Appendix A

A Crash Course in C

This appendix gives a brief introduction to C for beginners who have some programming experience in a high-level language such as MATLAB. It is not intended as a complete reference; there are lots of great C resources and references out there for a more complete picture. This appendix is also not specific to the Microchip PIC. In fact, I recommend that you start by programming your laptop or desktop so you can experiment with C without needing extra equipment like a PIC32 board.

A.1 Quick Start in C

To get started with C, you need three things: a desktop or laptop, a text editor, and a C compiler. You use the text editor to create your C program, which is a plain text file with a name ending with the postfix `.c`, such as `myprog.c`. Then you use the C compiler to convert this program into machine code that your computer can execute. There are many free C compilers available. I recommend the `gcc` C compiler, which is part of the free GNU Compiler Collection (GCC, found at `http://gcc.gnu.org`). GCC is available for Windows, Mac OS, and Linux. For Windows, you can download the GCC collection in MinGW.[1] (If the installation asks you about what tools to install, make sure to include the `make` tools.) For Mac OS, you can download the full Xcode environment from the Apple Developers site. This installation is multiple gigabytes, however; you can instead opt to install only the "Command Line Tools for Xcode," which is much smaller and more than sufficient for getting started with C (and for this appendix).

Many C installations come with an Integrated Development Environment (IDE) complete with text editor, menus, graphical tools, and other things to assist you with your programming projects. Each IDE is different, however, and the things we cover in this appendix do not require a sophisticated IDE. Therefore we will use only *command line tools*, meaning that we initiate the compilation of the program, and run the program, by typing at the command line. In Mac OS, the command line can be accessed from the Terminal program. In Windows, you can access the command line (also known as MS-DOS or Disk Operating System) by searching for `cmd` or `command prompt`.

To work from the command line, it is useful to learn a few command line instructions. The Mac operating system is built on top of Unix, which is almost identical to Linux, so Mac/Unix/Linux use the same syntax. Windows is similar but slightly different. See the table of a few useful commands below. You can find more information online on how to use these commands as well as others by searching for command line commands in Unix, Linux, or DOS (disk operating system, for Windows).

[1] You are also welcome to use Visual C from Microsoft. The command line compile command will look a bit different than what you see in this appendix.

function	Mac/Unix/Linux	Windows
show current directory	pwd	cd
list directory contents	ls	dir
make subdirectory newdir	mkdir newdir	mkdir newdir
change to subdirectory newdir	cd newdir	cd newdir
move "up" to parent directory	cd ..	cd ..
copy file to filenew	cp file filenew	copy file filenew
delete file file	rm file	del file
delete directory dir	rmdir dir	rmdir dir
help on using command cmd	man cmd	cmd ?

Following the long-established programming tradition, your first C program will simply print "Hello world!" to the screen. Use your text editor to create the file HelloWorld.c:

```
#include <stdio.h>
int main(void) {
  printf("Hello world!\n");
  return(0);
}
```

Your text editor could be Notepad in Windows or TextEdit on the Mac. You could even use Microsoft Word if you insisted. I personally prefer emacs, but it's not easy to get started with! Text editors packaged with IDEs help enforce a consistent look to your programs. Whichever editor you use, you should save your file as plain text, not rich text or any other formatted text.

To compile your program, navigate from the command line to the directory where the program sits. Then, assuming your command prompt appears as >, type the following at the prompt:

```
> gcc HelloWorld.c -o HelloWorld
```

This should create the executable file HelloWorld in the same directory. (The argument after the -o output flag is the name of the executable file to be created from HelloWorld.c.) Now to execute the program, type

```
> HelloWorld
```

The response should be

```
Hello world!
>
```

If the response is instead command not found or similar, your computer didn't know where to look for the executable HelloWorld. On Mac/Unix/Linux, you can type

```
> ./HelloWorld
```

where the "." is shorthand for "current directory," telling your computer to look in the current directory for HelloWorld.

If you've succeeded in getting this far, you have a working C installation and you are ready for the rest of this appendix. If not, time to get help from friends or the web.

A.2 Overview

If you are familiar with a high-level language like MATLAB, you have some idea of loops, functions, program modularity, etc. You'll see that C syntax is different, but that's not a big deal. Let's start instead by focusing on important concepts you must master in C which you don't have to worry about in MATLAB:

- **Memory, addresses, and pointers.** A variable is stored at a particular *address* in memory as 0's and 1's. In C, unlike MATLAB, it is often useful to have access to the memory address where a variable is located. We will learn how to generate a *pointer* to a variable, which contains the address of the variable, and how to access the contents of an address, i.e., the *pointee*.

- **Data types.** In MATLAB, you can simply type `a = 1; b = [1.2 3.1416]; c = [1 2; 3 4]; s = 'a string'`. MATLAB figures out that `a` is a scalar, `b` is a vector with two elements, `c` is a 2×2 matrix, and `s` is a string, and automatically keeps track of the type of the variable (e.g., a list of numbers for a vector or a list of characters for a string) and sets aside, or *allocates*, enough memory to store them. In C, on the other hand, you have to first *define* the variable before you ever use it. For a vector, for example, you have to say what *data type* the elements of the vector will be—integers or numbers with a decimal point (floating point)—and how long the vector will be. This allows the C compiler to allocate enough memory to hold the vector, and to know that the binary representations (0's and 1's) at those locations in memory should be interpreted as integers or floating point numbers.

- **Compiling.** MATLAB programs are typically run as *interpreted* programs—the commands are interpreted, converted to machine-level code, and executed while the program is running. C programs, on the other hand, are *compiled* in advance. This process consists of several steps, but the point is to turn your C program into machine-executable code before the program is ever run. The compiler can identify some errors and warn you about other questionable code. Compiled code typically runs faster than interpreted code, since the translation to machine code is done in advance.

Each of these concepts is described in Section A.3 without going into detail on C syntax. In Section A.4 we will look at sample programs to introduce the syntax, then follow up with a more detailed explanation of the syntax.

A.3 Important Concepts in C

We begin our discussion of C with this caveat:

Important! C consists of an evolving set of standards for a programming language, and any specific C installation is an "implementation" of C. While C standards require certain behavior from all implementations, a number of details are left as implementation-dependent. For example, the number of bytes used for some data types is not fully standard. C wonks like to point out when certain behavior is required and when it is implementation-dependent. While it is good to know that differences may exist from one implementation to another, in this appendix I will often blur the line between what is required and what is common. I prefer to keep this introduction succinct instead of overly precise.

A.3.1 Data Types

Binary and hexadecimal. On a computer, programs and data are represented by sequences of 0's and 1's. A 0 or 1 may be represented by two different voltage levels (low and high) held by a capacitor and controlled by a transistor, for example. Each of these units of memory is called a **bit**.

A sequence of 0's and 1's may be interpreted as a base-2 or **binary** number, just as a sequence of digits in the range 0 to 9 is commonly treated as a base-10 or **decimal** number. In the decimal numbering system, a multi-digit number like 793 is interpreted as $7 * 10^2 + 9 * 10^1 + 3 * 10^0$; the rightmost column is the 10^0 (or 1's) column, the next column to the left is the 10^1 (or 10's) column, the next column to the left is the 10^2 (or 100's) column, and so on. Similarly, the rightmost column of a binary number is the 2^0 column, the next column to the left is the 2^1 column, etc. Converting the binary number `00111011` to its decimal representation, we get

$$0 * 2^7 + 0 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 32 + 16 + 8 + 2 + 1 = 59.$$

We can clarify that a sequence of digits is base-2 by writing it as $\texttt{00111011}_2$ or `0b00111011`, where the `b` stands for "binary."

To convert a base-10 number x to binary:

1. Initialize the binary result to all zeros and k to a large integer, such that 2^k is known to be larger than x.

2. If $2^k \le x$, place a 1 in the 2^k column of the binary number and set x to $x - 2^k$.
3. If $x = 0$ or $k = 0$, we're finished. Else set k to $k - 1$ and go to line 2.

The leftmost digit in a multi-digit number is called the **most significant digit**, and the rightmost digit, corresponding to the 1's column, is called the **least significant digit**. For binary representations, these are often called the **most significant bit (msb)** and **least significant bit (lsb)**, respectively.

Compared to base-10, base-2 has a more obvious connection to the actual hardware representation. Binary can be inconvenient for human reading and writing, however, due to the large number of digits. Therefore we often use base-16, or **hexadecimal** (hex), representations. A single hex digit represents four binary digits using the numbers 0..9 and the letters A..F:

base-2	base-16	base-10	base-2	base-16	base-10
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

Thus we can write the eight-digit binary number `00111011`, or `0011 1011`, more succinctly in hex as `3B`, or $\texttt{3B}_{16}$ or `0x3B` to clarify that it is a hex number. The corresponding decimal number is $3 * 16^1 + 11 * 16^0 = 59$.

Bits, bytes, and data types. Bits of memory are grouped together in groups of eight called **bytes**. A byte can be written equivalently in binary or hex (e.g., `00111011` or `3B`), and can represent values between 0 and $2^8 - 1 = 255$ in base-10. Sometimes the four bits represented by a single hex digit are referred to as a **nibble**. (Get it?)

A **word** is a grouping of multiple bytes. The number of bytes depends on the processor, but four-byte words are common, as with the PIC32. A word `01001101 11111010 10000011 11000111` in binary can be written in hex as `4DFA83C7`. The msb is the leftmost bit of the leftmost byte, a `0` in this case.

A byte is the smallest unit of memory that has its own **address**. The address of the byte is a number that represents where the byte is in memory. Suppose your computer has 4 gigabytes (GB)[2], or $4 \times 2^{30} = 2^{32}$ bytes, of RAM. Then to find the value stored in a particular byte, you need at least 32 binary digits (8 hex digits or 4 bytes) to specify the address.

An example showing the first eight addresses in memory is shown below.

...	7	6	5	4	3	2	1	0	address
	11001101	00100111	01110001	01010111	01010011	00011110	10111011	01100010	value

Now assume that the byte at address 4 is part of the representation of a variable. Do these 0's and 1's represent an integer, or part of an integer? A number with a fractional component? Something else?

The answer lies in the **data type** of the variable at that address. In C, before you use a variable, you have to *define* it and its type. This tells the compiler how many bytes to set aside for the variable and how to write or interpret 0's and 1's at the address(es) used by that variable. The most common data types come in two flavors: integers and floating point numbers (numbers with a decimal point). Of the integers, the two most common types are `char`[3], often used to represent keyboard characters, and `int`. Of the floating point numbers, the two most common types are `float` and `double`. As we will see shortly, a `char` uses 1 byte and an `int` usually uses 4, so two possible interpretations of the data held in the eight memory addresses could be

[2]In common usage, a kilobyte (KB) is $2^{10} = 1024$ bytes, a megabyte (MB) is $2^{20} = 1,048,576$ bytes, a gigabyte is $2^{30} = 1,073,741,824$ bytes, and a terabyte (TB) is $2^{40} = 1,099,511,627,776$ bytes. To remove confusion with the common SI prefixes that use powers of 10 instead of powers of 2, these are sometimes referred to instead as kibibyte, mebibyte, gibibyte, and tebibyte, where the "bi" refers to "binary."

[3]`char` is derived from the word "character." People pronounce `char` variously as "car" (as in "driving the car"), "care" (a shortening of "character"), and "char" (as in charcoal), and some just punt and say "character." Up to you.

...	7	6	5	4	3	2	1	0	address
	11001101	00100111	01110001	01010111	01010011	00011110	10111011	01100010	value
	int							char	

where byte 0 is used to represent a `char` and bytes 4-7 are used to represent an `int`, or

...	7	6	5	4	3	2	1	0	address
	11001101	00100111	01110001	01010111	01010011	00011110	10111011	01100010	value
				char		int			

where bytes 0-3 are used to represent an `int` and byte 4 represents a `char`. Fortunately we don't usually have to worry about how variables are packed into memory.

Below are descriptions of the common data types. While the number of bytes used for each type is not the same for every processor, the numbers given are common. (Differences for the PIC32 are noted in Table A.1.) Example syntax for defining variables is also given. Note that most C statements end with a semicolon.

`char`

Example definition:

```
char ch;
```

This syntax defines a variable named `ch` to be of type `char`. `char`s are the smallest common data type, using only one byte. They are often used to represent keyboard characters. You can do a web search for "ASCII table" (pronounced "ask-key") to find the American Standard Code for Information Interchange, which maps the values 0 to 127 to keyboard characters and other things. (The values 128 to 255 may map to an "extended" ASCII table.) For example, the values 48 to 57 map to the characters '0' to '9', 65 to 90 map to the uppercase letters 'A' to 'Z', and 97 to 122 map to the lowercase letters 'a' to 'z'. The assignments

```
ch = 'a';
```

and

```
ch = 97;
```

are equivalent, as C equates characters inside single quotes to their ASCII table numerical value.

Depending on the C implementation, `char` may be treated by default as `unsigned`, i.e., taking values from 0 to 255, or `signed`, taking values from -128 to 127. If you plan to use the `char` to represent a standard ASCII character, you don't need to worry about this. If you plan to use the `char` data type for integer math on small integers, however, you may want to use the specifier `signed` or `unsigned`, as appropriate. For example, we could use the following definitions, where everything after `//` is a comment:

```
unsigned char ch1; // ch1 can take values 0 to 255
signed char ch2;   // ch2 can take values -128 to 127
```

`int` (also known as `signed int` or `signed`)

Example definition:

```
int i,j;
signed int k;
signed n;
```

`int`s are typically four bytes (32 bits) long, taking values from $-(2^{31})$ to $2^{31}-1$ (approximately ± 2 billion). In the example syntax, each of `i`, `j`, `k`, and `n` are defined to be the same data type.

We can use specifiers to get the following integer data types: `unsigned int` or simply `unsigned`, a four-byte integer taking nonnegative values from 0 to $2^{32}-1$; `short int`, `short`, `signed short`, or `signed`

type	# bytes on my laptop	# bytes on PIC32
char	1	1
short int	2	2
int	4	4
long int	8	4
long long int	8	8
float	4	4
double	8	4
long double	16	8

Table A.1: Data type sizes on two different machines.

`short int`, a two-byte integer taking values from $-(2^{15})$ to $2^{15}-1$ (i.e., $-32,768$ to $32,767$); `unsigned short int` or `unsigned short`, a two-byte integer taking nonnegative values from 0 to $2^{16}-1$ (i.e., 0 to $65,535$); `long int`, `long`, `signed long`, or `signed long int`, often consisting of eight bytes and representing values from $-(2^{63})$ to $2^{63}-1$; and `unsigned long int` or `unsigned long`, an eight-byte integer taking nonnegative values from 0 to $2^{64}-1$. A `long long int` data type may also be available.

```
float
```

Example definition:

```
float x;
```

This syntax defines the variable `x` to be a four-byte "single-precision" floating point number.

```
double
```

Example definition:

```
double x;
```

This syntax defines the variable `x` to be an eight-byte "double-precision" floating point number. The data type `long double` (quadruple precision) may also be available, using 16 bytes (128 bits). These types allow the representation of larger numbers, to more decimal places, than single-precision `float`s.

