C/Unix Programmer's Guide Practice Key

July 17, 2024

Jason W. Bacon

Contents

July 17, 2024

0.1 Using this key

This key to the practice problems is provided to allow students to immediately *check their own work*. Do not look at this answer key before writing down your own answers. In doing so, you would only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams. Writing things in your own words vastly improves your understanding.

0.2 Practice Problem Instructions

- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest version of this document.
- Read one section of this document and corresponding materials if applicable.
- Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
- Do the practice problems *on your own*. Do not discuss them with other students. If you want to help each other, discuss *concepts* and illustrate with different examples if necessary. Coming up with the correct answer on your own is the only way to be sure you understand the material. If you do the practice problems on your own, you will succeed in the subject. If you don't, you won't.

  If you're still not clear after doing the practice problems, wait a while and do them again. This is how athletes perfect their game. The same strategy works for any skill.
- Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Remove all words that don't add value to the explanation. Brevity and clarity are the most important aspects of good communication. Unnecessarily lengthy answers are often an attempt to obscure a lack of understanding and may lead to reduced grades. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Check the answer key to make sure your answer is correct and complete.

  DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE BEST OF YOUR ABILITY. In doing so, you only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.
- ALWAYS explain your answer. No exceptions. E.g., justify all yes/no or other short answers, show your work or indicate by other means how you derived your answer for any question that involves a process, no matter how trivial it may seem, draw a diagram to illustrate if necessary. This will improve your understanding and ensure full credit for the homework.
- Verify your own results by testing all code written, and double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.
- For programming questions, adhere to all coding standards as defined in the text, e.g. descriptive variable names, consistent indentation, etc.

Chapter 1

Introduction

1.1 How to Proceed

1.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why do we need a book when we can just learn Unix and C functions from the man pages?

   The man pages are written as highly technical references, not as tutorials. They are often difficult for beginners to understand.

2. What is the best way to utilize the examples in the book?

   Type them in and run them. Even better, modify them or write similar programs. The more you play, the more you learn!

3. Where does proficiency come from?

   From *doing*.

1.2 Why use C?

1.2.1 Language Performance

1.2.2 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How much of the original Unix operating system was written in C?

   About 90%.

2. List 5 advantages of C over other high-level languages.

   - Fastest of all portable languages
   - Literally any program can be written in C, ranging from those we might otherwise do in assembly language to complex applications.
   - C is the most portable language every created. We can program virtually any device with a CPU in C.

- C is small and flexible, so it is easy to master and extensible via library functions.
- C is popular, so there are many resources available to help us program in C.

3. Is C too low-level for application programming?

   No. Most features of high-level languages that are not present in C are usually provided by library functions, which are about as easy to use.

4. Can you do object-oriented programming in C?

   Yes. OOP can be done in any language.

5. How fast is a C program compared to the same program design implemented in an interpreted language such as Python, Perl, or R?

   The C program will be on the order of 100 times as fast on average. The exact ratio varies greatly, depending on the exact algorithms used and how many compiled functions the interpreted program utilizes.

6. Why is C so much simpler than other high-level languages?

   Most functionality is provided by libraries rather than compiler features.

7. Is it enough that a program runs fast enough on your computer? Why or why not?

   No. Others may run it on slower hardware or on systems running other programs at the same time.

8. What is the relationship between C and C++?

   C++ is a superset of C, i.e. C++ includes (almost) all the features of C.

9. Does C++ enforce object-oriented programming?

   No. It only provides features that make it convenient.

1.3 Why use Unix?

1.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Is Unix an operating system?

   No. It was originally, but is now a set of standards to which operating systems conform.

2. Which mainstream operating systems are Unix compatible?

   All of them, except MS Windows.

3. If you write a program on a Linux system with an Intel x86 processor, how hard will it be to port the program to FreeBSD on an ARM processor?

   It will be trivial and likely won't require any changes at all.

4. What is POSIX?

   Portable Operating System Standard based on Unix. An official set of standards to which Unix systems conform.

5. What types of devices run Unix?

   Everything from small embedded systems and cell phones to supercomputers.

6. How much does a Unix license typically cost?

   While there are some commercial Unix operating systems, many of them are free and open source.

7. What is the major benefit of using Unix and C together?

   You will never be dependent on a given OS or compiler vendor.

8. How does the Unix kernel make it possible for programs written on a PC to run on a RISC workstation or server?

   The kernel fully encapsulates the hardware, so programs are not hardware-specific. Programs only see the kernel interface, which is the same regardless of underlying hardware.

1.4 Addendum: What is Systems Programming?

Part I

Introduction to Computers and Unix

Chapter 2

Binary Information Systems

2.1 Why do I need to know this stuff?

2.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why is it important for programmers to understand number systems?

   Understanding the range and accuracy limitations of computer number systems is crucial to writing correct and efficient programs.

2.2 Representing Information in Binary

2.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is binary?

   A way of representing information using only two possible states.

2. How do we represent binary information on paper?

   Using 0 and 1 to represent the two different states.

3. How is binary represented inside a computer?

   Using two different voltages, such as 0V and 5V.

4. Why do computers use binary rather than decimal?

   Simplicity. It's easier to build circuits that use 2 possible states than circuits that use 10 possible states.

2.3 The Usual Jargon

2.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Define each of the following:

   (a) Bit
   Binary digit. 0 or 1. 0V or 5V.

   (b) Byte
   8 bits.

   (c) Word
   The number of bits a computer can process at once. Usually 16, 32, or 64.

   (d) Longword
   A fuzzy term. Usually at least 32 bits. Maybe be the same as a word, or twice as many bits.

   (e) Shortword
   16 bits.

   (f) Nybble
   4 bits.

2.4 Binary Number Systems

2.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the differences between binary numbers and decimal numbers?

   Base of 2 instead of 10. Digits 0 and 1 instead of 0 through 9.

2. What do binary and decimal number systems have in common?

   Both are examples of the Arabic numeral system.

3. What is the main difference between the abstract number sets we use in mathematics and the number sets represented in computers?

   Abstract sets have infinite range, computers have limited range, so they are a subset of the abstract number sets.

4. What is a fixed point number system?

   A number system with a fixed number of whole and fractional digits.

5. What is an integer number system?

   A fixed point system with no fractional digits.

6. What is a floating point system?

   A system that can use a flexible number of whole and fractional digits.

7. What is a major disadvantage to floating point as compared to integers in computer systems?

   Floating point operations take about three times as long.

8. How are floating point systems implemented in computers?
   Using a format similar to scientific notation.
9. What are the three parts of a floating point number?
   Mantissa, radix, exponent.
10. What steps are necessary to add two floating point numbers?
    - Equalize the exponents
    - Add the mantissas
    - Normalize the result

2.5 Binary Fixed Point and Binary Integers

2.5.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the decimal value of 1101.1_2?
   13.5_10
2. What is the binary value of 49.125_10?
   110001.001_2
3. What are LSB and MSB?
   Least significant byte, most significant byte.
4. What is the LSB in 100010?
   0
5. What is the MSB in 010111?
   0
6. What is the range of a 7-digit unsigned decimal number system?
   0 to 9999999_10
7. What is the range of a 10-bit unsigned binary number system? Show your answer in binary, in powers of 2, and in decimal.
   0000000000 to 1111111111, 0 to 2^10 - 1, 0 to 1023
8. How many kinds of people are there in the world?
   10

2.6 Binary Arithmetic

2.6.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. (a) What is 1100_2 + 0111_2 in 4-bit binary?

       0011

   (b) What is the decimal value of the 4-bit sum?

       3

   (c) Is there an overflow? Explain in terms of the resulting bits and in terms of the resulting value.

       Yes. Carry out of the leftmost bit. 4-bit answer is smaller than one of the addends.

2. Explain multiple precision addition.

   Adding two numbers larger than the word size of the computer. First add the lower words, then add the next higher words + the carry bit from the lower words.

3. What is the down side of multiple precision arithmetic?

   It requires multiple instructions to complete, whereas arithmetic at or below the word size can be done in one instruction. Hence, it is much slower.

4. Represent +8 and -3 in 8-bit twos complement.

   00001000, 11111101

5. What are the decimal values of the 8-bit twos complement numbers 11111111 and 00001111?

   -1, +15

6. How many different unsigned integers can we represent with N bits?

   2^N

7. How many different signed integers can we represent with N bits?

   2^N

8. How many different floating point values can we represent with N bits?

   2^N

2.7 Signed Integers

2.7.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the range of a 12-bit twos complement system? Show your answer in binary, as powers of 2, and as decimal values.

   100000000000 to 011111111111, -2^11 to +2^11 - 1, -2048 to +2047

2. Which of the common integer sizes are sufficient to represent Avogadro's constant?

   None of them. Even a 64 bit unsigned integer only reaches 10^19.

3. What's the difference between unsigned addition and twos complement addition?

   They are exactly the same, except for how we detect overflow.

4. Add the following 8-bit twos comp values 01001000 and 01010011. Show results in twos comp and in decimal. Indicate whether an overflow occurs. Explain in terms of the bits computed and the value of the result.

   10011011_2 = -101_10. Yes, an overflow occurred. The sign is wrong as indicated by the leftmost bit. Adding two positives should not result in a negative.

2.8 Floating Point

2.8.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Does floating point allow us to represent more different values than integer systems with the same number of bits? Explain.
   No. With N bits, we can represent exactly 2^N different values, regardless of the format.
2. What are two advantages of floating point over integers?
   1. Support for fractional values. 2. Can represent much larger values than an integer with the same number of bits.
3. What are two disadvantages of floating point vs integers?
   1. Less precise than integers. 2. Arithmetic operations take about 3 times as long.

2.9 Floating Point Range and Precision

2.9.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the properties of floating point that programmers need to understand?
   Largest positive value, smallest positive value, smallest negative value, largest negative value, precision.
2. Which has a greater magnitude, the smallest negative value or the largest negative value?
   Smallest negative.
3. What is precision and of what is it a property?
   The number of significant figures that can be stored. It is a property of the number system (or device).
4. What is accuracy and of what is it a property?
   A measure of how close a value is to reality. It is a property of each particular value.
5. Define overflow.
   The true result of an operation exceeds the largest positive or smallest negative value that can be represented.
6. Define underflow.
   The true result of an operation is between the largest negative and smallest positive values that can be represented.
7. Define round-off error.
   The true result of an operation is altered to the nearest value that can be represented.
8. Define truncation error.
   The true result is adjusted to the next lower value that can be represented.
9. Which of the common floating point sizes can represent Avogadro's constant?
   All of them. The smallest common size, 32 bits, has a range up to about 10^38.

2.10 Other Number Systems (Bases)

2.10.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Convert F841.5_16 to binary.
   1111100001000001.0101_2
2. Convert 10010010100111.001_2 to hexadecimal.
   24A7.2_16
3. Convert 673.1_8 to binary.
   110111011.001_2
4. Convert 1011001011010011.01_2 to octal.
   131323.2_8
5. Convert 789.52_8 to binary.
   Invalid digits 8 and 9.

