has the usual problem: It's got `argv`, a pointer to the first element of an array, so it can find the elements of the array with no problem. But how does it know when to stop?

`main`'s first argument is the *argument count*, conventionally called `argc`. It's an <`int`>. It says how many arguments there are and therefore how long the argument vector is.

30.2 A Thousand Words about `argv` and `argc`

Here's a picture:

30.3 `argc` and `argv` in Practice

It takes a while to get comfortable enough with pointers to understand what's going on with `argv`. In the meantime, just remember this: `argv[0]` is a string which contains the program's first argument. `argv[1]` is a string which contains the program's second argument, and so on, up to `argv[argc-1]`, which is a string which contains the program's last argument.

30.4 The `echo` Program

Here's a program called `echo`, which prints out its command-line arguments onto the standard output:

```
#include <stdio.h>

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

  for (i=0; i<argc; i++)
    printf("%s ", argv[i]);

  printf("\n");
  return 0;
}
```

30.5 The `type` Program

Here's a program which types out the contents of the files named on its command line. It's like the MS-DOS `type` command.

```
#include <stdio.h>

int main(int argc, char **argv)
{
  int i;
  int c;
  FILE *cur_file;
  int num_errors = 0;
```

```
  for (i=1; i<argc; i++) {
    cur_file = fopen(argv[i], "r");
    if (cur_file == NULL) {
      printf("Couldn't open file %s.\n", argv[i]);
      num_errors += 1;
      continue;
    }

    while ((c = getc(cur_file)) != EOF)
      putchar(c);

    fclose(cur_file);
  }

  return num_errors;
}
```

Some notes: We used `continue`, which we haven't seen before. When the computer encounters a `continue` statement, it immediately starts the next iteration of the smallest loop it's in. Inside of `while` or `do–while` loop, `continue` skips the rest of the loop body and jumps right to the test. In a `for` statement, `continue` is the same, except it evaluates the update expression before skipping to the test. We use it here because if we can't open a file, we don't want to bother with the rest of the loop, which is for reading a file; we want to go on and try the next file immediately. `continue` starts the next pass through the loop immediately, without executing the rest of the loop body.

We use the variable `num_errors` to keep track of the number of problems we've had with the files so far, and return it when we're done. Note that this is consistent with the convention of returning 0 if your program was successful and nonzero if it encountered errors.

`argv[0]` conventionally contains the name of the command that's being run—in this case, `type`. If we entered the command

```
type foo bar baz
```

to type out the files `foo`, `bar`, and `baz`, then `argv[0]` would be `type`, and `argv[1]` would be `foo`. That's why we start opening files with the one named in `argv[1]`.

Note that we're always careful to close files with `fclose` when we're done. most systems enforce a limit on the number of files you can have open at once, so we need to recycle our streams when we're done using them.

31 Now You Know C

We just spent a long time learning a lot of ins and outs of the C language, and guess what? We're almost done. There's a very short list of language features that we haven't already seen, and some of those we won't bother with. In other words: You know the language.

This book is supposed to concern itself primarily with how to program, and secondarily with the C language itself, and now we're finally getting to studying programming instead of C.

Accordingly, we spent some time working on writing a simple program: `type` Complete code for `type` appears in section 30.5.

The point of working on this in class is to argue about style and documentation, to think about issues of formatting, and most important, program structure and how to write a good program.

32 `type`, Episode 1

The `type` program is a fundamental MS-DOS utility.[31] When you run it, you give it one or more filenames as arguments. `type` gets the contents of each file in turn and writes that data to the standard output.

This has at least uses:

- You can use `type` to view a file by dumping its contents to the screen.
- You can use `type` to concatenate[32] two files, by redirecting the standard output of `type` into a destination file.

32.1 Notes About the Design

We're writing this program with the *top-down design* method, which is one of very few effective techniques for writing a real program.[33]

[31] It's a fundamental utility in nearly all environments. For example, under UNIX, `type` is called `cat`.

[32] *concatenate* means to stick things together in a row. For example, if we concatenate the words 'foo' and 'bar' we get the word 'foobar'.

[33] Other techniques include: The *bottom-up* approach, where you implement a few fundamental primitive functions, and then some more complicated routines that depend on those, and then finally build up the program from the building blocks you've written; the *Mongolian*

Top-down design means that we start by thinking about what our program has to do, in general terms: "It will have to open a file, read the data from the file, write the data to the standard output, and close the file. It will have to do that over and over for each file named on the command line." Some of the sentences in the description correspond obviously with C constructs: 'repeat over and over' means a `while`. `do`–`while`, or `for` loop. Other sentences get turned into functions, such as `read_data_from_file()`.

This description becomes our function `main`. Then we start writing that functions that we called from `main` that aren't written yet: `read_data_from_file()`, for example. If a function is simple and we can see right away how to write it, we just go ahead; if it seems more complicated, we can top-down design it in the same way.

Functions let us break a big problem into little problems. Each function solves a little problem; then all we have to do is put the functions together. If the little problems turn out to be too big to handle, we can break them down into even littler problems.

32.2 What we got Done Today

```
#include <stdio.h>

int main(int argc, char **argv)
{
  int i;
  int c;
  FILE *cur_file;
  int num_errors = 0;

  for (i=1; i<argc; i++) {
    cur_file = fopen(argv[i], "r");
    if (cur_file == NULL) {
      printf("Couldn't open file %s.\n", argv[i]);
      num_errors += 1;
      continue;
    }

    while ((c = getc(cur_file)) != EOF)
      putchar(c);
```

hordes technique, where you just dive in and write the first thing that comes into your head. The Mongolian hordes technique is not particularly effective.

```
    fclose(cur_file);
  }

  return num_errors;
}
```

32.3 Some Notes About the Design Process

Our original plan was to have `open_file` handle errors and return 0 or 1 to `main` to indicate success or failure. But then we realized that won't work—on success, `open_file` gets a `FILE *` from `fopen`, and it *must* return this `FILE *` to `main` so that `main` can pass it off to the other functions that need it. The function to read the data from the file will need this value, as will the function that closes the file again.

So `open_file` must return a `FILE *` on success. The natural value to have it return on failure is that same one that `fopen` itself returns on failure: `NULL`. Furthermore, `open_file` isn't in a position to do any cleanup if `fopen` fails; `main` will need to skip that file completely and not try to read it or close it, but `open_file` can't do anything about that except warn `main` that something went wrong. So `open_file` ended up being a rather small function. That's all right. It's much easier to merge a small function back into the place where it's called and eliminate the call than it is to separate out a big block of code from one function and put it into a new function. Real programs often have one-line functions.

Our original plan was to have one function to read data in from the file and one to write it out again. But then we realized that since all we ever do with the data we read in is write it out again right away, we should put both things together and have one function (`spew`) which reads the data and writes it out immediately.

I want to push very hard on the idea that designs and programs do not spring forth out of your head fully-formed, like Athena from the forehead of Zeus. You get an idea, and then discover it doesn't work right, and then you fix it, like we did here. And if it doesn't work, the you can try something else.

33 Miscellaneous Things We Discussed

Actually writing a program brought up a few issues, some of them about design and others about language features.

33.1 Predefined Streams and the Standard Error Output

When you run your program, three stream variables are already set up. `stdin` is a value of type `FILE *`, which represents the standard input; similarly `stdout` represents the standard output. Doing `getc(stdin)` is *exactly* the same as doing `getchar()`—in fact, `getchar()` is usually a macro which expands to `getc(stdin)`. Similarly `printf` is usually a one-line function which calls

```
fprintf(stdout, ...);
```

to do the real work.

The third predefined stream is called `stderr`, which is short for *standard error output*. It's pretty much equivalent to `stdout`—it's normally attached to the screen, so that things you write to `stderr` appear on the screen. But it's not identical with `stdout`, and if the user redirects the standard output into a file by putting `> file` on the command line, the standard error is *not* redirected—it stays attached to the screen.

`stderr` is there so that you have a place to write error messages. If you used `printf` to write your error messages, they'd go onto the standard output, and might wind up in a file and never be seen. But if you write them with

```
fprintf(stderr, "Error in line %d.\n", lineno);
```

instead, they go onto the screen no matter where the regular output is going, and the user can see them.

33.2 `continue`

`continue` is covered in section 30.5.

It may seem like I keep pulling these things out of my hat, but really we're near the end. The only control structures we haven't seen yet are `switch` and `goto`. We won't do `goto`. I kept thinking I would bring up `switch` when we needed it (like `continue`), but it just hasn't come up yet.

33.3 Don't Ring the Bell

If you ring the bell on the computer, everyone in the room hears it.

Therefore, the question you have to ask yourself before writing a program that rings a bell to signal a certain event is, "Is this event so important that everyone in the room needs to know about it?"

Usually, the answer is 'no'.