The sizes of the data types, both on my laptop and the PIC32, are summarized in Table A.1. Note the differences; C does not enforce a strict standard.

Using the data types. If your program calls for floating point calculations, you can choose between `float`, `double`, and `long double` data types. The advantages of smaller types are that they use less memory and computations with them (e.g., multiplies, square roots, etc.) may be faster. The advantage of the larger types is the greater precision in the representation (e.g., smaller roundoff error).

If your program calls for integer calculations, you are better off using integer data types than floating point data types due to the higher speed of integer math and the ability to represent a larger range of integers for the same number of bytes.[4] You can decide whether to use `signed` or `unsigned char`s, or {`signed/unsigned`} {`short/long`} `int`s. The considerations are memory usage, possibly the time of the computations[5], and whether or not the type can represent a sufficient range of integer values. For example, if you decide to use `unsigned char`s for integer math to save on memory, and you add two of them with values 100 and 240 and assign to a third `unsigned char`, you will get a result of 84 due to *integer overflow*. This example is illustrated in the program `overflow.c` in Section A.4.

As we will see shortly, functions have data types, just like variables. For example, a function that calculates

[4] Just as a four-byte `float` can represent fractional values that a four-byte `int` cannot, a four-byte `int` can represent more integers than a four-byte `float` can. See the type conversion example program `typecast.c` in Section A.4 for an example.

[5] Computations with smaller data types are not always faster than with larger data types. It depends on the architecture.

the sum of two `double`s and returns a `double` should be defined as type `double`. Functions that don't return a value are defined of type `void`.

Representations of data types. A simple representation for integers is the *sign and magnitude* representation. In this representation, the msb represents the sign of the number (0 = positive, 1 = negative), and the remaining bits represent the magnitude of the number. The sign and magnitude method represents zero twice (positive and negative zero) and is not often used.

A much more common representation for integers is called *two's complement*. This method also uses the msb as a sign bit, but it only has a single representation of zero. The two's complement representation of an 8-bit `char` is given below:

binary	`signed char`, base-10	`unsigned char`, base-10
00000000	0	0
00000001	1	1
00000010	2	2
00000011	3	3
⋮		
01111111	127	127
10000000	−128	128
10000001	−127	129
⋮		
11111111	−1	255

As the binary representation is incremented, the two's complement (signed) interpretation of the binary number also increments, until it "wraps around" to the most negative value when the msb becomes 1 and all other bits are 0. The signed value then resumes incrementing until it reaches −1 when all bits are 1.

Another representation choice is *endianness*. The *little-endian* representation of an `int` stores the least significant byte at `ADDRESS` and the most significant byte at `ADDRESS+3`, while the *big-endian* convention is the opposite.[6] The convention used depends on the processor. For definiteness in this appendix, we will use the little-endian representation, which is also used by the PIC32.

`float`s, `double`s, and `long double`s are commonly represented in the IEEE 754 floating point format $(-1)^s * b * 2^c$, where one bit is used to represent the sign ($s = 0$ or 1); $m = 23/52/112$ bits are used to represent the significand b in the range 1 to $2 - 2^{-m}$; and $n = 8/11/15$ bits are used to represent the exponent c in the range $-(2^{n-1}) + 2$ to $2^{n-1} - 1$, where n and m depend on whether the type uses 4/8/16 bytes. Certain exponent and significand combinations are reserved for representing zero, positive and negative infinity, and "not a number" (`NaN`).

It is rare that you need to worry about the specific bit-level representation of the different data types: endianness, two's complement, IEEE 754, etc. You tell the compiler to store values and retrieve values, and it takes care of implementing the representations.

A.3.2 Memory, Addresses, and Pointers

Consider the following C syntax:

```
int i;
int *ip;
```

or equivalently

```
int i, *ip;
```

[6] These phrases come from *Gulliver's Travels*, where Lilliputians fanatically divide themselves according to which end of a soft-boiled egg they crack open.

These definitions appear to define the variables `i` and `*ip` of type `int`. The character `*` is not allowed as part of a variable name, however. The variable name is actually `ip`, and the special character `*` means that `ip` is a **pointer** to something of type `int`. The purpose of a pointer is to hold the address of a variable it "points" to. I often use the words "address" and "pointer" interchangeably.

When the compiler sees the definition `int i`, it allocates four bytes of memory to hold the integer `i`. When the compiler sees the definition `int *ip`, it creates the variable `ip` and allocates to it whatever amount of memory is needed to hold an address. The compiler also remembers the data type that `ip` points to, `int` in this case, so if later you use `ip` in a context that requires a pointer to a different variable type, the compiler will generate a warning or an error. Technically, the type of `ip` is "pointer to type `int`."

Important! Defining a pointer only allocates memory to hold the pointer. It does **not** allocate memory for a pointee variable to be pointed at. Also, simply defining a pointer does not initialize it to point to anything valid.

When we have a variable and we want the address of it, we apply the **reference operator** to the variable, which returns a "reference" (i.e., a pointer to the variable, or the address). In C, the reference operator is written `&`. Thus the following command makes sense:

```
ip = &i;  // ip now holds the address of i
```

The reference operator always returns the lowest address of a multi-byte type. For example, if the four-byte `int i` occupies addresses 0x0004 to 0x0007 in memory, `&i` will return 0x0004.[7]

If we have a pointer (an address) and we want the contents at that address, we apply the **dereference operator** to the pointer. In C, the dereference operator is written `*`. Thus the following command makes sense:

```
i = *ip;  // i now holds the contents at the address ip
```

However, you should never dereference a pointer until it has been initialized to point at something using a statement such as `ip = &i`.

As an analogy, consider the pages of a book. A page number can be considered a pointer, while the text on the page can be considered the contents of a variable. So the notation `&text` would return the page number (pointer or address) of the text, while `*page_number` would return the text on that page (but only after `page_number` is initialized to point at a page of text).

Even though we are focusing on the concept of pointers, and not C syntax, let's go ahead and look at some sample C code, remembering that everything after `//` on the same line is a comment:

```
int i,j,*ip; // define i, j as type int, as well as ip as type "pointer to type int"
ip = &i;     // set ip to the address of i (& references i)
i = 100;     // put the value 100 in the location allocated by the compiler for i
j = *ip;     // set j to the contents of the address ip (* dereferences ip), i.e., 100
j = j+2;     // add 2 to j, making j equal to 102
i = *(&j);   // & references j to get the address, then * gets contents; i is set to 102
*(&j) = 200; // content of the address of j (j itself) is set to 200; i is unchanged
```

The use of pointers can be powerful, but also dangerous. For example, you may accidentally try to access an illegal memory location. The compiler is unlikely to recognize this during compilation, and you may end up with a "segmentation fault" when you execute the code.[8] This kind of bug can be difficult to track down, and dealing with it is a C rite of passage. More on pointers in Section A.4.8.

A.3.3 Compiling

The process loosely referred to as "compilation" actually consists of four steps:

[7]This is the right way to think about it conceptually, but in fact the computer may automatically translate the value of `&i` to an actual physical address.

[8]A good name for a program like this is `coredumper.c`.

1. **Preprocessing.** The preprocessor takes the `program.c` source code and produces an equivalent `.c` source code, performing operations such as stripping out comments. The preprocessor is discussed in more detail in Section A.4.3.

2. **Compiling.** The compiler turns the preprocessed code into *assembly* code for the specific processor. This process converts the code from standard C syntax into a set of commands that can be understood natively by the processor. The compiler can be configured with a number of options that impact the assembly code generated. For example, the compiler can be instructed to generate assembly code that trades off time of execution with the amount of memory needed to store the code. Assembly code generated by a compiler can be inspected with a standard text editor. In fact, coding directly in assembly is still a popular, if painful, way to program microcontrollers.

3. **Assembling.** The assembler takes the assembly code and produces processor-dependent machine-level binary *object* code. This code cannot be examined using a text editor. Object code is called *relocatable*, in that the exact memory addresses for the data and program statements are not specified.

4. **Linking.** The linker takes one or more object codes and produces a single executable file. For example, if your code includes pre-compiled libraries, such as printout functions in the `stdio` library (described in Sections A.4.3 and A.4.15), this code is included in the final executable. The data and program statements in the various object codes are assigned to specific memory locations.

In our `HelloWorld.c` program, this entire process is initiated by the single command line statement

```
> gcc HelloWorld.c -o HelloWorld
```

If our `HelloWorld.c` program used any mathematical functions in Section A.4.7, the compilation would be initiated by

```
> gcc HelloWorld.c -o HelloWorld -lm
```

where the `-lm` flag tells the linker to link the math library, which may not be linked by default like other libraries are.

If you want to see the intermediate results of the preprocessing, compiling, and assembling steps, Problem 40 gives an example.

For more complex projects requiring compilation of several files into a single executable or specifying various options to the compiler, it is common to create a `makefile` that specifies how the compilation is to be done, and to then use the command `make` to actually create the executable. The use of `makefile`s is beyond the scope of this appendix. Section A.4.16 gives a simple example of compiling multiple C files to make a single executable program.

A.4 C Syntax

So far we have seen only glimpses of C syntax. Let's begin our study of C syntax with a few simple programs. We will then jump to a more complex program, `invest.c`, that demonstrates many of the major elements of C structure and syntax. If you can understand `invest.c` and can create programs using similar elements, you are well on your way to mastering C. We will defer the more detailed descriptions of the syntax until after introducing `invest.c`.

Printing to screen. Because it is the simplest way to see the results of a program, as well as the most useful tool for debugging, let's start with the function `printf` for printing to the screen. We have already seen it in `HelloWorld.c`. Here's a slightly more interesting example. Let's call this program file `printout.c`.

```
#include <stdio.h>

int main(void) {
```

```
  int i; float f; double d; char c;

  i = 32; f = 4.278; d = 4.278; c = 'k'; // or, by ASCII table, c = 107;
  printf("Formatted output:\n");
  printf(" i = %4d  c = '%c'\n",i,c);
  printf(" f = %19.17f\n",f);
  printf(" d = %19.17lf\n",d);
  return(0);
}
```

The `17lf` in the last `printf` statement is "seventeen ell eff."

The first line of the program

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

tells the preprocessor that the program will use functions from the "standard input and output" library, one of many code libraries provided in standard C installations that extend the power of the language. The `stdio.h` function used in `printout.c` is `printf`, covered in more detail in Section A.4.15.

The next line

```
int main(void) {
```

starts the block of code that defines the `main` function. The `main` code block is closed by the final closing brace }. Each C program has exactly one `main` function. The type of `main` is `int`, meaning that the function should end by returning a value of type `int`. In our case, it returns a `0`, which indicates that the program has terminated successfully.

The next line defines and allocates memory for four variables of four different types, while the line after assigns values to those variables. The `printf` lines will be discussed after we look at the output.

Now that you have created `printout.c`, you can create the executable file `printout` and run it from the command line. Make sure you are in the directory containing `printout.c`, then type the following:

```
> gcc printout.c -o printout
> printout
```

(Again, you may have to use `./printout` to tell your computer to look in the current directory.) On my laptop, here is the output:

```
Formatted output:
  i =   32  c = 'k'
  f = 4.27799987792968750
  d = 4.27799999999999958
```

The main point of this program is to demonstrate formatted output from the code

```
  printf("Formatted output:\n");
  printf(" i = %4d  c = '%c'\n",i,c);
  printf(" f = %19.17f\n",f);
  printf(" d = %19.17lf\n",d);
```

Inside a `printf` statement, everything inside the double quotes is printed to the screen, but some character sequences have special meaning. The `\n` sequence creates a newline (carriage return). The `%` is a special character, indicating that some data will be printed, and for each `%` in the double quotes, there must be a variable or other expression in the comma-separated list at the end of the `printf` statement. The `%4d` means that an `int` type variable is expected, and it will be displayed right-justified using 4 spaces. (If the number is more than 4 digits, it will take as much space as is needed.) The `%c` means that a `char` is expected. The `%19.17f` means that a `float` will be printed right-justified over 19 spaces with 17 spaces after the decimal point. The `%19.17lf` means that a `double` (or "long float") will be printed right-justified over 19 spaces, with 17 after the decimal point. More details on `printf` can be found in Section A.4.15.

The output of the program also shows that neither the `float f` nor the `double d` can represent 4.278 exactly, though the double-precision representation comes closer.

Data sizes. Since we have focused on data types, our next program measures how much memory is used by different data types. Create a file called `datasizes.c` that looks like the following:

```
#include <stdio.h>

int main(void) {
  char a, *bp; short c; int d; long e;
  float f; double g; long double h, *ip;

  printf("Size of char:                    %2ld bytes\n",sizeof(a));// "% 2 ell d"
  printf("Size of char pointer:            %2ld bytes\n",sizeof(bp));
  printf("Size of short int:               %2ld bytes\n",sizeof(c));
  printf("Size of int:                     %2ld bytes\n",sizeof(d));
  printf("Size of long int:                %2ld bytes\n",sizeof(e));
  printf("Size of float:                   %2ld bytes\n",sizeof(f));
  printf("Size of double:                  %2ld bytes\n",sizeof(g));
  printf("Size of long double:             %2ld bytes\n",sizeof(h));
  printf("Size of long double pointer:     %2ld bytes\n",sizeof(ip));
  return(0);
}
```

The first two lines in the `main` function define nine variables, telling the compiler to allocate space for these variables. Two of these variables are pointers. The `sizeof()` operator returns the number of bytes allocated in memory for its argument.

Here is the output of the program:

```
Size of char:                1 bytes
Size of char pointer:        8 bytes
Size of short int:           2 bytes
Size of int:                 4 bytes
Size of long int:            8 bytes
Size of float:               4 bytes
Size of double:              8 bytes
Size of long double:        16 bytes
Size of long double pointer:  8 bytes
```

We see that, on my laptop, `int`s and `float`s use 4 bytes, `short int`s 2 bytes, `long int`s and `double`s 8 bytes, and `long double`s 16 bytes. Regardless of whether it points to a `char` or a `long double`, a pointer (address) uses 8 bytes, meaning we can address a maximum of $(2^8)^8 = 256^8$ bytes of memory. Considering that corresponds to almost 18 quintillion bytes, or 18 billion gigabytes, we should have enough available addresses for a laptop!

