

# **Fundamentals of System Programming**

**Keio University's Shonan Fujisawa Campus**

**Spring 2015**



## **Fundamentals of System Programming**

original by Hiroyuki Kusumoto  
adapted by Rodney Van Meter

**October 2, 2015**



# Contents

<b>Course Introduction</b>	<b>iii</b>
0.1 To the Independent Reader . . . . .	v
0.2 Goals and Structure of the Class . . . . .	v
0.3 Learning Outcomes . . . . .	vi
0.4 To the Instructor . . . . .	viii
0.5 This Class in the Keio Shonan Fujisawa Campus Curriculum . . .	viii
0.6 Why C? . . . . .	x
0.7 Why a New Book? . . . . .	xi
0.8 How to Read this Book . . . . .	xi
0.9 Additional Recommended Books . . . . .	xii

## I Core Concepts and the C Programming Language in a UNIX Environment 1

<b>1 The Traditional “Hello, World!”</b>	<b>3</b>
1.1 Concepts . . . . .	3
1.2 Java and C . . . . .	3
1.2.1 Object-oriented languages and procedural languages . . . .	3
1.2.2 Interpreter and Compiler . . . . .	4
1.2.3 Similarities . . . . .	7
1.2.4 A Few Differences . . . . .	7
1.3 C language brief summary . . . . .	8
1.3.1 Overall Structure of a Program . . . . .	8
1.3.2 Compile and Execution . . . . .	8
1.3.3 Comments . . . . .	9
1.3.4 Variable and Type . . . . .	10
1.3.5 Function . . . . .	12
1.3.6 Function and Types . . . . .	14
1.3.7 Automatic Type Conversion . . . . .	18

1.3.8	Statement . . . . .	19
1.3.9	Expression . . . . .	22
1.3.10	Library . . . . .	23
1.3.11	Standard Input/Output Library . . . . .	24
1.4	Tools . . . . .	25
1.4.1	Gcc . . . . .	25
1.4.2	Make . . . . .	25
1.4.3	man (the Unix manual) . . . . .	26
1.4.4	Other UNIX utilities and commands . . . . .	26
1.5	Exercises . . . . .	26
<b>2</b>	<b>Fundamental Data Types</b>	<b>29</b>
2.1	Concepts . . . . .	29
2.2	Integer Data Types . . . . .	29
2.3	Floating Point Number Data Types . . . . .	33
2.4	Mixed arithmetic operation for an integer and a floating point number	34
2.5	Manipulating bitfields and bits within numbers . . . . .	34
2.6	Tools . . . . .	35
2.6.1	More on Make . . . . .	35
2.7	Exercises . . . . .	35
<b>3</b>	<b>Arrays, Structs and Pointers</b>	<b>37</b>
3.1	Concepts . . . . .	37
3.2	Arrays . . . . .	37
3.3	Pointers . . . . .	41
3.3.1	Pointer explanation and value . . . . .	42
3.3.2	Character arrays and pointers . . . . .	44
3.3.3	Pointer variables . . . . .	47
3.3.4	Manipulation of pointer variables . . . . .	50
3.4	Pointers and Functions . . . . .	53
3.4.1	Function calls and arguments . . . . .	53
3.4.2	How to pass arrays . . . . .	56
3.5	Structures . . . . .	59
3.6	Tools . . . . .	62
3.6.1	gcc: phases of compilation . . . . .	62
3.6.2	git . . . . .	62
3.7	Exercises . . . . .	62

<b>4</b>	<b>String Processing and Pointers</b>	<b>65</b>
4.1	Concepts . . . . .	65
4.2	Strings as arrays . . . . .	66
4.2.1	Copying and concatenating strings . . . . .	69
4.2.2	Comparison of strings . . . . .	72
4.2.3	Classifying characters . . . . .	74
4.2.4	Strings and numerical data . . . . .	76
4.3	Dealing with non-ASCII characters . . . . .	80
4.4	Structures and functions . . . . .	82
4.5	Tools . . . . .	86
4.5.1	The Preprocessor . . . . .	86
4.6	Exercises . . . . .	89
<b>5</b>	<b>Memory Management, Scope in Naming, and More Control</b>	<b>91</b>
5.1	Concepts . . . . .	91
5.1.1	Memory management . . . . .	91
5.1.2	Scope . . . . .	92
5.2	Dynamic memory use with pointers . . . . .	92
5.3	Scope of variables . . . . .	95
5.4	Control structures . . . . .	99
5.4.1	<code>switch</code> . . . . .	99
5.4.2	<code>break</code> . . . . .	102
5.4.3	<code>continue</code> statement . . . . .	104
5.5	Tools . . . . .	105
5.5.1	Interpreting gcc's error messages . . . . .	105
5.6	Exercises . . . . .	106
<b>6</b>	<b>Input/Output</b>	<b>111</b>
6.1	Concepts . . . . .	111
6.1.1	Files and I/O . . . . .	111
6.1.2	Command line arguments . . . . .	113
6.1.3	Environment variables . . . . .	114
6.2	Standard Input/Output . . . . .	114
6.3	Stream input and output . . . . .	116
6.3.1	<code>fopen()/fprintf()/fgets()/fclose()</code> . . . . .	116
6.3.2	Handling errors from library calls . . . . .	118
6.3.3	Skipping around in a file . . . . .	121
6.4	Input and output using file descriptors . . . . .	121
6.4.1	<code>open()/write()/read()/close()</code> . . . . .	121
6.4.2	Handling errors from system calls . . . . .	124
6.4.3	Skipping around in a file . . . . .	125

6.5	mmap()	126
6.6	Making file names from strings	126
6.7	Command line arguments	127
6.7.1	Command line options	129
6.8	Environment variables	129
6.9	Tools	130
6.9.1	Make: building programs from more than one file	130
6.10	Exercises	133
<b>First Interlude</b>		<b>137</b>
<b>7</b>	<b>Linked Lists and Recursion</b>	<b>139</b>
7.1	Concepts	139
7.1.1	Linked lists	139
7.1.2	Recursion	140
7.2	Self-referential structures	140
7.3	2-D Arrays	142
7.4	Recursion	142
7.5	Tools	143
7.5.1	Generating Dependencies in Make	143
7.6	Exercises	143
<b>8</b>	<b>Algorithms, Intermediate Data Structures, and Analysis</b>	<b>149</b>
8.1	Concepts	149
8.1.1	Big-O notation: How long will my computation take?	149
8.1.2	Sorting	151
8.1.3	Basic Data Structures	151
8.1.4	Basic Binary Search Trees	153
8.1.5	Red-Black Binary Trees	153
8.2	C	156
8.3	Tools	156
8.3.1	gprof: profiling execution	156
8.4	Exercises	157
<b>9</b>	<b>Finite State Machines</b>	<b>159</b>
9.1	Concepts	159
9.2	C	159
9.3	Tools	159
9.4	Exercises	159



<b>10 Writing and Testing System Utilities</b>	<b>161</b>
10.1 Concepts . . . . .	161
10.2 C . . . . .	162
10.3 Tools . . . . .	162
10.4 Exercises . . . . .	164
<b>Second Interlude</b>	<b>165</b>
<b>II Achieving Scale in the Real World: Concurrency, Parallelism, Communication and 167</b>	
<b>11 Soft Real-Time Computing</b>	<b>169</b>
11.1 Concepts . . . . .	169
11.2 C . . . . .	169
11.2.1 An Arduino multitasker . . . . .	169
11.3 Tools . . . . .	172
11.4 Exercises . . . . .	172
<b>12 Processes and Pipelines of Programs</b>	<b>173</b>
12.1 Concepts . . . . .	173
12.2 C . . . . .	173
12.3 Tools . . . . .	173
12.4 Exercises . . . . .	173
<b>13 Signals: Communicating Asynchronously</b>	<b>175</b>
13.1 Concepts . . . . .	175
13.2 C . . . . .	175
13.3 Tools . . . . .	175
13.4 Exercises . . . . .	175
<b>14 Threads: Abstracting Processors</b>	<b>177</b>
14.1 Concepts . . . . .	177
14.2 C . . . . .	177
14.3 Tools . . . . .	177
14.4 Exercises . . . . .	177
<b>15 Parallelism: OpenMP</b>	<b>179</b>
15.1 Concepts . . . . .	179
15.2 C . . . . .	179
15.3 Tools . . . . .	179
15.4 Exercises . . . . .	179

<b>Third Interlude</b>	<b>181</b>
<b>16 Communication: IP Networking Basics</b>	<b>183</b>
16.1 Concepts . . . . .	183
16.2 C . . . . .	183
16.3 Tools . . . . .	183
16.4 Exercises . . . . .	183
<b>17 Naming: the Domain Name Service</b>	<b>185</b>
17.1 Concepts . . . . .	185
17.2 C . . . . .	185
17.3 Tools . . . . .	185
17.4 Exercises . . . . .	185
<b>18 Communication: TCP Sockets 1</b>	<b>187</b>
18.1 Concepts . . . . .	187
18.2 C . . . . .	187
18.3 Tools . . . . .	187
18.4 Exercises . . . . .	187
<b>19 Distribution: TCP Sockets 2</b>	<b>189</b>
19.1 Concepts . . . . .	189
19.2 Tools . . . . .	189
19.3 Exercises . . . . .	189
<b>A C Syntax Supplement</b>	<b>191</b>
A.1 Storage Class . . . . .	191
A.1.1 volatile . . . . .	197
A.1.2 const . . . . .	198
A.2 Unions . . . . .	199
A.3 Macros . . . . .	199
A.3.1 Object-like macros . . . . .	199
A.3.2 Function-like macros . . . . .	200
A.3.3 The trap of macros . . . . .	200
A.4 Conditional Compilation . . . . .	201
A.4.1 Using macros . . . . .	201
A.4.2 Platform dependent compilation . . . . .	204
A.5 Operators . . . . .	205
A.5.1 Operator precedence . . . . .	205
A.5.2 Conditional operator for expressions . . . . .	205
A.5.3 = operator . . . . .	206
A.5.4 Increment and decrement operators . . . . .	207

A.5.5 Comma (,) operator . . . . .	208
A.6 Fine Points of Execution Order . . . . .	209
A.6.1 Undefined behavior . . . . .	209
A.6.2 Logical expressions . . . . .	210
A.7 Function Pointers . . . . .	211
A.7.1 Example: the <code>qsort</code> library function . . . . .	211
A.7.2 Array of pointers to functions . . . . .	212
<b>B Development Environment Addendum: MacOS</b>	<b>215</b>
B.1 LLVM . . . . .	215
B.2 Clang . . . . .	215
B.3 LLDB . . . . .	215
B.4 Use of Xcode . . . . .	215
<b>C Development Environment Addendum: Windows</b>	<b>217</b>
<b>D Data Structures and Algorithms Used in the Linux Kernel</b>	<b>219</b>



# Course Introduction

In which we introduce the course and its goals, and its place in the curriculum.

You use computers every day; are you curious how those systems actually work? Do you want to improve them? Do you want to know how software and hardware work together to create the behavior we expect from a computer? Are you curious about the relationship between the beautiful but abstract ideas of “computer science” and the actual device on which you are reading this book (if you are reading it in electronic format)? Perhaps you have heard of, or even use, an open source operating system such as Linux or FreeBSD, and (in a secret dream you only sometimes admit to yourself) you want to contribute to the project, or adapt it to your own needs, or even start your own from scratch. If so, studying computer systems can be enlightening and fun. The first step is to develop some of the fundamental skills of a “systems programmer”, after which it gets easier to study computer operating systems and architecture. In this class, we will learn the basics of life as a systems programmer, while learning the programming language **C**.

The computer you are sitting in front of (as well as your smart phone) contains a large pile of software that can roughly be categorized as the kernel, device drivers, system utilities, libraries, and applications. The core part of the operating system is known as the *kernel*, and the software modules that control the devices attached to it are known as *device drivers*. The kernel controls the actual hardware resources, and programs can request the kernel to do certain things, such as reading and writing files, using functions known as *system calls*. Above the kernel sit many *utility* programs that help the machine boot, configure the network and other devices, manage users, write and compile programs for the machine, and many other tasks. *Libraries* provide useful functions so that you don't have to implement them yourself. *Applications*, such as your web browser or mail client, are the programs with which users most often interact to conduct various tasks – they are the reason that most users buy a particular computer. The technical distinctions between utilities and applications are minor, but very

roughly the system needs the utilities but is otherwise happy without the applications, and the user needs the applications and wants the system to function properly but is otherwise only dimly aware of utilities.

A *systems programmer* is a person who writes code and works within and around the system itself, with little regard for applications. She builds utilities for making life better for *system managers*, who run computer systems for others to use. She may also build libraries that application programmers use, and extend the functionality of the system. Many system managers write code and consider themselves to be systems programmers, as well. A special subset of systems programmers are *kernel hackers*, who dive into the kernel itself; such work is not for the fainthearted, but is fun and prestigious.

Systems programmers often write utilities in *scripting* languages such as Python, Ruby, and Perl, or in what are called *shell* programming languages. The lowest-level tools that manipulate bits or do repair work on the system are often written in the language C, which was developed originally in the early 1970s by Dennis Ritchie at Bell Labs, where he worked with Ken Thompson and Brian Kernighan. C is also the language of choice for those who care about speed and efficiency in execution, for reasons we may discuss during the semester. The core functionality of systems is also often implemented using C. Some examples of software written in C:

- The kernels of operating systems such as FreeBSD and Linux, for example, are written in C.
- Many device drivers (the software that controls hardware devices such as disks or flash drives and keyboards).
- Many classic utilities (though if they were rewritten from scratch today a different language might be chosen).
- Embedded and real-time systems, and software for which execution efficiency and predictability are important.
- Network protocols and utilities that manipulate data at the bit and byte level.
- The run-time systems for some other interpreted languages.

C is also a historically important language, serving as the basis on which the implementers of C++, C#, Objective C, Java and others built larger, richer languages. Simpler languages for devices such as Arduino were developed as subsets of C. The basic syntax of expressions, assignments, loops, and conditionals will look familiar across many languages thanks to this heritage.

Because C is a “low-level high-level language,” C is an excellent tool for learning how the underlying computer itself operates. The CPU of any computer executes individual *instructions*. Instructions themselves are just numbers that tell the CPU whether to add, subtract or move data. They can be written by humans in *assembly language*, but that is tedious and tied to a specific type of CPU. C allows all of the same operations as assembly language, but is more portable and faster to write. However, C allows you to make many mistakes that other languages protect you from!

## 0.1 To the Independent Reader

If you are reading this book on your own, rather than as the text for a class, your reasons for reading and your approach to the text may be rather different than a registered student. You will find this book especially helpful if you are a moderately experienced programmer but with no experience in C, and if you are interested in open source systems such as FreeBSD or Linux. For you, this book should be pretty much self-contained; most of the concepts you need to understand the code and data structures in the Linux kernel are covered here. You will find enough on the C language, as well as computer science concepts such as sorting, binary trees and big-O notation. However, the concepts of operating systems will require substantial further study, and you should follow this book with an introduction to operating systems, such as Tanenbaum.

You can skim, or even skip, the next several sections and pick up at Section 0.6.

## 0.2 Goals and Structure of the Class

Our focus in this class will be on effective use of C for specific tasks. We will learn:

- the basics of the C programming language;
- proper software engineering style;
- some basic data structures in C;
- tools for software engineering;
- proper use of system calls (including managing error returns);
- a little bit about how the underlying system executes your computation;  
and
- how to build software systems such as simple network services.

Every week, in class there will be:

- a short lecture on programming *concepts*;
- in-class *exercises* (your best chance to get help);
- new *tools* introduced;
- *homework* to be done outside of class; and
- later in the semester, we will use the *attendance server* to track attendance.

### 0.3 Learning Outcomes

The Association for Computer Machinery (ACM) maintains a recommended curriculum for undergraduate programs in computer science. The detailed curriculum includes about 1,300 separate learning outcomes across 19 Knowledge Areas. (The KAs do not correspond directly to classes.) Some KAs we address:

- SDF: Software Development Fundamentals
- SE: Software Engineering
- OS: Operating Systems

This class will cover *at least* the following outcomes:

- SDF/Algorithms and Design:
  - 1. Discuss the importance of algorithms in the problem-solving process. [Familiarity]
  - 3. Create algorithms for solving simple problems. [Familiarity]
  - 4. Use a programming language to implement, test, and debug algorithms for solving simple problems. [Usage]
  - 5. Implement, test, and debug simple recursive functions and procedures. [Usage]
  - 8. Apply the techniques of decomposition to break a program into smaller pieces. [Usage]
- SDF/Fundamental Programming Concepts:
  - 1. Analyze and explain the behavior of simple programs involving the fundamental programming constructs variables, expressions, assignments, I/O, control constructs, functions, parameter passing, and recursion. [Assessment]



- 2. Identify and describe uses of primitive data types. [Familiarity]
- 3. Write programs that use primitive data types. [Usage]
- 4. Modify and expand short programs that use standard conditional and iterative control structures and functions. [Usage]
- 5. Design, implement, test, and debug a program that uses each of the following fundamental programming constructs: basic computation, simple I/O, standard conditional and iterative structures, the definition of functions, and parameter passing. [Usage]
- 6. Write a program that uses file I/O to provide persistence across multiple executions. [Usage]
- 7. Choose appropriate conditional and iteration constructs for a given programming task. [Assessment]
- 8. Describe the concept of recursion and give examples of its use. [Familiarity]
- 9. Identify the base case and the general case of a recursively-defined problem. [Assessment]
- SDF/Fundamental Programming Concepts:
  - 1. Discuss the appropriate use of built-in data structures. [Familiarity]
- SDF/Development Methods:
  - 1. Trace the execution of a variety of code segments and write summaries of their computations. [Assessment]
  - 8. Apply a variety of strategies to the testing and debugging of simple programs. [Usage]
  - 10. Construct and debug programs using the standard libraries available with a chosen programming language. [Usage]
  - 12. Apply consistent documentation and program style standards that contribute to the readability and maintainability of software. [Usage]
- SE/Tools and Environments
  - 2. Describe how version control can be used to help manage software release management. [Familiarity]
  - 3. Identify configuration items and use a source code control tool in a small team-based project. [Usage]

Other elements of the ACM CS curriculum are partially covered, and additional items may be covered, as well. Some of these concepts are naturally repeated in different classes in different contexts, and using different programming languages. For more detail, please see the ACM CS2013 curriculum.

## 0.4 To the Instructor

⇒ *This needs to be compared to some notion of CS pedagogy.*

Learning of a new programming language alongside a first introduction to serious programming and CS concepts, proceeds in several phases, for which there seems to be no real shortcut:

1. “Type this, and tell me what happens,” as students learn to edit and recognize basic syntactic constructs;
2. add this exact text to your existing (small) program;
3. make minor modifications to the behavior of a program;
4. create a program from scratch for a simple task (e.g., printing a simple sequence of numbers such as a Fibonacci sequence or the polyrhythm of Exercise 2.4);
5. be able to discuss a simple algorithmic idea (such as sorting) and translate the concept into code; and finally
6. become a true student of computer science (or computer engineering or software engineering), shifting away from “learning to program” and toward learning about the core ideas and how to apply them intelligently, economically and elegantly in order to solve problems that he or she cares about.

Somewhere along the way, the student must learn how to build a program, read and address compiler error messages, and debug simple problems, via `printf()` or a debugger.

These phases constitute the equivalent of learning addition, subtraction, multiplication and division in mathematics. Once these phases have been passed, it becomes possible to move on to algebra and carry on an “adult” conversation about core concepts, and the pace of learning accelerates. However, the transition from step 4 to step 5 above is the most difficult one in learning in CS, and must be managed carefully. It occurs at around the First Interlude in this book, and by the time students reach Chapter 8 should be more or less complete.

## 0.5 This Class in the Keio Shonan Fujisawa Campus Curriculum

This book will very likely serve as the text for both “Fundamentals of System Programming” and “System Programming”, a two-semester sequence. The first

### 0.5. THIS CLASS IN THE KEIO SHONAN FUJISAWA CAMPUS CURRICULUMix

half of the book, “Core Concepts and the C Programming Language in a UNIX Environment,” represents the content of the fundamentals class, and the second half of the book, “Concurrency, Parallelism, Communication and Distribution in Modern Systems,” is the content of the advanced class.

Prerequisites:

- Fundamentals of Information Technology 1
- Fundamentals of Information Technology 2 (recommended or concurrent, but not an official prereq)
- Data Structures and Programming (officially recommended)

This class falls early in the expected sequence of classes at SFC, but should not be your first experience with programming. You are assumed to already be familiar with the basic concepts necessary to express a simple algorithm in a procedural language, including:

- variables, expressions and assignment statements;
- binary and hexadecimal numbers;
- basic Boolean logic (**and**, **or** and **not**);
- concept of a character and character set;
- conditional execution (**if** statements);
- loops (**for** or **while** statements); and
- functions or procedures, with or without arguments and return values.

All of the above are covered in the prerequisite classes listed above. In addition, some basic mathematics are helpful for some of the exercises, as well as to understand some of the abstractions:

- linear algebra: vector addition and dot products, matrix multiplication;
- sequences; and
- the rudiments of probability.

This may be your first experience with *compiled* programs instead of *interpreted* programs.

Courses this class will prepare you for:

- System Programming

- Computer Architecture (n.b.: System Programming recommended, may be taken concurrently)
- Operating Systems (n.b.: System Programming recommended, may be taken concurrently; OS not offered in 2015)
- Computer Graphics

Many labs around SFC build projects on Arduino or Raspberry Pi, two popular, small computing platforms. Learning C and a little about systems will also prepare you to better understand and take advantage of these platforms.

The ultimate goal of this two-class sequence is to prepare you to understand computer systems, including being ready to understand operating system concepts and read the kernel code for an OS such as FreeBSD or Linux. To that end, beyond familiarity with the C language itself (as both of those kernels are written in C) and the fundamental concept of a *pointer*, it is necessary to understand these basic data structures:

- arrays,
- stacks,
- singly- and doubly-linked lists,
- basic binary trees (lookup, insert, delete, and balance), and
- hash tables.

All but the last of these are covered in the first half of this book. Systems also depend heavily on the notion of a *finite state machine* (or just *state machine*), covered in Chapter 9.

## 0.6 Why C?

⇒ *Partially discussed in prior section. Bring in suggestions from a friend.*

Why C? C is a compact and high performance language which has served as much of the inspiration for C, C++, Java, Javascript and a host of other languages. Learning C prepares you to solve most problems in a concise way with the support of a compiler.

## 0.7 Why a New Book?

⇒*To be written. In a few words: 64 bits, IPv6, OpenMP, git. Looking for balance between teaching language and UNIX versus teaching CS.*

The landscape of computing, computer science, and the Internet is in a state of constant evolution. The intent of this book is to familiarize you with the state of the art in C programming from a systems perspective.

## 0.8 How to Read this Book

If an exercise has a number in square brackets, such as [5], after the problem, then that number represents the amount of time the exercise should take, in minutes. This assumes a student doing a reasonable job of keeping pace with the course. Those marked 5 minutes should be doable as fast as you can type. Those marked 120 minutes will take a couple of hours of thinking, writing, and debugging.

Each chapter will have one or more “creative” problems (marked with ✦), typically related to common logic puzzles or math algorithms, and one or more “capstone” problems (marked with ◉). When you can do the capstone problems, you have fairly mastered the material in the chapter.

Chapter capstone problems:

⇒*Still under development.*

- “Hello, world!”
- sort program
- buddy memory allocator
- quine
- finite state machine
- local shell
- parallel matrix multiply
- client for attendance server, or remote shell (n.b.: dangerous!)

⇒*Describe the meaning of the different box types.*

**Note that many of the examples in the book are not complete, or may be inefficient, not robust against errors, or even wrong. This may be done to illustrate a particular point, and those shortcomings are often corrected in later examples or the exercises.**

## 0.9 Additional Recommended Books

A couple of traditional textbooks are listed here, though of course you are not limited to just these books. It's worth noting that some old, good textbooks still have not been updated to a modern 64-bit environment. They may implicitly treat an integer as 32 bit integer, so you need to exercise some caution when reading them.

- B. Kernighan and D. Ritchie, *The C Programming Language*, 2nd edition, is the original book on C, and remains compact and clear, an excellent introduction to the language itself.
- W. Stevens, Stephen Rago, *Advanced Programming in the UNIX Environment*, 3rd edition, ISBN 978-0321637734
- Jon Bentley, *Programming Pearls*, 2nd Edition, ISBN 978-0201657883
- Stephens on TCP/IP
- McCool on structure parallel programming
- Cormen *et al.*, *Introduction to Algorithms*, 3rd ed., is perhaps the most common textbook on algorithms and data structures. As a CS textbook, rather than a language introduction, the examples are all in pseudocode, and translating them into C will take some practice. Over 1,000 pages long, it is an outstanding book, but requires a certain amount of fortitude to attack! However, the individual chapters are very accessible. This is a book worth having as a reference, and for a lifetime of study.

## Part I

# Core Concepts and the C Programming Language in a UNIX Environment





# Chapter 1

## The Traditional “Hello, World!”

In which we briefly compare C to Java, and fly through our first C programs using concepts to be developed in more detail throughout the course. You are assumed to already be familiar with the basic idea of an algorithm, variables, expressions, assignment statements, conditional execution and loops, and functions. However, this may be your first time with the syntax of C.

### 1.1 Concepts

⇒ *To be written.*

### 1.2 Java and C

First, we review the basics of C, which you may have learned already in the “Data Structures and Programming” class, and differences between C and other popular languages such as Java and C++.

#### 1.2.1 Object-oriented languages and procedural languages

Java language is an object-oriented language, in which programming consists of defining an object and methods to manipulate the defined object. So, it is called

an object-oriented language.

In contrast, C is called a *procedural language*. Everything that the computer actually does is specified by the programmer, and in general happens exactly in the order specified. To avoid having to repeat a sequence of commands, they can be grouped into *procedures* or *functions*. C is often referred to as being “close to the hardware,” in that it can be used to manipulate data at very low levels, as is necessary to control devices, and because the actions specified by the programmer are simple primitives close to the way the CPU actually executes instructions.

C has a bad reputation for bugs, especially security-related bugs. Finding these can be very difficult. C assumes that the programmer knows what she is doing, and allows her to do things that other languages might prevent, both for good and for bad. However, proper software engineering practices reduce the incidence of these occurring.

Unlike Java, there isn’t a clear distinction between objects and methods to manipulate the objects. Description of the procedures is a program creation. There is no a concept of an object and methods which are bound to the object.

For example, in Java (an object oriented language), we define the object class “car”, then “step on the accelerator”, and “step on the brake” within its class. So, the “step on the accelerator” method can only manipulate the “car” class. Assume there is another class “refrigerator”. We cannot apply the “step on the accelerator” method to the “refrigerator” class, even if we want to.

However, in C, we define methods for manipulating objects and the objects themselves separately. So, we may create an incorrect program which applies “step on the accelerator” to a “refrigerator” class, usually with bad results.

In this way, the design policies and concepts of programming languages are quite different. There are many, many programming languages and even classes of languages; you should learn several languages during your time here, and learn to apply each when it’s the right tool for the job.

### 1.2.2 Interpreter and Compiler

If you have just started programming, you probably encountered an *interpreted* language as your first programming language. If you have used R, Matlab, Octave, most forms of BASIC, or macros in tools such as Excel, you have used interpreted languages. HTML and JavaScript are also interpreted languages. With an interpreted language, you as the programmer don’t have to do anything special to prepare the program to be executed – you just read it in using another program. That program is called the *interpreter*, and it can read a human-written program directly and figure out what to do with it.

Other languages use a *compiler* to translate a program from the human-readable form into one better for the computer to execute. The input is called

the *source* language, and the output is called the *target* language, or often the *object* or *binary*, if the target language can't directly be read by a human. If the object can be executed directly by the CPU with minimal help from the OS, it is called an *executable*. We will see that getting from C to an executable is a multi-step process, though sometimes the compiler will hide that from us to make our lives simpler.

One of the differences between Java and C is the difference between their execution methods. In Java, first, we create an object program (binary) from a source program by compiling with the Java compiler, called the *class file* in Java parlance. In this case, the target language is not quite appropriate for direct use by the CPU; it is in a form called *byte code*, which requires a Java interpreter to execute, known as the Java Virtual Machine (JVM). The advantage is that this object program written in byte code is in a common, standard format, rather than specific to the type of CPU it our computer, and so can be used in any computer with a Java interpreter.

In contrast, the C compiler creates the executable file from the source program. After that, this file is executed directly on the operating system. Because this executable file is different for every type of computer, if you want to run the program on a different type of computer, you need to compile and create the executable file for that computer.

Let's see a simple example.

```
// Hello.java
class Hello {
    public static void main(String args[]){
        System.out.println("Hello Java World!");
    }
}
```

```
% ls
Hello.java
% javac Hello.java
% ls
Hello.class Hello.java
% java Hello
Hello Java World!
%
```

In this way, in Java, **Hello.class** is created by compiling the source file (**Hello.java**), which is executed by the interpreter **java**.

Next, a C language example is shown.

```
/* hello.c */
#include <stdio.h>
#include <stdlib.h>
int main(){
    printf("Hello C World!\n");
    exit(0);
}
```

```
% ls
hello.c
% gcc hello.c
% ls
a.out hello.c
% a.out
Hello C World!
%
```

In this way, you can compile the source file (`hello.c`) to create the executable file (`a.out`) in the C language, then execute it as a command. The executable file that you created by the C language can be used the same as `ls` command, `date` commands, and other commands. Many of the commands in the UNIX operating system are written in C. For example, there are source files such as `date.c` and `ls.c`, then those are compiled into the executable files, which are `date` and `ls` commands.

**In Java, when you compile a source program, a class file, such as `Hello.class`, is created, named according to the original source file name. However, in C, the executable file name becomes `a.out` unless you explicitly specify a different name. Later, we will learn how to specify a file name for the executable.**

For reference, a program example in the C++ language is shown.

```
#include <iostream>

main(){
    std::cout << "Hello, world!\n";
}
```

To output and display a string on the screen, in C, the `printf()` function is called with the parameter “`Hello C World!\n`”, which is the string to be displayed. The string is quoted with double quotes “ ” as in Java.

### 1.2.3 Similarities

C has many similarities with Java:

- Java borrowed a bunch of C's syntax, especially for operators and markers such as loop and function delimiters.
- What we call *control flow* for simple sequences of statements is the same:
  - loop using a **for** statement;
  - loop using a **while** statement;
  - conditional branches and selections using **if** and **else**; and
  - selections by **switch** and **case** are the same.
  - Exiting and control transfer from a block by **break** or **continue** are the same.
- The basic built-invariable types and usage such as **int** and **double** are almost the same.
- Arithmetic operation and logical operation are almost the same.

### 1.2.4 A Few Differences

#### Using Functions From Elsewhere

In Java, the desire to use another program module is declared by the **import** statement. But in C, the similar statement **#include** is used. Strictly speaking, they are different concepts. However, they are similar in the sense that the use of other functions that are prepared in advance.