2.11 Character Representation

2.11.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the limitations of ASCII?
   It only supports English characters.
2. What is ISO?
   The International Standards Organization.
3. How are ISO character sets related to ASCII?
   The first 128 characters are the same. Characters 128 to 255 represent non-English characters and other extensions to ASCII.
4. What is EBCDIC?
   An obsolete character set that preceded ASCII and ISO sets, used on mainframes.
5. Are ASCII and ISO codes numbers?
   No, they are just bit patterns, but are often written as decimal numbers to avoid writing so many binary digits.

Chapter 3

Hardware and Software

3.1 What Makes Computers Tick?

3.2 The Main Components

3.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a CPU? What is a core? What is the relationship between them?

   CPU = either physical chip or logical unit that executes instructions. A single CPU chip usually contains multiple cores.

2. What are volatile and non-volatile storage?

   Volatile storage needs power to retain its contents, non-volatile does not.

3. What is ROM? Is it volatile or non-volatile? What is it used for?

   Read-only Memory. It is non-volatile. All computers need some ROM to contain startup instructions for when they are powered on.

4. What is RAM? What might be a better name for it and why?

   Random Access Memory. RWM might be better, since ROM is also randomly accessible.

5. What are I/O devices for?

   Generally, for communicating with humans.

6. What is mass storage for? What are three common types of mass storage?

   Long-term non-volatile storage, for holding data when the computer is not powered on. Magnetic disk, tape, and flash memory such as SSDs and USB thumb drives.

3.3 Programs and Programming Languages

3.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is machine language?

   Binary codes that cause a CPU to generate electronic signals that drive the rest of the computer.

2. What is assembly language?

   For the most part, a symbolic form of machine language, with some extra goodies for convenience.

3. What are two major drawbacks to programming in assembly language?

   - It is architecture-specific, i.e. it is not portable.
   - Instructions are primitive, so the programs are very long.

4. What does an assembler do? How complex is it and why?

   It converts assembly language to machine language. It is very simple, since assembly language is basically the same as machine language, and can be translated one line at a time.

5. What is a high-level language?

   A more intuitive and portable language that is much easier to use than assembly language.

6. What does a compiler do? How complex is it and why?

   It converts high level language source code to machine code. A compiler is very complex, since it has to convert high-level constructs such as loops and subprograms to sequences of machine instructions. This cannot be done one line at a time.

7. What does an interpreter do? How does it differ from a compiler.

   It reads source code and executes it directly, without ever outputting equivalent machine language as a compiler does.

8. Which will run faster, and by how much, a compiled program or an equivalent interpreted program?

   The compiled program will run orders of magnitude faster, since the complex task of parsing the source code is done before it starts running. With an interpreted language, parsing occurs while it is running.

3.4 The Programming Process

3.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the relationship between algorithms and programs? Describe an example of each.

   A program is an implementation of an algorithm. Selection sort and quicksort are algorithms that can be implemented in any programming language, and in many ways in a given language.

2. Describe stepwise refinement.

   Stepwise refinement is the process of breaking down a problem incrementally, each time into only a few components. It is then applied recursively to each component until the resulting components are trivial to implement.

3. What is a top-down design?

   It is a representation of the results of stepwise refinement.

4. What is a flowchart?

   It is a graphical representation of one level of refinement.

3.5 Engineering Product Life Cycle

3.5.1 Specification

3.5.2 Design

3.5.3 Implementation and Testing

3.5.4 Production

3.5.5 Support and Maintenance

3.5.6 Hardware Only: Disposal

3.5.7 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Briefly describe the six stages of the engineering product life cycle.
   - Specification: Clearly define what is to be accomplished.
   - Design: Clearly describe a plan for accomplishing the specified goals, without drifting into implementation details.
   - Implementation: Build the product (or a prototype of the product) closely following the design.
   - Testing: Thoroughly test the product throughout the implementation stage and the life of the product.
   - Maintenance: Fix bugs and update the product to maintain its usefulness.
   - Disposal: For hardware products, define what should happen to the product when it reaches the end of its useful life.
2. Describe the three major types of testing and when they occur.
   - Incremental testing: Thoroughly testing each added component during the implementation process.
   - Alpha testing: In-house testing by developers and/or other project members after implementation is complete.
   - Beta testing: Testing by real customers after alpha testing and before product release to the general public. This should not reveal any major problems, but may often lead to improvements in the user interface.
3. What is the most likely reason if someone is having a hard time figuring out what code to write for a new section of a program?

   Inadequate design. The design does not clearly explain how the program should work, i.e. it does not spell out the algorithm(s) that need to be coded. If the design is complete, implementation should be easy.
4. What is the most likely reason someone is having a hard time locating a bug in some code they just wrote?

   Insufficient incremental testing. They wrote too much new code without testing frequently along the way.

Chapter 4

Unix Overview: Enough to Make You Dangerous

4.1 What is an Operating System?

4.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the major components of an operating system?
   - The kernel is a library of functions that control the hardware.
   - The bootstrap system is software that is initially loaded by the BIOS/firmware and completes the loading of the system.
   - The user interface provides a way for people to interact with the system.
   - Utility (userland) programs provide basic functionality for getting work done. They often eliminate the need to write new programs.
2. What is a protected mode operating system and where is it essential?

   One that prevents processes from accessing each others' memory and other resources. It is essential for any multiuser and/or multitasking OS.

4.2 Unix Operating Systems (Was "The Unix Operating System")

4.2.1 The Cost of Complexity

4.2.2 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the purpose of the TERM environment variable?

   It allows programs to properly control the terminal screen and interpret keyboard input.
2. What is a shell?

   A program that implements a command-line-interface.

3. What is a GUI?

   A graphical user interface, a type of menu interface using a graphical display with icons as well as text menus.

4. What is an advantage of a CLI over a GUI?

   It provides instant access to unlimited functionality, which would require a cumbersome system of submenus in a GUI.

5. What are the components of a Unix command and how are they recognized?

   - The command name is either the filename of a program or a command built into the shell. It is the first part of every command.
   - Flag arguments are keywords or characters recognized by the command that tell it how to behave. They almost always begin with a '-'.
   - Data arguments are actual data or the names of filesystem objects containing the data. They should never begin with a '-'.

6. What separates command components?

   Whitespace (spaces or tabs).

7. What is a process?

   The execution of a program.

4.3 The Unix File-system

4.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a file on a Unix system?

   A sequence of bytes on some storage medium, nothing more.

2. What is a filesystem?

   A system for organizing files, directories, and other objects on a disk partition.

3. What is a directory?

   A special type of file that holds the names and locations of all files and other objects in a filesystem. Sometimes called a "folder" as an analogy to a paper filing system.

4. What is an absolute/full pathname of a filesystem object? Give an example.

   The complete path from the root directory to the object. /usr/home/joe.

5. How do we recognize an absolute pathname?

   It begins with a '/' or a '~'.

6. What is a home directory?

   A directory assigned to each user, where most or all of their files are stored.

7. What is a CWD and what is it a property of?

   A current working directory is the absolute pathname of a directory prepended to all non-absolute pathnames. It is a property of all Unix processes and can be thought of as the directory a process is currently "in".

8. If the CWD is /home/joe, what is the absolute pathname of "Programs/prog1.c"?

   /home/joe/Programs/prog1.c

9. How can you change the CWD of your shell process to /usr/local/bin?

   cd /usr/local/bin

10. How can you change the CWD of your shell process to your home directory?

    cd or **cd ~** or **cd ~/**.

11. How can you rename a file called "-prog2.c" to "prog2.c", given that a '-' indicates a flag argument? (Use the **mv** command.)

    mv ./prog2.c

12. How can you change the CWD of your shell process to the parent of the CWD?

    cd ..

13. How can you see the permissions on all the files in /etc?

    ls -l /etc

14. How can you change the permissions on the directory Programs so that members of the group can read it, but nobody else can?

    chmod g+rx,o-rwx Programs

4.4 The Shell Environment

4.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why are arguments containing whitespace problematic and how do we get around the problem?

   Whitespace is a separator for Unix command, so the shell thinks that an argument containing whitespace is actually multiple separate arguments. We can work around it by enclosing the argument in single or double quotes, or by escaping every whitespace character (precede it with a '\').

2. What are internal commands?

   Commands that are part of the shell and do not require a new process to run.

3. What are external commands?

   Filenames of programs separate from the shell, that are run under a separate "child" process.

4.5 Getting Help

4.5.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What suffering did our forefathers have to endure in order to read documentation? How was this injustice finally eradicated?

   They had to get out of their chairs to go get the printed manual. This injustice was ended when the man page (online documentation) was invented.

2. How can you find out what arguments are required by the **strcmp()** function?
   man strcmp
3. What are man pages generally good for and not so good for?
   Good as references to look up details on a command or function. Not so good as tutorials.
4. How can you find out what command or functions are available related to computing the sine of an angle?
   apropos sine
5. How can you find out what keystrokes control the viewer while reading a man page?
   Press 'h' for the help screen.
6. How can you use man pages to discover related commands and functions?
   Check the "SEE ALSO" section of each man page.

4.6 Some Useful Commands

4.6.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What Unix command could you use for each of the following tasks? Just name the command, no explanation needed.
   - Copy files
     cp
   - Move or rename files
     mv
   - View a text file on screen at a time
     more
   - View the first N lines of a file
     head
   - View the last N lines of a file
     tail
   - Search a file for strings or patterns
     grep
   - Sort a text file line-by-line
     sort
   - Create a directory
     mkdir
   - Reformat a C program
     indent
   - Combine files into an archive
     tar
   - Display a calendar for this month
     cal

4.7 A Few Shortcuts with T-shell and Bash

4.7.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How can you see a list of recently executed commands?

   history

2. How can you execute the most recent command beginning with "cc"?

   !cc

3. How can you scroll back through the last few commands?

   Up and down arrow keys.

4. How can you remove all the files in the CWD with names ending in ".core"?

   rm *.core

4.8 Unix Input and Output

4.8.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the standard streams used by all Unix processes and to what device are they normally connected?

   - The standard input, normally connected to the terminal keyboard.
   - The standard output is used for normal output and is normally connected to the terminal screen.
   - The standard error is used for error and warnings, and is also normally connected to the terminal screen.

2. Is it correct to say that a Unix program takes input from the keyboard? Why or why not?

   No. Programs connect to streams or file descriptors, not hardware devices. Programs don't usually know what device or file is on the other end of the connection.

3. What is redirection?

   Disconnecting a stream from one device or file and connecting it to a different one.

4. Show a Unix command that writes the last 10 lines of the file input1.txt to the file input1-last-10.txt.

   tail input1.txt > input1-last-10.txt

5. Show a Unix command that runs the program prog1, which is in the CWD, and reads input from the standard input, using input1.txt as input, and shows the output of the program one screen at a time.

   ./prog1 < input1.txt | more

6. What is device independence?

   A property of Unix that treats all I/O devices the same as an ordinary file.

4.9 Job Control

4.9.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the difference between a foreground process and a background process?

   A foreground process receives input from the keyboard.

2. How can you terminate the foreground process?

   Type Ctrl+c.

3. How can you pause the foreground process, list the contents of the CWD, and then resume the process in the foreground?

   ```
   Ctrl+z
   shell-prompt: ls
   shell-prompt: fg
   ```

4. How can you run a ./prog1 so that is immediately runs in the background and you can continue using the shell for other commands?