Overflow. Now let's try the program `overflow.c`, which demonstrates the issue of integer overflow mentioned in Section A.3.1.

```
#include <stdio.h>

int main(void) {
  char i = 100, j = 240, sum;
  unsigned char iu = 100, ju = 240, sumu;
  signed char is = 100, js = 240, sums;

  sum = i+j; sumu = iu+ju; sums = is+js;
  printf("char:          %d + %d = %3d or ASCII %c\n",i,j,sum,sum);
  printf("unsigned char: %d + %d = %3d or ASCII %c\n",iu,ju,sumu,sumu);
  printf("signed char:   %d + %d = %3d or ASCII %c\n",is,js,sums,sums);
  return(0);
}
```

In this program we initialize the values of some of the variables when they are defined. You might also notice that we are assigning a `signed char` a value of 240, even though the range for that data type is -128 to 127. So something fishy is going on. When I compile and run the program, I get the output

```
char:           100 + -16 =  84 or ASCII T
unsigned char:  100 + 240 =  84 or ASCII T
signed char:    100 + -16 =  84 or ASCII T
```

One thing we notice is that, with my C compiler at least, `char`s are the same as `signed char`s. Another thing is that even though we assigned the value of 240 to `js` and `j`, they contain the value -16. This is because the binary representation of 240 has a 1 in the 2^7 column, but for the two's complement representation of a `signed char`, this column indicates whether the value is positive or negative. Finally, we notice that the `unsigned char ju` is successfully assigned the value 240 (since its range is 0 to 255), but the addition of `iu` and `ju` leads to an overflow. The correct sum, 340, has a 1 in the 2^8 (or 256) column, but this column is not included in the 8 bits of the `unsigned char`. Therefore we see only the remainder of the number, 84. The number 84 is assigned the character T in the standard ASCII table.

Type conversion. Continuing our focus on the importance of understanding data types, we try one more simple program that illustrates what can happen when you mix data types in a mathematical expression. This is also our first program that uses a helper function beyond the `main` function. Call this program `typecast.c`.

```
#include <stdio.h>

void printRatio(int numer, int denom) {
  double ratio;

  ratio = numer/denom;
  printf("Ratio, %1d/%1d:                          %5.2f\n",numer,denom,ratio);
  ratio = numer/((double) denom);
  printf("Ratio, %1d/((double) %1d):               %5.2f\n",numer,denom,ratio);
  ratio = ((double) numer)/((double) denom);
  printf("Ratio, ((double) %1d)/((double) %1d):  %5.2f\n",numer,denom,ratio);
}

int main(void) {
  int num = 5, den = 2;

  printRatio(num,den);
  return(0);
}
```

The helper function `printRatio` is of type `void` since it does not return a value. It takes two `int`s as input arguments and calculates their ratio in three different ways. In the first, the two `int`s are divided and the result is assigned to a `double`. In the second, the integer `denom` is **typecast** or **cast** as a `double` before the division occurs, so an `int` is divided by a `double` and the result is assigned to a `double`.[9] In the third, both the numerator and denominator are cast as `double`s before the division, so two `double`s are divided and the result is assigned to a `double`.

The `main` function simply defines two variables, `num` and `den`, and passes their values to `printRatio`, where those values are copied to `numer` and `denom`, respectively. The variables `num` and `den` are only available to `main`, and the variables `numer` and `denom` are only available to `printRatio`, since they are defined inside those functions.

Execution of any C program always begins with the `main` function, regardless of where it appears in the file.

After compiling and running, we get the output

[9] The typecasting does not change the variable `denom` itself; it simply creates a temporary `double` version of `denom` which is lost as soon as the division is complete.

```
Ratio, 5/2:                          2.00
Ratio, 5/((double) 2):               2.50
Ratio, ((double) 5)/((double) 2):    2.50
```

The first answer is "wrong," while the other two answers are correct. Why?

The first division, `numer/denom`, is an *integer* division. When the compiler sees that there are `int`s on either side of the divide sign, it assumes you want integer math and produces a result that is an `int` by simply truncating any remainder (rounding toward zero). This value, 2, is then converted to the floating point number 2.0 to be assigned to the variable `ratio`. On the other hand, the expression `numer/((double) denom)`, by virtue of the parentheses, first produces a `double` version of `denom` before performing the division. The compiler recognizes that you are dividing two different data types, so it temporarily **coerces** the `int` to a `double` so it can perform a floating point division. This is equivalent to the third and final division, except that the typecast of the numerator to `double` is explicit in the code for the third division.

Thus we have two kinds of type conversions:

- **Implicit** type conversion, or **coercion**. This occurs, for example, when a type has to be converted to carry out a variable assignment or to allow a mathematical operation. For example, dividing an `int` by a `double` will cause the compiler to treat the `int` as a `double` before carrying out the division.

- **Explicit** type conversion. An explicit type conversion is coded using a casting operator, e.g., `(double) <expression>` or `(char) <expression>`, where `<expression>` may be a variable or mathematical expression.

Certain type conversions may result in a change of value. For example, assigning the value of a `float` to an `int` results in truncation of the fractional portion; assigning a `double` to a `float` may result in roundoff error; and assigning an `int` to a `char` may result in overflow. Here's a less obvious example:

```
float f;
int i = 16777217;
f = i;            // f now has the value 16,777,216, not 16,777,217!
```

It turns out that $16{,}777{,}217 = 2^{24} + 1$ is the smallest positive integer that cannot be represented by a 32-bit `float`. On the other hand, a 32-bit `int` can represent all integers in the range -2^{31} to $2^{31} - 1$.

Some type conversions, called **promotions**, never result in a change of value because the new type can represent all possible values of the original type. Examples include converting a `char` to an `int` or a `float` to a `long double`.

We will see more on use of parentheses (Section A.4.1), the scope of variables (Section A.4.5), and defining and calling helper functions (Section A.4.6).

A more complete example: `invest.c`. Until now we have been dipping our toes in the C pool. Now let's dive in headfirst.

Our next program is called `invest.c`, which takes an initial investment amount, an expected annual return rate, and a number of years, and returns the growth of the investment over the years. After performing one set of calculations, it prompts the user for another scenario, and continues this way until the data entered is invalid. The data is invalid if, for example, the initial investment is negative or the number of years to track is outside the allowed range.

The real purpose of `invest.c`, however, is to demonstrate the syntax and a number of useful features of C.

Here's an example of compiling and running the program. The only data entered by the user are the three numbers corresponding to the initial investment, the growth rate, and the number of years.

```
> gcc invest.c -o invest
> invest
Enter investment, growth rate, number of yrs (up to 100): 100.00 1.05 5
Valid input?  1

RESULTS:
```

```
Year   0:      100.00
Year   1:      105.00
Year   2:      110.25
Year   3:      115.76
Year   4:      121.55
Year   5:      127.63

Enter investment, growth rate, number of yrs (up to 100): 100.00 1.05 200
Valid input?  0
Invalid input; exiting.
>
```

Before we look at the full `invest.c` program, let's review two principles that should be adhered to when writing a longer program: modularity and readability.

- *Modularity.* You should break your program into a set of functions that perform specific, well-defined tasks, with a small number of inputs and outputs. As a rule of thumb, no function should be longer than about 20 lines. (Experienced programmers often break this rule of thumb, but if you are a novice and are regularly breaking this rule, you're likely not thinking modularly.) Almost all variables you define should be "local" to (i.e., only recognizable by) their particular function. Global variables, which can be accessed by all functions, should be minimized or avoided altogether, since they break modularity, allowing one function to affect the operation of another without the information passing through the well-defined "pipes" (input arguments to a function or its returned results). If you find yourself typing the same (or similar) code more than once, that's a good sign you should figure out how to write a single function and just call that function from multiple places. Modularity makes it much easier to develop large programs and track down the inevitable bugs.

- *Readability.* You should use comments to help other programmers, and even yourself, understand the purpose of the code you have written. Variable and function names should be chosen to indicate their purpose. Be consistent in how you name variables and functions. Any "magic number" (constant) used in your code should be given a name and defined at the beginning of the program, so if you ever want to change this number, you can just change it at one place in the program instead of every place it is used. Global variables and constants should be written in a way that easily distinguishes them from more common local variables; for example, you could WRITE CONSTANTS IN UPPERCASE and Capitalize Globals. You should use whitespace (blank lines, spaces, tabbing, etc.) consistently to make it easy to read the program. Use a fixed-width font (e.g., `Courier`) so that the spacing/tabbing is consistent. Modularity (above) also improves readability.

The program `invest.c` demonstrates readable modular code using the structure and syntax of a typical C program. The line numbers to the left are not part of the program; they are there for reference. In the program's comments, you will see references of the form `==SecA.4.3==` that indicate where you can find more information in the review of syntax that follows the program.

```
1  /******************************************************************************
2   * PROGRAM COMMENTS (PURPOSE, HISTORY)
3   ******************************************************************************/
4
5  /*
6   * invest.c
7   *
8   * This program takes an initial investment amount, an expected annual
9   * return rate, and the number of years, and calculates the growth of
10  * the investment.  The main point of this program, though, is to
11  * demonstrate some C syntax.
12  *
13  * References to further reading are indicated by ==SecA.B.C==
14  *
```

```
15   * HISTORY:
16   * Dec 20, 2011   Created by Kevin Lynch
17   * Jan 4, 2012    Modified by Kevin Lynch (small changes, altered comments)
18   */
19
20  /*****************************************************************************
21   * PREPROCESSOR COMMANDS   ==SecA.4.3==
22   *****************************************************************************/
23
24  #include <stdio.h>      // input/output library
25  #define MAX_YEARS 100   // Constant indicating max number of years to track
26
27  /*****************************************************************************
28   * DATA TYPE DEFINITIONS (HERE, A STRUCT)  ==SecA.4.4==
29   *****************************************************************************/
30
31  typedef struct {
32    double inv0;                    // initial investment
33    double growth;                  // growth rate, where 1.0 = zero growth
34    int years;                      // number of years to track
35    double invarray[MAX_YEARS+1];   // investment array   ==SecA.4.9==
36  } Investment;                     // the new data type is called Investment
37
38  /*****************************************************************************
39   * GLOBAL VARIABLES   ==SecA.4.2, A.4.5==
40   *****************************************************************************/
41
42  // no global variables in this program
43
44  /*****************************************************************************
45   * HELPER FUNCTION PROTOTYPES  ==SecA.4.2==
46   *****************************************************************************/
47
48  int getUserInput(Investment *invp);     // invp is a pointer to type ...
49  void calculateGrowth(Investment *invp); // ... Investment ==SecA.4.6, A.4.8==
50  void sendOutput(double *arr, int years);
51
52  /*****************************************************************************
53   * MAIN FUNCTION   ==SecA.4.2==
54   *****************************************************************************/
55
56  int main(void) {
57
58    Investment inv;                // variable definition, ==SecA.4.5==
59
60    while(getUserInput(&inv)) {    // while loop ==SecA.4.14==
61      inv.invarray[0] = inv.inv0;  // struct access ==SecA.4.4==
62      calculateGrowth(&inv);       // & referencing (pointers) ==SecA.4.6, A.4.8==
63      sendOutput(inv.invarray,     // passing a pointer to an array ==SecA.4.9==
64                 inv.years);       // passing a value, not a pointer ==SecA.4.6==
65    }
66    return(0);                     // return value of main ==SecA.4.6==
67  } // ***** END main *****
68
69  /*****************************************************************************
70   * HELPER FUNCTIONS   ==SecA.4.2==
71   *****************************************************************************/
72
73  /* calculateGrowth
```

```
74   *
75   * This optimistically-named function fills the array with the investment
76   * value over the years, given the parameters in *invp.
77   */
78  void calculateGrowth(Investment *invp) {
79
80    int i;
81
82    // for loop ==SecA.4.14==
83    for (i=1; i <= invp->years; i=i+1) {   // relational operators ==SecA.4.10==
84                                           // struct access ==SecA.4.4==
85      invp->invarray[i] = invp->growth * invp->invarray[i-1];
86    }
87  } // ***** END calculateGrowth *****
88
89
90  /* getUserInput
91   *
92   * This reads the user's input into the struct pointed at by invp,
93   * and returns TRUE (1) if the input is valid, FALSE (0) if not.
94   */
95  int getUserInput(Investment *invp) {
96
97    int valid;      // int used as a boolean ==SecA.4.10==
98
99    // I/O functions in stdio.h ==SecA.4.15==
100   printf("Enter investment, growth rate, number of yrs (up to %d): ",MAX_YEARS);
101   scanf("%lf %lf %d", &(invp->inv0), &(invp->growth), &(invp->years));
102
103   // logical operators ==SecA.4.11==
104   valid = (invp->inv0 > 0) && (invp->growth > 0) &&
105     (invp->years > 0) && (invp->years <= MAX_YEARS);
106   printf("Valid input?  %d\n",valid);
107
108   // if-else ==SecA.4.13==
109   if (!valid) { // ! is logical NOT ==SecA.4.11==
110     printf("Invalid input; exiting.\n");
111   }
112   return(valid);
113 } // ***** END getUserInput *****
114
115
116 /* sendOutput
117  *
118  * This function takes the array of investment values (a pointer to the first
119  * element, which is a double) and the number of years (an int).  We could
120  * have just passed a pointer to the entire investment record, but we decided
121  * to demonstrate some different syntax.
122  */
123 void sendOutput(double *arr, int yrs) {
124
125   int i;
126   char outstring[100];  // defining a string ==SecA.4.9==
127
128   printf("\nRESULTS:\n\n");
129   for (i=0; i<=yrs; i++) {  // ++, +=, math in ==SecA.4.7==
130     sprintf(outstring,"Year %3d:  %10.2f\n",i,arr[i]);
131     printf("%s",outstring);
132   }
```

```
133   printf("\n");
134 } // ***** END sendOutput *****
```

A.4.1 Basic Syntax

Comments. Everything after a `/*` and before the next `*/` is a comment. Comments are stripped out in the preprocessing step of compilation. They are only there to make the purpose of the program, function, loop, or statement clear to yourself or other programmers. Keep the comments neat and concise for program readability. Some programmers use extra asterisks or other characters to make the comments pretty (see the examples in `invest.c`), but all that matters is that `/*` starts the comment and the next `*/` ends it.

If your comment is short, you can use `//` instead. Everything after `//` and before the next carriage return will be ignored.

Semicolons. A code statement must be completed by a semicolon. Some exceptions to this rule include preprocessor commands (see `PREPROCESSOR COMMANDS` in the program and Section A.4.3) and statements that end with blocks of code enclosed by braces { }. A single code statement may extend over multiple lines of the program listing until it is terminated by a semicolon (see, for example, the assignment to `valid` in the function `getUserInput`).

Braces and blocks of code. Blocks of code are enclosed in braces { }. Examples include entire functions (see the definition of the `main` function and the helper functions), blocks of code executed inside of a `while` loop (in the `main` function) or `for` loop (in the `calculateGrowth` and `sendOutput` functions), as well as other examples. In `invest.c`, braces are placed as shown here

```
while (<expression>) {
  /* block of code */
}
```

but this style is equivalent

```
while (<expression>)
  {
    /* block of code */
  }
```

as is this

```
while (<expression>) { /* block of code */ }
```

Which brings us to...

