C and Data Structures - a well-structured approach

Joseph S. Sventek

September 15, 2017

Preface

This textbook introduces data structures to 1st year computer science students through the medium of the C programming language. It supports delivery of the 3rd quarter of the University of Oregon's introductory computer science sequence.

Why C?

You may ask "Why C" for such a course in the 21st century? The first two quarters of the Oregon introductory sequence cover computational thinking and object-oriented programming using Python as the language of artifact expression. An earlier version of the data structure course used Java as the medium through which to convey basic data structure concepts.

When students who had completed the Java-based course were finally confronted with designing and constructing moderately-sized application and system programs, we discovered that they were insufficiently exposed to the actual mechanics involved in building such data structures for use in anger.

The burgeoning growth of the Internet of Things, and the prevalence of programming these devices in C and C++, provide further impetus to introduce students to programming in these languages early in their academic careers.

General approach of the text

The foci of this text are three-fold:

- introduce the essential aspects of C to enable students to construct well-engineered data structure implementations; these aspects are compared/contrasted to equivalent aspects of Python to aid acquisition and understanding; not all of the language elements are covered, so the interested student is referred to the many excellent texts on C; our preference is Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2nd edition, Prentice Hall, ISBN 978-0131103627;

- introduce the concept of Abstract Data Types, and a well-structured approach to building ADTs in C that provides a bridge to similar usage in C++; again, these elements are compared/contrasted to equivalent Python aspects (object-oriented programming);
- introduce data types of increasing complexity, implemented as ADTs; of particular note is a section on well-designed hash functions for use in hash tables.

Questions to aid self-study are provided in each section of the text, and there are suggested programming exercises at the end of each chapter.

All of the examples in the text are taken directly from a 64-bit Arch Linux[1] image running under Oracle's VirtualBox[2] virtualization environment. The appendices provide instructions for students to install VirtualBox and to access and install a 64-bit Arch Linux image to run in that environment.

In gratitude

Three books produced by Bell Laboratories colleagues were seminal in my acquisition and exploitation of programming languages, programming concepts and UNIX^{TM}:

- Brian W. Kernighan and P.J. Plauger. 1976. *Software Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN:020103669X.
- Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language (2nd ed.).* Prentice Hall Professional Technical Reference. ISBN:0131103709.
- Brian W. Kernighan and Rob Pike. 1984. *The UNIX Programming Environment.* Prentice Hall Professional Technical Reference. ISBN:013937681X.

I am particularly indebted to Brian Kernighan for his excellent writing style in these books, as it made it particularly easy to rapidly internalize the material. I have tried to emulate that style in this textbook.

I would also like to thank the support of my colleagues in the Department of Computer Science for their input to the course syllabus, and for their careful review of drafts of the text.

September 2017

Joe Sventek

[1]https://www.archlinux.org/
[2]https://www.virtualbox.org/

Contents

Chapter 1

Introduction

Computer Science is an interesting mix of mathematics, logic, problem solving, and engineering. This book is focused on the mathematics and engineering aspects, in that it addresses data structures and their implementation.

There has been significant emphasis upon "coding" from both political and commercial perspectives. Code academies, days of code, coder dojos, and many other activities/events have sprung up to entice people, young and old, to actively participate in the computer science revolution. While such activities can give an individual an idea of the power of being able to "code", this is very different from being a computer scientist.

A computer scientist must think computationally, as initially defined by Jeannette M. Wing: "Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction."[1] One particular abstraction level requires the computer scientist to select the most efficient data structure to solve a particular problem. This selection usually involves understanding the inherent space and time complexity of each possible data structure, and then determining the *best* data structure for the problem.

This book focuses on basic data structures, and their space and time complexities. It also attempts to provide you with a rationale for making choices among competing data structures to address a particular problem. Besides familiarizing you with the complexity of these data structures, this course is particularly focused on showing you how to implement the data structures, enabling you to see for yourself how the space and time complexities manifest themselves.

[1] Jeannette M. Wing, "Computational Thinking", *Communications of the ACM*, Vol. 49, No. 3, pp. 33-35, March 2006.

1.1 Assumptions

This book assumes that you have had an introduction to computational thinking, and have learned how to realize designs to solve problems using the Python programming language. It also assumes that you have learned to use the object-oriented features of Python to create abstractions for use in your solutions. Finally, it assumes that you have done most, if not all, of your Python programming using an Integrated Development Environment (IDE), such as IDLE or PyCharm.

1.2 Outline for the book

1.2.1 A beginner's guide to Linux

In order to achieve the goals of the course, we must introduce you to two basic realities for computer scientists:

- the Linux operating environment, and its program development tools; and
- the C programming language.

Thus, the first section of the textbook will focus on the Linux operating environment. It will present the command line environment provided by the shell, some of the useful standard commands distributed in the Linux environment, an introduction to the file system provided by Linux, and the "pipes and filters" philosophy behind standard Linux programs. The introduction to Linux will conclude with a description of the program development tools that we will be using when we implement data structures and programs that use them: `make`, `gcc`, `gdb`, and `valgrind`.

1.2.2 The C programming language

Since the book assumes you have prior experience with Python, it first reviews the principal aspects of Python that you are expected to have mastered. This is necessary since much of the material describing aspects of C will be compared/contrasted with equivalent aspects of Python.

Python is an interpreted language, and much of your assumed experience is using it with an IDE. Programs written in C must be compiled and linked to be executed. This section will first describe the edit, compile, link, execute program development cycle to be used with C.

Overall program structure, command arguments, and standard input, output, and error are then presented, and related to their Python equivalents.

Then the discussion will turn to:

- types, operators, and expressions;
- control flow;
- functions and program structure;
- pointers and arrays;
- structures; and
- input and output.

The section will finish with some example programs, showing how to build them using `make` and `gcc`, and how to use `gdb` and `valgrind` to debug your logic and track down heap memory problems.

1.2.3 Abstract data types

C is *not* an object-oriented language. Prior to the invention of OO languages, there was significant research and development around Abstract Data Types (ADTs); an abstract data type (ADT) is a mathematical model for a data type, defined by its behavior from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations - in other words, describes what it can do, not how to do it. ADTs were first proposed by Barbara Liskov and Stephen N. Zilles in 1974, as part of the development of the CLU language.[2] One particular implication of this approach is that there can be several different implementations that meet the behavior requirements specified in an ADT's interface.

Data structures are concrete representations of data, and are the point of view of an implementer, not a user. Since this book is focused on both interface *and* implementation, this section then focuses on a well-structured C implementation approach for defining the interface to an ADT, and one (or more) implementations of an ADT that incorporate information hiding and method dispatch. This implementation approach also introduces a standard mechanism to address the lack of garbage collection in C.

Several examples of this approach are demonstrated, concluding with an ADT for mutable strings in C.

1.2.4 Data Structures

This section starts off with a discussion of complexity metrics for data structures, in general, and big O notation, in particular.

It then proceeds to describe the interfaces, and make the interfaces concrete using the previously described implementation approach, for ADTs of the following types:

[2] Barbara Liskov and Stephen Zilles, "Programming with abstract data types", Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages, SIGPLAN Notices, Vol. 9, pp. 50–59, 1974.

- array lists;
- linked lists;
- queues;
- stacks;
- hash tables;
- priority queues; and
- searching and sorting.

1.2.5 The appendices

The book concludes with a number of appendices to support the mainline text.

A. An introduction to an open-source virtualization software package, VirtualBox[3], available from Oracle. This package enables you to execute Linux on your host system, which is how all of the examples in the book have been generated.

B. An introduction to Arch Linux[4], a simple, lightweight Linux distribution that performs *very* well when executed in a VirtualBox virtual machine. This appendix provides instructions for creating a VirtualBox virtual machine using a 64-bit Arch Linux image; it also provides instructions for downloading the 64-bit Arch Linux image.

[3]https://www.virtualbox.org/
[4]https://www.archlinux.org/

Chapter 2

A beginner's guide to Linux

2.1 Some history

It is likely that you have been using computers since you were very young. I recall sitting my 2-year old niece down in front of one of the first Macintosh^{TM} computers to play a game called "Lode Runner" in 1985. Admittedly a poor baby sitting approach, but it did the trick; she happily punched the keys for nearly an hour while I was trying to finish up a section of a research paper (using paper and pencil).

Model 33 teletype

The very first interactive computers had a single "console" upon which output from programs was displayed and through which input could be provided using the keyboard. The input and output speeds on such consoles were maddeningly slow, but one could at least get on with the job. The console shown in the figure also had a paper tape reader, a very early form of mass storage for digital information☺.

adm3a terminal

As processors became more powerful and memory became more plentiful, computers were fitted with serial line devices that enabled one to connect "terminals" to the system. Such terminals provided a similar interface to the system as the console, but usually consisted of a cathode ray tube for display and a keyboard; a window of 80 character lines x 24 lines of fixed-width font characters was provided on the CRT, and the typical transmission speeds of such devices was 32 times faster than for the earlier consoles. Multiple people could be connected to the system at one time, leading to "time-sharing" systems; one particular time-sharing system that was popular in universities and among computer science researchers was UNIX^{TM}.

UNIX^{TM} was developed by Ken Thompson and Dennis Ritchie at Bell Laboratories, along with contributions from most of the other researchers at the Bell Labs Murray Hill facility.

It had originally been implemented to ease the withdrawal experienced by Thompson and Ritchie when they were recalled from the Multics Project[1]. UNIX^{TM} was used internally within Bell Labs until Thompson spent a sabbatical year at Berkeley; as with all good computer scientists, Thompson took his environment (UNIX^{TM}) with him; when others at Berkeley discovered the power and simplicity of the environment, Western Electric (Bell Labs parent company) was persuaded to create an academic license that enabled Berkeley and other university computer science departments to have access to the system.

Students that graduated from institutions in which UNIX^{TM} was used found themselves also experiencing withdrawal when they went off to employment in industry or the national laboratories - the licensing of UNIX^{TM} was restricted to universities. Computer system vendors, such as HP, IBM, and Sun, observed this demand, and negotiated licensing arrangements with Western Electric to produce vendor-specific versions of UNIX^{TM} for their workstations and servers (HP/HP-UX, IBM/AIX, Sun/SunOS). Since these were vendor-specific, they presented slightly different libraries and system calls, complicating porting of code between the platforms. To eliminate most of the differences, the vendors, through the activities of a variety of consortia, worked with the IEEE to create the POSIX standards to define a standard set of library functions that could be depended upon when programming on a POSIX-compliant system.

One particularly unique innovation of UNIX^{TM} was how it read commands from a user, commonly called a command line interpreter (CLI). Prior operating systems usually provided a privileged program, sometimes embedded in the kernel itself, for reading and executing commands typed by the user through their terminal. The UNIX^{TM} kernel provided a complete set of system calls enabling processes to create and manage other processes; thus, the CLI could simply be another program. These user-level CLI's have become known as "shells", since the very first such CLI was called the shell, and the program one typed to invoke the shell was `sh`; `sh` was written by Steve Bourne.

Beginning in 1991, Linus Torvalds began work to create a new, free operating system kernel. This kernel, coupled with the applications from the GNU project, has eventually become known as Linux. Linux is open source, licensed under the GNU General Public License. Given its open-source nature, it is now the dominant operating system used in academia, replacing the many flavors of UNIX^{TM}.

2.2 The user interface

The primary interface between a user and Linux is the terminal window - i.e., a window on the screen that emulates the 80x24 terminal windows found on earlier UNIX^{TM} systems. A command line interpreter, usually *bash*[2], reads each command line typed by the user in the window, creates one or more processes needed to execute the command line, and (normally) waits for those processes to finish execution before prompting the user for another command line.

[1]Need a reference for this

[2]The name is an acronym for *Bourne-again shell*, a pun on the [Bourne] shell that it replaces.

This is obviously quite different from the point-and-click mechanism with which you are undoubtedly familiar; Linux systems do provide a graphical user interface that supports the point-and-click mechanism where it is appropriate - e.g., invocation of your browser, creation of another terminal window. Program development on Linux systems is nearly always performed using the command line interface provided by a terminal window in which `bash` is executing. We will, therefore, restrict our discussion to this environment.

2.3 The command line interpreter - `bash`

`bash` reads the commands typed by a user in its terminal window and executes those commands. `bash` displays a prompt string, reads the line of input typed, and executes the command. The prompt string can be customized, as will be shown later; until then, we will use the string "$ " as the prompt.

Consider the following example:[3][4]

```
$ date                   show today's date
Tue Jun 27 15:09:02 PDT 2017
$ pwd                    print the current working directory
/home/me
$ date; pwd              a semicolon is a command separator
Tue Jun 27 15:09:43 PDF 2017
/home/me
```

From the last example, you see that you can place more than one command on a single command line. A `;` is a command separator; "a ; b" tells the shell to first execute *a*, then execute *b*.

After `bash` displays the prompt string, it reads everything you type up to when you press the `Enter` key. (On some systems the key is labeled `Return`; we will refer to it as the `Enter` key in this text.) If you wish to erase a character before pressing `Enter`, you should press the `Backspace` key. If you have made a number of mistakes, and wish to simply erase the entire line and start again, you can type *ctl-u*[5]; sometimes *ctl-u* is called the *line kill* character.

Both `date` and `pwd` are commands that do not require any additional information in order to do their jobs.[6] Most programs require additional *command arguments* to provide additional information to the program while it is executing.

Consider the program `echo` - it has a particularly simple job: print the supplied

[3]You are encouraged to try these commands on your Linux system as you read along. You should see the same or similar output from the commands.

[4]In the examples in this chapter, the text that you type is in **`boldface`**, while that displayed by the shell and commands is in `normalface`.

[5]The expression *ctl-x* means you should press the `x` key while holding down the `Ctrl` key.

[6]`date` *can* take a number of arguments that determine how to format the date string that it prints. Type "linux man date" as a search string in your favorite browser to see the arguments that `date` understands.

arguments and exit, as in the following examples:

```
$ echo this                print 'this' on the output
this
$ echo this and that       print 'this and that'
this and that
$ echo this  and that      two spaces between 'this' and 'and'
this and that             the extra space has disappeared
$ echo 'this  and that'    quote the input
this  and that            the extra space was maintained
```

These examples demonstrate a number of features of `bash`.

- The shell breaks up the line of input that you typed into separate words; the words can be separated by blanks or tabs, or by punctuation like the semicolon.
- The first word in a command (`echo` above) is the program to execute; we will discuss later how the shell looks for the program file that corresponds to that word.
- The other words in the command are provided to the program as a list of words, for it to do with as it sees fit. `echo` simply prints each of these words, placing a single space between each pair of words.
- It does not matter how many spaces or tabs separate two words; it does not change the list of words that the shell gives to the program.
- If you want to have many words in a single argument, you can quote the phrase using `'` or `"`. In the last example above, `echo` received a list with a single "word" in it, consisting of the phrase `'this  and that'` *without* the quote characters.

We saw earlier that `bash` gives a special meaning to the character `;` — now we see that it gives special meaning to `'` and `"`, as well. In fact, `bash` gives special meanings to most non-alphanumeric characters. If you want to provide an argument to a program that contains a non-alphanumeric character, the easiest way to prevent the shell from giving it its special meaning is to quote the argument. For example:

```
$ echo A semi-colon '(;)' is a command separator.
A semi-colon (;) is a command separator.
```

A command argument that needs to contain an apostrophe, `'`, can be escaped using a quotation mark, `"`, and vice versa.

By convention, command arguments that start with `-` are considered flags to a program - i.e., they change the way the program does its task; these are usually a single letter following the hyphen, and are called short flags. If additional information is required when a short flag is specified, that information must be the next word that immediately follows the short flag, as in `-n name`. Finally, if you wish to specify several short flags on a single command line, you can usually collapse them into a single flag - e.g., `command -a -b -c` can usually be written as `command -abc`.

A second flag convention, called long flags, has a flag starting with `--`, and is usually written out in full; for example, `command -a` might be the same as `command --all`. If additional information is required when a long flag is specified, it is written as `--name=value` - i.e., the information associated with the flag is part of the same command argument. You will note in the previous example that a hyphen, `-`, is not special to `bash`, nor is a period, `.`[7]. The equals sign, `=`, is another non-alphanumeric character that has no special meaning to `bash`. The non-special nature of these characters enables these two flag conventions.

Let's look at examples of the use of both short and long flags. The command `ls` lists the files found in one or more directories; if no directories are specified, the files in the current directory are listed. `ls` understands a number of flags that dictate how it displays the files in a directory. The examples below show both the short and long form of some of these flags.

```
$ ls                     list the contents of the current directory
book  calendar.data  Music  Pictures  shopping.list  src
$ ls -a                  list the entire contents of the current directory
.   .bashrc   .vimrc  calendar.data  Pictures       src
..  .profile  book    Music          shopping.list
$ ls --all               long form of -a
.   .bashrc   .vimrc  calendar.data  Pictures       src
..  .profile  book    Music          shopping.list
$ ls --group-directories-first  places directories first, no short version
book  Music  Pictures  src  calendar.data  shopping.list
$ ls -p                  append / to indicate directories
bin/  calendar.data  Music/  Pictures/  shopping.list  src/
$ ls --indicator-style=slash    long version of -p
bin/  calendar.data  Music/  Pictures/  shopping.list  src/
$ ls -t *.*              display files ordered by modification time, newest first
shopping.list  calendar.data
$ ls -rt *.*             same, but oldest first (reverse sort)
calendar.data  shopping.list
$ ls -w 20 *.*           output is 20 characters wide
calendar.data
shopping.list
$ ls --width=20 *.*      the long form of -w 20
calendar.data
shopping.list
```

After any flags, what about the other arguments to a command? Most commands need to

[7]This latter assertion is not completely true - if the 1st "word" in a command is `.`, `bash` does something special.

work on files, so the non-flag arguments are typically filenames. There are other sorts of information that a command might need; for example, a program that searches for textual patterns in a file requires at least one argument indicating the pattern we wish to find.

In the last four examples above, we provided "*.*" as an argument to `ls`. What does that mean?

As we mentioned previously, most of the non-alphanumeric characters available on the keyboard have a special meaning to `bash`. When `bash` breaks up the command into words, it looks for four particular special characters, (`*`, `?`, `[`, and `]`), as these indicate that `bash` should perform a pattern match against filenames in the current directory. In the last two `ls` examples above, "*.*" indicates that `bash` should replace that string by all filenames that consist of 1 or more characters before a `.`, and 0 or more characters after. In our directory, this pattern matches exactly two files, `calendar.data` and `shopping.list`. `bash` replaces the single "word", "*.*", by two words in the list presented to `ls`. The wildcard character, `*`, is often used in the shell to select a subset of files to be processed by the command.

The `?` in a command argument indicates that it matches any single character at that point - e.g., "jo?n" matches `john` or `joan`, but not `johan`. The square brackets enable the specification of a range of characters to match at a particular location in the filename - e.g., "ls a.[ch]" would match files named `a.c` and `a.h` in the current directory, but would *not* match `a.x`. One can also specify a range of characters within the square brackets - e.g., "ls *.[a-d]" would match any files that end in `.a`, `.b`, `.c`, or `.d`.[8]

What should you do if you start a program by mistake? Most commands can be stopped by typing *ctl-c*, often known as the `INTERRUPT` character. Some programs, like text editors, will stop whatever the program is doing when you type the `INTERRUPT` character, but enable you to issue another command to the program after it has stopped. Closing the terminal window will stop most programs, as well.

2.4 Simple commands

Linux provides you with a number of simple commands to manipulate files and your environment. This section covers some of the more useful ones.

2.4.1 Obtaining help

Online manual pages for all of the commands in Linux are available over the Internet; it is a good idea to maintain a browser window open while you are working so that you can access these manual pages. A search query of the form "linux man *command*" will yield

[8]Note that the wildcard expansion is done by `bash`, *not* by the command itself (in this case, `ls`). By performing such substitution in the command line interpreter, it means that all programs benefit from this feature.

several links to online man pages for *command*. Additionally, there is a directory available at `http://man7.org/linux/man-pages/dir_all_by_section.html`; you can skim it quickly for commands that might be relevant to what you want to do. There is also an introduction to the system at `http://www.tldp.org/LDP/intro-linux/intro-linux.pdf` that gives an overview of how things work.

Depending upon how complete a Linux system you have, the man pages for most of your commands may also be available on your Linux system. If so, you can display the manual page for *command* by typing "man *command*" to `bash`. Thus, to read about the `ls` command, type

```
$ man ls
```

2.4.2 Creating files

Information on Linux systems is stored in files. In order to enter information into a file, as well as to modify that information, you will need to use a text editor. It is likely that you have experience using *document* editors, such as Microsoft^{TM} Word. Document editors not only enable you to enter and edit information in a document, it also enables you to specify how that information should be formatted when it is displayed. Most files in a Linux system do not require such formatting information - i.e., the content of the file is a sequence of characters, with the end of line being the only type of formatting needed.

Every Linux system has several screen editors; the one you choose to use is a matter of personal taste. The Arch Linux image described in Appendix B comes with `nano` (`https://wiki.archlinux.org/index.php/nano`) and `vi` (`https://wiki.archlinux.org/index.php/Vim`). You may also install any of a number of other editors, such as `emacs` and `gedit`. See `https://en.wikipedia.org/wiki/List_of_text_editors` for a list of text editors that has been compiled in Wikipedia.

2.4.3 Listing your files

We have previously encountered `ls` in section 2.3. In this section, we will provide examples of one other flag to `ls` that is heavily used.

As we will describe in the next section, Linux stores a number of items of information about each file in the file system; this information is referred to as *metadata*. `ls` can be used to see some of this metadata:

```
$ ls -l
total 24
drwxrwxr-x 2 me me 4096 Jul  6 14:59 book
-rw-rw-r-- 1 me me  141 Jul  6 14:59 calendar.data
drwxrwxr-x 2 me me 4096 Jul  6 14:59 Music
drwxrwxr-x 2 me me 4096 Jul  6 14:59 Pictures
-rw-rw-r-- 1 me me   86 Jul  6 14:59 shopping.list
drwxrwxr-x 2 me me 4096 Jul  6 14:59 src
```

As you can see, the `-l` flag gives a "long" listing that provides this metadata; the first line indicates the number of blocks of disk space occupied by the listed files. Each subsequent line provides information about an individual file:

- the first character indicates if the file is a directory (`d`) or a normal file (`-`);
- the next 9 characters indicate permissions to read, write, or execute the file; the first 3 characters are for the owner of the file (`me` in this case); the next 3 characters are for the group with which this file is associated (`me` is the associated group); the next 3 characters are for everyone else;
- next we have the number of links to the file; this will be discussed in the next section on the file system;
- the owner of the file (`me`) and the associated group (`me`) follow;
- next we have the size of the file (in bytes);
- this is followed by the month, day, and time of last modification;
- and finally, we have the name of the file.

We previously noted that after all of the flags, one can specify one or more file names to `ls`, which then restricts its activity to those files. For example,

```
$ ls -l calendar.data
-rw-rw-r-- 1 me me  141 Jul  6 14:59 calendar.data
```

2.4.4 Naming your files

Most operating systems, and Linux is no exception, have rules about creating legal filenames. Firstly, there is usually a limit on the length of a filename; early operating systems had very severe restrictions; Linux restricts the length of a filename to 255 characters. It is unusual for anyone to want to type that many characters as an argument to a command, so, in practice, you will usually use far fewer letters in your filenames.

Secondly, what are the legal characters in a filename? Linux allows any character in a filename except for `/` and a null character; this does *not* mean that you should start putting lots of strange characters in your filenames. The POSIX specification[9] is quite clear on characters to use in filenames that are portable across *all* POSIX-conformant systems[10]:

- any upper-case character from the set `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`;
- any lower-case character from the set `"abcdefghijklmnopqrstuvwxyz"`;
- any digit from the set `"0123456789"`; and
- any character from the set `"._-"`.

We have already seen that `-` is used to introduce flags in bash command lines, so you are *strongly* recommended to avoid starting your filenames with a `-`. We have also seen above that filenames that start with a `.` are *hidden* - i.e., they are not displayed by `ls` unless you specify the `-a` or `--all` flags; thus, you should avoid starting your filenames with a `.` unless you want their existence to be hidden in this way.

2.4.5 What's in a file?

We often try to give files descriptive names in order to remember their contents. When that fails, one often resorts to displaying the contents of the file to jog one's memory.

You could certainly use your favorite editor to display the contents, using whatever commands it provides. While this works, it is not the most efficient way to display the contents, since the editor is designed to enable you to *modify* the file.

Linux provides several commands that can be used to display the contents of a file:

- The simplest program is `cat`, which simply prints the contents of each file argument on the terminal.

  ```
  $ cat shopping.list
  1 bottle of milk
  2 granny smith apples
  10 hot house tomatoes
  1 six-pack of Coca Cola
  ```

- `cat` works perfectly well for short files, like `shopping.list`, but for very long files the contents will be displayed so rapidly in your terminal window that you will only see the last screenful of lines (normally 24 lines) in the terminal window. Linux provides two commands that will show one screenful at a time, waiting for an action

[9] `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap03.html#tag_03_280`
[10] Linux is a POSIX-conformant system

from the user to continue the display, to search for a pattern, or to perform other tasks.

- `more` is an especially primitive program for paging through text one screenful at a time. Often, this is all that is needed.
- Counterintuitively, `less` is a program similar to `more`, but with more features, such as backward movement in the file; additionally, `less` does not have to read the entire input file before starting, so with large input files it starts up *much* faster than text editors like `vi`.

2.4.6 Moving, copying, removing files - `mv`, `cp`, `rm`

In your prior experience with computer systems, you will have had the occasional need to rename a file. This is done with the `mv` command:

```
$ mv shopping.list Shopping.List
$ ls *.*
calendar.data  Shopping.List
$ cat shopping.list
cat: shopping.list: No such file or directory
```

The file `shopping.list` has been "moved" to a file named `Shopping.List`. The old filename has disappeared, as evidenced by the output from `ls` and `cat`. This example also shows that filenames are case-sensitive - i.e., the name `shopping.list` is different from `Shopping.List`.

Note that if the target filename in a `mv` command already exists, it is replaced.

To make a *copy* of a file, one uses the `cp` command:

```
$ cp Shopping.List 20170706-shopping.list
```

saves an archive copy of the file `Shopping.List`.

To remove a file, one uses the `rm` command:

```
$ rm Shopping.List shopping.list
rm: cannot remove 'shopping.list': No such file or directory
```

As you can see, `rm` warns you if one of the files specified did not exist. You can then invoke `ls` to verify that `rm` did its job:

```
$ ls *.*
20170706-shopping.list  calendar.data
```

2.4.7 Other useful programs

Let's re-create our shopping list file (recall that we removed it above) for use with the other programs described in this section.

```
$ cp 20170706-shopping.list shopping.list
$ cat shopping.list
1 bottle of milk
2 granny smith apples
10 hot house tomatoes
1 six-pack of Coca Cola
```

2.4.7.1 Count lines, words, and characters - `wc`

Suppose we need to know the number of different types of items that are contained in the shopping list. We could count the number of lines on the screen after displaying the file using `cat`; this type of processing of files happens often enough that the program `wc` is provided - `wc` counts the number of characters, words (each word is a sequence of non-whitespace characters, separated from other words by whitespace), and lines. The number of lines in `shopping.list` is exactly what we need:

```
$ wc shopping.list
 4 17 85 shopping.list
$ wc -l shopping.list
4 shopping.list
$ wc -w shopping.list calendar.data
 17 shopping.list
 19 calendar.data
 36 total
```

If we simply invoke `wc` without any flags, it will print the number of lines, number of words, and number of characters for each file specified in the command line. If we specify `-l`, `wc` restricts itself to counting lines; `-w` or `-c` restricts `wc` to counting words or characters, respectively. If more than one file is specified in the command line, `wc` displays the counts for each file, and provides a total in each category at the end.

2.4.7.2 Find occurrence of a pattern - `grep`

Suppose we do not remember whether we added apples to our shopping list. The command `grep`, which stands for **g**et **r**egular **e**xpression and **p**rint, will search the file arguments for lines that match a pattern. The following shows us using `grep` to answer our question about apples.

```
$ grep apple shopping.list
2 granny smith apples
```

Thus we see that we did add granny smith apples to the list.

We see that the first non-flag argument to `grep` is the pattern to search for - in this case, it is simply the string "apple". `grep` understands much more powerful patterns, called *regular expressions*; we recommend that you consult the Linux man entry for `grep` for more discussion of these more powerful patterns.

Suppose that your friend purchased apples on the way home from class, so that you do not need to purchase them when you go to the store; the following shows how to print all lines that do *not* match the pattern.

```
$ grep -v apple shopping.list
1 bottle of milk
10 hot house tomatoes
1 six-pack of Coca Cola
```

If you specify two or more file arguments to `grep`, it will prefix each matching line with the name of the file in which it was found.

```
$ grep apple shopping.list calendar.data
shopping.list:2 granny smith apples
```

Unsurprisingly, the term "apple" is not found in `calendar.data`; even so, since we specified two filenames in the `grep` command, it prefixes the matching line in `shopping.list` with the name of the file.

2.4.7.3 Sorting files - `sort`

This command sorts its input into alphabetical order, by default. The order can be changed using various flags which will be shown below. Let's sort our shopping list.

```
$ sort shopping.list
1 bottle of milk
1 six-pack of Coca Cola
10 hot house tomatoes
2 granny smith apples
```

Note that the default sorting order of characters is blank, then digit, then upper-case letter, then lower-case letter. This explains why the "milk" line appears before the "Coke" line (s in 'six' comes after b in 'bottle'), why the "tomatoes" line appears after the "Coke" line ('1 ' comes before '10'), and why the "apple" line comes last ('2' comes after '1').