   ```
   ./prog1 < input.txt >& output.txt &
   ```

4.10 Shell Variables and Environment Variables

4.10.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a shell variable?

   A string variable used by a shell process.

2. Can we use any name we want for a shell variable?

   No, some are reserved for special purposes.

3. What is an environment variable?

   A string variable like a shell variable, but it is inherited by child processes. This is a way to communicate information to new processes that are created.

4. Are environment variables always created by a shell process?

   No, they are a property of every Unix process.

5. How does the shell find the programs that constitute external commands?

   It checks the colon-separated list of directories in the `PATH` environment variable.

4.11 Shell Scripts

4.11.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a shell script?

   A text file containing a shebang line and a sequence of Unix commands.

2. What Unix feature makes shell scripts possible?

   Device independence. Since a keyboard and a file are basically the same thing, the shell doesn't care which its input comes from.

3. Which of the many Unix shells is best for writing portable scripts?

   The POSIX Bourne shell, since it is standardized across all Unix systems and every Unix system has it.

4. Is the best shell for scripts also the best for interactive use?

   No, many shells have better interactive features than the standard Bourne shell.

5. What should the shebang line look like for a Bourne shell script?

   #!/bin/sh -e

6. What should the shebang line look like for a bash script?

   #!/usr/bin/env bash

4.12 Advanced: Make

4.12.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What does make do?

   It compares time stamps on a source and target file, and runs the given command if the source file is newer.

2. What can make be used for?

   Anything where one file is generated from the content of another file.

3. What is make commonly used for?

   Compiling programs from multiple source files.

Part II

Programming in C

Chapter 5

Getting Started with C and Unix

5.1 What is C?

5.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Is C a high-level language or a low-level language?

   It is both. It has all of the most important high-level features (flow control, subprograms, type definitions, etc) while also offering low-level capabilities and performance.

2. Compare C and C++.

   C is a subset of C++. C is a very simple language that relies on library functions for most functionality. C++ is a very complex language with many features built into the compiler.

3. How long should it take to master C? C++?

   C: One semester. C++: Several semesters.

5.2 C Program Structure

5.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the major components of a C program?

   (See text)

2. What is a free-format language?

   One where the end of a line does not mark the end of a statement. Statements can span multiple lines and lines can contain multiple statements.

5.3 A Word about Performance

5.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How does the performance of C programs compare to the same algorithm/design implemented in other languages?

   C is generally the fastest and most memory efficient. It is comparable to assembly language, usually faster than other compiled languages, and orders of magnitude faster than interpreted languages.

2. What are the major factors in program performance, from most to least important?

   See list in text.

3. What is profiling and how does it help us with program performance?

   Profiling is identifying where programs spend most of their time. It helps us by focusing our valuable time where it matters most. Optimizing code that accounts for 0.1% of the run time is usually a waste of man hours.

5.4 Some Early Warnings

5.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why does C not check for run time errors by default?

   These checks require additional machine instructions and therefore slow down programs considerably. C leaves it to the programmer to decide which checks are necessary.

5.5 Coding and Compiling a C Program

5.5.1 Compilation Stages

5.5.2 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is an IDE?

   Integrated Development Environment, a text editor specialized for writing programs.

2. What are some advantages of IDEs over simple text editors?

   IDEs allow the user to easily compile and run a program without leaving the editor. They also offer features such as syntax colorization to make the code more readable and flag typos immediately.

3. What are the stages in building an executable from a C source file?

   See text.

4. What command should usually be used to compile C programs on a Unix system?

   Use cc, because it is portable. Using gcc or clang explicitly generally serves no purpose and is not portable. FreeBSD and macOS use clang, while Linux uses gcc, for example.

5. What is the safe way to run the object code optimizer so that debuggers will work and the executable will run on all related CPUs?

   Just use -O or -O2. Other optimizations may break the source code map or generate non-portable machine instructions.

6. How do we get as much help as possible finding bugs in our code from clang or gcc?

   Compile with -Wall, which requests all possible warning messages.

7. What other tools can we use to check for potential problems in our code?

   Tools such as cpplint and splint (generally any tool with "lint" in the name) while check for style issues, security holes, etc.

8. Following the sine example in the lab exercises, write a C program that asks the user for a number and prints the square root of the number. The program should print the best possible input prompt. Use the `scanf()` function for input and the standard library `sqrt()` function to compute the square root. Check the man page for `sqrt()` to see what header files and compiler flags it requires.

   Make sure the program compiles without warnings when using `-Wall`. Also run **cpplint** and **splint** on the code to check for style issues.

   What is the best name for the source file?

```
// Source file name square-root.c
#include <stdio.h>
#include <sysexits.h>
#include <math.h>

int     main()

{
    double  number;

    fputs("Enter a non-negative number: ", stdout);
    if ( scanf("%lf", &number) == 1 )
    {
        printf("The square root of %f is %f\n", number, sqrt(number));
        return EX_OK;
    }
    else
    {
        fputs("Input was not a number.  Please try again.\n", stderr);
        return EX_DATAERR;
    }
}
```

Chapter 6

Data Types

6.1 Introduction

6.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why is it important to choose optimal data types?

   The wrong data types can lead to incorrect output or significantly reduced performance.

2. What happens if we discover a need to change a data type after a program is "finished" (knowing that no program is ever really finished)?

   We will have to spend time thoroughly retesting the program to ensure that the change did not cause any regressions (new bugs).

6.2 Variables

6.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How does machine language access data in memory?

   Using a binary memory address.

2. What is a variable in a high-level language?

   An identifier that refers to a memory address and has an associated data type.

3. What happens when a user types a real number as input to the `scanf()` function?

   The function converts the sequence of characters typed into binary floating point format and stores the result in the variable provided.

4. Should variable names of the same type be placed on the same line or on separate lines?

   This is largely a matter of personal taste. Just keep the code well-organized, with consistent indentation and spacing.

5. What is the advantage of using complete words as variable names instead of abbreviations?

   Unambiguous identifiers make the code self-documenting, which makes it easier to read and reduces the need for comments. It is a form of enlightened laziness, i.e. minimizing effort in an intelligent way.

6. How much training should a user need before using a program?

   Ideally, none. Some programs may require knowledge of a domain such as business, biology, or physics, but people with that knowledge should be led by the program to enter correct input.

6.3 C's Built-in Data Types

6.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a scalar variable?

   A dimensionless variable, i.e. one that holds a single value.

2. What is an aggregate variable?

   One that holds multiple values, such as an array, vector, structure, or class,

3. What is the size of an `int` in C? A long? An unsigned int?

   int and unsigned int: Generally either 16 bits or 32 bits, depending on the CPU on which the program is compiled. long: 32 or 64 bits, depending on the CPU.

4. What is the range of an `int` in C? A long? An unsigned long?

   int: Either -32,768 (-2^15) to +32,767 (+2^15 - 1) or -2,147,483,648 (-2^31) to +2,147,483,647 (+2^31 - 1), depending on the CPU on which the program is compiled. long: -2^31 to +2^31 - 1 (about +/- 2.1 billion). Unsigned long: 0 to 2^32 - 1 (about 4.3 billion).

5. When should we use the `int` data type?

   When the most pessimistic assumptions hold, i.e. when -32,768 to +32,767 is enough range *and* 4 bytes per value is not too much memory (e.g. for a large array).

6.4 Constants

6.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the data type and decimal value of each of the following?

   (a) 45
   int, 45

   (b) 0xAu
   unsigned int, 10

   (c) 012L
   long, 10

(d) 3.1e2
310.0, double

(e) 3.1e2f
310.0, float

(f) 'A'
int, 65

(g) '\033'
int, 27

2. Show a C constant of type `int` with an octal value of 377.

   0377

3. Show a C constant of type unsigned long with a binary value of 10010011.

   0x93ul

4. Show a C constant of type double with a value of 6.02 * 10ˆ23.

   6.02e23

5. What are the decimal ASCII/ISO values of each byte in the string constant "123\r"?

   49 50 51 13 0

6. How can we assign an unsigned integer variable its maximum value without knowing the exact value?

   unsigned int x = -1;

7. What is the problem with the following code? Fix it.

```
double  base, height, triangle area;

triangle_area = 1 / 2 * base * height;
```

   Integer division results in an area of 0 every time.

```
triangle_area = 1.0 / 2.0 * base * height;
```

8. What is the problem with the following code? Fix it.

```
float base, height, triangle area;

triangle_area = 1.0 / 2.0 * base * height;
```

   It mixes float and double data types, causing promotions that will slow down the program.

```
float base, height, triangle area;

triangle_area = 1.0f / 2.0f * base * height;

// or

double  base, height, triangle area;

triangle_area = 1.0 / 2.0 * base * height;
```

9. What is "hard coding"?

   Writing constants straight into program statements.

10. What is a better alternative to hard coding?

    Use named constants. This makes the program more self-documenting and more maintainable since there is only one change needed to update the value.

11. How can we define a constant MAX_NAME_LEN to 50 in a C program?

    #define MAX_NAME_LEN 50 const int MAX_NAME_LEN = 50;

12. How can we define a constant MAX_NAME_LEN to 50 in a C compile command?

    cc -O -Wall -DMAX_NAME_LEN=50 program.c -o program

13. How do we ensure that our floating point constants don't cause unnecessary round-off error?

    Define them to the maximum precision of the data type, e.g. 16 digits for `double`. Better yet, use the constants defined in the standard C headers rather than redefining them.

6.5 Initialization in Variable Definitions

6.5.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the pros and cons of initializers in variable definitions?

   Pro: Saves 1 line of code.

   Con: Makes the code less cohesive. Initializing variables immediately before they need their initial value makes the code easier to read by keeping related statements together.

6.6 Choosing the Right Data Type

6.6.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why is it important to choose optimal data types?

   The wrong data type can lead to incorrect output and/or poor performance.

2. When should we use floating point types?

   As a last resort. Floating point types are less precise, floating point operations take about three times as long as integer operations, and floating point comparisons are unreliable due to round-off error.

3. When should we use `int`?

   Since an `int` may be either 16 or 32 bits, depending where the code is compiled, we should use it only when the range of a 16-bit integer is sufficient *and* a 32-bit integer is not too much memory to use (e.g. for large arrays).

4. What is the advantage of using `int` over explicitly using a 16-bit integer or a 32-bit integer?

   A 16-bit integer will be promoted before math operations on a 32-bit or larger CPU, which slows down the code. A 32-bit integer will require multiple precision arithmetic on a 16-bit processor, which will greatly slow down the code. An `int` is guaranteed to be the fastest data type on all CPUs and `int` operations will complete in a single instruction on all except 8-bit CPUs.

5. When should we use `short`?

   When we need to conserve memory, such as for large arrays, and the range of a 16-bit integer is sufficient. Every math operation on a `short` will result in a promotion on 32-bit and larger CPUs, so using short will slow down the code.

6. When should we use long?

   When neither `short` nor `unsigned short` provides enough range, and hence `int` won't work on 16-bit CPUs.

7. When should we use `unsigned`?

   When there is no chance that a value can be negative, and using an unsigned type will provide the range that would require a larger signed integer size. E.g., if values have a range of o to 40,000, an `int` won't work, but an `unsigned int` will. This is preferable to using `long`, which will require multiple precision arithmetic on some computers.