33.4 Don't Worry About the Output Device

One person said we should read the input and write the output in 80-character chunks, because the screen is probably 80 characters wide.

This is silly. The input might be something like:

```
short
words
stars
groak
fubar
flesh
```

in which case the fact that the screen is 80 characters wide is completely irrelevant—the lines in the input are only 5 characters each. On the other hand, maybe the lines in the input are very long, 2000 or 300 characters.

The specification for the program calls for "Write the data to the standard output."

Whenever you start thinking about things like the screen width, you have to step back and ask if the screen width is really relevant. The way to ask this is to say to yourself, 'Does it make sense to run my program on a terminal that prints the output on paper?' In this case the answer is 'yes'. Sure, you might want to type out the contents of some files, even if you're typing out on paper. Ask yourself 'Would it make sense to run my program if the screen were 1,065 characters wide?' Sure, why not? We're just typing out files.

Even though *our* displays are 80 characters wide, it's not relevant to the program. So don't put it in.

34 Problem 2.2 (`strrev`) from the Exam

> Write the function `strrev`, whose argument is a string, and which reverses its argument in place. That means that if we do:

```
char word[] = "Foo";
printf("%s\n", word);
strrev(word);
printf("%s\n", word);
```

the output should be `Foo`, followed by `ooF`.

This was one of the hard problems on the exam. Nobody actually turned in a working solution, although several people came close. First I'll show a working solution, and then we'll discuss some alternative solutions.

One thing a lot of people did was to write a program that actually printed out the string backwards. People who did this correctly got half credit, but it wasn't what the function was supposed to be about—the question asked for a function that would reverse a string 'in place'. That means that we want to be able to tell the function where our string is (presumably via a pointer), and the function will find the data there and reverse it and leave the answer in memory in the same place that the original word was. No input or output was required. This should have been clear from the example: `word` contains `Foo` when we print it out the first time; we run `strrev`, and nothing is printed out until the second `printf`, which demonstrates that `word` now contains `ooF`.

When you're asking yourself how to reverse a string in place, you might think to yourself, "Well, I could copy the string backwards into some auxiliary space, and then copy the reversed string back from the auxiliary space to the original space." To do that you need to know how to ask for extra space on the fly, and we didn't know how to do that at the time we took the exam.

34.1 A Solution

The solution I was hoping for took some ingenuity to find: We have two pointers, `s` and `e`. `s` starts out pointing to the first character in the string, and `e` starts out pointing to the last character in the string. We swap the characters that `s` and `e` are pointing to; that's easy. Then we increment `s` to point to the second character, and decrement `e` to point to the next-to-last character. We repeat, until both pointers are pointing to the middle letters of the string; then we're done.

Here's the code.

```
void strrev(char *s)
{
  char *e;
  int left=0;                          /* Count of characters left to swap */
  char temp;                           /* For swap */

  /* If length of string is less than 2, don't bother. */
  /* Note short-circuiting here—it's very important. */
  if (s[0] == '\0' || s[1] == '\0')
    return;

  /* First, point e at end of s and compute length of s: */
  for (e=s; *e != '\0'; e++)
    left++;

  /* e now points to NUL character at end of s. */
  /* left is the number of characters we have to swap. */

  e--;
  /* e now points at last character in s. */

  while (left > 1) {
    temp = *s; *s = *e; *e = temp; /* Swap characters at beginning and end */
    s++; e--;                      /* Move towards middle of string */
    left -= 2;                     /* two fewer characters to swap. */
  }

}
```

There's nothing new here, so it was fair game for the exam.

34.2 A Non-Solution

Some people did think of the auxiliary-space method, and handed in something like this:

```
void strrev(char *s)
{
  int i, len = strlen(s);
  char aux[len+1];              /* You can't do this */
```

```
  aux[len] = '\0';

  for (i=0; i<len; i++)
    aux[i] = s[len-1-i];        /* Copy string backwards */

  for (i=0; i<len; i++)
    s[i] = aux[i];              /* Copy string forwards */

  return;
}
```

This is ingenious, but it doesn't work. The catch is that an array's size must be determined at the time the program is compiled.[34] Nevertheless, people who did this correctly (not counting the illegal array declaration itself) got three-quarters credit.

34.3 Another Solution with `strdup`

Here's another auxiliary-space solution—this one does work. Unfortunately, nobody could have turned this in because we hadn't covered `strdup` by the time we had the exam:

```
void strrev(char *s)
{
  int i, len = strlen(s);
  char *aux;

  aux = strdup(s);

  ... /* Same as above */
}
```

`strdup`'s argument is a string. `strdup` does this:

1. It looks at the string,
2. It finds out (probably with `strlen`) how much space is necessary to store the string, (one byte for each character in the string, and an extra byte for the `NUL` character that terminates it),

[34] Some compilers do allow variable-sized arrays, as a non-portable extension.

3. It somehow finds and reserves enough space for a copy of the string,
4. It copies the string into the memory it's reserved, and
5. It returns a pointer to the first character in the new copy of the string.

If `strdup` fails, for example because it can't reserve enough free space for a copy of its argument, it returns the `NULL` pointer.

When we say that `strdup` 'reserves' space, we mean that the space won't be used for something else until we tell the computer that it's all right to do so. We can be sure that future calls to `strdup` will find other space, and that variables we declare will be allocated out of space other than that reserved by `strdup`.

34.4 `free`

The space reserved by `strdup` stays reserved until the program terminates or until we explicitly make the space available for re-use. This is called *freeing* the space; we do it with the `free` function. If `p` is a <pointer to `char`> which points to space reserved by `strdup`, the call `free(p)` frees that space; a variable might get put there later, or a future call to `strdup` might copy new data there and return a new pointer to it.

35 Comparing Characters and Strings

The comparison operators `<`, `<=`, `>`, and `>=` all work on characters as well as on numbers, in a reasonable way.

35.1 Comparing Single Characters

It's definitely true that:

```
'0' < '1' < ... < '8' < '9'
'A' < 'B' < ... < 'Y' < 'Z'
'a' < 'b' < ... < 'y' < 'z'
```

The relative ordering of these three classes, however, is implementation-dependent. Depending on what kind of machine you are using, it might be true

either that `'A' < 'a'` or that `'a' < 'A'`. On the machines we use, it's the case that `'9' < 'A' < 'Z' < 'a'`.

The `NUL` character is always less than any printable character. A printable character is one that makes a space or a mark on the screen.

We can use this character ordering to write a function that compares strings alphabetically.

35.2 Comparing Strings

We'll write a function which accepts two strings as arguments and returns some kind of answer to say which string comes earlier in an alphabetical ordering. We'll do something reasonable for nonalphabetic strings.

We'll say that a string is *less* than another string if it comes before that string in the dictionary. For example: `bar` is less than `foo`; `food` is less than `fool`; `fool` is less than `foolish`. Please note that 'less' does not mean 'shorter'. `foolish` is less than `zebra`, but it is not shorter than `zebra`.

Our function will examine the first character of each argument. If the characters are different, we know right away which string is least: The one whose first character is least. We can return an answer right away in this case. We'll return `-1` if the first argument is less than the second, and `1` if the first argument is greater than the second.

If the first two characters are equal, we'll go look at the second two characters, and soforth.

We keep doing this until either we find two characters that don't match (in which case we return `1` or `-1` as above) or until we reach the end of one or both strings.

If we reach the end of both strings at once, then they were identical, and we return `0`.

Otherwise, one of the strings is a *prefix* of the other, which means that it is just the same as the other string, only it is missing some characters off the end; for example, `foo` is a prefix of `foolish`. In this case the shorter string is least, and we should return `1` or `-1` as above.

That said, here's the code we wrote in class:

```
int strcmp(char *s1, char *s2)
{
  while(1) {
    if ( *s1 == '\0' && *s2 == '\0' )
      return 0;                  /* strings are identical */
    else if (*s1 == '\0')
      return -1;                 /* s1 is a prefix of s2 */
    else if (*s2 == '\0')
      return 1;                  /* s2 is a prefix of s1 */
    else if ( *s1 < *s2 )
      return -1;                 /* s1 is less */
    else if ( *s1 > *s2 )
      return 1;                  /* s2 is less */
    else {
      s1++; s2++;
    }
  }
}
```

This code works. It's functionally equivalent to a standard library function called `strcmp`, which operates in the same way. `strcmp` returns negative, zero, or positive, depending on whether its first argument is less than, equal to, or greater than its second argument. (`strcmp` doesn't always return -1, 0, or 1.) To use `strcmp`, you must include `<string.h>`.

35.3 Case-Insensitive Comparison

Our `strcmp` function does a thing that we might find peculiar: It says that `Snider` is less than `food`, because, in Turbo-C++, capital letters are always less than lowercase letters. We say that `strcmp` is *case-sensitive*, because it treats the words `Snider` and `snider` differently. We might wish to erase this distinction and make `Snider` greater than `food`, because `s`, capital or not, comes after `f` in the dictionary.