As indicated above, `sort` has many options to control the sort order - e.g., numerical order, by field within each line, reverse the order. Here are some examples, again using `shopping.list`.

```
$ sort -r shopping.list           reverse the order of the sort
2 granny smith apples
10 hot house tomatoes
1 six-pack of Coca Cola
1 bottle of milk
$ sort -n shopping.list           numeric sort on first field
1 bottle of milk
1 six-pack of Coca Cola
2 granny smith apples
10 hot house tomatoes
$ sort -k 2 shopping.list         sort on 2nd field
1 bottle of milk
2 granny smith apples
10 hot house tomatoes
1 six-pack of Coca Cola
```

Note that a field is defined as a sequence of non-whitespace characters separated from other fields by whitespace characters. Thus, in the last example, **b**ottle <**g**ranny <**h**ot <**s**ix.

2.4.7.4 Beginning and end of a file - `head` and `tail`

We discussed `more` and `less` above for displaying the contents of a file. A very common occurrence is the need to just see the first few lines, or the last few lines, of a file. This capability is provided by `head` and `tail`, respectively, as shown in the following examples.

```
$ head shopping.list              print the first 10 lines
1 bottle of milk
2 granny smith apples
10 hot house tomatoes
1 six-pack of Coca Cola
$ head -n 1 shopping.list         print the first line
1 bottle of milk
$ head --lines=1 shopping.list    print the first line
1 bottle of milk
$ tail shopping.list              print the last 10 lines
1 bottle of milk
2 granny smith apples
10 hot house tomatoes
1 six-pack of Coca Cola
$ tail -n 1 shopping.list         print the last line
```

```
1 six-pack of Coca Cola
$ tail --lines=1 shopping.list     print the last line
1 six-pack of Coca Cola
```

2.4.7.5 Comparing files - `cmp` and `diff`

It is common to need to compare files to understand *if* they are different, and if they are, *how* they differ. Earlier in this chapter, we made a copy of `shopping.list` named `20170706-shopping.list`. Perhaps, as part of a dietary regimen, we need to keep the shopping list that we use each day.

```
$ cat 20170706-shopping.list
1 bottle of milk
2 granny smith apples
10 hot house tomatoes
1 six-pack of Coca Cola
```

We have discovered that we do not eat enough tomatoes, such that we are building up a tomato mountain in the kitchen. Therefore, we change the quantity of tomatoes in `shopping.list` to 3 instead of 10, as shown below.

```
$ cat shopping.list
1 bottle of milk
2 granny smith apples
3 hot house tomatoes
1 six-pack of Coca Cola
```

Just before you go to the store, you cannot remember if you changed the quantity of tomatoes or not. You could just display the file, and look for the changes, but there may have been many changes, and we sometimes do not see subtle textual differences. Therefore, we can rely upon comparison tools to help us out.

The first comparison program, `cmp`, compares the two files and reports the first difference that it finds.

```
$ cmp shopping.list 20170706-shopping.list
shopping.list 20170706-shopping.list differ: byte 40, line 3
```

While `cmp` has indicated we made some change to `shopping.list`, we want to verify that we made the correct change, just in case. This is where `diff` comes handy.

```
$ diff shopping.list 20170706-shopping.list
3c3
< 3 hot house tomatoes
---
> 10 hot house tomatoes
```

The '3c3' line indicates that only line 3 of the two files are different; lines preceded by <are lines in the first file (`shopping.list` in this case), and lines preceded by >are lines in the second file (`20170706-shopping.list` in this case). If there had been more than one difference in the two files, each such difference would have been shown in this way.

2.5 The file system

We have introduced the basic notion of a file, and metadata about a file, in the previous section. Now we need to understand how Linux organizes the file system.

In your previous computing experience, you have undoubtedly been exposed to folders via a graphical user interface. A folder contains files and, often, other folders. Double clicking on a folder usually causes its contents to be displayed. Double clicking on a file in a folder usually causes an application associated with that type of file to execute on that file. Most systems provide some way to visualize the file tree hierarchy; here is the hierarchy for the current state of this book. Note that directories are shown in **blue** and files are shown in **green**.

```
CaDS
├── book.tex
├── ch00
│   ├── Preface.pdf
│   ├── Preface.synctex.gz
│   └── Preface.tex
├── ch01
│   ├── ch01.pdf
│   ├── ch01.synctex.gz
│   └── ch01.tex
├── ch02
│   ├── adm3a.jpg
│   ├── ch02.pdf
│   ├── ch02.synctex.gz
│   ├── ch02.tex
│   ├── me
│   │   ├── 20170706-shopping.list
│   │   ├── book
│   │   ├── calendar.data
│   │   ├── Music
│   │   ├── Pictures
│   │   ├── shopping.list
│   │   └── src
│   ├── me.tgz
│   └── model33teletype.jpg
├── Outline.pdf
├── Outline.synctex.gz
└── Outline.tex
```

Partial directory tree

Different folders can each contain a file with the same name, such as `shopping.list`, for example, although the contents of the files are different. This means that the unique name for a file is a sequence of directories from the root to the file itself; for example, in this directory tree, rooted at `CaDS`, the unique name for `shopping.list` is the sequence of names `{ CaDS, ch02, me, shopping.list }`. This complete sequence of names is termed a *pathname*, as it describes a path from the root to the file that is of interest.

Linux provides an hierarchical file system of this type. The root of the tree has the name `/`, and the complete pathname for `shopping.list`, assuming that `CaDS` is a directory in the root, would be `/CaDS/ch02/me/shopping.list` - i.e., the complete pathname starts with the root, and each subsequent pair of names in the path are separated by `/`. While this overloading of the use of `/` may seem strange, one quickly becomes accustomed to it.

Given such a hierarchical structure, it is common to use family terms to describe the relationships between directories. If a directory `a` contains a directory `b`, `a` is the *parent* of `b`, and `b` is the *child* of `a`. `a`'s parent is the *grandparent* of `b`, and so on.

2.5.1 Current working directory and home directory

It would be extremely tedious to have to type the full pathname each time you wanted to refer to a file, so `bash` maintains a notion of your *current working directory*. It also maintains a notion of your *home directory*, which has been assigned to you when your account was created; whenever you start up a terminal window, your current working directory is your home directory. If you type a file name that does *not* start with a `/`, it assumes that you are naming the file relative to the current working directory. All of the examples in the previous sections of this chapter assume that our current working directory is `/CaDS/ch02/me`, such that invoking `cat shopping.list` will yield the contents of `shopping.list` on the screen.

For the purposes of this textbook, your home directory is `/home/me`. Your home directory never changes, while your current working directory can, as we will now show[11].

```
$ pwd                            what is our current working directory
/home/me                         just started bash, we are in our home directory
$ ls -p                          what's in our home directory?
20170706-shopping.list  calendar.data  Pictures/      src/
book/                   Music/         shopping.list
$ cd Music                       change into the Music directory
$ pwd                            let's see if we were successful
/home/me/Music                   yes we were
$ ls                             what's in the Music directory? nothing
$ cd ..; pwd                     go up one level in the directory hierarchy
/home/me                         we are back home
$ cd src; pwd; cd; pwd           cd into src; what does cd without a directory do?
/home/me/src
/home/me                         ok, cd without a directory takes us home
$ cd /home/me/Pictures; pwd      we can specify a full pathname to cd, as well
/home/me/Pictures
$ cd ~; pwd                      what does '~' mean?
/home/me                         ok, it's shorthand for home
```

You may wonder why we need the `'~'` shorthand for the home directory; it can be used as the first character of a pathname, as in `~/shopping.list`, to access files that are in your

[11]We will no longer show the text you type in **boldface**. It should be obvious as it follows the `bash` prompt, "$ ".

home directory. Thus, no matter where your current working directory is in the hierarchy, you can easily access files that are directly reachable from your home directory.

The use of `'..'` seems a little strange for referring to the parent of our current working directory. Why is this the case? Remember when we used the `-a` flag to `ls` earlier:

```
$ pwd
/home/me
$ ls -a
.   .bashrc   .vimrc  calendar.data  Pictures       src
..  .profile  book    Music          shopping.list
```

Notice that there are two entries in the directory named `'.'` and `'..'`. These entries were placed in the directory when the directory was created. `'..'` points to the parent of this directory (here it points to `/home`); `'.'` points to the directory itself (here it points to `/home/me`.

Let's navigate up the tree to see what happens.

```
$ cd Music; pwd                  let's start somewhere interesting
/home/me/Music
$ cd ..; pwd                     go up one level
/home/me
$ cd ..; pwd                     again
/home
$ cd ..; pwd                     and again
/
$ cd ..; pwd                     will this work?
/
```

We see from this little experiment that the root is its own parent; we can ask to change our working directory to root's parent, but we stay in the same place.

Just as we can use `'~'` as shorthand for our home directory, we can use use `'.'` as shorthand for our current directory and `'..'` as shorthand for the parent of our current directory. Consider the following commands; the examples refer to the file system shown on page 19.

```
$ pwd
/CaDS/ch02/me
$ ls -p ..
adm3a.jpg  ch02.synctex.gz  DirectoryTree.pdf  me.tgz
ch02.pdf   ch02.tex         me/                model33teletype.jpg
$ cd Music; pwd
/CaDS/ch02/me/Music
$ cat ../shopping.list                  Need to look at shopping list again
```

```
1 bottle of milk
2 granny smith apples
3 hot house tomatoes
1 six-pack of Coca Cola
$ head -n 1 ../../me/shopping.list     Admittedly unusual, but works
1 bottle of milk
$ cd ..; tail -n 1 ./shopping.list     ./name treated identically to name
1 six-pack of Coca Cola
```

The last two examples are meant to show that you can introduce `'.'` and `'..'` as elements of a pathname. The last example may seem strange, but later you will see a situation where using `'./'` comes in handy.

2.5.2 Creating a new directory - `mkdir`

Let's suppose that we want to create a directory to hold all of our archival shopping lists.

```
$ cd                        go home
$ mkdir ShoppingLists       create our directory
$ cd ShoppingLists          make it our working directory
$ ls -a                     what's in a newly-created directory?
.  ..                       only the link to our parent and ourselves
$ cd ..                     up to our parent
$ ls -l ShoppingLists       it's empty, so no output from ls
$ ls -dl ShoppingLists      -d flag says describe the directory
drwxrwxr-x 2 me me 4096 Jul 11 16:00 ShoppingLists
```

There is also an `rmdir` command that can be used to remove a directory; this will only work *if* the directory is empty - i.e., the only entries in the directory are `'.'` and `'..'`.

2.5.3 The Linux hierarchical file system structure

Linux organizes all of the files in the system into a single hierarchy. There can be millions of files in the file system of a single Linux system; for the Arch Linux system used in this book, there are >160,000 files and >13,000 directories.

For typical use of a Linux system, the most important directories are:

- `/bin` - contains basic progams (like `bash`) that are required during the boot process;
- `/sbin` - contains programs that must be accessed by the system administrator;
- `/lib` - contains dynamic libraries and static support files needed in the boot process;
- `/etc` - contains configuration files for the system;
- `/home` - contains the home director for each user (`/home/me` in our case);
- `/usr` - has several important sub-directories:
  - `/usr/bin` - contains programs that are accessed by all users,
  - `/usr/sbin` - contains programs that must be accessed by the system administrator,
  - `/usr/lib` - contains dynamic libraries and static support files for the programs in `/usr/bin` and `/usr/sbin`,
  - `/usr/include` - contains include files needed by C and C++ programs,
  - `/usr/share/doc` and `/usr/share/man` - contain manuals, documentation, examples, etc.,
  - `/usr/local` - has `bin`, `include`, `lib`, `etc`, etc. directories for locally added software.

```
/
├── bin
├── boot
├── dev
├── etc
├── home
│   └── me
├── lib
├── lib64
├── lost+found
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin
├── srv
├── sys
├── tmp
├── usr
│   ├── bin
│   ├── include
│   ├── lib
│   ├── lib64
│   ├── local
│   │   ├── bin
│   │   ├── etc
│   │   ├── games
│   │   ├── include
│   │   ├── lib
│   │   ├── man
│   │   ├── sbin
│   │   ├── share
│   │   └── src
│   ├── sbin
│   ├── share
│   └── src
```

2.6 Back to the shell

2.6.1 Environment variables

We have already discussed how the shell breaks up the command you typed into words, uses the 1st word in that sequence to find the program to run, and passes all of the other words to the program for it to interpret. Besides passing these arguments to a program, it also maintains a set of (name, value) pairs that it provides to the program for it to use if it wishes - this set of (name, value) pairs is known as the *environment*.

The command to type to see what is currently in the environment is `env`:

```
$ env                   display the environment
 * * *                  several variables specific to particular programs
XTERM_SHELL=/bin/bash   the shell reading a new terminal window
USER=me                 my identity
PWD=/home/me            my current working directory
HOME=/home/me           my home directory
 * * *                  many other variables specific to particular programs
```

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/bin/site_perl:
/usr/bin/vendor_perl:/usr/bin/core_perl
 * * *                    many other variables specific to particular programs
```

Each environment variable has a name consisting of upper-case letters, digits, and underscores ('_'). Each is displayed in the form "NAME=value". We can use our friend `echo` to see the current value of a particular environment variable as follows:

```
$ echo $USER              what is the current value of USER?
me                        as we expected
```

When `bash` encounters `$ENVIRONMENT_VARIABLE_NAME` in a command line, it replaces it with the current value of that variable. Thus, in the example above, `bash` replaces `$USER` with `me`, sending that as the command argument to `echo`, which dutifully prints `me` on the terminal.

Environment variables are just a specific case of shell variables, which enable the user to assign values to names and refer to those values using the `$name` syntax.

How does one establish the value of a variable? How does one remove a variable? How does one make that variable part of the environment? The easiest way to answer these questions is to show a number of examples.

```
$ ID=/usr/local/include   'ID' has a very long value
$ echo $ID                did it work?
/usr/local/include        yes
$ bash                    create a child process running bash
$ echo $ID                see if the child process knows the value of 'ID'
                          no, it is not part of the environment
$ exit                    exit the child bash process
$ export ID               make this variable part of the environment
$ bash                    create a child process running bash
$ echo $ID                see if the child process knows the value of 'ID'
/usr/local/include        yes
$ exit                    exit the child bash process
$ unset ID                remove 'ID' from the set of variables
$ echo $ID                make sure it has been removed
                          yes, it has no value
```

`bash` makes the environment available to the process that it creates to run your program; the program can access the value associated with an environment variable through a library call (`getenv()`) provided by the system.

A particular environment variable, `PATH`, is of particular importance to `bash`, as it tells `bash` where to look for the program that you have requested. `PATH` consists of a sequence of directory names separated by colons (`:`). After `bash` has broken up your command into words, it then proceeds to search for an executable file with the name you have specified in each component of `PATH` until it either finds such a file, or it has exhausted the `PATH`. In the former case, it then runs the program in a process, giving it the remaining arguments for it to process.

A standard program is provided which will perform this search *without* starting up the program - `which`. Let's look at it in action.

```
$ which bash              where is bash stored?
/usr/bin/bash
$ which wc                where is wc stored?
/usr/bin/wc
$ which pyhton3           look for python3, but a typo
which: no pyhton3 in (/usr/local/sbin:/usr/local/bin:/usr/bin:
/usr/bin/site_perl:/usr/bin/vendor_perl:/usr/bin/core_perl)
$ pyhton3                 let's see what bash says
bash: pyhton3: command not found
```

From these examples, we can deduce a couple of things:

- most of the standard programs are in `/usr/bin`; and
- if the command you have typed cannot be found using `PATH`, `bash` will report it, and `which` will also report the search path used.

While this is interesting, if the search path was fixed, then there would not be much point in discussing it. Fortunately, since `PATH` is an environment variable, you can change it, and `bash` will begin to use the new version of `PATH` on the very next command you type. Before showing you how this is done, why might we want to change `PATH`?

We've previously discussed the `/usr/local` branch of the file system as a place where site-specific programs are typically installed on your system. You really wouldn't want to put these programs in `/usr/bin`, since the next release of the operating system would require you to remember all of the non-standard programs that you placed in `/usr/bin` so that you could reinstall them. Sometimes you want a site-specific version of one of the standard programs, and whenever that program is invoked, you want the site-specific version to be found instead of the standard one. And, finally, you might want to have a number of your own programs (it is really easy to do) that you use regularly, and you want `bash` to be able to find them in the same way.

Note that the search performed by `bash` goes from left to right in the sequence of directories that make up `PATH`. Thus, if you want to find your version of `ls` instead of the one stored in `/usr/bin`, then your version will have to be in a directory that is earlier in `PATH` than `/usr/bin`. Let's experiment with this a bit.

```
$ mkdir ~/bin            let's create a personal bin directory
$ cp /usr/bin/ls ~/bin   make a copy of ls in that directory
$ sudo cp /usr/bin/ls /usr/local/bin   make a copy in /usr/local/bin
[sudo] password for me:  type your password, followed by Enter; no echo
$ ls -l ~/bin/ls /usr/local/bin/ls /usr/bin/ls
-rwxr-xr-x 1 me   me   130552 Jul 12 15:11 /home/me/bin/ls
-rwxr-xr-x 1 root root 130552 Mar 12 07:09 /usr/bin/ls
-rwxr-xr-x 1 root root 130552 Jul 12 15:10 /usr/local/bin/ls
$ PATH=/usr/bin:/bin      set path to just /usr/bin and /bin
$ which ls
/usr/bin/ls              found it in /usr/bin
$ PATH=/usr/local/bin:$PATH            insert /usr/local/bin at the front
$ which ls
/usr/local/bin/ls        found it in /usr/local/bin
$ PATH=~/bin:$PATH       insert ~/bin at the front
$ which ls
/home/me/bin/ls          found it in ~/bin
$ rm ~/bin/ls            remove copy from ~/bin
$ which ls
/usr/local/bin/ls        found it in /usr/local/bin
$ sudo rm /usr/local/bin/ls            remove copy from /usr/local/bin
[sudo] password for me:
$ which ls
/usr/bin/ls              found it in /usr/bin
$ echo $PATH
/home/me/bin:/usr/local/bin:/usr/bin:/bin
```

Only two tricky things in this exercise:

- much like we can do in most programming languages, we can define a new value for a variable in terms of its previous value; thus, the expression `PATH=/usr/local/bin:$PATH` causes the shell to append the current value of `PATH` (`$PATH`) to the string `/usr/local/bin:`, and to assign the concatenated string as the new value of `PATH`; and
- there is a command available in Linux, `sudo`, that allows you to become the root user to exercise one command; if you perform `ls -dl /usr/local/bin`, you will see that it is owned by the `root` user, and you do not have write permission to that directory; by using the `sudo` command, you can perform the requested command (first `cp` to create the copy there, and later `rm` to remove the copy) as if you were the root user; the Arch Linux virtual machine knows that your account (`me`) is permitted to use `sudo`, all you need to do is type your password to be able to perform the command.

2.6.2 Input, output, error output

So far, we have focused on typing commands on the keyboard to the shell, the shell parses your commands to determine which program to run, it creates a process to run that program, and the shell makes the arguments and the environment available to that program. The shell then waits for that program to finish before prompting the user for another command. All of our examples so far take some action and output the results to the screen; if a program detects something wrong, it displays an error message on the screen. Computers would be not nearly as useful to us if this was all that could be done.

When a program comes to life, three data channels are defined:

- standard input - this is the default channel from which the program can read data; it is normally the keyboard;
- standard output - this is the default channel to which the program can write results of its processing; it is normally the terminal window; and
- standard error output - this is the default channel to which the program can write error messages; it is normally the terminal window.

The shell sets up these standard channels for a program when it starts it in a process. Thus, the shell is in a position to *redirect* these channels to files or other objects in the system.

2.6.2.1 Input redirection

First, let's focus on taking our input from a file instead of the keyboard; consider the following example.

```
$ cd                       make sure we are home
$ cat shopping.list        reacquaint ourselves
1 bottle of milk
2 granny smith apples
3 hot house tomatoes
1 six-pack of Coca Cola
$ cat                      invoke without a filename
line 1                     you type this
line 1                     cat echoes the line
ctl-d                      you type this to indicate end-of-file
$ cat <shopping.list       what is this?
1 bottle of milk           we obtain the same results
2 granny smith apples
3 hot house tomatoes
1 six-pack of Coca Cola
```

There are several things going on here that make this work:

- when `cat` is invoked, if any arguments are supplied, it copies the contents of those files to the terminal window;
- if invoked without any arguments, it copies the standard input to the terminal window;
- in the first case above where `cat` was invoked without arguments, the standard input was the keyboard; thus, you had to type on the keyboard to provide `cat` with characters that it could write to the terminal window; note that *ctl-d* is the character to type at the keyboard to indicate an end of file;
- in the second case above, the shell interpreted `<shopping.list` to mean that standard input for `cat` should come from `shopping.list`, not from the keyboard; it resets standard input for `cat` to be from the file *and* removes `<shopping.list` from the set of arguments made available to `cat`;
- `cat` doesn't care, as it has been invoked without any arguments, so it simply reads from standard input until an end of file is detected.

Thus we see that `bash` interprets yet another special character, `<`, to indicate that the standard input for the program should come from a file rather than from the keyboard. This redirection occurs without *any* knowledge on the part of the program (`cat` in this case). Correct behavior does demand that the program read from standard input if it has not received any file arguments.[12]

2.6.2.2 Output redirection

Often one would like to capture the output of a program into a file, not have it displayed in the terminal window; in fact, in creating this textbook, a large number of files have been generated in this way in order to show you the actual interaction dialogs that occur on the system. Let's consider the following example:

```
$ echo '$ ls -l'; ls -l   show command and output
$ ls -l
total 24
drwxrwxr-x 2 me me 4096 Jul  6 14:59 book
-rw-rw-r-- 1 me me  141 Jul  6 14:59 calendar.data
drwxrwxr-x 2 me me 4096 Jul  6 14:59 Music
drwxrwxr-x 2 me me 4096 Jul  6 14:59 Pictures
-rw-rw-r-- 1 me me   86 Jul  6 14:59 shopping.list
```

[12] Some programs, such as `cat`, understand an argument consisting solely of a hyphen (`-`) to mean that it should also read from the standard input, as in `cat file1 - file2` - i.e., copy contents of `file1` to the terminal window, then copy the contents from standard input to the terminal window, and finally copy contents of `file2` to the terminal window. The man page for a particular command will indicated if a particular program interprets its arguments in this way.

```
drwxrwxr-x 2 me me 4096 Jul  6 14:59 src
$ echo '$ ls -l' >tmp.out where is the output?
$ ls -l >>tmp.out         and this output?
$ cat tmp.out             maybe it's in here?
$ ls -l                   yes, it is
total 24
drwxrwxr-x 2 me me 4096 Jul  6 14:59 book
-rw-rw-r-- 1 me me  141 Jul  6 14:59 calendar.data
drwxrwxr-x 2 me me 4096 Jul  6 14:59 Music
drwxrwxr-x 2 me me 4096 Jul  6 14:59 Pictures
-rw-rw-r-- 1 me me   86 Jul  6 14:59 shopping.list
drwxrwxr-x 2 me me 4096 Jul  6 14:59 src
```

Thus, we see that there is yet another special character understood by the shell. If it detects an argument of the form `>filename`, it redirects the program's standard output to that file. If it detects an argument of the form `>>filename`, it redirects the program's standard output to the end of that file (i.e., its output is appended to the current contents of the file).

The ability to perform such redirection *again* depends upon each program writing to standard output by default.

It's a bit clunky having to break up our original command line (`echo '$ ls -l'; ls -l`) into separate commands, and appending the output of all but the first command to the file. The shell also understands grouping commands to act as a single "command", such that you can redirect the input or output for the "command" in one go, as in:

```
$ (echo '$ ls -l'; ls -l) >tmp.out group the enclosed commands
$ cat tmp.out
$ ls -l
total 24
drwxrwxr-x 2 me me 4096 Jul  6 14:59 book
-rw-rw-r-- 1 me me  141 Jul  6 14:59 calendar.data
drwxrwxr-x 2 me me 4096 Jul  6 14:59 Music
drwxrwxr-x 2 me me 4096 Jul  6 14:59 Pictures
-rw-rw-r-- 1 me me   86 Jul  6 14:59 shopping.list
drwxrwxr-x 2 me me 4096 Jul  6 14:59 src
```

Parentheses (more special characters) can be used to group the enclosed commands into a single "command", and all of the output, appropriately serialized, can be directed to a file. The way that this is done is that the shell acts as if you had typed the following command:

```
$ bash -c "echo '$ ls -l'; ls -l" >tmp.out
```

i.e., it creates a subshell executing the enclosed commands, with the standard output of that subshell redirected to `tmp.out`. This also shows you that the standard input and standard output of commands executed by a shell in the absence of redirection are the standard input and standard output of the shell, itself.

2.6.2.3 Error redirection

It stands to reason that since we can redirect standard input and standard output, it is likely that we can redirect standard error output, as well. You might think that the right way to do this would be to select another special character, say ☹, and for `bash` to interpret `☹filename` to mean redirect standard error output to `filename`, and to interpret `☹☹filename` to mean to redirect standard error output to the end of `filename`. Unfortunately, `bash` has used up all of the special characters (we have not introduced all of them, yet). Instead, `bash` interprets `2>filename` and `2>>filename` to mean that standard error output should be redirected.

Let's look at some examples of standard error output redirection:

```
$ cat Shoping.list           typo in filename
cat: Shoping.list: No such file or directory
$ cat Shoping.list 2>tmp.err capture the error message in a file
$ cat tmp.err                see if it is there
cat: Shoping.list: No such file or directory
$ cat readme.1st 2>>tmp.err  try another non-existent file
$ cat tmp.err                see if it is there
cat: Shoping.list: No such file or directory
cat: readme.1st: No such file or directory
```

Usually one does not redirect standard error output to a file; instead, we want the error messages to be displayed on the screen so that we can see the error messages as our programs run. Occasionally, especially if a set of commands are all being executed by the shell with the output of the commands redirected to a file, we would also like to see the error messages embedded in the same file so we can see the context in which the errors occur, much as we would (on the screen) if neither output nor error output were redirected. `bash` has a syntax for specifying this, shown below.

```
$ cat shopping.list Shoping.list >tmp.out 2>&1
$ cat tmp.out
1 bottle of milk
2 granny smith apples
3 hot house tomatoes
1 six-pack of Coca Cola
cat: Shoping.list: No such file or directory
```

The non-intuitive expression `2>&1` means redirect standard error output onto the same stream as the standard output (the `1` at the end of the expression). In the same way that standard error output is represented by `2`, standard output is represented by `1`. In fact, we could redirect standard output in the following way:

```
$ cat shopping.list 1>tmp.out
$ cat tmp.out
1 bottle of milk
2 granny smith apples
3 hot house tomatoes
1 six-pack of Coca Cola
```

That is, `digit>` works for each output stream known to `bash`. In this book, we restrict ourselves to standard output (`1`) and standard error output (`2`).

2.6.3 Pipes and multiple processes

In the previous section we described how `bash` performs input, output, and error redirection. Being able to do so would be pretty useless if commands did not read from standard input by default, write results to standard output by default, and write error messages to standard error output by default. Given that programs do conform to this standard, `bash` can now provide significant expressive and computational power by managing multiple processes and enabling these processes to talk to each other *without knowing that they are doing so.* This is done through an abstraction known as *pipelines.*

Suppose you wanted to know the number of files in a particular directory. How would you do so? One has to invoke `ls` on the directory to access its contents. We could make a special version of `ls` that understood a flag `--count`, which would indicate that it would simply print the number of files found in each directory specified on the command line. `--all` could also be specified with this new flag, indicating that hidden files that start with a period (`.`) would also be counted. Seems reasonable so far.

But we already have `wc` which will count lines, words, and characters in a file. If there was some easy way to put the output of `ls` into a file, and then have `wc` take its input from that file, we would have a solution without having to modify `ls`. But we just discussed redirection in the last section, so we can already do this! Let's try it out on the current directory.

```
$ ls .
book  calendar.data  Music  Pictures  Shopping.List  src  tmp.err  tmp.out
$ ls . >tmp.out
$ wc -w <tmp.out          count the words in tmp.out
8
$ rm tmp.out
```

It is clear that this works. There's the problem that we have to choose a temporary filename for the output that does not collide with one of the files already in the directory (which we did *not* do), that the temporary file will be listed by `ls` (and, thus, be counted by `wc`), and that we have to remember to remove the temporary file after we are done.

Operating systems have long supported the ability for processes running programs to communicate with each other (called *interprocess communication (IPC)*, oddly enough☺). Linux supports several different types of IPC that enable different styles of interaction. In this case, we would like to have a way for two child processes[13] to interact, with one process writing data as if to a file and the other process reading that data, as if coming from a file. Linux provides an abstraction in the operating system called a *pipe* which provides this ability; each pipe has a write end and a read end; if one process is given the write end, and another the read end, the two processes interact without knowing that a process is on the other end.

`bash` is already able to redirect input, output, and error output. So, if it know you wanted to plumb your two processes together, it could ask the operating system for a pipe, redirect one process's standard output to the pipe, and the redirect the other process's standard input to the pipe. As you might have guessed, `bash` has another special character that indicates the need for plumbing two processes together.

```
$ ls .
book  calendar.data  Music  Pictures  Shopping.List  src  tmp.err
$ ls . | wc -w
7
```

This works exactly as intended. Note that we removed `tmp.out` in our previous example, so there are only 7 files in the directory.

A vertical bar (`|`) is the special character that separates the communicating programs. `bash` reads everything up to a semicolon or the end of line, then breaks that line up into individual commands separated by `|` characters. It then creates a pipe for each `|` symbol, and redirects standard output for the command to the left of the `|` to the pipe, and the standard input for the command to right of the `|` to the pipe. After starting each of these processes, it waits for each of them to complete.

Note that pipelines are only concerned with standard output and input; they do not affect standard error output. If you need to have both the output and error messages go through a pipe, the previous syntax continues to work:

```
$ cat shopping.list Shoping.list 2>&1 | more
1 bottle of milk
2 granny smith apples
3 hot house tomatoes
1 six-pack of Coca Cola
cat: Shoping.list: No such file or directory
```

[13]In the same way that we discussed parent/child relationships between directories and files, when `bash` creates a process to run your command, it is the process's parent, and the process is `bash`'s child.

A common use of this mixing of output and error messages on a pipe is when you wish to observe the programs in action *and* you want to capture the blended output in a file. The program `tee` copies its standard input to its standard output, as well as to each of the files in its argument list.