8. How do we choose between `float` and `double`?

   Use float only to save memory, e.g. for large arrays. It is not generally faster than double and it severely limits precision to about 7 decimal digits.

9. What is the optimal data type for a scalar variable containing a person's age.

   Use `int` or `unsigned int`. Both provide sufficient range and guarantee the fastest possible math operations.

10. What is the optimal data type for a large array of a people's ages.

    Use `unsigned char` (range 0 to 255), probably using a `typedef` to define `tiny_uint_t` or similar to avoid confusion. This will minimize memory use, and the expense of some promotions to `int` for math operations. A `char` (range -128 to +127) will probably work, since the oldest person on record lived to 122. But there is no reason not to use `unsigned` here.

11. What is the optimal data type for a large grid of ocean depths measured in feet.

    Use `unsigned short`. The maximum depth of the Mariana trench is about 35,000 feet, beyond the range of a 16-bit signed integer, but well within the range of a 16-bit unsigned integer. Since the grid will contain a large number of values, we want to minimize memory use.

12. What is the optimal data type for a scalar variable containing molar concentration of a salt solution, ranging from roughly 10^-10 to 10^20.

    Use `double`. The complexity of avoiding floating point here outweighs the benefit. Even if we choose very small units, the range would be up to 10^30, which would require multiple precision integer arithmetic, eliminating most of the benefit of using integers. Using `float` for one value would only save 4 bytes of memory and would not improve performance.

13. What is the optimal data type for a large array containing molar concentrations of a salt solution, ranging from roughly 10^-10 to 10^20, and measured to an accuracy of 5 significant figures.

    Use `float` to save memory. It has sufficient range, and the 6-7 digits of precision are enough to preserve the 5 accurate significant figures, as long as the code is carefully written to control round-off error.

14. What is the optimal data type for a scalar variable containing net income of a company with a maximum of $2,000,000 per year.

    Use `long`, storing the value in pennies to avoid floating point. 200,000,000 cents is well within the range of a 32-bit signed integer (a long may be either 32 bits or 64 bits) and income can be negative. An `int` will not be big enough on 16-bit CPUs. It may seem unlikely that we would use this code on a small embedded device, but we should never make such assumptions.

15. What is the optimal data type for a large array of incomes similar to the previous question.

    Use `int32_t`, since `long` is 64 bits on some systems and we only need 32 bits to provide sufficient range. This will cut memory use in half on 64-bit systems.

6.7 Creating New Type Names: Typedef

6.7.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Show a type definition equating `age_t` to `unsigned char`.

```
typedef unsigned char  age_t;
```

6.8 Addendum: Enumerated Types

Chapter 7

Simple Input and Output

7.1 The Standard I/O Streams

7.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How does all Unix I/O ultimately occur?

   Using low-level read() and write() functions.

2. What is a stream?

   A buffering mechanism that allows programs to read or write one character at a time using a memory buffer. A block read or write occurs when the write buffer is full or when the last character is read from the read buffer.

3. What standard streams are available in all Unix processes and to what devices are they connected?

   The stdin, stdout, and stderr streams. They are connected to the terminal by default, but can be redirected to any file or I/O device.

4. How do we find out what header files are required for a C library function?

   man function-name

7.2 Single Character I/O

7.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How does putchar() work with the stdout stream buffer?

   It writes a single character to the stdout buffer. When the buffer is full, it is "flushed" to the file or device and the buffer is marked empty.

2. What data type does getchar() return and why?

   It returns `int`, because `char` and `short` values are promoted to `int`. Using `char` would therefore entail overhead costs.

7.3 String I/O

7.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What problems might occur when using `gets()` and why?

   The gets() function does not know how big the character array is, to if the user enters a string larger than the array, it will overflow and corrupt other variables.

2. Write a C program that asks the user for their name, and displays a simple greeting. Be sure to test this and all programs before submitting.

```
What is your name? Joe Piscopo
Hey, Joe Piscopo
```

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

#define MAX_NAME_LEN    100

int     main()

{
    char    name[MAX_NAME_LEN + 1];

    fputs("What is your name? ", stdout);
    fgets(name, MAX_NAME_LEN + 1, stdin);
    fputs("Hey, ", stdout);
    fputs(name, stdout);

    return EX_OK;
}
```

7.4 Numeric I/O

7.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why do `char`, `short`, and `int` all use the same placeholders in `printf()`?

   Because `char` and `short` are promoted to `int` when passed to functions.

2. Do "%d" and "%ld" mean the same thing?

   They do on some systems, which can lead to writing non-portable code if we're not careful.

3. Why does `scanf()` not use the same format specifiers for `char`, `short`, and `int`, like `printf()`?

   Because `scanf()` receives addresses as arguments, not various types of integer values, so no promotions occur.

4. Where do we find a complete list of format specifiers for `printf()` and `scanf()`?

   man printf, **man scanf**

5. Write a C program that asks the user for the length and width of a rectangle and prints the area and perimeter.

```
Please enter the length and width of the rectangle: 2 3
The area is 6.000000 and the perimeter is 10.000000.
```

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

int     main()

{
    double  length, width, area, perimeter;

    fputs("Please enter the length and width of the rectangle: ", stdout);
    if ( scanf("%lf %lf", &length, &width) == 2 )
    {
        area = length * width;
        perimeter = 2.0 * length + 2.0 * width;
        printf("The area is %f and the perimeter is %f.\n", area, perimeter);
    }
    else
        fputs("Error reading length and width.  Please try again.\n", stderr);

    return EX_OK;
}
```

7.5 Using fprintf() for Debugging

7.5.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is caveman debugging? Is it obsolete?

   Caveman debugging is the simplest form of tracing a program, by inserting print statements so we can see what is happening while the program runs. It will never be obsolete. It's easy, portable, and it works.

2. To where should we print debug output?

   To `stderr`, to keep it separate from normal output and because `stderr` is normally unbuffered, so we see all debug output before a program crashes.

7.6 Addendum: Robust I/O

7.6.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why should we always test for the success of input and output functions?

   Obviously, we don't want a program to ignore errors and continue as if nothing is wrong.

2. Are there any exceptions to the "always check" rule?

   Yes, it is normally OK not to verify the success of output to the terminal, since the problem will likely be obvious to the user anyway.

Chapter 8

C Statements and Expressions

8.1 Simple Expressions

8.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What abstract data types does C provide?
   None. Everything is a number or just bits. All abstract types are defined by the programmer.
2. What is a statement in C?
   Any expression followed by a semicolon.

8.2 C Operators

8.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is 7 / 2?
   3
2. What is 7 % 2?
   1
3. What is 7.0 / 2.0?
   3.5
4. What are the values of a, b, c and d after the following code segment?

```
    int     a, b, c, d;

    a = 5;
    b = 7;
    c = a++;
    d = ++b;
```

   a = 6, b = 8, c = 5, d = 8;

8.3 Mixed Expressions

8.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a promotion and when does it occur?

   Conversion of a smaller data type to a larger one. It occurs when two different data types are operands to the same operator.

2. What is a demotion and when does it occur?

   Conversion of a larger data type to a smaller one. It occurs only when assigning an expression to a variable of a smaller type.

3. When are `char` and `short` values promoted?

   Every time they are used with a math operator, whether or not they are mixed with a higher type.

4. List all the promotions, demotions, and mathematical operations that occur when evaluating the following expression. Indicate the data type of each intermediate result using standard C constants.

```
    char        a = 2, b = 6;
    long        c = 4, d = 10;
    double      x = 9.0, y = 3.0;
    float       z;

    z = b / a * c / d + x * y;
```

   (a) Promote char 2 to int
   (b) Promote char 6 to int
   (c) 6 / 2 = 3
   (d) Promote 3 to long 3L
   (e) 3L * 4L = 12L
   (f) 12L / 10L = 1L
   (g) 9.0 * 3.0 = 27.0
   (h) Promote 1L to double 1.0
   (i) 1.0 + 27.0 = 28.0
   (j) Demote 28.0 to float 28.0F
   (k) z = 28.0F

5. Alter the expression above so that no integer divisions occur.

```
    // Cast either a or b to double
    z = (double)b / a * c / d + x * y;
```

8.4 Bitwise Operators

8.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the hexadecimal value of c after the following? Try to do it in your head or on paper first, then check by writing a 2-line program that prints c.

```
    unsigned char   c = 0xf0 >> 3;
```

   0x1e

2. What is the hexadecimal value of c after the following?

```
    char   c = 0xf0 >> 3;
```

   0xfe

3. What is the hexadecimal value of c after the following?

```
    unsigned char   c = 0x1c & 0xf2;
```

   0x10

4. What is the hexadecimal value of c after the following?

```
    unsigned char   c = 0xf0 ^ -1;
```

   0x0f

5. What is the hexadecimal value of c after the following?

```
    unsigned char   c = 0xfc | 7;
```

   0xff

6. What operator and mask would you use to clear bits 2 and 3 of a char?

   & 0xf3

7. What operator and mask would you use to set bits 0 and 7 of a char?

   | 0x81

8. What will happen if you accidentally use a logical operator such as || in place of a bitwise operator such as | ?

   The compiler will happily generate instructions for the logical operator and the program will produce incorrect output.

9. Write a C program that inputs an integer and prints the value times 2, the value times 4, and the value times 8, using the most efficient method possible.

```
Please enter an integer: -2
-2 * 2 = -4
-2 * 4 = -8
-2 * 8 = -16
```

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

int     main()

{
    int     c;

    fputs("Please enter an integer: ", stdout);
    scanf("%d", &c);
    printf("%d * 2 = %d\n", c, c << 1);
    printf("%d * 4 = %d\n", c, c << 2);
    printf("%d * 8 = %d\n", c, c << 3);

    return EX_OK;
}
```

8.5 Addendum: More Performance Tricks

8.5.1 Polynomial Factoring

8.5.2 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Rewrite the following expression to make it compute faster:

```
    double  x, y;
    int     a;

    y = 4.5 * x * x * x - 5.3 * x * x + a * 32;
```

```
    y = x * x * (4.5 * x - 5.3) + a << 5;
```

Chapter 9

Decisions with If and Switch

9.1 Program Flow

9.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the two kinds of statements in a C program and what do they do?
   - Expressions followed by semicolons, which perform computations.
   - Flow control statements, which alter the normal top-to-bottom sequence of execution.

9.2 Boolean Expressions

9.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a Boolean expression?

   An algebraic expression with a value of true or false.

2. What are the primary Boolean operators?

   AND (A AND B is true only if A and B are both true), OR (A OR B is true if either A or B is true), and NOT (NOT A is true only if A is false).

3. How are Boolean values treated in C?

   They are integers, with 0 meaning false and any non-zero value meaning true.

4. How are most Boolean expressions formed?

   Using relations between values of other types, such as `x == y`.

9.3 The if-else Statement

9.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What do we need to watch out for when using floating point in relational expressions?

   Round-off (or truncation) error may alter the value of the Boolean expression from what it should be. We should never use == or != with floating point, and other relational operators must be used with great caution.

2. Should we use == or = in an `if` expression?

   We can use either or both, as long as we understand what they are doing. == performs a comparison, and = performs an assignment.

3. What is a compound statement and where can we use one?

   A compound statement is any sequence of statements enclosed in curly braces. We can use one pretty much anywhere that a single statement would go.

4. How should we format `if-else` blocks?

   Consistently indent the code under the `if` and under the `else`.

5. Write a C code segment that checks to see of the value of variable `count` is between 0 and `MAX_COUNT` (inclusive), and if not, prints an error message stating that it's out of range.