Whitespace. Whitespace, such as spaces, tabs, and carriage returns, is only required where it is needed to recognize keywords and other syntax. The whole program `invest.c` could be written without carriage returns after the semicolons, for example. Indentations and carriage returns should be used consistently, however, to make the program readable. Carriage returns should be used after each semicolon, statements within the same code block should be left-justified with each other, and statements in a code block nested within another code block should be indented with respect to the parent code block. Text editors should use a fixed-width font so that alignment is clear. Most IDE editors provide fixed-width fonts and automatic indentation to enhance readability.

Parentheses. C has a set of rules defining the order in which operations in an expression are evaluated, much like standard math rules that say $3 + 5 * 2$ evaluates to $3 + (10) = 13$, not $(8) * 2 = 16$. If you are uncertain of the default order of evaluation, use parentheses `( )` to enclose sub-expressions to enforce the evaluation order you want. More deeply nested parenthetical expressions will be evaluated first. For example, $3 + (40/(4 * (3 + 2)))$ evaluates to $3 + (40/(4 * 5)) = 3 + (40/20) = 3 + 2 = 5$. Parentheses can be used to control the order of evaluation for non-mathematical statements, too. An example is shown in `getUserInput` of `invest.c`:

```
valid = (invp->inv0 > 0) && (invp->growth > 0) &&
        (invp->years > 0) && (invp->years <= MAX_YEARS);
```

Each relational expression using `>` and `<=` (Section A.4.10) is evaluated before applying the logical AND operators `&&` (Section A.4.11).

A.4.2 Program Structure

`invest.c` demonstrates a typical structure for a program written in one `.c` file. When you write larger programs, you may wish to break your program into multiple files. In this case, exactly one of these files must contain a `main` function, and you have some choices as to which variables and functions in one file are visible to other files. In this appendix we will focus on programs that consist of a single file (apart from any C libraries you may include, as we will discuss in Section A.4.3). Section A.4.16 gives a simple example of a program broken up into multiple C files.

Let's consider the seven major sections of the program in order of appearance. `PROGRAM COMMENTS` describe the purpose of the program and its revision history. `PREPROCESSOR COMMANDS` define constants and "header" files that should be included, giving the program access to library functions that extend the power of the C language. This is described in more detail in Section A.4.3. In some programs, it may be helpful to define a new data type, as shown in `DATA TYPE DEFINITIONS`. In `invest.c`, several variables are packaged together in a single record or `struct` data type, as described in Section A.4.4. Any `GLOBAL VARIABLES` are then defined. These are variables that are available for use by all functions in the program. Because of this special status, the names of global variables could be Capitalized or otherwise written in a way to remind the programmer that they are not local variables (Section A.4.5).

The next section of the program contains the `HELPER FUNCTION PROTOTYPES` of the various helper functions. A prototype of a function declares the name of the function that will be defined later, its data type, and the data types of the **arguments** passed to the function. As an example, the function `printRatio` is of type `void`, since it does not return a value, and takes two arguments, each of type `int`. The function `getUserInput` takes a single argument which is a pointer to a variable of type `Investment`, a data type which is defined a few lines above, and returns an `int`.

The next section of the program, `MAIN FUNCTION`, is where the `main` function is defined. Every program has exactly one `main` function, and it is where the program starts execution. The `main` function is of type `int`, and by convention it returns a 0 if it executes successfully, and otherwise returns a nonzero value. In `invest.c`, `main` takes no arguments, hence the `void` in the argument list. On the other hand, when a program is run from the command line, it is possible to specify arguments to `main`. For example, we could have written `invest.c` to be run with a command such as this:

```
> invest 100.0 1.05 5
```

To allow this, `main` would have been defined with the following syntax:

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

Then when the program is invoked as above, the integer `argc` would be set to 4, the number of whitespace-separated strings on the command line, and `argv` would point to a vector of 4 strings, where the string `argv[0]` is 'invest', `argv[1]` is '100.0', etc. You can learn more about arrays and strings in Section A.4.9.

Finally, the last section of the program is the definition of the `HELPER FUNCTIONS` whose prototypes were given earlier. It is not strictly necessary that the helper functions have prototypes, but if not, every function should be defined before it is used by any other function. For example, none of the helper functions uses

another helper function, so they could have all been defined before the `main` function, in any order, and their function prototypes eliminated. The names of the variables in a function prototype and in the actual definition of the function need not be the same; for example, the prototype of `sendOutput` uses variables named `arr` and `years`, whereas the actual function definition uses `arr` and `yrs`. What matters is that the prototype and actual function definition have the same number of arguments, of the same types, and in the same order. In fact, in the arguments of the function prototypes, you can leave out variable names altogether, and just keep the comma separated list of argument data types.

A.4.3 Preprocessor Commands

In the preprocessing stage of compilation, all comments are stripped out of the program. In addition, the preprocessor encounters the following preprocessor commands, recognizable by the `#` character:

```
#include <stdio.h>      // input/output library
#define MAX_YEARS 100   // Constant indicating max number of years to track
```

Constants. The second line defines the constant `MAX_YEARS` to be equal to 100. The preprocessor searches for each instance of `MAX_YEARS` in the program and replaces it with 100. If we later decide that the maximum number of years to track investments should be 200, we can simply change the definition of this constant, in one place, instead of in several places. Since `MAX_YEARS` is a constant, not a variable, it can never be assigned another value somewhere else in the program. To indicate that it not a variable, a common convention is to write constants in UPPERCASE. This is not required by C, however.

Included libraries. The first line of the preprocessor commands in `invest.c` indicates that the program will use the standard C input/output library. The file `stdio.h` is called a **header** file for the library. This file is readable by a text editor and contains a number of constants that are made available to the program, as well as a set of function prototypes for input and output functions. The preprocessor replacea the `#include <stdio.h>` command with the header file `stdio.h`.[10] Examples of function prototypes that are included are

```
int printf(const char *Format, ...);
int sprintf(char *Buffer, const char *Format, ...);
int scanf(const char *Format, ...);
```

Each of these three functions is used in `invest.c`. If the program were compiled without including `stdio.h`, the compiler would generate a warning or an error due to the lack of function prototypes. See Section A.4.15 for more information on using the `stdio` input and output functions.

During the linking stage, the object code of `invest.c` is linked with the object code for `printf`, `sprintf`, and `scanf` in your C installation. Libraries like `stdio` provide access to functions beyond the basic C syntax. Other useful libraries are briefly described in Section A.4.15.

Macros. One more use of the preprocessor is to define simple function-like *macros* that you may use in more than one place in your program. Here's an example that converts radians to degrees:

```
#define RAD_TO_DEG(x) ((x) * 57.29578)
```

The preprocessor will search for any instance of `RAD_TO_DEG(x)` in the program, where `x` can be any expression, and replace it with `((x) * 57.29578)`. For example, the initial code

```
angle_deg = RAD_TO_DEG(angle_rad);
```

is replaced by

[10]This assumes that our preprocessor can find the header file somewhere in the "include path" of directories to search for header files. If the header file `header.h` sits in the same directory as `invest.c`, we would write `#include "header.h"` instead of `#include <header.h>`.

```
angle_deg = ((angle_rad) * 57.29578);
```

Note the importance of the outer parentheses in the macro definition. If we had instead used the preprocessor command

```
#define RAD_TO_DEG(x) (x) * 57.29578   // don't do this!
```

then the code

```
answer = 1.0 / RAD_TO_DEG(3.14);
```

would be replaced by

```
answer = 1.0 / (3.14) * 57.29578;
```

which is very different from

```
answer = 1.0 / ((3.14) * 57.29578);
```

Moral: if the expression you are defining is anything other than a single constant, enclose it in parentheses, to tell the compiler to evaluate the expression first. You can even enclose a single constant in parentheses; it doesn't cost you anything.

As a second example, the macro

```
#define MAX(A,B)      ((A) > (B) ? (A):(B))
```

returns the maximum of two arguments. The `?` is the *ternary operator* in C, which has the form

```
<test> ? return_value_if_test_is_true : return_value_if_test_is_false
```

The preprocessor replaces

```
maxval = MAX(13+7,val2);
```

with

```
maxval = ((13+7) > (val2) ? (13+7):(val2));
```

Why define a macro instead of just writing a function? One reason is that the macro may execute slightly faster, since no passing of control to another function and no passing of variables is needed.

A.4.4 Defining Structs and Data Types

In most programs you write, you will do just fine with the data types `int, char, float, double`, and variations. Occasionally, though, you will find it useful to define a new data type. You can do this with the following command:

```
typedef <type> newtype;
```

where `<type>` is a standard C data type and `newtype` is the name of your new data type, which will be the same as `<type>`. Then you can define a new variable `x` of type `newtype` by

```
newtype x;
```

For example, you could write

```
typedef int days_of_the_month;
days_of_the_month day;
```

You might find it satisfying that your variable `day` (taking values 1 to 31) is of type `days_of_the_month`, but the compiler will still treat it as an `int`.

A more useful example is when you have several variables that are always used together. You might like to package these variables together into a single record, as we do with the investment information in `invest.c`. This packaging can be done with a `struct`. The `invest.c` code

```
typedef struct {
  double inv0;                    // initial investment
  double growth;                  // growth rate, where 1.0 = zero growth
  int years;                      // number of years to track
  double invarray[MAX_YEARS+1];   // investment values
} Investment;                     // the new data type is called Investment
```

replaces the data type `int` in our previous `typedef` example with `struct {...}`. This syntax creates a new data type `Investment` with a record structure, with *fields* named `inv0` and `growth` of type `double`, `years` of type `int`, and `invarray`, an array of `double`s. (Arrays are discussed in Section A.4.9.) With this new type definition, we can define a variable named `inv` of type `Investment`:

```
Investment inv;
```

This definition allocates sufficient memory to hold the two `double`s, the `int`, and the array of `double`s. We can access the contents of the `struct` using the "`.`" operator:

```
int yrs;
yrs = inv.years;
inv.growth = 1.1;
```

An example of this kind of usage is seen in `main`.

Referring to the discussion of pointers in Sections A.3.2 and A.4.8, if we are working with a pointer `invp` that points to `inv`, we can use the "`->`" operator to access the contents of the record `inv`:

```
Investment inv;     // allocate memory for inv, an investment record
Investment *invp;   // invp will point to something of type Investment
int yrs;
invp = &inv;        // invp points to inv
inv.years = 5;      // setting one of the fields of inv
yrs = invp->years;  // inv.years, (*invp).years, and invp->years are all identical
invp->growth = 1.1;
```

Examples of this usage are seen in `calculateGrowth()` and `getUserInput()`.

A.4.5 Defining Variables

Variable names. Variable names can consist of uppercase and lowercase letters, numbers, and underscore characters '_'. You should generally use a letter as the first character; `var`, `Var2`, and `Global_Var` are all valid names, but `2var` is not. C is case sensitive, so the variable names `var` and `VAR` are different. A variable name cannot conflict with a reserved keyword in C, like `int` or `for`. Names should be succinct but descriptive. The variable names `i`, `j`, and `k` are often used for integers, and pointers often begin with `ptr_`, such as `ptr_var`, or end with `p`, such as `varp`, to remind you that they are pointers. These are all to personal taste, however.

Scope. The **scope** of a variable refers to where it can be used in the program. A variable may be *global*, i.e., usable by any function, or *local* to a specific function or piece of a function. A global variable is one that is defined in the `GLOBAL VARIABLES` section, outside of and before any function that uses it. Such variables can be referred to or altered in any function.[11] Because of this special status, global variables are often Capitalized. Global variable usage should be minimized for program modularity and readability.

[11] You could also define a variable outside of any function definition but *after* some of the function definitions. This quasi-global variable would be available to all functions defined after the variable is defined, but not those before. This practice is discouraged, as it makes the code harder to read.

A local variable is one that is defined in a function. Such a variable is only usable inside that function, after the definition.[12] If you choose a local variable name `var` that is also the name of a global variable, inside that function `var` will refer to the local variable, and the global variable will not be available. It is not good practice to choose local variable names to be the same as global variable names, as it makes the program confusing to understand.

A local variable can be defined in the argument list of a function definition, as in `sendOutput` at the end of `invest.c`:

```
void sendOutput(double *arr, int yrs) {    // ...
```

Otherwise, local variables are defined at the beginning of the function code block by syntax similar to that shown in the function `main`.

```
int main(void) {
  Investment inv;    // Investment is a variable type we defined
  // ... rest of the main function ...
```

Since this definition appears within the function, `inv` is local to `main`. Had this definition appeared before any function definition, `inv` would be a global variable.

Definition and initialization. When a variable is defined, memory for the variable is allocated. In general, you cannot assume anything about the contents of the variable until you have initialized it. For example, if you want to define a `float x` with value 0.0, the command

```
float x;
```

is insufficient. The memory allocated may have random 0's and 1's already in it, and the allocation of memory does not generally change the current contents of the memory. Instead, you can use

```
float x = 0.0;
```

to initialize the value of `x` when you define it. Equivalently, you could use

```
float x;
x = 0.0;
```

Static local variables. Each time a function is called, its local variables are allocated space in memory. When the function completes, its local variables are thrown away, freeing memory. If you want to keep the results of some calculation by the function after the function completes, you could either return the results from the function or store them in a global variable. An alternative is to use the `static` modifier in the local variable definition, as in the following program:

```
#include <stdio.h>

void myFunc(void) {
  static char ch='d';   // this local variable is static, allocated and initialized
                        // only once during the entire program
  printf("ch value is %d, ASCII character %c\n",ch,ch);
  ch = ch+1;
}

int main(void) {
  myFunc();
  myFunc();
  myFunc();
  return 0;
}
```

[12]Since we recommend that each function be brief, you can define all local variables in that function at the beginning of the function, so we can see in one place what local variables the function uses. Some programmers prefer instead to define variables just before their first use, to minimize their scope. Older C specifications required that all local variables be defined at the beginning of a code block enclosed by braces { }.

The `static` modifier in the definition of `ch` in `myFunc` means that `ch` is only allocated, and initialized to `'d'`, the first time `myFunc` is called during the execution of the program. This allocation persists after the function is exited, and the value of `ch` is remembered. The output of this program is

```
ch value is 100, ASCII character d
ch value is 101, ASCII character e
ch value is 102, ASCII character f
```

Numerical values. Just as you can assign an integer a base-10 value using commands like `ch=100`, you can assign a number written in hexadecimal notation by putting "`0x`" at the beginning of the digit sequence, e.g.,

```
unsigned char ch = 0x4D;
```

This form may be convenient when you want to directly control bit values. This is often useful in microcontroller applications.

A.4.6 Defining and Calling Functions

A function definition consists of the function's data type, the function name, a list of arguments that the function takes as input, and a block of code. Allowable function names follow the same rules as variables. The function name should make clear the purpose of the function, such as `getUserInput` in `invest.c`.

If the function does not return a value, it is defined as type `void`, as with `calculateGrowth`. If it does return a value, such as `getUserInput` which returns an `int`, the function should end with the command

```
return(val);
```

or

```
return val;
```

where `val` is a variable of the same type as the function. The `main` function is of type `int` and should return 0 upon successful completion.