```
$ command [arguments, if any] 2>&1 | tee log
```

will cause `command` to run and display its output and error output in the terminal window, as if it was not being piped into `tee`. When the command is finished, `log` will have an exact copy of the output and error messages, in context, as well. This is often used when marking programming projects, as it enables the marker to observe the student's program under test as well as to capture a log to give to the student as part of the assessment.

2.7 Compression and file packaging

It is not uncommon that you need to share an entire directory of files with another user, on a different machine. The mechanism by which such inter-machine sharing is achieved is beyond the scope of this book. The mechanism by which you package such files before you share them is an important aspect of Linux use, so we will cover the basics here.

Consider the directory tree shown on page 19. Here, we will use another standard program, `find`, which will perform commands on all files in a directory tree; the following will print the name of each file found in the tree rooted at `CaDS`:

```
$ find CaDS -print
CaDS
CaDS/book.tex
CaDS/ch00
CaDS/ch00/Preface.pdf
CaDS/ch00/Preface.synctex.gz
CaDS/ch00/Preface.tex
CaDS/ch01
CaDS/ch01/ch01.pdf
CaDS/ch01/ch01.synctex.gz
CaDS/ch01/ch01.tex
CaDS/ch02
CaDS/ch02/adm3a.jpg
CaDS/ch02/ch02.pdf
CaDS/ch02/ch02.synctex.gz
CaDS/ch02/ch02.tex
CaDS/ch02/dir.out
CaDS/ch02/me
CaDS/ch02/me/20170706-shopping.list
```

```
CaDS/ch02/me/book
CaDS/ch02/me/calendar.data
CaDS/ch02/me/Music
CaDS/ch02/me/Pictures
CaDS/ch02/me/shopping.list
CaDS/ch02/me/src
CaDS/ch02/me.tgz
CaDS/ch02/model33teletype.jpg
CaDS/Outline.pdf
CaDS/Outline.synctex.gz
CaDS/Outline.tex
```

Definitely a less informative representation than the figure on page 19 ☺.

There are a number of ways that we could share each of these files with other individuals (or ourselves) on another machine that does not share this file system: by attaching each file to an email message, by using a network file transfer program (such as `scp` or `ftp`), or by copying each file to a cloud storage provider (such as DropBox, Apple's iCloud, or Microsoft's OneDrive). While this one-file-at-a-time approach would work, it does not make it easy to share all the files at once in an easy and consistent way. The better way would be to make a new file that contains the contents of all of the other files along with meta-information about each contained file so that the individual files can be extracted at the other end. You have probably used ZIP files for such things in your previous computer experience.

2.7.1 `tar`

The `tar` program packages many files together into a single disk file (often called an *archive*), and can restore individual files from the archive. `tar` is named after **t**ape **ar**chive, as it was initially created to move files to/from magnetic tapes.

Let's create an archive of the files in the `CaDS` directory:

```
$ tar -cvf CaDS.tar CaDS | column
CaDS/                                   CaDS/ch02/dir.out
CaDS/book.tex                           CaDS/ch02/me/
CaDS/ch00/                              CaDS/ch02/me/20170706-shopping.list
CaDS/ch00/Preface.pdf                   CaDS/ch02/me/book/
CaDS/ch00/Preface.synctex.gz            CaDS/ch02/me/calendar.data
CaDS/ch00/Preface.tex                   CaDS/ch02/me/Music/
CaDS/ch01/                              CaDS/ch02/me/Pictures/
CaDS/ch01/ch01.pdf                      CaDS/ch02/me/shopping.list
CaDS/ch01/ch01.synctex.gz               CaDS/ch02/me/src/
CaDS/ch01/ch01.tex                      CaDS/ch02/me.tgz
CaDS/ch02/                              CaDS/ch02/model33teletype.jpg
```

```
    CaDS/ch02/adm3a.jpg                      CaDS/Outline.pdf
    CaDS/ch02/ch02.pdf                       CaDS/Outline.synctex.gz
    CaDS/ch02/ch02.synctex.gz                CaDS/Outline.tex
    CaDS/ch02/ch02.tex
```

The flags to `tar` that we have used have the following meanings: `-c` means create an archive, `-v` means write the name of each file as it is added, and `-f filename` means to create the archive in `filename`. As you can see, `tar` allows you to collect all flags into a single argument; since `f` is included in the flag argument, the name of the archive file to be created must immediately follow `-cvf`. The filename arguments for inclusion can either be regular files, or the name of a directory; in the latter case, all files contained in the directory are included in the archive; if an included file is a directory, then its contents are also included in the archive. Note that we have used another Linux command, `column`, to pack the list of file and directory names into columns across the screen to more efficiently use the vertical space in the book; if we had not piped the output of `tar` into `column`, each filename would have appeared on a single line.

Suppose I have received `CaDS.tar` from someone, and it is currently stored in `/home/jsven`. To check the contents of the archive, I can say

```
    $ tar -tf CaDS.tar | column
    CaDS/                                    CaDS/ch02/dir.out
    CaDS/book.tex                            CaDS/ch02/me/
    CaDS/ch00/                               CaDS/ch02/me/20170706-shopping.list
    CaDS/ch00/Preface.pdf                    CaDS/ch02/me/book/
    CaDS/ch00/Preface.synctex.gz             CaDS/ch02/me/calendar.data
    CaDS/ch00/Preface.tex                    CaDS/ch02/me/Music/
    CaDS/ch01/                               CaDS/ch02/me/Pictures/
    CaDS/ch01/ch01.pdf                       CaDS/ch02/me/shopping.list
    CaDS/ch01/ch01.synctex.gz                CaDS/ch02/me/src/
    CaDS/ch01/ch01.tex                       CaDS/ch02/me.tgz
    CaDS/ch02/                               CaDS/ch02/model33teletype.jpg
    CaDS/ch02/adm3a.jpg                      CaDS/Outline.pdf
    CaDS/ch02/ch02.pdf                       CaDS/Outline.synctex.gz
    CaDS/ch02/ch02.synctex.gz                CaDS/Outline.tex
    CaDS/ch02/ch02.tex
```

The `-t` command to `tar` indicates that I want to see a table of contents. Again, we have used `column` to pack the output into columns.

If I execute the following:

```
    $ tar -tvf CaDS.tar CaDS/ch02/ch02.tex
    -rw-r--r-- me/me   94095 2017-07-14 11:28 CaDS/ch02/ch02.tex
```

I see a verbose listing about `CaDS/ch02/ch02.tex`. It looks very similar to the output of `ls -l`, in that it shows each file's protections, owner/group, size, modification date, and its name.

If I want to extract all of the files into my home directory, I would say the following:

```
$ tar -xf book.tar
```

I can check to see that it has worked by executing

```
$ ls CaDS
book.tex  ch00  ch01  ch02  Outline.pdf  Outline.synctex.gz  Outline.tex
```

Sometimes you want to extract a particular file onto the standard output. This can be achieved using the following command:

```
$ tar -xOf CaDS.tar CaDS/ch01/ch01.tex >chapter1.tex
```

There are many other options supported by `tar`. See `tar(1)` for more information.[14]

2.7.2 Compression

Files on a computer system often have a significant amount of redundancy in them, such that they occupy more space than is theoretically required to represent the contained information. Linux provides tools for performing two styles of compression/inflation:

- The tools `compress`, `uncompress`, and `zcat` use an adaptive Lempel-Ziv coding to remove the redundancy. `compress` encodes the content of a file using adaptive Lempel-Ziv coding; `uncompress` and `zcat` decode an encoded file, producing the original file.
- The tools `gzip`, `gunzip`, and `gzcat` use Lempel-Ziv coding (LZ77) to remove the redundancy. `gzip` encodes the content of a file using Lempel-Ziv coding; `gunzip` and `gzcat` decode an encoded file, producing the original file.

The default behavior of `compress` and `gzip` is to replace each file argument by an encoded file with an extension of `.Z` or `.gz`, respectively, while keeping the same ownership modes, access, and modification times. For example,

```
$ cp shopping.list sl.1        make a couple of files to compress
$ cp shopping.list sl.2
$ gzip sl.1                    Encode sl.1 to sl.1.gz, remove sl.1
```

[14]This indicates that you should look at the man page for `tar` in section 1 of the Linux users manual. This can be achieved using a browser, as indicated earlier in the chapter, or by typing `man 1 tar` to the shell.

```
$ compress sl.2               Encode sl.2 to sl.2.Z, remove sl.2
```

The default behavior of `uncompress` and `gunzip` is to replace each file argument with an extension of `.Z` or `.gz`, respectively, by an unencoded file with the `.Z` or `.gz` removed from the name while keeping the same ownership modes, access, and modification times. For example,

```
$ gunzip sl.1.gz              Unencode sl.1.gz to sl.1, remove sl.1.gz
$ uncompress sl.2.Z           Unencode sl.2.Z to sl.2, remove sl.2.Z
```

The default behavior of `zcat` and `gzcat` unencode each file argument with an extension of `.Z` or `.gz`, respectively, and write the unencoded content to standard output. For example,

```
$ gzcat sl.1.gz               Unencode sl.1.gz, writing contents to standard output
$ zcat sl.2.Z                 Unencode sl.2.Z, writing contents to standard output
```

The most common use of these types of tools are for very large data files. The principle use of compression technologies is in conjunction with file packaging tools for files that are to be exchanged. These are described in the next section

2.7.3 Compression and File Packaging

2.7.3.1 gzip and tar

We described the use of `tar` in section 2.7.1 to create `tar` archives and to extract files from `tar` archives. We also described in section 2.7.2 the use of `gzip` to compress files. Therefore, you can easily create a compressed `tar` archive by using `tar` and `gzip` as follows:

```
$ tar -cf CaDS.tar CaDS; gzip CaDS.tar
```

After completing these commands, you will be left with a file `CaDS.tar.gz`, and `CaDS.tar` will have been deleted. To access the files in the compressed archive, you will need to execute

```
$ gunzip CaDS.tar.gz; tar -tf CaDS.tar
```

Remember that the `gunzip` invocation deletes the `CaDS.tar.gz` file.

While this approach works, it suffers from several deficiencies:

- The constant conversion from uncompressed to compressed and back again represents significant wasted computational resources.
- During `gzip` and `gunzip` processing, both uncompressed and compressed versions of the `tar` archive are resident on the disk.
- If the long-term stored form of the `tar` archive is the compressed form, having to convert back to uncompressed for *any* access is unintuitive, at best.

Fortunately, `tar` can compress and uncompress as part of its processing. The `-z` flag tells `tar` to create a compressed archive during creation and file addition, and to uncompress the data in a compressed archive when extracting files or listing its contents.

The commands:

```
$ tar -zcf CaDS.tar.gz CaDS
$ tar -ztf CaDS.tar.gz
$ tar -zxf CaDS.tar.gz CaDS/ch01/ch01.tex
$ tar -zxOf CaDS.tar.gz CaDS/ch01/ch01.tex >chapter1.tex
```

creates a gzipped `tar` archive containing the files in the `CaDS` directory, lists the table of contents of the archive, extracts a particular file from the archive into the current working directory, and extracts a particular file from the archive onto standard output, respectively.

Gzipped `tar` archives are so prevalent in Linux systems that such an archive is usually named with a `.tgz` extension, as in

```
$ tar -zcf CaDS.tgz CaDS
```

We will use the `.tgz` extension in the rest of the book when we have need for a gzip'ed `tar` archive.

2.7.3.2 `zip` and `unzip`

While use of `tar` to create compressed archives is the predominant method on Linux for creating compressed packages of files, there is another method which you may find useful, especially if you are exchanging packages with non-Linux systems. If you have encountered ZIP files in your previous computer use, you will know that a ZIP archive is similar to a compressed `tar` archive; the contents have been compressed such that some of the redundancy in the contained files has been *squeezed* out in the archive. To extract one or more files from the ZIP archive, you need a program that can also uncompress the data as it is extracted.

Linux provides the `zip` and `unzip` commands[15] for creating ZIP archive files and extracting files from a ZIP archive, respectively. The following dialog shows use of `zip` and `unzip` to create, list, extract to standard output, and extract the contents of an archive `example.zip`. It assumes that we have a directory named `tmp` in the current working directory.

```
$ ls tmp
cat   cat.c   tento6.txt
$ zip example.zip tmp/*
  adding: tmp/cat (deflated 70%)
  adding: tmp/cat.c (deflated 36%)
  adding: tmp/tento6.txt (deflated 80%)
$ unzip -l example.zip
Archive:  example.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
     8710  2016-08-30 12:44   tmp/cat
      327  2016-08-30 12:43   tmp/cat.c
 46301948  2016-08-30 12:44   tmp/tento6.txt
---------                     -------
 46310985                     3 files
$ unzip -p example.zip tmp/cat.c >mycat.c
$ unzip -o example.zip
Archive:  example.zip
  inflating: tmp/cat
  inflating: tmp/cat.c
  inflating: tmp/tento6.txt
```

This creates `example.zip` containing all of the files in `tmp` using the adaptive Lempel-Ziv encoding as used in `compress`. As each file is added, `zip` prints the size of the file, the compressed size, and the percentage of compression on standard output. The `unzip -l` command enables you to determine the contents of an archive. The `unzip -p` command enables you to extract a member of the archive to standard output; in this case, we redirect standard output to `mycat.c`. Finally, to extract the entire contents of an archive, you invoke `unzip archive-name`; the files are extracted into the current working directory; if a filename includes a directory name (e.g., `tmp/cat.c`), the file will be extracted into that directory; the directory will be created if it does not exist. If a file already exists, `unzip` will prompt you about each file unless you have specified the `-o` flag, which indicates overwrite existing files without prompting.

[15]The Arch Linux image described in the appendices does not have `zip` or `unzip` installed. If you need to use it, you need to execute `sudo pacman -S zip` and answer the question posed in the affirmative.

2.8 Summary

This chapter has introduced you to the Linux system as experienced by a user.

It started off discussing the basic features of the shell, how commands are initiated and provided arguments, and presented a number of basic commands that will enable you to get started using the system. As we discuss C programming in a later chapter, we will introduce additional programs that are needed to develop software written in C for use on Linux (and other Posix-compliant systems).

We then moved on to discuss the major aspects of the file system, and the most important directories in the file system for a software developer.

Armed with knowledge of the file system, we then returned to additional features of the shell that enable sophisticated use of the shell.

Finally, we described how to create files (archives) that contain other files for sharing with other users, both in uncompressed and compressed formats.

The C Programming Language

This section of the book focuses on C programming on Linux, tools provided to facilitate that programming, and a well-structured approach to constructing Abstract Data Type implementations in C. We assume that you have sound knowledge of imperative and object-oriented programming in Python, albeit using an Integrated Development Environment (IDE) like IDLE or PyCharm.

We first review the primary elements of Python programming with which you should be comfortable. Then we describe the program development cycle used in Linux to develop programs written in C and the overall structure of a C program. Then we delve into aspects of C necessary to understand and exploit basic data structures, the ultimate goal of the book. This is followed by a discussion of two important tools to support C program development: `gdb` and `valgrind`. We finish by describing a well-structured approach to constructing Abstract Data Type implementations in C.

Chapter 3

A review of Python Programming on Linux

Why review Python? Since we assume you are comfortable with Python, it is often easiest to acquire a new programming language by being able to compare and contrast it with a language that you already understand. Thus, this chapter will focus on the primary elements of Python to enable comparison with C.

Note that we are focused on Python v3.* - all examples and code fragments follow that language standard.

3.1 Built-in Types

With most languages, one distinguishes between three different kinds of data types:

- built-in primitive types;
- built-in structured types; and
- programmer-defined structured types.

This section is devoted to the built-in types; programmer-defined structured types will be covered in a later section.

3.1.1 Built-in primitive types

Python provides the following built-in primitive types:

- the *integer* type (`int`) is used for representing the set of values `{..., -3, -2, -1, 0, 1, 2, 3, ...}`;

- the *floating point* type (`float`) is used to represent real numbers; the numbers are stored as a mantissa and an exponent (scientific notation); in most cases, the representation is an approximation of the actual real number due to the number of bits devoted to the mantissa and exponent;
- the *boolean* type (`bool`) represents the truth values `True` and `False`.

Python also defines a *complex* type to represent complex numbers; since C does not have a built-in equivalent of that type, we will ignore it going forward.

The following operators are defined on the numeric types:

`x + y`	sum of x and y
`x - y`	difference of x and y
`x * y`	product of x and y
`x / y`	quotient of x and y
`x // y`	integer division (floored quotient of x and y)
`x % y`	remainder of (x // y)
`-x`	x negated
`+x`	x unchanged
`abs(x)`	absolute value/magnitude of x
`int(x)`	x converted to integer
`float(x)`	x converted to floating point
`x ** y`	x to the power y

The following comparison operators, when used with numeric types, generate boolean values:

`x < y`	x is strictly less than y
`x <= y`	x is less than or equal to y
`x > y`	x is strictly greater than y
`x >= y`	x is greater than or equal to y
`x == y`	x is equal to y
`x != y`	x is not equal to y

Boolean values can be combined using the usual boolean operators:

`x or y`	if x is `False`, then y, else x	y is only evaluated if x is `False`
`x and y`	if x is `False`, then x, else y	y is only evaluated if x is `True`
`not x`	if x is `False`, then `True`, else `False`	has lower priority than non-Boolean operators, so `not a == b` is interpreted as `not (a == b)`

3.1.2 Built-in structured types

Python has a rich collection of built-in structured types; many of these built-in types are implementations of some of the data structures we will be covering later in the textbook.

For the time being, the only built-in structured type that has a loose equivalence to C is the `list` type and its correspondence to the array structured type of C. Even then, the only true correspondence is the use of `[i]` as an index into the list/array, to access the i^{th} element of the list/array, with the origin index being `0`.

Python also includes a built-in *string* type (`str`), which is an immutable sequence of Unicode code points. Elements of the string can be accessed by indexing, but the character at an index cannot be changed.

3.2 Variables, Block Structure, and Scoping

3.2.1 Variables

Variables are named memory locations; you can assign a value to a location, and you can later access the value stored in that location. Python imposes three rules on the characters that make up a variable name:

- a variable name must start with a letter or an underscore;
- the remaining characters of the variable name may consist of letters, numbers, and underscores; and
- variable names are case sensitive.

You can assign a value to a variable, and reference its value, using the following syntax:

```
my_variable = 42
print('The meaning of life, the universe, and everything is', my_variable)
```

The print command will print out the string argument, a space, and the string representation of `my_variable` on the standard output.

You do not have to declare variable names, or the type of data that a variable can hold. The first line above caused `my_variable` to come into existence, and assigned `42` to it.

3.2.2 Block Structure

A Python program is constructed from code blocks.[1] A *block* is a piece of Python program text that is executed as a unit. The following are blocks: a module, a function body, and a class definition. Additionally, a script file (a file given as standard input to the interpreter or specified as a command line argument to the interpreter) is a code block.

Names refer to objects. Names are introduced by name binding operations.

[1]Most of the material in this section is derived from the Python Reference Model, `https://docs.python.org/3/reference/executionmodel.html`, section 4 entitled "Execution model".

The following constructs bind names: formal parameters to functions, `import` statements, class and function definitions (these bind the class or function name in the defining block), and targets that are identifiers (variable names) if occurring in an assignment, a `for` loop header, or after `as` in a `with` statement or `except` clause. An `import` statement of the form `from ... import *` binds all names defined in the imported module, except those that begin with an underscore.

If a name is bound in a block, it is a local variable of that block, unless declared as `nonlocal` or `global`. If a name is bound at the module level, it is a global variable. If a variable is used in a code block but not defined there, it is a *free variable*.

Each occurrence of a name in the program text refers to the *binding* of that name established by the following name resolution rules.

3.2.3 Scoping

A *scope* defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. If the definition occurs in a function block, the scope extends to any blocks contained within the defining one, unless a contained block introduces a different binding for the name.

When a name is used in a code block, it is resolved using the nearest enclosing scope. The set of all such scopes visible to a code block is called the block's *environment*.

If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block.

The namespace for a module is automatically created the first time a module is imported. The main module for a script is always called `__main__`.

3.3 User-defined types

Python enables the creation of user-defined types through the use of classes. Python classes provide all the standard features of Object-Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data.

Class members (including the data members) are *public*; even the name-mangling provided for member names that start with `__` doesn't truly hide those members from direct access by the user.

Since data members are public, there is no need to define methods in a class, as the data members can be accessed/modified directly by the application. Such “classes” act like records or structures in other languages.

3.3.1 Function Annotations

PEP 3107[2] defines a syntax for adding arbitrary metadata annotations to Python functions. These annotations have the following attributes:

- function annotations, both for parameters and return values, are completely optional;
- function annotations are nothing more than a way of associating arbitrary Python expressions with various parts of a function at compile-time; and
- Python does not attach *any* particular meaning or significance to annotations.

The syntax for function parameters take the form of optional expressions that follow the parameter name, as in:

```
def foo(a: expression, b: expression = 5):
    . . .
```

In pseudo-grammar, parameters now look like `identifier [: expression] [= expression]` - i.e., annotations always precede a parameter's default value and both annotations and default values are optional. Similar to the use of equal signs to indicate a default value, colons are used to mark annotations.

The syntax for the type of a function's return value is as follows:

```
def sum() -> expression:
    . . .
```

That is, the parameter list can now be followed by a literal `->` and a Python expression.

Why are we introducing function annotations here? Even though they are optional, it provides a mechanism for indicating in the function signature the types of the parameters and return values for functions and methods. Even though this is typically documented in the function/method docstring, the proximity of the annotations to the parameters in the function definition are a boon to someone reading your code. And, as we shall see, such type information is required by languages such as C.

3.4 Scripts - invoking Python programs from the command line

We will be building executable programs from C source files that can be invoked by typing commands to `bash`. Since we assume that your Python experience has primarily been

[2] `https://www.python.org/dev/peps/pep-3107/`

using an IDE, at this juncture we need to discuss how one invokes a Python program from the command line, accessing the arguments provided by the shell.

3.4.1 A simple example

There are a number of infinite series expressions for π. One in particular is

$$\pi \;=\; \sqrt{12}\,\sum_{i=0}^{\infty}\frac{(-1)^i}{((2i+1)\;3^i)}$$

Let's write a python program, `approx_pi.py`, that approximates π by truncating this sum after a fixed number of terms; this fixed number is provided as an argument to the program. Besides computing the approximation, the program also indicates the percentage error of the approximation relative to the value from `math.pi`.

```
"""
approx_pi.py:  Approximate pi by truncating infinite series
"""
import math          # needed for sqrt(), fabs() and pi
from sys import argv # needed to access arguments

def approx_pi(terms:int) -> float:
    """
    Generate an approximation to pi by truncating infinite series
    args: terms: number of terms from the infinite series
    returns: approximate value of pi
    """
    sum = 0.0
    for i in range(terms):
        sum += (-1)**i/((2*i+1)*3**i)
    return math.sqrt(12) * sum

def main() -> None:
    argc = len(argv)
    if argc == 2:
        terms = int(argv[1])
        pi = approx_pi(terms)
        diff = math.fabs(pi - math.pi)
        print('pi[{}] = {}, error = {:.5%}'.format(terms, pi, diff/math.pi))
    else:
        print("usage: python3 approx_pi.py terms")

if __name__ == "__main__":
    main()
```

What are the main points to note in this simple program?

- We need to import `argv` from `sys` in order to access arguments passed by the shell.[3]
- `argv` is simply a list of strings; `argv[0]` is the name of the script that was invoked (`approx_pi.py` in this case); thus, `argv[1]` is the number of terms in the summation to use.
- If the script is invoked as `python3 approx_pi.py 5`, then the global variable `__name__` has the string value `"__main__"`; if this is the case, then the script invokes `main()`, causing it to obtain the argument and invoke `approx_pi()`; if another program has `import`ed `approx_pi`, then the functions will be defined, but `main()` will not be invoked.

Thus, this is the way to write a module so that it can be invoked directly by the shell, and can also be `import`ed by other modules.[4]

3.5 File I/O, and standard input, output, and error

3.5.1 File I/O

Besides accessing command arguments, your Python program will invariably need to read data from files, and write data to files. Python defines a built-in `File` class; an object of that class is returned whenever you invoke the built-in `open()` function. `open()` is used in the following ways:

- open an existing file for reading: `fr = open(filename, 'r')`
- open a file for writing: `fw = open(filename, 'w')`
- open a file for writing at the end: `fw = open(filename, 'a')`

In the latter two cases, if `filename` does not already exist, it is created. If an existing file is opened at `'w'` access, its contents are overwritten.

Once a file is open for reading, one can either read a fixed number of characters (`fr.read(size)`), the entire file (`fr.read()`), or a single line (`fr.readline()`); the return value from these calls is the string that was read. Once a file is open for writing, one can write a string to the file (`fw.write(string)`); if you wish to write data that is not already a string, you must convert it to a string before calling `write()`. The return value from the `write()` call is the number of characters written.

Calls to `print()` generate `write()` calls on standard output.

[3]To be sure, there are better ways to obtain arguments, for example using `argparse` in Python; we have focused on `sys.argv` due to its similarity to argument access in C.

[4]If one simply wanted to write a module that would *always* be invoked from the shell, one could dispense with the test of `__name__` and the definition of `main()`; use of the structure described above prevents premature optimization of your Python code and is more similar to the structure of a C program.

3.5.2 Standard input, output, and error

When your Python program comes to life, three of these file objects have already been created for you:

- `sys.stdin` - standard input
- `sys.stdout` - standard output
- `sys.stderr` - standard error output

As you can see, these objects are available from the `sys` module. `sys.stdin` has been opened for reading, while `sys.stdout` and `sys.stderr` have been opened for writing. By default, reading from `sys.stdin` causes characters to be read from the keyboard. Of course, if standard input has been redirected in the shell, as in

```
$ python3 xyz.py <file.data
```

calls to `sys.stdin.readline()` will read from `file.data`.

In a similar fashion, writing to `sys.stdout` or `sys.stderr` causes characters to be written to the terminal window, unless standard output (and/or standard error output) have been redirected in the shell, as in

```
$ python3 xyz.py >file.data
```

calls to `sys.stdout.write()` will write to `file.data`.

Chapter 4

C programming on Linux

4.1 The edit-compile-link-execute (ECLE) cycle

Python is an interpreted language. You present blocks written in Python to the interpreter; it interprets the code, executing as it goes along.

C and many other programming languages are *not* interpreted. A programmer using these languages must perform the following steps:

- use an **e**ditor or other tools to generate one or more files containing program fragments in that language; these are called *source* files;
- **c**ompile/transform each of these files from source to a more binary representation; the tool used to perform this transformation is called a *compiler*, and the binary representation for each source file is called an *object* file;
- **l**ink together each of the object files, along with any code needed from system libraries, into a single executable program file; and, finally,
- **e**xecute the program file, with suitable arguments and inputs, to see if it performs correctly.

If the program does not execute correctly, you will need to edit one or more of the source files, compile, link, and execute again. Do this until the program executes correctly; thus, the *ECLE* cycle.

4.1.1 `gcc` - the C compiler on Linux

The GNU Compiler Collection is a compilation system that enables you to compile source files written in C, C++, Objective-C, Fortran, Ada and Go. This system has been adopted as part of Linux.

`gcc` is the name of the C compiler on Linux. `gcc` has a large number of flags, only a few of which we will cover in this book.

Suppose we have a source file named `prog.c` that we have created using our favorite editor. Note that, by convention, C source files have a `.c` extension. Now suppose that we wish to compile `prog.c` into a binary object file. The command needed to do so is:

```
    $ gcc -c prog.c
```

The `-c` flag tells `gcc` to compile only (`gcc` also performs the link task); if there are no errors in `prog.c`, `gcc` will leave the compilation results in a file named `prog.o`. Note that `.o` is the conventional extension for an object file.

`gcc` can also link object files into an executable program. Assuming that `prog.c` contains all of the user-defined logic needed, then the following command will produce an executable program in a file named `prog`:

```
    $ gcc -o prog prog.o
```

Finally, you can execute your program by typing the following command:

```
    $ ./prog
```

You may wonder why you have to type `./prog`. Recall that `bash` uses the `PATH` environment variable to find the executable program file corresponding to the command you have typed. If you simply typed `prog`, `bash` would look in each of the directories found in `PATH`. By typing a command with a slash(`/`) in it, `bash` does not search through those directories, instead simply executing the file as typed. And, as we recall from the section 2.5.1, the directory `'.'` simply means the current directory, so `./prog` will cause the executable file named `prog` in the current directory to be executed, and the `/` in the name turns off the search by `bash`. Of course, you could add `.` to the front of `PATH`, but this can sometimes cause trouble if you give your program an identical name to one of the standard programs provided in Linux.

Having to invoke `gcc` twice to build `prog` may seem like one invocation too many. In fact, for simple programs like `prog.c`, it can all be done in one command, as in:

```
    $ gcc -o prog prog.c
```

The presence of a `.c` file in the argument list causes `gcc` to first compile it into an object file, to then link that object file into the executable program, *and* to then delete the object file.

Most programs will consist of multiple source files, each providing different functionality (like different modules used in a Python program). As you use the ECLE cycle to debug your program, you seldom change all of the source files at once; usually, you need to change one source file to resolve the current bug that the program is exhibiting; this only requires that you compile that particular source file again, and then link all of the object files together into your executable program file.

If your C source file has language errors, `gcc` will report these errors on standard error output. There are a large number of flags to `gcc` that control the reporting of warnings regarding your usage of the C language; some of these warnings are indicative of poor programming practices that will likely lead to your program executing incorrectly. Decades of experience recommend that you specify `-W -Wall` as compilation flags when compiling your source files, as these warning flags will do a reasonable job of reporting such poor programming practices. Therefore, we strongly recommend that your compile command lines look like:

```
$ gcc -c -W -Wall prog.c
```

The next section describes a tool to help you with the ECLE cycle, automating the necessary recompiles and the relink of your program after you have made changes to source files.

4.1.2 `make` - a tool to help you with the ecle cycle

It should be clear that one can establish a set of dependencies between the files that make up your executable program. The executable program file depends upon the constituent object files; if one of them changes, then the program file must be regenerated by relinking the object files. Likewise, each object file depends upon its source file; if the source file has changed, then the object file must be recreated by recompiling the source file; since the object file is now changed, the executable program file must be recreated. Clearly, a tool that keeps track of these (and other) dependencies, and what actions are required to recreate a file that depends upon a file that has changed will be extremely useful. `make` is such a tool.

`make` requires that you specify the dependencies between your files. It also requires that you specify what action to take to recreate a dependent file when one of the files upon which it depends has changed. Finally, it has built in rules for the most common types of actions (e.g., going from a `.c` to a `.o`).

How can `make` possibly work? Recall from Chapter 2 that the file system keeps metadata about each file, and that one such piece of metadata was the modification date/time for the file. Given a specification of the dependencies between your files, `make` can determine if a dependent file needs to be recreated if the modification date of any of the files upon which the dependent file depends has a later modification date than the dependent file. Upon detection of such a situation, it then applies the action that you have specified for recreating the dependent file (or applies one of the built-in, implicit rules) to recreate the dependent file, which will, of course, now have a modification date later than any of the files upon which it depends.

By convention, we keep all source files for a related set of programs in a separate directory. In that directory, a file named `Makefile` contains the specification of the file dependencies, and the actions that should be taken. Let's look at an example `Makefile` for `prog`:

```
CFLAGS=-W -Wall
OBJECTS=prog.o

prog: $(OBJECTS)
	gcc -o prog $(OBJECTS)

prog.o: prog.c
```

What does this mean? Let's look at each line in turn.

- `CFLAGS=-W -Wall`
  The built-in rules for converting from a C source file to a C object file knows to look for a variable named `CFLAGS`; if this is defined, it will use it in the `gcc` command to compile a source file into its object file.
- `OBJECTS=prog.o`
  It is good practice to create a variable named `OBJECTS` for all of the object files that must be linked together to create our program. In this case, there is only one; if there had been other files, the entire set would have been listed as part of the definition of `OBJECTS`, with the files separated by blanks.