```
    if ( (count < 0) || (count > MAX_COUNT) )
        fprintf(stderr, "count must be between 0 and %d.\n", MAX_COUNT);
```

6. Write a C code segment that checks to see if the variable `temperature` is above `MAX_TEMPERATURE`. It should then check another variable `hot_time`, print a warning if it is greater than `WARNING_TIME` minutes, and call the function `shutdown()` if it is greater than `CRITICAL_TIME`.

```
    if ( temperature > MAX_TEMPERATURE )
    {
        if ( hot_time > CRITICAL_TIME )
            shutdown();
        else if ( hot_time > WARNING_TIME )
            fprintf(stderr, "Warning: System overheated for %d minutes.\n",
                    hot_time);
    }
```

7. Simplify the following `if` statement using De Morgan's rule:

```
    if ( ! ((a < 10) || (a > 20)) )
        statement;
```

```
    if ( (a >= 10) && (a <= 20) )
        statement;
```

9.4 Switch

9.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Write a C code segment that compares an integer variable `color` to the constants RED, GREEN, and BLUE, and prints a string matching the color in each case.

```
// A much more concise implementation is possible using an array of
// strings.  This will be covered in a later chapter.
switch(color)
{
    case    RED:
        puts("Red");
        break;
    case    GREEN:
        puts("Green");
        break;
    case    BLUE:
        puts("Blue");
        break;
    default:
        puts("Invalid color constant.");
}
```

9.5 The Conditional Operator

9.5.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Write a C statement that assigns `z` the minimum of `x` and `y`, using only a conditional operator.

```
z = x < y ? x : y;
```

9.6 Performance

9.6.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Write an optimal `if-else` statement that prints "Just another day in the cheese state." if the temperature is below 90 F (32 C) in Milwaukee, or "It's a hot one in Wisconsin." if it isn't. Explain your code in a comment.

```
// Check temperature >= 90 rather than temperature < 90, because
// we want the expression to be false most of the time to improve
// efficiency.  If statements are compiled so that the else clause
// is the faster one.
if ( temperature >= 90 )
    puts("It's a hot one in Wisconsin.");
else
    puts("Just another day.");
```

2. Write an optimal `if-else` statement that checks the temperature in Milwaukee in January, and prints "Better bring the beer in." if the temperature is below 30 F (-1 C) or above 50 F (10 C), and "It's OK to leave the beer outside" otherwise. Explain your code in a comment.

```
    // Check for temp < 30 first, because this is more likely to be
    // true than temp > 50.  If the first relation is true, the second
    // need not be evaluated.
    if ( (temperature < 30) || (temperature > 50) )
        puts("Better bring the beer in.");
    else
        puts("It’s OK to leave the beer outside.");
```

Chapter 10

Repetition: Loops

10.1 Loops and Performance

10.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the execution path?

   The sequence of instructions as they are executed by a process.

2. What are the two general ways to speed up a program? (5 to 10 words each)

   1. Shorten the execution path. 2. Replace instructions in the execution path with faster ones.

3. How can we measure the total run time of a Unix process?

   Run it under the time command.

4. How can we monitor resource use while a process is running?

   Use the **top** command.

10.2 The Universal Loop: while

10.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What often causes an infinite loop?

   Forgetting the housekeeping, such as incrementing the loop variable. (Other valid answers are acceptable).

2. What is loop overhead? How can we eliminate it, and at what cost.

   Loop overhead is anything that does contribute to producing results, such as housekeeping and loop condition checks. We can eliminate it by asking the compiler to unroll loops. This may significantly increase the size of the program by replicating the loop body many times.

10.3 The do-while Loop

10.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How does the behavior of a `do-while` loop differ from that of a `while`?

   A `do-while` always executes at least once.

2. Do `while` and `do-while` loops exhibit the same performance? Why or why not?

   No. A `do-while` is slightly faster because checking the condition at the end of the loop eliminates the need for an unconditional jump, which is extra overhead.

10.4 The for Loop

10.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How does a `for` loop differ from a `while` loop?

   A `for` loop essentially *is* a `while` with all the loop elements collected into one place.

2. How do direct floating point comparisons affect loops? How do we solve this problem?

   Round-off error may cause an infinite loop, or the wrong number of iterations. To solve this, we should avoid using floating point, or use a tolerance if we must perform floating point comparisons.

3. Write a C program that prints the square root of every integer value from 1 to 100.

```c
/***************************************************************************
 *  Description:
 *      Print square root of every integer from 1 to 100
 *
 *  History:
 *  Date        Name        Modification
 *  2023-03-03  Jason Bacon Begin
 ***************************************************************************/

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

int     main(int argc,char *argv[])

{
    int     c;

    for (c = 1; c <= 100; ++c)
        printf("sqrt(%d) = %f\n", c, sqrt((double)c));

    return EX_OK;
}
```

10.5 Nested Loops

10.5.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. When optimizing a program with nested loops, where should we focus our efforts?
   In the body of the innermost loops. These statements are executed more than any others in the program.

Chapter 11

Functions

11.1 Subprograms for Modularity

11.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Write a top-down design for identifying lines in a file that contain a string provided by the user (i.e. how the **grep** command works).

   - Open the file.
   - Check each line for the string.
     - Read a line.
     - Compare the string to each segment of the line beginning at the first character and ending at the last character minus the string length.
     - Repeat until all lines have been read and checked.
   - Close the file.

11.2 Reusability and Encapsulation

11.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How can we avoid duplicating the effort of writing subprograms that are useful to more than one application?

   Put them in a library rather than make them part of the program.

2. What is encapsulation?

   Bundling functions or other types of programs with the data types on which they operate, to form a class.

11.3 Writing Functions

11.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why do we need prototypes in C?

   The compiler needs to know the interface of each function before it encounters a call to that function. The function definition may not be seen before the call, since it may be part of a library, another source file, or may appear later in the same source file.

2. Where are most prototypes found? Why?

   In header files. This makes them easily accessible to any source file.

3. Write a C program that prints a 10 x 10 multiplication table. Use a function called `prod()` that returns the product of the two arguments. Place the function definition after `main()` and a prototype before.

```
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100
```

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

int     prod(int a, int b);

int     main(int argc,char *argv[])

{
    int     a, b;

    for (a = 1; a <= 10; ++a)
    {
        for (b = 1; b <= 10; ++b)
            printf("%4d", prod(a, b));
        putchar('\n');
    }
    return EX_OK;
}


int     prod(int a, int b)

{
    return a * b;
}
```

11.4 Local Variables

11.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What does *scope* mean and of what is it a property?

   It is a property of every variable. It refers to the portion of a program in which the variable is visible.

2. Can a C program have multiple variables with the same name? Why or why not?

   Yes, if they do not have the same scope.

11.5 Arguments

11.5.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How do formal argument variables differ from local variables in a function?

   The *only* difference is that argument variables are initialized to a value sent by the calling function. Otherwise, they are exactly like any other local variable with function scope.

2. How are C arguments passed?

   By value. A copy of the argument's value in the caller is copied to the formal argument variable, which is a local variable in the function.

3. Draw a possible memory map showing the memory addresses, variable names, and contents of the arguments in `main()` and all variables in `pow()` below after `pow()` is called, but before the body begins executing. Be sure to account for the size of each variable and the fact that memory is byte addressable. Assume a starting address of 1000.

```
double  pow(double base, unsigned exponent)

{
    double  p;
    ...

    return p;
}


int     main()

{
    double      b = 2.0;
    unsigned    e = 10;

    printf("%f\n", pow(b, e));

    return EX_OK;
}
```

```
Address Variable    Contents
1000    b           2.0         8 bytes
1008    e           10          4 bytes
1012    base        2.0         8 bytes, copied from b
1020    exponent    10          4 bytes, copied from e
1024    p           ?           8 bytes, uninitialized
1032
```

4. When are arguments promoted?

   When the argument type in the caller is `char`, `short`, or `float`, and the formal argument type is not provided by a prototype or known function definition. This happens mainly with functions like `printf()`, which take a variable number of arguments.

11.6 Library Functions

11.6.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Where does most functionality come from an C programs?

   From functions found in the standard libraries, and other functions that we write ourselves.

2. How can we get information about the `chmod()` library function, given that there is a Unix command with the same name?

   man -a chmod

11.7 Documenting Functions

11.7.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How can we tell if our function is not cohesive?

   We have difficulty giving it a simple name that fully indicates what it does.

11.8 Top-down Programming and Stubs

11.8.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Write a stub for a `pow()` function that will eventually return a real base raised to a non-negative integer exponent.

```
double  pow(double base, unsigned exponent)

{
    // Return a simple value indicating the base and exponent were
    // received correctly

    return base + exponent;
}
```

11.9 Advanced: Recursion

11.9.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Write a recursive function to compute a power of a real base raised to a non-negative integer exponent.

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

double  power_slow(double base, unsigned exponent);
double  power_fast(double base, unsigned exponent);

int     main(int argc,char *argv[])

{
    unsigned    exponent;

    for (exponent = 0; exponent <= 10; ++exponent)
        printf("%f %f\n", power_slow(2.0, exponent), power_fast(2.0, exponent));

    return EX_OK;
}


/***************************************************************************
 *  Description:
 *      Slow recursive power function.  This function is crap, but
 *      sufficient to demonstrate understanding of recursion.
 *      It uses recursion to do O(N) iteration where N is the exponent,
 *      adding function call overhead to each multiplication.
 *
 *  History:
 *  Date        Name        Modification
 *  2023-03-13  Jason Bacon Begin
 ***************************************************************************/

double  power_slow(double base, unsigned exponent)

{
    if ( exponent == 0 )
        return 1.0;
    else
        return base * power_slow(base, exponent - 1);
}
```

```
/***************************************************************************
 *  Description:
 *      log(N) recursive power function.  Takes advantage of redundancy
 *      in integer powers, i.e. base ** 10 = (base ** 5) ** 2.
 *      The latter takes 5 multiplications instead of 9.  Applying
 *      recursively brings us down to about log(N) where N is the exponent.
 *
 *  History:
 *  Date        Name        Modification
 *  2023-03-13  Jason Bacon Begin
 ***************************************************************************/

double  power_fast(double base, unsigned exponent)

{
    if ( exponent == 0 )
        return 1.0;
    else
    {
        double  half = power_fast(base, exponent / 2);

        if ( (exponent & 0x1) == 0 ) // 1 clock cycle check for odd integer
            return half * half;
        else
            return half * half * base;
    }
}
```

11.10 Advanced: Scope and Storage Class

11.10.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the 4 segments of an executable and what do they contain?