To do this, we can use the standard function `tolower`. `tolower` accepts one argument, which is a character. If the character is not an uppercase letter, `tolower` returns the character it was passed; if the argument was an uppercase letter, `tolower` returns the lowercase version of that letter. For example, `tolower('X')` is `'x'`; `tolower('y')` is `'y'`; `tolower('2')` is `'2'`.

What we'll do is convert the characters in our strings to lowercase before we compare them; then the `S` in `Snider` will behave as though it were a lowercase S.

Replace the lines

```
else if ( *s1 < *s2 )
  return -1;
else if ( *s1 > *s2 )
  return 1;
```

with

```
else if ( tolower(*s1) < tolower(*s2) )
  return -1;
else if ( tolower(*s1) > tolower(*s2) )
  return 1;
```

. The function we get is called `strcasecmp` on UNIX systems, but it seems to be called `stricmp` or `strcmpi` under Turbo-C++.

`tolower`, and a collection of similar functions, such as `toupper`, are declared in `<ctype.h>`.

36 `fgets`

`fgets` is an awfully useful function. It takes three arguments. The third argument is a stream. `fgets` reads characters from the stream. The first argument is a pointer to a space in which `fgets` can store the characters it reads. The second argument is an `<int>`; it says how big that space is. If the second argument is n, then `fgets` will read no more than $n - 1$ characters from the stream and store them into the buffer. It might read fewer, because `fgets` stops after it reads a newline character.

After storing the characters it read into the buffer, it sticks a `NUL` character on the end. (That's why it reads at most $n-1$ characters; it needs to save space for the `NUL`.)

If it succeeds, `fgets` returns its first argument, which is not usually very useful since we already know what the first argument was. But if it fails, `fgets` returns the `NULL` pointer.

In short, `fgets` reads a line of input from a certain stream, stopping when the buffer it's storing the input into is full.

36.1 `gets`

`fgets` has a dysfunctional little brother, `gets`. `gets` only reads from the standard input, its only argument is a pointer to the space to in which you want the input stored.

The question: How does `gets` know how much space you've arranged for it?

The answer: It doesn't.

If the line that `gets` is reading is too big to fit in the array you've provided, `gets` happily writes past the end of the array and destroys who-knows-what.

For this reason, serious programmers never use `gets`.

37 Debugging Facilities

Turbo-C++ has reasonably good debugging facilities. You can step through part of your program one statement at a time, watching how certain variables change, and you can run your program normally and have it pause when control reaches a certain line.

This is all covered in detail in your texbook, pages 723–733.

37.1 Stepping

The `F7` and `F8` keys will run your program one step at a time; each time you press one, your program will run one more statement. When you press `F7` or `F8`, the program runs, and stops again when control reaches the next line.

The difference between `F7` and `F8` is that if the current line contains a function call, the pressing `F7` will go stop at the next executed line, which is inside the called function, but `F8` will run the function calls and stop only when control reaches the next line in the calling function. `F8` skips the details of the function calls; they're run, but you don't have to step into them and go through the one step at a time like you do with `F7`.

37.2 Inspecting Variables

Under the `Debug` menu is a selection called `Inspect`. If you select this menu item, you'll get a dialog box that lest you enter an expression. A window will

pop up on the screen that displays the value of the expression. For example, if you wanted to see the current value of the expression `*ptr`, type `*ptr` and a window will pop up with the value of `*ptr`. The value in this window will change if the value of `*ptr` changes as you step through the program.

37.3 Setting Breakpoints

Normally when you pick `Run` from the `Run` menu, the program runs without stopping. You can set a *breakpoint* at a line of code, and execution will pause when control reaches that line. Then you can examine the values of variables with the inspector, step through critical sections of code one step at a time, or set or clear more breakpoints.

To set a breakpoint on a line, move the cursor to that line and choose `Toggle Breakpoint` from the `Debug` menu. When you `Run` the program, control will stop when control reaches the line with the breakpoint. To continue execution normally, choose `Run` again; the program will pick up where it left off and run until it hits another breakpoint.

To remove a breakpoint, move the cursor to the line with the breakpoint and choose `Toggle Breakpoint` again.

37.4 Restarting the Program

The `Run` command only starts your program from the beginning if you've changed it or if it's completely finished. If you've stopped at a breakpoint or stepped partway through the program, the `Run` command tells Turbo-C++ to continue running the program where it left off.

If you really to want to start all over again from the beginning, choose `Program Reset` from the `Run` menu.

37.5 Setting Command-Line Arguments

You don't have to break into a DOS shell to run your program with command-line arguments. If you choose `Arguments` under the `Run` menu, you can enter the arguments you want to run your program with.

38 Dynamic Memory Allocation

We are finally going to learn how to allocate memory on the fly, getting more when we need it, a subject not usually covered in introductory C courses.

38.1 `malloc`

We've already seen `strdup`, which finds memory somewhere and reserves it. How did `strdup` find that memory?

Chances are it called `malloc`. `malloc` finds memory by asking the operating system for a big bunch of memory and then parceling it out a little at a time as you ask for it.[35] It's `malloc` that takes care of recording how big each parcel is so that `free` can free it properly.

To call `malloc`, pass as an argument the number of bytes of space you want to allocate. `malloc` will reserve that many bytes and return a pointer to the memory it found. If it fails for any reason, such as because all the memory is already reserved, `malloc` returns (all together now) the `NULL` pointer.

38.2 `sizeof`

Frequently you want to allocate enough memory to store an object of a certain type; you need to be able to tell `malloc` how many bytes of space the object takes up so it knows how much to allocate. There's an operator in C which tells you how big an object is: `sizeof`.

If *type* is the name of a type, the value of the expression `sizeof(`*type*`)` is the number of bytes needed to hold that type; for example, on our machines, `sizeof(int)` is 2 and `sizeof(double)` is 8.

If we want to get enough space for an array of 13 <`int`>s, we can do `malloc(13 * sizeof(int))` or even `malloc(sizeof(int[13]))`.[36]

There's another way to use `sizeof`: If e is an expression with type t, then the value of `sizeof(`e`)` is the same as the value of the expression `sizeof(`t`)`.

[35] `malloc` does it this way, instead of going to the operating system every time, because operating system requests are very slow.

[36] `int[13]` is the name of the type <array of 13 `int`s>. To construct the name of a certain type, just write a declaration for a variable of that type, and then discard the variable name and the semicolon from the declaration.

39 structs

We've seen how to use an array to store many values of the same type, and even to treat the collection of values as a unit. Now we'll look at the complementary type, the *structure*, which allows you to group several different object together as a unit.

39.1 Creating a New Structure Type

The typical example of a structure is this: You want to keep track of the employees in your organization. Each employee has a name (a string of no more than 50 characters), a social security number (a <long int>), and a salary (an <int>). You want to handle several such records.

You can create a new type, a `struct employee`, which keeps all of this information in one place. First, you define the new type:

```
struct employee {
  char name[51];
  long int ssn;
  int salary;
} ;
```

When you write this in your program, it defines a new type, the type `struct employee`. A variable of type `struct employee` has three *members*: `name`, an <array of 50 `chars`>, `ssn`, a <long int>, and `salary`, an <int>.

`employee` is called the *tag* of the structure.

39.2 Creating `struct` Variables

To declare a variables of type `struct employee`, just write:

```
struct employee smithers, marketing[12], *the_employee;
```

`smithers` is a `struct employee`; `marketing` is an array of twelve of these structures, and `the_employee` is a pointer to one of these structures.

39.3 The . Operator

Suppose `smithers` is a `struct employee` variable, as above. Suppose we want to store the information that Smithers' social security number is `314-15-9265`. Here's how we would do that:

```
smithers.ssn = 314159265;
```

The `.` operator is the *structure member* operator. Its left operand must be a structure of some kind; its right operand must be the name of one of the members of that structure. The entire expression refers to the specified member of the specified structure. So similarly,

```
if ( smithers.salary > 10000 )
  ...
```

asks if Smithers' salary exceeds ten thousand dollars.

Similarly, let's suppose the array `marketing`, which is an array of 12 `struct employees`, has been initialized with the names, salaries, and social security numbers of the members of the marketing department. Here's how we'd compute the average salary in the marketing department:

```
long int totalsalary = 0;

for (i=0; i<12; i++)
  totalsalary += marketing[i].salary;
printf("Average salary is %f.\n", totalsalary/12.0);
```

`marketing` is an array of `struct employees`, so `marketing[i]` is a `struct employee`, and `marketing[i].salary` is the salary member of that `struct employee`.