-
  ```
  prog: $(OBJECTS)
  	gcc -o prog $(OBJECTS)
  ```
  The first line says that `prog` depends upon the definition of `OBJECTS`; since `OBJECTS` is defined as `prog.o`, this means that `prog` depends upon `prog.o`. `make` checks the modification date for `prog.o`; if the file does not exist, it looks for a rule in the Makefile to see how to create `prog.o`, and executes that rule. Armed with the modification date for `prog.o`, it compares it to the modification date for `prog`, and if it is newer, executes the second line, which is the rule used to update `prog`. Note that the action lines must be indented by a TAB character; there can be multiple action lines, each indented by a TAB; an empty line indicates the end of the action lines associated with that rule.
- `prog.o: prog.c`
  This indicates that `prog.o` depends upon `prog.c`. The absence of a rule for updating `prog.o` tells `make` to use its built-in rule.

The lines starting with `prog:` and `prog.o:` define *targets*. You can ask `make` to "make" one of the targets defined in a `Makefile` by typing the following command to `bash`:

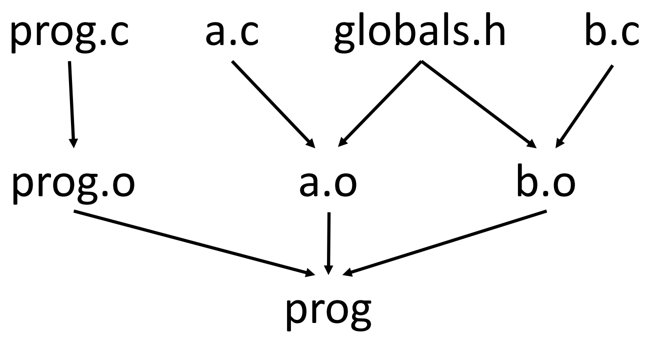
```
$ make name_of_target
```

In our particular case, we could type any of the following commands:

```
$ make prog.o
$ make prog
$ make
```

The first command asks `make` to compile `prog.c` to produce `prog.o` if `prog.c` is newer than `prog.o`. The second command asks `make` to link `prog.o` to produce `prog` if `prog.o` is newer than `prog`; as a side effect, if `prog.c` is newer than `prog.o`, `make` would compile `prog.c` first, and then would have to recreate `prog`, since `prog.o` has to be newer than `prog` in that case. The third command asks `make` to produce the first target that it finds, top to bottom, in `Makefile`; in this case, it would recreate `prog`.

If all of our programs were just a single source file, like `prog`, `make` might seem like overkill. Usually your programs will consists of several source files, and need to link to special libraries; in such cases, capturing the knowledge of the dependencies between files and the actions to take to update a target if it is older than any of the files upon which it depends in a `Makefile`, and using `make` to make sure our targets are up-to-date is a definite boon. Let's assume that the code in `prog.c` refers to functions defined in `a.c` and `b.c`. Furthermore, let's assume that some global definitions needed by `a.c` and `b.c` are in `globals.h`.[1] This yields the following dependency graph between source files, object files, and program files.



The following `Makefile` captures this dependency graph along with any non-built-in rules needed to update a target if it is older than any of its dependencies:

```
CFLAGS=-W -Wall
OBJECTS=prog.o a.o b.o

prog: $(OBJECTS)
        gcc -o prog $(OBJECTS)
```

[1]We will see later how source code can "include" information from another file, thus establishing a dependency of the compiled object file upon the "included" file.

```
prog.o: prog.c
a.o: a.c globals.h
b.o: b.c globals.h
```

What has changed from our previous `Makefile`? We have augmented the definition for `OBJECTS` to include `a.o` and `b.o`, and we have added dependencies for these two additional object files.

Let's summarize what we have learned so far:

- we store all of the source, object, and program files for a particular program in its own directory;
- we create a file named `Makefile` that captures the dependencies between source, object, and program files, and provides custom rules needed to update a target if it needs to be updated;
- we define a variable named `CFLAGS` at the top of the makefile to specify some of the command arguments needed when compiling a C source file to its object file;
- we define a variable named `OBJECTS` at the top of the makefile to collect together the object file names needed to build our program;
- the first target shows the dependency of our program file upon the objects, and provides the appropriate `gcc` command to link those together to create our program file;
- this is followed by a dependency line for each object file, showing the dependency upon the relevant source file *and* any files "included" by that source file;
- we do not provide rules to perform the compilation, instead relying upon the built-in rule within `make` for compiling C source files.

There are many other things one can specify in the `Makefile` which are beyond the scope of this book. This final section describes a couple of the more common things you will see in makefiles, and which you are encouraged to add to your makefiles. As with our previous discussion, we will motivate the discussion with an example `Makefile`.

```
CFLAGS=-W -Wall
OBJECTS=a.o b.o
PROGRAMS=prog1 prog2

all: $(PROGRAMS)

prog1: prog1.o $(OBJECTS)
        gcc -o prog1 prog1.o $(OBJECTS)

prog2: prog2.o $(OBJECTS)
        gcc -o prog2 prog2.o $(OBJECTS)

prog1.o: prog1.c
prog2.o: prog2.c
```

```
a.o: a.c globals.h
b.o: b.c globals.h

clean:
        rm -f prog1.o prog2.o $(OBJECTS) $(PROGRAMS)
```

This is a slightly more complicate use case - we have two programs, `prog1` and `prog2`, that use functions defined in `a.c` and `b.c`. It makes sense to build both programs in the same directory. Thus, we have made the following changes to the previous `Makefile`:

- `OBJECTS=a.o b.o`
  we have removed `prog.o` from the variable `OBJECTS`, as we only want to capture here those object files needed by both programs;
- `PROGRAMS=prog1 prog2`
  we have added a variable named `PROGRAMS` in which we can record all of the programs that can be built by this `Makefile`; we didn't have to do this, but it is good practice, especially as in the future there is every likelihood that you will add one or more new programs to this set;
- `all: $(PROGRAMS)`
  we define the first target to be dependent upon all programs *without* an update rule; if you type `make` or `make all`, this will cause `make` to check that all of the programs are up-to-date, and if not, bring them up-to-date;

-
  ```
  prog1: prog1.o $(OBJECTS)
          gcc -o prog1 prog1.o $(OBJECTS)
  ```
  the old dependency and rule for `prog` is modified to build `prog1`; note that since we removed `prog.o` from `OBJECTS`, we have to include `prog1.o` in the dependency line and the rule; we also add an equivalent target and rule for `prog2`;

-
  ```
  prog1.o: prog1.c
  prog2.o: prog2.c
  ```
  we obviously need to replace the old target for `prog.o` by equivalent targets for `prog1.o` and `prog2.o`;

-
  ```
  clean:
          rm -f prog1.o prog2.o $(OBJECTS) $(PROGRAMS)
  ```
  finally, it is common to add a target named `clean`; when you invoke `make clean`, it executes the `rm` command to remove all of the object files and the program files, leaving only the source files and the makefile; note the `-f` flag to `rm` - this has the effect of preventing `rm` from asking you if "you are really sure" about removing the specified files; it also prevents it from warning you if any of the specified files are not found.

Finally, there is often an `install` target in makefiles; we will revisit this at the end of this chapter.

4.2 A simple, example program

Recall that in section 3.1 we introduced a simple program that approximates π as a truncated infinite series. The code below is the C version of this program.

```
#include <stdio.h>      /* for access to printf() */
#include <math.h>       /* for access to sqrt(), pow(), fabs(), M_PI */
#include <stdlib.h>     /* for access to atoi() */

double approx_pi(int terms) {
    double sum = 0.0;
    int i;

    for (i = 0; i < terms; i++){
        double di = (double)i;
        sum += pow(-1.0, di) / ((2.0 * di + 1.0) * pow(3.0, di));
    }
    return sqrt(12.0) * sum;
}

int main(int argc, char *argv[]) {
    int terms;
    double pi, diff;

    if (argc == 2) {
        terms = atoi(argv[1]);
        pi = approx_pi(terms);
        diff = fabs(pi - M_PI);
        printf("pi[%d] = %.16f, error = %.5f%%\n", terms, pi, 100.0*diff/M_PI);
    } else {
        printf("usage: %s terms\n", argv[0]);
    }
    return 0;
}
```

As you can see, there are a number of similarities between the Python and C source files, as well as a number of differences. The following describes these similarities and differences.

- The beginning of the program is devoted to accessing functions and values from other files/modules. Python requires you to `import math` in order to access the square root function, the floating point absolute value function, and the value of π; it also requires you to `import sys` in order to access the command arguments.

  C requires you to `#include <math.h>` in order to access the square root function, the floating point absolute value function, and the value of π (denoted `M_PI`); C does not have a syntactic representation for exponentiation, so `math.h` also defines a function `double pow(double x, double y)` which raises `x` to the y^{th} power, returning it as a `double`. Note that in C, `double` is the default representation of a floating point number.

C does not require a special include to access arguments, since the signature for `main()` has the number of arguments and the arguments themselves passed in as values; the `char *argv[]` argument simply states that `argv` is a list of character strings, just as `sys.argv` is in Python.

C also requires you to `#include <stdlib.h>` in order to access the `atoi()` function; this converts a string representation of a number into an integer, just as the `int()` built-in function does in Python.

Finally, C requires you to `#include <stdio.h>` in order to access the `printf()` function; unlike Python, C does not have a built-in function to print data on standard output. Additionally, `printf()` acts like a combination of Python's `print()` function and the `format()` method on a string.

- C requires that the return type of a function be defined in its signature; additionally, sound use of C demands that you provide the type of each argument to a function.

  Unlike Python, where all lines of a block must be at the same indentation level, C uses curly braces (`{}`) to delimit a block. Thus, the entire body of `approx_pi()` is within a pair of curly braces; the body of the `for` loop in the function is also within a nested pair of curly braces.

  C requires that all variables be declared before use. Thus, immediately after the signature line for `approx_pi()`, we declare variables named `sum`, which is a `double` *and* initialized to `0.0`, and `i`, which is an `int`.

  Note that immediately after the `for()` statement, we also declare `di`, which is a `double` and is assigned the value of `i` converted to a `double`. Explicit casts from one type to another are done as shown. `di` is only defined in the body of the `for()` loop, not outside of it.

  The computation of `sum` is identical to our Python code, except that each occurrence of `x**y` had to be converted to an invocation of `pow(x, y)`. The `return` is also identical.

  Note that each statement in the function is terminated by a semicolon (`;`). This is because C allows statements to span multiple lines, as well as allowing multiple statements to be placed on a single line.

- `main()` is very similar between the two languages. Of course, in C we need to declare each variable before use.

  The call to `printf()` looks a bit strange; the first argument is like the string template upon which we invoked the `format()` method in our Python code, although `printf()` uses percent signs to specify where an argument value should be placed (unlike the curly braces used in Python). Another difference is that there is no format specifier equivalent to `{:%}`, thus we had to multiply the fractional difference by `100`; finally, since `%` is the special character in the format string for `printf()`, we had to double it in order for it to print out a percent sign.

4.2.1 Do they give the same results?

First, let's build a `Makefile` to construct our `approx_pi` executable.

```
CFLAGS=-W -Wall
OBJECTS=approx_pi.o
LIBRARIES=-lm

approx_pi: $(OBJECTS)
        gcc -o approx_pi $(OBJECTS) $(LIBRARIES)

approx_pi.o: approx_pi.c

clean:
        rm -f approx_pi $(OBJECTS)
```

Note that we have added another variable, `LIBRARIES`, to the makefile. To access the library of math routines, one must add `-lm` to the `gcc` command that builds `approx_pi`. By defining it as a variable at the top of the makefile, it is easy for you to remember which libraries you are using and for you to add additional libraries easily if your code grows to use other libraries. Routines accessed from `<stdio.h>` and `<stdlib.h>` are resolved from the default system library that `gcc` consults whenever it is linking together a program.

Now, let's build `approx_pi`, and then compare the output from it and `approx_pi.py`.

```
$ make
cc -W -Wall   -c -o approx_pi.o approx_pi.c
gcc -o approx_pi approx_pi.o -lm
$ ls -l approx_pi*
-rwxr-xr-x 1 me me 8360 Jul 30 11:49 approx_pi
$ python3 approx_pi.py 5
pi[5] = 3.1426047456630846, error = 0.03222%
$ ./approx_pi 5
pi[5] = 3.1426047456630846, error = 0.03222%
```

While these look the same, we can use a pipeline and `diff` to be completely sure, as in the following.

```
$ python3 approx_pi.py 5 >tmp.out
$ ./approx_pi 5 | diff - tmp.out
$ rm -f tmp.out
```

Here we have captured the output from the Python script into `tmp.out`. The subsequent pipeline executes `approx_pi` with the same number of terms of the summation, piping the output to `diff`. The bare hyphen (`-`) as an argument to `diff` tells it to use its standard input; thus, `diff` compares its standard input to `tmp.out`. The lack of output from `diff` indicates that the two files are exactly the same. We then remove `tmp.out` as good file system hygiene.

4.3 Variables, types, operators, and expressions

4.3.1 Variable names

The names of variables in C are made up of alphabetic letters, digits, and the underscore (`_`) character. The first character of a name must be a letter or an underscore; you are urged *not* to begin variable names with an underscore as system library routines often use the underscore as the first letter to avoid collisions with names you define. Upper and lower case letters are distinct. Conventional practice is to use all upper case for defined symbolic constants (e.g., `BUFFER_SIZE`). Conventions for variables vary:

- all lower case: in this situation, if you have a variable name with two or more words, use the underscore to separate the words (e.g., `modification_time`);
- mixed case/start lower: for single word variable names, all lower case; for multi-word variable names, use "camel case" - i.e., capitalize the first letter of the 2nd and subsequent words (e.g., `modificationTime`);
- mixed case/start upper: as we shall see in the next chapter, we will capitalize the first letter of each word in the name for an abstract data type (e.g., `PriorityQueue`).

At least the first 31 characters of a variable name are significant (more on Linux, but if you are going to port your programs to other C compilers on other operating systems, the language standard only guarantees 31.) Keywords in the language, such as `if`, `else`, `int`, `float`, ... are reserved words and must be in lower case.

In this book, we will use mixed case/start lower and mixed case/start upper in our examples.

Variables must be declared before they are used, unlike in Python. In order to declare variables, we first need to understand the data types that C supports.

4.3.2 Basic data types

C has only a few basic data types:

- `char` - this is a single byte, capable of holding one character in the local character set;
- `int` - this is an integer, typically reflecting the natural size of integers on the machine;
- `float` - single-precision floating point;

- `double` - double-precision floating point.

`short` and `long` qualifiers apply to integers; for example `short int i` or `long int counter`, or even `long long int packetCount`; note that these examples show how one declares the type of a variable in your program. The name `int` can be omitted when using the `short` or `long` qualifiers.

As described above, the actual precision for an `int` can vary from machine to machine, thus making it hard to write completely portable source code. Thus, the standard has specified that the following are true:

- `short int` is at least 16 bits of precision;
- `long int` is at least 32 bits of precision;
- `long long int` is at least 64 bits of precision.

`signed` and `unsigned` qualifiers can be applied to `char` or any integer type:

- unsigned numbers are always positive or zero, obeying the laws of arithmetic modulo 2^n, where n is the number of bits in the type;
- signed numbers vary from $2^{-(n-1)}$ to $2^{(n-1)} - 1$, obeying the laws of 2s-complement arithmetic.

`long double` specifies extended-precision floating point; it is not often used, and will not be used in this book. There are standard header files, `<limits.h>` and `<float.h>`, that contain symbolic constants for all of these sizes, along with other properties of the machine and compiler.

4.3.3 Structured types

C supports the creation of arrays of a given type, accessed via indexing. For example, one would declare an array of `25` integers as follows:

```
int myArray[25];
```

Arrays indices start at `0`; thus, the legal indices for `myArray` above are `0 .. 24`. One refers to the element at index `10` as `myArray[10]`.

C does *not* have a special built-in type for *strings*. Instead, a *string* is an array of characters, as in

```
char buf[1024];
```

We will discuss strings in more detail later in the chapter.

C also supports the definition of *structures*, similar to data-only classes in Python. We will discuss structures later in this chapter.

4.3.4 Constants/literals

You will need to be able to use literal values for different types in your program; these are usually termed *constants* in C. The following table indicates how to express constant integer values for different integer types.

signed integer	`int`	`1234`
signed long integer	`long`	`1234L`
signed long long integer	`long long`	`1234LL`
unsigned integer	`unsigned`	`1234U`
unsigned long integer	`unsigned long`	`1234UL`
unsigned long long integer	`unsigned long long`	`1234ULL`

Floating point constants contain a decimal point or an exponent or both; the type of the constant is `double` unless a suffix is provided.

single-precision floating point	`float`	`123.4f`	`1e-2f`	`1.2e7f`
double-precision floating point	`double`	`123.4`	`1e-2`	`1.2e7`
extended-precision floating point	`long double`	`123.4L`	`1e-2L`	`1.2e7L`

4.3.5 Character and string constants

A character constant is an integer, written as a single character within single quotes, such as `'x'`. Escape sequences, such as `'\n'`, are character constants. The following table shows the legal escape sequence character constants.

`\a`	alert (bell)	`\b`	backspace
`\f`	formfeed	`\n`	newline (end of line)
`\r`	carriage return	`\t`	horizontal tab
`\v`	vertical tab	`\\`	backslash
`\?`	question mark	`\'`	single quote
`\"`	double quote	`\0`	null (end of string)

A string constant/literal is a sequence of 0 or more characters surrounded by double quotes ("); the quotes are *not* part of the string, only serving to delimit the string contents. As described previously, a string is an array of characters; a string constant is just such an array of characters, with a null character (`'\0'`) at the end.

It is important to understand the difference between character and string literals. `'x'` is an integer, representing the numerical value of the letter `x` in the machine's character set. `"x"` is an array of characters, 2 characters long, consisting of `'x'` followed by `'\0'`.

Given what we have seen so far, we can write a simple function that calculates the length of a string.

```
int strlen(char s[]) {
    int i, len;

    len = 0;
    for (i = 0; s[i] != '\0'; i = i + 1) {
        len += 1;
    }
    return len;
}
```

You should convince yourself that this function works correctly if passed an empty string (i.e., `""`). The production version of `strlen()`, along with many other useful string-manipulation functions, are defined in `<string.h>` and can be used if you `#include <string.h>` in your source files.

4.3.6 Variable declarations

All variables must be declared before use. Each declaration specifies a type, and associates one or more variable names with that type; for example:

```
int first, last, step;
char c, buf[1024];
```

A variable may be initialized in its declaration, as in:

```
int formatChar = '%';
int bufferSize = MAXBUF + 1;
char keyword[] = "expedite";
```

The last example above shows that one does not need to specify the size of an array if you are initializing it with a string literal.

4.3.7 Variable scope

Variables can be declared outside of any function definition, in which case they are referred to as *external* variables. External variables can be accessed by *any* code in any function in any source file that is linked together with the file that declares that external variable. For external variables, initialization is done only once, before the program starts to execute; as a result, the initializer must be a constant expression. If an external variable is not explicitly initialized, it is initialized to zero by default.

Variables declared at the top of a block (after an opening left curly brace `{`) are referred to as *automatic* variables. An explicitly initialized automatic variable is initialized each time the defining block is entered; the initializer can be any expression. An automatic variable for which there is no explicit initializer has an undefined value.

The qualifier `const` can be applied to the declaration of any variable to indicated that its value will not change, as in:

```
const char errmsg[] = "processing error";
```

`const` can also be used with arguments to functions to indicate that the function does not change that array, as in:

```
int strlen(const char str[]);
```

4.3.8 Arithmetic operators

The binary operators `+`, `-`, `*`, and `/` are defined for both integer and floating point types; the modulus operator, `%`, is also defined for integer types. For integers x and y, `x / y` yields the integral number of times that `y` goes into `x`, while `x % y` yields the remainder from that division. Or more succinctly, `y * (x / y) + (x % y)` is identical to `x`.

4.3.9 Relational and logical operators

C does not possess a Boolean type. C interprets an integer value of 0 as false, and any integer value that is *not* 0 as true. The relational and logical operators described below return 1 when the result is true, 0 when false.

The following comparison operators, when used with numeric types, generate boolean values:

`x < y`	x is strictly less than y
`x <= y`	x is less than or equal to y
`x > y`	x is strictly greater than y
`x >= y`	x is greater than or equal to y
`x == y`	x is equal to y
`x != y`	x is not equal to y

Boolean values can be combined using the usual boolean operators:

`x || y`	if x is `False`, then y, else x	y is only evaluated if x is `False`
`x && y`	if x is `False`, then x, else y	y is only evaluated if x is `True`
`!x`	if x is `False`, then `True`, else `False`	has lower priority than non-Boolean operators, so `!a == b` is interpreted as `!(a == b)`

4.3.10 Type conversions

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a "narrower" operand into a "wider" one without losing information, such as converting an integer to floating point in an expression like `f + i`.

Expressions that do not make sense, like using a floating point value as an index into an array, are illegal, and will generate compiler errors. Expressions that might lose information, such as assigning a wider integer type to a narrower one, are *not* illegal, but will generate a warning from the compiler to alert you to the potential danger in doing so.

Of particular importance is to note that a `char` is just a small integer, so, as unusual as it may seem, `char` variables and constants may be freely used in arithmetic expressions. One must exercise caution, though, as many such uses make assumptions about contiguity of sequences of digits or letters, or about the relationship between lower and upper case letters. It is much safer to rely upon functions in `<ctype.h>` for performing such manipulations: `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `tolower()`, and `toupper()`.

If a binary operator has operands of different types, the "narrower" type is promoted to the "wider" type before the operation proceeds, and the result is of the "wider" type. If there are no unsigned operands, the following informal rules are followed:

```
if either operand is long double
    convert the other to long double
else if either operand is double
    convert the other to double
else if either operand is float
    convert the other to float
else
    convert char and short to int
    if either operand is long long
        convert the other to long long
    else if either operand is long
        convert the other to long
```

Conversions take place across assignments; the value of the right hand side is converted to the type of the left hand side, which is the type of the result; "wider" integers are converted to "narrower" ones by dropping the excess high order bits. Floating point to integer conversions cause truncation of any fractional part.

You can avoid depending upon implicit conversions by explicitly coercing the result of an expression using a *cast*; these are of the form `(`*type-name*`)` *expression*. The result of *expression* is converted to *type-name* using the type conversions rules. For example, given an integer `N`, we take its square root by invoking `sqrt((double)N)`. In this case, the compiler converts the integral value of `N` to a `double` before invoking `sqrt()`. The cast produces the value of `N` of the proper type; `N` itself is not altered.

4.3.11 Increment and decrement operators

Incrementing and decrementing variables happens so often in C programs that there is special syntax for it.

`++x`	adds `1` to `x` before returning its value
`x++`	adds `1` to `x` after returning its value
`--x`	subtracts `1` from `x` before returning its value
`x--`	subtracts `1` from `x` after returning its value

Consider the following code fragment:

```
int x, y, n = 5;

x = n++;
y = ++n
printf("%d %d %d\n", n, x, y);
```

What do you think will be printed out?

`n` starts out as `5`. `x = n++` says to assign the current value of `n` to `x`, then increment `n`; this leaves `x` with a value of `5`, and `n` with a value of `6`. `y = ++n` says to increment `n`, and then assign the value of `n` to `y`; this leaves `n` with a value of `7`, and `y` with a value of `7`. Thus, when `printf()` is invoked, we should see the following on standard output:

```
7 5 7
```

Postfix autoincrement (`x++`) is often used when assembling strings in a character array, as in:

```
if (c == '\n')
    buf[i++] = c;
```

It is also used with pointers, which will be discussed later in this chapter.

4.3.12 Assignments and assignment operators

Assignment of the value of an expression to a variable has the same syntax as for Python, except for the required semicolon at the end.

```
variable = expression;
```

The entire assignment statement is also an expression, and returns a value - i.e., the value that was assigned to the variable. As a result, the following types of statements are not only legal, are are examples of efficient and elegant C source:

```
var1 = var2 = expression;
if ((status = fetch(...)) != EOF) ...
```

The first example above simply shows that one can assign the same value to several variables in a single line; this statement is processed right to left - i.e., `expression` is evaluated and assigned to `var2`. The value of that assignment is the value of `expression`, and this value is assigned to `var1`. The second example shows invocation of the function `fetch()`, assigning its returned value to `status`; the value of this assignment is then compared to `EOF` to conditionally execute the body of the `if` statement.

Besides normal assignment, it is often the case that one wants to evaluate an expression, then perform a binary operation between a variable and that expression, and reassign the result of that binary operation to the variable. We have seen this in Python with the `+=` syntax. C has very rich support for these *assignment operators*, all of the form `variable op= expression`; this is equivalent to `variable = (variable) op (expression)` except that `variable` is only evaluated once. The operators `+`, `-`, `*`, `/`, and `%` all have assignment operator forms.[2]

As with assignments, assignment operators have a value (the final value of the variable), and can occur in expressions. A good example is

```
while ((n += 5) < LIMIT) ...
```

4.3.13 Conditional expressions

C has a ternary operator that is useful in many situations. It has the form:

$x = expr_1 \ ? \ expr_2 : expr_3;$

This is equivalent to

[2]There are other binary operators, associated with bit manipulation, which also have assignment operator forms. Since we are not covering bit manipulation in this book, they have not been mentioned in this section.

```
if (expr$_1$) {
    x = expr$_2$;
else
    x = expr$_3$;
}
```

except that it also returns a value which can be exploited in an expression, just like assignment and assignment operators. You will see examples of its usage later in the book.

4.4 Control flow

4.4.1 Statements and blocks

An expression becomes a *statement* when it is followed by a semicolon.