```
text    machine instructions
data    static data (variables defined as static, global variables, constants)
stack   local auto variables, function call data
heap    memory allocated via malloc() or new
```

2. When are static variables initialized? Auto variables?

   Static variables are initialized at compile time, so they contain their initial value when the program starts. Auto variables are initialized when they are instantiated on the stack, when the block in which they are defined begins execution.

3. When should we use the `register` storage class?

   In the 1970s. Modern compilers use registers much more effectively at the machine code level than we can at the source code level.

11.11 Advanced: The inline Request

11.11.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What effect does inlining a function have?

   The function body is copied to the location of each function call instead of jumping to and back from the function at that point. The effect is similar to using a macro instead of a function.

Chapter 12

Programming with make

12.1 Overview

12.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What does **make** do?

   It rebuilds one or more target files when the source files from which they are built have changed.

2. How is each rule in a makefile interpreted?

   If any source is newer than the target, then run the associated command(s) below.

3. How does **make** know what is a target file and what is a command?

   Target files begin in column 1 and commands are indented with a TAB character.

4. What is **make** most commonly used for?

   Building an executable file from one or more source files written in a compiled language.

12.2 Building a Program

12.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Show the compiler commands needed to build an executable called `calc` from source files `calc.c` and `functions.c`.

```
cc -c -O calc.c
cc -c -O functions.c
cc calc.o functions.o -o calc
```

2. What C compiler command should usually be used by default in a makefile? Why?

   We should use **cc** by default. There is generally no reason to explicitly use **clang** or **gcc**, since they are linked to **cc** if they are the default compiler for a given Unix system.

3. How does **make** know when a source file needs to be recompiled?

   Unix records the last modification time on every file in the file system. If any source file in a make rule is newer than the target, then the commands are executed.

12.3 Make Variables

12.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the purpose of **make** variables?

   Same as variables in any language. To make the makefile more readable and more flexible.

2. How do we set a variable in such a way that it can be overridden by **make** command-line arguments or environment variables?

   Assign it a default value using `?=`. The assignment will only occur if the variable is not already assigned.

3. What variables should be used to specify the C compiler? C compiler flags? The linker? Link flags?

   CC for C compiler, CFLAGS for compile flags, LD for linker, LDFLAGS for link flags.

4. Write a makefile, using standard **make** variables, that builds the executable "calc" from files "calc.c" and "functions.c".

```
# Values that the user cannot override

BIN      = calc
OBJS     = calc.o functions.o

# Set only if the user (or package manager) has not provided a value
# -Wall:    Issue all possible compiler warnings
# -g:       Compile with debug info to help locate crashes, etc.

CC       ?= cc
CFLAGS   ?= -Wall -O -g
LD       ?= ${CC}

${BIN}: ${OBJS}
        ${LD} -o ${BIN} ${OBJS} -lm

calc.o: calc.c Makefile
        ${CC} -c calc.c

functions.o: functions.c
        ${CC} -c functions.c
```

12.4 Phony Targets

12.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Which target in a makefile is checked first, if no target is specified in the **make** command?

   The first one encountered, reading from the top down.

2. How do we ensure that the install target runs when specified, even if there is a file called "install" in the directory?

   .PHONY: install

12.5 Using Header Files

12.5.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What kind of code belongs in a source file (.c)?

   Only function definitions and #includes.

2. What kind of code belongs in header files?

   Everything except function definitions: Macros, prototypes, typedefs.

12.6 Makefile Generators

12.6.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the goal of makefile generators?

   To automate a software build on computers running different operating systems and/or with different configurations.

2. What is the main problem with makefile generators?

   They cannot foresee all the possible variables they will encounter and therefore do not work for everyone. When they fail, it presents a very difficult problem.

3. What is an alternative to makefile generators that leads to less problematic software deployment?

   Providing a simple build system that leaves dependencies to a package manager, which is more sophisticated and reliable than custom build scripts.

Chapter 13

The C Preprocessor

13.1 Macros and Constants: #define

13.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Show a command to view the preprocessor output for the file prog1.c.

   ```
   cc -E prog1.c
   ```

2. Show how to define the constant DEBUG with a value of 1, both within a program and from the command line.

   ```
   #define DEBUG   1
   ```

   ```
   cc -DDEBUG=1 prog.c
   ```

3. Show how to define a macro called INVERSE that produces the mathematical inverse of a number.

   ```
   #define INVERSE(x)  (1 / (x))
   ```

4. What operators should we avoid using in macros? Why?

   ++ and --, since the the argument may be evaluated more than once, causing a variable to be incremented or decremented twice or more.

5. Why should we use parentheses extensively in macro definitions?

   To ensure that precedence and associativity don’t alter the behavior depending on the expression given as an argument.

13.2 Functions vs. Macros

13.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why are macros typically faster than functions? When are they not?

   Macros have no function call overhead, since they are expanded in-place rather than jumped to and back from. An inlined function behaves more or less like a macro.

2. How can we help programmers using our macros avoid side-effects like those caused by ++ and --?

   Use all upper-case names, so they know it's a macro.

13.3 Header Files: #include

13.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the difference between angle brackets and double quotes in `#include`?

   Filenames enclose in angle brackets are system headers, which **cpp** looks for in `/usr/include` and other system directories. Filenames enclosed in quotes are project headers, which are generally in the CWD.

2. What should we put in ".c" source files and what should we put in headers?

   Source files: Function definitions. Header files: Everything else. Macros, prototypes, and type definitions are typically useful to multiple source files, so putting them in a header makes them easily accessible.

13.4 Advanced: Conditional Compilation

13.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Show how to print "list_size = " followed by the value of `list_size`, only if the macro DEBUG is defined.

```
// Any variable called list_size should be type size_t
#ifdef DEBUG
    fprintf(stderr, "list_size = %zu\n", list_size);
#endif
```

2. When should we use platform-detection predefined macros such as `__FreeBSD__` and `__linux__`?

   As seldom as possible. We should always try to find a way to write portable code rather than maintain different variants for different operating systems.

13.5 Advanced: Other Directives

13.6 Advanced: The Paste Operator:

13.7 Advanced: Predefined Macros

13.8 Addendum: Advanced: Variadic Macros (C99)

Chapter 14

Pointers

14.1 Pointers: This Stuff is BIG!

14.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are two benefits of learning about C pointers?

   1. We can make programs more efficient by avoiding wasteful movement of large amounts of data. 2. We gain a better understanding of how things like references work at the hardware level.

14.2 Pointer Basics

14.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the difference between a C pointer variable and other C variables?

   A pointer contains the address of an object, while non-pointer variables contain the object itself.

2. What is an address?

   An unsigned integer that acts like a subscript to memory, which is essentially an array of bytes.

3. How do the ++ and −− operators affect pointers and non-pointers?

   They add or subtract 1 to/from a non-pointer, while they add or subtract the size of the object pointed to to/from a pointer.

4. What is the difference between a pointer and an integer?

   A pointer may not be the same size than an integer, and it is incremented differently from integer values to account for the size of the object to which it points.

14.3 Defining Pointer Variables

14.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Show a single variable definition that defines two `double` variables called `a` and `b`, and two pointers to `doubles` called `ptr1` and `ptr2`.

```
    double  a, b, *ptr1, *ptr2;
```

14.4 Using Pointers: Indirection

14.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Show a possible memory map for the following variables, assuming the address of `a` is 2000, an `int` is 32 bits, and an address is 64 bits.

```
    int     a = 5, *ptr = &a, b = 10;
```

```
Address Variable    Content
2000    a           5
2004    ptr         2000
2012    b           10
```

2. Show how to print the value of **b** in the previous question using the pointer `ptr`.

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

3. Is there any cost to using pointers to access an object instead of accessing it directly from a non-pointer variable?

   Yes, there is an extra step in fetching the address of the data from the pointer variable before the object can be accessed.

4. Will the following code work? What does it mean? Is there a better approach? Why?

```
    double  *ptr = NULL;

    if ( ptr )
    {
        printf("%f\n", *ptr);
    }
```

It will work if we are trying to verify that `ptr` does not contain `NULL`. A more readable approach is to show the comparison explicitly:

```
    if ( ptr != NULL )
```

The first example is cryptic, since it equates the memory address `(void *)0` to `false` and a non-zero memory address to `true`. This is just trying to show off our cleverness for no real benefit.

14.5 Pointers as Function Arguments

14.5.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Write a C function that prompts the user for and returns the coefficients A, B, and C of a quadratic equation $Ax^2 + Bx + C = 0$.

```
int     get_coefficients(double *a, double *b, double *c)

{
    fputs("Please enter the coefficients A, B, and C: ", stdout);

    // a, b, and c are pointers, so no &s here
    return scanf("%lf %lf %lf", a, b, c);
}
```

14.6 Typedefs and Pointers

14.7 Addendum: C99: The `restrict` Pointer Modifier

Chapter 15

Arrays and Strings

15.1 One-dimensional Arrays

15.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the difference between a scalar variable and an array?

   A scalar variable holds 1 value and is dimensionless. An array holds multiple values and has at least one dimension.

2. What are two drawbacks to fixed size arrays?

   1. Waste memory 2. Limited size

3. What is a potential problem with simple arrays defined inside functions?

   The default storage class is `auto`, which means the memory is allocated on the stack, which means the size of the array is severely limited. (8 MiB on Linux)

4. How do we get past the limits of stack size for arrays?

   Define them as `static` or dynamically allocate them with `malloc()`.

5. Draw a possible memory map of the following variables, assuming a 64-bit CPU, and a starting address of 1000.

   ```
       float   vector[4] = { 5, 2, 1, 7 };
       size_t  vector_len = 0;
       int     c = 1;
   ```

   ```
   Address Var         Value   Bits
   1000    vector[0]   5       32
   1004    vector[1]   2       32
   1008    vector[2]   1       32
   1012    vector[3]   7       32
   1016    vector_len  0       64
   1024    c           1       32
   ```

6. What happens when we write to an array using an out-of-bounds subscript?

   Most likely, adjacent variables will be corrupted.

7. Are higher level languages that support vector operations faster than C, since they don't need to use loops?

   Certainly not. They use loops at the machine code level, but the loops are often hidden at the source code level.

15.2 "Foreach" Loops

15.3 Arrays and Pointers

15.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the difference between an array name and a pointer variable in C?

   The ONLY difference is that an array name is a constant, i.e. we cannot change the address it represents. Otherwise, they are interchangeable.

2. How does using a pointer to access array elements improve performance?

   Each array access using a subscript involves an address calculation of base-address + subscript * sizeof(data-type). A pointer already contains the address of the array element, so this calculation is eliminated.

15.4 Typedefs and Arrays

15.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Show a typedef for an array of MAX_NAME_LEN + 1 characters.

```
typedef char name_t[MAX_NAME_LEN + 1];
```

15.5 Advanced: More Fun with Pointers

15.6 Arrays and Functions

15.6.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Write a C function that reads one line of text from stdin into a character array, stopping if the array is full before a newline is read. The newline should not be included in the string. The function should return the size of the string read.

```
size_t  read_line(char string[], size_t array_size)

{
    size_t  length;

    // Leave room for the null terminator
    while ( ((string[length] = getchar()) != '\n') &&
            (length < array_size - 1) )
        ++length;
    string[length] = '\0';

    return length;
}
```

15.7 Lookup Tables

15.7.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why do lookup tables need to be defined as `static`?

   Variables of type `auto` are initialized at run time, which is very time consuming and hence defeats the purpose of the lookup table.

2. Write a C function that uses a lookup table to return any integer power of 10 from exponents of 0 through 9. Justify your choice of data type.