As a final example, this is some code which prints out the name and social security number of the highest-paid member of the marketing department:

```
int highest_salary = 0;
int i, highest_salary_index;

for (i=0; i<12; i++)
  if (marketing[i].salary > highest_salary) {
    highest_salary = marketing[i].salary;
    highest_salary_index = i;
  }

printf("The highest-paid employee is %s (%d).\n",
       marketing[highest_salary_index].name,
       marketing[highest_salary_index].ssn);
```

40 The Linked List

Suppose we want to write a program which reads an input file, counts the number of occurrences of each different word in the input, and then prints out a report of the form "`There were 27 occurrences of the word 'I', 53 occurrences of the word 'the'...`". You might want to have an array that will hold words and another that will hold counts, but that doesn't work, because the arrays have a fixed size, say 500 elements each, and if the input contains 501 different words, you're stumped. We need to have a data structure that can grow arbitrarily large if we need it to.

The simplest example of such a data structure is the *linked list*. A linked list is a collection of *nodes* in some order; each node contains some information about how to find the 'next' node, and perhaps some auxiliary information as well. The last node has some kind of marker that says it's last.

To keep track of a linked list, all we need to do is remember where the first node is. Then that first node contains information (called a *link*) that tells us where the second node is; the second node has a link to tell us where the third node is, and soforth. Clearly the list can be any length.

40.1 The Nodes are Structs

Here's a simple way to implement a linked list: it keeps track of as many numbers as we like:

```
struct listnode {
  int data;
  struct listnode *next;
} ;
```

This `struct` has two members: `data`, which actually holds a number, and `next`, which points to the next node in the list.

40.2 Adding a New Node to a List

Suppose the address of the first node in the list is stored in the variable `firstnode`, whose type is `struct listnode *`. Suppose also we discover that we need to remember one more number, `newnumber`. How can we add that number onto the list of numbers in a list of these nodes?

```
struct listnode *temp;

temp = malloc(sizeof(struct listnode));

(*temp).next = firstnode;
(*temp).data = newnumber;
firstnode = temp;
```

First we create enough space for a new list node, with `malloc`; then we link the node to the rest of the list by setting its `next` pointer to point to the old first node. Then we store the number we want to remember into the data member of the new node. Then we remember that the new node is now first, with `firstnode = temp`. We can do this as many times as we want, until the memory runs out.

40.3 Getting the Data Back Out of the List

Now suppose we want to average the numbers stored in each node. It's easy:

```
struct listnode *current;
int average;
long int sum = 0L;
int nodecount = 0;

for (current = firstnode;
     current != NULL;
     current = (*current).next) {
  sum += (*current).data;
  nodecount += 1;
}

average = sum / nodecount; /* round down */
```

Here we've assumed that the `next` pointer of the last node in the list is `NULL`. This makes perfect sense, and we can be sure that no other node but the last has `NULL` for its `next` pointer, because all the other `next` pointers point to other nodes, whose addresses are not `NULL`.

I hope it's obvious how this is all relevant to the problem of managing an arbitrarily large stack for your calculator.

40.4 The -> Operator

The operation of accessing the member of a structure via a pointer to the structure is common. We've used the construction `(*foo).bar` several times already. C lets you use a short cut.

In all circumstances, the expression `foo->bar` is identical with `(*foo).bar`. `foo` must be a pointer to a structure, and `bar` must be the name of one of the members of that structure.

40.5 Other Operations on Structures

The things you can do with a structure are limited. You can find its address with `&`, and you can access its members with `.` .

Since the ANSI standard, some things which were Too Much Work are no longer considered to be too Much Work:

You can pass a structure as an argument to a function (in which case the function gets a copy of the structure, the same way it does when you pass an

<int>) and you can return a structure value as a return value from a function.

You can compare two structures of the same type for equality or inequality with == and !=; two structures are 'equal' if each pair of corresponding members are equal.

You can assign a structure value to a variable of the right structure type with =; the members are copied one at a time.

You can't do anything else with a structure.

41 Global Variables and Type Declarations

Normally, you declare a variable inside a function. Then the variable is created when the function is called, is destroyed when the function returns, and the name of the variable is known only within the function.

This holds for structure type definitions also; if you define a structure type inside a function, the type and its name are only known within that function. That's not too useful; we would like the structure type definition to be visible everywhere.

If you write a declaration or type definition outside of all the functions, it is called a *global* variable or definition. It becomes visible to every function that follows the declaration or definition in the entire file. We have been doing this with function prototypes all along; in fact you can put a function prototype inside another function, and the information that the prototype communicates is only known locally, and now outside the function in which the prototype appears. Doing this is rarely useful.

If you write `int foo;` at the top of your program, outside all the functions, then every function in your entire file will have access to the variable `foo`. If one function changes the value of `foo`, the others can see the change; the name `foo` refers to the same variable, no matter which function is using it. Furthermore, unlike a local variable, which is created when the function that owns it is called and destroyed again when the function returns, a global variable is created when your program is run and is not destroyed until the program completes.

A global variable is an abnormal communication mechanism; functions can use it to communicate values back and forth even though it may not be obvious that they're doing so; for this reason, you should avoid using global variables in your program. There are occasions when it is appropriate: when some large piece of information is really global, and most of the functions make frequent

references to it.[37]

Function prototypes, structure type definitions, manifest constants, and other information that doesn't change, on the other hand, are appropriate things to make global.

42 A Program to Count Words

At the end of these notes you'll find the complete code for the example we chipped away at in class: It reads 'words' from the standard input, counts the number of occurrences of each word, and prints a report when it's done. It uses a linked list to keep track of what words it seen and how many times it's seen each word. There is no arbitrary upper bound on the number of different words it can handle.

```
/* Program to read words from standard input, count number of
   occurrences of each word, and write report onto standard output.

   A 'word' is a sequence of non−whitespace characters.  Words therefore
   are separated by whitespace.

   28 July 1992  Mark−Jason Dominus (MJD) mjd@saul.cis.upenn.edu

   Copyright 1992 Mark−Jason Dominus and the Trustees of the University
   of Pennsylvania. All rights reserved.
*/

#include <stdio.h>
#include <ctype.h>              /* For isspace() */
#include <string.h>
#include <malloc.h>

#define MAXWORDLEN 80           /* We fail on words longer than
                                   80 characters */

char * getword(void);
struct wordcount_node * count(char *word, struct wordcount_node *list);
```

[37] For example, a program might read a *configuration file* that specifies certain details about how the program should work; the information in the configuration file is stored in a large structure. Nearly every function has to consult one or another member of this structure, so it might be appropriate to make the structure variable global.

```
struct wordcount_node * create_new_node(void);

struct wordcount_node {
  char *word;
  int count;                        /* number of occurrences of this word */
  struct wordcount_node *next;      /* pointer to next node in list */      30
} ;

int
  main(void)
{
  char *newword;
  struct wordcount_node *list;      /* Pointer to list of words seen so far */
  struct wordcount_node *cur;

                                                                            40
  list = NULL;                      /* Empty list */

  /* Read in words and count them */
  while ((newword = getword()) != NULL) {
    list = count(newword, list);
    free(newword);                  /* 'count' made a copy of 'newword', so
                                       we can throw 'newword' away. */
  }

  /* Print out accumulated information */                                   50
  for (cur = list; cur != NULL; cur = cur->next)
    printf("%s %d\n", cur->word, cur->count);

  return 0;
}

/*
  search for 'word' in 'list'.
  If it's there, increment the count associated with it.
  Otherwise, create a new node, and attach it to the front of the list.    60
  In either case, return a pointer to the head of the list when done.

  Note that this works correctly if 'list' is NULL. */

struct wordcount_node *
  count(char *word, struct wordcount_node *list)
{
  struct wordcount_node *cur;
```

```
  struct wordcount_node *newnode;
                                                                    70
  for (cur = list; cur != NULL; cur = cur->next)
    if (strcmp(word, cur->word) == 0) {
      cur->count++;
      return list;
    }

  /* If we're here, we didn't find the word. */

  if ((newnode = create_new_node()) == NULL) {
    fprintf(stderr, "Out of memory.\n");                            80
    exit(1);
  }

  /* Attach new node to rest of list */
  newnode->next = list;

  /* We don't know if our caller is going to reuse or overwrite
     the memory that contains 'word', so we'd better copy it if we want
     to keep it around. */
  newnode->word = strdup(word);                                     90

  newnode->count = 1;

  return newnode;
}

/* Create a new list node and return a pointer to it.
   Return NULL if out of memory or on some other failure.
   */
struct wordcount_node *                                             100
  create_new_node(void)
{
  return (struct wordcount_node *) malloc(sizeof(struct wordcount_node));
}


/* Read a word from standard input; return NULL on EOF.
   Has poor behavior on words longer than MAXWORDLEN characters.
   */
char *                                                              110
  getword(void)
{
```

```
  char buf[MAXWORDLEN];
  int i=0, c;

  /* Eat up white space that precedes a word, if any. */
  while ( isspace(c = getchar()) && c != EOF   )
    /* nothing */ ;

  if (c != EOF) ungetc(c,stdin);                                    120

  /* Read word */
  while ( !isspace(c = getchar()) && c != EOF   )
    buf[i++] = c;

  buf[i] = '\0';

  if (i == 0) return NULL;       /* We hit EOF without reading anything */
  else  return strdup(buf);
}                                                                   130
```

42.1 Notes on the Code

What follows here are notes on the code; the numbers in the margins are source code line numbers. I hope you will read the code carefully and try to understand what each part is doing. The notes only explain the fine points; most of the details of how the program works are not explained in the notes.