#### The Class and Inheritance

Java and C++ have the class and its inheritance. The concept of object-oriented programming languages are implemented in the language. However, the C language does not have classes, nor inheritance. Therefore, to describe a data structure similar to the object-oriented program requires some programming techniques.

#### Pointers and Memory Management

Important functions not in Java, but in C, are pointers and memory management. In the Java language, when you need a new variable or storage area in the program, you create them using the **constructor**. However, in C, you need to create them using pointers and a memory management library. We will study this in later chapters.

## 1.3 C language brief summary

### 1.3.1 Overall Structure of a Program

A program written in C consists of functions. The next example consists of a function `gcd()` which calculates the greatest common divisor of two integers, and a `main()` function.

A `main()` function is the function which is called first and executed. It corresponds to the `public static void main(...)` in the Java language. Other functions are called by the `main()` directly or indirectly, and executed.

Along with functions, declaration of global variables and control statements for the preprocessor are also program components.

More importantly, in C, functions exist in the program independently. They aren't methods which are bound to the objects.

`exit()` is the function which terminates program execution. Its only parameter is the exit status of the program. In normal termination, the parameter is set to 0. We won't describe the details here, but it removes temporary files which were created by the `tmpfile()` function.

### 1.3.2 Compile and Execution

As mentioned before, in the C language, an executable file is created by the compiler. For example, to compile and run a program, first, create a file named `gcd.c` using a text editor such as `emacs`.

```
/* gcd.c */
#include <stdio.h>
#include <stdlib.h>

int gcd(int a, int b){
    int c;
    while(b != 0){
        c = a%b;
        a = b;
        b = c;
    }
    return a;
}

int main() {
    int i,j,k;
    i = 1000;
    j = 35;
    k = gcd(i,j);
    printf("G.C.D. of %d and %d is %d.\n",i,j,k);
    exit(0);
}
```

Next, compile the program and run it.

```
% gcc gcd.c -o gcd
% ./gcd
G.C.D. of 1000 and 35 is 5.
%
```

Note that we used `-o gcd` with our command to compile the program. Historically, for C compilers, if you do not specify this output, it will create a program called `a.out`. You may see and execute the program as follows:

```
% gcc gcd.c
% ls
a.out gcd.c
% ./a.out
G.C.D. of 1000 and 35 is 5.
%
```

### 1.3.3 Comments

The area between `/*` and `*/` is a comment, which doesn't affect the processing of the program. You should write comments which help the readability and

understanding of the program. Also, lines beginning with `//` are comments as well, as in Java and C++.

### 1.3.4 Variable and Type

Before using a variable, it must be declared. As shown below, a declaration consists of variable names and a type for those variables.

```
type variable name, variable name, ...;
```

There are two type of variables, *local variables* and *external variables*. A local variable is defined within a function, and instantiated within the function. An external variable is defined outside of a function, and instantiated such that it is usable across all functions. This relation between local variables and external variables resembles the relation between class variables and instance variables in the Java language, but is not exactly the same relation. An external variable is also called a *global variable*.

The C language grammar strictly distinguishes the terms “declaration” and “definition”. A “declaration” is used to inform the compiler what type a variable has. It may inform the compiler that a variable is an `int` type, or `double` type, or another type for a variable in a program. In contrast, “definition” is used to allocate a storage area for a variable in that location of the program.

As we will learn in a later section, the declaration for a variable may exist at several places in the program. However, the definition for a variable can only exist at one location in the program.

In the following example, an external variable with integer type `int z` is defined outside of the function.



```
/* gcd2.c */
/* Calculate GCD by a modified
 * Euclidean algorithm.
 * written by: Rod Van Meter, 2015/6/1
 * Basic idea:
 *   gcd(a,b) = gcd(a-b,b) for all a>b
 * Tricks:
 *   1. '%' effectively orders a,b on
 *      first iteration
 *      since, for a>b, b%a = b.
 *   2. '%' instantly does N iterations
 *      of Euclidean
 *      algorithm where n=floor(a/b)
 * Computational complexity:
 *   O(1)
 * In:
 *   a,b - *positive* integers only
 *        (range unchecked)
 * Out:
 *   gcd(a,b)
 * Side effects/throws:
 *   none
 */
#include <stdio.h>
#include <stdlib.h>

int z; /* definition of an external variable */

int gcd(int a, int b){
    int c; /* definition of a local variable */
    while(b != 0){
        c = a%b;
        a = b;
        b = c;
    }
    return a;
}

int main() {
    int i,j; // definition of two local variables
    i = 1000;
    j = 35;
    z = gcd(i,j);
    printf("G.C.D. of %d and %d is %d.\n",i,j,k);
    exit(0);
}
```

Note that this time, we have added an extensive description of what the program does and how it does it in the comments at the top of the program. These comments describe the inputs and output of the function `gcd()`, and any limitations (in this case, warning you that it may not work properly if you give it 0 or a negative number as an argument).

As shown, local variables are defined between `{` and `}` within a function. External variables are defined outside the `{` and `}` of any function.

The `printf()` function displays the variable `z` that holds the result of calculation. Its first argument is a string of characters called the “format string.” Usually, characters within “ ” are displayed as a string literally, exactly the way you typed them. However, `%` within “ ” and the following characters are exceptions. This part is substituted by the value which is designated by the variable arguments that come after the format string. When you want to print a variable as an integer, `%d` is placed within “ ” as a designator. When an argument is a floating point number, `%f` is placed within the format string as a designator. Other designators are learned in later chapters.

**Why are they called “external” variables, even though they are located within a file such as `gcd2.c`? As we will see in later chapters, such variables can be referred to and used by other functions written in other files, so they are called “external” variables.**

To display the value of variable `z`, the `printf()` function is used. This `printf()` function is one of the standard input output library functions, but this is not part of the C language. The C language itself does not define input or output. The `printf` function is defined as a part of the standard library of functions that are accessible by all C programs. We will look at this library in more detail later.

### 1.3.5 Function

A program consists of several functions. A function is defined in the following example. In this example, the function calculates the mean of two numbers.

```
double mean(double x, double y) {
    double c;
    c = (x + y)/2;
    return c;
}
```

This program defines the function `mean` which has two real numbers (floating point numbers) as arguments, and returns a floating point number as the

return value. 'x' and 'y' are called *parameters*, or formal parameters, or dummy arguments. When this function is actually called, the arguments' values are substituted into these parameters, then the function is executed. The arguments are called actual arguments when a function is actually called.

A function body consists of the definition of local variables (in this example, `double c;`) and statements which calculate a value.

A function execution flows from the head of the function to the tail of the function where the `return` statement is located. The value of the variable at `return` becomes the return value of the function. After execution of `return`, the execution of the function completes, and the control is returned to the point at which the function was called. Statements after a `return` are ignored.

You may write a formula as the argument to `return`, so the following example will give the same result as the previous example.

```
double mean(double x, double y) {
    return (x + y)/2;
}
```

Furthermore, a function without a return value should be declared to have the `void` type. The following example calculates the mean of two numbers, but displays the mean in the function, and does not return any value to the caller.

```
void mean(double x, double y) {
    double c;
    c = (x + y)/2;
    printf("c=%d\n",c);
}
```

A complete program using the function which calculates the mean of two numbers is shown.

```
/* mean.c */
#include <stdio.h>
#include <stdlib.h>

double mean(double x, double y) {
    double c;
    c = (x + y)/2;
    return c;
}

int main() {
    double a,b,c; /* local variable definition */
    a = 12.0;
    b = 5.0;
    c = mean(a,b);
    printf("The mean of a and b is  %f.\n",c);
    exit(0);
}
```

You get the result after compiling and execution as below.

```
% gcc mean.c -o mean
% ./mean
The mean of a and b is  8.500000.
%
```

### 1.3.6 Function and Types

In the above example, the function `int gcd()` that calculates the greatest common divisor, and the function `main()`, which is executed first, are defined. In this case, `gcd()` is called by `main()`, and its definition appears before `main()`. From the compiler’s point of view, the only important factor is that the types of any arguments and the return value of the function have to be declared before it is called.

If you want to put a function later in a file, then you need to indicate explicitly the types of the return value and arguments before the definition of any function (in this case, `main()`) that calls it. This is called the *prototype declaration*.

```
/* gcd3.c */
#include <stdio.h>
#include <stdlib.h>

/* function prototype declaration */
int gcd(int, int);

int main() {
    /* first, definition of local variables */
    int i,j,k;
    /* now the code, beginning by assigning
       values to some variables */
    i = 1000;
    j = 35;
    k = gcd(i,j);
    printf("G.C.D. of %d and %d is %d.\n",i,j,k);
    exit(0);
}

int gcd(int a, int b){
    /* definition of local variables */
    int c;

    /* now the loop */
    while(b != 0){
        c = a % b;
        a = b;
        b = c;
    }
    return a;
}
```

If there is no prototype declaration, the return value of a function without a prototype declaration is treated as type `int`. So, in this case, the program executes properly.

```
/* example5.c */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i,j,k; /* definition of local variables */
    i = 1000;
    j = 35;
    k = gcd(i,j);
    printf("G.C.D. is %d.\n",k);
    exit(0);
}

int gcd(int a, int b){
    int c; /* definition of local variables */
    while(b != 0){
        c = a%b;
        a = b;
        b = c;
    }
    return a;
}
```

However, when you compile the program with `-Wall` option, a warning message is displayed at compilation. You should realize that you forgot the necessary prototype declaration. In this case, `a.out` is still created, and you can execute the program.

```
% gcc -Wall example5.c
example5.c: In function 'main':
example5.c:9: warning: implicit declaration
of function 'gcd'
%
```

For `example5.c`, assuming an `int` type without a prototype declaration produces a correct compilation. So, the program executes properly even though the compiler gave a warning message. However, here is an example that causes incorrect compilation.

```

/* example6.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
    int a,b;
    a = 10;
    b = f1(a);
    printf("b=%d\n",b);
    exit(0);
}

double f1(int v){
    double x;
    x = v * 3.141;
    return x;
}

```

The compilation of this program causes the following errors.

```

% gcc examle6.c
example6.c:13: error: conflicting types for
'f1'
example6.c:8: error: previous implicit
declaration of 'f1' was here
%

```

When you compile the program on an older computer system, you may compile and execute as follows.

```

% gcc example6.c
example6.c:13: warning: type mismatch with
previous implicit declaration
example6.c:8: warning: previous implicit de
claration of 'f1'
example6.c:13: warning: 'f1' was previously
implicitly declared to return 'int'
% a.out
b=10
%

```

The output should be computed as  $3.14 \times 10$  is 31.4, then truncated to the integer 31. However, in this example, 10 is displayed as the result.

When you look at the warning message in the example program that cannot

be compiled, you realize that the return type of function `f1()` is treated as an implicit type `int` at line 8, but the actual definition is type `double` in line 13.

To correct this example, you need to use a prototype declaration for the function first.

```
/* example6good.c */
#include <stdio.h>
#include <stdlib.h>

double f1(int);

int main(){
    int a,b;
    a = 10;
    b = f1(a);
    printf("b=%d\n",b);
    exit(0);
}

double f1(int v){
    double x;
    x = v * 3.141;
    return x;
}
```

When you compile this example, you get the correct value 31.

```
% gcc example6good.c
% a.out
b=31
%
```

### 1.3.7 Automatic Type Conversion

In the C language, a value always has a type. As in Java, there is an `int` type that indicates an integer number, and a `double` type that indicates a floating point number. Furthermore, there is a `char` type that is for character handling.

The C language can convert the types more easily than Java language.



```
// convert.java
class convert {
    public static void main(String args[]){
        int i;
        double a;
        char name;
        i = 100;
        a = i;
        name = i;
    }
}
```

Compiling this Java program produces an error message at the substitution of `i` for `name`.

```
% javac convert.java
convert.java:9: Incompatible type for =.
Explicit cast needed to convert int to char.
    name = i;
           ^
1 error
%
```

The same program is written in C as follows.

```
/* convert.c */
#include <stdlib.h>
int main(){
    int i;
    double a;
    char name;
    i = 100;
    a = i;
    name = i;
    exit(i);
}
```

This program can be compiled as-is. As this example shows, the language C has flexibility in handling types, everything is up to the programmers. So, you have to keep in mind this flexibility and be careful.

### 1.3.8 Statement

A function consists of several statements. There are several types of statements.

**Assignment Statement**

```
variable = expression;
```

This computes a right-hand side value, then assigns the value to a left-hand side variable.

```
x = y+z-3*a;
```

You may use an element of an array (described later) instead of `variable`.

**Conditional Statement**

```
if (expression) {
    statement 1; ...
} else {
    statement 2; ...
}
```

This computes the value of `expression`. If the value is not 0, the program executes `statement 1`. Otherwise if the value is 0, the program executes `statement 2`.

```
if (x == 1) {
    y = 3;
    z = 2;
}
else {
    y = 5;
    z = 4;
}
```

If you test multiple conditions, `else if` is inserted as follows.

```
if (expression 1) {
    statement 1; ...
} else if (expression 2) {
    statement 2; ...
} else {
    statement 3; ...
}
```

This example first computes the value of `expression 1`. If it is not 0, it executes `statement 1`. If the value of `statement 1` is 0, it computes a value of `statement 2`. If it is not 0, it executes `statement 2`. If both of the values of `statement 1` and `statement 2` are 0, it executes `statement 3`.

You may insert an arbitrary number of `else if` clauses.

**Loop(1)**

```
while (expression) {
    statement; ...
}
```

Computes `expression`. If the value is not 0, execute `statement`, then compute the `expression` again. This iterates until the value becomes 0.

```
x = 0;
while (x < 10) {
    printf("%d\n", x);
    x = x+1;
}
```

**Loop(2)**

```
for (expression 1; expression 2; expression 3){
    statement; ...
}
```

First, this evaluates `expression 1`. Then, it evaluates `expression 2`. If the value is 0, this execution ends. If the value is not 0, it executes `statement`, then evaluates `expression 3`. After that, evaluates `expression 2` again, tests the value is 0 or not. If the value is 0, the execution ends. If the value is not 0, it executes `statement` again, and evaluates `expression 3` again. It iterates this process.

```
for (x = 0; x < 10; x = x+1) {
    printf("%d\n", x);
}
```

Unlike Java, in an old C language standard (ISO C90), you cannot declare loop control variables in `expression 1`. In this case, you need to define variables for loop control outside the `for` statement. Therefore, the following example the compiler to report a grammatical error in the C90 standard.

```
#include <stdio.h>
int main(){
    for (int x = 0; x < 10; x = x+1) {
        printf("%d\n", x);
    }
    return 0;
}
```

Compilation results as follows.

```
% gcc hello.c
hello.c: In function 'main':
hello.c:3: error: 'for' loop initial
  declaration used outside C99 mode
%
```

However, we can compile without error under the newer standard of the language C, ISO C99. If you want to compile with `gcc`, you need to add `-std=c99` option for telling the compiler to apply the standard as follows.

```
% gcc -std=c99 hello.c
%
```

### Return statement

```
return expression;
```

This computes `expression`, the value becomes the return value of the function, and returns the execution control to the point at which the function was called.

### 1.3.9 Expression

There are several types of expressions.

- variable: `v`
- reference to arrays: `v[i]`
- basic arithmetic operations, arithmetic remainder: `x+y`, `x-y`, `x*y`, `x/y`, `x%y`
- comparison: `x>y`, `x<y`, `x>=y`, `x<=y`
- equal, not equal: `x==y`, `x!=y`
- bitwise logical and, or: `x&y`, `x|y`
- logical product, logical add: `x&&y`, `x||y`
- add one, subtract one: `v++`, `v--`, `++v`, `--v`
- function call: `f(x,y,...)`

Comparison, equal, and not equal return 1 when a condition is satisfied, and return 0 when a condition is not satisfied. In Java, `boolean` type indicates true/false. However, in C, integer 0 and 1 (technically, any non-zero value). This

also shows that the C language is not as strict about types. So, you need to be careful about type handling and conversion.

### 1.3.10 Library

In Java, `import` statements enable other prepared functions to be used.

```
// Applet package
import java.awt.*;
import java.applet.Applet;
public class MyApplet extends Applet {
    .....
}
```

In C, a group of functions that are prepared in advance is called a **library**. There are many libraries. The first library we have seen is the standard input/output library, which contains the functions (`printf()`, `fprintf()`, `scanf()`, `fscanf()`, `fopen()`, `fclose()`, `fgets()`, `gets()`, `fputs()`, `puts()`). These functions display characters, accept characters from a keyboard, and read and write files.

Another important library is the character string library, which contains `strcmp()`, `strncmp()`, `strcat()`, `strcpy()`, and `strlen()`. These functions compare strings, change character case from upper to lower, and calculate the length of a string.

A square root function is included in a mathematical library.

To use libraries, you need to declare your intent to use the libraries in a program. In the C Language, you use `#include` at the beginning of a program.

```
#include <stdio.h>    /* standard input/output */
#include <math.h>    /* mathematical library */
#include <string.h>  /* strings library */

main(){
    ....
    ....
}
```

Furthermore, you may need to specify libraries at compilation. The standard input output library and strings library are automatically sought by the compiler. However, for other libraries, you need to specify which ones you want to include by using `-l` at compilation. For example, if you want to use the mathematical library known as `m`, you need to type the following command to compile programs.

```
% gcc program.c -lm
```

Technically, the distinction is that the `#include` statement tells the compiler what functions are available to be used as part of the library, and the `-l` option tells compiler<sup>1</sup> which libraries to include in the final binary it is building.

### 1.3.11 Standard Input/Output Library

By now, you have seen the basic use of the `stdio` library a couple of times, but let's look a little closer. To input numbers and characters from the keyboard, or to display numbers and characters on a screen, the **Standard Input/Output Library** is used. Furthermore, it is used to read numbers and characters from a file, or to write numbers and characters into a file in a program. To use the standard input output library, you specify `#include <stdio.h>` at the beginning of a program.

`printf()` function displays characters. `printf()` example is shown below.

```
/* printfexample.c */
#include <stdio.h>
#include <stdlib.h>
int main(){
    int i,j;
    double a,b;
    char *name = "Hello";
    i = 10; j = 12345;
    a = 10.5; b = 0.0123;
    printf("i=%d, j=%d\n",i,j);
    printf("a=%f, b=%f\n",a,b);
    printf("name=%s\n",name);
    exit(0);
}
```

You get the following result.

```
% gcc printfexample.c
% a.out
i=10, j=12345
a=10.500000, b=0.012300
name=Hello
%
```

You need to specify different specifiers to display integers (`int`), floating point numbers (`double`), and strings (`char *`). The details can be found by using the Unix manual pages, or of course by searching the web. This command will search section 3 of the Unix manual for the description of the function `printf()`.

<sup>1</sup>Actually, the phase of compilation known as *linking*.

```
% man 3 printf
(lots of output)
```

## 1.4 Tools

So far, we have seen the most fundamental tool, the C compiler itself. If you have been working the examples, you have also begun to familiarize yourself with a text editor useful for programming; students in this class often use Emacs, vi, or nano. We are going to need three more major tools: an automated system for building software (we will use the Unix standard `make`), a revision control system (we will use `git`), and a debugger (we will use `gdb`).

### 1.4.1 Gcc

The version of the C compiler we are using is `gcc`. It has many options for controlling its behavior; we will see more of them throughout the semester.

In the examples so far, we have just let the compiler create its default output file, `a.out`. But from now on let's use the `-o` option to create the file we want instead.

```
% gcc -o example example.c
```

### 1.4.2 Make

So far, we have only been using one source code file for each program, but complicated programs (like the Linux kernel) may have thousands. `make` saves us from having to remember which things need to be compiled, by applying a set of rules to figure out what output file is made using which input file.

Put this in a file called `Makefile`:

```
example: example.c
    cc -o example example.c
```

(Note that you have to have a real tab character at the start of that second line.)

Now do the following:

```
% make
```

<code>cat</code>	Prints out the contents of a file.
<code>cp</code>	Makes a copy of a file. (Can also put more than one file into one output file.)
<code>gcc</code>	Compiles a C program.
<code>gdb</code>	Helps you debug programs.
<code>ls</code>	Lists the files in a directory.
<code>make</code>	Builds programs for you.
<code>man</code>	Print “manual” pages describing commands – including these!
<code>more</code>	Prints the contents of a file a screen at a time.
<code>rm</code>	Deletes or removes a file.

Table 1.1: Some common UNIX utilities you will use in this class.

If your file `example.c` is older than the program `example`, you should see `make` actually execute the `cc` command *for* you.

### 1.4.3 man (the Unix manual)

Just above, we saw the initial use of `man`. This is an ancient, standard way of finding information about a system call or library function in Unix systems. It may seem simpler these days to simply search the web, but be careful that you are seeing the man page corresponding to the OS and version you are using.

### 1.4.4 Other UNIX utilities and commands

## 1.5 Exercises

**1.1** Write a program `ex1-1.c` that calculates the following number sequence. Calculate the sequence to the 30th number.

1 1 2 3 5 8 13 . . . . . (Do you recognize this sequence? What happens when you use a different pair of numbers instead of 1, 1 for the first two?)

**1.2** Referring to `example1.c`, write a program `ex1-2.c` that calculates the greatest common divisor and the least common multiple of two numbers. To find the greatest common divisor, the function `gcd()` in the text can be used. Since we haven’t yet worked with input from the keyboard, insert the two numbers directly into your program. Do the same thing in the following exercises 1.3 and 1.4.

**1.3** In Japanese primary school, you should learn about Tsuru Kame San problem, obtaining the respective numbers of cranes and tortoises from the total



of their heads and legs – assuming, of course, that cranes have two legs, and tortoises have four legs. Write a program `ex1-3.c` that calculates Tsuru Kame San. In the program, two numbers, the total of heads and the total of legs are specified. Then the program outputs the number of cranes and the number of tortoises.

- 1.4** If a natural number is greater than 1, and has no positive divisors other than 1 and itself, the number is called a prime number. For example, 2, 7, and 19 are prime numbers. Write a program `ex1-4.c`, that determines whether a number is a prime number or not. There are various methods to determine whether a given number  $n$  is prime. For example, by iterating the divisor from 2 to (the number minus 1), you can test whether the number is divisible by a divisor without a remainder. If all of them are not divisible, the number is a prime number.



## Chapter 2

# Fundamental Data Types

In which we discuss the strengths and limitations of integers and floating point numbers. We start building our first real looping programs from the definition of the problem.

### 2.1 Concepts

⇒ *To be written.*

### 2.2 Integer Data Types

The language C has several types of integer data with different size and signs (plus/minus) as follows. The character type (char) is treated as a special case of the integer types. And, the actual size of integer type varies by implementation.

Type	Explanation
<code>char</code>	smallest addressable unit of the machine that can contain basic character set. Actual type can be either signed or unsigned depending on implementation 1 byte
<code>short</code>	short signed integer type. At least 16 bits in size. -32768 ~ 32767 (16 bits)
<code>int</code>	basic signed integer type. At least 16 bits in size.
<code>long</code>	long signed integer type. At least 32 bits in size. -2147483648 ~ 2147483647 (32 bits)
<code>long long</code>	long long signed integer type. At least 64 bits in size. -9223372036854775808 ~ 9223372036854775807 (64 bits)

Type	Explanation
<code>unsigned char</code>	same as <code>char</code> , but unsigned 0 ~ 255 (when 1 byte is 8 bits)
<code>unsigned short</code>	unsigned integer type. At least 16 bits in size. 0 ~ 65535 (16 bits)
<code>unsigned int</code>	same as <code>int</code> , but unsigned
<code>unsigned long</code>	unsigned integer type. At least 32 bits in size. 0 ~ 4294967295 (32 bits)
<code>unsigned long long</code>	unsigned integer type. At least 64 bits in size. 0 ~ 18446744073709551615 (64 bits)

To display an unsigned integer or a `long long` type integer using the `printf`

```

% gcc -Wall -o long long.c
long.c:12:7: warning: integer constant is so large that it is unsigned
long.c: In function 'main':
long.c:12: warning: this decimal constant is unsigned only in ISO C90
long.c:13: warning: format '%d' expects type 'int', but argument 2 has type
'long long unsigned int'
long.c:14: warning: format '%u' expects type 'unsigned int', but argument 2 has type
'long long unsigned int'
% ./long
p=-1
p=4294967295
a=-1
a=4294967295
a=-1
a=18446744073709551615
%

```

function, you need to pay attention to using the correct format specifier.

```

/* long.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
    unsigned int p;
    unsigned long long a;

    p = 4294967295;
    printf("p=%d\n",p);
    printf("p=%u\n",p);
    a = 18446744073709551615;
    printf("a=%d\n",a);
    printf("a=%u\n",a);
    printf("a=%lld\n",a);
    printf("a=%llu\n",a);
    exit(0);
}

```

You get the following result after compiling and run this program.

To suppress the errors on line 9 and line 12, you need to add `U` or `ULL` to specify an unsigned integer. The warnings at line 13 and 14 occurred because we are attempting to display signed integers with the format string specifiers `%d` and `%lld`, but `p` and `a` are defined as unsigned integers.

The warning messages from line 9 and line 12 do not occur when you add `U` or `ULL` to integer constants.

```
/* long2.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
    unsigned int p;
    unsigned long long a;

    p = 4294967295U;
    printf("p=%d\n",p);
    printf("p=%u\n",p);
    a = 18446744073709551615ULL;
    printf("a=%d\n",a);
    printf("a=%u\n",a);
    printf("a=%lld\n",a);
    printf("a=%llu\n",a);
    exit(0);
}
```

In this way, the range of integers varies from type to type.

The next program calculates factorials and displays the results. Run the program with various types of integers for `a` in which the result is stored.

```
/* longfact.c */
#include <stdio.h>
#include <stdlib.h>
int main(){
    unsigned int i;
    unsigned long long a;

    a = 1;
    for(i=1;i<30;i++){
        a = a * i;
        printf("%llu\n",a);
    }
    exit(0);
}
```

## 2.3 Floating Point Number Data Types

We used `double` to define a floating point number. Another type is `float`, which is 32 bits on most systems. There is very limited use of floating point numbers in this text, so only a brief explanation is given.

Type	Explanation
<code>float</code>	in most systems, IEEE 754 single precision floating point format $\pm 1.17549435 \times 10^{-38} \sim$ $\pm 3.40282347 \times 10^{38}$
<code>double</code>	in most systems, IEEE 754 double precision floating point format $\pm 2.2250738585072014 \times 10^{-308} \sim$ $\pm 1.7976931348623157 \times 10^{308}$
<code>long double</code>	in many systems, extended floating point format. IEEE 754 quadruple floating point format, $\pm 3.36210314311209350626$ $2677817321752603 \times 10^{-4932} \sim$ $\pm 1.18973149535723176508$ $5759326628007016 \times 10^{4932}$

Some computer systems use different floating point number formats. The header file `float.h` has some definitions for floating point numbers that the system can handle. Some examples are shown below. In most recent 64 bit systems, these same values are usually used.

In C, the notation `#define DBL_MIN 2.2250738585072014E-308` is called a MACRO. At the time of compilation, the literal string `DBL_MIN` is substituted with the literal string `2.2250738585072014E-308`. This shows the absolute minimum value that a double precision floating point number can represent.

```
...
#define DBL_MIN          2.2250738585072014E-308
#define DBL_MIN_10_EXP  (-307)
#define DBL_MAX_EXP      1024
#define DBL_MAX          1.7976931348623157E+308
#define DBL_MAX_10_EXP   308
...
```

## 2.4 Mixed arithmetic operation for an integer and a floating point number

The C language allows types to be mixed in expressions, and permits operations that result in type conversions happening implicitly in some cases. If a floating point number and an integer are mixed in an arithmetic expression, an integer is converted to a floating point number before the operation. For example,  $3.5 + 1$  is computed as  $3.5 + 1.0$ . In an assignment, a conversion is implicitly carried out according to the type of the left hand side variable.

```
int a;
double b;

a = 3.0; /* a == 3 (convert to integer) */
a = 1.5; /* a == 1 (convert to integer) */
b = 3; /* b == 3.0 (convert to floating
        point number) */!!
```

Other than these implicit type conversion, a **cast** is used for explicitly conversion.

The following example converts the value of an expression to **int**.

```
(int) expression
```

The following example converts a value of expression to **double**.

```
(double) experssion
```

The following example converts an integer **i** to **double**, then divides it by 2, and assigns it to the variable **x**. In the program, the integer 2 is converted to **double** implicitly.

```
int i;
double x;
x = ((double)i) / 2;
```

If a division is computed without casting to **double**, it is computed as an integer division, and any remainder is abandoned. The result is different from floating point division, so you must be careful to do the conversion first.

## 2.5 Manipulating bitfields and bits within numbers

⇒ *Checking for a power of two, finding powers of two above and below. Testing bits.*



## 2.6 Tools

### 2.6.1 More on Make

Since we now have more than one program we are writing, you might be interested in the fact that a single Makefile can be used to build more than one program. Of course, it is best if they are related in some way, otherwise you are better off keeping them separate!

```
CFLAGS=-Wall

all: hello typesizes

hello: hello.o
    $(CC) $(CFLAGS) -o hello hello.o

hello.o: hello.c
    $(CC) $(CFLAGS) -c hello.c

typesizes: typesizes.o
    $(CC) $(CFLAGS) -o typesizes typesizes.o

typesizes.o: typesizes.c
    $(CC) $(CFLAGS) -c typesizes.c
```

Note the appropriate use of whitespace, both tabs and blank lines. Note the use of variables within the Makefile, such as `CFLAGS`, and how they are referenced. This makes it easy to update how a whole group of files are to be compiled.

A Makefile is a set of *rules* for how to build programs. The first line of a block specifies the *target* before the colon, then the *dependencies* (there may be more than one). The lines below, which must start with a tab, specify the commands to execute in order to build the target.

Later we will see how the dependencies can be created and maintained automatically.

## 2.7 Exercises

⇒ Add some bitfield exercises, aimed at supporting Exercise 5.3.

**2.1** `fprange.c`: 1. Min: Write a program that starts with `float x = 1.0`, and divides it by two repeatedly until it becomes zero. (Trust me, this will happen. *Why* does this happen?) 2. Max: Do the same thing getting *larger* until something happens. (What happens? *Why*?) 3. Repeat with `double`.

- 2.2** Write a program `ex2-1.c` that defines two integer variables as local variables in `main`, passes those two as arguments to a function, then assigns the number which is not smaller than the other number to the first argument, and assigns the remaining number to the second argument. For example, when `x=10,y=12` at the top of the `main`, after the function call, those become `x=12,y=10`.
- 2.3** In the previous exercise, write a program similar to 2.2 with three integer variables instead of two integer variables. In this program, those numbers should be arranged from larger variables to small variables.
- 2.4** ★ In a musical *polyrhythm*, separate parts are played at the same time using different time signatures. Write a program that prints out the first 100 steps of a polyrhythm. It should print “Fizz” on any multiple of 3 and “Pop” on any multiple of four, and “FizzPop” on multiples of both. For other numbers, it should print a simple timekeeping sound, such as “Tap” or “Chick”. For extra credit, demonstrate to your instructor that you can sing or play the polyrhythm!
- 2.5** ♯ Write a program to calculate the value of  $\pi$  (pi), using both the Gregory-Leibniz and Nilakantha methods. Print out the calculated value for every term through the first fifty. Compare the relative error.