```
// Data type for exponent doesn't matter much, since it has a range of 0 to 9
uint32_t    power_of_10(unsigned exponent)

{
    // 32-bit unsigned integer has a range of up to 4.29 x 10 ^ 9
    // unsigned int may be 16 bits
    // unsigned long may require multiple precision arithmetic
    static uint32_t    powers[] = {
        1, 10, 100, 1000, 1000, 10000, 100000, 1000000, 10000000,
        100000000, 100000000 };

    return powers[exponent];
}
```

15.8 Pointer Arguments and const

15.8.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Show a pointer variable definition equivalent to the following:

```
    double  vector[] = { 1.0, 2.0, 3.0 };
```

```
    double * const vector = { 1.0, 2.0, 3.0 };
```

2. Write a C function that returns the average of the values in a vector. The array may or may not be full. Make sure the function is incapable of causing side effects.

```
// Any of the following protect the array from modification:
// double  mean(double const vector[], size_t elements)
// double  mean(const double vector[], size_t elements)
// double  mean(const double *vector, size_t elements)
double  mean(double const *vector_ptr, const size_t elements)

{
    double  sum;
    size_t  c;      // Preserve elements for computing mean

    // No need to preserve the base address of the array, so we
    // can increment vector_ptr
    for (c = 0, sum = 0.0; c < elements; ++c)
        sum += *vector_ptr++;

    return sum / elements;
}
```

15.9 Multi-dimensional Arrays

15.9.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why don't we need the first dimension of a 2-dimensional array in a formal argument?

   The compiler only needs to know the number of columns in each row in order to calculate the starting address of each row.

15.10 Addendum: Performance and Portability

15.10.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How does the use of large arrays hurt program performance?

   It forces the computer to use larger, slower memory levels, thus increasing average memory access time.

2. How does the unnecessary use of arrays limit the utility of programs?

   It limits the amount of data a program can process to the size of memory where the program is being run. If an array is not used, they the program has no limits.

Chapter 16

Dynamic Memory Allocation

16.1 Dynamic Memory Allocation: `malloc`

16.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the advantage of dynamic memory allocation over fixed size arrays?

   Fixed size arrays have to be big enough for the biggest case, so they usually waste a lot of memory.

2. Is address space free?

   No, not if we want our code to be portable and reliable on systems with limitations.

16.2 Basic Usage

16.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. When should we add a `free()` call to a C program?

   At the same time we add the corresponding `malloc()`. Otherwise, we might forget and have a memory leak.

2. Describe one pro and one con of garbage collectors?

   Pro: The programmer need not worry about memory leaks. Con: Higher general memory use, random delays at run time when the garbage collector kicks in.

3. How do we ensure that the size portion of a `malloc()` argument is always correct?

   Either use a typedef instead of a hard-coded basic type, or give an object to `sizeof()` instead of a type.

4. How do we ensure that we don't forget the size portion of a `malloc()` argument altogether?

   Use a wrapper that requires two arguments instead of calling `malloc()` directly.

16.3 How `malloc()` Keeps Track: Heap Tables

16.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How do we avoid a fragmented heap table and the associated performance hit?

   Allocate memory in a few large blocks rather than many small ones.

2. How to we minimize the expense of `realloc()` calls?

   Minimize the number of calls, by making a few large changes to an array size rather than many small ones.

16.4 Pointer Arrays

16.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are two advantages of pointer arrays over fixed size 2 dimensional arrays?

   1. They minimize or eliminate wasted space by allocating the correct number of rows and the correct size for each row. 2. Swapping rows in a pointer array is a scalar operation that does not require moving any data.

16.5 Command-line Arguments: argc and argv

16.5.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is `argc`?

   The number of arguments received by `main()`, including the program name itself.

2. What is `argv[0]` and how is it used?

   The name of the program as it was invoked. Some programs have multiple filenames and behave differently, depending on the name used to invoke them.

16.6 The Environment: envp

16.6.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is an alternative to using `envp` in a C program?

   Use the `getenv()` library function to read individual environment variables.

Chapter 17

Advanced: Function Pointers

17.1 Simple Function Pointers

17.2 Function Pointer Tables

Chapter 18

Structures and Unions

18.1 Structures

18.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How are structures related to classes?

   Classes are based on structures, and add member functions and private data members to help enforce encapsulation.

2. Show a type definition for a structure containing a person's first name, last name, and age.

```
typedef struct
{
    char    *first_name;
    char    *last_name;
    int     age;
}   person_t;
```

3. Can whole structure objects be assigned in C?

   Yes, but this is very costly, so it should be avoided.

18.2 Pointers to Structures

18.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is an advantage to using pointers to structures rather than structure objects directly?

   Pointers are scalar objects, so manipulating them is much faster. Using pointers, we can avoid moving and copying aggregate objects like structures.

18.3 Structures, Functions, and OOP

18.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

---

1. What rule must we follow in order to implement a class in C?

   A given function must only directly access the members of one kind of structure. In this way, the function is a member of only one class.

2. Show a mutator function that sets the `age` field in the following structure.

```
typedef struct
{
    char    *first_name;
    char    *last_name;
    int     age;
}   person_t;
```

```
void    person_set_age(person_t *person, int age)

{
    person->age = age;
}
```

18.4 Nesting Structures

18.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

---

1. How can two different structures share common fields?

   By nesting a structure containing the common fields in both of them.

2. Show a structure definition for a mammal containing a Boolean field `furry` and a nested structure containing fields `average_weight` and `average_lifespan`, which can be shared with structures for reptiles, birds, fish, etc.

```
typedef struct
{
    unsigned    average_weight;
    unsigned    average_lifespan;
}   animal_t;

typedef struct
{
    animal_t    animal_attributes;
    bool        furry;
}   mammal_t;
```

18.5 Lists of Structures

18.5.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the problem with fixed size arrays of structures?

   Same as all fixed size arrays: wasted space when the array is not fully utilized, which is most of the time. The problem is worse with structures, since they are larger objects than scalar types.

2. What is the down side of linear linked lists?

   They can only be accessed sequentially, which defeats the purpose of storing data in random access memory.

3. How can we get the memory savings of a linear linked list without sacrificing random access?

   Dynamically allocate an array of structures or structure pointers.

4. What is a major advantage of pointer arrays over arrays of structures?

   With pointer arrays, we can reorder the list without moving structure objects, which would be expensive.

18.6 Initializing Structures

18.6.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are two advantages of using a macro to initialize a structure?

   1. We need only update the initializer in one place when the structure changes. 2. It reduces redundant clutter in the code.

2. How does a designated structure initializer avoid bugs?

   It remains valid if structure members are reorganized. Non-designated initializers match the initial value to a field only by its position in the structure definition.

18.7 Advanced: Unions

18.7.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the difference between a structure and a union?

   All members of a union represent the same memory location.

2. What is the problem with using a union to address individual bytes of an integer? What is the solution?

   The position of the bytes is endian-dependent. The first byte may be the lowest byte of the integer on some systems and the highest on others. Use bit operators and masks for endian-independent byte access.

18.8 Advanced: Structure Alignment

18.9 Advanced: Bit Fields

18.10 Addendum: Advanced: Enforcing OOP in C, Opaque Structures

18.11 Addendum: Advanced: Flexible Array Members (C99)

18.12 Addendum: Advanced: void pointers

Chapter 19

Debugging

19.1 Thinkin' it Through

19.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How do we avoid the need for a debugger?

   Test code incrementally during implementation, to minimize the number of bugs in the first place.

2. What is the first step in correcting a bug?

   Locate it. Finding a bug without worrying about what is causing it is generally easy. Determining the cause is then easier once we know where to look.

19.2 Making Programs Talk: Debug Code

19.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is caveman debugging? Is it obsolete?

   Inserting print statements into code so we can "watch" it execute. It is not obsolete. It is simple, portable, and very useful.

2. Where should debug output be sent and why?

   To stderr, since it is unbuffered by default, and this allows us to separate it from normal output using redirection.

3. How do we disable debug output when we no longer need it?

   Comment it out, or guard it with an `if` statement or an `#if` or `#ifdef` directive. Removing it would be risky, since it may be needed in the future.

19.3 Unix Debuggers: Which One?

19.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How do we prepare a program for optimal debugging?

   Compile with `-g` and without aggressive optimizations, so that the program includes an accurate address map that the debugger can use to point to specific lines in the source code.

19.4 The GNU Debugger: `gdb`

19.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Show how to force a process to terminate and generate a core file, then show a backtrace using **gdb**.

```
shell-prompt: kill -ABRT pid
shell-prompt: gdb program core-file
(gdb) bt
```

19.5 Addendum: The LLVM Debugger: `lldb`

19.5.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Show how to force a process to terminate and generate a core file, then show a backtrace using **lldb**.