16 `<ctype.h>` is here to provide the definition of `isspace`, which we use later, in function `getword`.

18 `<malloc.h>` declares `malloc`, `free`, and other related functions.

27 This is the defintion of the ystructure type we'll use for a node in our linked list. Each node has three parts: A pointer to the word it represent, a count of the number of occurrences of that word seen so far, and a pointer to the next node.

41 We could describe a list of n nodes this way: It has a head node, which has some data associated with it, and which has a pointer to a list of $n-1$ elements. So a list of 2 nodes has a head node, which has some data and which has a pointer to a list of 1 node. A list of 1 node has a head node, which has some data and which has a pointer to a list of 0 nodes. But this latter pointer is actually `NULL`, because the head node is the last node in the list, so we've just suggested that it might be wise to consider the `NULL` pointer as a 'pointer to a list with 0 nodes'.

When we initialize `list`, we initialize it with a 'pointer to the empty list', `NULL`, and it turns out that the function `count`, which manipulates lists, does the right thing if we pass it this pointer.

45 `count` might append a new node onto the head of the list, and we need a way to apprise `main` of that fact, so that it can remember where its list starts. We do this by returning a pointer to the new head node from `count` after we've attached it, and main just stores this pointer in `list`, the variable it was using to hold a pointer to the first node in the list.

We wouldn't have to bother with this if `count` just attached the new node to the tail of the list instead of to the head, but that's uaully harder to do, because the head of a list is usually easier to find than the tail.

46 `getword` allocated space for `newword`; we passed `newword` to `count`, which in turn made a copy of it to store in the list, and so the copy we got back from `getword` can be freed now.

Why have `count` copy `newword` at all? If you're writing a function like `count` and you need to save some data that was passed in, it's a good idea to make a copy because you never know when your calling function might decide to destroy the original.

51 Note that this code works even if the input contained no words at all: In that case, the body of the `while` loop on lines 44–49 was never executed at all, and `list` is still `NULL`; we exit the program without printing anything.

66 `count` accepts two arguments: a word and a pointer to the first node in a list. It searches the list for a node whose `word` member is `word`; if it finds it, it performs its primary function and increments the count in that node. Otherwise, it manufactures, initializes, and links in a new node. In either case it returns a pointer to the new head node of the list (which might be the same as the old head node) to tell its caller whether or not the list has gotten longer.

71 Note that this code also works if the calling function passes in `NULL` for `list`: the condition on the `for` loop fails immediately and we proceed to like 77, where we begin the process of adding a new node to the list.

81 We used the `exit` function here. `exit` completely terminates the program when it is called; control returns to the operating system. Accordingly, `exit` does not have a return value. Contrast `exit`, which terminates 'normally', with `abort`, which terminates 'abnormally'. `exit` accepts one argument, which is treated the same way a return value from `main` is; in fact, the effect of `exit(n)` is identical to that of a `return n;` from `main`.[38] Here we `exit` with a return status of `1`, signalling that something went wrong.

90 We copy `word` here; see the note for line 46.

92 If we're initializing a new node, we initialize its count to 1, because we've seen exactly one occurrence of the word that the node represents.

100 `create_new_node` is a separate function for the usual reasons:

1. It's functionally separate from the other functions.
2. It might one day be called from more than one place, since it performs a generally useful function.
3. It might one day change and include more complicated node-building apparatus such as initializations.
4. If we ever again need a function that allocates and returns a list node, we might be able to come steal this one.

103 My compiler complains about this line, because the return value from `malloc` is a <pointer to `char`>, which is implicitly converted to a <pointer to `struct wordcount_node`> when we return it from the function. Usually,

[38] It might be more accurate to say that a `return` from main is identical to a call to `exit`, because compilers frequently compile it as one.

if you're converting pointers in this way, it means you made a mistake. On some machines, certain pointer conversions will fail altogether.[39] However, the value returned from `malloc` is an exception: The memory it points to is guaranteed to be suitable for storing any variable whatever, and the pointer is guaranteed to be freely convertible to any other pointer type. `malloc` must take special pains to ensure that this is the case. The compiler complains only because it doesn't know enough about `malloc`.

111 I wrote this function two years ago, and I've been re-using it ever since.

113 We allocate a buffer of `MAXWORDLEN` characters to hold the input word, but this function never checks to make sure it isn't writing past the end of the buffer. If the input contains an extremely long word, `getword` will happily write past the end of `buf`. This is a serious error.

117 `isspace` is a function whose argument is a character; it returns **true** if the argument is a white space character and **false** otherwise. White space characters include the blank, tab, and newline. `isspace` was declared in `<ctype.h>`. (Line 16.)

120 We haven't seen `ungetc` yet; it's like the opposite of `getc`: it 'unreads' a character. After you `ungetc` a character onto a stream, the next attempt to read the stream will proceed as if you had never read that character; the first character returned by the reading function will be the character you 'unread'. `ungetc` is limited: You can't un-get more than one character at a time, and you can't un-get a second character until you've read the first one back. `ungetc` is declared in `<stdio.h>`.

128 the loop on lines 123–124 reads characters from the standard input into the buffer `buf`[40], and stops when it reads a space or hits `EOF`. Now suppose the last word in the file was not followed by a space; say the last word is `visible`, and then after the `e`, nothing. `getword` has read `visible` into the buffer, and has just hit `EOF`. What do we want it to do?

We do not want `getword` to return `NULL` the instant it sees `EOF`, because if it did our caller would never find out about `visible`. So instead, we return `NULL` only if there is nothing in the buffer; that is, when `i == 0`. In the case of `visible`, `i` is 7, so we duplicate `visible` and return the copy; then the *next* time `getwords` is called, it hits `EOF` right away, without reading any characters into the buffer `buf`, and so `i` is 0 and it returns `NULL`.

[39] For example, consider a computer with a main processor and an auxiliary processor which is very good a floating-point arithmetic and which has its own auxiliary memory, optimized for storing floating-point numbers. The compiler might very well decide to store <`int`>s into the main memory and <`float`>s into the auxilliary memory; in that case you wouldn't be able to interconvert <pointer to `int`> and <pointer to `float`>. This is a somewhat contrived example.

[40] *buffer* is a generic word that describes a part of memory that input is being read into.

43 Doubly-Linked Lists

We've seen how to use a linked list to keep track of an arbitrarily large amount of information, growing the list as we need to. We can search through the list and look for something, or print out all the data in the list.

As a further illustration of dynamic memory techniques, let's suppose that we need to be able to walk through a list backwards as well as forwards. That's the one thing we can't do well with the linked lists we've seen: Once we get to a node, we don't remember how we got there.

The solution is simple, of course: Instead of one pointer per node, we have two: One that points forwards and one that points back.

In class we developed code to read an arbitrarily long list of integers from the user, terminated by a `0`, print the numbers out in the order they were entered, and then print the numbers in reverse order. We needed to traverse the list in both directions.

43.1 A Program to Print a List of Numbers Forward and Backward

The code we developed in class appears at the end of these notes; these are some comments on various details of the code.

We had a big argument in class about one of these details: The problem of how to start the list off. There were two main camps: One camp said that we should start the list off with a 'bogus node', so that the functions that operated on lists would always have something to work on. The other camp held that the pointers to the head and tail nodes of the list should start out `NULL`, and that the functions that operate on lists should check for that and behave slightly differently, if necessary, if they saw that they were operating on an empty list.

Neither method has advantages.

Disadvantages of the 'bogus node' solution include:[41]

- The head and tail of the list are now drastically different, because the head

[41] We said in class that another disadvantage is that the bogus node must be distinguished in some way. In this program, we distinguish it by storing `BOGUS_VALUE` into its `number` member, but in some circumstances a ready-made bogus value may not be available. However, if no appropriate bogus value had been available, we could have distinguished the bogus node because it would be first in the list and therefore it would have a `NULL` pointer in its member for pointing to the previous node in the list.

has a bogus node, but the tail doesn't. This means that the functions to walk through the list from head to tail will look substantially different, whereas they probably should look almost the same since they are doing almost the same thing.