```
x = 0;
i++;
printf(...);
```

Unlike some other languages, where the semicolon is a statement separator, in C the semicolon is a statement terminator. Curly braces (`{}`) are used to group declarations and statements together into a *compound statement*, also known as a *block*. A block is syntactically equivalent to a single statement.

4.4.2 Conditional execution, `if-else`

As in Python, there are occasions where you will want different bits of code to be executed depending upon the state of your program. The syntax for `if-else` is

```
if (expression)
    statement$_1$
else
    statement$_2$
```

with the `else` part optional. `expression` is evaluated; if it is true (i.e., has a non-zero value), `statement`$_1$ is executed; if false, `statement`$_2$ is executed.

Since the `else` part is optional, there is an ambiguity when an `else` is omitted from a nested sequence of `if`'s. This ambiguity is resolved by associating the `else` with the closest previous `else`-less `if`.

In Python, you were also exposed to `elif` to perform multi-way decisions. In C, this is done as follows:

```
if (expression1)
    statement1
else if (expression2)
    statement2
else if (expression3)
    statement3
. . .
else
    statementN
```

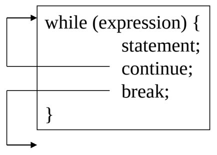
The expressions are evaluated in order; if any expression is true, the statement associated with it is executed, and the processing of the entire chain is terminated. Again, the trailing `else` can be omitted, although this is not a particularly good idea; if the preceding expressions capture all of the legal situations, then the `else` clause can catch illegal usage of your code.

4.4.3 Test at the top - `while` and `for`

The safest way to loop through code until a terminating conditions is reached is to test before entering the loop each time. Python enables this through its `while` and `for` statements. C provides a `while` statement that is very similar to that in Python; C's `for` statement, on the other hand, is much more general than Python's.

The syntax for the `while` statement is as follows:

```
while (expression)
    statement
```



`expression` is evaluated; if it is non-zero, `statement` is executed and `expression` is re-evaluated. This cycle continues until `expression` is zero, at which point execution resumes *after* `statement`. C provides a `break` statement that enables an early exit from a `while` loop.[3] The `break` statement in a `while` loop causes execution to resume after `statement`. C also provides a `continue` statement to cause the next iteration of the `while` loop; in particular, execution will resume at the test of `expression`.[4]

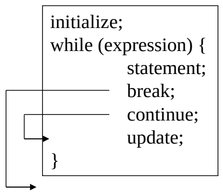
The syntax for the `for` statement is as follows:

```
for (expr1; expr2; expr3)
    statement
```

[3]`break` can also be used to achieve an early exit from `for` and `switch` statements, as well.
[4]`continue` can also be used to cause the next iteration in a `for` statement, as well.

`expr`$_1$ is known as the initialization field, and `expr`$_3$ is known as the update field.

This is equivalent to:



```
expr1;
while (expr2) {
    statement
    expr3;
}
```

except that the behavior of
the `continue` statement differs; in a `for` loop, a `continue` causes
`expr`$_3$ to be executed immediately, and then the test of `expr`$_2$.

While it is possible to put any legal C statements into `expr`$_1$ and `expr`$_3$, it is bad style to force unrelated computations into these fields in a `for` statement; these are best reserved for loop control operations appropriate to the situation.

What happens if you need to perform two or more statements in the initialization or update fields of a `for` loop? C provides a *comma* operator that enables one to specify multiple expressions in a statement; a pair of expressions separated by a comma (`,`) is evaluated left to right, and the type and value of the results are the type and value of the rightmost expression.

The comma operator can be used legally anywhere in your program,[5] but is most often used in the initialization and update fields of a `for` statement, as in

```
for (i=0, j=0; i <= M; i++, j++)
    statement
```

4.4.4 Multi-way decision based upon constants

While being very general, the multi-way decision control described above using nested `if-else` statements is not particularly efficient if one is attempting to compare the result of an expression against a set of constant values and take different actions based upon the result. C provides the `switch` statement to enable an efficient mechanism for such tests.

The syntax for the `switch` statement is as follows:

```
switch (expression) {
    case const-expr1: statements1
    case const-expr2: statements2
    case const-expr3: statements3
```

[5] While its use is legal anywhere, such use outside of the initialization and update fields of a `for` statement should be avoided, as it leads to potentially obscure code.

```
        . . .
        case default: statements$_N$
    }
```

In the switch statement, the cases simply serve as labels. If the `expression` matches one of the constants in a particular label, execution starts at the `statements` associated with that `case` label and continues *until the end of the `switch` statement* or until it encounters a `break` statement, at which point it will execute the first statement after the `switch` statement.

N.B. Since the default semantics (continue to the end) is almost *never* what one wants to happen, it is *critical* that you always include a `break` statement at the end of the set of statements associated with each `case` label. In other words, this is how you should use it!

```
    switch (expression) {
        case const-expr$_1$: statements$_1$; break;
        case const-expr$_2$: statements$_2$; break;
        case const-expr$_3$: statements$_3$; break;
        . . .
        case default: statements$_N$
    }
```

4.4.5 Labels and goto as a last resort

Most assembly languages, and some earlier “high-level” languages support inserting labels in your source code, and being able to force execution to continue at one of those labels. Sound software engineering principles proscribe use of the goto.[6]

Unfortunately, sometimes one needs to abandon processing in some deeply-nested control structure. The `break` statement enables us to escape from the innermost loop or `switch`, but cannot help us if we are several levels deep. Thus, labels and goto are most commonly used in such situations, as depicted below.

```
    for (. . .)
        for (. . .) {
            . . .
            if (disaster)
                goto error;
        }
    error:   /* code to clean up the mess */
```

[6]See Edgar Dijkstra, “Go To Statement Considered Harmful”, Communications of the ACM, Vol. 11, No. 3, pp. 147-148, March 1968.

A label has the same form as a variable name, and is followed by a colon (`:`). A label can be attached to any statement *in the same function* as the `goto` statement. The scope of a label is the entire function.

4.5 Functions and program structure

A C program is composed of functions and global data. One of the functions must have the name `main()`, as the runtime will call that function after it has initialized the process in which your program will execute.

4.5.1 Functions

Each function definition has the following form:

```
return-type function-name(argument declarations) {
    declarations and statements
}
```

Various parts of the function declaration may be absent; a minimal function declaration is:

```
dummy() {}
```

This function does nothing and returns nothing. When the return type is omitted, it defaults to `int`. Code that is dependent upon this default behavior is very dangerous; good software engineering practice dictates that one must *always* use function prototypes to declare the types of the function arguments and its return type, as this enables the compiler to make sure that you are using the function correctly.

Communication between functions is via arguments to and return values from a function; functions can also communicate through external global data variables. Functions can occur in any order in the source file, and the source program can be split into multiple files. A function *cannot* be split over files. One can pre-declare function signatures in a source file to guarantee that their use in the source file are type-correct; alternatively, the functions can be defined in an order that guarantees that a function is defined prior to first use in the source file.

Returning a value from a function to its caller is achieved via the `return` statement:

```
return expression;
```

The calling function is free to ignore the returned value, although this is not good software engineering practice.

An `expression` is not required after the `return` keyword; in such a situation, no value is returned to the caller. Control also returns to the caller (with no return value) if execution encounters the closing `}` in the function definition.

It is not illegal, but probably a sign of trouble, if a function returns a value in one place but not in another. If a function fails to return a value, its "value", if checked by the caller, is guaranteed to be garbage.

4.5.2 An example program

Let's construct a program to print each line of standard input that contains a particular "pattern" or string of characters - a limited version of the Linux `grep` program. The main program falls neatly into three pieces:

```
while (there's another line)
    if (the line contains the pattern)
        print the line
```

The *print the line* is simply a call to `printf()`, defined in `<stdio.h>`. The *while (there's another line)* is a call to a function `fgets()` defined in `<stdio.h>`. Its signature is the following:

```
char *fgets(char buf[], int size, FILE *stream);
```

You pass `fgets()` a character array (`buf[]`), the size of that array (`size`), and a stream of characters from an open file (`stream`). Each time `fgets()` is called, it copies the next line of input from `stream` into `buf`, and places `'\0'` after the line so that `buf` is a legal string in C; if it copied a line into `buf`, it returns the address of `buf` as its function value. If there are no more lines on `stream` when `fgets()` is called, it returns `NULL` as its function value and does nothing to `buf`. `fgets()` is similar to the `readline()` function in Python except that you have to provide `fgets()` with a buffer into which the next line of input is copied.

Thus, our `main()` is starting to look as follows:

```
#include <stdio.h>

#define BUFSIZE 1024

#define UNUSED __attribute__((unused))

int main(UNUSED int argc, UNUSED char *argv[]) {
    char buf[BUFSIZE];

    while (fgets(buf, BUFSIZE, stdin) != NULL) {
        if (the line contains the pattern)
```

```
                printf("%s", buf);
        }
    }
```

In fact, if you simply delete the *if (the line contains the pattern)* from this program, you will have a filter that simply copies its standard input to its standard output.

What's the odd definition for `UNUSED`, and why do we use it in the signature for `main()`? We previously advised that the warning flags to specify when compiling with `gcc` are `-W -Wall`. One of the warnings that will be generated with these flags is if there is a function parameter or an automatic variable that is *not* referenced in the function; in general, such a situation may be a strong indication of a problem in your code. `gcc` has provided a set of extensions for specifying attributes in your source code; one of those attributes enables you to indicate that a particular parameter or variable is unused *on purpose.* If you flag a parameter or variable with this attribute, `gcc` will not issue the corresponding warning, since you have proactively indicated that it will not be used. In our case, we are not using the command arguments, so flagging `argc` and `argv` as unused is appropriate.

So, how do we determine if one string is included in another string? We could write a function to do this, and we will show this later. But, in the meantime, we should look in `<string.h>` to see if such a function has been supplied by the runtime system. This can be easily done by asking your search engine to look for "linux man string"; somewhere in the first few results of the search you will see an entry entitled "string(3) - Linux manual page - man7.org".

A perusal of that page will yield the signature

```
    char *strstr(const char *haystack, const char *needle);
```

seemingly just the function we need. Clicking on the link for `strstr(3)` in the "SEE ALSO" section gets us to the man page for `strstr()`, which indicates that it returns `NULL` if `needle` is *not* contained in `haystack`, and the address of the location in `haystack` where `needle` is first found if it is.

Thus, our completed program (named `find.c`) looks as follows:

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

    #define BUFSIZE 1024

    #define UNUSED __attribute__((unused))

    int main(UNUSED int argc, UNUSED char *argv[]) {
        char buf[BUFSIZE];
        char pattern[] = "ould";   /* pattern to search for */

        while (fgets(buf, BUFSIZE, stdin) != NULL) {
            if (strstr(buf, pattern) != NULL)
                printf("%s", buf);
        }
        return 0;
    }
```

A program that has a static pattern to search for is not particularly useful, as we would have to edit and recompile the program to search for other patterns. We will look at how to make the program more dynamic later in the chapter.

4.5.3 External variables

A C program consists of a set of external objects which are either variables or functions. External variables are defined outside of any function, and are thus potentially available to many functions. Functions are *always* external since C does not permit functions to be defined inside of other functions. External variables and functions have the property that all references to them using the same name, even from functions compiled separately, are references to the same thing.

The *scope* of a name is the part of the program within which the name can be used. For an automatic variable declared at the beginning of a block, the scope is the block in which the name is declared. The scope of an external variable or a function lasts from the point at which it is declared to the end of the file being compiled.

If an external variable is to be references before it is defined, or it is defined in a different source file from the one in which it is being used, then an `extern` declaration is required.

It is important to distinguish between the *declaration* of an external variable and its *definition.* A declaration announces the properties of a variable, primarily its type; a definition also causes storage to be set aside for the variable.

For example, if the lines

```
int sp;
double val[MAXVAL];
```

appear outside of any function, they define the external variables `sp` and `val`, cause storage to be set aside for each of them, and serve as a declaration for the rest of that source file.

On the other hand, the lines

```
extern int sp;
extern double val[];
```

declare for the rest of the source file that `sp` is an `int` and that `val` is a `double` array; they do not create the variables or reserve storage for them.

Only one definition of an external variable is allowed among all of the files that make up a program; initialization is restricted to that single definition.

4.5.4 Static variables

External variables enable two types of access to named storage: 1) by functions within a source file, and 2) by functions in other source files. Often we require external storage so that functions within a file can share, but wish to hide that information from functions in other source files. The keyword `static`, if prefixed to external variable definitions, achieves this level of hiding - i.e., all functions in the source file can access the static variables, but they are hidden from functions in other source files.

The following example declarations

```
static int sp;
static double val[MAXVAL];
```

define the types and cause storage to be created for `sp` and `val`. These variables can only be accessed by functions in the source file where these declarations occur.

External static declarations can be used for functions, as well. If a function is declared `static`, its name is *invisible* outside of the file in which it is declared. We will exploit `static` functions in later chapters when defining abstract data type methods.

Finally, `static` can also be applied to variables declared within functions. Internal `static` variables are only visible within the function in which they are declared, just as automatic variables; unlike automatics, the `static` variables retain their values across calls to the defining function. The following example shows a typical use of an internal `static` variable:

```
int someFunction(void) {
    static int initialized = 0;

    if (! initialized) {
        initialized++;
        /* perform required initialization */
    }
    /* logic of someFunction() */
    return /* appropriate value */
}
```

4.5.5 Header files

We use header files (ending in `.h`) to specify types, symbolic constants and function prototypes. We use source files (ending in `.c`) to define (and initialize) external variables and define functions. In our pattern matching program, we included `<stdio.h>`, which defined the variable `stdin` and the functions `fgets()` and `printf()`; we also included `<string.h>`, which defined the function `strstr()`.

Let's suppose that we wish to replace our call to `strstr()` by our own function, with the signature

```
int isSubString(const char *needle, const char *haystack);
```

where `isSubString()` returns 1(true) if `needle` is contained in `haystack`, and 0 (false) if not. We will create two files: `issubstring.h`, which defines the function signature, and `issubstring.c`, which implements the function.

issubstring.h
```
int isSubString(const char *needle, const char *haystack);
```

issubstring.c
```
#include "issubstring.h"
#include <string.h>

int isSubString(const char *needle, const char *haystack) {
    return strstr(haystack, needle) != NULL;
}
```

Admittedly, `isSubString()` is a very simple function, simply using `strstr()` to do the hard work, and returning the correct return type. Note that our source file includes the header file using quotes (`""`) around the name instead of angle brackets (`<>`) - files included using angle brackets are searched for in standard directories in the file system; files included using quotes are first searched for in the current directory, then in the standard directories.

You may ask, why include the header file in the source? This guarantees that the file signature defined in the header, which will be used by other files, is identical to that defined in the source.

Our main program, `find.c`, must now look as follows:

```
#include "issubstring.h"
```

```
#include <stdio.h>

#define BUFSIZE 1024

#define UNUSED __attribute__((unused))

int main(UNUSED int argc, UNUSED char *argv[]) {
    char buf[BUFSIZE];
    char pattern[] = "ould";   /* pattern to search for */

    while (fgets(buf, BUFSIZE, stdin) != NULL) {
        if (isSubString(pattern, buf))
            printf("%s", buf);
    }
    return 0;
}
```

We no longer need to include `<string.h>` here, since this code does not use `strstr()`. It does need to include `"issubstring.h"`, since we are now calling that function. And, obviously, we replace the call to `strstr()` with a call to `isSubString()`.

All that is left to support ease of building and debugging is to create a Makefile for our pattern matching program.

```
CFLAGS=-W -Wall
OBJECTS=find.o issubstring.o

find: $(OBJECTS)
        gcc -o find $(OBJECTS)

find.o: find.c issubstring.h
issubstring.o: issubstring.c issubstring.h

clean:
        rm -f $(OBJECTS) find
```

You might ask “Why indicate a dependency for `find.o` upon `issubstring.h` but not for `stdio.h`?”. The standard include files are very stable, and do not change. Our local include files, on the other hand, are likely to change as we debug the program. Thus, we want `make` to include our local include files in the dependency graph.

Now, let’s eliminate the dependence of `find` on a compiled pattern. We want our program to take a single argument, which is the pattern we wish to look for. Here is the absolute final version of `find.c`.

```
#include "issubstring.h"
```

```
#include <stdio.h>

#define BUFSIZE 1024

int main(int argc, char *argv[]) {
    char buf[BUFSIZE];
    char *pattern;   /* pattern to search for, taken from argv[1] */

    if (argc != 2) {
        printf("usage: %s pattern\n", argv[0]);
        return 1;
    }
    pattern = argv[1];
    while (fgets(buf, BUFSIZE, stdin) != NULL) {
        if (isSubString(pattern, buf))
            printf("%s", buf);
    }
    return 0;
! }
```

Note that we removed the `#define` of `UNUSED`, as well as application of that attribute to `argc` and `argv`, since we *are* using the argument parameters. We first check that the user has provided a pattern argument by comparing `argc` with the value `2` - remember that `argv[0]` is the command name specified in the `bash` command line, and `argv[1]` is the first real argument to the program. If the user has either not specified the pattern, or provided too many arguments, we remind the user of the correct command line and return a non-zero value, indicating an error. Otherwise, we use the pattern in argv[1].

4.5.6 Block structure

While functions *cannot* be defined inside of other functions, variables can be defined in a block-structured fashion within a function. Declarations of variables, including initializations, can follow the left brace (`{`) that introduces any compound statement, not just the one that begins a function. Variables declared in this way hide any identically named variables in outer blocks, and remain in existence until the matching right brace (`}`).

Good software engineering practice recommends that you avoid variable names that conceal names in an outer scope, as the potential for confusion and error is too great. Consider the following example - what number will be printed out?

```
#include <stdio.h>

int number = 42;
int main(int argc, char *argv[]) {
    int number = 10;
    int i, j;

    i = 5;
    j = 23;
    {
        int number;

        for (number = i; number < j; number++)
            ;
    }
    printf("%d\n", number);
    return 0;
}
```

4.5.7 Initialization

In the absence of explicit initialization, external and static variables are guaranteed to be initialized to zero. Automatic variables have undefined initial value - i.e., in the absence of explicit initialization, they should be assumed to contain garbage. Scalar variables can be initialized when they are defined by following the name with = and an expression. For external and static variables, the initializer *must* be a constant expression; the initialization is done once, before the program begins execution. For automatic variables, the initializer is not restricted to being a constant - it may be any expression involving previously defined values, even function calls; the explicit initialization of automatic variables is performed each time the function or block is entered.

An array may be initialized by following its declaration with a list of initializers enclosed in braces and separated by commas, as in

```
int days[] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
```

If the size of the array is omitted, as in this example, the compiler computes the length of the array from the number of initializers. If the size of array is specified, and there are fewer initializers than its declared size, the missing elements will be zero for external, static, and automatic variables. If there are more initializers than elements in the array, a compiler error is generated.

C does not maintain the length of an array at runtime, unlike Python. When using an initialization as for `days[]` above, your code may need to know the number of elements that the compiler actually created. There are two common ways to do this:

- you can append a value to the list of initializers that is obviously different from the

others (e.g., a `-1` in `days[]`), such that at runtime you can count the number of items in the array until hitting the terminating value; or
- a more C-savvy way to do this is to use the `sizeof` compile-time operator to define a constant that is the length of the array; `sizeof(type-name)` is replaced at compile time with the number of bytes that an instance of `type-name` will occupy in memory; `sizeof variable-name` is replaced at compile time with the number of bytes that the variable will occupy in memory; the following code shows how to exploit this to yield a defined constant that is the size of `days[]`:

  ```
  int days[] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
  #define DAYS_LENGTH (sizeof days / sizeof(int))
  ```

  Your code may refer to `DAYS_LENGTH` whenever it needs to limit its accesses to the legal index values.

4.5.8 The C preprocessor

Every C compiler consists of multiple passes. The preprocessor is the first pass of the compiler; during this pass, the preprocessor replaces commands that it understands with other text, obtained from other files or from symbolic constants. The commands that are understood are: `#include`, `#define`, `#if`, and `#ifdef`.

4.5.8.1 File inclusion

We have already encountered this earlier. Any source line of the form

```
#include "filename"
#include <filename>
```

is *replaced* by the contents of `filename`.

If `filename` is delimited by quotes, the file is first search for in the directory where the source file is found. If it is not found there, or if `filename` is delimited by angle brackets, known directories are searched for `filename`. An included file may itself contain `#include` lines.

File inclusion has very different semantics from `import` statements in Python. Python `import` explicitly brings in names from the referenced modules. File inclusion simply replaces each `#include` statement with the contents of that file.

There are often several `#include` lines at the beginning of a source file. These include common `#define` statements and extern declarations, or access to function prototype declarations, such as `<stdio.h>`. `#include` is the preferred way to tie the declarations together for a large program. Note that when an included file is changed, all files that depend upon the included file must be recompiled.

4.5.8.2 Macro substitution

A macro definition has the form

```
#define name replacement-text
```

Subsequent occurrences of `name` in the source file will be replaced by `replacement-text`. `name` has the same form as a C variable name, while `replacement-text` is arbitrary. Normally, `replacement-text` is the rest of the macro definition line; a long definition may be continued onto several lines by placing a `\` at the end of each line to be continued. The scope of `name` is from the point of definition to the end of the source file being compiled.

A macro definition may use previous definitions. Substitutions do *not* take place within quoted strings - i.e., if `YES` is a defined macro name, there would be no substitution in `printf("YES")` or `YESMAN`.

A name may be defined with absolutely any replacement text; for example, the following is legal and often used:

```
#define forever for(;;)  /* infinite loop */
```

Macros can be defined with arguments, as in

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

When invoked in your code, although it looks like a function call, `max()` expands into in-line code. Each occurrence of a formal parameter to the macro is replaced by the corresponding actual argument. Therefore,

```
x = max(p+q, r+s);
```

becomes

```
x = (((p+q) > (r+s)) ? (p+q) : (r+s));
```

If an expression causes side effects (e.g., `n++`), macros can give the wrong results; for example, `x = max(i++, j++);` will yield the wrong results, as the larger of `i` or `j` will be incremented twice.

As you may have noticed in our `max()` example, you must also be extremely generous with parentheses to make sure that the appropriate order of evaluation is preserved. For example, if we define

```
#define square(x) x * x
```

what happens if we invoke

```
square(z+1)
```

Fix the definition in order to obtain the correct results.

4.5.8.3 Conditional evaluation

There are a number of statements that are used to control preprocessing, thus providing a way to include code selectively, depending upon the value of conditions evaluated during compilation. `#if` evaluates a constant integer expression; the expression may *not* include `sizeof` or casts. If the expression is non-zero, subsequent lines until an `#endif` or `#elif` or `#else` are processed; if the expression is zero:

- if an `#elif expr1` is found, then `expr1` is evaluated; if non-zero, subsequent lines until an `#endif/#elif/#else` are processed; if zero, repeat this step;
- if an `#else` is found, then subsequent lines until an `#endif` are processed.

The expression `defined(name)` is `1/0` if `name` is defined/not.

If the contents of a header file are included more than once while compiling a source file, it can lead to all kinds of difficulties. To make sure that the contents of a header file (e.g., `hdr.h`) are included only once, `hdr.h` surrounds the actual contents with a conditional like this:

```
#if !defined(_HDR_H_)
#define _HDR_H_

/* actual contents of hdr.h go here */

#endif /* _HDR_H_ */
```

This allows header files to include all other header files upon which it depends without having to worry about multiple inclusions of some common header files. All of the standard header files (e.g., `<stdio.h>`) do this; you should do this with your header files, as well.

There is nothing special about using `_HDR_H_` as the defined symbol to indicate that the file has been included; you just need to pick a name that will not collide with other defined constants. Leading and trailing underscores, replacing the `.` by an underscore, and converting all letters to upper case is a common approach used by C and C++ programmers.

The `#elif` construct is to enable a switch-like choice of lines to process, as in

```
#if SYSTEM == OSX
    #define HDR "osx.h"
#elif SYSTEM == LINUX
    #define HDR "linux.h"
#elif SYSTEM == WINDOWS
    #define HDR "windows.h"
#else
    #define HDR "default.h"
#endif /* SYSTEM */
#include HDR
```

Finally, `#ifdef name` is a synonym for `#if defined(name)`, and `#ifndef name` is a synonym for `#if !defined(name)`.

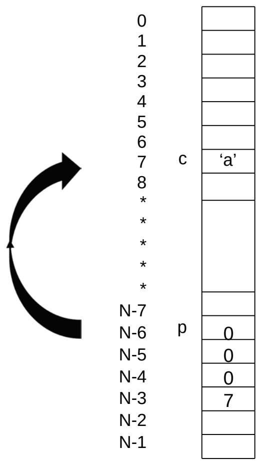
4.6 Pointers and arrays

Up to this point, it is not clear why C would be preferred over any other language. The set of basic data types is sparse, and arrays are the only structured built-in type. What's so special about C?

C supports a pointer data type; a pointer is a data variable that contains the address of (i.e., *points to*) another variable. Pointers to data are integrally related to arrays. Additionally, C supports pointers to functions, a feature that we will exploit when we introduce abstract data types.

4.6.1 Pointers to data

A typical computer has an array of consecutively numbered (or addressed) memory cells that can be manipulated individually or in contiguous groups; in the figure to the right, we are assuming `N` cells, numbered `0 .. N-1`.



Now suppose that we have a `char` variable named `c`, and that it is assigned to address `7`.[7] Furthermore, we have a pointer to a character, `p`, that is assigned to address `N-6`.

We can make `p` point to `c` with a statement of the form `p = &c;`. The unary operator `&` gives the address of a variable, it is verbalized as "address of". After executing such a statement, `p` is said to "point to" `c`.

`&` can only be applied to variables and

[7]The linker decides where to place variables when linking the program together.

array elements; it cannot be applied to expressions or constants.

Once one has a pointer, how do you get at the contents
of the variable to which it points? The unary operator `*` is the
indirection or dereferencing operator; when applied to a pointer,
it accesses the object to which the pointer points. In our previous example, `*p` would yield `'a'`, which is the character stored in `c`. The following artificial sequence of statements show the use of `&` and `*`.

```
int x = 1, y = 2, z[10];
int *p, *q;         /* p and q are pointers to an int */

p = &x;             /* point now points to x */
y = *p;             /* y is now 1 */
*p = 0;             /* x is now 0 */
q = &z[0];          /* q now points to z[0] */
p = q;              /* p now points to z[0] */
```

Note that the declaration for a pointer to an `int` is `int *p;` - i.e., it indicates that the expression `*p` can be used anywhere that an `int` is legal; it also indicates that `p` must be dereferenced to yield an `int` - i.e., `p` is a pointer to an `int`.

Pointers are constrained to point to a particular type of object - in this case, `p` is a pointer to an `int`.

4.6.2 Call by value and pointers

When you call a C function, the value of the argument is passed to the function. The function can not only read the values passed, but can modify them; since they are copies, the caller's copies of those values *are not* changed. Thus, given call by value, there is no direct way for a function to alter a variable in the calling function.

Suppose we need a function to swap two values as part of an algorithm. A naive approach would be as follows:

```
void swap(int x, int y) {   /* Will not work */
    int temp;

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

If a caller invokes `swap(a, b)`, upon return from `swap()`, `a` and `b` will still have the values they had before the call to `swap()`. This is because the function is swapping copies of the actual arguments, not the arguments themselves.

What happens if we modify `swap()` as follows?

```
void swap(int *px, int *py) {   /* swap *px and *py */
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

If a caller now invokes `swap(&a, &b)`, upon return the values in `a` and `b` will truly be swapped.

You saw similar situations in Python when you passed mutable data structures (like lists or dictionaries), since a called function could return a value for the function, modify a mutable data structure, or both. By passing pointer parameters to a function, the function can modify the variables to which the pointers point, as well as return a function value.

4.6.3 Pointers and arrays

Pointers and arrays are strongly related in C, in that any operation that can be achieved by array subscripting can also be done with pointers.

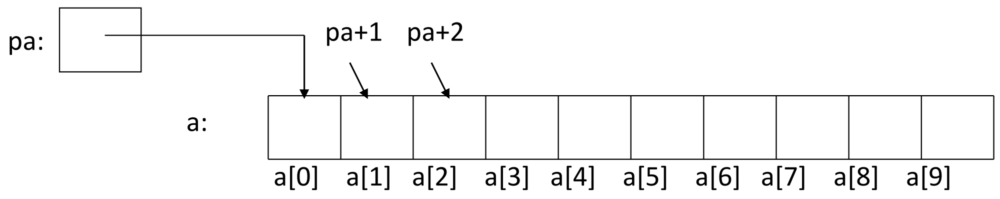
Consider the following declaration:

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

This defines an integer array named `a` of size `10` - i.e., a block of `10` consecutive `int` objects in memory named `a[0]`, `a[1]`, ..., `a[9]`. `a[i]` refers to the `i`th element of the array.

Now assume that `pa` is a pointer to an integer, declared as

```
int *pa;
```



The assignment `pa = &a[0];` causes `pa` to point to element zero of `a` - i.e., `pa` contains the address of `a[0]`. If `x` is an integer, the assignment `x = *pa;` copies the contents of `a[0]` into `x`. *By definition*, `pa + 1` points to the next element past `pa`; `pa + i` points `i` elements past `pa`; and, `pa - i` points `i` elements before `pa`.

The preceding statements are true regardless of the type or size of the variables in the array `a`. The meaning of "add 1 to a pointer", and by extension, all pointer arithmetic, is that `pa + 1` points to the next object of that type beyond `pa`, and that `pa + i` points to the `i`th object of that type beyond `pa`.

The value of a variable of type array is the address of the `0`th element of the array - i.e., `a == &a[0]`. Thus, the following are equivalent:

```
pa = &a[0];
pa = a;
```

There are several ramifications of this strong relationship between pointers and arrays:

- a reference to `a[i]` can be written as `*(a + i)`;
- a reference to `&a[i]` is identical to `a + i`;
- `pa[i]` is identical to `*(pa + i)`;
- since a pointer is a variable, expressions like `pa = a` and `pa++` are legal;
- since an array name is not a variable, expressions like `a = pa` and `a++` are *illegal*;
- when an array name is passed as an argument to a function, what is passed is the address of the initial element; within the called function, the argument is a local variable; thus, an array name parameter is a pointer;
- as formal parameters in a function definition, `s[]` and `*s` are equivalent; thus, if an array name has been passed as the actual argument in a call, the function can believe that it has been handed either an array or a pointer;
- a part of an array can be passed to a function by passing a pointer to the beginning of the sub-array - e.g., `f(&a[2])` or `f(a+2)`.

4.6.4 Pointer arithmetic

If `p` is a pointer to some element of an array, then `p++` increments `p` to point to the next element, and `p += i` increments it to point `i` elements beyond the current element.

There is a distinguished pointer value, `NULL`, which means that the pointer does not point at anything valid; `NULL` is defined in `<stdio.h>`, `<stdlib.h>` and `<string.h>`.

Pointer values can be compared using `==`, `!=`, `>`, `>=`, `<`, and `<=`.

Adding or subtracting an integer from a pointer causes the behavior defined previously. Subtracting two pointers is also valid; if `p` and `q` point to elements of the same array, and if `p < q`, then `q - p + 1` is the number of elements from `p` to `q`, inclusive.

Thus, valid pointer arithmetic operations are:

- assignment of pointers of the same type;
- adding or subtracting a pointer and an integer;
- subtracting or comparing two pointers to members of the same array;
- assigning or comparing to `NULL`.

The following operations on pointers are *invalid*:

- add, multiply, or divide two pointers;
- add a float or double to a pointer;
- assign a pointer of one type to a pointer of another type.[8]

4.6.5 `void *` pointers and heap memory

4.6.5.1 `void *` pointers

As we shall see later in this chapter, pointers to structures act somewhat like object references in object-oriented languages. Nearly all object-oriented languages, Python included, have a base class `Object` from which all other classes inherit. C provides a generic pointer, `void *`; any pointer can be cast to `void *` and back again without loss of information. `void *` is used to construct modules that provide generic capabilities at runtime; we will be using these in our abstract data types.