Gregory-Leibniz:

$$\pi = 4 \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{2k-1} = 4\left(1 - \frac{1}{3} + \frac{1}{5} \cdots\right) \quad (2.1)$$

Nilakantha:

$$\begin{aligned} \pi &= 3 + \sum_{k=1}^{\infty} \frac{(-1)^{k+1} \times 4}{2k \times (2k+1) \times (2k+2)} \\ &= 3 + \frac{4}{2 \times 3 \times 4} - \frac{4}{4 \times 5 \times 6} \cdots \end{aligned} \quad (2.2)$$

## Chapter 3

# Arrays, Structs and Pointers

In which we meet *arrays* and *pointers*, which are the bane of every programmer's existence. However, they are also insanely useful, and you should learn to use them well in order to master C. Pointers correspond to the *memory addresses* that the CPU uses when retrieving data from main memory. We will also catch our first glimpse of *structs*, or complex *data structures*, which will recur throughout the course.

### 3.1 Concepts

⇒ *To be written.*

### 3.2 Arrays

Arrays are a convenient way to handle many items of the same type. You may consider arrays to be vectors in mathematics, or tables. Arrays are defined as follows.

```
type array_name[size], ...;
```

For example, an array of 10 integer values is defined as follows.

```
int a[10];
```

To refer to an element of an array, an index (or subscript) is used. For example, to use the 4th element of the above array `a`, you specify the element as `a[3]`. You can any integer expression as an index. Index 0 specifies the first

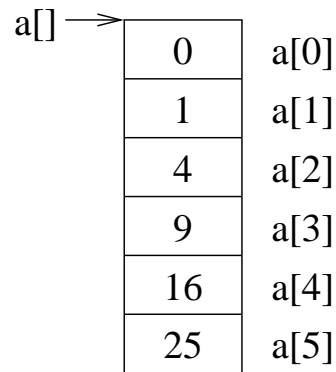


Figure 3.1: The first six elements of array `a` in `array1.c`. The name `a` itself can be used as a pointer to the start of the array.

element, and `size - 1` specifies the last element. If an index is out of this range, the result is undefined. *Be careful, the compiler will let you do this, and it will cause hard-to-find bugs!*

The following example defines an integer array of size 10, assigns 1 to each element by looping, then displays them all using a second loop.

```

/* array1.c */
#include <stdio.h>
#include <stdlib.h>
int main(){
    int i, a[10];
    for(i=0; i<10; i++){
        a[i] = i*i;
    }
    for(i=0; i<10; i++){
        printf("a[%d] = %d\n",i,a[i]);
    }
    exit(0);
}

```

The result of this program is the following.

```

% ./array1
a[0] = 0
a[1] = 1
a[2] = 4
a[3] = 9
a[4] = 16
a[5] = 25
a[6] = 36
a[7] = 49
a[8] = 64
a[9] = 81
%

```

The first `for` statement assigns values from 0 to 9 to a variable `i`. In the second loop, we use `printf()` with a format string constructed for displaying the name and index like `a(5)` before printing the value of the variable.

*However*, the above program is actually *very poor programming practice*. Why?

The size of our array is 10. That number 10 appears in *three* separate places in the program. What happens if we decide to make it 20? Then we must change it in *all three places*. If we miss one, the program will behave improperly. A more maintainable, and readable, form of the program is:

```

/* array1corrected.c */
#include <stdio.h>
#include <stdlib.h>
#define ARRAYSIZE 10
int main(){
    int i, a[ARRAYSIZE];
    for(i=0; i<ARRAYSIZE; i++){
        a[i] = 1;
    }
    for(i=0; i<ARRAYSIZE; i++){
        printf("a(%d) = %d\n",i,a[i]);
    }
    exit(0);
}

```

The following example shows what happens if you specify an index out of the array range. Such errors are common in C programs, but they are less likely to happen if you use the `#define` preprocessor statement effectively.

```
/* arrayerror.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
    int i, a[100];
    for(i=0; i<100; i++){
        a[i] = i;
    }
    for(i=0; i<10000; i=i+100){
        printf("a[%d]=%d\n",i,a[i]);
    }
    exit(0);
}
```

The size of array is 100, so indices vary from 0 to 99. Basically in the C language, a compilation error does not occur even when a program tries to use indices out of the range.

At the run time, an error occurred as follows. **Segmentation fault** is displayed. However, the program terminates not necessarily at the first element beyond the range, but at some later point in execution.

```
% ./arrayerror
a[0]=0
a[100]=-1073742836
a[200]=50
a[300]=1028083265
Segmentation fault
%
```

The array access is done without *bounds checking* (checking the indices and size) at either compile time or run time, allowing efficient execution, but you need to pay extra attention when programming.

In recent systems, some index checking can be performed. Debugging tools which we will describe later also enable index checking at run time.

The following program does not cause a Bus Error. So, you may not realize there is an error in the program.

```
/* arrayerror2.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
    int i, j, k, a[5];

    for(i=0; i<5; i++){
        a[i] = i;
    }
    i = 10, j = 11, k = 12;
    printf("i=%d,j=%d,k=%d\n",i,j,k);
    a[9] = 20, a[10] = 21, a[11] = 22;
    printf("i=%d,j=%d,k=%d\n",i,j,k);
    exit(0);
}
```

This program uses an out of range `a[9]`. The result is shown below.

```
% ./arrayerror2
i=10,j=11,k=12
i=22,j=21,k=20
%
```

The variables `i`, `j`, `k` and `a` are stored somewhere in memory, in this case, on the *stack*. (We will discuss stacks more later.) The exact layout will vary depending on many things, but in the execution shown here, `i` was in memory in the place where `a[11]` would have been, *if* `a` had been that large. In this case, `k` is stored after the location of `a[4]`. So, after 20 is assigned to `a[9]`, displaying `k` results in the value 20. This kind of *memory corruption* is a common type of bug associated with mismanaged memory in C.

### 3.3 Pointers

During the execution of programs written in C, data are stored in memory on the computer. To identify the location in memory, each memory has an *address*. The *pointer* indicates the address.

It is not strictly true that *all* data are stored in memory; they may instead be stored in the CPU's *registers* if they are used frequently. On modern CPUs, registers do not have an address. Thus, if the programmer writes a program using a pointer to a particular variable, the compiler is forced to assign the variable to being in memory rather

than a register. This distinction will not be important during this class.

### 3.3.1 Pointer explanation and value

For example, an integer variable `x` is defined.

```
int x;
```

This variable exists at some location in the memory system of the computer. The number which indicates the location at which the variable exists is called the address or memory address. When you want to use the address of the variable, rather than the value of the variable, place an ampersand in front of the name:

```
&x
```

The following example shows the value of the location of a variable. Of course, addresses are different in different computer systems. Note the `%p` specifier for displaying a pointer in the `printf()` format string.

```
/* pointer.c */
#include <stdio.h>
#include <stdlib.h>
int main(){
    unsigned int i;
    i = 10;
    printf("i = %d, &i = %p\n",i,&i);
    exit(0);
}
```

The result run at `ccz00` is shown below.

```
ccz00% ./pointer
i = 10, &i = ffbff79c
```

The result run at `zmac???` is shown below.

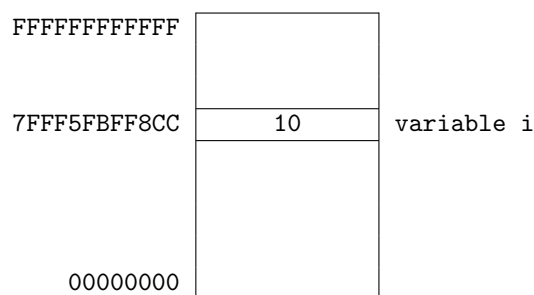
```
zmac000% ./pointer
i = 10, &i = 0x7fff5fbff8cc
```

These examples show `&i` in the `printf` with `%p` specifier. In case you are not familiar with them, the following table shows some decimal numbers and corresponding hexadecimal numbers.



Hexadecimal	Decimal
0 ~ 9	0 ~ 9
A	10
B	11
C	12
D	13
E	14
F	15
10	16
100	256
1000	4,096
FFFF	65,535
FFFFFF	16,777,215
FFFFFFFF	4,294,967,295

In most systems, depending on their computer architecture and operating systems, the C language uses memory address from 0 to 0xffffffff, or to 0xffffffffffffffff. For example, the following figure shows the memory address, stored values, and name of variables in the zmac?? machines at SFC.



When two variables are used in a program, such as `pointer2.c`, the memory and variables are shown.

```

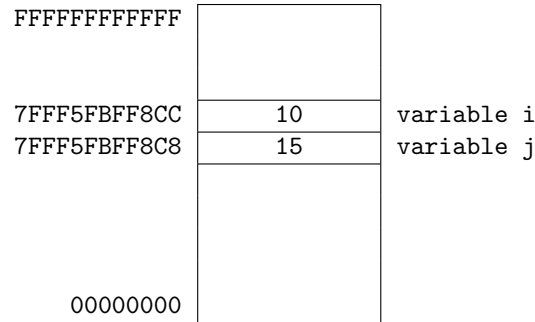
/* pointer2.c */
#include <stdio.h>
#include <stdlib.h>
int main(){
    unsigned int i, j;
    i = 10, j = 15;
    printf("i = %d, &i = %p\n",i,&i);
    printf("j = %d, &j = %p\n",j,&j);
    exit(0);
}

```

This gets the following result.

```
zmac999% ./pointer2
i = 10, &i = 0x7fff5fbff8cc
j = 15, &j = 0x7fff5fbff8c8
```

The following figure shows the relationship between memory locations and variables.



The address of the storage space for the variable `i` (`7FFF5FBFF8CC`) and the address of the storage space for the variable `j` differ by 4. This difference corresponds the number of bytes which are required to store an integer type (`int`) variable. This example shows that memories are allocated in sequence when a system assigns storage space for variables.

### 3.3.2 Character arrays and pointers

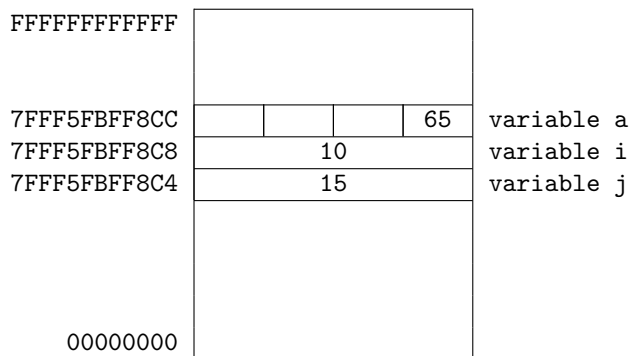
Next, character variables (written `char`), and arrays of characters (written `char []`) are discussed.

```
/* pointer3.c */
#include <stdio.h>
#include <stdlib.h>
int main(){
    unsigned int i;
    char a;
    unsigned int j;
    i = 10; j = 15; a = 'A';
    printf("i = %d, &i = %p\n",i,&i);
    printf("j = %d, &j = %p\n",j,&j);
    printf("a = %d, &a = %p\n",a,&a);
    exit(0);
}
```

This program gets the following result.

```
zmac000% ./pointer3
i = 10, &i = 0x7fff5fbff8c8
j = 15, &j = 0x7fff5fbff8c4
a = 65, &a = 0x7fff5fbff8cf
```

The following figure shows the relationship between memory locations and variables. The value of the character variable `a` is 65 which corresponds to ASCII code for the uppercase letter `A`.



After the address of the storage area for variable `i` (`7FFF5FBFF8C8`), the storage area of character variable `a` is allocated, and its area is 1 byte in length. Then, the 3 bytes of address from `7FFF5FBFF8C` to `7FFF5FBFF8E` are not used.

Next, a character array example is shown.

In the C language, a character string is treated as an array of `char` type, `char[]`. A character string is a sequence of characters. The end of a character string is demarked by `0` in an array of `char` type. This `0` is called the **null character**, or `NULL` character.

For example, a string "HELLO" is stored in a character array defined as `char a[8]` as follows.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
H	E	L	L	0	\0	?	?

`\0` indicates the value is 0. Be aware, it does not indicate a character '0'.

As in our previous examples, the values of pointers are examined as follows.

```

/* pointer4.c */
#include <stdio.h>
#include <stdlib.h>
int main(){
    unsigned int i;
    char a[8];
    unsigned int j;
    i = 10; j = 15;
    a[0] = 'H'; a[1] = 'E'; a[2] = 'L';
    a[3] = 'L'; a[4] = 'O'; a[5] = '\0';
    printf("i = %d, &i = %p\n",i,&i);
    printf("j = %d, &j = %p\n",j,&j);
    printf("&a = %p\n",&a);
    printf("a[2] = %d, &a[2] = %p\n",
           a[2],&a[2]);
    printf("a[] = %s\n",a);
    exit(0);
}

```

This program gets the following result.

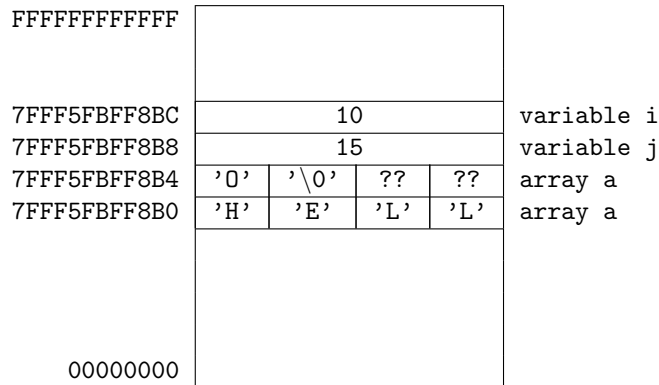
```

zmac000% ./pointer4
% ./pointer4
i = 10, &i = 0x7fff5fbff8bc
j = 15, &j = 0x7fff5fbff8b8
&a = 0x7fff5fbff8c0
a[2] = 76, &a[2] = 0x7fff5fbff8c2
a[] = HELLO

```

In this example, a new format and specifier for `printf`, `printf("a[] = %s\n",a);` are used. `%s` treats the corresponding variable as a pointer to an array of character variables, and displays the string indicated from that pointer to the termination character (`'\0'`).

The result and the relationship between memory location and variables are displayed in the next figure. The contents of the character string are displayed as characters rather than corresponding ASCII code value for readability. In actual memory, 72 is stored for `'H'`, 76 is stored for `'L'`, and so on.



As shown in this example, each variable shows the location (address) in memory. Depending on the type of a variable, the size of area allocated in the memory differs. For example, 4 bytes are allocated for an integer (`long int`), and 1 byte for a character (`char`).

### 3.3.3 Pointer variables

As we have seen, the location of the storage area allocated for a variable can be displayed using `&`. Also, a variable that stores the location of another variable can be defined. This is called a *pointer variable*. It is defined by prepending `*` to the `variable name`.

```
type *variable name;
```

In the next example, `p` holds the address of variable `x`. You will see that `*p` has the same value as `x`.

You can assign an arbitrary name to a variable. However, for readability and understandability, for pointer variables, names beginning with `p`, are usually assigned. If you know you are creating a variable that always holds a pointer to `x`, you can name it `px`, for clarity. When a single-character variable name is desired (as in small loops), `p`, `q`, or `r` is often used.

```
int x;
int *p;
p = &x;
```

An integer variable is a box that holds an integer value. Similarly, a pointer variable is a box that holds a pointer. However, in this case, the box holds a piece of paper that says “An integer value is stored in the box ...”.

By executing the following program, confirm the contents of `x` and `p`.

```

/* pointer-change.c */
#include <stdio.h>
#include <stdlib.h>
int main(){
    int x; int *p;
    x = 7;
    p = &x;
    printf("x = %d, &x = %p\n",x,&x);
    printf("p = %p, *p = %d\n",p,*p);
    printf("&p = %p\n",&p);
    exit(0);
}

```

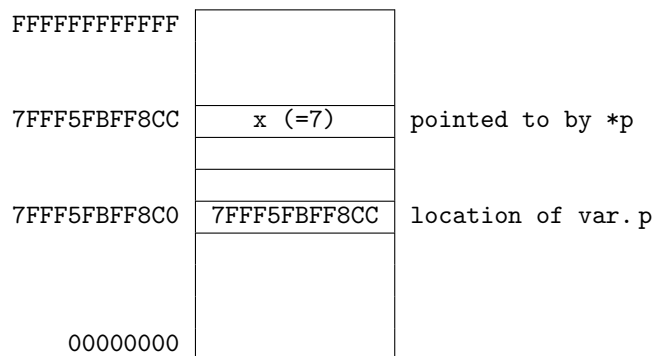
The following result is obtained.

```

% ./pointer-change
x = 7, &x = 0x7fff5fbff8cc
p = 0x7fff5fbff8cc, *p = 7
&p = 0x7fff5fbff8c0

```

This result is shown in the following figure.



The location of variable `p` is `7FFF5FBFF8C0`, and the value stored in that location is `7FFF5FBFF8CC`. A pointer to an integer `*p` indicates that the value in the memory address is `7FFF5FBFF8CC`, and its type is `int`.

The contents of data that pointers refer to can be modified. Data pointed by pointers can be referred to as

`*pointer`

For example, in the following program, a location `*p`, referred to by a pointer variable `p`, contains an integer variable `x`. That value is assigned to `y`, then the

integer variable `x`, referred by the pointer `p`, is given the value 3. Consequently, the initial value of `x`, 7, is assigned to `y`, and the value of `x` becomes 3.

```

/* pointer-change2.c */
#include <stdio.h>
#include <stdlib.h>
int main(){
    int x,y;  int *p;

    p = &x;
    x = 7;
    y = *p;
    *p = 3;
    printf("x = %d, y = %d\n",x,y);
    printf("*p = %d\n",*p);
    printf("x = %d, &x = %p\n",x,&x);
    printf("p = %p, &p = %p\n",p,&p);
    printf("&x = %p, &y = %p\n",&x,&y);
    exit(0);
}

```

The result of the program's execution is shown. The value of pointer variable `p` and the address of the integer variable `x` have the same value, `7FFF5FBFF8CC`.

```

% ./pointer-change2
x = 3, y = 7
*p = 3
x = 3, &x = 0x7fff5fbff8cc
p = 0x7fff5fbff8cc, &p = 0x7fff5fbff8c0
&x = 0x7fff5fbff8cc, &y = 0x7fff5fbff8c8

```

The memory allocation is shown as follows.

FFFFFFFF		
7FFF5FBFF8CC	x (=3)	location referred to by *p
7FFF5FBFF8C8	y (=7)	location of var. y
7FFF5FBFF8C0	7FFF5FBFF8CC	location of var. p
00000000		

In this example, even before the assignment of 7 to `x`, the pointer variable `p` can be given the address of `x`. Even before you know the value of `x`, you know the address where `x` will be stored.

### 3.3.4 Manipulation of pointer variables

You can add and subtract an integer value to a pointer variable. Adding 1 to a pointer causes the pointer to refer to the next data element in memory. The name of an array is considered to be a pointer constant that refers to the beginning of the array.

```

/* pointer-op.c */
#include <stdio.h>
#include <stdlib.h>
#define ARRAYSIZE
int main(){
    int a[ARRAYSIZE]; int *p;
    int i;

    for(i=0;i<ARRAYSIZE;i++){
        a[i] = i;
    }
    printf("&a[0]=%p,&i=%p,&p=%p\n",
           &a[0],&i,&p);
    p = a;
    printf("p=%p,*p=%d\n",p,*p);
    p++;
    printf("p=%p,*p=%d\n",p,*p);
    exit(0);
}

```

At the time of execution, the value of the pointer variable `p` is `7FFF5FBFF890`. That value corresponds to the address of the integer variable `a[0]`. The content of `a[0]` is 0, as set by our initialization code in the `for` loop.

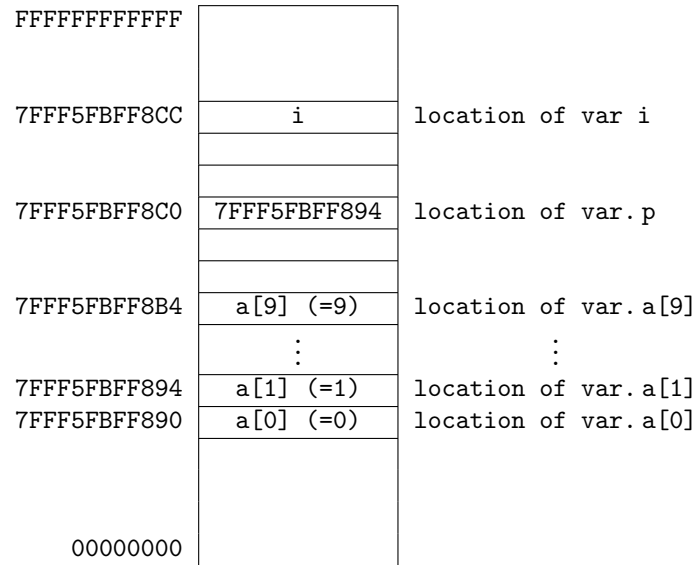
```

% ./pointer-op
&a[0]=0x7fff5fbff890,&i=0x7fff5fbff8cc,
  &p=0x7fff5fbff8c0
p=0x7fff5fbff890,*p=0
p=0x7fff5fbff894,*p=1

```

The result is shown in the next figure.





An important part is the calculation of `p++`. The `++` operator, the increment operator, adds 1 to a value. However, in this example, the value is increased by 4, from `7FFF5FBFF890` to `7FFF5FBFF894`. When an increment operator is applied to a pointer, the value is increased by the size of the variable that the pointer indicates. Using this mechanism, a sequence of variables of the same type are handled correctly.

The next example is our first real encounter with *character strings*. In C, a string is a sequence of `char` variables, and the end of a string is the terminal character (`'\0'`). Therefore, a pointer to a character variable, `char *`, has the same type as a pointer to the first character of a string.

The following table shows this relationship. A fixed-length array for a string, `char a[6]`, and a single character variable, `c`, are compared.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
A	B	C	\0	?	?

c
Z

↑ `*p` pointer to `a[0],c`

The string "ABC" is assigned to array `a[6]`. The character variable `c` has the character Z.

The next program shows the allocation of variables in the memory.

```

/* pointer-op2.c */
#include <stdio.h>
#include <stdlib.h>
int main(){
    char a[6]; char *p;
    char c;

    a[0] = 'A'; a[1] = 'B'; a[2] = 'C';
    a[3] = '\0';
    printf("&a[0]=%p,&c=%p,&p=%p\n",&a[0],&c,&p);
    p = a;
    printf("p=%p,*p=%d\n",p,*p);
    p++;
    printf("p=%p,*p=%d\n",p,*p);
    exit(0);
}

```

The result is shown below. 65 is the ASCII code value of the character A, and 66 is the ASCII code value of the character B.

```

% ./pointer-op2
&a[0]=0x7fff5fbff8c0,&c=0x7fff5fbff8cf,
  &p=0x7fff5fbff8b8
p=0x7fff5fbff8c0,*p=65
p=0x7fff5fbff8c1,*p=66

```

The contents of memory are shown in the next picture.

FFFFFFFF					
7FFF5FBFF8CC				c	variable c
7FFF5FBFF8C4					array a
7FFF5FBFF8C0	'A'	'B'	'C'	'\0'	array a
7FFF5FBFF8B8	7FFF5FBFF8C1				variable p
00000000					

Because the size of the character array `char a[6]` is 6, it occupies 6 bytes from address `7FFF5FBFF8C0` to `7FFF5FBFF8C5` in the memory. At the calculation of `p++`, the value of the pointer is increased by 1 to point to the next character in the array. Initially, the value is `7FFF5FBFF8C0`, which points to 'A'. After an increment, it becomes `7FFF5FBFF8C1`, which points to 'B'.

## 3.4 Pointers and Functions

Pointers have a strong relationship with the passing of arguments to functions.

### 3.4.1 Function calls and arguments

The next program intends to calculate two times `a`. It passes two arguments, `a` and `b` to the function `multi2`, and assigning the result to `b`

```
/* function-p.c */
#include <stdio.h>
#include <stdlib.h>

void multi2(int a, int b){
    b = a * 2;
    printf("In multi2: a=%d,b=%d\n",a,b);
    return;
}

int main(){
    int x=0,y=0;

    x = 10; y = 0;
    multi2(x,y);
    printf("In main: x=%d,y=%d\n",x,y);
    exit(0);
}
```

This programs seems to be good. However, we get the following result.

```
% ./function-p
In multi2: a=10,b=20
In main: x=10,y=0
```

In the function `multi2`, the value is multiplied by 2 correctly. However, when the control returns to the `main`, the value of `y` is unchanged, and still 0. What is wrong?

When you call the function `multi2(x,y)`, the **values** of `x` and `y`, at the time of call, are passed to the function. In the example, 10 (`x`) and 0 (`y`) are passed.

In the function `multi2`, `x` and `y` are the arguments. Their values are assigned to the variables `a` and `b` which can be used only inside the function. Therefore, when the function has completed, and the control returns to the `main`, the values of `x` and `y` are unchanged.

As in other previous examples, to discover the storage location of variables, let's examine the addresses of variables using constructs such as `&x`. The following result will be obtained.

FFFFFFFFFFFF		
7FFF5BFF8CC	p (=10)	location of var. p
7FFF5BFF8C8	q (=0)	location of var. q
	⋮	
7FFF5BFF8AC	a (=10)	location of var. a
7FFF5BFF8A8	b (=20)	location of var. b
00000000		

To run this program correctly, you have to pass the *locations* of variables `x` and `y` to the function `multi2` instead of the *value* of `x` (10). Therefore, on the caller's side, the addresses of variables are passed as following.

```
multi2(&x,&y);
```

On the callee side, it receives the addresses of their storage locations, then it processes assuming those locations store integer values.

```

/* function-p2.c */
#include <stdio.h>
#include <stdlib.h>

void multi2(int *a, int *b){
    *b = *a * 2;
    printf("In multi2: a=%d,b=%d\n",*a,*b);
    return;
}

int main(){
    int x,y;

    x = 10; y = 0;
    multi2(&x,&y);
    printf("In main: x=%d,y=%d\n",x,y);
    exit(0);
}

```

In the function `multi2`, both the first argument and the second argument are pointers to integers.

```
void multi2(int *a, int *b)
```

The next figure shows the storage locations of variables and their values in this example.

FFFFFFFFFFFF		
7FFF5FBFF8CC	x (=10)	location of x
7FFF5FBFF8C8	y (=0→20)	location of y
	⋮	
7FFF5FBFF8A8	a(=7FFF5FBFF8CC)	location of a
7FFF5FBFF8A0	b(=7FFF5FBFF8C8)	location of b
00000000		

The modified program gets the following result.

```
% ./function-p2
In multi2: a=10,b=20
In main: x=10,y=20
```

### 3.4.2 How to pass arrays

When you pass arrays to functions, the name of an array is used as an argument. The called function receives those arguments as arrays.

```
/* array-p.c */
#include <stdio.h>
#include <stdlib.h>
#define MAXNUM 10
void array(int b[]){
    int i;
    for(i=0;i<MAXNUM;i++){
        b[i] *= 2;
    }
    return;
}

int main(){
    int i, a[MAXNUM];

    for(i=0;i<MAXNUM;i++){
        a[i] = i;
    }
    array(a);
    for(i=0;i<MAXNUM;i++){
        printf("a(%d)=%d\n",i,a[i]);
    }
    exit(0);
}
```

`#define MAXNUM 10` is a MACRO. In this example, the string `10` is substituted for the string `MAXNUM` at compilation time. By using this MACRO, when you want to modify the size of the array from `10` to `20`, you have to change only one line, this MACRO definition.

This example initializes the integer array `a[]`, then multiplies those initialized values by `2` in the function `array`.

```
% ./array-p  
a(0)=0  
a(1)=2  
a(2)=4  
a(3)=6  
a(4)=8  
a(5)=10  
a(6)=12  
a(7)=14  
a(8)=16  
a(9)=18  
%
```

The next example uses a two-dimensional array. Unlike the previous one-dimensional array, you have to specify the size of the array in the prototype for the called function, such as `void array(int b[MAXROW] [MAXCOL])`. If you failed to specify this size, a compilation error occurs, because the called function has no way to know how to handle the array. Because of its very direct relationship to how system stores data, multi-dimensional arrays are not supported as cleanly as in some other languages.

```
/* array-p2.c */
#include <stdio.h>
#include <stdlib.h>
#define MAXROW 5
#define MAXCOL 4
void array(int b[MAXROW] [MAXCOL]){
    int i,j;
    for(i=0;i<MAXROW;i++){
        for(j=0;j<MAXCOL;j++){
            b[i][j] *= 2;
        }
    }
    return;
}

int main(){
    int i, j;
    int a[MAXROW] [MAXCOL];

    for(i=0;i<MAXROW;i++){
        for(j=0;j<MAXCOL;j++){
            a[i][j] = i*j;
        }
    }
    array(a);
    for(i=0;i<MAXROW;i++){
        for(j=0;j<MAXCOL;j++){
            printf("a(%d)(%d)=%d\n",i,j,a[i][j]);
        }
    }
    exit(0);
}
```



This example gets the following result.

```
% ./array-p2
a(0)(0)=0
a(0)(1)=0
a(0)(2)=0
a(0)(3)=0
a(1)(0)=0
a(1)(1)=2
a(1)(2)=4
a(1)(3)=6
a(2)(0)=0
a(2)(1)=4
a(2)(2)=8
a(2)(3)=12
a(3)(0)=0
a(3)(1)=6
a(3)(2)=12
a(3)(3)=18
a(4)(0)=0
a(4)(1)=8
a(4)(2)=16
a(4)(3)=24
%
```

## 3.5 Structures

When you create a complex program, you may want to handle multiple variables as a chunk. For example, to handle a complex number or a fractional number, it is better that you can name and handle variables which hold a pair of integers or a pair of real numbers, instead of two separate integer variables or real numbers. In C, you can define a new data type by combining existing basic data type such as `int`. This data type is called a **Structure**.