The function definition

```
void sendOutput(double *arr, int yrs) {  // ...
```

indicates that `sendOutput` returns nothing and takes two arguments, a pointer to type `double` and an `int`. When the function is called with the statement

```
sendOutput(inv.invarray, inv.years);
```

the `invarray` and `years` fields of the `inv` structure in `main` are copied to `sendOutput`, which now has its own local copies of these variables, stored in `arr` and `yrs`. The difference is that `yrs` is simply data, while `arr` is a pointer, specifically the address of the first element of `invarray`, i.e., `&(inv.invarray[0])`. (Arrays will be discussed in more detail in Section A.4.9.) Since `sendOutput` now has the memory address of the beginning of this array, *it can directly access, and potentially change, the original array seen by* `main`. On the other hand, `sendOutput` cannot by itself change the value of `inv.years` in `main`, since it only has a copy of that value, not the actual memory address of `main`'s `inv.years`. `sendOutput` takes advantage of its direct access to the `inv.invarray` to print out all the values stored there, eliminating the need to copy all the values of the array from `main` to `sendOutput`.

The function `calculateGrowth`, which is called with a pointer to `main`'s `inv` data structure, takes advantage of its direct access to the `invarray` field to change the values stored there.

When a function is called with a pointer argument, it is sometimes called a *call by reference*; the call sends a reference (address, or pointer) to data. When a function is called with non-pointer data, it is sometimes called a *call by value*; data is copied over, but not an address.

If a function takes no arguments and returns no value, we can define it as `void myFunc(void)` or `void myFunc()`. The function is called using

```
myFunc();
```

A.4.7 Math

Standard *binary* math operators (operators on two operands) include `+`, `-`, `*`, and `/`. These operators take two operands and return a result, as in

```
ratio = a/b;
```

If the operands are the same type, then the CPU carries out a division (or add, subtract, multiply) specific for that type and produces a result of the same type. In particular, if the operands are integers, the result will be an integer, even for division (fractions are rounded toward zero). If one operand is an integer type and the other is a floating point type, the integer type will generally be coerced to a floating point to allow the operation (see the `typecast.c` program description of Section A.4).

The modulo operator `%` takes two integers and returns the remainder of their division, i.e.,

```
int i;
i = 16%7;  // i is now equal to 2
```

C also provides `+=`, `-=`, `*=`, `/=`, `%=` to simplify some expressions, as shown below:

```
x = x * 2;  y = y + 7;    // this line of code is equivalent...
x *= 2;     y += 7;       // ...to this one
```

Since adding one to an integer or subtracting one from an integer are common operations in loops, these have a further simplification. For an integer `i`, we can write

```
i++;  // adds 1 to i, equivalent to i = i+1;
i--;  // equivalent to i = i-1;
```

In fact we also have the syntax `++i` and `--i`. If the `++` or `--` come in front of the variable, the variable is modified before it is used in the rest of the expression. If they come after, the variable is modified after the expression has been evaluated. So

```
int i=5,j;
j = (++i)*2;   // after this line, i is 6 and j is 12
```

but

```
int i=5,j;
j = (i++)*2;   // after this line, i is 6 and j is 10
```

But your code would be much more readable if you just wrote `i++` before or after the `j=i*2` line.

If your program includes the C math library with the preprocessor command `#include <math.h>`, you have access to a much larger set of mathematical operations, some of which are listed here:

```
int     abs      (int x);              // integer absolute value
double  fabs     (double x);           // floating point absolute value
double  cos      (double x);           // all trig functions work in radians, not degrees
double  sin      (double x);
double  tan      (double x);
double  acos     (double x);           // inverse cosine
double  asin     (double x);
double  atan     (double x);
double  atan2    (double y, double x); // two-argument arctangent
double  exp      (double x);           // base e exponential
double  log      (double x);           // natural logarithm
double  log2     (double x);           // base 2 logarithm
double  log10    (double x);           // base 10 logarithm
double  pow      (double x, double y); // raise x to the power of y
double  sqrt     (double x);           // square root of x
```

These functions also have versions for `float`s. The names of those functions are identical, except with an 'f' appended to the end, e.g., `cosf`.

When compiling programs using `math.h`, remember to include the linker flag `-lm`, e.g.,

```
gcc myprog.c -o myprog -lm
```

The math library is not linked by default like most other libraries.

A.4.8 Pointers

It's a good idea to review the introduction to pointers in Section A.3.2 and the discussion of call by reference in Section A.4.6. In summary, the operator `&` references a variable, returning a pointer to (the address of) that variable, and the operator `*` dereferences a pointer, returning the contents of the address.

These statements define a variable `x` of type `float` and a pointer `ptr` to a variable of type `float`:

```
float x;
float *ptr;
```

At this point, the assignment

```
*ptr = 10.3;
```

would result in an error, because the pointer `ptr` does not currently point to anything. The following code would be valid:

```
ptr = &x;         // assign ptr to the address of x; x is the "pointee" of ptr
*ptr = 10.3;      // set the contents at address ptr to 10.3; now x is equal to 10.3
*(&x) = 4 + *ptr; // the * and & on the left cancel each other; x is set to 14.3
```

Since `ptr` is an address, it is an integer (technically the type is "pointer to type float"), and we can add and subtract integers from it. For example, say that `ptr` contains the value n, and then we execute the statement

```
ptr = ptr + 1;    // equivalent to ptr++;
```

If we now examined `ptr`, we would find that it has the value $n + 4$. Why? Because the compiler knows that `ptr` points to the type `float`, so when we add 1 to `ptr`, the assumption is that we want to increment one `float` in memory, not one byte. Since a `float` occupies four bytes, the address `ptr` must increase by 4 to point to the next `float`. The ability to increment a pointer in this way can be useful when dealing with arrays, next.

A.4.9 Arrays and Strings

One-dimensional arrays. An array of five `float`s can be defined by

```
float arr[5];
```

We could also initialize the array at the time we define it:

```
float arr[5] = {0.0, 10.0, 20.0, 30.0, 40.0};
```

Each of these definitions allocates five `float`s in memory, accessed by `arr[0]` (initialized to 0.0 above) through `arr[4]` (initialized to 40.0). The assignment

```
arr[5] = 3.2;
```

is a mistake, since only `arr[0..4]` have been allocated. This statement would likely compile just fine, because compilers typically do not check for indexing arrays out of bounds. The best result at this point would be for your program to crash, to alert you to the fact that you are overwriting memory that may be allocated for another purpose. More insidiously, the program could seem to run just fine, but with difficult-to-debug erratic behavior. Bottom line: never access arrays out of bounds!

In the expression `arr[i]`, `i` is an integer called the *index*, and `arr[i]` is of type `float`. The variable `arr` by itself is actually a pointer to the first element of the array, equivalent to `&(arr[0])`. The address `&(arr[i])` is located at the address `arr` plus `i*4` bytes, since the elements of the array are stored consecutively, and a `float` uses four bytes. Both `arr[i]` and `*(arr+i)` are correct syntax to access the `i`'th element of the array. Since the compiler knows that `arr` is a pointer to the four-byte type `float`, the address represented by `(arr+i)` is `i*4` bytes higher than the address `arr`.

Consider the following code snippet:

```
float arr[5] = {0.0, 10.0, 20.0, 30.0, 40.0};
float *ptr;
ptr = arr + 3;
// arr[0] contains 0.0 and ptr[0] = arr[3] = 30.0
// arr[0] is equivalent to *arr; ptr[0] is equivalent to *ptr and *(arr+3);
// ptr is equivalent to &(arr[3])
```

If we'd like to pass the array `arr` to a function that initializes each element of the array, we could call

```
arrayInit(arr,5);
```

or

```
arrayInit(&(arr[0]),5);
```

The function definition for `arrayInit` might look like

```
void arrayInit(float *vals, int length) {

  int i;

  for (i=0; i<length; i++) vals[i] = i*10.0;
  // equivalently, we could substitute the line below for the line above
  // for (i=0; i<length; i++) {*vals = i*10.0; vals++;}
}
```

The pointer `vals` in `arrayInit` is set to point to the same location as `arr` in the calling function. Therefore `vals[i]` refers to the same memory contents that `arr[i]` does.

Note that `arr` does not carry any information on the length of the array. This is why we have to separately send the length of the array to `arrayInit`.

Strings. A string is an array of `char`s. The definition

```
char s[100];
```

allocates memory for 100 `char`s, `s[0]` to `s[99]`. We could initialize the array with

```
char s[100] = "cat"; // note the double quotes
```

This places a 'c' (integer value 99) in `s[0]`, an 'a' (integer value 97) in `s[1]`, a 't' (integer value 116) in `s[2]`, and a value of 0 in `s[3]`, corresponding to the NULL character and indicating the end of the string. (You could also do this, less elegantly, by initializing just those four elements using braces as we did with the `float` array above.)

You notice that we allocated more memory than was needed to hold "cat." Perhaps we will append something to the string in future, so we might want to allocate that extra space just in case. But if not, we could have initialized the string using

```
char s[] = "cat";
```

and the compiler would only assign the minimum memory needed.

The function `sendOutput` in `invest.c` shows an example of constructing a string using `sprintf`, a function provided by `stdio.h`. Other functions for manipulating strings are provided in `string.h`. Both of these libraries are described briefly in Section A.4.15.

Multi-dimensional arrays. The definition

```
int mat[2][3];
```

allocates memory for 6 `int`s, `mat[0][0]` to `mat[1][2]`, which can be thought of as a two-dimensional array, or matrix. These occupy a contiguous region of memory, with `mat[0][0]` at the lowest memory location, followed by `mat[0][1]`, `mat[0][2]`, `mat[1][0]`, `mat[1][1]`, and `mat[1][2]`. This matrix can be initialized using nested braces,

```
int mat[2][3] = {{0, 1, 2}, {3, 4, 5}};
```

Higher-dimensional arrays can be created by simply adding more indexes. In memory, a "row" of the rightmost index is completed before incrementing the next index to the left.

Static vs. dynamic memory allocation. A command of the form `float arr[5]` is called *static memory allocation*. This means that the size of the array is known at compile time. Another option is *dynamic memory allocation*, where the size of the array can be chosen at run time.[13] With the C library `stdlib.h` included using the preprocessor command `#include <stdlib.h>`, the syntax

```
float *arr; // arr is a pointer to float, but no memory has been allocated for the array
int i=5;
arr = (float *) malloc(i * sizeof(float));  // allocate the memory
```

allocates `arr[0..4]`, and

```
free(arr);
```

releases the memory when it is no longer needed.[14] If `malloc` cannot allocate the requested memory, perhaps because the computer is out of memory, it returns a NULL pointer (i.e., `arr` will have value 0).

A.4.10 Relational Operators and TRUE/FALSE Expressions

`==`	equal
`!=`	not equal
`>, >=`	greater than, greater than or equal to
`<, <=`	less than, less than or equal to

Relational operators operate on two values and evaluate to 0 or 1. A 0 indicates that the expression is FALSE and a 1 indicates that the expression is TRUE. For example, the expression `(3>=2)` is TRUE, so it evaluates to 1, while `(3<2)` evaluates to 0, or FALSE.

The most common mistake is using `=` to test for equality instead of `==`. For example, using the `if` conditional syntax (Section A.4.13), the test

```
int i=2;
if (i=3) printf("Test is true.");
```

[13] Dynamic memory is allocated from the *heap*, a portion of memory set aside for dynamic allocation (and therefore is not available for statically allocated variables and program code). You may have to adjust linker options setting the size of the heap.

[14] Bookkeeping has kept track of the size of the block associated with the address `arr`, so you don't need to tell `free` how much memory to release.

will always evaluate to TRUE, because the expression (i=3) assigns the value of 3 to i, and the expression evaluates to 3. Any nonzero value is treated as logical TRUE. If the condition is written (i==3), it will operate as intended, evaluating to 0 (FALSE).

Be aware of potential pitfalls in checking equality of floating point numbers. Cconsider the following program:

```
#include <stdio.h>
#define VALUE 3.1
int main(void) {
  float  x = VALUE;
  double y = VALUE;
  if (x==VALUE) printf("x is equal to %lf.\n",VALUE);
  else printf("x is not equal to %lf!\n",VALUE);
  if (y==VALUE) printf("y is equal to %lf.\n",VALUE);
  else printf("y is not equal to %lf!\n",VALUE);
  return 0;
}
```

You might be surprised to see that your program says that x is not equal while y is! In fact, neither x nor y are exactly 3.1 due to roundoff error in the floating point representation. However, by default, the constant 3.1 is treated as a double, so the double y carries the identical (wrong) value. If you want a constant to be treated explicitly as a float, you can write it as 3.1F, and if you want it to be treated as a long double, you can write it as 3.1L.

A.4.11 Logical Operators

Logical operators include AND, OR, and NOT, written as &&, ||, and !, respectively. Here are some examples:

```
     (3>2) && (4!=0)  // (TRUE) AND (TRUE) evaluates to TRUE
     (3>2) || (4==0)  // (TRUE) OR (FALSE) evaluates to TRUE
    !(3>2) || (4==0)  // NOT(TRUE) OR (FALSE) evaluates to FALSE
```

Another example is given in getUserInput, where four expressions are AND'ed. As always, if you are unsure of the order of evaluating a string of logical expressions, use parentheses to enforce the order you want.

A.4.12 Bitwise Operators

~	bitwise NOT
&	bitwise AND
|	bitwise OR
^	bitwise XOR
>>	shift bits to the right (shifting in 0's from the left)
<<	shift bits to the left (shifting in 0's from the right)

Bitwise operators act directly on the bits of the operand(s), as in the following example:

```
unsigned char a=0xC, b=0x6, c;  // in binary, a is 0b00001100 and b is 0b00000110
c = ~a;     // NOT; c is 0xF3 or 0b11110011
c = a & b;  // AND; c is 0x04 or 0b00000100
c = a | b;  // OR;  c is 0x0E or 0b00001110
c = a ^ b;  // XOR; c is 0x0A or 0b00001010
c = a >> 3; // SHIFT RT 3; c is 0x01 or 0b00000001, one 1 is shifted off the right end
c = a << 3; // SHIFT LT 3; c is 0x60 or 0b01100000, 1's shifted to more significant digits
```

Much like the math operators, we also have the assignment expressions &=, |=, ^=, >>=, and <<=, so a &= b is equivalent to a = a&b.

A.4.13 Conditional Statements

If-Else. The basic `if-else` construct takes this form:

```
if (<expression>) {
  // execute this code block if <expression> is TRUE, then exit
}
else {
  // execute this code block if <expression> is FALSE
}
```

If the code block is a single statement, the braces are not necessary. The `else` and the block after it can be eliminated if no action needs to be taken when `<expression>` is FALSE.

`if-else` statements can be made into arbitrarily long chains:

```
if (<expression1>) {
  // execute this code block if <expression1> is TRUE, then exit this if-else chain
}
else if (<expression2>) {
  // execute this code block if <expression2> is TRUE, then exit this if-else chain
}
else {
  // execute this code block if both expressions above are FALSE
}
```

An example `if` statement is in `getUserInput`.