One important aspect of `void *` pointers is that you *cannot* dereference them; attempts to do so will generate a compiler or a runtime error.

The most common initial exposure to `void *` pointers is through the dynamic memory allocation routines defined in `<stdlib.h>`.

4.6.5.2 Heap memory

Many of the data structures used to solve problems grow dynamically - i.e., one cannot know when the program starts how much memory a particular data structure will occupy. Languages like C (and Python) provide *heap memory* that can be allocated as a data structure needs to grow. In Python, every time you invoke a constructor (e.g., `list()`, `dict()`, `set()`, or the constructor for any class), memory is allocated from the heap. Python keeps track of the references to heap-allocated memory; when there are no more references to a chunk of heap-allocated memory, it can be returned to the heap. This is known as *garbage collection*.

[8] It is possible to use an explicit cast to assign pointers of different types. This will be discussed in the following section.

C provides a set of routines for allocating and freeing heap memory (in `<stdlib.h>`), but does not track references to heap-allocated memory, so does not provide garbage collection. Thus, your program must *itself* keep track of references to heap blocks, and free blocks when there are no more references. Failure to do so causes *memory leaks* in your program, which should be avoided. We will discuss the `valgrind` program later in this chapter which helps you find memory leaks.

The function prototypes for the routines in `<stdlib.h>` are as follows:

```
/*
 * malloc: return a pointer to space for an object of size `size' bytes, or
 *         NULL if the request cannot be satisfied. The space is uninitialized.
 */
void *malloc(size_t size);

/*
 * free: deallocates space pointed to by `ptr'; it does nothing if `ptr' is  NULL.
 *       `ptr' must be a pointer to space previously allocated by malloc(), calloc()
 *       or realloc().
 */
void free(void *ptr);

/*
 * calloc: returns a pointer to space for an array of `nmemb' elements, each
 *         of size `size' bytes, or NULL if the request cannot be satisfied.
 *         The space is initialized to zero bytes.
 */
void *calloc(size_t nmemb, size_t size);

/*
 * realloc: adjusts the size of the memory block pointed to by `ptr' to `size' bytes,
 *          returning a pointer to the resized block; the contents will be unchanged
 *          in the range from the start of the region up to the minimum of the old and
 *          new sizes; if the new size is larger, the added memory will not be
 *          initialized; if a new block had to be allocated, a free(ptr) was done
 */
void *realloc(void *ptr, size_t size);
```

These prototypes use a type `size_t`, which is also defined in `<stdlib.h>`. Think of it as an integer.

How do you know the number of bytes that you need to ask for? We discussed `sizeof` earlier, as this compile-time expression is replaced by the number of bytes needed for an instance of a type or for a particular variable. When you invoke `malloc()` to allocate some heap memory, a pointer to the first byte in the block is returned to you as a `void *`. Let's look at an example.

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

int *p;

p = (int *)malloc(sizeof(int));
if (p != NULL) {
    *p = 42;
    /* other uses of the allocated memory */
    free(p);    /* deallocate the memory when done */
} else {
    printf("Error allocating memory for integer\n");
}
```

What does this code do? We include `<stdlib.h>` so that we can call `malloc()` and refer to `NULL`; we also include `<stdio.h>` so that we can call `printf()`. We declare a pointer of the appropriate type. We then call `malloc()`, using `sizeof()` to specify the number of bytes needed for an integer. We use the cast `(int *)` to explicitly convert from a `void *` pointer (returned by `malloc()` to an `int *`. We check to see if the `malloc()` was successful; if so, we use the allocated memory, then return it to the heap; if not, we print an error message.

There are two important aspects to this example:

- you should *always* check the return result from `malloc()` (or `calloc()` or `realloc()`); if the allocation failed, and your code attempts to dereference a `NULL`, your program will abort with a *segmentation violation*; if the returned value is `NULL`, your program needs to take appropriate action;
- the line

  ```
      int *p = (int *)malloc(sizeof(int));
  ```

  is an integer-specific version of a standard pattern that you will see in code that uses `malloc()`; for any given type `type`, the allocation of an instance of that type will look as follows:

  ```
      type *p = (type *)malloc(sizeof(type));
  ```

  - i.e., `sizeof(type)` is the argument to `malloc()`, the `void *` return from `malloc()` is cast to `type *`, and the result is assigned to a variable of `type *`.

4.6.5.3 An example program

The following program reads up to the first 100 lines from standard input, stores those lines in dynamic memory, prints each of the stored lines, and then frees the dynamic memory. We use two functions from `<string.h>`: `strlen()`, which counts the number of characters in a character array up to, but not including, the `'\0'` that terminates the string, and `strcpy()`, which copies the second argument string into the first. This example program also uses a number of other aspects of the language that we have discussed so far.

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

#define NLINES 100
#define MAXLINESIZE 1024

int main(int argc, char *argv[]) {
    char *lines[NLINES];
    char buf[MAXLINESIZE];
    char *p;
    int i;
    int nl = 0;

    while (nl < NLINES && fgets(buf, MAXLINESIZE, stdin) != NULL) {
        p = (char *)malloc(strlen(buf) + 1);   /* why the +1? */
        strcpy(p, buf);
        lines[nl++] = p;
    }
    for (i = 0; i < nl; i++)
        printf("%s", lines[i]);
    for (i = 0; i < nl; i++)
        free((void *)lines[i]);
    return 0;
}
```

Why did we have to specify the size for the heap buffer as `strlen(buf) + 1`? The value returned by `strlen()` does not include the `'\0'` that indicates the end of string. If we are going to make a legal copy of a string, we have to allocate enough bytes to hold all of the characters *and* the `'\0'`; thus, we add one to the number of characters. Forgetting to account for the terminating 0 byte is an *exceedingly* common error that students make. In fact, one has to make copies of string on the heap so often that `<string.h>` defines a function, `char *strdup(char *str);`, that allocates the correct number of bytes on the heap, copies `str` into that block, and returns a `char *` pointer to the heap memory. You are encouraged to use `strdup()` for this very common operation and to avoid not allocating space for the 0 byte.

4.6.6 Character pointers and functions

The most common pointers that you will encounter are pointers to characters. Strings are arrays of characters, with each character in the string occupying one position in the array; one additional position at the end of the string holds the 0-byte, `'\0'`, to terminate the string.

A string literal is written as: `"This is a string"`. The literal is stored as an array of characters in *read-only* memory, with the terminating 0-byte. When a string literal/constant is specified as an argument to a function, a `char *` pointer to the first character in the literal is passed to the function.

Note that C does *not* provide any operators for processing an entire string as a unit! Arrays of characters are used, and a library of functions, defined in `<string.h>`, enables typical string manipulation.

Consider the following two declarations:

```
char amsg[] = "this is a string";
char *pmsg = "this is a string";
```

`amsg` is an array, just big enough to hold the sequence of characters and the 0-byte; this array is placed in read-write memory, and the characters in the array can be changed by subsequent logic. `pmsg` is a pointer, and it points to the first character of an array in read-only memory that holds the sequence of characters and the 0-byte; the characters in the array *cannot* be changed, but subsequent logic can cause `pmsg` to point to a different string in memory.

4.6.7 Pointer arrays - pointers to pointers

Since pointers are variables themselves, they can be stored in arrays just as other data types can. In fact, we have seen variables that are arrays of pointers earlier in the chapter - `argv`, the argument vector that is passed as the second argument in the call to `main`.

As you may recall, that argument was declared as `char *argv[]` - what does this mean? It means that `argv[0]` is of type `char *`; thus our previous assertion that `argv` is a list/array of strings; since strings are arrays of characters, then each array element is a pointer to an array of characters - i.e., points to the first character in the array.

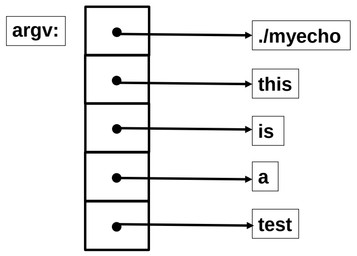
Suppose that you are asked to write a program that acts just like the standard `echo` command. Let's give it a try:

```
#include <stdio.h>    /* so we can access printf() */

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

    for (i = 0; i < argc; i++) {
        printf("%s%s", (i > 0) ? " " : "", argv[i]);
    }
    printf("\n");
    return 0;
}
```

Let's see if this works.

```
$ gcc -W -Wall -o myecho myecho.c
$ echo this is a test
this is a test
$ ./myecho this is a test
./myecho this is a test
```

Not exactly what we had in mind; it appears that `argv[0]` contains the command that you typed to `bash`, and that the standard `echo` command does not print that. No big problem, we have to make two simple changes to the code: 1)modify the initialization clause of the `for` loop to `i = 1`, and 2) modify `i > 0` in the first `printf()` invocation to `i > 1`.



The first `printf()` statement looks a bit weird, since it uses the ternary conditional. What exactly is it doing? Let's deconstruct it piece by piece:

- the format string is `"%s%s"`, which means that two additional arguments to `printf()` are expected, both strings;
- if this is not the first time through the for loop (`i > 1`), we want to put a single space; if not, we do not want to put a leading character; the ternary argument produces `" "` in the former case, and `""` in the latter;
- the appropriate `argv[]` value, which is a character pointer, is the final argument.

4.6.7.1 Initializing arrays of pointers

Suppose you wanted to define a list of keywords that your program would understand as user commands. For example, you have a simple image display program that supports the commands `up`, `down`, `left`, `right`, `zoom in`, and `zoom out`. You can declare and initalize an array of pointers to these keywords with the following code:

```
char *keywords[] = {
    "up",
    "down",
    "left",
    "right",
    "zoom in",
    "zoom out",
    NULL
};
```

Note the addition of `NULL` at the end of the array of pointers; we described earlier how one can put a *sentinel* value at the end of a compiler-constructed array so you know how big it is; `NULL` is an excellent value to use when you have an array of character pointers. By the way, `argv[]` actually is terminated by `NULL` in this way.

4.6.8 Multi-dimensional arrays

C provides rectangular multi-dimensional arrays, although in practice they are much less used than arrays of pointers. A multi-dimensional array is declared as follows:

```
int matrix[100][50];
```

This indicates that `matrix` has 100 rows, each with 50 elements. The value in `matrix` at the `i`th row and `j`th column is `matrix[i][j]`.

One can initialize a multi-dimensional array. The following code shows how to do so for a small, two-dimensional array.

```
int matrix[2][4] = {
    { 1, 2, 3, 4 },
    { 5, 6, 7, 8 }
};
```

4.6.9 Pointers to functions

A function itself is not a variable in C, but it is possible to define pointers to functions. These can be assigned to pointer variables, placed in arrays, passed as arguments to functions, returned by functions, etc. Let's look at one use of function pointers.

Consider a sort program that sorts strings in files, like the standard `sort` program in Linux. Sometimes, we want to sort the strings lexicographically (i.e., as character strings); at other times, there may be a number at the beginning of each line, and we would like the lines to be sorted numerically according to the leading number. As with the standard `sort` program, the user should be able to choose the type of sort by specifying a flag argument when invoking the program.

The pseudocode for our `main()` looks something like the following:

```
process command arguments
read all lines of input
```

```
sort them
print them in order
```

Assuming that there is a `sort()` function that performs the *sort them* part of the pseudocode, we need to have some way to inform that function how we want the strings to be compared. If the function prototype for `sort()` has the following form, we can achieve this flexible form of sorting.

```
void sort(char *lines[], int left, int right, int (*comp)(char *l1, char *l2))
```

What does this prototype tell us?

- `sort()` sorts the array of pointers to strings, `lines[]`;
- it actually sorts a slice of the array, specified by `left` and `right`; and
- whenever `sort()` needs to compare two entries in `lines[]`, it will invoke the function pointed to by `comp`; this function takes two `char *` arguments, and the return value from the function will be a value `< 0` if `l1 < l2`, the value `0` if `l1 == l2`, or a value `> 0` if `l1 > l2`. The function `strcmp()` in `<string.h>` is exactly such a function, and compares the strings lexicographically.

Suppose we have read `n` lines of text, such that `lines[0] .. lines[n-1]` have valid pointers. If we wanted to do a lexicographical sort, `main()` would invoke `sort()` as:

```
#include <string.h>

sort(lines, 0, n-1, strcmp);
```

Now suppose that we want to do a numeric sort. To do so, we will have to define a numeric compare function that matches the prototype for the function pointer argument in `sort()`. The following will do the trick.

```
#include <stdlib.h>

int numcmp(char *l1, char *l2) {
    int i1 = atoi(l1);
    int i2 = atoi(l2);
    return i1 - i2;
}
```

and `main()` would invoke `sort()` as

```
sort(lines, 0, n-1, numcmp);
```

4.6.9.1 Care when defining function pointers

Due to the precedence of C's operators, you have to be careful when defining function pointers. For example, consider the following function prototype:

```
int *f(void *arg);
```

defines a function named `f` that returns a pointer to an integer, while

```
int (*pf)(void *arg);
```

defines `pf` as a pointer to a function returning an integer.

In general, when defining a function pointer, you should always surround the name of the pointer (`*name`) by parentheses to avoid such mistakes. We will use function pointers when constructing abstract data types in C, so we will have plenty of practice.

4.7 Structures

A *structure* is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. A structure declaration looks as follows:

```
struct [tag] {
    declarations
};
```

The keyword `struct` introduces a structure declaration, which is a list of variable declarations enclosed in curly braces; the `tag` is an optional name that can be used to refer to this structure type in the future. The variables named in the declarations within the braces are called *members*.

A `struct` declaration defines a type; it is equivalent to a Python class definition for which there are no member functions. The right brace that terminates the list of members may be followed by a list of variable names, as in

```
struct { . . . } x, y, z;
```

A `struct` declaration that is *not* followed by a list of variables reserves no storage; it merely describes a template or the shape of a structure. In such a case, a `tag` must be specified in order to define instances of the structure later.

Suppose that we have defined the following `struct`:

```
struct point {
    int x;
    int y;
};
```

We can declare an instance of a `point` using

```
struct point pt;
```

We can initialize a structure in its declaration as in

```
struct point maxpt = { 320, 200 };
```

We can refer to a member of a particular `struct` as *structure-name.member-name*, as in

```
printf("(%d,%d)\n", pt.x, pt.y);
```

Structures can be nested; for example, if we represent a rectangle as a pair of points denoting diagonally opposite corners, we can define

```
struct rect {
    struct point ll;
    struct point ur;
};
```

If we declare `screen` as

```
struct rect screen;
```

then `screen.ll.x` refers to the `x` coordinate of the lower left corner (`ll`) of `screen`.

4.7.1 Legal operations on a structure

- Copy it as a unit.
- Assign to it as a unit.
- Pass it by value as a function argument.
- Return it by value as a function return value.
- Takes its address using `&`.
- Access its members.
- A global structure may be initialized using a list of constant member values.
- An automatic structure may be initialized using runtime expressions, as with automatic variables.
- It may *not* be compared with another structure using `==` or `!=`.

4.7.2 Pointers to structures

Passing large structures by value can be very inefficient. We can declare pointers to structures just as we do for built-in data types, as in

```
struct point *pp;
```

This indicates that `pp` is a pointer to structures of type `struct point`. If `pp` points to a point structure, then `*pp` is the structure itself, and `(*pp).x` and `(*pp).y` are the members.

Pointers to structures are so frequently used that an alternative notation is provided to access members. If `p` is a pointer to a structure, then `p->member-name` is equivalent to `(*p).member-name`.

4.7.3 Arrays of structures

Of course, we can create an array of structures. Recall our previous example of an array of keywords. Let's modify it slightly and show how it might be used:

```
struct key {
    char *word;
    int value;
};

struct key keywords[] = {
    { "up", 1},
    { "down", 2},
    { "left", 3},
    { "right", 4},
    { "zoom in", 5},
    { "zoom out", 6},
    { NULL, -1}
};

int mapKeyword(char *word) {
    int i;

    for (i = 0; keywords[i].word != NULL; i++)
        if (strcmp(word, keywords[i].word) == 0)
            return keywords[i].value;
    return keywords[i].value;
}
```

The function `mapKeyword()` maps from one of the string commands to an integer value. The code that solicited the string command from the user would call this routine, and then process the return value in a `switch` statement to perform the requested action.

4.7.4 Self-referential structures

As we shall see later in the text, many data structures that we use for common algorithms require that a structure contain one or more members that can *point at* instances of that structure - e.g., linked lists, binary trees. For a singly-linked list of integers, we would define the following structure type for nodes in the list:

```
struct node {
    struct node *next;
    int value;
};
```

In C, as soon as the compiler has seen `struct tag`, any subsequent code can refer to this tag; for members of that struct, one can declare pointers to instances of that structure. It makes no sense for a structure to contain a member which is an instance of that structure; where would the recursion end?

4.7.5 Typedefs

C provides a facility for creating new data type *names*. For example,

```
typedef int Length;
```

makes the name `Length` a synonym for `int`.

The type name `Length` can be used in declarations, casts, etc. in exactly the same way that `int` can be used; for example:

```
Length len, maxlen;
Length *lengths[25];
```

We can use `typedef` to also define synonyms for pointers.

```
typedef char *String;
```

makes `String` a synonym for `char *`.

The most common use of `typedef` is with respect to structures. Let's revisit our point structure and define a new type name for it.

```
typedef struct point {
    int x;
    int y;
} Point;
```

This particular style is extremely common - i.e., the tag for the structure starts with a lower-case letter, while the synonym for `struct tag` is `Tag`. With the above definition of `Point`, our definition for a rectangle can become

```
typedef struct rectangle {
    Point ll;
    Point ur;
} Rectangle;
```

and we can declare the variable `screen` as `Rectangle screen;`.

Note that `typedef` does *not* create a new type in any sense; it merely adds a synonym for some existing type. For example, we can declare variables as `struct point p1;` or as `Point p2;`. Both declarations achieve the same goal, of creating a variable that holds a `struct point` - i.e., both `p1` and `p2` have exactly the same properties.

It is also possible to create a synonym for function pointers. For example:

```
typedef int (*PFI)(char *, char *);
```

creates the type name `PFI` for "pointer to function (of two `char *` arguments) returning an `int`"; it could be used as in the following:

```
PFI strcmp, numcmp;
```

4.7.6 Structs and the heap

The `sizeof` compile time operator works with `struct`'s just like it does for built-in types. This enables us to create instances of our structures on the heap using `malloc()`.

Consider the following example:

```
typedef struct node {
    struct node *next;
    int val;
} Node;

Node *head = NULL;    /* head of singly-linked list */
Node *tail = NULL;    /* tail of singly-linked list */
```

```
int addNode(int value) { /* add node to tail of list */
    Node *n = (Node *)malloc(sizeof(Node));

    if (n != NULL) {
        n->val = value;
        n->next = NULL;
        if (head == NULL)
            head = n;
        else
            tail->next = n;
        tail = n;
        return 1;    /* return success indication */
    } else {
        return 0;    /* return failure indication */
    }
}
```

The call to `malloc()` looks just like those we have seen before. Through the cast, we have a pointer to our structure, and can manipulate the members of the allocated structure. This example is for a singly-linked list, which we will see later in the textbook.

4.7.7 Unions

A *union* is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements. Unions enable the manipulation of different kinds of data in a single area of storage, without embedding any machine-dependent information in your program.

Consider the symbol table for a compiler; the program could have integer, floating point, or character string literals; we would like to define a single structure type for a constant in our symbol table. The value of a particular constant must be stored in a variable of the appropriate type, but it is most convenient for table management if the value occupies the same amount of storage and is stored in the same place, regardless of its type.

The syntax is similar to that for structures. For this symbol table application:

```
union constValue {
    int ival;
    double dval;
    char *sval;
} u;
```

defines a variable named `u` which at different times can hold an integer, a double-precision floating point number, or a pointer to a character array. The tag identifier, in this case `constValue`, is optional. `u` will be large enough to hold the largest of the three types. Any one of these types may be assigned to `u` and then used in expressions.

Of course, your program must have some way to keep track of what type of value the `union` currently holds. One normally creates a structure, called a *discriminated union*, which consists of a member for keeping track of the type of value in the `union`, and a member which is the union. Given the definition of `constValue` above,

```
#define INTVALUE 1
#define DOUBLEVALUE 2
#define STRINGVALUE 3

struct sTableEntry {
    int uType;
    union constValue u;
} s;
```

Suppose that our program had read the integer constant `42` in its input, and needed to place that information in `s` declared above. The following two lines would do the trick:

```
s.uType = INTVALUE;
s.u.ival = 42;
```

Given the definition of `sTableEntry` above, the following code would display the current value in `s` on standard output:

```
switch(s.uType) {
    case INTVALUE:    printf("%d\n", s.u.ival); break;
    case DOUBLEVALUE: printf("%f\n", s.u.dval); break;
    case STRINGVALUE: printf("%s\n", s.u.sval); break;
    default:          printf("illegal discriminator value - %d\n", s.uType);
}
```

As with structures, pointers to unions can be created/used, as can `malloc()` be used to dynamically allocate a union on the heap.

4.8 Input and output

Input and output facilities are not part of the C language itself. The standard I/O library, defined in `<stdio.h>`, provides the ANSI standard library of I/O functions. A program comes to life with standard input (`stdin`), standard output (`stdout`), and standard error output (`stderr`) predefined. By default, `stdin` is associated with your keyboard, and `stdout` and `stderr` are associated with your terminal window. If the program was invoked with I/O redirection in the command line, the associated standard streams will point to the file or pipe specified.

4.8.1 Single character input and output

The simplest input mechanism is to read one character at a time from `stdin`, using `getchar()`:

```
int getchar(void);
```

`getchar()` returns the next input character from `stdin` each time it is called, or `EOF` when it encounters the end of file.

The function

```
int putchar(int ch);
```

puts the character `ch` onto `stdout`. We have previously seen that `printf()` also prints its output on `stdout`. Calls to `putchar()` and `printf()` can be interleaved - output appears in the order in which the calls were made.

4.8.2 Formatted input - `scanf()`

The function `scanf()` is the input analog to `printf()`, providing many of the same conversion facilities in the opposite direction:

```
int scanf(char *format, ...);
```

`scanf()` reads characters from standard input, interprets them according to the specification in `format`, and stores the results in the remaining arguments. Note that all of the arguments into which `scanf()` stores the results *must* be pointers, as we discussed in section 4.6.2. `scanf()` stops when it reaches the end of the format string, or when some input fails to match the control specification in the format string. `scanf()` returns as its value the number of successfully matched and assigned input items. If an end of file is detected while scanning, `EOF` is returned. The next call to `scanf()` resumes scanning standard input immediately after the last character scanned in the current call.

The `scanf()` format string usually contains conversion specifications, which are used to control conversion of input. It may contain:

- blanks or tabs, which are ignored;

- ordinary characters (not `%`), which are expected to match the next non-white-space character of the input stream;
- conversion specifications, consisting of the character `%`, an optional assignment suppression character `*`, an optional number specifying a maximum field with, an optional `h`, `l` or `L` indicating the width of the target, and a conversion character.

The basic `scanf()` conversions are shown in the following table.

character	**input data**	**argument type**	**comment**
`d`	decimal integer	`int *`	`ld` -> `long *`, `Ld` -> `long long *`
`i`	integer	`int *`	the integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X); same `l` and `L` modifiers as for decimal integer
`o`	octal integer	`int *`	with or without leading 0; `lL` modifiers as for decimal integer
`u`	unsigned decimal integer	`unsigned *`	`lu` -> `unsigned long *`, `Lu` -> `unsigned long long *`
`x`	hexadecimal integer	`int *`	with or without leading 0x/0X; `lL` modifiers as for decimal integer
`c`	character	`char *`	the next input character is copied; the normal skip over white space is suppressed
`s`	unquoted character string	`char *`	address of an array of characters large enough for the string and a terminating `'\0'`
`e,f,g`	floating point number	`float *`	with optional sign, optional decimal point and optional exponent; `lf` -> `double *`, `Lf` -> `long double *`
`%`	literal %	no assignment	

Here are a couple of examples.

- Suppose we want to read input lines that contain dates of the form "dd Month yyyy":

```
int day, year;
char monthname[20];

if (scanf("%d %s %d", &day, monthname, &year) != 3) {
    printf("input was not in the form dd Month yyyy\n");
}
```

- Now suppose that the required format is "mm/dd/yyyy":

```
int day, month, year;
```

```
        if (scanf("%d %d %d", &month, &day, &year) != 3) {
            printf("input was not in the form mm/dd/yyyy\n");
        }
```

4.8.3 File access

Given the name of a file as a string, one can open it for reading/writing, read from it or write to it, and close it. `<stdio.h>` defines a stream type `FILE *`; a successful file open returns one of these streams, the I/O routines and the close routine take one of these streams as an argument. `<stdio.h>` defines `stdin`, `stdout`, and `stderr` as instances of a `FILE *`. The function prototypes are as follow.

prototype	**comment**
`FILE *fopen(char *name, char *mode);`	most common mode values are `"r"` to open for reading, `"w"` to open for writing (overwriting any existing contents), and `"a"` to open for appending
`int getc(FILE *fp);`	return next character from stream, returning `EOF` if end of file
`int putc(int c, FILE *fp);`	write character to stream
`int fclose(FILE *fp);`	close stream
`int fscanf(FILE *fp, char *format, ...);`	scan stream according to `format`
`int fprintf(FILE *fp, char *format, ...);`	output values to stream according to `format`
`char *fgets(char *buf, int size, FILE *fp);`	fetch next line, including `'\n'` into buf, returning `buf` if successful, `NULL` if end of file
`int fputs(char *buf, FILE *fp);`	output characters in `buf` to stream

There are also versions of `scanf()` and `printf()` that work with characters buffers instead of `FILE *` streams.

```
    int sscanf(char *buf, char *format, ...);
    int sprintf(char *buf, char *format, ...);
```

Finally, while functions in the `scanf()` family return the number of conversions that were successfully completed, the functions in the `printf()` family return the number of characters written to the stream/buffer; `sprintf()` always writes a terminating 0-byte, but does not include it in the returned count. Thus, the following pattern is sometimes seen in code that is constructing a complex string in a buffer.

```
    char *p, buf[4096];

    p = buf;
    p += sprintf(p, format1, v1, v2, ..., vn);
    p += sprintf(p, format2, w1, w2, ..., wm);
```

```
p += sprintf(p, format3, x1, x2, ..., xo);
* * *
/* buf contains the concatenated formatted outputs */
```

4.9 Debugging your programs

The GNU debugger, usually called GDB and named `gdb` as an executable file, is the standard debugger for the Linux operating system. It can be used to debug programs written in a number of programming languages, including C, C++, Objective-C, Fortran, Java, and many others.

GDB was first written by Richard Stallman in 1986 as part of the GNU system. It is free software, released under the GNU General Public License, and is included in all Linux distributions.

GDB enables you to inspect what a program is doing at certain points of execution. Errors like *segmentation faults* are often quite easy to find with the help of `gdb`.

This section provides a brief introduction into the use of `gdb`. Complete documentation for `gdb` may be obtained from `https://www.gnu.org/software/gdb/documentation/`.

4.9.1 Preparing your program for use with gdb

Normally, you compile your program as follows:

```
$ gcc [flags] -o <executable file> <source files>
```

As advised in Section 4.1.1, `[flags]` should be replaced by `-W -Wall`. In order for the `executable file` to be used with `gdb`, one must add an additional flag, `-g`, to the command line, as in the following example:

```
$ gcc -W -Wall -g -o prog prog.c
```

You can now execute `prog` under `gdb`'s control using the following command:

```
$ gdb prog
```

This command simply starts up `gdb`; section 4.9.2 describes how to proceed with actually running the program under `gdb`.

`gdb` provides an interactive shell, enabling you to recall history using the arrow keys and to auto-complete (most) words using the `TAB` key. At any time, you may ask `gdb` for help with a command by typing the following to the `gdb` prompt:

```
(gdb) help command
```

4.9.2 Running the program under `gdb`

To run your program under `gdb`, you type the following command:

```
(gdb) run [arguments]
```

If your program requires arguments, obtained through `argv[]` in `main()`, you must specify them after the `run` command. You may also specify I/O redirection (`<file` and/or `>file`) along with the command arguments.

This runs the program - if there are no serious problems, the program should run to successful completion. If the program has issues, `gdb` assumes control after the program unsuccessfully terminates, and displays some useful information about the program, such as the line number where it terminated, parameters to the enclosing function, etc.

Consider the following program in a file `test.c`:

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

#define UNUSED __attribute__((unused))

size_t foo_len(const char *s) {
    return strlen(s);
}

int main(UNUSED int argc, UNUSED char *argv[]) {
    const char *a = NULL;

    printf("size of a = %ld\n", foo_len(a));
    return 0;
}
```

The following dialog shows execution of this program using `gdb`:

```
GNU gdb (GDB) 7.9
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./test...done.
(gdb) run
```

```
Starting program: /home/cis415/LPE/gdb/test

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7ab820a in strlen () from /usr/lib/libc.so.6
(gdb)
```

As you can see, we encountered a *segmentation fault* in `strlen()`, which caused the program to terminate. You may recall that segmentation faults result from attempting to access virtual address `0`. Thus, it is likely that the argument to `strlen()` is a NULL pointer. Since we define `a` to be `NULL` in `main()`, pass `a` to `foo_len()`, which then passes `a` on to `strlen()`, we have found the source of our segmentation fault. [9]

If we change the declaration for `a` to the following:

```
const char *a = "This is a test string";
```

the following dialog results:

```
GNU gdb (GDB) 7.9
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./test...done.
(gdb) run
Starting program: /home/cis415/LPE/gdb/test
size of a = 21
[Inferior 1 (process 26063) exited normally]
(gdb)
```

To exit from `gdb`, one uses the `quit` command. If you are attempting to `quit` after a program has terminated successfully, `gdb` lets you `quit` directly, as in:

```
(gdb) quit
$
```

If, on the other hand, the program has terminated unsuccessfully, `gdb` requests that you verify your intention to quit, as in:

```
(gdb) quit
```

[9] Note that if we had simply invoked `test` in the shell, the shell would have yielded the relatively unhelpful message of

```
Segmentation fault (core dumped)
```

```
A debugging session is active.