The next example shows how to define a structure.

```
struct {
    type member name;
    ....
} variable name of the structure;
```

The next example shows another form of the definition with the name of the structure. The name can be used in other parts of the program.

```
struct type name of structure {
    type member name;
    ....
} variable name of structure;
```

In this definition, you may omit the variable name, and only define the structure.

```
struct type name of structure {
    type member name;
    ....
} ;
```

In the following example, to make a program which handles fractional number computation, a structure to specify the fractional number is defined. `struct frac` is the name of structure for fractional numbers. Then, `bo` will hold the denominator (bunbo in Japanese), `shi` will hold the numerator (bunshi in Japanese). Those names are arbitrary, but it is better to use some meaningful name.

```
struct frac{
    int shi;
    int bo;
} f;
```

When you use elements of structures, you have to specify the element by combining the structure's name and a member name.

```
variable name . member name
```

The following example assigns 3/5 to a fractional number structure.

```
f.shi = 3;
f.bo = 5;
```

The next example calculates the sum of two fractional numbers. `fgets()` and `sscanf()` are functions for one line input and conversion to integer type variables. Details are discussed in the chapter for input and output. For the moment, you only have to understand that two numbers separated by a slash are assigned to a numerator variable and a denominator variable.

```
/* structure-frac.c */
#include <stdio.h>
#include <stdlib.h>
#define MAXLINE 256

struct frac{
    int shi;
    int bo;
};

int main(){
    int ret;
    char input[MAXLINE];
    struct frac f,g,h;

    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%d/%d",&f.shi, &f.bo);
    if(ret != 2) {exit(1);}
    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%d/%d",&g.shi, &g.bo);
    if(ret != 2) {exit(1);}

    h.shi = f.shi * g.bo
          + g.shi * f.bo;
    h.bo = f.bo * g.bo;
    printf("%d/%d + %d/%d = %d/%d\n",
           f.shi, f.bo,
           g.shi, g.bo,
           h.shi, h.bo);
    exit(0);
}
```

The result of the program is shown.

```
% ./structure-frac
1/3 (1st fgets input)
2/5 (2nd fgets input)
1/3 + 2/5 = 11/15 (output)
```

If you try writing this program without structures, you will understand the improvement of readability we gain by using structures (exercise 3.3).

## 3.6 Tools

### 3.6.1 gcc: phases of compilation

C compilers, including `gcc`, go through different *phases* from the time they start reading your program:

- **preprocessor:** the preprocessor processes all of the commands in your source code that start with `#`. We have already seen `#include` and `#define`. The output of this (which you generally never see) is a longer, purer C program, fed directly into the next phase of compilation.
- **compiling:** turns the C program into chunks of executable code for this processor.
- **linking:** takes the various chunks of executable code, finds the necessary libraries, and puts the whole together into a program that will run on this operating system.

### 3.6.2 git

Git is a system for tracking and sharing changes in your software (or almost any other file). For the moment, do this:

```
% git config --global user.name "Your Name"
% git config --global user.email your.email@your.domain
% git init
% git add example.c Makefile
% git commit
(use the editor, add a comment)
```

## 3.7 Exercises

- 3.1** Write the following program `ex2-3.c`. For the array `a[MAXITEM]`, assign 1 to `MAXITEM` from the head of the array (the first element of the array is `a[0]`). Then, calculate `a[i]*a[i]` for all `a[i]` of the array, assign those values to the array `b[i]`, then display all `b[i]` with `printf`.

`MAXITEM` is a macro, defined at the beginning of the program as,

```
#define MAXITEM 10
```

With this definition, of course the numbers 1,4,9,...,100 should be displayed as `b[i]`.

- 3.2** Referring to the program `array-p2.c` which passes a two-dimensional array to a function, write a program which passes a three dimensional array to a function. For example, the array may defined at the beginning of the program as:

```
int a[MAXROW] [MAXCOL] [MAXZ];
```

- 3.3** A fractional number addition is shown in the example for structures (`structure-frac.c`). Write this program without structures.
- 3.4** \* In the fractional number addition program using structures,  $2/5 + 2/5$  becomes  $20/25$ , when of course we would like to get  $4/5$ . Write a program which displays a result in an irreducible fraction. (How can you reduce a fraction?)
- 3.5** In the fractional number example, addition is shown. Write the program `ex2-7.c` which computes addition, subtraction, multiplication, and division, then displays all four results.
- 3.6** Write a program that calculates the *dot product* (*inner product*) of two vectors of floating point numbers and displays it. These vectors should be the same size, of course, and that size should be defined at compile time using a `#define`. The inner product of two vectors is a scalar,

$$\vec{a} \cdot \vec{b} = \sum_{i=0}^{n-1} a_i \cdot b_i \quad (3.1)$$

for two vectors of length  $n$ . (Note the first index is 0 here, in keeping with C, though many math textbooks would define it starting from 1.)

- 3.7** Write a program to print the addresses local variable in `main()`, a second function called from `main()`, and a third function called from inside the second function. How do they change across calls?
- 3.8** In this problem, we will see the first vague hint of code related to networks, use a system library function, pointers, and iteration, all in one problem.

The library function `if_nameindex()` returns a pointer to `struct if_nameindex`. Use it to create a program that prints out a list of all of the network interfaces in your system.

The `if_nameindex` structure:

```
struct if_nameindex{
    u_int if_index;
    char * if_name;
};
```

The function returns an array of these structs. For more information see the manual page (`man if_nameindex`).

Include files to use:

```
#include<sys/types.h>
#include<net/if.h>
```

Your program should print out something like this:

```
./iflist
1 lo
2 eth0
3 eth1
4 sit0
```

Be sure to use the function `if_freenameindex()` after you have finished printing out the list.

## Chapter 4

# String Processing and Pointers

In which we learn how to handle strings of text using pointers and library functions.
---

### 4.1 Concepts

The C approach to handling strings represented an advance over earlier high-level languages such as FORTRAN. Strings can be variable length. (This seems so natural now that it is hard to imagine how alien it was in the early 1970s.) A string is *null terminated*; that is, a special character (the null character) marks the end of the string. This allows algorithms to loop until finding the terminator, rather than for some fixed count.

The techniques presented here are important for handling filenames, processing strings input from the keyboard or other devices, or processing data from text files. (We will see more on file I/O in Chapter 6.) As such, they are an important component of many large programs.

Many standalone text processing utilities are written in C, although many programmers today find it more productive to write such tools in Python, Ruby or Perl.

This chapter includes a number of examples, but the UNIX man pages for the library functions provide additional detail. Please refer to them for return values and more information on constraints in their use.

## 4.2 Strings as arrays

A string is an array of characters, and can be treated just like an array of any other data type, except that as we noted strings are usually variable length. The next program creates a string by putting a character into every element of an array.

```
/* hello.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
    char a[8];

    a[0] = 'H'; a[1] = 'E';
    a[2] = 'L'; a[3] = 'L';
    a[4] = '0'; a[5] = '\0';
    printf("a[] = %s\n",a);
    exit(0);
}
```

Note the `'\0'` put into `a[5]`. This is our *null terminator* character. The backslash tells the compiler to put the following number into the character, rather than the ASCII code for the character `'0'` (which is `0x30`, not `0`).

In the C language, the data extending from any intermediate point in the string to the terminal character `'\0'` also can be interpreted as a string. So, advancing a pointer to a string one by one, you get strings that gradually become shorter.

```
/* hellop.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
    char a[8];
    char *p;

    a[0] = 'H'; a[1] = 'E';
    a[2] = 'L'; a[3] = 'L';
    a[4] = '0'; a[5] = '\0';
    printf("a[] = %s\n",a);
    for(p = a; *p != '\0'; p++){
        printf("%s\n",p);
    }
    exit(0);
}
```



`for(p = a; *p != '\0'; p++)` is frequently used syntax. It advances a pointer across a string step by step, then exits the loop when it encounters the terminal character `'\0'`. An execution example is shown.

```
% ./hellop
a[] = HELLO
HELLO
ELLO
LLO
LO
O
%
```

Now, let's create a function which determines the location of a specified character in a string. This function is called with two arguments, the pointer to the string and a target character. To get a string we want to search, we will use the library function `fgets()`.

```
/* search-c.c */
#include <stdio.h>
#include <stdlib.h>
#define MAXLINE 256

int searchc(char *s, char c){
    char *p;
    int i;

    i = 1;
    for(p = s; *p != '\0'; p++,i++){
        if(*p == c){
            break;
        }
    }
    if(*p == '\0'){
        return -1;
    }else{
        return i;
    }
}

int main(){
    char input[MAXLINE];
    int c, loc;

    printf("Enter a string\n");
    fgets(input,MAXLINE,stdin);
    printf("Enter a character\n");
    c = getc(stdin);
    loc = searchc(input,c);
    if(loc < 0){
        printf("%c is not found.\n",c);
    }else{
        printf("%c is found at position %d in the string.\n",
            c,loc);
    }
    exit(0);
}
```

The result is shown.

```
% ./search-c
Enter a string
ABCDEF
Enter a character
C
C is found at position 3 in the string.
%
```

The `fgets()` function is used for string input:

```
char *fgets(char *s, int size, FILE *stream);
```

The first argument is a pointer to a character string. It designates the storage location where the input string can be found. The second argument is an integer value which indicates the maximum size of the storage. The third argument designates which input channel is to be used. As in the example program, when `stdin` is specified, the program reads a string from the standard input, which is usually the keyboard. The input/output system in C will be taken up in more detail in later chapters.

### 4.2.1 Copying and concatenating strings

In C, strings are treated as arrays of character data. To understand C's string processing, the important concept is that the name of an array of characters corresponds to a pointer to a character variable.

As an example, the following program manipulates two strings input by `fgets`.

```

/* string-example.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#define MAXLINE 256

int main(){
    char str1[MAXLINE], str2[MAXLINE];
    char str3[MAXLINE*2];
    char *s;

    /* First, make sure strings are empty */
    str1[0] = '\0';
    str2[0] = '\0';
    str3[0] = '\0';

    printf("Enter the first string:\n");
    fgets(str1,MAXLINE,stdin);
    printf("Enter the second string:\n");
    fgets(str2,MAXLINE,stdin);
    /* Display the input strings */
    printf("str1 is %s\n",str1);
    printf("str2 is %s\n",str2);
    /* replace the \n character with '\0' */
    s = strchr(str1,'\n');
    if(s!=NULL){
        *s = '\0';
    }
    s = strchr(str2,'\n');
    if(s!=NULL){
        *s = '\0';
    }
    /* String copy (see below) */
    strncpy(str3,str1,MAXLINE);
    printf("%s is copied to str3\n",str3);
    /* String concatenation */
    strncat(str3,str2,MAXLINE);
    printf("Concatenating str2 to str3 becomes %s.\n",
        str3);
    exit(0);
}

```

This program reads two strings with the function `fgets`, assigns these strings to two variables `str1` and `str2`, then displays them. The last part of the input strings includes the line feed character, `\n`. To eliminate this character, the program uses the `strchr()` function to search for the character, and replaces it with the termination character `'\0'`.

If the `strchr()` call didn't find the null terminator, it is likely that the line

the user typed filled or overflowed the buffer, that we have reached the end of whatever file we are reading, or there was an error. (We should have checked the return value of the `fgets()`.) To learn more about the behavior of this, try setting `MAXLINE` to some small value like 4, and typing long strings at the prompt.

Next, the program copies string `str1` to the string variable `str3` by the function `strncpy()`, and displays it. Then, the program uses `strncat()` to concatenate `str2` to `str3`, and display the modified `str3`. Note that we chose to use `strncpy()` instead of `strcpy()`, and `strncat()` instead of `strcat()`. In this case, just above, we have already guaranteed that the strings are each shorter than `MAXLINE`, including the null terminator, so we could have safely used the other functions. However, as a matter of good habit, you should get accustomed to using the versions that include limits on how much data they will copy. If you do not, you may leave your programs open to *buffer overflow* attacks, which open far too many security holes in the world today – don't let yours be one of them! (Buffer overflows are related to the array bounds checking we discussed back on page 40.)

Example execution results are as follows.

```
% ./string-example
Enter the first string:
abc          <--- input string
Enter the second string:
def          <--- input string
str1 is abc  <--- displayed string
str2 is def  <--- displayed string
abc is copied to str3
Concatenating str2 to str3 becomes abcdef
%
```

The `strchr()` function determines whether the specified character exists in the strings or not. The usage is

```
#include <string.h>
char *strchr(const char *s, int c);
```

For copying a string, `strncpy` is used. As shown below, the first argument designates the location of the destination, the second argument designates the location of the source. This function returns the argument `dst`.

```
#include <string.h>
char *strncpy(char *restrict dst,
               const char *restrict src, size_t n);
```

For concatenating strings, the function `strncat()` is used. As shown below, the first argument designates the source, the second argument designates the string to be added to `s1`. This function returns `s1`.

```
#include <string.h>
char *strncat(char *restrict s1,
const char *restrict s2, size_t n);
```

### 4.2.2 Comparison of strings

To compare two strings, we can use the functions `strcmp()` and `strncmp()`.

As shown below, the first argument and the second argument designate the two strings to be compared. First, the function compares the first character of the first string and the first character of the second string. If those two characters are the same, the function compares the second characters, and so on. If the two strings are the same, the function returns 0. When the function encounters different characters at some point, it will return an integer value that is negative or positive. That value shows the magnitude correlation between the two different characters at that point. If  $s1 > s2$ , the function returns a positive value; if  $s1 < s2$ , it returns a negative value.

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

The function `strncmp()` is almost same as `strcmp()`, but it only compares strings up to the  $n$ th character. This number is specified by the third argument. *In general, you should use `strncmp()`.*

```
#include <string.h>
int strncmp(const char *s1, const char *s2,
size_t n);
```

The next program accepts two strings, and displays the comparison result.

```
/* string-cmp.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXLINE 256

int main(){
    char str1[MAXLINE], str2[MAXLINE];
    char *s;
    int cmp;

    printf("Enter the first string\n");
    fgets(str1,MAXLINE,stdin);
    printf("Enter the second string\n");
    fgets(str2,MAXLINE,stdin);
    /* substituting the last \n with
       the NULL character */
    s = strchr(str1,'\n');
    if (s!=NULL){
        *s = '\0';
    }
    s = strchr(str2,'\n');
    if (s!=NULL){
        *s = '\0';
    }
    /* display strings */
    printf("str1 is %s\n",str1);
    printf("str2 is %s\n",str2);
    /* compare strings */
    cmp = strcmp(str1,str2);
    if (cmp == 0) {
        printf("The two strings are the same\n");
    }else if (cmp>0) {
        printf("%s is larger than %s\n",
            str1,str2);
    }else{
        printf("%s is larger than %s\n",
            str2,str1);
    }
    exit(0);
}
```

The result is shown below.

```
% ./string-cmp
Enter the first string
abc
Enter the second string
def
str1 is abc
str2 is def
def is larger than abc
```

### 4.2.3 Classifying characters

Sometimes, you need to classify a character, determining whether it is upper case, lower case, white space, and so on. The standard library includes functions for this purpose and for converting cases. You have to include the header file `ctype.h` to use these functions.

```
#include <ctype.h>
int toupper(int c);
int tolower(int c);
int isalpha(int c);
int isupper(int c);
int islower(int c);
int isdigit(int c);
int isxdigit(int c);
int isalnum(int c);
int isspace(int c);
int ispunct(int c);
int isprint(int c);
int isgraph(int c);
int iscntrl(int c);
int isascii(int c);
```

These functions check whether `c`, which must have the value of an unsigned char or EOF, falls into a certain character class according to the current locale.

`toupper()` and `tolower()` convert the character to upper case and lower case respectively. Non-alphabetic characters are unchanged.

`isalpha()` and other functions check whether the character falls into a certain character class. They return 1 when the character falls into the class, and return 0 otherwise.

The following program checks input characters one by one, and displays the result of classification. Enter several types of characters and examine the result.



```
/* is-example.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAXLINE 256

int main(){
    char str1[MAXLINE];
    char *s;

    printf("Enter a string\n");
    fgets(str1,MAXLINE,stdin);
    /* substituting the NULL character
       for the last \n */
    s = strchr(str1,'\n');
    if(s!=NULL){
        *s = '\0';
    }
    /* Displaying the string */
    printf("The input string is %s\n",str1);
    /* Check the string */
    for(s = str1; *s != '\0'; s++){
        if(isdigit(*s)){
            printf("%c is a number.\n",*s);
            continue;
        }
        if(isalpha(*s)){
            printf("%c is an alphabetic character",*s);
            if(isxdigit(*s)){
                printf(" and is used in hexadecimal notation");
            }
            printf(".\n");
            continue;
        }
        if(isspace(*s)){
            printf("%c is a white space character.\n",*s);
            continue;
        }
        printf("%c is some other type of character.\n",*s);
    }
    exit(0);
}
```

The result is shown. Pay attention to using the `continue` statement. That avoids executing extra `if` statements.

```

% ./is-example
Enter a string
Because we can 14.
The input string is Because we can 14.
B is an alphabetic character and is used in hexadecimal notation.
e is an alphabetic character and is used in hexadecimal notation.
c is an alphabetic character and is used in hexadecimal notation.
a is an alphabetic character and is used in hexadecimal notation.
u is an alphabetic character.
s is an alphabetic character.
e is an alphabetic character and is used in hexadecimal notation.
  is a white space character.
w is an alphabetic character.
e is an alphabetic character and is used in hexadecimal notation.
  is a white space character.
c is an alphabetic character and is used in hexadecimal notation.
a is an alphabetic character and is used in hexadecimal notation.
n is an alphabetic character.
  is a white space character.
1 is a number.
4 is a number.
. is some other type of character.

```

#### 4.2.4 Strings and numerical data

Sometimes, you want to convert a string to a numeric value (either integer or floating point), or vice versa. In C, the functions `strtol()` and `sprintf()` can be used for this purpose.

The `strtol()` function has the following form. The function converts the initial part of the string in `str` to a long integer value according to the given base. The base must be between 2 and 36. If `endptr` is not `NULL`, `strtol()` stores the address of the first invalid character in `*endptr`. This can be used to efficiently skip over white space, commas, etc.

```

#include <stdlib.h>
long strtol(const char *str, char **endptr,
           int base);

```

The following example assigns an input number to an `int` and a `long` type integer, then checks the result. This points out the problem of assigning a `long` integer to an `int`.

```
/* str-tol.c */
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
#include <limits.h>

#define MAXLINE 256

int main(){
    int i;
    long k;
    char input [MAXLINE];

    fgets(input,MAXLINE,stdin);
    i = strtol(input,NULL,10);
    k = strtol(input,NULL,10);
    printf("i=%d\n",i);
    printf("k=%ld\n",k);
    if (k == LONG_MAX)
        printf("pegged k at LONG_MAX\n");
    exit(0);
}
```

The result is shown below.



```
/* str-sscanf-bad.c */
#include <stdio.h>
#include <stdlib.h>

#define MAXLINE 256

int main(){
    int i,ret;
    char input[MAXLINE];

    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%d",&i);
    printf("i=%d\n",i);
    exit(0);
}
```

The result is shown below.

```
% ./str-sscanf-bad
10
i=10
% ./str-sscanf-bad
a
i=32767
%
```

As you can see, it executes correctly when numbers are entered. However, when non-number strings are input, it stores an incorrect value in the variable. To detect this error, you have to check the return value of `sscanf()`. The corrected program is shown below.

```
/* str-sscanf.c */
#include <stdio.h>
#include <stdlib.h>

#define MAXLINE 256

int main(){
    int i,ret;
    char input[MAXLINE];

    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%d",&i);
    if(ret <= 0){
        printf("Invalid input\n");
    }else{
        printf("i=%d\n",i);
    }
    exit(0);
}
```

The value of `ret` has the number of items that are correctly read. In this example, if `i` is correctly read, `ret` should be 1. The execution result is shown.

```
% ./str-sscanf
10
i=10
% ./str-sscanf
a
Invalid input
%
```

### 4.3 Dealing with non-ASCII characters

*This is important:* You are living in Japan, where, in case you haven't noticed, the dominant language is not English or American!

In the early days of computing, as computers spread throughout the world, countries working with written forms of language other than the basic 26-letter alphabet urgently needed some ability to deal with the local language, and since resources (especially memory) were scarce, computers weren't networked, and international standards bodies for information technologies weaker, many local solutions developed.

Unfortunately, because the many solutions used overlapping numbers for different characters in different languages, as a programmer *it is not possible to*

*always, perfectly distinguish the character, or even language, you are holding.* All robust solutions require some additional information somewhere that specifies the *character set*, telling you how to interpret a string or character.

The original C language and library assumed that the ASCII character set was in use, which was a fair assumption for the time and place since the 'A' in ASCII stands for "American", and C was developed in the United States by Dennis Ritchie. Nowadays, it is a poor assumption as the World Wide Web is literally worldwide.

Dealing with the Japanese language in a computer can't be done using regular 8-bit characters. 8 bits can represent 256 characters at most, and thousands are needed for a basic Japanese character set and tens of thousands for Chinese. Here, we will *very briefly* talk about dealing with other types of characters.

Ken Thompson, the primary architect of the original version of UNIX, and Rob Pike, another of Bell Labs' operating systems superstars, developed the UTF-8 multibyte, variable-length character encoding in the early 1990s. UTF-8 is recommended by the World Wide Web Consortium (W3C) as a default encoding for XML and HTML documents. W3C usage statistics suggest that, in 2015, over 80% of documents on the web are in UTF-8. UTF-8 uses one to four 8-bit bytes for a single character (called a "code point"), and theoretically supports more than one million code points.

C and the C library support (at least) two ways of dealing with character sets using more than one 8-bit byte. We will talk about *wide characters* and *multibyte characters*. A quick summary and references to functions and other information sources can be found on BSD-derived systems by checking the manual page, `man multibyte`.

UTF-8 allows differing numbers of bytes to be used to store each character. This makes handling characters difficult using the functions we have discussed in this chapter, for example because it is possible that you will accidentally split a character in two when copying or concatenating strings, and because you cannot directly index an array using an integer and be certain which ordinal number of character you are touching. In C, when processing strings in memory, it is therefore often convenient to first convert a string of multibyte characters into fixed-size variables known as wide characters, represented as the type `wchar_t`. The library functions `mbtowc()` and `wcrtomb()` convert single characters back and forth, and the functions `mbstowcs()` and `wcstombs()` convert strings.

As noted, strings in general do not describe the character set in which they are encoded, so we must know that *a priori*. The *environment variable* `LANG` and several others starting with `LC_` specify the language that a process expects, although that alone may not be enough as some languages (including Japanese) have multiple, differing character sets for representing the language. HTML or XML and some other documents often begin with a header in ASCII that defines

the *encoding* for the rest of the document. A single file may actually switch between encodings in different regions.

The hassles of doing more than basic string handling in C, especially dealing with non-ASCII strings and the security weaknesses we have discussed in this chapter, are strong incentives for performing many text processing tasks in other languages. Ruby, developed in Japan, is especially strong at non-ASCII processing, although Python also now handles these tasks well.

The standards for C specify that the language's own reserved words be in the standard English alphabet. Identifiers such as variable names are actually allowed to be in another language, although not all compilers may support this behavior. For maximum compatibility in code and maximum readability for programmers around the world, we recommend that you use only core ASCII characters in identifiers. Comments and strings, of course, may be in the local language, as appropriate.

⇒ *One or two examples should be included here.*

## 4.4 Structures and functions

The next example shows how structures are passed to a function. First, the structure `struct bunsuu` defines the fraction numbers. (*Bunsuu* is Japanese for fraction.) Next, two `bunsuu` arguments are passed to a function, and the return value of the function also has the type `struct bunsuu`.



```

/* struct-p.c */
#include <stdio.h>
#include <stdlib.h>
#define MAXLINE 256

struct bunsuu {
    int shi;
    int bo;
};

struct bunsuu keisan(struct bunsuu p,
    struct bunsuu q){
    struct bunsuu r;

    r.shi = p.shi * q.bo + p.bo * q.shi;
    r.bo = p.bo*q.bo;
    return r;
}

int main(){
    int ret;
    char input [MAXLINE];
    struct bunsuu a,b,c;

    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%d/%d",&a.shi, &a.bo);
    if(ret != 2) {exit(1);}
    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%d/%d",&b.shi, &b.bo);
    if(ret != 2) {exit(1);}

    c = keisan(a,b);
    printf("%d/%d + %d/%d = %d/%d\n",
        a.shi,a.bo,b.shi,b.bo,c.shi,c.bo);
    exit(0);
}

```

The result is shown.

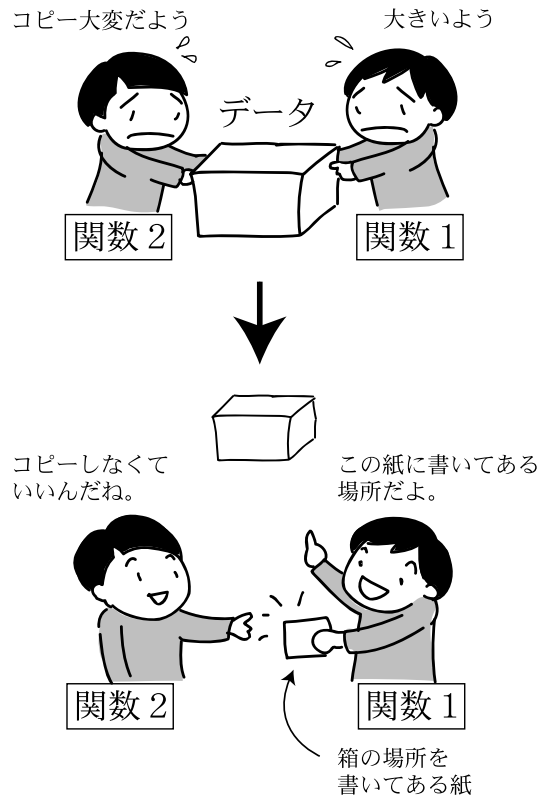
```

% ./struct-p
1/2 <-- input
1/3 <-- input
1/2 + 1/3 = 5/6
%

```

However, structures are usually passed to functions with pointers, avoiding data copies. For example, the two integer variables in our `struct bunsuu` consume 8 bytes or 16 bytes. But, a pointer only need the size of the pointer (4

bytes or 8 bytes). It may seem a minor point here, but when structures become large, the penalty in execution time for copying them becomes significant. Moreover, because local variables are stored on the stack, they rapidly consume your available stack space, which is often limited.



The next example shows how to pass structures using pointers. The result in this case is the same, but you have to understand the difference. Note that the third argument is used to hold the result, and its contents are modified *in place*.

```

/* struct-p2.c */
#include <stdio.h>
#include <stdlib.h>
#define MAXLINE 256

struct bunsuu {
    int shi;
    int bo;
};

void keisan(struct bunsuu *p,
            struct bunsuu *q,
            struct bunsuu *r){

    (*r).shi = (*p).shi * (*q).bo
              + (*p).bo * (*q).shi;
    (*r).bo = (*p).bo*(*q).bo;
    return;
}

int main(){
    int ret;
    char input [MAXLINE];
    struct bunsuu a,b,c;

    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%d/%d",&a.shi, &a.bo);
    if(ret != 2) {exit(1);}
    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%d/%d",&b.shi, &b.bo);
    if(ret != 2) {exit(1);}

    keisan(&a,&b,&c);
    printf("%d/%d + %d/%d = %d/%d\n",
           a.shi,a.bo,b.shi,b.bo,c.shi,c.bo);
    exit(0);
}

```

In this example, you have to understand the binding priority between “.” and “\*”. “.” indicates a member of a structure, and “\*” indicates dereferencing of a pointer. Writing simply `*p.bo` means `*(p.bo)`. This will give you the pointer that is the structure member named `p.bo`, not the member `bo` of the structure `*p`. To indicate the latter, `*p` must be put in parentheses, `(*p).bo`.

Since member references using pointers to structures occur frequently, a special notation is introduced in C. The following two formulas have the same meaning.

```
(*variable_name).member_name;
variable_name->member_name;
```

Using this simplified notation, the example program's fraction calculation can be rewritten as follows.

```
(*r).shi = (*p).shi * (*q).bo
          + (*p).bo * (*q).shi;

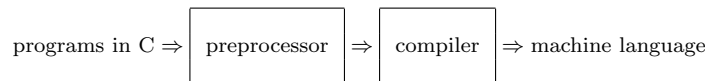
r->shi = p->shi * q->bo
        + p->bo * q->shi;
```

## 4.5 Tools

⇒ We will go back and pick up `git`, which we didn't get to in prior weeks.

### 4.5.1 The Preprocessor

In Section 3.6.1, we first presented the phases of compilation. Now let's look a little more closely at the preprocessor. The preprocessor performs a stage of compilation known as, well, *pre-processing*. A program written in C isn't translated to the language that the machine understands directly by the compiler. First, it is processed by the preprocessor, then passed to the actual C compiler. The compiler itself has several phases, and in the end outputs a not-quite-complete file that includes the binary *machine language*.



Instructions to the preprocessor begin with '#' in your program. We have already seen `#include`:

```
#include <file name>
```

In a previous lecture, `#include` was described as an instruction to use libraries. Actually, the preprocessor *replaces* that line in your program with the contents of the designated file. When the filename is specified with '<>', it means the file is provided by the system. The preprocessor searches for the specified file in system directories such as `/usr/include`. When the filename is specified with '"', the preprocessor searches for the specified file in the directory or directories specified by the user (typically including your current directory, named '.', or "dot", or "cwd").

Other than `#include`, `#define`, which we have also already been using, is one of the most common statements used to instruct the preprocessor.

```
#define MACRO_NAME Value
```

This is called a *macro definition*. In your program, `MACRO_NAME` is replaced with `Value` by the preprocessor before the program is passed to the compiler. For example, using a meaningful macro name to specify a constant number increases the readability and maintainability of the program, rather than specifying the number with the numeric value itself. For example, the second call to `sin()` is more readable than the first:

```
#include <stdio.h>
#include <math.h>
#define PI 3.1415926535
...
int main() {
    ...
    x = sin(1.570796327);
    y = sin(PI/2);
}
```

Besides simple numeric constants, there are often parameters of your program, such as the maximum supported problem size or timing intervals or the like, that you define at compile time, and that you might change later. In the array example program `array1.c` (p. 38), the size of the array is 10. If you want to change the size of the array from 10 to 20, you have to change 10 in the program to 20 in three different places. If you use macro definition, you only need to modify one line as follows:

```
/* arraymacro.c */
#include <stdio.h>
#include <stdlib.h>
#define ARRAYSIZE 20

int main(){
    int i, a[ARRAYSIZE];
    for(i=0; i<ARRAYSIZE; i++){
        a[i] = 1;
    }
    for(i=0; i<ARRAYSIZE; i++){
        printf("a(%d) = %d\n",i,a[i]);
    }
    exit(0);
}
```

There may seem to be little advantage for such a small example program, but for larger programs, this kind of *software engineering discipline* pays dividends throughout the life of the program (which is almost always longer than

you thought it would be). I once watched an inexperienced young programmer searching through a moderately long program with many loops over a 127-entry array that he was adapting to be a 128-entry array; it was tedious and, of course, error-prone; he accidentally changed a 127 somewhere in the program that corresponded to a *different* meaning than the array size, and had to debug that. The mnemonic character of `#defines` aids in understanding, debugging, and maintenance!

If you want to see the expanded program produced by the preprocessor, you may compile with the `-E` option. The expanded result is displayed on the standard output (e.g. `display`). This example redirects the output to the specified file.

```
% gcc -E arraymacro.c > arraymacro.cpre
% wc arraymacro.c*
    15      31    245 arraymacro.c
  1623   4635  41692 arraymacro.cpre
  1638   4666  41937 total
```

The `wc` utility counts the lines, words and characters (bytes) in a file. You can see that our fifteen-line C program expanded to 1,623 lines! Looking at `arraymacro.cpre` using an editor, most of the original content of our program `arraymacro.c` appears at the end of the file. Uses of the macro definitions have been replaced with their expanded values. Most of the length of our pre-processed file comes from the direct inclusion of the header files `stdio.h` and `stdlib.h`.