Switch. If you would like to check if the value of a single expression is one of several possibilities, a `switch` may be simpler than a chain of `if-else` statements. Here is an example:

```
char ch;
// ... omitting code that sets the value of ch ...
switch (ch) {
  case 'a':      // execute these statements if ch has value 'a'
    <statement>;
    <statement>;
    break;       // exit the switch statement
  case 'b':
    // ... some statements
    break;
  case 'c':
    // ... some statements
    break;
  default:       // execute this code if none of the previous cases applied
    // ... some statements
}
```

A.4.14 Loops

`for` loop. A `for` loop has the following syntax:

```
for (<initialization>; <test>; <update>) {
  // code block
  }
```

If the code block consists of only one statement, the surrounding braces can be eliminated.

The sequence is as follows: at the beginning of the loop, the `<initialization>` statement is executed. Then the `<test>` is evaluated. If it is TRUE, then the code block is executed, the `<update>` is performed, and we return to the `<test>`. If it is FALSE, the `for` loop is exited.

The following `for` loop is in `calculateGrowth`:

```
for (i=1; i <= invp->years; i=i+1) {
  invp->invarray[i] = invp->growth*invp->invarray[i-1];
}
```

The <initialization> step sets i=1. The <test> is TRUE if i is less than or equal to the number of years we will calculate growth in the investment. If it is TRUE, the value of the investment in year i is calculated from the value in year i-1 and the growth rate. The <update> adds 1 to i. In this example, the code block is executed for i values of 1 to invp->years.

It is possible to perform more than one statement in the <initialization> and <update> steps by separating the statements by commas. For example, we could write

```
for (i=1,j=10; i <= 10; i++, j--)  { /* code */ };
```

if we want i to count up and j to count down.

while loop. A while loop has the following syntax:

```
while (<test>) {
  // code block
}
```

First, the <test> is evaluated, and if it is FALSE, the while loop is exited. If it is TRUE, the code block is executed and we return to the <test>.

In main of invest.c, the while loop executes until the function getUserInput returns 0, i.e., FALSE. getUserInput collects the user's input and returns an int that is 0 if the user's input is invalid and 1 if it is valid.

do-while loop. This is similar to a while loop, except the <test> is executed at the end of the code block.

```
do {
  // code block
} while (<test>);
```

break and continue. If anywhere in the loop's code block the command break is encountered, the program will exit the loop. If the command continue is encountered, the rest of the commands in the code block will be skipped, and control will return to the <update> in a for loop or the <test> in a while or do-while loop. Examples:

```
while (<test1>) {
  if (<test2>) break;  // jump out of the while loop
  // ...
}

while (<test1>) {
  if (<test2>) continue;  // skip the rest of the loop and go back to <test1>
  x = x+3;
}
```

A.4.15 Some Useful Libraries

Libraries can be used in your C program if you include the .h header file that defines the library function prototypes.[15] We have already seen examples of functions in header files such as stdio.h, which contains input/output functions; math.h in Section A.4.7; and stdlib.h in Section A.4.9.

It is well beyond our scope to provide details on the standard libraries in C. If you are interested, try a web search on "standard libraries in C." Here we highlight a few particularly useful functions in stdio.h, string.h, and stdlib.h.

[15] Reminder: if you include <math.h>, you should also compile your program with the -lm flag, so the math library is linked during the linking stage.

Input and Output: `stdio.h`

```
int printf(const char *Format, ...);
```

The function `printf` is used to print to the "standard output," which, for a PC, is typically the screen. It takes a formatting string `Format` and a variable number of extra arguments, determined by the formatting string, as indicated by the `...` notation. The keyword `const` means that `printf` cannot change the string `Format`.

An example comes from our program `printout.c`:

```
int i; float f; double d; char c;
i = 32; f = 4.278; d = 4.278; c = 'k';
printf("Formatted string:  i = %4d  c = '%c'\n",i,c);
printf("f = %25.23f  d = %25.23lf\n",f,d);
```

which produces the output

```
Formatted string:  i =   32  c = 'k'
f = 4.27799987792968750000000  d = 4.27799999999999958077979
```

The formatting strings consist of plain text, the special character `\n` that prints a newline, and directives of the form `%4d` and `%25.23f`. Each directive indicates that `printf` will be looking for a corresponding variable in the argument list to insert into the output. A non-exhaustive list of directives is given here:

`%d`	Print an `int`. Corresponding argument should be an `int`.
`%u`	Print an `unsigned int`. Corresponding argument should be an integer data type.
`%ld`	Print a `long int`.
`%f`	Print a `float`.
`%lf`	Print a `double`, or "long float."
`%c`	Print a character according to the ASCII table. Argument should be `char`.
`%s`	Print a string. Argument should be a pointer to a `char` (first element of a string).
`%X`	Print an `unsigned int` as a hex number.

The directive `%d` can be written instead as `%4d`, for example, meaning that four spaces are allocated to write the integer, which will be right-justified in that space with unused spaces blank. The directive `%f` can be written instead as `%6.3f`, indicating that six spaces are reserved to write out the variable, with one of those spaces being the decimal point and three of the spaces after the decimal point.

```
int sprintf(char *str, const char *Format, ...);
```

Instead of printing to the screen, `sprintf` prints to the string `str`. An example of this is in `sendOutput`.

```
int scanf(const char *Format, ...);
```

The function `scanf` is a formatted read from the "standard input," which is typically the keyboard. Arguments to `scanf` consist of a formatting string and pointers to variables where the input should be stored. Typically the formatting string consists of directives like `%d`, `%f`, etc., separated by whitespace. The directives are similar to those for `printf`, except they don't accept spacing modifiers (like the 5 in `%5d`).

For each directive, `scanf` expects to see a pointer to a variable of that type in the argument list. A very common mistake is the following:

```
int i;
scanf("%d",i);  // WRONG!  We need a pointer to the variable.
scanf("%d",&i); // RIGHT.
```

The pointer allows `scanf` to put the input into the right place in memory.

`getUserInput` uses the statement

```
scanf("%lf %lf %d", &(invp->inv0), &(invp->growth), &(invp->years));
```

to read in two doubles and an integer and place them into the appropriate spots in the investment data structure. `scanf` ignores the whitespace (tabs, newlines, spaces, etc.) between the inputs.

```
int sscanf(char *str, const char *Format, ...);
```

Instead of scanning from the keyboard, `scanf` scans the string pointed to by `str`.

```
FILE* fopen(const char *Path, const char *Mode);
int fclose(FILE *Stream);
int fscanf(FILE *Stream, const char *Format, ...);
int fprintf(FILE *Stream, const char *Format, ...);
```

These commands are for reading from and writing to files. Say you've got a file named `inputfile`, sitting in the same directory as the program, with information your program needs. The following code would read from it and then write to the file `outputfile`.

```
int i;
double x;
FILE *input, *output;
input = fopen("inputfile","r");    // "r" means you will read from this file
output = fopen("outputfile","w");  // "w" means you will write to this file
fscanf(input,"%d %lf",&i,&x);
fprintf(output,"I read in an integer %d and a double %lf.\n",i,x);
fclose(input);                     // these streams should be closed ...
fclose(output);                    // ... at the end of the program
```

```
int fputc(int character, FILE *stream);
int fputs(const char *str, FILE *stream);
int fgetc(FILE *stream);
char* fgets(char *str, int num, FILE *stream);
int puts(const char *str);
char* gets(char *str);
```

These commands get a character or string from a file, write (put) a character or string to a file, put a string to the screen, or get a string from the keyboard.

String Manipulation: `string.h`

```
char* strcpy(char *destination, const char *source);
```

Given two strings, `char destination[100],source[100]`, we cannot simply copy one to the other using the assignment `destination = source`. Instead we use `strcpy(destination,source)`, which copies the string `source` (until reaching the string terminator character, integer value 0) to `destination`. The string `destination` must have enough memory allocated to hold the source string.

```
char* strcat(char *destination, const char *source);
```

Appends the string in `source` to the end of the string `destination`.

```
int strcmp(const char *s1, const char *s2);
```

Returns 0 if the two strings are identical, a positive integer if the first unequal character in `s1` is greater than `s2`, and a negative integer if the first unequal character in `s1` is less than `s2`.

```
size_t strlen(const char *s);
```

The type `size_t` is an unsigned integer type. `strlen` returns the length of the string `s`, where the end of

the string is indicated by the string terminator character (value 0).

```
void* memset(void *s, int c, size_t len);
```

`memset(` writes `len` bytes of the value `c` (converted to an `unsigned char`) starting at the beginning of the string `s`. So

```
char s[10];
memset(s,'c',5);
```

would fill the first five characters of the string `s` with the character `'c'` (or integer value 99). This can be a convenient way to initialize a string.

General Purpose Functions in `stdlib.h`

```
void* malloc(size_t objectSize)
```

`malloc` is used for dynamic memory allocation. An example use is in Section A.4.9.

```
void free(void *objptr)
```

`free` is used to release memory allocated by `malloc`. An example use is in Section A.4.9.

```
int rand()
```

It is sometimes useful to generate random numbers, particularly for games. The code

```
int i;
i = rand();
```

places in `i` a pseudo-random number between 0 and `RAND_MAX`, a constant which is defined in `stdlib.h` (2,147,483,647 on my laptop). To convert this to an integer between 1 and 10, you could follow with

```
i = 1 + (int) ((10.0*i)/(RAND_MAX+1.0));
```

One drawback of the code above is that calling `rand` multiple times will lead to the same sequence of random numbers every time the program is run. The usual solution is to "seed" the random number algorithm with a different number each time, and this different number is often taken from a system clock. The `srand` function is used to seed `rand`, as in the example below:

```
#include <stdio.h>   // allows use of printf()
#include <stdlib.h>  // allows use of rand() and srand()
#include <time.h>    // allows use of time()

int main(void) {
  int i;
  srand(time(NULL)); // seed the random number generator with the current time
  for (i=0; i<10; i++) printf("Random number:  %d\n",rand());
  return 0;
}
```

If we take out the line with `srand`, this program produces the same ten "random" numbers every time we run it. Note that this program includes the `time.h` library to allow the use of the `time` function.

```
void exit(int status)
```

When `exit` is invoked, the program exits with the exit code `status`. `stdlib.h` defines `EXIT_SUCCESS` with value 0 and `EXIT_FAILURE` with value −1, so that a typical call to `exit` might look like

```
exit(EXIT_SUCCESS);
```

A.4.16 Multiple File Programs and Libraries

Our programs have been making use of the stdio library, which provides a number of functions allowing printing to the screen and reading input from the keyboard. We gain access to these functions because of two things:

1. The preprocessor command `#include <stdio.h>` inserts the header file `stdio.h` consisting of function prototypes that declare functions such as `printf`, allowing you to use `printf` in your program.

2. The linker links in the pre-compiled library object code for `printf`.

Thus a library consists of a header file and object code. We could also loosely define a library to consist of a header file and a C file without a `main` function.

The purpose of a library is to gather together functions that are likely to be useful in many programs, so you don't have to rewrite the code for each program. Let's look at a simple example.

A Simple Example: the rad2volume Library

Say you plan to write a number of programs that all need a function to calculate the volume of a sphere given its radius. Instead of putting the same function into a bunch of different C files, you decide to write one helper C file, `rad2volume.c`, with a function `double radius2Volume(double r)` that you make available to other C files. For good measure, you decide to make the constant `MY_PI` available also. To test your new rad2volume library consisting of `rad2volume.c` and `rad2volume.h`, you create a `main.c` file that uses it. The three files are given below.

```
// ***** file:  rad2volume.h *****
#ifndef RAD2VOLUME_H                // "include guard"; don't include twice in one compilation
#define RAD2VOLUME_H                // second line of the "include guard"

#define MY_PI 3.1415926             // constant available to files including rad2Volume.h
double radius2Volume(double r);     // prototype available to files including rad2Volume.h

#endif                              // third line, and end, of "include guard"
```

```
// ***** file:  rad2volume.c *****
#include <math.h>                   // for the function pow
#include "rad2volume.h"             // if the header is in the same directory, use "quotes"

double cuber(double x) {            // this function is not available externally
  return(pow(x,3.0));
}

double radius2Volume(double rad) {  // function definition
  return((4.0/3.0)*MY_PI*cuber(rad));
}
```

```
// ***** file:  main.c *****
#include <stdio.h>
#include "rad2volume.h"

int main(void) {
  double radius = 3.0, volume;
  volume = radius2Volume(radius);
  printf("Pi is approximated as %25.23lf.\n",MY_PI);
  printf("The volume of the sphere is %8.4lf.\n",volume);
  return 0;
}
```

The C file rad2volume.c contains two functions, cuber and radius2Volume. The function cuber is only meant for internal, private use by rad2volume.c, so there is no prototype in rad2volume.h. On the other hand, the function radius2Volume is meant for public use by other C files, so a prototype for radius2Volume is included in the library header file rad2volume.h. The constant MY_PI is also meant for public use, so it is defined in rad2volume.h. Now radius2Volume and MY_PI are available to any file that includes rad2volume.h. In this case, they are available to main.c and rad2volume.c.

Each of main.c and rad2volume.c is compiled independently to create the object codes main.o and rad2volume.o. These object codes are then linked to create the final executable. main.c compiles successfully because it expects that, during the linking stage, MY_PI and rad2Volume will be properly linked (in this case, to main.o).

Note the three lines making up the *include guard* in rad2volume.h. During preprocessing of a C file, if rad2volume.h is included, the flag RAD2VOLUME_H is defined. If the same C file tries to include RAD2VOLUME.h again, the include guard will recognize that RAD2VOLUME_H already exists and therefore skip the prototype and constant definition, down to the #endif. Without include guards, if we wrote a .c file including both header1.h and header2.h, for example, not knowing that header2.h already includes header1.h, we would get a compilation error due to duplicate declarations.

The two C files can be compiled into object codes using the commands

```
gcc -c rad2volume.c -o rad2volume.o
gcc -c main.c -o main.o
```

where the -c flag indicates that the code should be compiled and assembled, but not linked. The result is the object codes rad2volume.o and main.o. The two object codes can be linked into a final executable using

```
gcc rad2volume.o main.o -o myprog
```

Alternatively, the single command

```
gcc rad2volume.c main.c -o myprog
```

could be used in place of the preceding three lines to compile and link without writing the object files. Executing myprog, the output is

```
Pi is approximated as 3.14159260000000006840537.
The volume of the sphere is 113.0973.
```

Generalizing

Generalizing the simple example above, a header file defines constants, macros, new data types, and function prototypes that are needed by the files that #include them. A header file can be included by C source files or other header files. Figure A.1 illustrates a project consisting of one C source file with a main function and two helper C source files without a main function. (Every C project has exactly one .c file with a main function.) Each of the helper C files has its own header file, and a "library," in our simplified context, is a C helper file together with its header file. This project also has one other header file, general.h, without an associated C file. This header is for general constant, macro, and data type definitions that are not specific to either library nor the main C file. The arrows indicate that the pointed-to file includes the pointed-from header file.