        Inferior 1 [process 26028] will be killed.

Quit anyway? (y or n) y
$
```

4.9.3 Other useful commands

If all `gdb` provided was extra information about the source of the fault that killed your program, it would not have to be interactive. The real power of a debugger is that it lets you stop the execution of your program, inspect the contents of memory, and view other aspects of your program as it executes.

One useful command to `gdb` is to inspect the sequence of calls that resulted in the faulty termination of your program. This command is `bt` (for *backtrace* – `backtrace` is also a legal command to obtain this functionality). Let's ask for the backtrace after our faulty program above exits with a segmentation fault:

```
(gdb) bt
#1  0x00007ffff7ab8201 in strlen () from /usr/lib/libc.so.6
#2  0x000000000040055e in foo_len (s=0x0) at test.c:8
#3  0x0000000000400583 in main (argc=1, argv=0x7fffffff3878) at test.c:14
(gdb)
```

For this simple example, the backtrace corroborates our logic above regarding the source of the segmentation fault.

A particularly useful command to `gdb` is to set a *breakpoint* in the program; this is done with a command of the form:

```
(gdb) break location
```

When your program reaches a breakpoint, it will pause and control returns to `gdb`. You must set your initial breakpoint[s] before you `run` the program; you may introduce additional breakpoints whenever your program has paused and `gdb` has regained control.

Let's set a breakpoint in our faulty program before we execute it [10]:

```
(gdb) break foo_len
Breakpoint 1 at 0x400552: file test.c, line 8.
(gdb) run
Starting program: /home/cis415/LPE/gdb/test

Breakpoint 1, foo_len (s=0x0) at test.c:8
8     return strlen(s);
(gdb) print s
$1 = 0x0
```

[10] The boilerplate text from `gdb` will no longer be shown in dialogs.

```
    (gdb)
```

We specified that a breakpoint should be set at the start of `foo_len()`. Alternatively, since we know that the function starts at line 8 of test.c, we could have specified

```
    (gdb) break test.c:8
```

In fact, you can set a breakpoint at any statement in your source files, they do not have to coincide with the start of a function. Since you typically will not know the line numbers for your source files, setting breakpoints at the start of a function is a very easy way to pause your program before a bug manifests itself.

When we issue the `run` command, the program starts to execute `main()`; when `foo_len()` is invoked, control returns to `gdb`. At the `gdb` prompt, we may then issue other commands. In this case, we invoked the `print` command to see what the value is for the argument `s`, which is shown to be `0x0`; note that the function arguments are provided when the breakpoint is encountered. Note also that when we asked `gdb` to print the value of `s`, it printed '`$1 = 0x0`'; we could equally have asked it to print the value of `$1`, as `gdb` represents the arguments to a function positionally as `$1`, `$2`, ...

You can use the `print` command to not only display the values of function arguments by name, but to any local or global variables by name, as well. If a variable is a pointer, say '`struct foo *p;`', specifying `print *p` to `gdb` will cause the contents of the `struct foo` to which `p` currently points to be printed out.

Any time `gdb` regains control after the program starts running, we can continue the execution of the program by issuing the `c` (for *continue* – `continue` is also a legal command to obtain this functionality) command.

We can also single step our program after execution is paused. There are two forms of the single step functionality:

- The next line of code is executed by specifying the `step` command. This will execute *just* the next line of code. If the next line of code contains a function invocation, `step` *steps* into that function. This enables you to dive deep into a sequence of call frames to get to the bottom of your problem.
- You can also execute the next line of code by specifying the `next` command. If the next line of code contains a function invocation, using `next` does *not* step into that function. Thus, `next` simply enables you to step through the current function.

Typing `step` or `next` many times is extremely tedious. `gdb` interprets a bare carriage return to mean "re-execute the previous command". Thus, if you need to step/next through many lines of code, repeatedly typing `RETURN` will eliminate a bit of the tedium.

We can clear a breakpoint by issuing the `clear` command; this command requires the same argument as used in a `break` command to set the breakpoint. In our example above, a '`clear foo_len`' command will remove the breakpoint set upon entry to `foo_len()`.

Breakpoints permit `gdb` to regain control when a statement is about to be executed.

Another way for `gdb` to regain control is through *watchpoints*. A watchpoint allows you to monitor the values of variables, pausing the program when a watched variable changes value.

To set a watchpoint, you use the `watch` command:

```
(gdb) watch my_var
```

Now, whenever `my_var`'s value is modified, the program is paused and the old and new values are printed. `gdb` interprets the scope of the variable name in a `watch` command based upon the program scope at the time the `watch` command is executed. This means that you can set watchpoints for global variables before running the program; if you wish to set a watchpoint for a static local variable within a function, you need to set a breakpoint for that function, set a watchpoint for that variable when the function is entered the first time, and then clear the breakpoint.

There are a multitude of other features provided by `gdb`. Please consult materials at `https://www.gnu.org/software/gdb/documentation/` as you become familiar with `gdb`.

4.10 Managing heap memory

Valgrind is a program-execution monitoring framework. It comes with many tools; the tool upon which we are focused is the *memcheck* tool.

Memcheck detects and reports the following types of memory errors:

- Use of unitialized memory.
- Reading/writing to heap memory after it has been freed.
- Reading/writing off the end of malloc'd blocks.
- Heap allocated memory leaks.
- Mismatched use of malloc vs free.
- And many more ...

4.10.1 Invoking `valgrind`

To test whether '`prog arguments`' correctly allocates and uses memory, we run it with `valgrind` as follows:

```
$ valgrind prog arguments
```

As with `gdb`, the source files must be compiled with the `-g` flag to `gcc`. `valgrind` supports a large number of options; the user manual may be consulted at `http://valgrind.org/docs/manual/manual.html`.

4.10.2 Use of unitialized memory

valgrind keeps track of each program variable and each block allocated by malloc() to determine when that variable/block has been written to. If your program attempts to read from a variable or heap block before it has been initialized, valgrind will flag each occurrence appropriately.

Consider the following program.

```
#include <stdlib.h>

#define UNUSED __attribute__((unused))

int main(UNUSED int argc, UNUSED char *argv[]) {
    int x, y;

    if (x < 3)          /* x not initialized */
        y = 4;
    else
        y = 5;
    return 0;
}
```

When executed by valgrind, the following output results:

```
==26569== Memcheck, a memory error detector
==26569== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==26569== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==26569== Command: ./uninit
==26569==
==26569== Conditional jump or move depends on uninitialised value(s)
==26569==    at 0x4004C5: main (uninit.c:8)
==26569==
==26569==
==26569== HEAP SUMMARY:
==26569==     in use at exit: 0 bytes in 0 blocks
==26569==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==26569==
==26569== All heap blocks were freed -- no leaks are possible
==26569==
==26569== For counts of detected and suppressed errors, rerun with: -v
==26569== Use --track-origins=yes to see where uninitialised values come from
==26569== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

As you can see, valgrind indicates that there an unitialized value was used in main, at line 8 of uninit.c.

4.10.3 Reading/writing to heap memory after it has been freed

`valgrind` keeps track of each `malloc()` allocated block to determine when it is returned to the heap. Attempts to access an already freed heap block cause `valgrind` to flag each occurrence appropriately.

Consider the following program.

```
#include <stdlib.h>
#include <string.h>
#define UNUSED __attribute__((unused))

int main(UNUSED int argc, UNUSED char *argv[]) {
    char *s, *str="this is a string";
    int c, n;

    n = strlen(str) + 1;
    s = (char *)malloc(n);
    free(s);
    strcpy(s, str);       /* accessing freed heap memory */
    return 0;
}
```

When executed by `valgrind`, the following output results:

```
==26628== Memcheck, a memory error detector
==26628== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==26628== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==26628== Command: ./alreadyfree
==26628==
==26628== Invalid write of size 1
==26628==    at 0x4C2D610: strcpy (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==26628==    by 0x40062E: main (alreadyfree.c:12)
==26628==  Address 0x51d9040 is 0 bytes inside a block of size 17 free'd
==26628==    at 0x4C2B200: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==26628==    by 0x40061B: main (alreadyfree.c:11)
==26628==
==26628== Invalid write of size 1
==26628==    at 0x4C2D623: strcpy (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==26628==    by 0x40062E: main (alreadyfree.c:12)
==26628==  Address 0x51d9050 is 16 bytes inside a block of size 17 free'd
==26628==    at 0x4C2B200: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==26628==    by 0x40061B: main (alreadyfree.c:11)
==26628==
==26628==
==26628== HEAP SUMMARY:
==26628==     in use at exit: 0 bytes in 0 blocks
==26628==   total heap usage: 1 allocs, 1 frees, 17 bytes allocated
==26628==
==26628== All heap blocks were freed -- no leaks are possible
==26628==
==26628== For counts of detected and suppressed errors, rerun with: -v
```

```
==26628== ERROR SUMMARY: 17 errors from 2 contexts (suppressed: 0 from 0)
```

Apparently, `strcpy` attempts to access the first and last byte of the destination string, thus causing `valgrind` to flag two invalid writes.

4.10.4 Reading/writing off the end of malloc'd blocks

`valgrind` keeps track of the length of each `malloc()` allocated block to check for attempted access beyond the allocation. Such attempts cause `valgrind` to flag each occurrence appropriately.

Consider the following program.

```
#include <stdlib.h>
#include <string.h>
#define UNUSED __attribute__((unused))

int main(UNUSED int argc, UNUSED char *argv[]) {
    char *s, *str="this is a string";
    int c, n;

    n = strlen(str);
    s = (char *)malloc(n);      /* no space for 0-byte */
    strcpy(s, str);             /* invalid write error */
    c = s[n];                   /* invalid read error */
    free(s);
    return 0;
}
```

When executed by `valgrind`, the following output results:

```
==26596== Memcheck, a memory error detector
==26596== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==26596== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==26596== Command: ./overrun
==26596==
==26596== Invalid write of size 1
==26596==    at 0x4C2D623: strcpy (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==26596==    by 0x40061F: main (overrun.c:11)
==26596==  Address 0x51d9050 is 0 bytes after a block of size 16 alloc'd
==26596==    at 0x4C29F90: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==26596==    by 0x400608: main (overrun.c:10)
==26596==
==26596== Invalid read of size 1
==26596==    at 0x40062D: main (overrun.c:12)
==26596==  Address 0x51d9050 is 0 bytes after a block of size 16 alloc'd
==26596==    at 0x4C29F90: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==26596==    by 0x400608: main (overrun.c:10)
==26596==
==26596==
==26596== HEAP SUMMARY:
```

```
==26596==     in use at exit: 0 bytes in 0 blocks
==26596==   total heap usage: 1 allocs, 1 frees, 16 bytes allocated
==26596==
==26596== All heap blocks were freed -- no leaks are possible
==26596==
==26596== For counts of detected and suppressed errors, rerun with: -v
==26596== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

This program shows the typical cause of these overrun problems – forgetting to allocate space for the 0-byte at the end of a string. Another common source of this problem is to allocate an array of *n* items in a heap memory block, and then attempt to use *n* as an index into that block, forgetting that the array is indexed by *0 .. n-1*.

4.10.5 Heap allocated memory leaks

`valgrind` keeps track of each `malloc()` allocated block to determine when it is returned to the heap. Attempts to terminate the program when there are still outstanding allocations cause `valgrind` to flag those occurrences.

Consider the following program.

```
#include <stdlib.h>
#include <string.h>
#define UNUSED __attribute__((unused))

int main(UNUSED int argc, UNUSED char *argv[]) {
    char *s, *str="this is a string";
    int c, n;

    n = strlen(str) + 1;
    s = (char *)malloc(n);
    strcpy(s, str);
    return 0;        /* forgot to free heap memory */
}
```

When executed by `valgrind`, the following output results:

```
==26646== Memcheck, a memory error detector
==26646== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==26646== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==26646== Command: ./memleak
==26646==
==26646==
==26646== HEAP SUMMARY:
==26646==     in use at exit: 17 bytes in 1 blocks
==26646==   total heap usage: 1 allocs, 0 frees, 17 bytes allocated
==26646==
==26646== LEAK SUMMARY:
==26646==    definitely lost: 17 bytes in 1 blocks
==26646==    indirectly lost: 0 bytes in 0 blocks
==26646==      possibly lost: 0 bytes in 0 blocks
```

```
==26646==    still reachable: 0 bytes in 0 blocks
==26646==         suppressed: 0 bytes in 0 blocks
==26646== Rerun with --leak-check=full to see details of leaked memory
==26646==
==26646== For counts of detected and suppressed errors, rerun with: -v
==26646== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

If you wish to see details of leaked memory, use the following invocation:

```
$ valgrind --leak-check=full prog arguments
```

4.10.6 Mismatched use of malloc vs free

`valgrind` keeps track of each `malloc()` allocated block to determine when it is returned to the heap. Attempts to free memory that is not allocated (e.g. has already been `free()`'ed or is an address that does not reside on the heap) cause `valgrind` to flag those occurrences.

Consider the following program.

```
#include <stdlib.h>
#include <string.h>
#define UNUSED __attribute__((unused))

int main(UNUSED int argc, UNUSED char *argv[]) {
    char *s, *str="this is a string";
    int c, n;

    n = strlen(str) + 1;
    s = (char *)malloc(n);
    strcpy(s, str);
    free(s);
    free(s);              /* freed a second time */
    return 0;
}
```

When executed by `valgrind`, the following output results:

```
==26612== Memcheck, a memory error detector
==26612== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==26612== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==26612== Command: ./dblfree
==26612==
==26612== Invalid free() / delete / delete[] / realloc()
==26612==    at 0x4C2B200: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==26612==    by 0x40063A: main (dblfree.c:13)
==26612==  Address 0x51d9040 is 0 bytes inside a block of size 17 free'd
==26612==    at 0x4C2B200: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==26612==    by 0x40062E: main (dblfree.c:12)
==26612==
==26612==
==26612== HEAP SUMMARY:
==26612==     in use at exit: 0 bytes in 0 blocks
==26612==   total heap usage: 1 allocs, 2 frees, 17 bytes allocated
==26612==
==26612== All heap blocks were freed -- no leaks are possible
==26612==
==26612== For counts of detected and suppressed errors, rerun with: -v
==26612== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

4.10.7 General advice

`valgrind` is an enormously useful tool. Nearly every sophisticated C program makes extensive use of the heap; it is essential that all memory allocation and use errors be eliminated from your programs.

The previous sections have demonstrated how different types of memory misuse manifest themselves in the output from `valgrind`. The only time you should be satisfied that you have eliminated all problems with your memory use is if the output from `valgrind` looks like the following dialog.

```
==26596== Memcheck, a memory error detector
==26596== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==26596== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==26596== Command: ./allok
==26596==
==26596== HEAP SUMMARY:
==26596==     in use at exit: 0 bytes in 0 blocks
==26596==   total heap usage: 1 allocs, 1 frees, 16 bytes allocated
==26596==
==26596== All heap blocks were freed -- no leaks are possible
==26596==
==26596== For counts of detected and suppressed errors, rerun with: -v
==26596== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Chapter 5

Abstract Data Types

5.1 What are they?

In computer science, an *abstract data type (ADT)* is a mathematical model for data types, where a data type is defined by its behavior (semantics) from the point of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. This contrasts with data structures, which are concrete representations of data, and are the point of view of an implementer, not a user.

Formally, an ADT may defined as a "class of objects whose logical behavior is defined by a set of values and a set of operations"; this is analogous to an algebraic structure in mathematics. ADTs are a theoretical concept in computer science, used in the design and analysis of algorithms, data structures, and software systems, and do *not* correspond to specific features of computer languages - mainstream computer languages do not directly support formally-specified ADTs. However, various language features, such as abstract base classes, correspond to certain aspects of ADTs. ADTs were first proposed by Liskov and Zilles in 1974, as part of the CLU language.

5.2 How do they compare to objects?

Object-oriented languages enable you to create your own data types. In most of these languages, you define a class as a template for instances of that class. The template specifies data elements associated with each instance, which may be public or private, and a set of methods that can be invoked on each instance. Most of these languages support inheritance, in which one can *inherit* implementation aspects from one (or more) parent classes.

Class specifications are a mixture of specification and implementation. Some object-oriented languages support *pure virtual classes*, which specify methods that must be supported but do *not* provide any implementation; such pure virtual classes are usually called *abstract base classes*. A class which inherits from an abstract base class must implement the methods that are defined in the abstract base class.

Since an ADT is defined by its behavior (the methods it supports and the results of invoking those methods), it acts like a pure virtual class/abstract base class. Since C does not support inheritance, our approach to ADTs consists of two parts:

- we define the behavior of an ADT through a header file which defines function prototypes for the methods and actions performed when one invokes one of these methods; a program that wishes to use instances of a particular ADT will `#include` the corresponding header file; and
- an implementation of an ADT is provided through a source file; when the user has created instances of an ADT and invoked methods on those instances, the user must link an implementation into their program.

It is important to note that the implementation of the ADT is hidden from the user; the header file provides the "contract" between the user and the implementation. That is why we have used the indefinite article (an) when discussing the implementation. One of the strengths of ADTs is that there can be multiple implementations of the abstract type; these different implementations may have different characteristics in terms of space and time complexity. We will cover this in more detail later in the book.

5.3 A well-structured approach to ADTs in C

This section describes a well-structured approach to constructing and using ADTs in C. It is not the only way to do it, but it provides similar syntax and structure to that provided by object-oriented languages such as Java, C++, and Python.

Every ADT will have a constructor function; when invoked, the constructor will return a pointer to a data structure in which private data elements are stored, and which contains function pointers to all of the methods defined in the ADT. Since the data elements are private (to the user), alternative implementations of an ADT are readily supported.

5.3.1 Alternative implementations behind an interface

Let's look back at the header and source files for `isSubString()` in the last chapter.

issubstring.h
```
int isSubString(const char *needle, const char *haystack);
```

issubstring.c
```
#include "issubstring.h"
#include <string.h>
```

```
int isSubString(const char *needle, const char *haystack) {
    return strstr(haystack, needle) != NULL;
}
```

We defined the function prototype for `isSubString()` in the header file; any program that wishes to use the function must call it according to that prototype, and must link the object file associated with compilation of an implementation.

We could have implemented `isSubString()` differently, as in

issubstring.c (alternative version)

```
#include "issubstring.h"
#include <string.h>

int isSubString(const char *needle, const char *haystack) {
    int nl = strlen(needle);
    int hl = strlen(haystack);
    int i;

    for (i = 0; i <= (hl - nl); i++)
        if (strncmp(needle, haystack+i; nl) == 0)
            return 1;
    return 0;
}
```

Thus we can see that specifying the prototype for one or more functions in a header file already gives us the support we need to use different implementations of those functions.

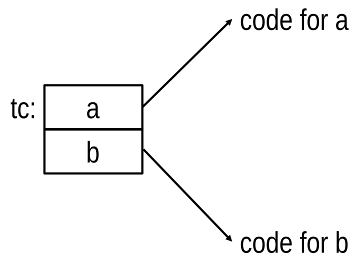
5.3.2 Dispatch table basics

If you think about classes in Python, when you define a class (we'll refer to it as `TheClass`), you have to define an `__init__()` method which is called when your code invokes the constructor, `TheClass()`; assume we have done so, and assigned the return value from the constructor to a variable named `tc`.

Suppose `TheClass` also defines methods named `a()` and `b()`; if you want to invoke `a()` on `tc`, you invoke it as `tc.a()`. A similar syntax is used to invoke `b()`.

In a sense, when we define `TheClass`, we define a global function `TheClass()`, and an instance of the class contains function pointers to `a()` and `b()`, so that they can be invoked from `tc`. A data structure that contains function pointers is often called a *dispatch table*; the dispatch table for this simple Python example is shown below.

```
class TheClass:
    def __init__(self, arguments):
        # statements to initialize `self'

    def a(self):
        # statements to implement fxn

    def b(self):
        # statements to implement fxn

tc = TheClass()
```



In Python, when you invoke `tc.a()`, the interpreter actually rewrites the call to `a()`, passing `tc` to `a()` as the first argument, `self`. This is needed, since the code in `a()` may need to access instance-specific data items that are stored in `tc` in order to do its work.

In a similar fashion, each C ADT will have a constructor function, which returns a pointer to a `struct` that has:

- a `void *` pointer to another struct in which the instance-specific data is kept; and
- a function pointer for each method defined for the ADT.

Let's assume that we have a C implementation of `TheClass`. There will be a header file, `theclass.h`, that defines the constructor function prototype and the structure that is the dispatch table.

The constructor function prototype has the following signature:

```
typedef struct theClass TheClass;  /* forward reference */
const TheClass *TheClass_create(/* args needed */);
```

We first declare that `TheClass` is a synonym for `struct theClass`. Do not worry, we will define `struct theClass` next. Note that we have declared that the pointer returned by the constructor has the `const` attribute. This means that the compiler can help prevent the user from modifying the fields in the structure which represents the instance of `TheClass`.

Next we need to define `struct theClass`.

```
/* now define the dispatch table */
struct theClass {
    /* the private data of the instance */
    void *self;

    /* function pointer to method a() */
    void (*a)(const TheClass *tc);

    /* function pointer to method b() */
    void (*b)(const TheClass *tc);
};
```

This states that `TheClass_create()` will return a pointer to this structure; the `self` member points at the private data of the instance, and the remaining members of the structure are function pointers to member functions.

How would a user program create an instance of `TheClass` and use it?

```
#include "theclass.h"

const TheClass *tc = TheClass_create(/* args */);

tc->a(tc);
tc->b(tc);
```

Having to specify the `tc` twice seems a bit repetitive, but we do not have something like the Python interpreter rewriting our statements for us. Additionally, this looks a bit like the invocation of an external object in C++, except for having to provide *self* in each method call.

5.3.3 The interface (header file)

The previous section showed parts of the header file for `TheClass`. Here it is in its entirety.

```
#ifndef _THECLASS_H_
#define _THECLASS_H_

typedef struct theClass TheClass;  /* forward reference */
const TheClass *TheClass_create(/* args needed */);

/* now define the dispatch table */
struct theClass {
    /* the private data of the instance */
    void *self;

    /* function pointer to method a() */
    void (*a)(const TheClass *tc);

    /* function pointer to method b() */
    void (*b)(const TheClass *tc);
};

#endif /* _THECLASS_H_ */
```

Let's enumerate the contents again:

- `#ifndef ... #define ... #endif` wrapper
  this enables you to `#include` this header file wherever it makes sense in your source code and other header files *without* having to worry if the compile of a single source file will somehow `#include` the file more than once, thus attempting to redefine function signatures and structure declarations;
- `typedef struct theClass TheClass;`
  this simply establishes `TheClass` as a synonym for `struct theClass` so `TheClass` can be used in the signatures that follow;
- `const TheClass *TheClass_create(/* args needed */);`
  this is the function prototype for the constructor; `/* args needed */` simply indicates that each ADT's constructor will be defined to take the appropriate ADT-specific arguments in order for the constructor to create an ADT instance; if successful, the constructor returns a pointer to the dispatch table for the ADT instance; if unsuccessful, it returns `NULL`;
- `struct theClass { ... };`
  the structure declaration defines the members of the dispatch table; in particular,
  - `void *self;`
    defines a pointer to a structure that contains the instance-specific data; it is a `void *` to hide the shape and contents of that structure, thus enabling different implementations of the ADT to be used, since the caller has no dependency upon the implementation-dependent data;
  - `void (*a)(const TheClass *tc);`
    `void (*b)(const TheClass *tc);`
    for each method, a function pointer member of the dispatch table is defined with the appropriate signature; each such method must have a first argument of type `const TheClass *` - i.e., the dispatch table structure returned from the constructor.

5.3.3.1 Mandatory method[s] in an ADT interface

Some object-oriented languages, such as Python, advise that one provide some standard methods in your class if you want it to be "pythonic". These include `__str__()` which is called whenever a programmer invokes `str(object)` on your object, `__iter__()` to return an iterator over your object, `__getitem__()` if your object can naturally be indexed, and many others.

The constructor for an ADT will have to allocate the instance-specific data structure *and* the dispatch table from the heap. Since C is not garbage-collected, we are going to mandate that *every* ADT interface must define a method of the following form:

```
/* destroys the TheClass instance */
void (*destroy)(const TheClass *tc);
```

The implementation of this method must free all heap memory that was allocated to implement the class instance: the dispatch table, the instance-specific structure `self`, *and* any other heap memory that was allocated and pointed to by `self`.

5.3.3.2 Mandatory method[s] in the ADT interface of a container

Later in the book, we will see that many of our data structures act as containers for user data. The `destroy` signature for those ADTs will have an additional parameter:

```
void (*destroy)(const TheClass *tc, void (*freeFxn)(void *element));
```

The `freeFxn` parameter is a pointer to a function that knows how to free up any storage associated with each element that is contained within the container; this function will be called for each element *before* `self` and the dispatch table are returned to the heap. If you specify the 2nd parameter as `NULL`, nothing will be done to the contained elements.

Container ADTs will also define and implement several other mandatory methods, shown below.

```
/* clear container; freeFxn has same semantics as for destroy;
   empty container upon return */
void (*clear)(const TheClass *tc, void (*freeFxn)(void *element));

/* returns the number of elements in the container */
long (*size)(const TheClass *tc);

/* returns true if the container is empty, false if not */
int (*isEmpty)(const TheClass *tc);

/* returns array of void * pointers to elements;
   # of elements returned in *len; NULL if heap failure */
void **(*toArray)(const TheClass *tc, long *len);

/* return iterator over the container, NULL if heap failure */
const Iterator *(*itCreate)(const TheClass *tc);
```

Let's provide more explanation for each of these methods.

- `void (*clear)(const TheClass *tc, void (*freeFxn)(void *element));`
  Instead of destroying the ADT instance, this method simply purges all elements from the container, leaving the container in the same state as it was immediately after invoking the constructor. `freeFxn`, if not `NULL`, will be invoked on each element in the container, as for the `destroy()` method.

- `long (*size)(const TheClass *tc);`
  This method acts like the `len()` built-in Python method - it returns the number of elements in the container.

- `int (*isEmpty)(const TheClass *tc);`
  This method returns 1/0 if the container is/is-not empty. This method is not strictly needed, since it provides the same information as the following expression:

  ```
  tc->size(tc) == 0L
  ```

- `void **(*toArray)(const TheClass *tc, long *len);`
  This method allocates an array of `void *` pointers long enough to point to each of the elements in the container, and copies the element pointers into the array; the number of elements in the array are returned in `*len`. If there were memory allocation failures, `NULL` is returned. After the programmer has finished using the array of pointers, the programmer must return the array to the heap using `free()`, as in:

  ```
  void **array;
  long len;

  array = tc->toArray(tc, &len);
  /* use the array */
  free(array);
  ```

- `const Iterator *(*itCreate)(const TheClass *tc);`
  This method, similar to the `__iter__()` method in Python, returns an iterator ADT instance; upon receipt of an iterator instance, the programmer can traverse the container using the following pseudo-code:

  ```
  const Iterator *it;
  void *v;

  it = tc->itCreate(tc);

  while (it->hasNext(it)) {
      (void) it->next(it, &v);
      /* suitably cast v and use the value so obtained */
  }
  ```

```
    it->destroy(it);
```

The use of iterators to traverse a container *without* knowledge of its exact structure is excellent software engineering practice, and should be used in preference to the `toArray()` method. `toArray()` is maintained as a mandatory ADT method for containers for backwards compatibility.

5.3.3.3 The `Iterator` ADT

The observant reader will have noted that the `itCreate()` *factory* method returns an ADT instance of type `Iterator`. Here is the header file for that ADT.

```
#ifndef _ITERATOR_H_
#define _ITERATOR_H_

/* interface definition for generic iterator */

typedef struct iterator Iterator; /* forward reference */

/* creates an iterator from the supplied arguments; it is for use by the
 * iterator factory methods in ADTs; iterator assumes responsibility for
 * elements[] if create is successful - i.e., it_destroy will free the array
 * of pointers
 *
 * returns pointer to iterator if successful, NULL otherwise */
const Iterator *Iterator_create(long size, void **elements);

/* now define struct iterator */
struct iterator {
    /* the private data of the iterator */!
    void *self;

    /* returns 1/0 if the iterator has a next element */!
    int (*hasNext)(const Iterator *self);

    /* returns the next element from the iterator in `*element'
     *
     * returns 1 if successful, 0 if unsuccessful (no next element) */
    int (*next)(const Iterator *self, void **element);

    /* destroys the iterator */
    void (*destroy)(const Iterator *self);
};

#endif /* _ITERATOR_H_ */
```

Now let's see how we implement the `Iterator` ADT.

5.3.4 The implementation (source file)

Given the header file for an ADT, an implementation must do the following:

- determine what instance-specific information is required in `self`;
- define the `struct` that will hold this information;
- determine the parameters that need to be passed as arguments to the constructor so that it can do its job; after this determination, the header file must be modified so that the constructor prototype reflects this determination;
- provide implementations of each of the defined methods;
- implement the constructor, which must do the following:
  - allocate an instance of the dispatch table; if an allocation failure, return NULL;
  - allocate an instance of `self`; if an allocation failure, free up the dispatch table and return NULL;
  - fill in the members of `self`; if any of those fields are allocated from the heap, and if any of those allocations fail, it must free all allocations to that point, and return NULL;
  - store the address of `self` into the dispatch table;
  - store the function pointers for the methods in the dispatch table; and
  - return the dispatch table as the return value of the constructor.

5.3.4.1 `Iterator` implementation

We will use a particularly simple `Iterator` implementation that works for all collection classes. For this implementation, each instance requires three data items:

- an array of pointers to the elements of the container;
- the number of pointers in that array; and
- an array index which represents the next item to return to the user.