A macro definition may be written as a function with arguments. The next example shows how to define a macro that calculates the square of a number.

```
#define SQUARE(x) x*x
```

Using this macro, `'SQUARE(a)'` is expanded to `'a*a'`, and `'SQUARE(3)'` is expanded to `'3*3'`.

It is important to note that the macro definition doesn't care about the contents of its arguments, they are simply text strings. `'SQUARE(x+1)'` is expanded to `'x+1*x+1'`. So, the middle term `1*x` is calculated before the additions, in accordance with normal operator precedence rules:

```

/* macro-bad-sample.c */
#include <stdio.h>
#include <stdlib.h>
#define SQUARE(x)  x*x
int main(){
    int i,j;
    i = 5;
    j = SQUARE(i+1);
    printf("i=%d, SQUARE(i+1)=%d\n",i,j);
    exit(0);
}

```

In this program the loop variable `i` is set to 5. The programmer's intention is to calculate the square of `i+1`, or square of 6. However, the following result is obtained:

```

% ./macro-bad-example
i=5, SQUARE(i+1)=11
%

```

This failed because `SQUARE(i+1)` is expanded to `5+1*5+1`. To prevent this unintended expansion, you have to use parentheses.

```

#define SQUARE(x)  (x)*(x)

```

In this way, '`SQUARE(x+1)`' is expanded to '`(x+1)*(x+1)`'. Always, using parentheses is advised.

## 4.6 Exercises

- 4.1** Write a program that reads a string from the keyboard, converts all lowercase characters in the input string to uppercase characters, and displays them. Your program should use array indices instead of pointers.

To convert a character from the lowercase to the uppercase, the `toupper()` function in the standard library is used. You will need to add the line `#include <ctype.h>`.

To read the string, use `fgets()`. To count the number of characters input, use the library function `strlen(char *)`.

- 4.2** Write a program that produces the same result as the previous exercise. In this program, use a function to perform the actual work. Other than `main()`, you have to write the function `convtoupper(char *)` and pass a pointer to the string to this function. This function should convert lowercase characters to uppercase characters.

- 4.3** Write a program that reads a word from the keyboard, checks whether or not the word is a palindrome (such as “dad”, “mom”, “deed”, etc.), and prints the result.
- 4.4** Write a program that reads three strings from the keyboard. Assuming those strings are **A**, **B**, **C**, create three string variables that store **ABC**, **CBA**, **BACB**. Then, display those three newly created strings. Be careful that each of your string buffers is large enough to hold the maximum size its string might be.
- 4.5** Write a program that reads a long string from the keyboard, divides it into 5 character strings, stores them to variables, and displays them. You can specify the maximum size as `#define MAXLINE 256` and set the size of the other variables to be `MAXLINE/5+1`.
- 4.6** ★ Write a program that reads a string from the keyboard, randomizes the sequence of the strings, and displays the randomized string. The number of each letter in the output string must be the same as the number of each letter in the input string. You will want to use the library function `random()`. You may also need to know about the *modulo* operator `'%'`, which we have not used before: `13 % 4` will return 1, for example.
- 4.7** Write a program that performed the *tsuru-kame-san* in Exercise 1.3. In this exercise, the program reads the total numbers of heads and legs from the keyboard. Furthermore, *error checking* should be increased. When a non-number is input from the keyboard, a warning message should be displayed, then the program should prompt the user to reenter valid numbers, and loops without terminating. Also, the program should check that the numbers of heads and legs make sense. When a solution cannot be calculated, have the program display a warning and prompt the user to reenter numbers, then read the new numbers without terminating.
- 4.8** Write a program that reads two fractional numbers, and computes their sum, difference, product, and quotient, then displays all four results. In this program, those four calculations should be performed in four separate functions. Two arguments to each function are pointers to each fraction.



## Chapter 5

# Memory Management, Scope in Naming, and More Control

In which we learn how to manage memory for ourselves, and become grateful for programming systems that take care of the problem for us. We also discuss the scope of variable names.

### 5.1 Concepts

#### 5.1.1 Memory management

Modern computer systems manage much of their memory dynamically. As programs deal with more data during execution, additional memory must be *allocated* for use. When the program finishes using the memory, it should be *freed*. Many modern programming languages take care of this for you, using a technique known as *garbage collection* (GC). The concept of garbage collection was developed by John McCarthy, first for use in his programming language Lisp. (McCarthy invented several profoundly important ideas in computer science; he is also credited with the original idea for multiprogramming (or multitasking), and he coined the term “artificial intelligence”.)

Garbage collection requires strong typing of variables, limitations on the use of pointers, and complex support for tracking which chunks of memory are still in use and which are not. This tracking is often done using *reference counting* *reference count*, a standard technique in computer systems in which every pointer to a

structure or array is kept track of; when the last pointer is no longer in use, the structure may be garbage collected. The earliest systems used a *mark and sweep* two-phase approach in which every pointer in the program is checked first, and every data structure reachable from a pointer is marked. The whole of memory is then swept in the second phase, and any unmarked structures are garbage collected and put back into the free pool for reuse.

As you can imagine, GC is a moderately complex process to get right, even when using only a single, serial process (we will see threads and other multiprogramming techniques in Chapter 14 and elsewhere). Moreover, GC consumes a lot of CPU time, and can cause programs to stop doing other work while GC is performed, so it is not suitable for real-time systems. Because of C's looser constraints on types and the use of pointers, and the CPU overhead, GC is not used in C.

Instead, in this chapter we will learn how to manage memory for ourselves. This is a process fraught with danger, trekking through swamps filled with snakes and alligators and quicksand. It is intimately intertwined with the proper use of pointers. However, if you have the courage to see it through, at the end you will know how to write efficient, elegant programs that behave predictably.

The key is to understand the conditions in which new memory is allocated, and how to determine when memory is no longer in use and therefore can be freed. If your programs are written so that both are clear from the structure, you will have no troubles.

### 5.1.2 Scope

The *scope* of a variable is the area of the program over which its name can be used. Simply put, many languages support *local variables* valid only within a function call, and *global variables* that are valid over the entire lifetime of the program. A variable may also be valid over a region of the program known as a code block.

## 5.2 Dynamic memory use with pointers

Usually, data are stored in variables, and storage areas are allocated before execution. This storage allocation is done at the time of program compilation. However, when the program reads and processes data from a file, pre-allocated storage may be not sufficient. To adapt storage allocation, a dynamic storage allocation scheme is used in C. The dynamically allocated data storage area is called the **HEAP**<sup>1</sup>. To allocate a storage area at execution time, the `malloc()`

<sup>1</sup>“Heap” is also the name of a specific type of data structure for organizing a group of elements, which we won't cover here.

function is used, and the `free()` function is used to free the allocated area.

- `void *malloc(size_t size)`: allocates `size` bytes area, and returns the pointer to the area. Note that the return type is `void *`. This is a pointer to a variable of an unnamed type, and can be assigned to any variable pointer.
- `free(void *p)`: frees the area that is allocated by the `malloc()` function. The area is indicated by the pointer `p`.

In the function `malloc()`, the size of data is specified in terms of bytes. Because the size of the specific data type varies from computer to computer, the `sizeof()` operator is used to determine the size of the specific type.

```
sizeof(type)  
sizeof(variable_name)
```

This operation provides the size of `type` or `variable_name` in bytes. The following example shows how to use the `malloc()` function.

```

/* malloc-example.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

#define MAXLINE 16

int main(){
    int len;
    char input[MAXLINE];
    char *s,*p,*q;

    fgets(input,MAXLINE,stdin);
    s = strchr(input,'\n');
    if(s!=NULL){
        *s = '\0';
    }
    len = strlen(input);
    p = malloc(sizeof(char)*(len+1));
    strcpy(p,input);
    q = malloc(sizeof(char)*(len+4));
    strcpy(q,p);
    strcat(q,"END");
    printf("*p=%s\n",p);
    printf("*q=%s\n",q);
    printf("input=%p,&len=%p\n",input,&len);
    printf("p=%p,q=%p\n",p,q);
    printf("&p=%p,&q=%p,&s=%p\n",&p,&q,&s);
    exit(0);
}

```

The program reads a string, allocates a storage area for the string using `malloc()`. After the allocation, the program copies the input string to that area. Next, it allocates another area that is 3 characters larger than the previously allocated area, copies the input string to that area using `strcpy()`, then adds the string "END" to the end of that string. Note that here we have used `strcpy()` instead of `strncpy()`—since we have been careful to track the size of the string and allocate enough memory, but using `strcpy()` is still the better habit.

The execution result is shown below.

```
% ./malloc-example
ABC
*p=ABC
*q=ABCEND
input=0x7fff5fbff8b0,&len=0x7fff5fbff8ac
p=0x100100080,q=0x100100090
&p=0x7fff5fbff898,&q=0x7fff5fbff890,
&s=0x7fff5fbff8a0
```

Figure 5.1 depicts the memory allocation of a sample run of this program. It is important to understand the location of the 16-byte array `input[MAXLINE]` and the locations pointed to by the pointers `p` and `q`.

The area beginning at `0x000100100080` is the heap area. Note the difference between the addresses of variables on the heap and the addresses of ordinary variables. The latter, which were defined as local variables in `main()`, are allocated space on the stack.

### 5.3 Scope of variables

The concept of scope applies to all variables. The next program contains three functions, `main()`, `f1()`, and `f2()`.

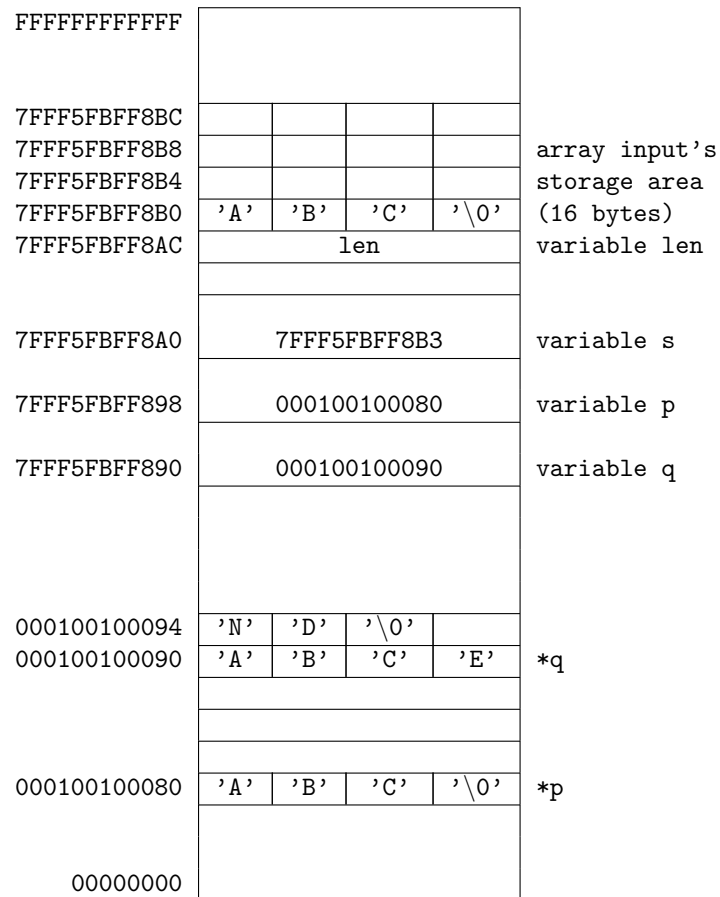


Figure 5.1: Memory map for a sample execution of malloc-example.

```
/* scope-example.c */
#include <stdio.h>
#include <stdlib.h>

int i = 1; /* global */

void f1(){
    int i = 100; /* f1 local */

    for(;;){
        int i = 1000; /* for local */
        printf("i = %d (in for in f1)\n",i);
        {
            i = 2000;
            int i;
            i = 10000;
            printf("i = %d (in for in f1-2)\n",i);
        }
        printf("i = %d (in for in f1)\n",i);
        break;
    }
    printf("i = %d (in f1)\n",i);
}

void f2(){
    printf("i = %d (in f2)\n",i);
}

int main(){
    int i;

    i = 10;
    printf("i = %d (in main)\n",i);
    f1();
    f2();
    printf("i = %d (in main)\n",i);
    exit(0);
}
```

The result is shown below. Several values for *i* are printed. Can you identify which *i* comes from which definition?

```
% ./scope-example
i = 10 (in main)
i = 1000 (in for in f1)
i = 10000 (in for in f1-2)
i = 2000 (in for in f1)
i = 100 (in f1)
i = 1 (in f2)
i = 10 (in main)
```

*Global variables* are defined outside of any function (including `main()`). They have the widest scope, and can be used anywhere. They can even be used from other source files, if an external declaration is provided for them in the other files.

Another variable type is the *local variable*. Local variables are defined inside a block. A block is the area between the open and close curly brackets `{` and `}`. Usually, local variables are defined at the beginning of the function. Furthermore, local variables cease to exist once the function (or the block) that created them is completed.

As shown in the example, if there are variables that have the same name, the variable that is defined in the innermost block has priority. For example, in the function `f1()`, `i = 100`, but inside of the `for(;;)` loop, `i = 1000` is given priority.



## 5.4 Control structures

Control structures are used to select which portions of your program are executed and how many times. They are implemented as conditional branches and loops. Like Java, C has `for`, `while`, and `if` statements, which we have already seen. In this section, other control structures are discussed.

### 5.4.1 `switch`

When you want to process several cases based on the value of an expression, you need a lot of `if...else if...else if...` clauses. In such cases, the `switch` statement may be more effective.

Depending on the input, the following example determines whether you meant 'Yes' or 'No'.

```
/* ifelse-example.c */
#include <stdio.h>
#include <stdlib.h>
#define MAXLINE 256

int main(){
    int ret; char cin;
    char input[MAXLINE];

    printf("Yes or No ? ");
    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%c",&cin);
    if(ret <= 0){
        printf("Invalid input\n");
        exit(1);
    }
    if (cin == 'y')
        printf("Yes entered\n");
    else if (cin == 'Y')
        printf("Yes entered\n");
    else if (cin == 'n')
        printf("No entered\n");
    else if (cin == 'N')
        printf("No entered\n");
    else
        printf("I don't know %c\n",
            cin);
    exit(0);
}
```

If `{` and `}` are added to demarcate a block, the program will be as follows.

```

/* ifelse-example.c */
#include <stdio.h>
#include <stdlib.h>
#define MAXLINE 256

int main(){
    int ret; char cin;
    char input[MAXLINE];

    printf("Yes or No ? ");
    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%c",&cin);
    if(ret <= 0){
        printf("Invalid input\n");
        exit(1);
    }
    if (cin == 'y'){
        printf("Yes entered\n");
    }else if (cin == 'Y'){
        printf("Yes entered\n");
    }else if (cin == 'n'){
        printf("No entered\n");
    }else if (cin == 'N'){
        printf("No entered\n");
    }else{
        printf("I don't know %c\n",
            cin);
    }
    exit(0);
}

```

You can see that this has become tedious and error prone. Instead, we can use the `switch` statement, which is syntactically simpler and visually easier for humans to parse. The general form of the `switch` statement is shown below.

```

switch ( expression ) {
    case constant: statement; ... statement; break;
    case constant: statement; ... statement; break;
    ...
    default: statement; ... statement; break;
}

```

First, the program evaluates `expression`. Next, it searches for the corresponding `constant`, then executes the set of `statement`s following. After executing the `break` statement, execution of the `switch` statement completes and the program executes the next statement just after the `switch` block. When none of the

`constant` clauses matches the value of `expression`, the statements after `default` are executed. If there is no `default`, and no match, no `statement` is executed.

The previous example can be rewritten using `switch` as follows.

```
/* switch-example.c */
#include <stdio.h>
#include <stdlib.h>
#define MAXLINE 256

int main(){
    int ret; char cin;
    char input[MAXLINE];

    printf("Yes or No ? ");
    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%c",&cin);
    if(ret <= 0){
        printf("Invalid input\n");
        exit(1);
    }
    switch(cin){
        case 'y':
            printf("Yes entered\n");
            break;
        case 'Y':
            printf("Yes entered\n");
            break;
        case 'n':
            printf("No entered\n");
            break;
        case 'N':
            printf("No entered\n");
            break;
        default:
            printf("I don't know %c\n",
                cin);
    }
    exit(0);
}
```

At the `break` statement, the execution of `switch` ends. Control is transferred to the next statement after the `switch` block. You may combine clauses that need the same processing. Note the different layout style used for yes and no. This was done just for illustration; you should pick one style and stick with it, for clarity.

```

#include <stdio.h>
#include <stdlib.h>
#define MAXLINE 256

int main(){
    int ret; char cin;
    char input[MAXLINE];

    printf("Yes or No ? ");
    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%c",&cin);
    if(ret <= 0){
        printf("Invalid input\n");
        exit(1);
    }
    switch(cin){
        case 'y':
        case 'Y':
            printf("Yes entered\n");
            break;
        case 'n': case 'N':
            printf("No entered\n");
            break;
        default:
            printf("I don't know %c\n",
                cin);
    }
    exit(0);
}

```

### 5.4.2 break

As shown in the `switch` examples we have just seen, the `break` statement completes execution of the statement. It can also be used to finish or abort execution of a `while` statement or a `for` statement. For example, the next program searches for the first number that is between `p` and `q`, and divisible by `r`.

Executing the `break` statement completes the innermost `switch`, `while`, `for`, or `do` statement that includes the `break` statement.

```
/* break-example.c */
#include <stdio.h>
#include <stdlib.h>
#define MAXLINE 256

int main(){
    int i, p, q, r, ret;
    char input[MAXLINE];

    printf("Enter the beginning number ? ");
    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%d",&p);
    if(ret <= 0) {exit(1);}
    printf("Enter the ending number ? ");
    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%d",&q);
    if(ret <= 0) {exit(1);}
    printf("Enter a divisor ? ");
    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%d",&r);
    if(ret <= 0) {exit(1);}
    for (i = p; i < q; i++) {
        if(i%r==0){
            printf("From %d to %d, the first number",
                p,q);
            printf(" divisible by %d is %d\n",
                r,i);
            break;
        }
    }
    exit(0);
}
```

The result is shown below.

```
% ./break-example
Enter the beginning number ? 100
Enter the ending number ? 300
Enter a divisor ? 13
From 100 to 300, the first number
divisible by 13 is 104
%
```

### 5.4.3 continue statement

The `continue` statement skips the innermost loop of `switch`, `while` and `for` statements, then returns to the beginning of the loop. For example, in the next program, when the variable `i` is greater than or equal to 90, `statement 2` is not executed. `statement 1` is executed 100 times.

```
for (i = 0; i < 100; i++) {
    statement 1;
    if (i >= 90) continue;
    statement 2;
}
```

The next program displays numbers that are between `p` and `q`, and divisible by 7 and 13.

```
/* continue-example.c */
#include <stdio.h>
#include <stdlib.h>
#define MAXLINE 256

int main(){
    int i, p, q, ret;
    char input[MAXLINE];

    printf("Enter the beginning number ? ");
    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%d",&p);
    if(ret <= 0) {exit(1);}
    printf("Enter the ending number ? ");
    fgets(input,MAXLINE,stdin);
    ret = sscanf(input,"%d",&q);
    if(ret <= 0) {exit(1);}
    for (i = p; i < q; i++) {
        if(i%7 != 0){
            continue;
        }
        if(i%13 != 0){
            continue;
        }
        printf("%d is divisible by 7 and 13\n",
            i);
    }
    exit(0);
}
```

The result is shown below.

```
% ./continue-example
Enter the beginning number ? 100
Enter the ending number ? 300
182 is divisible by 7 and 13
273 is divisible by 7 and 13
%
```

## 5.5 Tools

### 5.5.1 Interpreting gcc's error messages

By now, you have hit numerous errors when attempting to compile your programs. Let's look a little more closely at the error messages generated by gcc. For example, we might see

```
% gcc-5 list1.c
list1.c:11:1: error: unknown type name 'nt'
  nt main(){
  ^
%
```

if our source code file looked like

```
...
nt main(){
  // we will use p as a temporary
...

```

If you are sharp-eyed, you might have noticed that instead of `int main()` we have `nt main()`, a simple typo where we left off (or accidentally deleted) the initial `i`. The error message from gcc is actually telling us exactly where to find the error: `list1.c:11:1` tells us it is in file `list.c` on line 11, at column 1. It tells us that it didn't recognize the word `nt` as an acceptable return value type; here C's moderately strong type checking catches an error for us.

If instead our source code file had a different typo,

```
...
int main){
  // we will use p as a temporary
...

```

we would get the following error message:

```
gcc-5 -o list1 list1.c
list1.c:11:9: error: expected '=', ',', ';',
'asm' or '__attribute__' before ')' token
int main){
    ^
```

Note that tells us where it ran into trouble (column 9 this time), and even includes the line of code and marks it for us with a `^` below the spot.

In this case, the compiler had more trouble figuring out what went wrong, so you must use a little more of your own brainpower. In some programming languages with a very restricted syntax (such as Lisp), the system can more easily recognize where you have gone wrong, but C's grammar (the set of rules defining the order in which words and symbols may appear) is more flexible, which makes it harder to pinpoint problems. The error message may point to a position far from the location of your actual mistake.

Note that this error message refers to `)` as a *token*, which is what compiler people call an individual term or symbol. A token may be more than one character long, as in `+=`.

## 5.6 Exercises

**5.1** If a number is only divisible by 1 and the number itself, the number is called a prime number (Sosuu in Japanese). For example, 2, 7, and 19 are prime numbers. In this exercise, the sieve of Eratosthenes (Greece, 276?–194 B.C.) is used for finding prime numbers between 1 and 10000. The program should execute the following algorithm.

- creates the sequence from 2 ~ 10000.
- the minimum number remaining is 2, so the program erases all multiples of 2.
- In the remaining numbers, the minimum number remaining is 3, so the program erases all multiples of 3.
- repeats this procedure.
- Remainings are prime numbers.

To program this algorithm, you may define the sequence of 2 ~ 10000 as an array.

```
char prime[10001];
```



In this method, `prime[i]` stores whether  $i$  is a prime number or not. For example, if `prime[i]` is 0,  $i$  is not a prime number. if `prime[i]` is 1,  $i$  is a prime number. First, the program initializes this array. Then, it changes the value of `prime[i]` based on the algorithm.

To test the program with a small number, it is better to define the following macro and change its value to 10001 at the final test.

```
#define MAXNUMBER 10001
```

- 5.2 Write a program that performs integer factorization from for numbers from 1 to 1000. The output example is shown below.

```
% ex4-8
2 = 2
3 = 3
4 = 2 x 2
5 = 5
6 = 2 x 3
....
102 = 2 x 3 x 17
103 = 103
104 = 2 x 2 x 2 x 13
....
```

- 5.3 Write an interactive demonstration of a *buddy memory allocator*. Your demo should allow 64 blocks to be allocated and deallocated, and it should correctly merge freed blocks. The output should be something like that in Fig. 5.2.

Each time your program splits a block, it should add a '|' to show where the split happened. '-' represents free blocks, and '#' represents in-use blocks. The two numbers printed out as in 0/4 represent the beginning address and number of blocks in a chunk.

Your program should take three one-letter commands: 'a' for allocate, 'f' for free, and 'q' for quit. The second argument on a line should be the number of blocks (for allocate) or the *address*, or first block number, of the allocated chunk (for free). If two neighboring blocks are both allocated, but were part of separate allocations, be careful to only free the one in the actual request!

For large numbers of blocks, there are many possible data structures you could use to track the set of blocks that are in use and the size of chunks.

```

% ./buddy-demo
|-----|
How many blocks do you want to allocate/free?
a 4
(splitting 0/64)
(splitting 0/32)
(splitting 0/16)
(splitting 0/8)
Blocks 0-3 allocated:
|###|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
How many blocks do you want to allocate/free?
a 16
Blocks 16-31 allocated:
|###|----|-----|#####|-----|-----|-----|-----|-----|-----|-----|-----|
How many blocks do you want to allocate/free?
f 0
(merging 0/4 and 4/4)
(merging 0/8 and 8/8)
Blocks 0-3 freed:
|-----|#####|-----|-----|-----|-----|-----|-----|-----|-----|-----|
How many blocks do you want to allocate/free?
q
%

```

Figure 5.2: Interactive session with a buddy allocator demo.

For this small number, I encourage you to take the simplest approach you can.



## Chapter 6

# Input/Output

In which we learn the three basic paradigms of reading and writing files in C on UNIX systems: the `read()/write()` family of system calls; the `fread()/fwrite()` family of library calls for writing binary files that complement the simple text input/output of `fprintf()` and `fscanf()`; and `mmap()`. We will also, finally, learn how to parse command line arguments so we can specify names and numbers. Oh, and we'll see how to modify the behavior of a program depending on local environment variables that express a user's preference.

### 6.1 Concepts

#### 6.1.1 Files and I/O

You are probably aware that, in order to make data be *persistent* (not disappear when we reboot or lose power), data must be stored in *files* or in a *database* (which we will not otherwise cover in this class). Files allow us to keep data for a long time, or to share data between programs. (In the second half of this book, we will see other methods of sharing data between programs that are both executing at the same time; some of those methods build on the concepts in this chapter.)

The standard C library, when it was introduced in the early 1970s, simplified I/O through use of a few basic interfaces for reading and writing lines of files. In this class, we will use those interfaces almost exclusively. However, let's take a short look at the broader concepts.

Broadly speaking, there are several paradigms we can use for input/output, or I/O:

- line-oriented interfaces for reading and writing files one line at a time, used for text files such as configuration files, log files, (uncompiled) programs, HTML and XML files, and many homebrew data files for programs;
- character-oriented interfaces for dealing with one character at a time, often used for programs that react instantly to single-character commands, including editors;
- fixed-size or block-oriented interfaces, used to specify the exact number of bytes to be read or written (on some systems, the size of I/O block may be limited to e.g. a multiple of 512 bytes, but in UNIX systems this can be any size);
- record-oriented interfaces, which may be for either fixed-size records (such as writing out a specific data structure), or variable-sized (typically lines, as above); and
- memory-oriented interfaces.

The C library and the UNIX system calls collectively support all of these interfaces. UNIX directly supports only fixed-size I/O and memory-oriented interfaces; the rest are built as library functions which in turn use the system calls, as shown in Figure 6.1.

It is important to note that the standard C library and UNIX share a model of what a file *is*: they both assume that a file is simply an ordered sequence of 8-bit bytes. Any apparent structure is imposed only by the application software used to read or write the file, and if you write a file using one model (lines, for example) and read it using another (fixed size), you can misinterpret the data or even mess up the file. In some other operating systems, the OS itself may impose structure on a file, accessed through special system calls, which may be available to C programs. In this book, we will use only the standard C library and UNIX system calls.

All of these use a *file session* approach: first the file must be prepared, or *opened*, for I/O; the I/O operations are then performed; then the file is *closed*. With the exception of `mmap()`, while the file is opened, an *offset* is maintained by the system for you; if you perform I/O repeatedly, you will move through the file in a regular fashion from beginning to end. We will see below how that position may be modified when necessary.

I/O may be done *synchronously* or *asynchronously*. In this book, we will consider only synchronous I/O, which means that when the system call or library call completes, you can behave as if the I/O is finished – either the data is read into your buffer, or written to disk or flash or other target. (This latter is not strictly true, which we will discuss when we discuss closing files and “syncing” files to disk.)

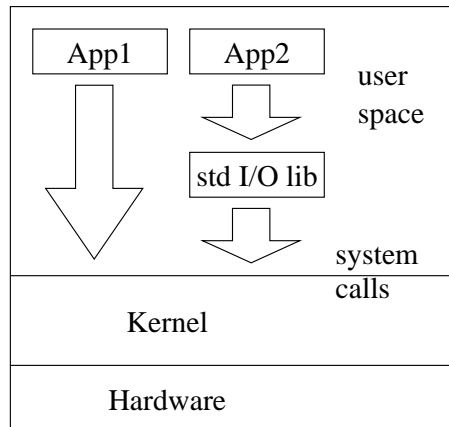


Figure 6.1: Relationship of applications the standard I/O library and to system calls. Any operation that touches a hardware device such as a disk, screen window, keyboard or network interface ultimately does so using a system call, but may solicit the help of the standard I/O library to simplify some operations.

All of these functions assume a single contiguous memory area, or *buffer*, is used for each operation. It is possible to collect data from multiple areas for a single operation. This is known as *scatter-gather*, which we will not use in this book.

Table 6.1 lists input and output functions frequently used in C. To read and write line by line, you can use `fprintf()`, `fscanf()` and other functions, which will read or write variable amounts of data depending on the content of the data itself. To read and write byte by byte, you can use `getc()` and `putc()` and the like, all from the standard I/O library. To read or write a fixed amount of data, with no additional processing, you can use the UNIX system calls `read()` and `write()`.

The input or output location specified with `fprintf()` and `fscanf()` is called a **stream**. The input or output location specified with `read()` and `write()` functions is called a **file descriptor**. We will look at both, below.

### 6.1.2 Command line arguments

⇒ *To be written.*

### 6.1.3 Environment variables

⇒ *To be written. Especially important when internationalizing software.*

## 6.2 Standard Input/Output

In Java, the `java.io` package is used to read and write files. In C, either the standard input/output library or the UNIX system calls may be used. The UNIX system calls give you a “bare” interface to reading and writing any kind of data in a file, whereas the standard I/O library gives you partially processed data, often making your life easier. A third paradigm involves rather different modes of thinking, essentially making part of a file into a part of your memory.

The `fgets()` function for one line input, and the `printf()` function for formatted output have been discussed; these are both from the standard library. The `printf()` function outputs numbers and characters to your screen, or more correctly to what we call the *standard output*. Why is this called the “standard” output? When no output device is specified by the programmer, it is sent to a common output determined at execution time. Below, we will see how to modify where this output goes.

Our original “Hello, world” example displays a string to the standard output.

```
/* hello.c */
#include <stdio.h>
#include <stdlib.h>
int main(){
    printf("Hello C World!\n");
    exit(0);
}
```

The result is shown below.



```
% gcc hello.c -o hello
% ./hello
Hello C World!
%
% ls
hello hello.c
% ./hello > out
% ls
hello out hello.c
% cat out
Hello C World!
%
```

First, executing `hello` displays `Hello C World!` Next, adding `> out` *redirects* redirection the output to the file `out`. In this way, the actual location of the standard output can be changed at execution time. Note that this behavior is a feature of UNIX shells, but any C implementation (except some embedded environments) will support standard input and output, although perhaps not redirection in this fashion.