Assuming all the files are in the same directory, the project in Figure A.1 can be built by the following four command-line commands, which create three object files (one for each source file) and link them together into myprog:

```
gcc -c main.c -o main.o
gcc -c helper1.c -o helper1.o
gcc -c helper2.c -o helper2.o
gcc main.o helper1.o helper2.o -o myprog
```

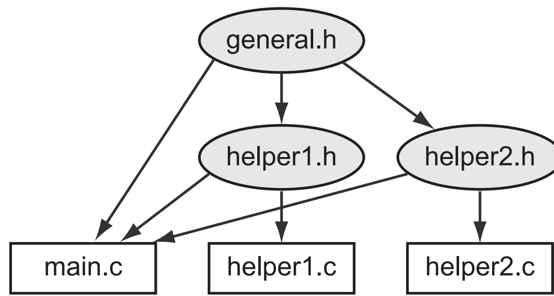

Figure A.1: An example project consisting of three C files and three header files. Arrows point from header files to files that include them.

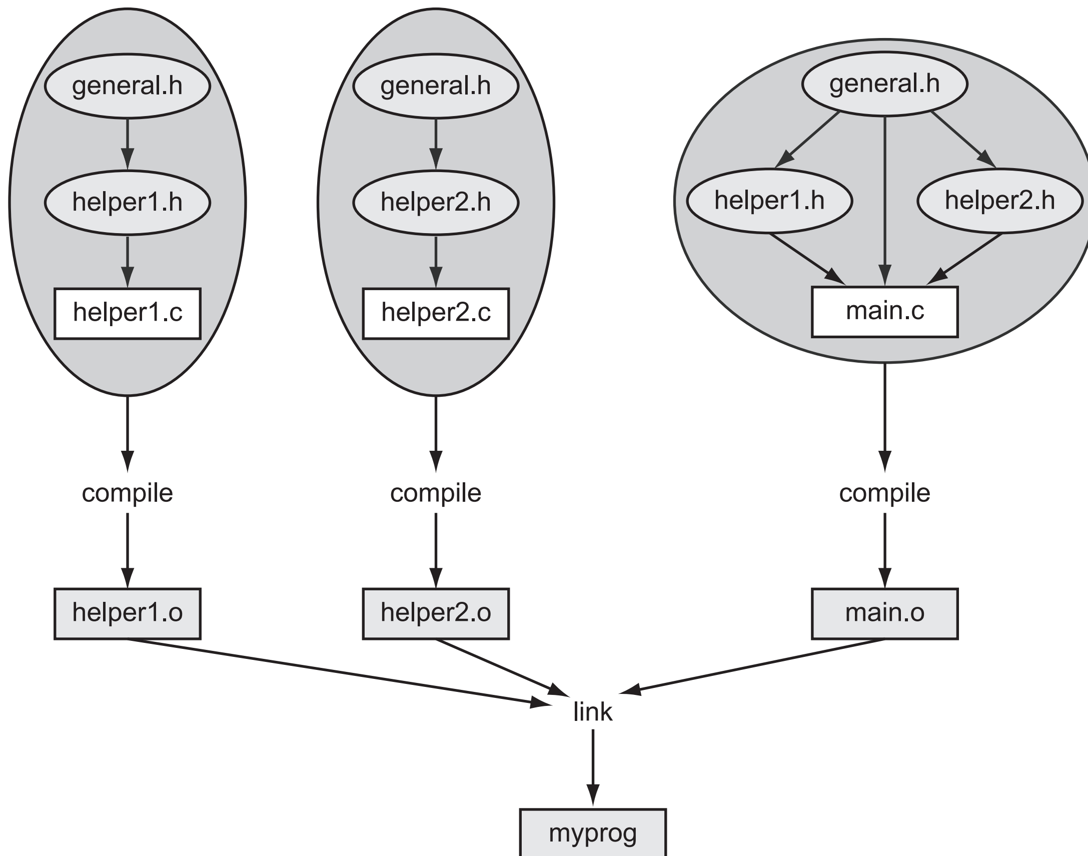

Figure A.2: The building of the project in Figure A.1.

The build is illustrated in Figure A.2. Each C file is compiled independently and requires the constants, macros, data types, and function prototypes needed to successfully compile into an object file. During compilation of a single C file to a single object code file, the compiler does not have (nor need) access to the source code for functions in other C files. If `main.c` uses a function in `helper1.c`, for example, it needs only a prototype of the function, provided by `helper1.h`, so the compiler knows the type of the function and what arguments it takes when it creates `main.o`. Calls to the function from `main.o` are linked to the actual function in `helper1.o` at the linker stage.

Most libraries, once finalized, should not be changed, so it is usually not necessary to recompile `helper1.c`

to helper1.o. Instead, your project can just use the precompiled helper1.o at the linking stage.

According to Figures A.1 and A.2, main.c has the following preprocessor commands:

```
#include "general.h"    // use "quotes" for header files that live in the same directory
#include "helper1.h"
#include "helper2.h"
```

The preprocessor replaces these commands with copies of the files general.h, helper1.h, and helper2.h. But when it includes helper1.h, it finds that helper1.h tries to include a second copy of general.h (see Figure A.1; helper1.h has a #include "general.h" command). Since general.h has already been copied in, it should not be copied again; otherwise we would have multiple copies of the same function prototypes, constant definitions, etc. To prevent this kind of error, header files should have include guards, as in our simple example above.

In summary, the general.h, helper1.h, and helper2.h header files contain definitions that are made public to files including them. We might see the following items in the helper1.h header file, for example:

- an include guard
- other include files
- constants and macros made public (and which may also be used by helper1.c)
- new data types (which may also be used by helper1.c)
- function prototypes of those functions in helper1.c which are meant to be used by other files
- possibly global variables that are *defined* (space allocated) in helper1.c but made available to other files by an extern *declaration* in helper1.h

If a variable, function prototype, or constant is private to one C file, you can just define and use it in that C file without including it in a header file. Also, global variables should not be defined (space allocated) in a header file which could be included by more than one C file in a project. A global should be defined in exactly one C file (e.g., int Global;), and then this variable can be made accessible by other files by adding the non-allocating declaration extern int Global; to a header file.

Makefiles

When you are ready to build your executable, you can type the gcc commands at the command line, as we have seen previously. A *makefile* simplifies the process, particularly for multi-file projects, by specifying the dependencies and commands needed to build the project. A makefile for our rad2volume example is shown below, where everything after a # is a comment.

```
# ***** file:  makefile *****
# Comment:  This is the simplest of makefiles!

# Here is a template:
# [target]:  [dependencies]
# [tab] [command to execute]

# The thing to the left of the colon in the first line is what is created,
# and the thing(s) to the right of the colon are what it depends on.  The second
# line is the action to create the target.  If the things it depends on
# haven't changed since the target was last created, no need to do the action.
# Note:  The tab spacing in the second line is important!  You can't just use
# individual spaces.

# "make myprog" or "make" links to create the executable
myprog:  main.o rad2volume.o
	gcc main.o rad2volume.o -o myprog

# "make main.o" produces main.o object code
```

```
main.o:  main.c rad2volume.h
        gcc -c main.c -o main.o

# "make rad2volume.o" produces rad2volume.o object code
rad2volume.o:  rad2volume.c rad2volume.h
        gcc -c rad2volume -o rad2volume.o

# "make clean" throws away any object files to ensure make from scratch
clean:
        rm *.o
```

With this `makefile` in the same directory as your other files, you should be able to type the command `make [target]`[16], where `[target]` is `myprog`, `main.o`, `rad2volume.o`, or `clean`. If the `target` depends on other files, `make` will make sure those are up to date first, and if not, it will call the commands needed to make them. For example, `make myprog` triggers a check of `main.o`, which triggers a check of `main.c` and `rad2volume.h`. If either of those have changed since the last time `main.o` was made, then `main.c` is compiled to create a new `main.o` before the linking step.

The command `make` with no target specified will make the first target (which is `myprog` in this case).

Make sure your `makefile` is saved without any extensions (e.g., `.txt`) and that the commands are preceded by a tab (not spaces).

There are many more sophisticated uses of makefiles which you can learn about from other sources.

[16]In some C installations `make` is named differently, like `nmake` for Visual Studio or `mingw32-make`. If you can find no version of `make`, you may not have selected the `make` tools installation option when you performed the C installation.

A.5 Exercises

1. Install C, create the `HelloWorld.c` program, and compile and run it.

2. Explain what a pointer variable is, and how it is different from a non-pointer variable.

3. Explain the difference between interpreted and compiled code.

4. Write the following hexadecimal (base-16) numbers in eight-digit binary (base-2) and three-digit decimal (base-10). Also, for each of the eight-digit binary representations, give the value of the most significant bit. (a) 0x1E. (b) 0x32. (c) 0xFE. (d) 0xC4.

5. What is 333_{10} in binary and 1011110111_2 in hexadecimal? What is the maximum value, in decimal, that a 12-bit number can hold?

6. Assume that each byte of memory can be addressed by a 16-bit address, and every 16-bit address has a corresponding byte in memory. How many total bits of memory do you have?

7. (Consult an ASCII table.) Let `ch` be of type `char`. (a) The assignment `ch = 'k'` can be written equivalently using a number on the right side. What is that number? (b) The number for `'5'`? (c) For `'='`? (d) For `'?'`?

8. What is the range of values for an `unsigned char`, `short`, and `double` data type?

9. How many bits are used to store a `char`, `short`, `int`, `float`, and `double`?

10. Explain the difference between `unsigned` and `signed` integers.

11. (a) For integer math, give the pros and cons of using `char`s vs. `int`s. (b) For floating point math, give the pros and cons of using `float`s vs. `double`s. (c) For integer math, give the pros and cons of using `char`s vs. `float`s.

12. The following `signed short int`s, written in decimal, are stored in two bytes of memory using the two's complement representation. For each, give the four-digit hexadecimal representation of those two bytes. (a) 13. (b) 27. (c) −10. (d) −17.

13. The smallest positive integer that cannot be exactly represented by a four-byte IEEE 754 `float` is $2^{24} + 1$, or 16,777,217. Explain why.

14. Technically the data type of a pointer to a `double` is "pointer to type `double`." Of the common integer and floating point data types discussed in this chapter, which is the most similar to this pointer type? Assume pointers occupy eight bytes.

15. To keep things simple, let's assume we have a microcontroller with only $2^8 = 256$ bytes of RAM, so each address is given by a single byte. Now consider the following code defining four local variables:

```
unsigned int i, j, *kp, *np;
```

Let's assume that the linker places `i` in addresses 0xB0..0xB3, `j` in 0xB4..0xB7, `kp` in 0xB8, and `np` in 0xB9. The code continues as follows:

```
              // (a) the initial conditions, all memory contents unknown
kp = &i;      // (b)
j = *kp;      // (c)
i = 0xAE;     // (d)
np = kp;      // (e)
*np = 0x12;   // (f)
j = *kp;      // (g)
```

For each of the comments (a)-(g) above, give the contents (in hexadecimal) at the address ranges 0xB0..0xB3 (the `unsigned int i`), 0xB4..0xB7 (the `unsigned int j`), 0xB8 (the pointer `kp`), and 0xB9 (the pointer `np`), at that point in the program, after executing the line containing the comment. The contents of all memory addresses are initially unknown or random, so your answer to (a) is "unknown" for all memory locations. If it matters, assume little-endian representation.

16. Invoking the `gcc` compiler with a command like `gcc myprog.c -o myprog` actually initiates four steps. What are the four steps called, and what is the output of each step?

17. What is `main`'s data type, and what is the meaning of its return value?

18. Give the `printf` statement that will print out a `double d` with eight digits to the right of the decimal point and four spaces to the left.

19. Consider three `unsigned chars`, `i`, `j`, and `k`, with values 60, 80, and 200, respectively. Let `sum` also be an `unsigned char`. For each of the following, give the value of `sum` after performing the addition. (a) `sum = i+j;` (b) `sum = i+k;` (c) `sum = j+k;`

20. For the variables defined as

```
int a=2, b=3, c;
float d=1.0, e=3.5, f;
```

give the values of the following expressions. (a) `f = a/b;` (b) `f = ((float) a)/b;` (c) `f = (float) (a/b);` (d) `c = e/d;` (e) `c = (int) (e/d);` (f) `f = ((int) e)/d;`

21. In each snippet of code in (a)-(d), there is an arithmetic error in the final assignment of `ans`. What is the final value of `ans` in each case?

   (a)
   ```
   char c = 17;
   float ans = (1 / 2) * c;
   ```
   (b) `unsigned int ans = -4294967295;`

   (c)
   ```
   double d = pow(2, 16);
   short ans = (short) d;
   ```
   (d) `double ans = ((double) -15 * 7) / (16 / 17) + 2.0;`

22. Truncation isn't always bad. Say you wanted to store a list of percentages rounded down to the nearest percent, but you were tight on memory and cleverly used an array of `chars` to store the values. For example, pretend you already had the following snippet of code:

```
char percent(int a, int b) {
  // assume a <= b
  char c;
  c = ???;
  return c;
}
```

You can't simply write `c = a / b`. If $\frac{a}{b} = 0.77426$ or $\frac{a}{b} = 0.778$, then the correct return value is `c` = 77. Finish the function definition by writing a one-line statement to replace `c = ???`.

23. Explain why global variables work against modularity.

24. What are the seven sections of a typical C program?

25. You've written a large program with a number of functions. Your program compiles without errors, but when you run the program with input for which you know the correct output, you discover that your program returns a wrong result. What do you do next? Describe your systematic strategy for debugging.

26. Erase all the comments in `invest.c`, recompile, and run the program to make sure it still functions correctly. You should be able to recognize what is a comment and what is not. Turn in your modified `invest.c` code.

27. The following problems refer to the program `invest.c`. For all problems, you should modify the original code (or the code without comments from the previous problem) and run it to make sure you get the expected behavior. For each problem, turn in the modified portion of the code only.

    (a) *Using* `if`*,* `break` *and* `exit`*.* Include the header file `stdlib.h` so we have access to the `exit` function (Section A.4.15). Change the `while` loop in `main` to be an infinite loop by inserting an expression `<expr>` in `while(<expr>)` that always evaluates to 1 (TRUE). (What is the simplest expression that evaluates to 1?) Now the first command inside the while loop gets the user's input. `if` the input is not valid, `exit` the program; otherwise continue. Next, change the `exit` command to a `break` command, and see the different behavior.

    (b) *Accessing fields of a* `struct`*.* Alter `main` and `getUserInput` to set `inv.invarray[0]` in `getUserInput`, not `main`.

    (c) *Using* `printf`*.* In `main`, before `sendOutput`, echo the user's input to the screen. For example, the program could print out `You entered 100.00, 1.05, and 5`.