- A list with no nodes is a perfectly reasonable entity. To avoid it for no reason smacks of superstition akin to the fear of the numeral zero prevalent in Europe in the Twelfth Century.

Disadvantages of the '`NULL` pointer' solution include:

- You might have to complicate the functions that manage lists to include special cases for empty lists.
- This complication will slow down the list-managing functions.

43.2 The Code

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

#define BOGUS_VALUE 0        /* Terminates input */

struct node {
  struct node *back;
  int number;
  struct node *forward;                                          10
} ;

int getnum(void);

/* returns pointer to new tail of list. */
struct node * store_input(int input, struct node *tail);

void print_forward(struct node *head);
void print_backward(struct node *tail);
                                                                 20
int main (void)
{
  int input;
  struct node *head, *tail;
```

```
  tail = head = malloc(sizeof(struct node));
  head->forward = NULL;
  head->number = BOGUS_VALUE;
  head->back = NULL;
                                                                    30
  while ((input = getnum()) != BOGUS_VALUE) {
    /* store input in the list */
    tail = store_input(input,tail);
  }

  /* print out forwards */
  print_forward(head);
  /* print out backwards */
  print_backward(tail);
                                                                    40
  return 0;
}

struct node * store_input(int input, struct node *tail)
{
  struct node *newnode;

  newnode = malloc(sizeof(struct node));
  newnode->number = input;
  tail->forward = newnode;                                          50

  newnode->forward = NULL;
  newnode->back = tail;

  return newnode;
}

void print_forward(struct node *head)
{
  head = head->forward;                                             60

  while(head != NULL) {
    printf("%d\n", head->number);
    head = head->forward;
  }
}

void print_backward(struct node *tail)
{
```

```
  while(tail->number != BOGUS_VALUE) {                    70
    printf("%d\n", tail->number);
    tail = tail->back;
  }
}

int getnum(void)
{
  int i;
   scanf("%d", &i);
  return i;                                               80
}
```

43.3 Notes on the Code

5 This is the definition of the bogus value that we're going to use to mark out bogus node. It's also the value that the user uses to indicate end-of-input. It's good to use the same value for both things: because the value marks the end-of-input, it can never be one of the data items we have to remember, and so therefore it's an ideal bogus node marker.

7–11 It might have been better to put the two pointers in this structure together, to emphasize the way they were related, but we put things in this order so that they would look good on the blackboard. Now that out program works, we can forget the blackboard, and we should probably reorder the structure members.

13 We wrote the prototype for `getnum` way in advance, as soon as we knew what we wanted the function to do, before we wrote the function itself. If we'd been paying more attention, we could have used the prototype to help remind us of how to write `getnum`. It would have been even better to hang a comment near the prototype.

24 `head` and `tail` are pointers to the first and last nodes in the list, respectively. We probably ought to add a comment to that effect.

26–29 These lines set up the bogus node at the head of the list. Note that we forgot to check for an error (`NULL`) return from `malloc`, and if `malloc` fails here we'll go and set the members of a nonexistent structure. Therefore this program has an error.

33 `store_input` is the function that appends a new node and a new datum to the end of the list. It returns a pointer to the new last node, so that `main` can keep track of where the last node is at all times.

It might have been more elegant to have `store_input` return `void`, and to pass in a pointer to `tail` itself instead of a copy of `tail`'s value. Then `store_input` could have changed `tail`'s value without intervention from `main`.

44–56 `store_input`, as promised, is simple; we have to manufacture a new node, fix its three members, and link it into the list by making the old `tail` node's `forward` member point to the new node. This last step, `tail->forward = newnode`, is the only one that would fail if the `tail` that were passed in were `NULL`; to make `store_input` work when `tail` is `NULL`, you need just skip this single statement. (This is not supposed to be obvious; please work through the code and see what happens.) So to convert `store_input` to operate on an empty list is a matter of adding one line of code: `if (tail != NULL)` just before `tail->forward = newnode`.

48 We forgot to check `malloc` here for an error return. This can result in disaster. A full solution would be to have `store_node` return `NULL` on failure, and have `main` check for a `NULL` return from `store_node` and abort if there was one.

58–66 `print_forward` uses the `NULL`-termination method to decide when to stop; it traverses the list, following the `forward` members, and stops after the `forward` member of the node it just processed was `NULL`. I think it's worth pointing out that this code works perfectly if the `head` that is passed in is a `NULL` pointer.

60 The line `head = head->forward` is there to skip past the bogus node at the head of the list before we start printing numbers.

70 If no appropriate bogus value had been available to distinguish the bogus node, we could have written `while(tail->back != NULL)` instead, so the possible lack of a bogus value is not a serious deficiency in the method we've used.

79 This is a quick hack on `getnum`; we should have added error checking and soforth. In particular, `getnum` should return `0` if it sees `EOF`, to signal end-of-input to `main`. As it is, our program goes into an infinite loop on `EOF`.

We stuck in this minimal `getnum` so we could get the program working and see if it had serious errors right away. This is perfectly legitimate. You should aim to get the program into a minimally working state as soon as possible, because that allows you to concentrate on major design decisions while postponing little details, such as `getnum`'s error handling, as long as possible.

A function like `getnum` that stands in for a better-developed version of the same function is called a *stub*.

44 Recursion

When I got this far in teaching the course, I made a list of topics I thought were important enough to cover, but which we somehow missed along the way. Most of the things on the list are miscellaneous trivia. Recursion, however, is powerful stuff, and you shouldn't get out of a programming class without seeing how it works.

Recursion is a programming methodology. The idea is that when we describe how to perform a certain task (which is what programming is), it is sometimes most natural to describe how to reduce the task to a simpler task of the same

type. Then you can reduce the simpler task to a still simpler task, and soforth, until finally the task is so simple it's trivial. Such methods are called *divide-and-conquer* methods or *recursive* methods.

44.1 The Factorial Function

The factorial function is a function from elementary mathematics. The function takes n nonnegative integer argument and yields a positive integer result. We write the factorial of n this way: $n!$.

The definition of the factorial function is:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$$

except when $n = 0$: $0! = 1$.

This definition yields a natural implementation in C:

```
long int fact(unsigned int n)
{
  long int total = 1;

  while (n>1)
    total *= n--;

  return total;
}
```

44.2 The Factorial Function

There's another way that mathematicians often define the factorial function:

$$x! = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot (x-1)! & \text{otherwise} \end{cases}$$

This definition is equivalent to the earlier definition, but it does a different thing: Instead of defining the factorial of a number explicitly, it defines the factorial in terms of a simpler factorial. To compute the simpler factorial we will need to compute the factorial of a number smaller still, and soforth, until finally we discover that all we need is the factorial of 0, which is defined to be 1.

This formulation also leads to a natural implementation in C:

```
long int fact(unsigned int n)
{

  if (n == 0)
    return 1L;
  else
    return n * fact(n-1);
}
```

And in fact this works. (The `1L` is a constant with value 1 and type `<long int>`.)

The factorial function needs to call itself, but that's not a problem. When a function calls another, the computer saves a description of the place to return to, and all the local variables and other information it will need in order to return safely, and jumps into the called function. The saved information is sometimes called an *environment*, but more often it's called a *frame*. The function call semantics that C adopts in order to allow this give C functions a property called *reentrancy*; a function can be called, called again before it returns the first time, can return, and then return again, and the returns will happen properly and will not interfere with one another.[42]

When the `fact` function calls itself, the same thing happens. All of `fact`'s local variables and state information are saved, and the computer calls `fact`, which means that a second invocation of `fact` is created, with its own local variables. The first `fact` is paused, and waits for the second `fact` to complete. If the second invocation of `fact` calls `fact` again, a second frame is created, with the second `fact`'s local variables saved in it; when the third `fact` returns, the second `fact` picks up where it left off.

There is no arbitrary limit on the number of such nested calls one can make. Each frame consumes a little memory, though, and so if you make to many calls without returning, you'll run out of memory and your program will fail. This rarely, if ever, happens in correct programs, because frames are not that big.

44.3 The Towers of Hanoi Problem

The factorial function was simple enough, but there was an obvious nonrecursive way to write it, and there's no reason to use recursion when an obvious iterative solution is available. In this section we'll see a problem that is difficult to solve iteratively, but trivial to solve recursively.

[42] Not all languages have reentrant functions. FORTRAN is the principal example. This is a serious deficiency because it means you can't write a recursive function in FORTRAN.

The puzzle is this: There are three vertical pegs, named $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$. There are some number, say 7, of circular discs, each of a different size, and each with a hole in the middle so that they can be placed on the pegs. Initially, all the discs are on peg $\mathcal{A}$, with the largest at the bottom, the next largest on top of that, and so on, with the smallest disc at the top.

The object of the puzzle is to transfer the entire tower to peg $\mathcal{C}$, subject to the following constraints:

1. You may only move one disc at a time.
2. You may never place a larger disc on top of a smaller disc.

After some thought, we arrive at a solution: To transfer a tower of n discs from $\mathcal{A}$ to $\mathcal{C}$, we can break the problem into three parts:

1. Transfer the smaller tower of $n-1$ discs from $\mathcal{A}$ to $\mathcal{B}$.
2. Move the nth disc, the largest, from $\mathcal{A}$ to $\mathcal{C}$.
3. Transfer the smaller tower of $n-1$ discs from $\mathcal{B}$ to $\mathcal{C}$.

In steps 1 and 3, we can ignore the large disc; it doesn't complicate matters at all. If we are performing these three steps in order to solve the problem with 7 discs, then steps 1 and 3 are just like the corresponding problem with only 6 discs.

How do we move a tower of 6 discs from one peg to another? Just apply the method again: Move a smaller tower of the top 5 discs out of the way, move the 6th disc to the destination, and move the smaller tower on top of it.

How do we move a tower of 5 discs? Just...

Eventually we'll reduce the problem to the trivial case, where we're asking how to move towers of 0 discs. How do we move a tower of 0 discs? Do nothing![43]

This suggests a straightforward implementation in C, which we'll see in section 45.

45 The Tower of Hanoi

As promised, here's the code for the Tower of Hanoi program.

[43] If this bothers you, then take one fewer step: We'll eventually reduce the problem to one of moving towers of only 1 disc from one pin to another; that's clearly trivial.

In the program, the three pegs are represented internally by the numbers 1, 2, and 3. As written, the program reads a number of rings from the user, and print out instructions to tell the user how to transfer a tower of this height from ring 1 to ring 3.

The workhorse function, `hanoi`, moves a tower of `num_rings` rings from peg `start_peg` to peg `end_peg`. The first thing it does is to figure out where the spare peg is; to do this it uses a simple trick: If pegs s and e are the start and end pegs, then peg $6 - s - e$ is the spare peg. (For example, if s is 2 and e is 1, the spare peg is $6 - 2 - 1$ or 3.)

If the height of the tower that `hanoi` is called upon to move has size 0, it returns immediately, because it doesn't need to do anything. Otherwise, it calls itself recursively to move the subtower of `num_rings` $-$ 1 rings from the start peg to the spare beg, calls `move` to move the largest ring from the start peg to the end peg, and then calls itself recursively again to move the subtower from the spare peg to the end peg.

```
void
  hanoi(int num_rings,
        int start_peg,
        int end_peg)
{
  int spare_peg = 6 - start_peg - end_peg;

  if (num_rings > 0) {
    hanoi(num_rings - 1, start_peg, spare_peg);
    move(num_rings, start_peg, end_peg);
    hanoi(num_rings - 1, spare_peg, end_peg);
  }

  return;
}
```

`move` is the function which is called each time we want to move a particular single ring. If we were writing our Tower of Hanoi program to do a fancy screen display which showed the rings flying around from peg to peg, we would put the code for drawing the rings on the screen in `move`. In this simple program, we'll be content to just print out an instruction about what peg should be moved where:

```
void move(int ring_num, int start, int end)
{
  printf("Move disk %d from peg %d onto peg %d.\n",
         ring_num, start, end);
```

```
  return;
}
```

We'll add a `main` which examines its command-line arguments to decide how many rings the user wants, and then just calls `hanoi` to print the instructions for moving a tower of that many rings from peg 1 to peg 3:

```
void hanoi(int num_rings, int start_peg, int end_peg);
void move(int ring_num, int start, int end);


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

  if (argc != 2) {
    fprintf(stderr, "Usage: %s number_of_rings\n", argv[0]);
    return 1;
  }

  num_rings = atoi(argv[1]);

  hanoi(num_rings, 1, 3);

  return 0;
}
```

The `atoi` function is something new: It accepts a string which is supposed to contain the string representation of an integer, and it returns the integer that the string represents. For example `atoi("119")` returns the integer 119. `atoi` returns 0 if there is an error, for example, because the string passed in was not composed of digits.

The program is almost trivial with recursion, but it would be very difficult to do without recursion, because the solution we have, to reduce the problem to a simpler case, is already organized along recursive lines.

46 Left-Over Language Features

> You taught me Language, and my profit on't
> is I know how to curse.
> — William Shakspere, *The Tempest*, I, ii.

This section will cover the last of the language features, things that are rarely or never important. Each of these I expected to bring up when the appropriate time came, at some moment when it would be useful, and the moment never came.

We'll see these features in increasing order of usefulness.

46.1 The , Operator

The , operator takes two operands, which must be expressions. To evaluate an expression of the form *expr1* , *expr2*, the computer evaluates *expr1*, and then evaluates *expr2*. The value of the entire expression is the value of *expr2*.

The , operator has two interesting properties: It guarantees that *expr1* will be evaluated first[44], and it guarantees a sequence point between the evaluation of the two expressions. This means that any side effects such as increments or assignments that are scheduled as a result of evaluating *expr1* will be completely resolved before *expr2* is evaluated.

Nevertheless, the , operator is mostly used when you want to stick two expressions where only one normally fits, such as in the condition of a `while` statement. The example that Kernighan and Ritchie give for the , operator is a program to reverse a string in place, which we've already seen: (They called it `reverse`.)

```
void strrev(char *s)
{
  int c, i, j;

  for (i=0, j=strlen(s)-1;  i<j;  i++, j++) {
    c = s[i];
    s[i] = s[j];
    s[j] = c;
  }
}
```

The commas that separate a function's arguments, the variables in a declaration, and soforth, are *not* , operators.

[44]Most operators, such as `+`, do not guarantee which of their operands will be evaluated first.

46.2 The `?:` Operator

The `?:` is unusual in that it has *three* operands. To evaluate the expression *expr1* `?` *expr2* `:` *expr3*, the computer first evaluates *expr1*. If *expr1* is true, the computer evaluates *expr2* and that value is the value of the entire expression. Otherwise, the computer evaluates *expr3* and that value is the value of the entire expression.

A `?:` expression is like a miniature `if–then`. Here's an example: This function prints out the elements of an array, separated by spaces, with ten elements per line.

```
void print_array(int *a, int nelems)
{
  int i;

  for (i=0; i<nelems; i++)
    printf("%d%c", a[i], (i+1)%10 ? ' ' : '\n');
}
```

Here's a simpler example: It sets the value of `a` to the value of either `x` or `y`, whichever is smaller:

```
a = (x > y) ? y : x ;
```

I swiped this example from K&R also.

46.3 The `switch` Statement

A `switch` *statement* has this form:

```
switch (expression) {
  case value1:
    statement11
    statement12
    ...
  case value2:
    statement21
    statement22
    ...
  ...
  default:
    statementd1
    statementd2
    ...
}
```

To execute a `switch` statement, the computer evaluates the *expression*. Control then passes to the statement immediately after one of the three following things, which are listed in order of preference:

1. A *case label* in the *statement*, which has the form

   `case` *constant* :

   where the value of the *constant* is the same as the value of the *expression*. If there is no case label whose value matches that of the *expression*, control passes to the statement immediately after:

2. A `default` *label* in the *statement*, which has the form

   `default` :

   . If there is no `default` *label*, control passes to the statement immediately after:

3. The end of the `case` statement.

Once into the compound statement part of the `switch`, control flows from top to bottom as usual. Control does *not* leave the compound statement except by flowing off the bottom of the statement or by encountering a `break` or `return` as usual. In particular, after the computer executes the last statement in one of the `case` sections, it continues on to the statements under the next `case` section. This usually isn't what we want, and so we almost always end each `case` section with a `break` statement.

It's so rare to actually allow control to fall through from one `case` section to the next that you should always put in a comment when you do let control fall through.

`case` can always be replaced by `if–else if–else`, but the restrictions on it allow the compiler to generate more efficient code, particularly if there are a lot of `case` labels.

46.4 Block-Scoped variables

We've only seen variables declared at the head of a function. These variables had names that were known only inside the function in which they were declared, and the variables were created each time the function was called and destroyed again when the function returned.

In fact, you can declare new variables at the head of *any* compound statement. The variables are created when control enters the statement, and destroyed again when control leaves the statement. The names of the variables are known only within the statement.

It's good to declare variables as belonging to as small a compound statement as possible, because it keeps the definition and use of the variable close together, because it restricts communication, and because it's easier to understand when you're reading the code—you know you can forget about the variable's value unless you're reading the code for the compound statement in which it's declared.

We've actually seen this before: I used it in the sample solution to the guessing game project.

46.5 true and false

We've been saying all semester that **false** was a zero value and that **true** was a nonzero value. That's only part of the story.

In any condition context, such as the condition part of a `while` or `for` loop, or the first operand of a `?:` expression, the following values mean **true** and **false**:

Type	**false**	**true**
number	0	non-zero
pointer	`NULL`	not `NULL`
character	`'\0'`	anything else

This means, in particular, that all those times we wrote `if (c != '\0')` we could have just written `if (c)`, and whenever we wrote `if (newnode != NULL)` we could just have written `if (newnode)` instead.

Opinion seems to be divided on whether or not this is bad style. Since opinion is divided, and about half the world does in fact use the short forms, it is therefore good style.

47 The Compiler and the Linker

The compiler is really two programs in one. The first part, the *compiler*, translates the C into whatever machine language is appropriate for the machine you're using. The translated version of the program is called an *object file*.

The second part of the compiler, the *linker*, does a simpler job. It looks through the object file for function calls, and it arranges that the right functions are called. When it's done, it might discover that you called some functions for which you didn't provide code, such as `printf`; these are called *unresolved references*. The linker then looks through *library files*, which contain pre-compiled functions, for functions whose names match the unresolved references. If it finds any, it includes the object code for those functions into your program. If there are still unresolved references, the linker complains that it couldn't finish compiling your program; otherwise it has resolved all the references and creates an executable program.

Having a linker means that you can write your program in several different files, compile them separately, months apart, have functions in one file call functions in another, and the linker will sort out all the calls at the end. If you change one function, you only need to recompile the file it's in and then re-link the program; you don't need to recompile the unchanged source files.

We didn't see how to do this because we never wrote a program long enough to deserve it.

48 Divide-and-Conquer Sorting

For a more serious application of recursion, consider the sorting problem again.

First observe that simple sorting algorithms, like insertion sort, run much faster on short lists than on long lists.

48.1 How Long Does insertion Sort Take?

Suppose we have a list of n things, and we want to find the smallest member. We have to scan over all the elements in the list, looking for the smallest one. The three fundamental operations we have to perform are looking at an element, comparing it to the current smallest, and copying the element into our 'current smallest' variable if it is smaller.

The number of times we have to perform the first two operations is clearly n. The number of times we have to perform the third operation is clearly no more than n. If all three operations take 1 unit of time,[45] then the running time of our algorithm to find the smallest element is no more than $3n$. It can be demonstrated that the number of times we can expect to have to perform the third step is about $(n+1)/2$, so the running time of the find-smallest-element operation averages about $(5n+1)/2$.

Thus if we double the length of the list, the time it takes to find the smallest element also doubles.

Now consider the insertion sort: We repeatedly find the smallest element, and copy it to an auxiliary array. To do this once on a list of n things takes $(5n+3)/2$ units of time ($(5n+1)/2$ for the search and 1 for the copy). We have to do it once for each element in the original list, so the total time to do an insertion sort on n things is about $n \cdot (5n+3)/2$ or $(5n^2+3n)/2$.

n	$(5n^2+3n)/2$
1	4
2	13
3	27
4	46
5	70
10	265
20	1030
40	4060
80	16120

It seems that if we double to length of the list, the time it takes to sort it approximately quadruples.

[45] This is usually true on conventional computers; however, the main conclusion of this section, that insertion sort takes time proportional to the square of the number of elements in the list, is still valid even if these three operations do not all take the same amount of time.

48.2 An Improvement to Insertion Sorting

Suppose we have a list of 20 things we want to sort. We can see from the table above that sorting them with insertion sort will take about 1,030 units of time.

Now suppose we broke the list of 20 things into two lists of 10 things. It would take 265 units of time to sort each of the two lists of 10 things, for a total of 530 units of time. If we would merge the two sorted lists together in less than 500 units of time, we'd have done the sort faster by splitting it into two pieces.

In fact, it's quick and easy to merge two sorted lists. You look at the first element in each list, find the smaller of the two, remove it from the list it's in, and copy it to an auxiliary list. This takes 4 units of time. Then you repeat the process until both lists are empty. If the two lists have a total of n things in them, you have to repeat this operation n times, so it takes $4n$ units of time. In the case above, n was 20, so it takes only 80 units of time to merge the two sorted lists of length 10.

That means the split-sort-and-merge technique has reduced the running time of our sort from 1030 units to 610 units.

If we were sorting 80 things instead, we would reduce the running time from 16120 units to 8440 ($4060 + 4060 + 320$), a savings of almost 50%. Clearly this is a substantial improvement.

48.3 Recursion

Now suppose we're sorting the 80 things. We break the list into two lists of 40 things each. We need to sort each list.

We've already seen that it's a substantial improvement to break a list in half, sort the parts separately, and merge them, rather than sorting the entire list. So let's take this sub-list of 40 elements and apply our improvement, by breaking it into two sub-sub-lists of 20 things each. To sort each list of 40 things by this method takes $1030 + 1030 + 160 = 2220$ units of time, for a total of $2220 + 2220 + 320 = 4760$ units of time, down from 16120.

But we already saw that by using the improved method to sort a list of 20 things we could get a speed up, so let's apply the improvement yet again, breaking the lists of 20 things into two lists of 10 things each. This reduces the running time of our sort to 3080 units. The improvement is less this time, because the savings we got by sorting lists of 10 things instead of 20 things is not so big, and because the cost of merging the lists back together an extra time is larger in proportion.

Nevertheless if we sort the lists of 10 things by breaking them each into two lists of 5 things and doing insertion sort on those lists and merging the results, we can sort our original 80 objects in 2400 time units, instead of the 16120 that our original straight insertion sort yielded.

So here's our improved sorting algorithm:

1. If the list to sort is very short, use insertion sort.
2. Otherwise, break the list into two lists of approximately equal size, sort each list with *this* algorithm, and merge the two sorted lists back together.

This algorithm is called *mergesort*. We won't see a C implementation, because to do it properly you get bogged down in a lot of little details about how to store the objects, and allocating auxiliary space, and handling odd-length lists which can't be broken evenly, and soforth. But it's clear that the algorithm will be recursive. Somewhere in our program we will have a function something like this:

```
void mergesort(struct list *data, int length)
{
  ...
  if (length < SMALL_SIZE )
    insertion_sort(data, length);
  else {
    split_list(data, top, bottom);
    mergesort(top,    length/2);
    mergesort(bottom, length/2);
    merge_lists(top, bottom, data);
    return;
  }
}
```

`mergesort` does a little preparatory work, calls itself do to the bulk of the work on simpler cases, and does a little cleanup work.

49 Optimization and Performance

By doing a careful analysis and by using a better algorithm, we were able to improve the speed of a sorting program enormously. The gain for a short list, of 80 things, was to cut the running time of the sort by 85%. The improvement

would be even greater for longer lists; for a 500-element list our mergesort would take about $\frac{1}{30}$ the time that the insertion sort would, instead of $\frac{1}{7}$.

This demonstration suggests a number of things: First, that program performance is largely determined by the overall efficiency of the algorithm the program uses, and much less so by micro details of the code.

Therefore, concern about whether you are doing one extra test or not each time through your add-a-node-to-a-list function is misplaced. Such considerations are usually dominated by more important matters.

Mergesort uses a clever algorithm and sorts n object is time that is approximately proportional to $n \cdot logn$; straight insertion sort sorts n objects in time that is approximately proportional to n^2. Changing a sort program to use a fast sort like mergesort rather than a slow sort like straight insertion sort will improve performance more than fussing around with extra tests.

There's a saying in the business that there are two rules for when to optimize:[46]

1. Don't do it.
2. (For experts only) Don't do it yet.

This is good advice. Write the code itself to be a straightforward implementation of a good algorithm, and write it to be clear to humans and easy to maintain, rather than to avoid a couple of unnecessary tests.

If, *after* the program is written, it actually turns out to be 'too slow' for some practical use, then optimize first by considering obvious waste in the program and by researching better algorithms.

If, after due consideration and research, you decide that no faster suitable algorithm is available, then micro-optimization may be appropriate. (Then again it may not.) There are tools available on most platforms that will analyze a run of your program and tell you where the program is spending most of its time. Then you can micro-optimize these parts. There's a rule of thumb called the '90-10' rule, which is that the program spends about 90% of its running time executing about 10% of the program, and the other 90% of the program is initializations and special cases that get executed rarely or only once.

Computer time is cheap these days, and if your program runs a little slow, that is probably all right. On the other hand programmer time is expensive, and if someone has to spend a lot of time figuring out your code because you made it obscure in an effort to save a few microseconds, then you've made a bad trade.

[46] The Berkeley UNIX fortune file attributes this to 'Michael Jackson'.

The summary is: Write for style, not for efficiency, because style is probably more important than you think it is. and efficiency is probably less important than you think it is. When you direct your attentions toward efficiency considerations, you should do so in a way that is likely to yield the most results: Improve the algorithm first, then, if you must, do real research to find out what parts of your code are actually slowing down the program, and fix those and nothing else.