The next example shows how to read from and redirect the standard input. This program reads one line from the standard input, then display the number of characters in that line.

```
/* stdinput.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(){
    char input[100];
    fgets(input,100,stdin);
    printf("Input is %d characters\n",
           (int)strlen(input));
    exit(0);
}
```

The following shows the result. Inputting the string `hello` results in a reported length of 6 characters, because the new line character is included, but not the trailing NUL character.

```
% ./stdinput
hello
Input is 6 characters
%
```

Using `<`, the next example uses the file `input` as the standard input. In this case, a length of 16 characters is reported.

```
% mv out input
% cat input
Hello C World!
% ./stdininput < input
Input is 15 characters
%
```

## 6.3 Stream input and output

### 6.3.1 `fopen()/fprintf()/fgets()/fclose()`

File I/O with a stream is done by following these steps:

- Call the `fopen()` function once, making the specified file available to use. (This is called “opening a file”)
- Call the `fprintf()` function once or more, writing numbers and characters to the file; or, the `fgets()` function, reading numbers and characters from the file.
- Call the `fclose()` function once, ending use of the file.

Three files (streams for I/O) are automatically opened at the start of the program execution: standard input, standard output, and standard error, specified by the names `stdin`, `stdout`, and `stderr`. Because they are already available, you can use `printf()`, `scanf()` and other functions without any preparation.

The next example creates the file `output`, then writes the string `Hello C World` into that file.

```
/* fprintf-example.c */
#include <stdio.h>
#include <stdlib.h>
int main(){
    FILE *fp;
    fp = fopen("output","w");
    fprintf(fp,"Hello C World\n");
    fclose(fp);
    exit(0);
}
```

The execution result is shown below. After running `fprintf-example`, the next prompt appears immediately, with no output or indication that anything

happened; after all your program doesn't send anything to `stdout`. But, using the `ls` command, you can see that the file `output` has been created. Using the `cat` command, you can see the contents of that file.

```
% gcc fprintf-example.c -o fprintf-example
% ./fprintf-example
% ls
fprintf-example fprintf-example.c output
% cat output
Hello C World
%
```

The first argument of the `fprintf()` function is the value that specifies the location of the output. Usually, the return value of the `fopen()` function is used. Otherwise, `stdout` and `stderr` can be used to specify the standard output and the standard error. For `fscanf()` and other functions that read values, `stdin` can be used to specify the standard input.

The following program uses the `fprintf()` function to display `Hello C World` on the standard output.

```
#include <stdio.h>
int main(){
    fprintf(stdout,"Hello C World\n");
    exit(0);
}
```

Next, what is the standard error output? For example, think of the output from `ls`. Normally it displays a list of files.

```
% ls
file1
% ls > list
% cat list
file1
% ls
file1 list
% ls file2 > list
file2: No such file or directory
% cat list

%
```

In this example, `>` is used to redirect the output of the `ls` command to the file `list`. However, the second `ls` command displays an error message, because the file `file2` doesn't exist. Even though we redirected `stdout`, the error message is displayed on the screen, not stored in the file `list`. The `ls` command outputs the list of files to the standard output, but error messages are output to the standard error output. When an error occurs, it is better that the message appears on the screen, not in the file.

The next example reads data from a file using `fgets()`.

```
/* fgets-example.c */
#include <stdio.h>
#include <stdlib.h>
#define MAXLEN 512
int main(){
    FILE *fp;
    char input[MAXLEN];
    fp = fopen("output","r");
    fgets(input,MAXLEN,fp);
    fclose(fp);
    printf("%s",input);
    exit(0);
}
```

This program opens the file `output` (containing data written by the previous example), reads one line using the `fgets()` function, then displays it using the `printf()` function.

### 6.3.2 Handling errors from library calls

In the previous example, what occurs if the file `output` doesn't exist?

```
% ls output
output: No such file or directory
% ./fgets-example
Bus error
% ls *core
core
% file a.out.core
core: ELF 32-bit LSB core file Intel 80386,
version 1 (SYSV), SVR4-style, SVR4-style,
from fgets-example'
%
```

When a program opens a non-existent file using `fopen()`, then attempts to read data from the file, a **Bus Error** occurs. (Depending on the operating system, a **Segmentation Fault** may occur instead.) Furthermore, a file named **core** (In some operating systems, **a.out.core**) is created. The **core** file is created by abnormal termination of the program. It is used to analyze the cause of the termination. Right now, this file isn't needed, so you may delete it using the **rm** command.

**Depending on the your OS and the setting of variables that control the behavior of your shell, the core file may not be created. You can use the `ulimit` command to check or change the value of `coredumpsize`. If the core file size limit is 0, no core dump will be created.**

On a Mac, you will need to change the core filesize limit, and core dump files will be stored in a special directory, `/cores`. Note that the core files are enormous, remove them when you are done debugging.

```
ulimit -c unlimited
$ ./fgets-example
Segmentation fault: 11 (core dumped)
$ ls -l /cores
total 900920
-r-----  1 rdv  admin  461271040 May 29 18:36 core.13804
drwxrwxr-t@ 3 root  admin      102 May 29 18:35 ./
drwxr-xr-x 40 root  wheel    1428 May 21 09:40 ../
$ file /cores/core.13804
/cores/core.13804: Mach-O 64-bit core x86_64
```

To prevent such an error, you need to check the return value of `fopen()`. When the file open fails, the `fopen` function returns a NULL pointer. *You should check the return result of every system or library call!*

```

/* fgets-example2.c */
#include <stdio.h>
#include <stdlib.h>
#define MAXLEN 512
int main(){
    FILE *fp;
    char input[MAXLEN];
    fp = fopen("output","r");
    if(fp == NULL){
        fprintf(stderr,"fopen error\n");
        exit(1);
    }
    fgets(input,MAXLEN,fp);
    fclose(fp);
    printf("%s",input);
    exit(0);
}

```

```

% ./fgets-example
fopen error
%

```

To display the details of the error, you can use the `perror()` library function. To use this function, the `errno.h` header file is required.

```

/* fgets-example3.c */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define MAXLEN 512
int main(){
    FILE *fp;
    char input[MAXLEN];
    fp = fopen("output","r");
    if(fp == NULL){
        perror("fopen");
        exit(1);
    }
    fgets(input,MAXLEN,fp);
    fclose(fp);
    printf("%s",input);
    exit(0);
}

```

The `perror()` function displays the reason for the error.

```
% ./fgets-example
fopen: No such file or directory
%
```

Keep in mind, whenever you create programs, that you should write procedures that handle abnormal conditions and errors. *Every* system call or library call should be checked for an error return! For that matter, many of your own functions should be defensively coded so that they check their arguments and check for problems in execution, and may want to return an error on some occasions, too.

### 6.3.3 Skipping around in a file

As we noted above, the system maintains a file offset for you. Sometimes, though, you want to move to a different place in a file, in order to read or write there. Some file formats, for example, start with a description of the rest of the file, so you might read the beginning then skip to the place containing the data you are interested in. This can be done by *seeking* in the file.

The standard C library provides several functions to seek within a file. `fseek()` is the only one you really need.

```
int
fseek(FILE *stream, long offset, int whence);
```

The argument `whence` tells the call to count from the beginning, the current position, or the end of the file, by specifying `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, respectively.

## 6.4 Input and output using file descriptors

### 6.4.1 `open()/write()/read()/close()`

The streams we have just discussed are a standard C approach to I/O. On a UNIX system, they are built on top of the `read()` and `write()` system calls. On some occasions, you may want to perform the I/O directly instead of using the library. Follow these steps to perform file input and output with file descriptors:

- Call the `open()` function, making the specified file available for use. At this time, the choice of mode (read-only, write-only, or read-and-write), action when the specified file doesn't exist, protection mode of the file, and other details can be specified. (See the `man 2 open` page for more details.)

- Call the `write()` function, writing the specified number of bytes to the file, or the `read()` function, reading the specified number of bytes from the file. You may call these more than once.
- Call the `close()` function, ending use of the file. *This is important, and you should check the return value!*

The `write()` and `read()` functions perform input and output based on a size in bytes. These functions have the following formats.

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf,
              size_t nbyte);
```

```
#include <unistd.h>
ssize_t read(int fildes, const void *buf,
             size_t nbyte);
```

`int fildes` is the file descriptor. A small integer is assigned to the descriptor, and that number distinguishes the location. The value 0 is assigned to the standard input, the value 1 is assigned to the standard output, and the value 2 is assigned to the standard error output. Other values are assigned dynamically as a result of the `open()` call.

The second argument `const void *buf` is the pointer that specifies the location of the character array. In that array, data is stored. The third argument `size_t byte` specifies how many bytes are to be read or written. The return type of the `read` and `write` functions is `ssize_t`, usually synonym for `int`. The return value is the actual number of read or written bytes. For example, calling the `read()` function specifying 1000 bytes returns 500 when only 500 bytes have been read. Usually, this phenomenon occurs at the end of a file.

The next example uses the `write` function to display `Hello C World`.

```
/* write-example.c */
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
int main(){
    const char *hello = "Hello C World\n";
    write(1,hello,strlen(hello));
    exit(0);
}
```

To use file descriptors other than the three standard ones, the `open()` function has to be called before use. The next program stores values in an array to the file `array.dat`.



```

/* arraystore.c */
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
int main(){
    int a[6]={2,3,5,8,13,21};
    int len,fd;
    fd = open("array.dat",O_WRONLY|O_CREAT,
              0644);

    len = sizeof(a);
    write(fd,a,len);
    close(fd);
    exit(0);
}

```

It creates `array.dat` for data storage. Using `ls`, you can see that the size of the file is 24, corresponding to the 6 ints of data.

```

% a.out
% ls -lo array.dat
-rw-r--r-- ..... 24 ..... array.dat
%

```

The next program reads this file `array.dat` and displays its contents.

```

/* arrayread.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
int main(){
    int a[100];
    int i,fd;
    fd = open("array.dat",O_RDONLY);
    read(fd,a,sizeof(int)*6);
    close(fd);
    for(i=0;i<6;i++){
        printf("a[%d]=%d\n",i,a[i]);
    }
    exit(0);
}

```

The result is shown below.

```
% ./arrayread
a[0]=2
a[1]=3
a[2]=5
a[3]=8
a[4]=13
a[5]=21
%
```

### 6.4.2 Handling errors from system calls

When `array.dat` doesn't exist when we run `arrayread`, what happens?

```
% ls array.dat
array.dat: No such file or directory
% ./arrayread
a[0]=0
a[1]=0
a[2]=-1073743960
a[3]=-1881084160
a[4]=5052
a[5]=17349
%
```

Unlike our earlier program that used `fopen()`, the program runs, and terminates without an error. But, the result is incorrect. This is important to note: *just because your program didn't crash doesn't mean it did the right thing!*

The reason the earlier program crashed was that we were using pointers to data structures (the file stream), and those pointers were junk. Using junk pointers is the quickest way to get your program to actually crash. This program didn't operate properly, but it didn't crash, because there was no misuse of pointers. It just silently did the wrong thing.

To prevent this type of problem, you have to check the return value of the `open()` function. The `open()` function returns `-1` when something goes wrong.

```

/* arrayread2.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
int main(){
    int a[100];
    int i,fd;
    fd = open("array.dat",O_RDONLY);
    if(fd == -1){
        perror("open array.dat");
        exit(1);
    }
    read(fd,a,sizeof(int)*6);
    close(fd);
    for(i=0;i<6;i++){
        printf("a[%d]=%d\n",i,a[i]);
    }
    exit(0);
}

```

The result is shown below. Before you run this program, confirm that the file `array.dat` doesn't exist.

```

% ./arrayread2
open array.dat: No such file or directory
%

```

Note that you must check the result of your `read()` system calls as well; it may be that the file exists but doesn't contain enough data to fulfill your request, in which case you will get less data than you expected. Your program should be prepared to handle this.

### 6.4.3 Skipping around in a file

To seek in a file being read or written using the system calls, use the system call `lseek()`.

```

#include <unistd.h>

off_t
lseek(int fildes, off_t offset, int whence);

```

Its use is (deliberately) almost identical to `fseek()`, except for the file descriptor argument. The argument `whence` tells the call to count from the beginning, the current position, or the end of the file, by specifying `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, respectively. Note that you do need the `unistd.h` include.

## 6.5 `mmap()`

⇒ *To be written.*

## 6.6 Making file names from strings

If file names used in programs are determined at compilation, it is easy to use those names. However, creating file `file-1` to `file-n` according to the input number `n` requires some new techniques. Fortunately, file names in C are just character strings of the type we have already been using.

To create names in a program, the `snprintf()` function is useful. `snprintf()` is almost same as the `fprintf` function, but the function outputs data to a specified character array in memory, rather than to the terminal or a file. The format is shown below.

```
#include <stdio.h>
int snprintf(char *s, size_t size, const char *format, ..);
```

The next program creates 5 files, `file1` to `file5`, then stores 1 to 5 in those files.

```
/* snprintfing-filenames.c */
#include <stdio.h>
#include <stdlib.h>
int main(){
    FILE *fp;
    char name[100];
    int i;
    for(i=1;i<=5;i++){
        snprintf(name,100,"file%d",i);
        printf("Creating %s\n",name);
        fp = fopen(name,"w");
        fprintf(fp,"%d\n",i);
        fclose(fp);
    }
    exit(0);
}
```

The result is shown below. At every creation of a file, the file name is displayed.

```
% ./snprintfing-filenames
Creating file1
Creating file2
Creating file3
Creating file4
Creating file5
% ls
file1 file2 file3 file4 file5
% cat file1
1
%
```

## 6.7 Command line arguments

Almost every command we have used so far from the shell, include `cp`, `rm`, `ls` and `make`, is actually a program compiled much like the approach we have been using. (To see where a program actually is, you can use `which make` and the like, and the shell will tell you.) When you add some text after the command name, that is called a *command line argument* (or arguments). How can you use command line arguments in your own programs? The `main()` function has two arguments we have ignored so far, known as `argc` and `argv`.

The next program performs like the `echo` command.

```
/* myecho.c */
#include <stdio.h>
#include <stdlib.h>
int i;
int main(int argc, char *argv[]){
    if(argc > 1){
        for(i=1; i<argc; i++){
            printf("%s ",argv[i]);
        }
        printf("\n");
    }
    exit(0);
}
```

The result is shown below.

```
% ./myecho uni ebi ikura
uni ebi ikura
% ./myecho
%
```

`argc` is of type `int`, and holds the number of arguments, which includes the command itself. `char *argv[]`; is the array of pointers to strings. To point to specific arguments, you can use `argv[0] ! $argv[1] ! $argv[2]` and so on. `argv[0]` holds the command name itself. This allows you to modify the action based on the command name by using `argv[0]`. The following example shows this technique.

```
/* morning.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[]){
    if( strcmp(argv[0],"morning") == 0 ){
        printf("Morning command.\n");}
    if( strcmp(argv[0],"ohayou") == 0 ){
        printf("Ohayou command.\n");}
    exit(0);
}
```

The result is shown below.

```
% a.out
% mv a.out morning
% morning
Morning command.
% mv morning ohayou
% ohayou
Ohayou command.
% ln -s ohayou morning
% morning
Morning command.
%
```

This technique is used in real systems. For example, the file compression and decompression programs, called `compress` and `uncompress` are actually the same program.

```
% cd /usr/bin
% ls -lo compress uncompress
-r-xr-xr-x  2 root  52352  7 14 2009  compress
-r-xr-xr-x  2 root  52352  7 14 2009  uncompress
% cmp compress uncompress
%
```

(The `cmp` utility checks whether or not two files are identical. If the two files are same, no message is displayed.)

### 6.7.1 Command line options

You have seen us use commands like `gcc -o filename`. The `-o` is called a *command line option*. You may of course write your own code to parse these arguments, but the work has already been done for you, twice: the library function `getopt()` parses single-letter options like `-o filename` (with or without an argument such as a filename), or `getopt_long()` supports longer option names such as `--output=filename`. The latter is more powerful, but more complicated to use.

⇒ *Examples to be written.*

## 6.8 Environment variables

To refer to environment variables such as `HOME` and `DISPLAY` in the program, use the `getenv()` function.

From the command line, environment variables can be displayed using the `printenv` command. They can be used to tell the program about its execution environment. The following example displays the environment variable `HOME`, which is the location of the user's home directory. Being able to find the user's home directory is valuable in many programs, for example because configuration files for some programs are stored there.

```

/* myprintenv.c */
#include <stdio.h>
#include <stdlib.h>
int main(){
    char *e;
    if ((e=getenv("HOME"))!=NULL){
        printf("%s\n",e);
    }
    exit(0);
}

```

```

% ./myprintenv
/home/kusumoto
%

```

The next program displays all environment variables in the array `*envp[]`. In *some* UNIX environments, `main()` may be defined to have a third argument, which points to the environment variables. This should be considered not portable code; it may not compile on some systems.

```

/* myprintenv2.c */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[],
        char *envp[]){
    int i;
    i = 0;
    while(envp[i]!=NULL){
        printf("%s\n",envp[i]);
        i++;
    }
    exit(0);
}

```

## 6.9 Tools

### 6.9.1 Make: building programs from more than one file

So far, all of our programs will fit comfortably into a single file; most are no longer than a single page. However, nearly all complex programs involve source code that is in more than one file. Let's take as an example a program that calculates the dot product of two vectors (as in Exercise 3.6 – yes, this is giving you a possible solution to that exercise!).

The portions of the program that actually calculate basic functions of vectors, such as the dot product, will be useful for many programs, not just this one



example. Moreover, there will eventually be a bunch of such functions, and our main program will get longer, too, so it makes sense to split things up into multiple files. Besides making the code easier to read, maintain, and share, this allows `make` to rebuild only the portions of the program that have changed during your edit-compile-test cycle; for most modest-sized programs this won't seem like a big deal on a modern machine, but it is a great help when building something as complex as the Linux kernel, for example.

Let's create a second source file, `veclib.c`. In order to use the functions in the file from our main program `dotproduct.c`, we will need the function prototypes, so we will create an accompanying header file, `veclib.h`. At the moment, `veclib.h` is very short:

```
/* veclib.h: prototypes for veclib.c */
double dotproduct(double a[], double b[], int len);
```

`veclib.c` contains the actual code for the function:

```
/* veclib.c: library of routines for simple vectors of doubles */
#include "veclib.h"

/*
 * dotproduct: calculate the dot product of two vectors
 * inputs: two pointers to vectors, length
 * (vectors are assumed to be the same length)
 * output: dot product (sum of element-wise products)
 * side effects: none
 * exceptions: only overflow should occur unless input
 * vectors contain NaN; unhandled here
 * complexity: O(len)
 */
double dotproduct(double a[], double b[], int len)
{
    float dotprod = 0.0;
    int i;

    for(i=0 ; i < len ; i++){
        dotprod+=a[i]*b[i];
    }
    return dotprod;
}
```

Note that our main program file, `dotproduct.c`, contains the *only* instance of the function `main()`:

```
/*
 * print the dot product of two fixed-size vectors
 * Author: Rod Van Meter, 2015/6/8
 */

#include <stdio.h>
#include <stdlib.h>
#include "veclib.h"

#define MAXNUM 100000000

/* dummy initialization routine -- sets each element to its
 * own index.
 * Inputs: pointer to array of ints, length (#elements)
 * Outputs: (void)
 * Side effects: modifies input array in place
 * Complexity(execution time): O(len)
 */
void InitializeArray(double a[], int len){
    int i;
    for(i=0;i<len;i++){
        a[i]=i;    /* implicit conversion */
    }
}

/* defined as globals here because vectors this large won't
 * fit on the stack, so we can't make them locals inside main().
 * To get around this and keep the variables local, you would
 * need to malloc() memory for them.
 */
double VectorA[MAXNUM];
double VectorB[MAXNUM];

int main(){
    float dotprod;

    InitializeArray(VectorA,MAXNUM);
    InitializeArray(VectorB,MAXNUM);
    dotprod = dotproduct(VectorA,VectorB,MAXNUM);
    printf("inner product=%f\n",dotprod);
    exit(0);
}
```

Of course, our Makefile must be expanded as well:

```

CFLAGS=-g
# or CC=clang
CC=gcc-5

all: dotproduct

dotproduct: dotproduct.o veclib.o
    $(CC) -o dotproduct $(CFLAGS) dotproduct.o veclib.o

dotproduct.o: dotproduct.c veclib.h
    $(CC) -c $(CFLAGS) dotproduct.c

veclib.o: veclib.c veclib.h
    $(CC) -c $(CFLAGS) veclib.c

clean:
    rm veclib.o dotproduct.o dotproduct

```

## 6.10 Exercises


- 6.1** Write a program that takes two command line arguments, the number of heads and legs, then calculates Tsuru-Kame san as in Exercise 4.7. This is the same as the prior exercise, except for the command line arguments and that error checks have to be implemented.

```

% ./tsuru 10 24
Tsuru = 8
Kame = 2
%

```

- 6.2** Write a program that checks whether the file specified as a command line argument exists or not. For example, you may call `open()` and check the return value. A better way is to use the `stat()` system call. *Note that even if you check once on the existence of a file, your `open()` call may still fail, for a variety of reasons, including that the file was deleted between the time you checked and the time your `open` call actually completes!*
- 6.3** Write a program that takes a filename as a command line argument, then outputs the content of that file to the standard output. Reading from the specified file and writing to the standard output are required. (Hint: without knowing the file size, you have to read some amount of the file, and output, then read again.)
- 6.4** Extend your program from the last exercise to perform some of the functions of the `cat` command. For example, `cat` has the following functions:

- concatenate function like `cat file1 file2 ...`,
  - Without an argument, it reads from the standard input.
- 6.5** Write a program that takes two file names as command line arguments, then tells us which file is larger than the other. The `stat()` function can be used. You may simply read an entire file to check the size of the file, although that would be very wasteful and potentially slow if the files are large.
- 6.6** Write a program that creates a file that contains 10,000 random integer numbers. Random numbers can be obtained by using the `random()` function.
- 6.7**  Write a program (call it `narabikae.c`) that can read a file one line at a time, sort those lines into alphabetical order using the library function `qsort()`, and print them out to a new file. You will have to use a *function pointer* for the comparison function in order to use any of the C library sort routines. A complete example of function pointers and use of `qsort()` are in the Appendix A.7.1; you may extend this program.

<code>printf()</code>	send formatted output to the standard output. Basically, line oriented.
<code>fprintf()</code>	Same as <code>printf()</code> , but specifying the output location.
<code>sprintf()</code>	Same as <code>printf()</code> , but output is a character array. <b>Be careful of overflow on the output array</b>
<code>snprintf()</code>	Same as <code>sprintf()</code> , but specifying the size of output array. This interface is better.
<code>scanf()</code>	formatted input from the standard input. Basically line oriented. <b>(Avoid use of this interface because of the possibility of buffer overflow)</b>
<code>fscanf()</code>	Same as <code>scanf()</code> , but specifying the input location
<code>sscanf()</code>	Same as <code>scanf()</code> , but specifying a character array for input.
<code>fgets()</code>	Read one line specifying the input location. Doesn't discard the new line character. Use this function.
<code>fputc()</code>	Output one character specifying the output location.
<code>putc()</code>	Same as <code>fputc()</code> , but usually, defined as a macro.
<code>putchar()</code>	Output one character to the standard output. Same as <code>putc(c, stdout)</code>
<code>fgetc()</code>	Read one character, specifying the input location
<code>getc()</code>	Same as <code>fgetc()</code> , but usually, defined as a macro.
<code>getchar()</code>	Read one character from the standard input. Same as <code>getc(stdin)</code>
<code>fread()</code>	Read a specified number of fixed-size objects (or records) from the specified input location.
<code>fwrite()</code>	Write a specified number of fixed-size objects (or records) to the specified output location.
<code>read()</code>	Read a specified number of bytes from the specified input location.
<code>write()</code>	Write a specified number of bytes to the specified output location.
<code>mmap()</code>	Make part of a file available as if it is memory; modifying the memory modifies the file.

Table 6.1: Standard I/O library functions for reading and writing data (above the double line) and UNIX system calls for reading and writing data (below the double line). The first group of library functions is the line-oriented interface, the second group is the character-oriented interface, and the third group is the block-oriented interface. Several of these functions have additional variants with slightly different arguments.



# First Interlude

If you can use the concepts and C constructs introduced in the first six chapters (along with the supplemental information in the appendices), you should be able to read a file, split it into records (at least in the simple case of one record per line), sort the records using a library routine, and output to a new file, as in Exercise 6.7. This is perhaps the most basic level at which you can say you are familiar with a programming language. In C, this requires a minimal facility with pointers and `malloc()/free()`.

Beginning in the next chapter and extending through the rest of Part I, we will work toward a more complete mastery of the language, and develop some of the central ideas in computer science along the way. We will focus on more involved uses of pointers to organize data in data structures that can be searched, sorted, and updated efficiently. We will also see some additional basic algorithms, and get a first look at how those algorithms behave as the size of the problem they are asked to solve grows.





## Chapter 7

# Linked Lists and Recursion

In which we meet both singly- and doubly-linked lists and dynamic 2-D arrays for the first time, and discover the profound concept of recursion.

## 7.1 Concepts

### 7.1.1 Linked lists

Figure 7.1 shows a simple *linked list*. In a linked list, each *element* of the list consists of one or more data values, and a way to find the next element in the list, shown by the arrows in the figure. In C, these are implemented using pointers. A list is inherently ordered and its first and last elements are called the *head* and the *tail* (which are the same if the list is only one element long).

To understand the utility of linked lists, let's compare to arrays, which were our first means of organizing groups of data of the same type. Back in Figure 3.1 on page 38, we saw an array of integers. Arrays are simple and fast; you can access the middle or last elements of the array as easily as the first. However, they have two major drawbacks: you must know how big your array will be at the time you create it, and modifying the set of elements in the array is tedious. If we want to add a new element at the beginning of the array, we must first move all of the existing elements to spots later in the array. With a linked list, however, adding elements is straightforward; we will see this in the exercises.

**Many languages support what they call “arrays”, but are dynamically sized and support insertion and deletion well. Underneath, these arrays are probably implemented using a more complex data structure, rather than the single, contiguous memory block that is a C array. Linked lists are one**

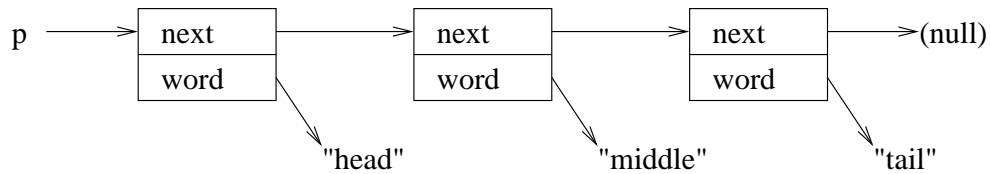


Figure 7.1: A three-element, singly-linked list of `struct wordlist`. The variable `p` is a `struct wordlist *`, a pointer to the structure.

**possibility, but more likely trees or hashes are used, which we will see in Chapter 8.**

We actually caught a glimpse of linked lists back in Exercise 3.8; if you have not done that exercise, this is a good chance to do so.

### 7.1.2 Recursion

*Recursion* is one of the most fundamental programming techniques. Elements of mathematical sequences are often expressed in terms of prior elements in the sequence, e.g.

$$\begin{cases} a_0 = 1 \\ a_n = 2 \times a_{n-1} \end{cases} \quad (7.1)$$

is one way to write a definition for the powers of two.

Recursion is possible in all modern programming languages, but is especially prominent in Lisp. Interestingly, some of the earliest digital computers did not properly support a stack, and therefore recursion was difficult.

## 7.2 Self-referential structures

One important structure usage of pointers is to create self-referential structures. Such a structure includes a pointer to a variable of the same type as the structure itself. For example,

```
struct wordlist {
    struct wordlist *next;
    char *name;
} *a;
```

This example defines the structure `wordlist`, and a variable `*a` that is a pointer to a `wordlist` structure. The structure consists of a pointer to a string,

name, and a pointer to a `wordlist` structure itself, `next`. This can be used to implement the *linked list* structure that is one of the most important concepts in programming.