    (d) *Altering a string.* After the `sprintf` command of `sendOutput`, try setting an element of `outstring` to 0 before the `printf` command. For example, try setting the third element of `outstring` to 0. What happens to the output when you run the program? Now try setting it to `'0'` instead and see the behavior.

    (e) *Relational operators.* In `calculateGrowth`, eliminate the use of `<=` in favor of an equivalent expression that uses `!=`.

    (f) *Math.* In `calculateGrowth`, replace `i=i+1` with an equivalent statement using `+=`.

    (g) *Data types.* Change the fields `inv0`, `growth`, and `invarray[]` to be `float` instead of `double` in the definition of the `Investment` data type. Make sure you make the correct changes everywhere else in the program.

    (h) *Pointers.* Change `sendOutput` so that the second argument is of type `int *`, i.e., a pointer to an integer, instead of an integer. Make sure you make the correct changes everywhere else in the program.

    (i) *Conditional statements.* Use an `else` statement in `getUserInput` to print `Input is valid` if the input is valid.

    (j) *Loops.* Change the `for` loop in `sendOutput` to an equivalent `while` loop.

    (k) *Logical operators.* Change the assignment of `valid` to an equivalent statement using `||` and `!`, and no `&&`.

28. Consider this array definition and initialization:

    ```
    int x[4] = {4, 3, 2, 1};
    ```

    For each of the following, give the value or write "error/unknown" if the compiler will generate an error or the value is unknown. (a) `x[1]` (b) `*x` (c) `*(x+2)` (d) `(*x)+2` (e) `*x[3]` (f) `x[4]` (g) `*(&(x[1]) + 1)`

29. For the (strange) code below, what is the final value of `i`? Explain why.

    ```
    int i,k=6;
    i = 3*(5>1) + (k=2) + (k==6);
    ```

30. As the code below is executed, give the value of `c` in hex at the seven break points indicated, (a)-(g).

```
unsigned char a=0x0D, b=0x03, c;
c = ~a;     // (a)
c = a & b;  // (b)
c = a | b;  // (c)
c = a ^ b;  // (d)
c = a >> 3; // (e)
c = a << 3; // (f)
c &= b;     // (g)
```

31. In your C installation, or by searching on the web, find a listing of the header file `stdio.h`. Find the function prototype for one function provided by the library, but not mentioned in this appendix, and describe what that function does.

32. Write a program to generate the ASCII table for values 33 to 127. The output should be two columns: the left side with the number and the right side with the corresponding character. Turn in your code and the output of the program.

33. We will write a simple *bubble sort* program to sort a string of text in ascending order according to the ASCII table values of the characters. A bubble sort works as follows. Given an array of n elements with indexes 0 to $n-1$, we start by comparing elements 0 and 1. If element 0 is greater than element 1, we swap them. If not, leave them where they are. Then we move on to elements 1 and 2 and do the same thing, etc., until finally we compare elements $n-2$ and $n-1$. After this, the largest value in the array has "bubbled" to the last position. We now go back and do the whole thing again, but this time only comparing elements 0 up to $n-2$. The next time, elements 0 to $n-3$, etc., until the last time through we only compare elements 0 and 1.

    Although this simple program `bubble.c` could be written in one function (`main`), we are going to break it into some helper functions to get used to using them. The function `getString` will get the input from the user; the function `printResult` will print the sorted result; the function `greaterThan` will check if one element is greater than another; and the function `swap` will swap two elements in the array. With these choices, we start with an outline of the program that looks like this.

```
#include <stdio.h>
#include <string.h>
#define MAXLENGTH 100       // max length of string input

void getString(char *str);  // helper prototypes
void printResult(char *str);
int greaterThan(char ch1, char ch2);
void swap(char *str, int index1, int index2);

int main(void) {
  int len;                  // length of the entered string
  char str[MAXLENGTH];      // input should be no longer than MAXLENGTH
  // here, any other variables you need

  getString(str);
  len = strlen(str);        // get length of the string, from string.h
  // put nested loops here to put the string in sorted order
  printResult(str);
  return(0);
}

// helper functions go here
```

    Here's an example of the program running. Everything after the first colon is entered by the user. Blank spaces are written using an underscore character, since `scanf` assumes that the string ends at the first whitespace.

```
Enter the string you would like to sort:  This_is_a_cool_program!
Here is the sorted string:  !T___aacghiilmoooprrss
```

Complete the following steps in order. Do not move to the next step until the current step is successful.

(a) Write the helper function `getString` to ask the user for a string and place it in the array passed to `getString`. You can use `scanf` to read in the string. Write a simple call in `main` to verify that `getString` works as you expect before moving on.

(b) Write the helper function `printResult` and verify that it works correctly.

(c) Write the helper function `greaterThan` and verify that it works correctly.

(d) Write the helper function `swap` and verify that it works correctly.

(e) Now define the other variables you need in `main` and write the nested loops to perform the sort. Verify that the whole program works as it should.

Turn in your final documented code and an example of the output of the program.

34. A more useful sorting program would take a series of names (e.g., `Doe_John`) and scores associated with them (e.g., 98) and then list the names and scores in two columns in descending order. Modify your bubble sort program to do this. The user enters a name string and a number at each prompt. The user indicates that there are no more names by entering `0 0`.

    Your program should define a constant `MAXRECORDS` which contains the maximum number of records allowable. You should define an array, `MAXRECORDS` long, of `struct` variables, where each `struct` has two fields: the name string and the score. Write your program modularly so that there is at least a `sort` function and a `readInput` function of type `int` that returns the number of records entered.

    Turn in your code and example output.

35. Modify the previous program to read the data in from a file using `fscanf` and write the results out to another file using `fprintf`. Turn in your code and example output.

36. Consider the following lines of code:

```
int i, tmp, *ptr, arr[7] = {10, 20, 30, 40, 50, 60, 70};

ptr = &arr[6];
for(i = 0; i < 4; i++) {
  tmp = arr[i];
  arr[i] = *ptr;
  *ptr = tmp;
  ptr--;
}
```

(a) How many elements does the array `arr` have?

(b) How would you access the middle element of `arr` and assign its value to the variable `tmp`? Do this two ways, once indexing into the array using `[]` and the other with the dereferencing operator and some pointer arithmetic. Your answer should only be in terms of the variables `arr` and `tmp`.

(c) What are the contents of the array `arr` before and after the loop?

37. The following questions pertain to the code below. For your responses, you only need to write down the changes you would make using valid C code. You should verify that your modifications actually compile and run correctly. Do not submit a full C program for this question. Only write the changes you would make using legitimate C syntax.

```
#include <stdio.h>
#define MAX 10

void MyFcn(int max);

int main(void) {
  MyFcn(5);
  return(0);
}

void MyFcn(int max) {
  int i;
  double arr[MAX];

  if(max > MAX) {
    printf("The range requested is too large.  Max is %d.\n", MAX);
    return;
  }
  for(i = 0; i < max; i++) {
    arr[i] = 0.5 * i;
    printf("The value of i is %d and %d/2 is %f.\n", i, i, arr[i]);
  }
}
```

(a) `while` loops and `for` loops are essentially the same thing. How would you write an equivalent `while` loop that replicates the behavior of the `for` loop?

(b) How would you modify the `main` function so that it reads in an integer value from the keyboard and then passes the result to `MyFcn`? (This replaces the statement `MyFcn(5);`.) If you need to use extra variables, make sure to define them before you use them in your snippet of code.

(c) Change `main` so that if the input value from the keyboard is between −`MAX` and `MAX`, you call `MyFcn` with the absolute value of the input. If the input is outside this range, then you simply call `MyFcn` with the value `MAX`. How would you make these changes using conditional statements?

(d) In C, you will often find yourself writing nested loops (a loop inside a loop) to accomplish a task. Modify the `for` loop to use nested loops to set the ith element in the array `arr` to half the sum of the first $i-1$ integers, i.e., `arr[i]` $= \frac{1}{2}\sum_{j=0}^{i-1} j$. (You can easily find a formula for this that doesn't require the inner loop, but you should use nested loops for this problem.) The same loops should print the value of each `arr[i]` to 2 decimal places using the `%f` formatting directive.

38. If there are n people in a room, what is the chance that two of them have the same birthday? If $n = 1$, the chance is zero, of course. If $n > 366$, the chance is 100%. Under the assumption that births are distributed uniformly over the days of the year, write a program that calculates the chances for values of $n = 2$ to 100. What is the lowest value n^* such that the chance is greater than 50%? (The surprising result is sometimes called the "birthday paradox.") If the distribution of births on days of the year is not uniform, will n^* increase or decrease? Turn in your answer to the questions as well as your C code and the output.

39. In this problem you will write a C program that solves a "puzzler" that was presented on NPR's CarTalk radio program. In a direct quote of their radio transcript, found here `http://www.cartalk.com/content/hall-lights?question`, the problem is described as follows:

> **RAY**: This puzzler is from my "ceiling light" series. Imagine, if you will, that you have a long, long corridor that stretches out as far as the eye can see. In that corridor, attached to the ceiling are lights that are operated with a pull cord.
>
> There are gazillions of them, as far as the eye can see. Let's say there are 20,000 lights in a row.

> They're all off. Somebody comes along and pulls on each of the chains, turning on each one of the lights. Another person comes right behind, and pulls the chain on every second light.
>
> **TOM**: Thereby turning off lights 2, 4, 6, 8 and so on.
>
> **RAY**: Right. Now, a third person comes along and pulls the cord on every third light. That is, lights number 3, 6, 9, 12, 15, etc. Another person comes along and pulls the cord on lights number 4, 8, 12, 16 and so on. Of course, each person is turning on some lights and turning other lights off.
>
> If there are 20,000 lights, at some point someone is going to come skipping along and pull every 20,000th chain.
>
> When that happens, some lights will be on, and some will be off. Can you predict which lights will be on?

You will write a C program that asks the user the number of lights n and then prints out which of the lights are on, and the total number of lights on, after the last (nth) person goes by. Here's an example of what the output might look like if the user enters 200:

```
How many lights are there? 200

You said 200 lights.
Here are the results:
  Light number   1 is on.
  Light number   4 is on.
  ...
  Light number 196 is on.
  There are 14 total lights on!
```

Your program `lights.c` should follow the template outlined below. Turn in your code and example output.

```
/***********************************************************************
 * lights.c
 *
 * This program solves the light puzzler.  It uses one main function
 * and two helper functions:  one that calculates which lights are on,
 * and one that prints the results.
 *
 ***********************************************************************/

#include <stdio.h>
#include <stdlib.h>         // allows the use of the "exit()" function
#define MAX_LIGHTS 1000000  // maximum number of lights allowed

// here's a prototype for the light toggling function
// here's a prototype for the results printing function

int main(void) {

  // Define any variables you need, including for the lights' states

  // Get the user's input.
  // If it is not valid, say so and use "exit()" (stdlib.h, Sec 1.2.16).
  // If it is valid, echo the entry to the user.

  // Call the function that toggles the lights.
  // Call the function that prints the results.
```

```
  return(0);
}

// definition of the light toggling function
// definition of the results printing function
```

40. We have been preprocessing, compiling, assembling, and linking programs with commands like

```
gcc HelloWorld.c -o HelloWorld
```

The `gcc` command recognizes the first argument, `HelloWorld.c`, is a C file based on its `.c` extension. It knows you want to create an output file called `HelloWorld` because of the `-o` option. And since you didn't specify any other options, it knows you want that output to be an executable. So it performs all four of the steps to take the C file to an executable.

We could have used options to stop after each step if we wanted to see the intermediate files produced. Below is a sequence of commands you could try, starting with your `HelloWorld.c` code. Don't type the "comments" to the right of the commands!

```
> gcc HelloWorld.c -E > HW.i  // stop after preprocessing, dump into file HW.i
> gcc HW.i -S -o HW.s         // compile HW.i to assembly file HW.s and stop
> gcc HW.s -c -o HW.o         // assemble HW.s to object code HW.o and stop
> gcc HW.o -o HW              // link with stdio printf code, make executable HW
```

At the end of this process you have `HW.i`, the C code after preprocessing (`.i` is a standard extension for C code that should not be preprocessed); `HW.s`, the assembly code corresponding to `HelloWorld.c`; `HW.o`, the unreadable object code; and finally the executable code `HW`. The executable is created from linking your `HW.o` object code with object code from the `stdio` (standard input and output) library, specifically object code for `printf`.

Try this and verify that you see all the intermediate files, and that the final executable works as expected.

If our program used any math functions, the final linker command would be

```
> gcc HW.o -o HW -lm          // link with stdio and math libraries, make executable HW
```

Most libraries, like `stdio`, are linked automatically, but often the math library is not, requiring the extra `-lm` option.

The `HW.i` and `HW.s` files can be inspected with a text editor, but the object code `HW.o` and executable `HW` cannot. We can try the following commands to make viewable versions:

```
> xxd HW.o v1.txt             // can't read obj code; this makes viewable v1.txt
> xxd HW v2.txt               // can't read executable; make viewable v2.txt
```

The utility `xxd` just turns the first file's string of 0's and 1's into a string of hex characters, represented as text-editor-readable ASCII characters 0..9, A..F. It also has an ASCII sidebar: when a byte (two consecutive hex characters) has a value corresponding to a printable ASCII character, that character is printed. You can even see your message "Hello world!" buried there!

Take a quick look at the `HW.i`, `HW.s`, and `v1.txt` and `v2.txt` files. No need to understand these intermediate files any further. If you don't have the `xxd` utility, you could create your own program `hexdump.c` instead:

```
#include <stdio.h>
#define BYTES_PER_LINE 16

int main(void) {
  FILE *inputp, *outputp;                           // ptrs to in and out files
  int c, count = 0;
  char asc[BYTES_PER_LINE+1], infile[100];

  printf("What binary file do you want the hex rep of? ");
  scanf("%s",infile);                               // get name of input file
  inputp = fopen(infile,"r");                       // open file as "read"
  outputp = fopen("hexdump.txt","w");               // output file is "write"

  asc[BYTES_PER_LINE] = 0;                          // last char is end-string
  while ((c=fgetc(inputp)) != EOF) {                // get byte; end of file?
    fprintf(outputp,"%x%x ",(c >> 4),(c & 0xf));   // print hex rep of byte
    if ((c>=32) && (c<=126)) asc[count] = c;       // put printable chars in asc
    else asc[count] = '.';                          // otherwise put a dot
    count++;
    if (count==BYTES_PER_LINE) {                    // if BYTES_PER_LINE reached
      fprintf(outputp,"  %s\n",asc);                // print ASCII rep, newline
      count = 0;
    }
  }
  if (count!=0) {                                   // print last (short) line
    for (c=0; c<BYTES_PER_LINE-count; c++)          // print extra spaces
      fprintf(outputp,"   ");
    asc[count]=0;                                   // add end-string char to asc
    fprintf(outputp,"  %s\n",asc);                  // print ASCII rep, newline
  }
  fclose(inputp);                                   // close files
  fclose(outputp);
  printf("Printed hexdump.txt.\n");
  return(0);
}
```