```
shell-prompt: kill -ABRT pid
shell-prompt: lldb -c core-file program
(lldb) bt
```

19.6 Addendum: Valgrind

Part III

Unix Library Functions and Their Use

Chapter 20

Building Object Code Libraries

20.1 Creating Libraries

20.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Which new functions should be placed in a library rather than made part of a specific application?

   Any functions that *might* be useful to another program.

20.2 Static or Dynamic?

20.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What happens when we link to a static library?

   Machine code is extracted from the library and inserted into our executable.

2. What happens when we link to a dynamic (shared) library?

   Only a reference to the library is inserted into our executable and the machine code is loaded directly from the library at run time.

3. What is the advantage of static libraries?

   The library is no longer needed after linking. All of the required machine code is present in the executable itself. With dynamic libraries, we need to have the correct version of the library present at run time, often multiple versions for different programs.

4. What are the advantages of a dynamic library?

   1. The executables are smaller, since they don't contain redundant copies of the library code. 2. Code loaded into memory can be shared by multiple applications, so total memory use is lower.

20.3 Creating Static Libraries

20.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Show the commands needed to create a static library `libstring.a` from `strcmp.o`, `strlen.o`, `strlcat.o`, and `strchr.o`.

   ```
   ar r libstring.a strcmp.o strlen.o strlcat.o strchr.o
   ranlib libstring.a
   ```

20.4 Creating Dynamic (Shared) Libraries

20.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the major challenge in creating dynamic (shared) libraries?

   They are not well standardized across Unix platforms. For instance, macOS uses a ".dylib" extension, whereas BSD and Linux use ".so". The tools for manipulating shared libraries are also different.

Chapter 21

Files and File Streams

21.1 FILE Streams

21.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How do FILE streams improve I/O performance?

   Actually reading or writing small amounts of data from/to a disk or I/O device would be very inefficient due to seek and latency times (the time used before the first byte is read or written). FILE streams buffer data in memory so that large blocks of data can be read or written at the device level even though programs perform smaller transactions.

21.2 The `FILE` Structure

21.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What happens the first time a program calls `getc()` on a new stream?

   Since the buffer starts empty, `read()` is called to fill it with a block of data. The `getc()` macro then returns the first character in the buffer and increments the buffer pointer so it points to the next one.

2. What happens the second time a program calls `getc()` on a new stream?

   Since the buffer was filled during the first call, `getc()` simply returns the next character in the buffer without needing to access the input device or file.

3. How often does `getc()` actually access an input device?

   Very seldom. If the buffer size is 1024 bytes, it will get 1023 of them without performing an actual input operation.

21.3 Basic Stream I/O Functions

21.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. When should we check the exit status of an input or output function?

   Always, except when writing to a terminal screen.

2. When should an `fclose()` call be added to a program?

   At the same time as the corresponding `fopen()`, to ensure that the file is closed as soon as possible. Forgetting to close a file can result in lost data.

3. Show how to open a file called "records.txt" for both reading and writing, without truncating the file.

   char *filename = "records.txt"; if ((fp = fopen(filename, "r+")) == NULL) { fprintf(stderr, "Error opening %s: %s\n", filename, strerror(errno)); exit(EX_NOINPUT); }

4. How do we know the difference between a read error and an end-of-file condition?

   Some functions return different values for the two conditions. Otherwise, we need to call `feof()` or `ferror()` afterward to tell them apart.

5. What are the pros and cons of `fread()` and `fwrite()` vs `fscanf()` and `fprintf()`?

   Reading and writing binary data (fread, fwrite) may be faster, but the data may not be portable across hardware using different endianness or floating point formats.

Chapter 22

String and Character Functions

22.1 Basic String Manipulation

22.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a string in C?

   An array of characters, with a null-terminator marking the end of the content.

22.2 String Functions

22.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why should we avoid copying strings (or other arrays)?

   It wastes CPU time, since it requires a loop, and it wastes memory, since we are duplicating large amounts of data. It is also difficult to ensure that the target array is always large enough, so strings often end up truncated.

2. What is the problem with `strcpy()`?

   It does not know the size of the target array, so it could overrun the end of it and corrupt other data.

3. Can we use the assignment operator, =, to copy strings?

   No. We can use it to assign pointers to strings, but this does not copy the string. It only copies the address.

4. How can we copy a string, ensuring that the target array is the correct size? Are there any risks?

   Use `strdup()`, which uses `malloc()` to allocate the target array. Risks: The `malloc()` could fail, and forgetting to free the memory will result in a memory leak.

5. What is the down side of using `strlen()` and how can we avoid it?

   It is costly, since it loops through the string looking for the null byte. We can avoid it by recording string lengths in integer variables rather than using `strlen()` repeatedly.

6. Why doesn't C support string comparison with ==?

   Because it can easily be done with a function. C does not include any features that can be implemented reasonably well as a function.

7. Where can we find more information about string functions?

   In the SEE ALSO sections of string function man pages, **man string**, and running **more /usr/include/string.h**.

22.3 Classifying Characters: The ctype Functions

22.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What can we find in the header `ctype.h`?

   Numerous macros for classifying and manipulating characters, such as `isalpha()`, `isdigit()`, `tolower()` etc.

22.4 Pattern Matching Functions

22.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Show a regular expression that matches an octal integer constant.

   0[0-7]*

2. Show a globbing pattern that matches all of the object files in the directory Program1.

   Program1/*.o

3. What is the difference between '*' in a regular expression (RE) and a globbing pattern?

   In an RE, it represents 0 or more occurrences of the character before it. In a globbing pattern, it represents 0 or more occurrences of any character.

22.5 Bulk Memory Manipulation

22.5.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the difference between `memcpy()` and `memmove()`?

   It is safe to use `memmove()` on overlapping memory blocks.

2. Does `memcmp()` have any advantages over `strcmp()`?

   Yes, it may be faster since it does not need to compare one byte at a time while searching for the null terminator.

Chapter 23

Odds and Ends

23.1 Math Functions

23.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Where do standard math functions reside, and what link option to we need to use them in our programs?

   They reside in libm.a or libm.so, so we use -lm.

2. What is the portable way to find out what math functions are available on our system?

   more /usr/include/math.h. FreeBSD has a man page that summarizes them, so **man math** can be helpful, but this is not portable.

23.2 Data Conversion Functions

23.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What function can we use to convert a string to an integer?

   strtol()

2. What function can we use to convert an integer to a string?

   snprintf()

23.3 Random Numbers

23.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What does "random" mean?

   Difficult to predict.

2. What should we do before using the `random()` function to ensure somewhat unpredictable results?

   Seed the random number generator by calling `srandom()` with a different value each time, such as the current clock time.

23.4 Basic Process Control

23.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What does `exit()` take as an argument?

   The exit code for the process, the same as we use in a `return` statement in `main()`.

23.5 Manipulating the Environment

23.5.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How can a C program find out the value of the `TERM` environment variable and assign it to a string variable called `term_type`?

   ```
   char    *term_type = getenv("TERM");
   ```

23.6 Sorting and Searching

23.6.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How do functions like `qsort()` and `bsearch()` manage to be polymorphic?

   They take the size of each element and a pointer to a comparison function as arguments, so that they can operate on an array of any data type.

23.7 Functions with Variable Argument Lists

23.8 Addendum: Advanced: `tgmath.h` (C99)

Chapter 24

Working with the Unix Filesystem

24.1 File Information: `stat()` and `fstat()`

24.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What function can we use to find out about the permissions on a file? A file descriptor?

   Use stat() for a filename, fstat() for a file descriptor.

2. How can we find out more about the `stat()` function?

   man 2 stat or **man -a stat**.

24.2 Changing File Information

24.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Briefly describe the process of adding read permissions for "other" on a file from a C program. Or write the code if you wish.

   Use `stat()` or `fstat()` to get the current permissions, turn on the read bit using a bitwise or, and use `chmod()` to set the new mode on the file.

24.3 Accessing Directories

Chapter 25

Low-Level I/O

25.1 Why Use Low-level I/O

25.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the advantage of using low-level I/O over FILE streams?

   The buffering mechanism used by FILE streams to read and write single characters uses a significant amount of CPU time. If we don't need to examine individual characters, low-level I/O greatly reduces CPU use and may speed up I/O.

2. What is the limitation of low-level I/O compared with FILE streams?

   Low-level I/O only transfers blocks of bytes. There is no efficient single-character handling like `getc()` and `putc()` or numeric I/O like `fprintf()` and `fscanf()`.

25.2 Basic Input and Output

25.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What does `open()` return? How does it relate to what `fopen()` returns?

   The `open()` function returns an integer file descriptor, or -1 if the open failed. This file descriptor is part of the `FILE` structure returns by `fopen()` and is used to flush and refill the `FILE` stream buffers using `read()` or `write()`.

2. How does `write()` relate to `putc()` and other FILE stream functions?

   FILE stream functions call write to empty the FILE stream buffer when it is full. They then continue adding characters to the start of the newly emptied buffer.

Chapter 26

Controlling I/O Device Drivers

26.1 Termios

Chapter 27

Unix Processes

27.1 Creating Processes

27.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Describe one pro and one con of using the `system()` function?

   Pro: Convenience, since we can utilize all the features of the shell in the command. Con: High overhead, since it actually starts up a new shell process and then passes the command to it.

2. How does a program know if it is running the parent process or the child process after calling `fork()`?

   The `fork()` function returns 0 to the child, and the PID of the child process to the parent.

3. What do the `exec` family of functions return upon successful completion?

   They don't. The calling program is replaced by an entirely new program, so there is nothing to return to.

4. How does a shell program know when the foreground command we ran under it is finished?

   The shell calls `wait()` to pause itself until a child process exits.

5. What is the advantage of the POSIX spawn interface over `fork()` and `exec()`?

   Convenience. It allows us to create new processes and set numerous parameters all in a single function call.

27.2 Redirection

27.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Briefly describe the process of redirecting the standard input of a child process.

   (a) Fork.

   (b) In the child process, close descriptor 0, so that it will be the lowest descriptor available to the next `open()`.

   (c) Open the file or device we want connected to the standard input. The `open()` call will use descriptor 0 since it is the lowest available.

Chapter 28

Interprocess Communication (IPC)

28.1 The Environment

28.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the major limitation of the environment as an IPC mechanism?

   It only allows processes to send messages to their children. They cannot send messages to their parents or any other process.

28.2 Signals

28.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What Unix command do we use to send a SIGCONT signal (which tells a stopped process to resume running) to a process. What C function?

   If you were grading a program where the student was tasked with writing a function that sends any signal to any process, how many points would you award for descriptive naming to a student who gave the function this name?

   Use the **kill** command or the `kill()` function. I would give 0 points for calling such a function "kill". Though, that may have been the only purpose of the function originally. More signal types have been added since, but changing the function name would break existing programs, so we're stuck with it.

2. What is a signal handler? How do we use one?

   A signal handler is a function that a program calls when it receives a particular signal. To use one, we must register it using `signal()` or `sigaction()`.

28.3 Pipes

28.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Show how to open a FILE stream that it piped to the standard input of a new process running the **more** command.

```
FILE    *output_stream;

if ( (output_stream = popen("more", "w")) == NULL )
{
    // Error out
}
```

2. Where do we use the `pipe()` function in order to set up communication between a parent and child process?

   Before calling `fork()`. The child process inherits the file descriptors for the read and write end of the pipe, so that both parent and child can send and receive messages through it.

28.4 Sockets

28.4.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the major advantage of sockets over other IPC mechanisms?

   Sockets can be used to communicate with processes running on different computers.

2. What is the major disadvantage of sockets when compared with other IPC mechanisms?

   Socket programming is very complex and requires significant development and testing time. Sockets should only be used where simpler mechanisms are not sufficient.

28.5 Shared Memory

28.5.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How do processes communicate using shared memory?

   One process places data at a given memory location and others read it when they know it is ready.

2. Does shared memory eliminate the need for process synchronization?

   No, this need is inherent in the design of the parallel computation. It doesn't matter whether processes use pipes, sockets, shared memory, etc.

Chapter 29

Threads

29.1 Overview

29.1.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How do threads differ from heavyweight processes?

   See the list in the text.

2. What are some examples how how threads can be used? Cite examples from the text and some of your own if possible.

   Multiprocessing to speed up parts of a program, running multiple server daemons to process client requests faster.

29.2 Addendum: POSIX Threads

29.2.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why are POSIX threads important to know about?

   POSIX threads are a long-time standard and are used by many existing programs written since the 1990s.

29.3 Addendum: OpenMP

29.3.1 Practice

Note Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What criteria must be satisfied in order to use an OpenMP parallel `for` loop?

   Each iteration of the loop must be independent from others, so that the OpenMP system can run multiple iterations at the same time.

2. How do we use OpenMP in a C program?

   -
     ```
     #include <omp.h>
     ```
   - Use the OpenMP `#pragma` directives to indicate parallel sections of code.
   - Compile with -fopenmp.

3. Write a C program that prints the squares of all integers from 1 to 100, using OpenMP to utilize all available cores on the computer. Do you notice anything odd about the output?

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

   int     main(int argc,char *argv[])

   {
       int     c;

       #pragma omp parallel for
       for (c = 1; c <= 100; ++c)
           printf("%d ** 2 = %d\n", c, c * c);

       return EX_OK;
   }
   ```

   The numbers don't necessarily print in order from 1 to 100.