```
/* list1.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct wordlist {
    struct wordlist *next;
    char *word;
};

int main(){
    // we will use p as a temporary
    // variable and q to hold our list
    struct wordlist *p, *q;

    // build the last one first
    q = malloc(sizeof(struct wordlist));
    q->next = NULL;
    q->word = malloc(sizeof(char)*4);
    strcpy(q->word, "END");
    // build the middle one
    p = malloc(sizeof(struct wordlist));
    p->word = malloc(sizeof(char)*6);
    strcpy(p->word, "...middle...");
    // and link it onto the front
    p->next = q;
    q = p;
    // now build the first one
    p = malloc(sizeof(struct wordlist));
    p->word = malloc(sizeof(char)*6);
    strcpy(p->word, "START");
    // and link it on the front
    p->next = q;
    q = p;

    // now let's walk the list
    // and print out the words
    for ( p = q ; p ; p = p->next )
        printf("%s ", p->word);
    printf("\n");
    exit(0);
}
```

This program implements the data structure in Figure 7.1. We first build the linked list by hand. Like a stack, a singly-linked list has the last item that was

added in the first position; it is *last in, first out*, or *LIFO*. After building the list, we “walk” it using the common idiom of `for ( p = q ; p ; p = p->next )`.

We set the size of memory storage area to be allocated by `malloc()` to be larger than the length of the strings by 1, because we must also include space for the terminating character `'\0'`.

⇒ *More on functions to add, search and remove from lists to be added.*

**This kind of manipulation of pointers to organize and reorganize data is one of the central elements of programming in C. You *must* master it! We will see more complex uses of pointers in complex data structures in Chapter 8.**

### 7.3 2-D Arrays

⇒ *To be written.*

### 7.4 Recursion

It’s easy to write a program that calculates a sequence such as the values in Equation 7.1 going up. But what if you want to write a program that calculates  $a_n$  directly? That’s more trouble. Instead, a simpler method is write a function that takes  $n$  as an argument, and calls *itself* with the argument  $n - 1$ . Of course, that call will then call itself with the argument  $n - 2$  and so on, until it gets down to some value it can calculate directly (in this case, the value for  $a_0$  is known).

Such a function call is called a **recursive call**. For example, you may have learned about factorials when studying probabilities and permutations. Remember the definition of the factorial:

$$\begin{cases} 0! = 1 \\ n! = n \times (n - 1)! \end{cases}$$

10! is calculated as follows:

$$\begin{aligned} 10! &= 10 \times 9! \\ &= 10 \times 9 \times 8! \\ &= 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times 1 \\ &= 3628800 \end{aligned}$$

10! is the product of 1 to 10. Using a `for` statement, a function to calculate the factorial can be written as follows:

```

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

```

Using a recursive call, the function can be written as follows:

```

int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}

```

This function corresponds well to the mathematical definition of the factorial. In a recursive call, the function itself is called within the function, but the arguments and local variables for the called function are stored in different locations on the stack from those of the caller function.

## 7.5 Tools

### 7.5.1 Generating Dependencies in Make

## 7.6 Exercises

- 7.1** Modify the program `list1.c` to print out the addresses of the `wordlist` structs and the `word` strings, and draw a memory map that accurately represents the layout. Include arrows that indicate the variables to which the pointers refer.
- 7.2** Using the `factorial()` example function, write a program that displays the factorial of 1 to 20.
- 7.3** The Fibonacci sequence is defined as follows.

$$\begin{cases} \text{fib}(0) = \text{fib}(1) = 1 \\ \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) & (n \geq 2) \end{cases}$$

Some initial numbers of the Fibonacci sequence are shown below.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, . . .

Write a program that calculates the numbers `fib(0)` to `fib(40)` with recursive calls. The function `fib()` calculates the Fibonacci sequence `fib(n)`, and is defined in the program. Be aware of the range that each type can represent.

- 7.4** Write a program that calculates the Fibonacci sequence to the 100th element with a `for` statement, not using recursive calls.
- 7.5** The factorial of a number  $n$  is represented by  $n!$ . There is also the notation  $n!!$ . It is defined as follows.

$$\begin{cases} 0!! = (-1)!! = 1 \\ n!! = n \times (n - 2)!! \end{cases}$$

Write a program that calculates  $1!!$  to  $15!!$ .

- 7.6** Repeat exercise 3.7 for a recursive function, such as factorial. The set of things stored on the stack for a function is called the *stack frame*. What is the minimum size for a stack frame? Is it zero when there are no local variables? If not, do you have any idea why not?
- 7.7** The example program `list1.c` does not properly free the memory it allocates before it exits. Modify it so that it does.
- 7.8** Add a new function to the program `list1.c`,

```
struct wordlist
*addfront(struct wordlist *p, char *newword)
```

that will add a word to the front of your linked list. It should call `malloc()` twice, once for the `wordlist` struct, and once to make a copy of `newword`. Replace all of your work in creating the list with calls to this new function.

- 7.9** Add a new function to the program `list1.c`,

```
struct wordlist
*addback(struct wordlist *p, char *newword)
```

that will add a word to the *end* of your linked list. It should call `malloc()` twice, once for the `wordlist` struct, and once to make a copy of `newword`. In order to do this, you will have to walk the list to the last entry, and modify that last structure's `next` pointer in place.

- 7.10** Take the program `list1.c` and add a function that will remove a word from the list. It should work properly even if the word found is in the middle of the list. Make it something like

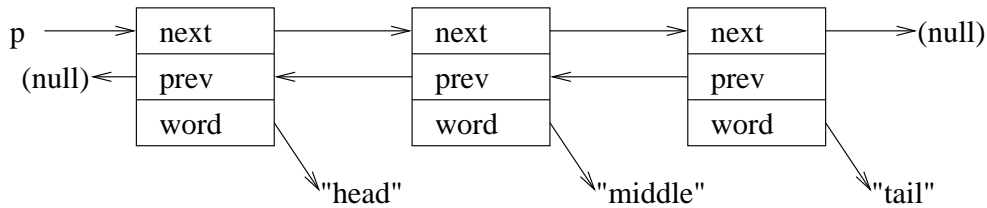


Figure 7.2: A three-element, doubly-linked list of `struct wordlist`. The variable `p` is a `struct wordlist *`, a pointer to the structure.

```

struct wordlist
*removeword(struct wordlist *p, char *word)
  
```

and use `strcmp()` to see if the word is the one you want to remove. The function should return a pointer to the first element in the remaining list if there are still words in the list, or `NULL` if they have all been removed. (*Why?*) It should return `-1` if the word is not found. Be sure to use `free()` to release the memory in your function.

- 7.11** Take the program `list1.c` and modify the structure to be a *doubly-linked list*, as in Figure 7.2. What is the value of such a list compared to a singly-linked list?
- 7.12** Write a program that registers words in an internal dictionary with structures using pointers. In a loop with `while` or `for`, read a word input by the user using `fgets()`. When `Control-D` is input, the program should exit from the loop, then display all registered words. (When `Control-D` is input, `fgets()` returns a `NULL` pointer. This way, the program can detect the end of input.) The program doesn't know the number of words that will be registered in advance, so it needs to allocate the memory dynamically.
- 7.13** Extend your program from Exercise 7.12. Write a program that registers pairs of keywords and their meanings (content) with structures using pointers. The program needs to allocate the memory for both keywords and contents.
- 7.14** ☞ Extend your program from Exercise 7.13 to always insert words *in alphabetical order*. To do this, you will have to walk the list until you find a word that is greater than the word you are adding, then modify the pointer in the word *prior* to that one in order to insert the new word, as shown in Figure 7.3. Is this easier with a singly-linked list, or a doubly-linked one?
- 7.15** If you take a linked list and make the tail of the list point back to the start

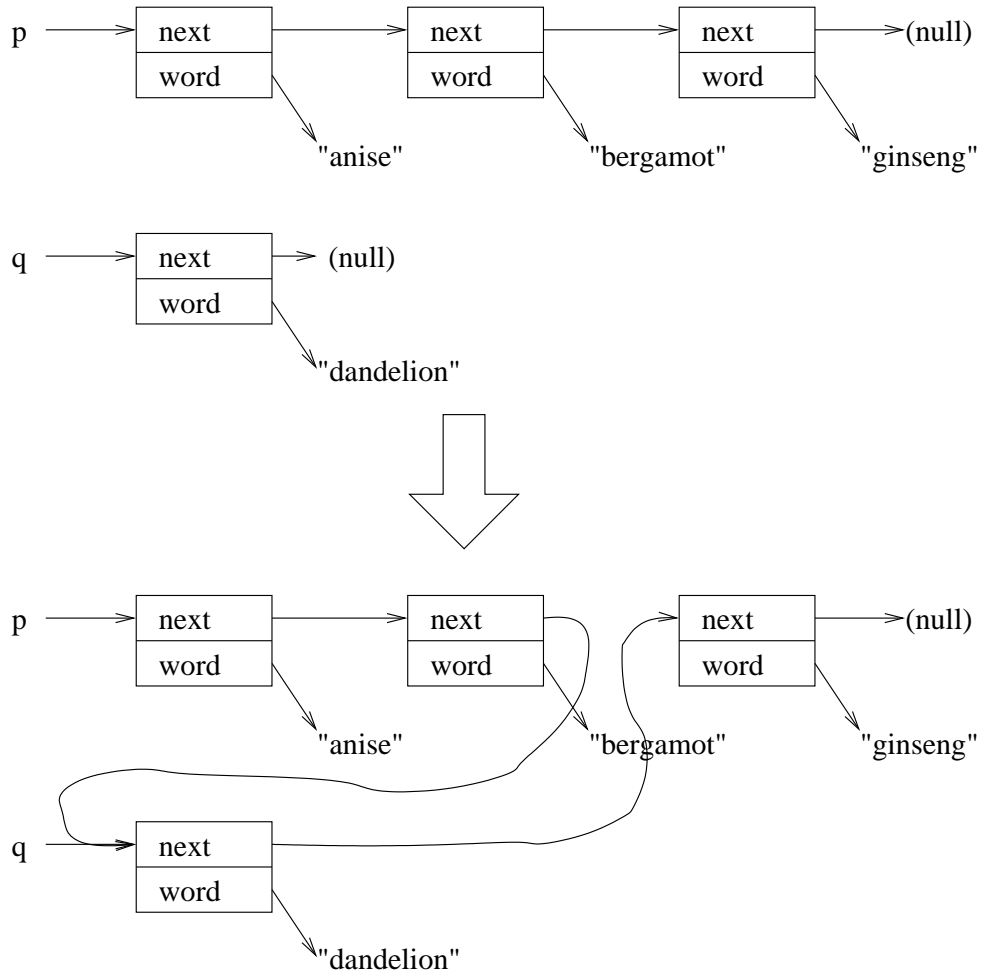


Figure 7.3: Inserting a new word into an ordered linked list. Above, the initial list *p* and new entry *q*; below, with "dandelion" inserted between "bergamot" and "ginseng".



of the list, rather than contain a null pointer, you have created a type of *ring* or *ring buffer*. Rings are useful when you want to dedicate a fixed amount of space to buffering data; for example, it may only be useful to keep the last ten error messages. Pick one of the linked list exercises and reimplement using a ring. How does your `for` loop have to change?

**7.16** ♾ In Japan, there are six face values of coins, 1, 5, 10, 50, 100, and 500. When you pay 10 YEN with coins, there are four ways of payment as follows.

- 10
- $5 \times 2$
- 5 and  $1 \times 5$
- $1 \times 10$

Write a program that counts how many ways there are to pay  $n$  YEN in coins.

To solve this exercise, define a recursive function `change(int n, int k)`. This function calculates how many ways you can pay  $n$  YEN with coins that are  $k$  YEN or smaller in face value.

- `change( $n$ , 1)` returns 1, because only 1 YEN coin is used.
- `change( $n$ , 5)` returns `change( $n$ , 1) + change( $n - 5$ , 5)`
- `change( $n$ , 10)` returns `change( $n$ , 5) + change( $n - 10$ , 10)`
- and so on



## Chapter 8

# Algorithms, Intermediate Data Structures, and Analysis

In which we learn to sort items, meet a basic tree and hash, and learn the Big-O notation so we can talk about how much work it is to do things.

### 8.1 Concepts

Now that we have a certain amount of basic training under our belt, let us briefly summarize what we know and put it from a simple programming mindset into a more theoretical computer science context. By analogy with mathematics, now that you know how to add, subtract, multiply and divide, we can begin to talk about algebra and some of the theory underlying math.

This chapter and the remainder of Part I of this book, then, should help crystallize *concepts* in your mind as well as continue the basic work of “muscle building” in learning C coding, preparing you to *use* what we have learned, *discuss* algorithms and data structures at a more mature, abstract level, and *study* more advanced material on your own, in your *kenkyuukai* (laboratory), or in class.

#### 8.1.1 Big-O notation: How long will my computation take?

⇒ Go back and look at *factorial()* function and compare number of calls using for loop or recursion. Be sure to mention that this notation is actually tied to a formal model of the computer itself.

Whether you intend to become a computer scientist, or a working programmer, you will need to know about the  $O(\cdot)$ , or *big-O*, notation, the vocabulary we use when discussing the behavior of algorithms. Even if you do not intend to become a computer professional, this notion of complexity is a fundamental and beautiful idea, and will lurk deep in your thinking as you solve problems in any field.

Now that you have been writing programs for a while, you have probably noticed that some programs are fast and some are slow. Some computers are also fast, while others are slow, even when running the same program. How can we characterize “fast” and “slow”? What do we mean when we use these terms?

If you are observant, you might have realized that the amount of time depends on the amount of data you are processing: doing something on a gigabyte of data naturally takes longer than doing the same operation on a megabyte. Up to this point, however, you probably have not had the opportunity to see *how* that “processing time” varies with the amount of data. In fact, for some purposes, processing ten times as much data takes ten times as long, while in other cases it may take far longer – a hundred times, a thousand times, or even become effectively impossibly long.

Theoretical computer scientists find this behavior fascinating in its own right, and it forms the foundation of their entire field, under the name *computational complexity*. As programmers and engineers, our immediate concerns are more practical: will my program be fast enough for my users, on the machines they have available, for the size of problem (or size of data set) they will need to use it on?

As a simple example, consider an operation such as the dot product of two vectors, which you wrote in Exercise 3.6 and we revisited on page 130. Our variable `len` is the number of elements in each vector. Here, let’s call it  $n$ , to make our notation look a little more mathematical. For each of the  $n$  elements in the vector, we had to perform a small, fixed number of operations: read the elements from the two vectors, multiply them, and add that result into our running total. Thus, we say,

Calculating the dot product of two vectors is  $O(n)$  (read “Oh-of-n”).

To be more exact, we should say this about a particular function  $f(n)$  that represents the running time of our program, so we really should say, “My program’s running time is  $f(n)$ , and  $f(n) = O(n)$ .”

More correctly,  $f(n) = O(n)$  says, “*Eventually*, as  $n$  becomes large, further growth in  $n$  results in nothing worse than linear growth in the total amount of computation.” If we make  $n$  ten percent bigger, execution time grows ten percent; if we make  $n$  twice as big, execution time doubles.

In contrast to the usual use of  $=$ , here we are describing an asymmetric relationship between the left and right sides of the statement.

## 8.1.2 Sorting

### 8.1.3 Basic Data Structures

Almost immediately upon beginning to program, you met the concept of individual variables, each holding a single element of data. However, individual data elements are the exception rather than the rule; most often, we have lots of data we want to use.

One of the most common functions of a computer is to organize raw data into a form that allows it to be searched or displayed easily. So far, we have seen several basic *data structures* for holding groups of data elements:

- **Arrays** are useful for data that we receive in a known order (e.g., we know they were stored in order in a file). Arrays generally work best when we can determine the number of elements we expect before we start storing the data into the array, so that we can allocate enough memory at the beginning. A one-dimensional array can be useful for e.g. a time sequence of events, such as the daily temperature or your bank balance. In C, character strings are also a form of one-dimensional array, with the unique characteristic that they are *null-terminated*. Two-dimensional or three-dimensional arrays can be used to hold many types of numeric data corresponding to real-world data samples or computer simulations, as well as many other uses.
- **Stacks** allow us to set aside the current chunk of work, deferring it in favor of a more urgent task or a subtask necessary to complete our current task. Stacks are used implicitly by the compiler to hold temporary data as we move through function calls, and are critical for implementation of *recursion*.
- **Linked lists**, both singly-linked and doubly-linked, were our first significant use of pointers to connect data structures together. They have the advantage of being dynamic and not having to worry about the number of elements before you begin. *Editing* a linked list to add or remove elements is straightforward, requiring only changing a few pointers. However, *searching* a linked list to find a particular element becomes tedious when the number of elements grows.

Generally, we want to organize the data to make it easier to find an item or display the set of items in some order that makes sense. An array is constructed assuming that data can easily be *indexed* using an integer; that is, the *search key*

for retrieving the  $n$ th element of an array is just the number  $n$ . Linked lists do not inherently require any given order, but an order is often imposed as the list is constructed, and the structure is geared to iterating on the `next` pointer, which is very efficient. Stacks do not generally offer a method for searching the entire stack for specific elements.

To extend our set of capabilities, we would like to be able to organize data so that:

- we do not have to know before we begin how many data elements we will have;
- it is easy to use a variety of types of search key, such as family name or given name, student ID number, or any application-specific key; and
- inserting or deleting elements is efficient, and has minimal effect on the efficiency of further lookup operations.

With the addition of two more basic data structures, we will have a reasonable set of mental tools for organizing data:

- **Trees** come in many forms useful for many purposes, from basic *binary trees* for simple indexing of data, to *Patricia tries* for looking up forwarding information for IP packets, to *B-trees* and *B+-trees* for efficient storage of data on disk. In this chapter we will meet binary trees and *red-black trees*, which provide more robust performance guarantees against unfavorable data patterns than the basic binary trees.
- **Hash tables** store data entries (or pointers to data entries) in a form of array, or table, that makes it relatively easy to store and find a given item without knowing before the table is created what the distribution of search keys will be like. Simple hash tables are used for managing caches in some systems, and fact the hardware memory cache in a microprocessor is a form of hash table. The current edition of this book does not cover hash tables, but you are *strongly* encouraged to seek out more information on them to help round out your basic knowledge.

In some cases, we might even like to allow for inexact matches, such as wildcards in strings or *longest prefix match* for IP addresses, or some notion of physically “nearby” for geographic databases using latitude and longitude or postal code. We will not address these variants in this book, but the foundation you will develop in this chapter will prepare you to understand those topics as you encounter them.

### 8.1.4 Basic Binary Search Trees

#### Basic Structure

In a *binary tree*, or *binary search tree*, we store data elements in *nodes*. Each node contains a *key* that identifies the data element, the data element itself (or a pointer to the element), and pointers to two *children* of the current node. It must be possible to compare two keys, and for our purposes here we will assume that keys are unique and a set of keys has a single, unique order. An example of a tree is shown in Figure 8.1, with each node in the tree marked with its key. Here, our keys are integers, though they may be any data type that meets the criteria we just described; alphanumeric strings are commonly used as keys.

The first node you encounter in a tree is called the *root*. Each node in a binary tree may have zero, one, or two children. (Other types of trees may allow more children, or impose the restriction that a node never has exactly one child.) Nodes with no children are called *leaf* nodes. Note that, in contrast to real trees, the root is at the top and the leaves are at the bottom in most pictures of binary trees! I suppose you can imagine that you are looking at the reflection of a tree in a pond, or drawing a tree as it exists on the opposite side of the world.

Edges in tree may be drawn as arrows or line segments, but “descending” the tree is always possible; depending on the implementation, a node might have a pointer to its *parent* (the node above it, in a typical figure), or your code may be required to track parents as necessary. A basic data structure for a tree might look like this:

```
struct node {
int key;
void *element;
struct node *left;
struct node *right;
};
```

In this example, `element` is a pointer to the data element corresponding to the node. The data element could instead appear directly in the node, as we have done with linked lists. The `key` may be a separate data field, as shown here, or may be part of (or calculated from) the `element`.

#### Lookup in a Tree

#### Inserting into a Tree

### 8.1.5 Red-Black Binary Trees

*This is probably the most difficult material in the first half of the book, but if you grasp this you are well on your way to becoming a solid programmer. A project*

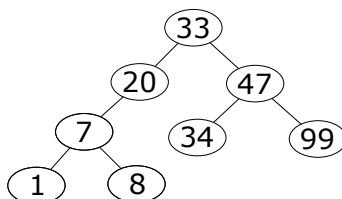


Figure 8.1: Example of a binary search tree with integer keys. Every node has two child pointers; in nodes with zero or one children, the other pointers are null.

*on red-black trees is an excellent final project for this class.*

The basic binary trees we have just covered are a wonderful invention: they are flexible, fast and simple. *Red-black trees* are substantially more complex than the most basic binary trees. Why would we bother creating something more complex, when it seems like basic trees do what we want?

The answer lies in Figure 8.2. The binary tree structure is beautiful, but very general. In the best case, every leaf node is  $O(\log n)$  depth from the root. However, a binary tree can become very *unbalanced*. In the worst case, it becomes equivalent to a simple linked list, as in the figure. In this case, finding a particular element can take  $O(n)$  time steps, rather than the  $O(\log n)$  we were aiming for when we created a binary tree.

To combat this imbalance and ensure reasonable performance even as a tree “ages” (goes through a number of inserts and deletes), computer scientists have come up with various forms of *self-balancing trees*. One such tree is a *red-black tree*, which stays *approximately* balanced, and requires only relatively inexpensive (if somewhat tricky) operations to maintain that balance.

A red-black tree has the following properties:

1. Every node is either red or black.
2. The root and leaves (which are *null* pointers or *nil* elements) are black.
3. Every red node has a black parent.
4. Every path from a node to all its descendants includes the same number of black nodes.

These properties can be formulated in several different equivalent ways; this version is due to Erik Demaine of MIT.

An example of a red-black tree is shown in Figure 8.3.



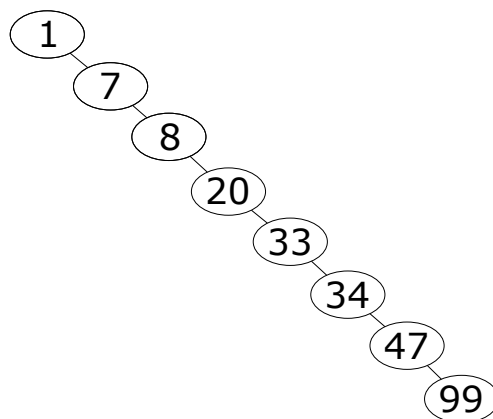


Figure 8.2: Example of an unbalanced binary search tree. This can occur when reading in a file that is already stored in order and performing a simple insert for each record.

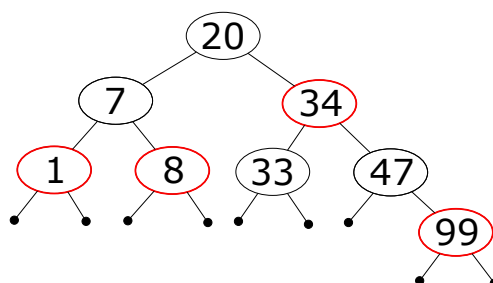


Figure 8.3: Example of a red-black tree. The small black dots are the nil leaves.

The red-black page on English Wikipedia has a few shortcomings, the biggest of which are: 1. I'm still not sure case 5 is correct (though mine seems to work using that code). 2. There is no implementation of the rotate operations!

I implemented the rotate operations, but it took me a couple of hours to get it right, and it involved some changes to the data structure. I will share the rotate routines with you. Note that the code may work directly with your implementation, and may not!

Besides the Wikipedia page, here are a few resources:

\* Explanation of the use of red-black trees in the Linux kernel: <https://www.kernel.org/doc/Documentation/http://lxr.free-electrons.com/source/Documentation/scheduler/sched-design-CFS.txt>

\* Online, browsable Linux kernel source, rbtrees.h: <http://lxr.free-electrons.com/source/include/linux/rbtrees.h>

\* corresponding C source: <http://lxr.free-electrons.com/source/lib/rbtree.c> \* Examples of kernel code that uses r-b trees: <http://lxr.free-electrons.com/source/drivers/base/regmap/regcache-rbtree.c> <http://lxr.free-electrons.com/source/kernel/sched/fair.c>

And a reference to one of the best books on algorithms in existence: <https://mitpress.mit.edu/books/introduction-to-algorithms> My copy is 2nd edition, the 3rd edition is current. In both, Chapter 13 is dedicated to red-black trees. The book is probably available in the Media Center. If not, and if you want to see (and photocopy) the explanation from my copy of the book, please email me or stop by Delta 211N.

Video of Erik Demaine lecturing on red-black trees: <http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-10-red-black-trees-rotations-insertions-deletions/>

<https://www.cs.usfca.edu/galles/visualization/RedBlack.html>

Good sequence for showing insert: 1000, 1100, 500, 700, 600 Shows 600 getting inserted as RL from its grandparent 500, then propagating up to take its place.

## 8.2 C

⇒ *To be written.*

## 8.3 Tools

⇒ *To be written.*

### 8.3.1 gprof: profiling execution

Sometimes your program is a lot slower than you think it ought to be, and it isn't obvious why. An *execution profiler* can help. For many years, the standard UNIX profiler has been **gprof**, though it has some limitations (especially with respect to parallelism, multithreading and locking) and does not work smoothly

with the Clang compiler or on Macintoshes. The demonstrations here assume you are working on a FreeBSD, Linux or Solaris machine.

## 8.4 Exercises

⇒ *To be written.*

- 8.1 Write your own program to sort a file of integers using the simple bubblesort algorithm. How many comparison and swap operations must be performed, as a function of the number of integers in the file,  $n$ ?
- 8.2 Implement a binary search on your sorted data.
- 8.3 Write a new sort program that implements the “mergesort” algorithm for a file of integers.
- 8.4 Compare your two sort programs: plot their execution times as you increase the size of the input files. Give an  $O()$  expression for the expected run time of both functions, for best, average, and worst case.
- 8.5 Source code for reading a file of student records and creating a simple binary tree indexed by student ID has been provided. So far, it does little besides read the file. Within the source code, there are a series of problems marked TASK in the comments. Complete these tasks.
- 8.6 Modify your source code from the last problem to reside in multiple source files, and use an appropriate `Makefile`.
- 8.7 The previous program does not balance the tree as the student records are read in. Characterize this: under what conditions will a balanced tree be created, and under what conditions will it be unbalanced? What is the worst-case performance?
- 8.8 Now modify your program to implement a *red-black tree* with balanced inserts. Characterize as in the last problem.
- 8.9 Instrument your code to print a message when the *root* is rotated. Read in a file with consecutive integer keys, starting with 1. What sequence of values does the root take as the file is read? How often is the root rotated? You might also find it enlightening to watch the tree itself grow. Describe the difference between the left half of the tree and the right half of the tree.



## Chapter 9

# Finite State Machines

In which we meet the legendary finite state machine (FSM), and perhaps a friend or two. We also make use of function pointers.

### 9.1 Concepts

*⇒ To be written.*

### 9.2 C

*⇒ To be written.*

### 9.3 Tools

*⇒ To be written.*

### 9.4 Exercises

*⇒ To be written.*



## Chapter 10

# Writing and Testing System Utilities

As a final “build your own light saber” test, we build our own system utility, *including* tests for whether or not it works.

### 10.1 Concepts

The well-known programmer Dan Kegel created an exercise that is excellent practice for disciplined systems programming. If you can do this task comfortably, you are well on your way to being an employable software engineer. This task requires a certain level of maturity in development processes. We will use it as our final exam. Just as a Jedi must build his own light saber, you must build your own systems utility to demonstrate your mastery and readiness to join the world.

#### **Self-check before applying for a software engineering job**

So you’d like to work as a software engineer, and are about to apply for a job. And you’ve already confident you can answer common interview questions. What else can you do to get ready?

Here’s an exercise that might be helpful:

1. Find some simple unix utility, e.g. `cmp`, `expr`, or `od` (these come for free with Linux and MacOSX, and can be gotten on Windows for free by installing Cygwin)

2. Try using them a little; get comfortable with their simplest uses (no fancy options)
3. Write a test for the utility, in your favorite scripting language, that prints PASS or FAIL at the end (and terminates with nonzero status on failure).
  - (a) Start with a very short script (say, 15 lines) that checks just one of the utility's most basic functions
  - (b) Get the test to pass
  - (c) Add another test case to the script and get it to pass again.
  - (d) Repeat a couple times until your test script has a handful of test cases (but is still short; 30 or so lines should do)
4. Finally, implement a simple version of the utility in the language you want to use at the job (Python, C++, Java, whatever), and make it pass the test you wrote above.

Repeat with several utilities (and maybe several languages) until you're comfortable doing it from start to finish in two hours.

If you can do that, you'll probably pass any paper-bag programming test they can throw at you. <http://www.kegel.com/academy/selfcheck.html>

*Acceptance testing* of software programs is an important part of *software quality assurance (SQA)* practice. Take pride in writing such tests and getting your code to pass them; it is a sign of superior craftsmanship and maturity as a developer. One of our fellow faculty members, in a former job, created a suite of several *thousand* tests as part of a database development project. Some of these tests were written by hand, but many of the individual tests were actually created by a separate program.

## 10.2 C

⇒ *To be written.*

## 10.3 Tools

In order to complete this task, you *must* have scripts that can execute your program and examine its output. You may write those tests in any language you wish; it will probably be easiest in a scripting language that handles character strings smoothly, such as Python or Ruby.

Below is an example in Python:



```
import sys
import subprocess as P

def main():
    u"""
    1. store the contents of the source file
    2. copy source_file to target_file
    3. verify whether stored content of the source file
       and the content of the target file are same.
    """
    # check arguments
    argv = sys.argv
    argc = sys.argv.__len__()
    if argc != 3:
        print_usage()
        sys.exit(1)
    source = argv[1]
    target = argv[2]

    # 1. store the contents of the source file
    stored_src = []
    with open(source) as s_file:
        stored_src.append(s_file.read())

    # 2. copy source_file to target_file
    retcode = P.call(["cp", sys.argv[1], sys.argv[2]])
    if retcode != 0:
        print("cp errored!", file=sys.stderr)

    # 3. verify whether contents of the source file
    # and of the target file are same.
    num_line = 1
    error = False
    with open(target) as t_file:
        if stored_src.pop(0) != t_file.read():
            print("line %d is different!" % num_line)
            error = True
            num_line += 1
    if error:
        print("Unfortunately either the source file
        or the target file is corrupted... I'm sorry about that.")
    else:
        print("Congratulations! You successfully copied the file!")

def print_usage():
    u""" print usage. """
    print(u"""usage:
python3 verified_copy.py source_file target_file""")

if __name__ == "__main__":
    main()
```

## 10.4 Exercises

- 10.1 Pick your favorite system utility. Choose carefully – you want one that will a *reasonable* amount of effort to duplicate!
- 10.2 Begin by writing a script in your chosen language that can test the simplest behavior of a utility. It should print **PASS** if the utility passes and **FAIL** if it fails. Your test should not rely on the utility itself to test its correctness. Question: assuming the utility works properly, how can you test the failure case of your tester?
- 10.3 Now write your own version of the utility, and work until it passes your test script.
- 10.4 In all probability, in the above you wrote only the successful case – such as `cp` correctly copying a file that actually exists. Extend your test suite by adding a new test script that tests what happens when you give the utility bad arguments (such as the name of a file that doesn't exist). Of course, the script should print **PASS** when the utility correctly reports an error.
- 10.5 Now extend your own utility until it passes the test for correct error handling.
- 10.6 Continue extending your suite of test scripts and the functionality of your utility until the behavior matches everything described in the utility's manual page, or some other threshold agreed with your instructor.

## Second Interlude

At this point, you should be very comfortable with the C language, several types of data structures using pointers, basic algorithm analysis, how to convert mathematical equations or pseudocode into a program, and a variety of Unix programming tools.

*⇒ Should we add OpenCL? Cuda? MPI? Hadoop? Even a batch system such as the Open Grid Engine?*



## Part II

# Achieving Scale in the Real World: Concurrency, Parallelism, Communication and Distribution in Modern Systems



# Chapter 11

## Soft Real-Time Computing

In which we see very simple use of timers, and write our own processing loop. We also discuss guidelines for safety-critical computing, though this alone will not make you a complete programmer of such systems!

### 11.1 Concepts

⇒ *To be written.*

⇒ *Discuss MISRA.*

### 11.2 C

⇒ *To be written.*

#### 11.2.1 An Arduino multitasker

Recently, I built an automatic feeder to feed the fish in my office aquarium. The feeder was printed using a 3-D printer, and I control it using an Arduino single-board computer with a motor driver and some additional LEDs. I want the fish to be fed once a day, but I also want the LED on the Arduino board to blink, so I can tell at a glance that the system is still running, and I put a nice pattern on the other LEDs at some interval (ten seconds, at the moment).

If you have worked with Arduino, you know that it uses a subset of C as its programming language, and that it has a very rudimentary runtime system. There is no multitasking provided. So, what do you do if you want your Arduino to do multiple things on separate timers? You write your own multitasking

scheduler! Here is a simple, *cooperative multitasking* loop that uses simple timers to establish when the next action will take place.



```

// prototypes for WorkItem functions. All are void.
void BlinkWork(); // just blink the board LED
void RunUpWork(); // do something a little fancier on the LEDs
void DinnerWork(); // feed the fish!

// work items for our simple scheduler
struct WorkItem {
    unsigned long interval; // how often to execute this, in seconds
    unsigned long NextTime; // time after which we will next execute this, in seconds
    void (*fun)(); // pointer to the function that does the work
};

// Actual array of items. Order in this should be irrelevant.
struct WorkItem WorkItems[] = {
    { ONESECOND, ONESECOND, BlinkWork },
    { 10*ONESECOND, 10*ONESECOND, RunUpWork },
    { ONEDAY, 10*ONESECOND, DinnerWork },
    { 0, 0, NULL }
};

void BlinkWork()
{
    // basic blink of the LED on the board
    // ledval is value of the LED, a global defined elsewhere
    if (ledval == HIGH)
        ledval = LOW;
    else
        if (ledval == LOW)
            ledval = HIGH;
    digitalWrite(OnBoardLed, ledval);
}

// our infinite work loop on Arduino automatic fish feeder
// Assumptions: all WorkItems complete, never block
// or go into infinite loop, and are short, so that
// average load is well below one (that is, the Arduino
// is idle most of the time).
// This is soft real time; any task might be delayed
// by maximum of completion time of all tasks on the list
// plus one second. If the total completion time of all
// tasks exceeds the interval of a task, it is possible
// for invocations of that task to be missed.
// Using the system-supplied millis() function, clock
// slip should be minimal.
// Scheduler sleeps for a second before coming back to
// look at list again.
void loop()
{
    int i;
    currentmillis=millis(); // get the current milliseconds from Arduino

    // Loop through WorkItems[] until we find NULL function pointer
    for ( i = 0 ; WorkItems[i].fun != NULL ; i++ ) {
        Serial.print("currentmillis: "); Serial.print(currentmillis);
        Serial.print(" NextTime: "); Serial.println(WorkItems[i].NextTime);
        if ( currentmillis > WorkItems[i].NextTime ) {
            // timer has expired, run our function
            WorkItems[i].fun();
            // set up time for next invocation
            WorkItems[i].NextTime += WorkItems[i].interval;
        }
    }

    delay(ONESECOND); // one second delay on loop
}

```

### 11.3 Tools

⇒ *To be written.*

### 11.4 Exercises

⇒ *To be written.*

- 11.1** Modify your polyrhythm program from Exercise 2.4 to read the rhythms from a file and print out the beats in real time. Use the UNIX `sleep()` function or another method of your choice to get the timing as close to possible right. Your rhythm definition file should include the beats per minute of the rhythm. The format of the file is up to you to decide, but you must write a *document* that describes it.

## Chapter 12

# Processes and Pipelines of Programs

In which we learn how to create, connect and control independent processes running programs of our choice, which is by far the easiest and safest way to do more than one thing at the same time. The *process* is essentially an abstraction of a computer.

### 12.1 Concepts

⇒ *To be written.*

### 12.2 C

⇒ *To be written.*

### 12.3 Tools

⇒ *To be written.*

### 12.4 Exercises

⇒ *To be written.*



## Chapter 13

# Signals: Communicating Asynchronously

In which we learn how to send a signal to another process. We also introduce the concepts of POSIX *safety*.

### 13.1 Concepts

⇒ *To be written.*

### 13.2 C

⇒ *To be written.*

### 13.3 Tools

⇒ *To be written.*

### 13.4 Exercises

⇒ *To be written.*



## Chapter 14

# Threads: Abstracting Processors

In which we learn that the process is an abstraction of a machine, and a thread is an abstraction of the CPU of that machine.

### 14.1 Concepts

*⇒ To be written.*

### 14.2 C

*⇒ To be written.*

### 14.3 Tools

*⇒ To be written.*

### 14.4 Exercises

*⇒ To be written.*





## Chapter 15

# Parallelism: OpenMP

In which we learn how to pay attention to the regularity of the structure of the parallelism in our programs. OpenMP is one of several new systems for expressing parallelism in C.

### 15.1 Concepts

⇒ *To be written.*

### 15.2 C

⇒ *To be written.*

### 15.3 Tools

⇒ *To be written.*

### 15.4 Exercises

⇒ *To be written.*



# Third Interlude

From this point forward, we focus entirely on concepts and programming for IP networks. Sockets, that is.



## Chapter 16

# Communication: IP Networking Basics

In which we get our first glimpse of how computers actually talk to each other.

### 16.1 Concepts

*⇒ To be written.*

### 16.2 C

*⇒ To be written.*

### 16.3 Tools

*⇒ To be written.*

### 16.4 Exercises

*⇒ To be written.*



## Chapter 17

# Naming: the Domain Name Service

In which we learn how to find the computer we want to talk to.

### 17.1 Concepts

⇒ *To be written.*

### 17.2 C

⇒ *To be written.*

### 17.3 Tools

⇒ *To be written.*

### 17.4 Exercises

⇒ *To be written.*





## Chapter 18

# Communication: TCP Sockets 1

In which we get our first complete taste of network programming, which is a fancy way of asking someone else to do your work for you. All examples are *address family independent*, and will work with both IPv4 and IPv6! (We will use what I call *itojun's rules*.)

### 18.1 Concepts

⇒ *To be written.*

### 18.2 C

⇒ *To be written.*

### 18.3 Tools

⇒ *To be written.*

### 18.4 Exercises

⇒ *To be written.*



## Chapter 19

# Distribution: TCP Sockets 2

In which we get our first complete taste of network programming, which is a fancy way of asking someone else to do your work for you.

### 19.1 Concepts

⇒ *To be written.*

### 19.2 Tools

⇒ *To be written.*

### 19.3 Exercises

⇒ *To be written.*

**19.1** Write a client that reliably connects to the attendance server via IPv4 or IPv6.



# Appendix A

## C Syntax Supplement

This appendix includes some deeper exploration of the language and its implementation in the real world. It is presented mostly as reference; you should read it to understand the language better, but for the most part you will not need this information to complete the exercises in this class.

### A.1 Storage Class

A relatively complete understanding of C requires understanding how it manages memory (storage). In this section, you will see some examples that demonstrate use and reuse of the stack memory and other classes of memory.

The following example shows calculation of the factorial using a recursive call.

```
int factorial(int n) {
    if (n == 0) return 1;
    else return n * factorial(n-1);
}
```

In this program, the arguments (and local variables, if any) are different for each instance of the function call. However, as we have noted, C does not initialize variables for you, and memory can be allocated and reallocated to different variables. The next couple of programs demonstrate this behavior.

```
/* localvar1.c */
#include <stdio.h>
#include <stdlib.h>

/* prototypes for forward reference */
void f1(int n);
void f2(int n);

int main()
{
    f2(10);
    f1(1);
    f1(2);
    f1(4);
    exit(0);
}

void f1(int n)
{
    int sum;
    /* note: sum is used without being
       initialized! */
    sum = sum + n;
    printf("sum=%d\n",sum);
}

void f2(int n)
{
    int a, b;
    a = n;
    b = n;
    printf("a,b=%d,%d\n",a,b);
}
```

On ccz??, zmac??, the result is shown below.

```
ccz00% ./localvar1
a,b=10,10
sum=11
sum=13
sum=17
```

There are some unexpected, perhaps strange results. You may have expected the following result:

```
ccz00% ./localvar1
a,b=10,10
sum=1
sum=3
sum=7
```

Thinking about this behavior and trying to isolate where we're going wrong, we may decide to change the argument to the function call `f2()` to 0.

```
/* localvar2.c */
/* everything but main() same as above */
int main()
{
    f2(0);
    f1(1);
    f1(2);
    f1(4);
    exit(0);
}
```

The result is shown below.

```
ccz00% ./localvar2
a,b=0,0
sum=1
sum=3
sum=7
```

Now, the expected result appears. Why is the first case incorrect? It is caused by the handling of local variables in a function. In `f1()`, `int sum;` is defined. However, the value of `sum` is undefined at that time. In a function call in C, local variables and arguments are stored in a newly assigned area at each function call. Then that storage area is released on exiting from the function. This area is part of the *stack*.

In the second example, the storage area that was used for `int sum;` during the previous `f1()` call is reused at the current `f1()` call. So, the previously used values remain **by chance**.

In the first example, temporary storage is used for `int a,b;`. First, 10 is assigned to both variables `a` and `b`. That temporary storage area is released at the completion of `f2()`. At the next call of `f1()`, the same area just released by the completion of `f2()` is reassigned for `f1()`'s temporary storage of `int sum`. So, the uninitialized `sum` has the value 10 **by chance**. Thereafter, our code modifies that and `sum` becomes 11,13,17.

If variables are initialized at the function call, what happens? The next program shows performs variable initialization.

```
/* localvar3.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
    f2(10);
    f1(1);
    f1(2);
    f1(4);
    exit(0);
}

f1(int n){
    int sum=0;
    sum = sum + n;
    printf("sum=%d\n",sum);
}

f2(int n){
    int a, b;
    a = n;
    b = n;
    printf("a,b=%d,%d\n",a,b);
}
```

The result is shown below. `sum` doesn't follow the pattern  $\hat{1},3,7$ ; instead, it does what it ought to with `sum` being a local variable.

```
ccz00% ./localvar3
a,b=10,10
sum=1
sum=2
sum=4
```

In this example, `int sum=0;` initializes `sum` during each function call. Therefore, the previous value doesn't remain.

So what should we do if we want to retain that value for `sum` (which seems likely, just given the name `sum`)? To retain the value of a local variable across multiple calls to the function, use the `static` storage class.



```
/* localvar4.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
    f2(10);
    f1(1);
    f1(2);
    f1(4);
    exit(0);
}

f1(int n){
    static int sum=0;
    sum = sum + n;
    printf("sum=%d\n",sum);
}

f2(int n){
    int a, b;
    a = n;
    b = n;
    printf("a,b=%d,%d\n",a,b);
}
```

The result is shown below.

```
a,b=10,10
sum=1
sum=3
sum=7
```

The storage for `static` variables remains after exiting from a function (or a block). Then, those values are used at the next execution of that function (or block). The initialization is done only once, at the first execution.

**However, when threads are used, you need to pay extra attention while programming. Because static variables are shared among all threads, you need to be careful if you are planning to call a function from multiple threads. Auto variables are safe, because those variables are created and assigned at function call time, and are unique to the thread.**

We went through this series of examples in order for you to understand both how and where local variables are stored, and the kinds of errors that result from failing to initialize variables. However, we could have saved ourselves some time in the beginning by enabling warnings in our compilation:

```

gcc -o localvar1 localvar1.c -Wall
localvar1.c:23:9: warning: variable 'sum'
is uninitialized when used here
      [-Wuninitialized]
    sum = sum + n;
      ~~~
localvar1.c:20:10: note: initialize the variable
'sum' to silence this warning
    int sum;
      ^
      = 0
1 warning generated.

```

Older C compilers may not provide such warnings, but modern ones generally do, and they will save you from many frustrating hours debugging mysterious behavior.

You may have gathered by now that the C compiler makes distinctions between variables depending on their *scope*. There are actually several classes, and under some circumstances we can control which class a variable is in using modifiers on its declaration.

#### **auto**

Called “auto variables”. At execution time, storage is automatically assigned (typically on the stack). These variables are of limited scope, and their storage is reclaimed when they go out of scope. Except for static variables, this class includes all local variables for functions or blocks. This is the only time that C will automatically allocate and recover memory for you; otherwise, variables either exist for the lifetime of the program, or you, the programmer, must explicitly manage their storage.

#### **const**

The **const** modifier indicates that the object can be initialized, but can't be assigned after initialization. In many simple cases, its effect is similar to using a **#define** macro, but a **const** variable actually has a single storage location assigned to it, and its address can be taken using **&**.

#### **extern**

Some variables are defined *externally*. This modifier informs the compiler that here, at the point of definition, it does not need to assign storage for this variable; it is assigned elsewhere, most probably in a separate module. Generally, this requires that the variable exist for the lifetime of the program (process), and so they are *global variables*.

**register**

This modifier informs the compiler that fast access to a variable is required, and asks the C compiler to generate machine code that uses a CPU register to store this variable. The address of this variable can't be obtained using the `&` operator. (An interesting exercise is to confirm what happens when you try this.) Given the strong capabilities of modern compilers, this modifier is generally unnecessary; the compiler will usually do a better job in deciding which variables are important enough to assign to registers than you can.

**static**

We have just seen **static**. A variable is defined within a block, and storage is assigned there statically. Static variables behave like global variables in terms of their permanence, but like local variables in terms of their accessibility. They may be used to track the state of some computation.

**volatile**

The **volatile** declaration is seldom used in ordinary application programs. It tells the compiler that some other piece of hardware besides this CPU may be changing the value, and so it should look at the true location of the variable every the program uses it; it should not copy it into a register and reuse that value, for example, as the optimizer might do for other variables.

**A.1.1 volatile**

Volatile needs a little more explanation. Obviously you would expect that this code can never exit:

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

int i;
extern int status;

int main(){
    status = 1;
    for(i=0;i<10000;i++){
        if(status == 0){
            break;
        }
    }
    exit(0);
}
```

But *this* code can, if another thread or a piece of hardware is updating the variable **status**. This is the correct syntax:

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

int i;
extern volatile int status;

int main(){
    status = 1;
    for(i=0;i<10000;i++){
        if(status == 0){
            break;
        }
    }
    exit(0);
}
```

Of course, this must be linked with another module somewhere that defines the variable `status`; it may be matched to e.g. a *device register* for some sort of peripheral that tells us whether or not the device has completed a chunk of work we asked it to do. This is common in device drivers, for example.

### A.1.2 `const`

This example shows the use of `const`.

```
/* const.c */
#include <stdio.h>
#include <stdlib.h>

const int i = 10;
int j = 5;
int main(){
    /* an illegal assignment to
       a const variable */
    i = j;
    printf("i,j=%d,%d\n",i,j);
    exit(0);
}
```

This program causes the following compilation error.

```
% gcc const.c -o const
const.c: In function 'main':
const.c:8: warning: assignment of read-only
variable 'i'
%
```

## A.2 Unions

A *emphunion* allows us to store different kinds of variables in the same location in memory. More precisely, it allows us to *view* the same memory location in different ways.

```
union universal_tag {
    int ival;
    float fval;
    char *sval;
} u, v;
```

In the above example, two variables `u` and `v` are each defined as a union. Unions are access by following syntax.

```
u.ival = 10;
u.fval = 30.4;
```

Note that care must be taken when the different members of the union are of different sizes.

Unions are useful when accessing certain hardware devices, where sometimes you may want to view a location as a large set of small values, or as a small set of larger ones (e.g., single-precision floating point numbers or double-precision floating point numbers). They are also useful when you are receiving a message across a network, or reading a structure from a file, when you don't know beforehand which of several types of message or structure may arrive.

## A.3 Macros

A preprocessor macro is defined by `#define`. There are two types of macros, object-like macros and function-like macros.

### A.3.1 Object-like macros

```
#define identifier replacement-list
```

For example,

```
#define MAXCLIENT 10
```

replaces exactly the string `MAXCLIENT` with exactly the string `10` wherever it finds it in the source code, before passing on to the next phase of the compiler.

### A.3.2 Function-like macros

⇒ *More detail to be added here.*

```
#define identifier(arguments) replacement-list
```

For example,

```
#define display(x) printf("%s is %d\n", #x, x)
display( abs(-10) );
```

is expanded to

```
printf( "%s is %d\n", "abs(-10)", abs(-10) );
```

Note two important features in this example: substitution for the macro argument is not done inside double-quoted strings; and in order to make a string out of the macro argument, you precede it with `#`.

### A.3.3 The trap of macros

Macros can give you unexpected results when not written carefully enough. They are *not* the same as function calls, though that is not always apparent to the programmer trying to use them, so the programmer who creates them should program defensively to minimize unexpected behavior.

```
/* macrodemo.c */
#include <stdio.h>
#include <stdlib.h>

#define plus2(x)  x+2
int main(){
    printf("%d\n", 3*plus2(4));
    exit(0);
}
```

The preprocessor expands this to:

```
3 * 4 + 2
```

and when the next compiler phase runs, the wrong result will be generated. You need to put parentheses around the arguments.

```
#define plus2(x)  (x+2)
```

## A.4 Conditional Compilation

Conditional compilation is used to adapt a single set of source code to multiple platforms, for debugging, or to customize functionality for specific customers. This is also called conditional inclusion. Once you understand the power of this approach, the importance of a Makefile as a form of *documentation* of how the program was built will be obvious.

### A.4.1 Using macros

A common use is to turn debugging code (especially `printf()` statements) on or off.

```
/* condition1.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
    int i;
    for(i=0;i<5;i++){
#ifdef DEBUG
        printf("factorial(%d) = ",i);
#endif /* DEBUG */
        printf("%d\n",factorial(i));
    }
    exit(0);
}
int factorial(int n) {
    if (n == 0) return 1;
    else return n * factorial(n-1);
}
```

```
% gcc condition1.c -o condition1
% ./condition1
1
1
2
6
24
% gcc -DDEBUG condition1.c -o condition1
% ./condition1
factorial(0) = 1
factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
%
```

Compiling in this manner, the `-DDEBUG` option forces the inclusion of the code between `#ifdef` and `#endif`. Note the `DEBUG` comment indicating the end of the `#if`. This is a common bit of programming style that improves readability of complex compilation cases. `#if` statements can be nested, but their indentation level is always at the left margin, so visually finding the end of a conditional block can be tricky.

Besides `#ifdef` and `#endif`, `#else`, `#elif`, `#ifndef` are used, and they also can be nested.

```
/* condition2.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
#ifdef SFC
    printf("SFC defined.\n");
#else
    printf("SFC not defined.\n");
#endif
#ifdef KEIO
    printf("KEIO defined.\n");
#else
    printf("KEIO not defined.\n");
#endif
    exit(0);
}
```



```
% gcc -DKEIO condition2.c -o condition2
% ./condition2
SFC not defined.
KEIO defined.
% gcc -DKEIO -DSFC condition2.c -o condition2
% ./condition2
SFC defined.
KEIO defined.
%
```

Numeric values and string constant can be assigned inside conditional compilation blocks.

```
/* condition3.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
#ifdef SFC
#define SFC "Shonan Fujisawa Campus"
#endif
    printf("SFC = %s\n",SFC);
#ifdef PI
#define PI 3.14159
#endif
    printf("Pi = %f\n",PI);
    exit(0);
}
```

```
% gcc -DSFC=\"SFC\" condition3.c
% a.out
SFC = SFC
Pi = 3.141590
% gcc -DPI=3.14 condition3.c
% a.out
SFC = Shonan Fujisawa Campus
Pi = 3.140000
%
```

### A.4.2 Platform dependent compilation

```
/* condition4.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
#if defined(sony)
    printf("This is compiled on a SONY machine.\n");
#elif defined(sun)
    printf("This is compiled on a SUN machine.\n");
#elif defined(linux)
    printf("This is compiled on a Linux machine.\n");
#elif defined(sgi)
    printf("This is compiled on an SGI machine.\n");
#endif
    exit(0);
}
```

```
ccn00% gcc condition4.c -o condition4
% ./condition4
This is compiled on a SONY machine.
% ssh ccz01
....
% gcc condition4.c -o condition4
% ./condition4
This is compiled on a SUN machine.
% ssh ccx00
....
% gcc condition4.c -o condition4
% ./condition4
This is compiled on an SGI machine.
%
```

`sun`, `sony`, `sgi` are defined automatically by the specific machine. You don't have to use the `-D` option to define them yourself. FreeBSD defines `__FreeBSD__` automatically.

When the conditional compilation directives `#if` and `defined()` are used, the logical sum (`||`), and logical product (`&&`) operators can be used.

```

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

int main(){
#if defined(sony) || defined(sun)
    printf("compiled in SONY or SUN machine.\n");
#elif defined(sgi)
    printf("This is compiled in SGI machine.\n");
#endif
    exit(0);
}

```

## A.5 Operators

### A.5.1 Operator precedence

Operators	Order of binding
() [] -> .	left to right
! ~ ++ -- - (cast) * & sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
& (bitwise AND)	left to right
^ (bitwise XOR)	left to right
(bitwise OR)	left to right
&&	left to right
	left to right
?:	right to left
= += -= *= /= %=  = &= ^=	right to left
,	left to right

### A.5.2 Conditional operator for expressions

The `if` statement is used to conditionally control execution of a block of C statements. Sometimes what we want to do is simply conditional assignment to a variable:

```

/* condassign1.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
    int x, y, z;
    int c;
    ...
    if (c)
        x = y;
    else
        x = z;
    exit(0);
}

```

This is tedious and uses a lot of vertical space in a program, reducing readability. Thus, a shorthand syntax is included in the language. The following performs exactly the same operation:

```

/* condassign2.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
    int x, y, z;
    int c;
    ...
    x = c ? y : z;
    exit(0);
}

```

### A.5.3 = operator

The order of binding is right to left.

```

/* complexassignment.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
    int a,i,j,k;
    i = 2; j = 5; k = 7;
    a = i = j = k;
    printf("a=%d,i=%d,j=%d,k=%d\n",a,i,j,k);
    exit(0);
}

```

How is `a = i = j = k;` calculated?

```
% ./complexassignment  
a=7,i=7,j=7,k=7
```

### A.5.4 Increment and decrement operators

We have already seen the increment and decrement operators, back on page 51, so we reiterate here for completeness.

```
/* incdemo1.c */  
#include <stdio.h>  
#include <stdlib.h>  
  
int main(){  
    int i;  
    i=1;  
    printf("i=%d\n",i++);  
    printf("i=%d\n",i);  
    printf("i=%d\n",++i);  
    exit(0);  
}
```

The result is shown below.

```
% ./incdemo1  
i=1  
i=2  
i=3
```

An example using pointers is shown below.

```
/* incdemo2.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
    int a[10], *i;
    i = &a[0];
    a[0] = 100;
    a[1] = 200;
    a[2] = 300;
    printf("address of i = %p\n",i);
    printf("*i = %d\n",*i);
    i++;
    printf("address of i = %p\n",i);
    printf("*i = %d\n",*i);
    i = i + 1;
    printf("address of i = %p\n",i);
    printf("*i = %d\n",*i);
    exit(0);
}
```

The result is shown below.

```
% ./incdemo2
address of i = 0x7fff5af43a70
*i = 100
address of i = 0x7fff5af43a74
*i = 200
address of i = 0x7fff5af43a78
*i = 300
```

As a result of `i++`, `i` changes from `0x7fff5af43a70` to `0x7fff5af43a74`. When the `++` and `--` operators are applied to pointers, the size of variable pointed by that pointer is added to the pointer.

### A.5.5 Comma (,) operator

The comma (,) operator is evaluated from left to right.

```
/* commademo.c */
#include <stdio.h>
#include <stdlib.h>

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

The result is shown below.

```
% ./commademo
i=0, j=0
i=1, j=1
i=2, j=2
%
```

## A.6 Fine Points of Execution Order

### A.6.1 Undefined behavior

Imagine the behavior of `x = x++`; . What result would you expect? Is `x` incremented before or after the result is assigned to the variable `x`?

```
/* undefined.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
    int x;
    x = 1;
    printf("x=%d\n",x);
    x = x++;
    printf("x=%d\n",x);
    x = 1;
    printf("x=%d\n",x);
    x = ++x;
    printf("x=%d\n",x);
    exit(0);
    exit(0);
}
```

The result is shown below.

```
ccn??% cc undefined.c -o undefinedcc
ccn??% ./undefinedcc
x=1
x=2
x=1
x=2
ccn??% gcc undefined.c -o undefinedgcc
ccn??% ./undefinedgcc
x=1
x=1
x=1
x=2
ccn??%
```

Two different C compilers (`cc` and `gcc`) produced different results on the same code, compiled in the same fashion on the same machine! The reason is that this behavior is *undefined*. In other words, *don't do this!*

### A.6.2 Logical expressions

Study this example carefully to understand the order in which logical expressions (those returning true or false) are evaluated. Evaluation of a chain of terms such as this aborts as soon as the final value can be determined definitively. `Or` (`||`) aborts as soon as a term returns true, and `and` (`&&`) aborts as soon as it finds one term that returns false. It can be used as a form of conditional execution. Use in the manner in this example is rare, but loop variable tests for null pointers are often implemented in this fashion to avoid dereferencing a null pointer.

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

int main(){
    int x, y, z;
    x = y = z = 1;
    ++x || ++y && ++z;
    printf("x=%d, y=%d, z=%d\n",x,y,z);

    x = y = z = 1;
    ++x && ++y || ++z;
    printf("x=%d, y=%d, z=%d\n",x,y,z);
    exit(0);
}
```



The result is shown below.

```
x=2, y=1, z=1  
x=2, y=2, z=1
```

## A.7 Function Pointers

The name that you give to a function – `f1()` or `gcd()` or what have you – can actually be used as a *function pointer*. It represents the memory address at which the function itself begins. (You did know that all code is also stored in memory, didn't you?)

### A.7.1 Example: the `qsort` library function

As an example, a library function provided in the C library is `qsort()`, which will sort a set of any objects for you, using C.A.R. Hoare's *quicksort* algorithm. In order to sort them for you, it must know where the objects are and how many of them there are, how big each object is, and how to compare two to decide which one should be ordered earlier and which should be ordered later. This comparison is done using a function that *you* write. The way the `qsort()` function knows where to find that function is through a pointer to that function, which you supply as an argument to `qsort()`.

**Note:** Questions about sorting are common questions on job interviews! We have already written a bubblesort routine, in Exercise 8.1 in Chapter 8. Here we are only using the quicksort function supplied by the library, but learning how the algorithm works will be valuable, and you are encouraged to do so.

```

/* qsortints.c */
#include <stdio.h>
#include <stdlib.h>

int intcmp(int *a, int *b){
    if (*a > *b){
        return (1);
    }
    if (*a < *b){
        return (-1);
    }
    return (0);
}

int main(){
    int d[10]={3, 5, 7, 9, 8, 2, 4, 1, 0, 6};
    int i;
    qsort((char *)d, 10, sizeof(int),
        (void *)intcmp);
    for(i=0;i<10;i++){
        printf("%d ",d[i]);
    }
    printf("\n");
    exit(0);
}

```

```

% ./qsortints
0 1 2 3 4 5 6 7 8 9
%

```

Our function `intcmp()` compares two integers. `qsort` has four arguments, as defined in the function prototype:

```

extern void qsort(void *, size_t, size_t,
    int (*)(const void *, const void *));

```

The second line of that is the definition of the function pointer, which must take two pointers as arguments. Because they are defined as `void()` pointers, they can point to any type of objects. If you want the compiler to enforce the type of object pointed to when you are creating your own function pointer definition, specify the type instead.

### A.7.2 Array of pointers to functions

Function pointers can be stored in arrays so that you can choose which one to call based on some other variable. This is a common means of implementing finite

state machines.

```
/* fnptrarray.c */
#include <stdio.h>
#include <stdlib.h>

int f0(),f1(),f2();
int (*flist[10])();

int main(int argc, char *argv[]){
    int i;
    flist[0] = f0;
    flist[1] = f1;
    flist[2] = f2;
    if(1<=argc && argc<=2){
        i = argc;
    }else{
        i = 0;
    }
    flist[i]();
    exit(0);
}

int f0(){
    printf("f0\n");
}
int f1(){
    printf("f1\n");
}
int f2(){
    printf("f2\n");
}
```

Various results are shown.

```
% ./fnptrarray
f1
% ./fnptrarray 2
f2
% ./fnptrarray 2 3
f0
% ./fnptrarray 2 3 4
f0
%
```



## Appendix B

# Development Environment Addendum: MacOS

*⇒ To be filled in.*

### B.1 LLVM

### B.2 Clang

### B.3 LLDB

<http://lldb.llvm.org/lldb-gdb.html> lists corresponding commands for gdb.

### B.4 Use of Xcode



## Appendix C

# Development Environment Addendum: Windows

You're on your own.





## Appendix D

# Data Structures and Algorithms Used in the Linux Kernel

At the beginning of this book, we set the goal of preparing you to read and understand such complex code bases as the Linux or FreeBSD kernel. Here is one summary of some of the concepts used in the Linux 2.6 kernel series, courtesy of Luis de Bethencourt, who adapted it from Vijay D'Silva:

- Linked lists, doubly linked lists, lock-free linked lists.
- B+ Trees with comments telling you what you can't find in the textbooks.
- Priority sorted lists used for mutexes, drivers, etc.
- Red-Black trees are used for scheduling, virtual memory management, to track file descriptors and directory entries, etc.
- Interval trees.
- Radix trees, are used for memory management, NFS related lookups and networking related functionality. (n.b.: A comment in the B+tree source notes that Linux radix trees "don't have anything in common with textbook radix trees.")
- Priority heap, which is literally, a textbook implementation, used in the control group system.
- Hash functions, with a reference to Knuth and to a paper.

- Some parts of the code, such as the Lustre LOV pool, implement their own hash functions.
- Hash tables used to implement inodes, file system integrity checks, etc.
- Bit arrays, which are used for dealing with flags, interrupts, etc. and are featured in Knuth Vol. 4.
- Semaphores and spin locks.
- Binary search is used for interrupt handling, register cache lookup, etc.
- Binary search with B-trees.
- Depth first search and variant used in directory configuration.
- Breadth first search is used to check correctness of locking at runtime.
- Merge sort on linked lists is used for garbage collection, file system management, etc.
- Bubble sort is amazingly implemented too, in a driver library.
- Knuth-Morris-Pratt string matching.
- Boyer-Moore pattern matching with references and recommendations for when to prefer the alternative.

<http://cstheory.stackexchange.com/questions/19759/core-algorithms-deployed>  
<http://luisbg.blogalia.com/historias/74062>

If you have an understanding of most of these structures, you are well on your way to being a strong programmer.

# Index

- acceptance testing, 162
- address, 41
- argc, 127
- argument, 12
- argv, 127
- auto variables, 196
  
- Bell Labs, iv, 81
- big-O notation, 150
- binary search tree, 153
- bounds checking, 40
- buffer, 113
- buffer overflow, 71
  
- character set, 81
- command line option, 129
- compiler, 4
- computational complexity, 150
- const, 196
- cooperative multitasking, 170
- CPU registers, 41
- CPU registers, 197
  
- database, 111
- dependency (build), 35
- dot product, 150
  
- environment variable, 81
- Eratosthenes, 106
- extern, 196
- external variable, 10
  
- file, 111
  
- file offset, 121
- file offset, 112
- file session, 112
- finite state machine, x
- function, 4
- function pointer, 211
  
- garbage collection, 91
- global variable, 10, 196
- global variables, 98
- gprof, 156
- grammar, 10, 106
  
- Hoare, C.A.R., 211
  
- increment operator, 51
- input/output (I/O), 111
- interpreter, 4
  
- Kegel, Dan, 161
- Kernighan, Brian, iv
  
- last in, first out (LIFO), 142
- linked list, 139, 141
- local variable, 10, 98
  
- machine language, 86
- make, 201
- McCarthy, John, 91
- memory allocation, 91
- memory corruption, 41
- modulo, 90
- multibyte characters, 81

object, 5

parameter, 13

persistence, 111

Pike, Rob, 81

pointer, x, 41

polyrhythm, viii, 36

procedure, 4

prototype declaration, 14

quicksort, 211

recursion, 140

red-black tree, 154

register, 197

ring, 147

Ritchie, Dennis, iv, 81

scatter-gather, 113

scope, 92, 196

seek, 121

skip, 125

software quality assurance (SQA), 162

source, 5

stack, 41, 84, 95, 140, 193, 196

stack frame, 144

standard output, 114

state machine, x

static, 197

Thompson, Ken, iv, 81

token, 106

union, 199

volatile, 197

wide characters, 81