Thus we need the following `struct` definition in the implementation:

```
typedef struct it_data {
    long next;
    long size;
    void *elements[];
} ItData;
```

Now we need to implement each method.

`hasNext()` Let's start with the `hasNext()` method.

```
static int it_hasNext(const Iterator *it) {
    ItData *itd = (ItData *)(it->self);
    return (itd->next < itd->size) ? 1 : 0;
}
```

Let's discuss each line in turn.

- `static int it_hasNext(const Iterator *it) {`
  This function conforms to the function signature for the `hasNext()` method; we have declared it to be a static function, since we do not want programs to be able to call this function directly, only through the dispatch table.
- `ItData *itd = (ItData *)(it->self);`
  This statement declares a variable of type `ItData *`, and casts the `self` member of the dispatch table from `void *` to `ItData *`, storing it in `itd`. This is essential, as code cannot dereference a `void *` pointer, so we must cast it back to its original type (`ItData *`) in order to access the instance-specific data.
- `return (itd->next < itd->size) ? 1 : 0;`
  We use the ternary operator to determine if `next` is less than `size`; if so, return 1/True; otherwise, return 0/False.

While this is quite succinct code, by now you should understand why it works correctly, and begin to appreciate the elegance of such code.

`next()` While the `hasNext()` method merely needed to consult the values in the instance-specific structure, `next()` needs to consult and modify the values. Here is the code for this method.

```
static int it_next(const Iterator *it, void **element) {
    ItData *itd = (ItData *)(it->self);
    int status = 0;
    if (itd->next < itd->size) {
        *element = itd->elements[itd->next++];
        status = 1;
    }
    return status;
}
```

Let's discuss each line in turn.

- `static int it_next(const Iterator *it, void **element) {`
  This function conforms to the function signature for the `next()` method; we have

declared it to be a static function, since we do not want programs to be able to call this function directly, only through the dispatch table. This function passes the next data item back through `*element`, returning 1/0 as the function value if there was a next element.

Given the existence of `hasNext()`, why do we have to do it this way? There is another standard pattern for using iterators, captured by the following code:

```
const Iterator *it;
void *v;

it = tc->itCreate(tc);

while (it->next(it, &v)) {
    /* suitably cast v and use the value so obtained */
}
it->destroy(it);
```

Since it is relatively easy to support both usage patterns, it was felt that our `Iterator` ADT should do so. The author prefers the `while has, fetch and do` approach, since this maps to more language and library systems; you are encouraged to use that structure.

- `ItData *itd = (ItData *)(it->self);`
  Same as for the `hasNext()` method.

- `int status = 0;`
  Since we are going to have to return whether there was another element, `status` captures whether this is the case; we assume that it is false by setting it to 0.

- `if (itd->next < itd->size) {`
  As with `hasNext()`, the availability of another element is dependent upon this inequality.

- `*element = itd->elements[itd->next++];`
  This causes the element at `itd->next` to be copied into `*element`, incrementing `itd->next` after the assignment has been made.

- `        status = 1;`
  Set the status to true.

- `return status;`
  Return the success/failure of the call.

You may have noticed in the code on page 126 that we have indicated `(void) it->next(it, &v);`. This is the way to indicate in the code that we are explicitly ignoring the return value, since we have invoked `hasNext()` before calling `next()`. Such explicit indication is similar to our use of `UNUSED` to flag variables and function parameters that we are not using.

destroy() The final method defined for an `Iterator` ADT instance is the `destroy()` method. One normally only writes this code after one has written the constructor, since the `destroy()` method must return all heap-allocated storage created by the constructor in the reverse order to the constructor. The iterator itself is particularly simple in terms of heap storage - just the dispatch vector and the instance-specific data. A careful reader of the header file will have noticed the following admonition:
`iterator assumes responsibility for elements[] if create is successful - i.e., it_destroy will free the array of pointers`;
thus, the `destroy()` method must also return the array of `void *` pointers.

Here is the code.

```
static void it_destroy(const Iterator *it) {
    ItData *itd = (ItData *)(it->self);
    free(itd->elements);
    free(itd);
    free((void *)it);
}
```

The only thing that may be considered unusual in this code is the need to cast `it` to a `void *` in the call to `free()`. Normally this would not be required. In this case, since `it` is `const`, we need to perform the cast to change the attribute for `it` from `const` to `non-const`, thus avoiding an error message from the compiler.

Iterator_create() The constructor for an `Iterator` is as straight-forward as was the `destroy()` method, just in reverse. As before, we'll show the code before we give a line-by-line description of what is going on.

```
static Iterator template = {NULL, it_hasNext, it_next, it_destroy};

const Iterator *Iterator_create(long size, void **elements) {
    Iterator *it = (Iterator *)malloc(sizeof(Iterator));

    if (it != NULL) {
        ItData *itd = (ItData *)malloc(sizeof(ItData));
        if (itd != NULL) {
            itd->next = 0L;
            itd->size = size;
            itd->elements = elements;
            *it = template;
            it->self = itd;
        } else {
            free(it);
            it = NULL;
        }
    }
    return it;
}
```

Let's discuss each line in turn.

- `static Iterator template = {NULL, it_hasNext, it_next, it_destroy};`
  Every time the constructor is called, it has to create an `Iterator` on the heap, and then assign `self`, and function pointers for `hasNext()`, `next()`, and `destroy()` to the newly-allocated structure. By creating this template, we can perform these assignments by simply doing a copy of the template into the newly-allocated structure.

- `const Iterator *Iterator_create(long size, void **elements) {`
  This is the signature for the constructor. The `itCreate()` methods in container ADTs will call this method after creating an array of `void *` pointers of length `size`.

- `Iterator *it = (Iterator *)malloc(sizeof(Iterator));`
  This is our old friend when using `malloc()` for allocating a structure of the correct size, casting it to the correct type, and assigning it to a pointer variable of the correct type.

- `if (it != NULL) {`
  We only carry on with the function if `it` is not `NULL`. If it is `NULL`, we pick up at the bottom of the block in the scope of this if statement, which performs a `return it` - thus if `it` is `NULL`, we return `NULL` as per the specification.

- `ItData *itd = (ItData *)malloc(sizeof(ItData));`
  Now allocate the instance-specific data structure from the heap.

-
  ```
  if (itd != NULL) {
      itd->next = 0L;
      itd->size = size;
      itd->elements = elements;
      *it = template;
      it->self = itd;
  ```
  With a succefful allocation of `itd`, fill in the `next`, `size`, and `elements` members of `itd`; copy `template` into the dispatch table structure on the heap; finally, replace the `self` member of the dispatch table with a pointer to the instance-specific structure.

-
  ```
  } else {
      free(it);
      it = NULL;
  ```
  The allocation for `itd` failed, so we need to return the storage for `it` to the heap, and set `it` to `NULL`.

- `return it;`
  Return the `Iterator` to the caller, or `NULL` if there were any memory allocation errors.

5.4 An complete example - a String ADT

Let's design and implement a mutable String ADT. As we learned in chapter 3, character strings in C are simply arrays of type `char`, with one character per array element. The end of the string is encoded as an element with the null value, `'\0'`. In order to manipulate these strings, one must use the functions defined in `<string.h>`.

Our mutable String ADT provides much of the functionality found in `<string.h>` and `<ctype.h>`. A mutable string is a sequence that can be indexed from 0 to N-1, where N is the length of the String. These are the methods that we would like to have on a String:

- obviously, our constructor should convert a C string into a String;
- return a new String which is a copy of an existing String;
- return a new String that is a slice of an existing String;
- destroy a String;
- append a C string to a String;
- assign a new character to a particular index in a String;
- insert a C string into a String before a particular index;
- convert all uppercase letters in the String to lowercase;
- remove all leading whitespace in the String;
- remove a character at a particular index;
- replace all occurrences of one C string by another C string in the String;
- remove all trailing whitespace in the String;
- strip all leading and trailing whitespace from a String; and
- convert all lowercase letters in the String to uppercase.
- compare two Strings, returning <0 | 0 | >0 if first < second | first == second | first > second, respectively;
- return true/false if a String ends with a particular C string;
- return index into String where a particular C string first matches, -1 if does not match;
- returns true if the string has at least one character and all characters are alphanumeric, false otherwise;
- returns true if the String has at least one character and all characters are digits, false otherwise;
- returns true if the String has at least one character and all characters are lowercase, false otherwise;
- returns true if the String has at least one character and all characters are whitespace, false otherwise;
- returns true if the String has at least one character and all characters are uppercase, false otherwise;
- return the length of the String;
- return index into String where a particular C string last matches, -1 if does not match;
- return true/false if a String starts with a particular C string;
- convert a String into a C string;

5.4.1 The header file

We already have a header file named `string.h`, which defines the `str*()` functions for manipulating C strings. On Linux, since the filenames are case-sensitive, we could name our header file `String.h`, and not have any conflict. Some operating systems, such as Windows, have case-insensitive filenames. There is nothing in our ADT implementations that will not work on Windows, or any other system that supports the 1989 version (or later) of the C standard and has case-insensitive filenames. To maximize the utility of this ADT, we will name the header file `stringADT.h`, and the implementation `stringADT.c`.

Here is the header file with most of the commentary and all vertical whitespace squeezed out.

```
#ifndef _STRINGADT_H_
#define _STRINGADT_H_
typedef struct string String;
const String *String_create(char *str);
struct string {
    void *self;
/* manager methods (additional constructors and destructor) */
    const String *(*copy)(const String *str);
    const String *(*slice)(const String *str, int begin, int end);
    void (*destroy)(const String *str);
/* mutator methods (change the String) */
    int (*append)(const String *str, char *suffix);
    int (*assign)(const String *str, int chr, int index);
    int (*insert)(const String *str, char *substr, int index);
    void (*lower)(const String *str);
    void (*lStrip)(const String *str);
    int (*remove)(const String *str, int index);
    int (*replace)(const String *str, char *old, char *new);
    void (*rStrip)(const String *str);
    void (*strip)(const String *str);
    void (*upper)(const String *str);
/* accessor methods (tell you about the String) */
    int (*compare)(const String *str, const String *other);
    int (*endsWith)(const String *str, char *suffix, int begin, int end);
    int (*index)(const String *str, char *substr, int begin, int end);
    int (*isAlpha)(const String *str);
    int (*isDigit)(const String *str);
    int (*isLower)(const String *str);
    int (*isSpace)(const String *str);
    int (*isUpper)(const String *str);
    int (*len)(const String *str);
    int (*rindex)(const String *str, char *substr, int begin, int end);
    int (*startsWith)(const String *str, char *prefix, int begin, int end);
/* miscellaneous methods */
    char *(*convert)(const String *str);
};
#endif /* _STRINGADT_H_ */
```

Besides the constructor, `String_create()`, which converts a C string into a `String`, we have two methods that create new `String`'s: `copy()` performs a deep copy of the `String` upon which it is invoked, and `slice()`, which performs a deep copy of the subrange of the `String` upon which it is invoked, as specified by [`begin`, `end`). `destroy()` destroys instances of `String`'s.

`append()`, `assign()`, `insert()`, `lower()`, `lStrip()`, `remove()`, `replace()`, `rStrip()`, `strip()`, and `upper()` modify the contents of the string, either by adding characters, deleting characters, or replacing characters.

`compare()`, `endsWith()`, `index()`, `isAlpha()`, `isDigit()`, `isLower()`, `isSpace()`, `isUpper()`, `len()`, `rindex()`, and `startsWith()` tell you things about the `String` itself, as well as in relation to C strings and other `String`'s.

Finally, `convert()` provides a C string that is the equivalent set of characters as currently make up the `String`.

5.4.2 The source file

We will show the implementation in pieces to facilitate understanding. First, let's discuss how to represent a `String`.

Since we are likely to be using functions in `<string.h>` and `<ctype.h>` to assist us in the implementation, we probably want to store the characters that make up a string as an array of `char`, and at all times this array is terminated by `'\0'`. This choice makes the implementation of the `convert()` method particularly easy, as we simply return the address of this array to the caller.

Since so many of the methods modify the `String`, either adding or removing characters, we will need to keep track of how large the character array is (how big a chunk of memory we allocated); we will also keep track of the current length of the string in that array. Of course, we could just invoke `strlen()` on that buffer whenever we needed to know the length, but it is much more efficient to keep track of the length, and avoid the runtime cost to calculate the length.

Most of the method implementations are very straightforward.

5.4.2.1 The preliminaries

```
/* implementation for String ADT */
#include "stringADT.h"
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

/* member data for String instance */
typedef struct st_data {
    int length;
    int size;
    char *buf;
} StData;

#define INCREMENT 1024 /* if we ever have to grow the string, do it
                           by this increment except for replace() */
```

As always with an ADT, the implementation must include the corresponding interface definition, thus assuring that we have the correct function signatures in the implementation. We include `<string.h>`, `<ctype.h>`, and `<stdlib.h>`, as we will need functions and definitions from those header files in the implementation. Finally, we define an `INCREMENT` size; whenever we need to add things to the character array, if we need more capacity, then we `realloc()` the buffer to be `size + INCREMENT`, except for resizing due to `replace()`.

5.4.2.2 The manager methods

`slice_copy()` Both `copy()` and `slice()` require that we duplicate part or all of the character array on the heap. This helper function is used by both of those implementations to perform that duplication.

```
/*
 * helper function that creates a copy of buf[begin]..buf[end], inclusive,
 * on the heap
 *
 * assumes that begin and end are legal indices into buf
 *
 * returns pointer to the copy, or NULL if malloc() failure
 */
static char *slice_copy(char *buf, int begin, int end) {
    int size = end - begin + 1 + 1;    /* +1 for '\0' */
    char *p;
```

```
        if ((p = (char *)malloc(size)) != NULL) {
            int i, j;

            for (i = begin, j = 0; i <= end; i++, j++)
                p[j] = buf[i];
            p[j] = '\0';
        }
        return p;
    }
```

`copy()` Most of the complexity in `copy()` is to recover from heap allocation failure. Note that we copy the dispatch table for `str` into the dispatch table for the copy; this initializes all of the function pointers, and all we have to do is cause the `self` member to point to the copy of the string.

```
    static const String *st_copy(const String *str) {
        StData *std = (StData *)str->self;
        String *nstr = (String *)malloc(sizeof(String));

        if (nstr != NULL) {
            StData *nstd = (StData *)malloc(sizeof(StData));

            if (nstd != NULL) {
                nstd->buf = slice_copy(std->buf, 0, std->length - 1);
                if (nstd->buf != NULL) {
                    nstd->length = std->length;
                    nstd->size = std->size;
                    *nstr = *str;
                    nstr->self = nstd;
                } else {
                    free(nstd);
                    free(nstr);
                    nstr = NULL;
                }
            } else {
                free(nstr);
                nstr = NULL;
            }
        }
        return nstr;
    }
```

`slice()` This code is very similar to that for `copy()`. Again, most of the complexity is handling heap allocation failures. Note that we copy the dispatch table for `str` into the dispatch table for the copy; this initializes all of the function pointers, and all we have to do is cause the `self` member to point to the copy of the string.

```
static const String *st_slice(const String *str, int begin, int end) {
    StData *std = (StData *)str->self;
    String *nstr = NULL;

    if (end == 0)
        end = std->length;
    if (begin >= 0 && end <= std->length && begin < end) {
        nstr = (String *)malloc(sizeof(String));
        if (nstr != NULL) {
            StData *nstd = (StData *)malloc(sizeof(StData));

            if (nstd != NULL) {
                nstd->buf = slice_copy(std->buf, begin, end - 1);
                if (nstd->buf != NULL) {
                    nstd->length = strlen(nstd->buf);
                    nstd->size = nstd->length + 1;
                    *nstr = *str;
                    nstr->self = nstd;
                } else {
                    free(nstd);
                    free(nstr);
                    nstr = NULL;
                }
            } else {
                free(nstr);
                nstr = NULL;
            }
        }
    }
    return nstr;
}
```

`destroy()` This method is quite straightforward: we need to return the buffer to the heap, then the `StData` instance, and then the dispatch table.

```
static void st_destroy(const String *str) {
    StData *std = (StData *)str->self;

    free(std->buf);
    free(std);
```

```
        free((void *)str);
    }
```

5.4.2.3 The mutator methods

`append()` This method appends the contents of the C string `suffix` to our `String` instance. If addition of the contents would overflow the character array, the array is extended using `realloc()` from `<string.h>`.

```
    static int st_append(const String *str, char *suffix) {
        StData *std = (StData *)str->self;
        int n = strlen(suffix);

        if ((std->length + n + 1) > std->size) {
            int nsize = std->size + ((n > INCREMENT) ? n : INCREMENT);
            char *p = (char *)realloc(std->buf, nsize);

            if (p == NULL)
                return 0;
            std->buf = p;
            std->size = nsize;
        }
        strcpy(std->buf+std->length, suffix);
        std->length = strlen(std->buf);
        return 1;
    }
```

`assign()` This method assigns a new character `chr` to the `String` at `index`; the previous character at that location is overwritten; the return value is `1` if `index` was legal (between 0 and length-1, inclusive), 0 otherwise. If you want to append a character at `index` length, you have to use `append()` above.

```
    static int st_assign(const String *str, int chr, int index) {
        StData *std = (StData *)str->self;
        int result = 0;

        if (index >= 0 && index < std->length) {
            std->buf[index] = chr;
            result = 1;
        }
        return result;
    }
```

`insert()` This method inserts the C string `substr` into the `String` before `index`; the return value is `1` if `index` was legal (between 0 and length, inclusive), 0 otherwise. If the insertion would overflow the character buffer, the buffer is extended using `realloc()`.

```
static int st_insert(const String *str, char *substr, int index) {
    StData *std = (StData *)str->self;
    int n = strlen(substr);
    int i, j;

    if (index < 0 || index > std->length)
        return 0;
    if ((std->length + n + 1) > std->size) {
        int nsize = std->size + ((n > INCREMENT) ? n : INCREMENT);
        char *p = (char *)realloc(std->buf, nsize);

        if (p == NULL)
            return 0;
        std->buf = p;
        std->size = nsize;
    }
    for (i = std->length, j = std->length + n; i >= index; i--, j--)
        std->buf[j] = std->buf[i];
    for (i = 0, j = index; i < n; i++, j++)
        std->buf[j] = substr[i];
    std->length += n;
    return 1;
}
```

`lower()` This method converts all uppercase characters in the `String` to lowercase.

```
static void st_lower(const String *str) {
    StData *std = (StData *)str->self;
    int i;

    for (i = 0; i < std->length; i++)
        std->buf[i] = tolower(std->buf[i]);
}
```

`lStrip()` This method removes all leading whitespace from the `String`.

```
static void st_lStrip(const String *str) {
    StData *std = (StData *)str->self;
    int i, j;
```

```
        for (i = 0; i < std->length; i++)
            if (! isspace(std->buf[i]))
                break;
        for (j = 0; i < std->length; i++, j++)
            std->buf[j] = std->buf[i];
        std->buf[j] = '\0';
        std->length = strlen(std->buf);
    }
```

`remove()` This method removes the character at `index` from the `String`. If `index` is legal, the return value is 1; otherwise, it is 0.

```
    static int st_remove(const String *str, int index) {
        StData *std = (StData *)str->self;
        int result = 0, i, j;

        if (index >= 0 && index < std->length) {
            for (j = index, i = index + 1; i < std->length; i++, j++)
                std->buf[j] = std->buf[i];
            std->buf[j] = '\0';
            std->length--;
            result = 1;
        }
        return result;
    }
```

`replace()` This method replaces all occurrences of `old` in the `String` by `new`. If this would cause the buffer to overflow, it is extended. The extension logic is to calculate the worst case number of times `old` can appear in the string (e.g., if `old` is "my", and the `String` is `10` characters long, then the worst case is if the string is "mymymymymy" - length of `String` divided by length of `old`, rounded up to the nearest whole number; this is `nold` in the code). Then, the length of the `String` after the replacement, in the worst case, is current length plus `nold` times the difference in lengths of the `old` and `new` strings.

```
    static int st_replace(const String *str, char *old, char *new) {
        StData *std = (StData *)str->self;
        int oldlen = strlen(old);
        int newlen = strlen(new);
        int nold, size;
        char *nbuf;
        int result = 0; /* assume malloc() fails */
        nold = ((std->length) / oldlen) + 1; /* worst case # of matches */
        size = std->size - nold * oldlen + nold * newlen;
        if (size < std->size)
            size = std->size;
        nbuf = (char *)malloc(size);
        if (nbuf != NULL) {
            int i, j;
            i = 0;
            j = 0;
            while (i < std->length) {
                if (strncmp(std->buf+i, old, oldlen) == 0) {
                    strncpy(nbuf+j, new, newlen);
                    j += newlen;
                    i += oldlen;
                } else {
                    nbuf[j++] = std->buf[i++];
                }
            }
            nbuf[j] = '\0';
            free(std->buf);
            std->buf = nbuf;
            std->length = j;
            std->size = size;
            result = 1;
        }
        return result;
    }
```

rStrip() This method removes all trailing whitespace from the `String`.

```
    static void st_rStrip(const String *str) {
        StData *std = (StData *)str->self;
        int i;

        for (i = std->length - 1; i >= 0; i--)
            if (! isspace(std->buf[i]))
                break;
        std->buf[i+1] = '\0';
        std->length = strlen(std->buf);
    }
```

strip() This method removes all leading and trailing whitespace from the `String`.

```
static void st_strip(const String *str) {

    st_lStrip(str);
    st_rStrip(str);
}
```

`upper()` This method converts all lowercase characters in the `String` to uppercase.

```
static void st_upper(const String *str) {
    StData *std = (StData *)str->self;
    int i;

    for (i = 0; i < std->length; i++)
        std->buf[i] = toupper(std->buf[i]);
}
```

5.4.2.4 The accessor methods

`compare()` This method compares two `String`'s; it returns a value < 0 if `str < other`, returns 0 if `str == other`, and returns a value > 0 if `str > other`.

```
static int st_compare(const String *str, const String *other) {
    StData *std = (StData *)str->self;
    StData *otd = (StData *)other->self;

    return strcmp(std->buf, otd->buf);
}
```

`endsWith()` This method returns true if the slice `str[begin:end]` "ends with" the characters in `suffix`; if `end == 0`, it means that the ending index is the length of `str`. To simply check if `str` "ends with" `suffix`, specify both `begin` and `end` as 0.

```
static int st_endsWith(const String *str, char *suffix, int begin, int end) {
    StData *std = (StData *)str->self;
    int nchars, suflen, result = 0;

    if (end == 0)
        end = std->length;
    nchars = end - begin;
    suflen = strlen(suffix);
    if (nchars >= suflen) {
        if (strncmp(suffix, std->buf + end - suflen, suflen) == 0)
            result = 1;
    }
    return result;
}
```

`index()` This method returns the index in `str[begin:end]` at which the first match to `substr` occurs. If there is no match, it returns -1.

```
static int st_index(const String *str, char *substr, int begin, int end) {
    StData *std = (StData *)str->self;
    int n = strlen(substr);
    int i, j;

    if (end == 0)
        end = std->length;
    for (i = begin, j = end - begin; i < end && j >= n; i++, j--)
        if (strncmp(std->buf + i, substr, n) == 0)
            return i;
    return -1;
}
```

`isAlpha()`, `isDigit()`, `isLower()`, `isSpace()`, and `isUpper()` These methods require that `str` have at least one character; the methods returns true if all of the characters in `str` are alphanumeric, digits, lowercase, whitespace, or uppercase, respectively.

```
static int st_isAlpha(const String *str) {
    StData *std = (StData *)str->self;
    int i;

    if (std->length > 0) {
        for (i = 0; i < std->length; i++)
            if (! isalpha(std->buf[i]))
                return 0;
        return 1;
    }
    return 0;
}

static int st_isDigit(const String *str) {
    StData *std = (StData *)str->self;
    int i;
    if (std->length > 0) {
        for (i = 0; i < std->length; i++)
            if (! isdigit(std->buf[i]))
                return 0;
        return 1;
    }
    return 0;
}

static int st_isLower(const String *str) {
    StData *std = (StData *)str->self;
    int i;

    if (std->length > 0) {
        for (i = 0; i < std->length; i++)
            if (! islower(std->buf[i]))
                return 0;
        return 1;
    }
    return 0;
}

static int st_isSpace(const String *str) {
    StData *std = (StData *)str->self;
    int i;

    if (std->length > 0) {
        for (i = 0; i < std->length; i++)
            if (! isspace(std->buf[i]))
                return 0;
        return 1;
    }
    return 0;
}

static int st_isUpper(const String *str) {
    StData *std = (StData *)str->self;
    int i;

    if (std->length > 0) {
        for (i = 0; i < std->length; i++)
            if (! isupper(std->buf[i]))
                return 0;
        return 1;
    }
    return 0;
```

```
}
```

`len()` This method returns the number of characters in `str`.

```
static int st_len(const String *str) {
    StData *std = (StData *)str->self;

    return std->length;
}
```

`rindex()` This method returns the index in `str[begin:end]` at which the last match to `substr` occurs. If there is no match, it returns -1.

```
static int st_rindex(const String *str, char *substr, int begin, int end) {
    StData *std = (StData *)str->self;
    int n = strlen(substr);
    int i, j;

    if (end == 0)
        end = std->length;
    for (i = end - n, j = end - begin; i >= begin && j >= n; i--, j--)
        if (strncmp(std->buf + i, substr, n) == 0)
            return i;
    return -1;
}
```

`startsWith()` This method returns true if the slice `str[begin:end]` "starts with" the characters in `prefix`; if `end == 0`, it means that the ending index is the length of `str`. To simply check if `str` "starts with" `prefix`, specify both `begin` and `end` as 0.

```
static int st_startsWith(const String *str, char *prefix, int begin, int end) {
    StData *std = (StData *)str->self;
    int nchars, prelen, result = 0;

    if (end == 0)
        end = std->length;
    nchars = end - begin;
    prelen = strlen(prefix);
    if (nchars >= prelen) {
        if (strncmp(prefix, std->buf + begin, prelen) == 0)
            result = 1;
    }
    return result;
}
```

5.4.2.5 The miscellaneous methods

`convert()` This method returns a C string representation of `str`.

```
static char *st_convert(const String *str) {
    StData *std = (StData *)str->self;

    return std->buf;
}
```

5.4.2.6 The constructor

The template `String` structure As with the `Iterator` implementation, we create a template for the dispatch table. It is important to note that the function pointers in the initialization of the template *must* be in exactly the same order as the definition for `struct string` in the header file. If they are not in the same order, in the best case the compiler will complain because the function pointer you have specified in a particular place does not match the appropriate method's function prototype; in the worst case, it does match the prototype, but delivers the wrong functionality.

```
static String template = {
    NULL, st_copy, st_slice, st_destroy, st_append, st_assign, st_insert,
    st_lower, st_lStrip, st_remove, st_replace, st_rStrip, st_strip, st_upper,
    st_compare, st_endsWith, st_index, st_isAlpha, st_isDigit, st_isLower,
    st_isSpace, st_isUpper, st_len, st_rindex, st_startsWith, st_convert
};
```

`String_create()` As with the `copy()` and `slice()` methods, most of the complexity is in handling possible heap allocation failures. You should find the code straightforward to understand.

```
const String *String_create(char *str) {
    String *st = (String *)malloc(sizeof(String));

    if (st != NULL) {
        StData *std = (StData *)malloc(sizeof(StData));
        if (std != NULL) {
            std->length = strlen(str); /* length of str */
            std->size = std->length + 1; /* account for '\0' */
            std->buf = (char *)malloc(std->size);
            if (std->buf != NULL) {
                strcpy(std->buf, str);
                *st = template;
                st->self = std;
            } else {
                free(std);
                free(st);
                st = NULL;
            }
        } else {
            free(st);
            st = NULL;
        }
    }
    return st;
}
```

Data Structures

This section starts off with a discussion of complexity metrics for data structures, in general, and big O notation, in particular. It also describes how one can empirically measure the runtime performance of an implementation.

It then proceeds to describe the interfaces, and make the interfaces concrete using the previously described implementation approach, for ADTs of the following types:

- array lists;
- queues;
- stacks;
- linked lists;
- hash tables;
- priority queues; and
- searching and sorting.

Chapter 6

Complexity metrics and runtime analysis

A *data structure* is a systematic way to organize and access data; an *algorithm* is a step-by-step procedure for performing a task. As a computer scientist, we are interested in "good" data structures and algorithms. How do we determine if a particular data structure or algorithm is good?

The generally accepted measure of "goodness" is the running time of algorithms and data structure methods; another measure that is sometimes important is the amount of memory consumed. Running time is a natural measure of "goodness", since we would like our programs to run as quickly as possible while delivering the correct results.

It is generally the case that an algorithm's or a data structure method's running time increases with the size of its input. The running time is also dependent upon the computer hardware and its software environment. If we keep the hardware and its software environment fixed, we can compare the running time of different algorithms and data structure methods to determine the relationship between the running time and the size of its input.

6.1 How do we measure running time?

If an algorithm or a data structure method has been implemented, we can empirically measure the running time by executing it on various test inputs, reporting the elapsed time to process each test input. If the only difference between the test inputs is the number of values, these measurements give us an idea of running time as a functions of the number of inputs, $\tau(n)$.

6.1.1 Measurement at the shell level

Linux supplies a program, `/usr/bin/time`, which will execute a program and report various measurements concerning the resources consumed by the program. Let's assume that we have a file named `verylargefile`, and that it has `12,480,100` lines, each line has a single word, and the total number of characters is `111,148,500`. Let's use `wc` on the file to count the lines, words, and characters, and use `/usr/bin/time` to determine the resource utilization of `wc`.

```
$ /usr/bin/time wc verylargefile
 12480100  12480100 111148500 verylargefile
1.59user 0.03system 0:01.63elapsed 99%CPU (0avgtext+0avgdata 429468
maxresident)k 0inputs+0output (1760major+0minor)pagefaults 0 swaps
```

As you can see, `/usr/bin/time` not only tells us the amount of time spent in the user program, time spent in the kernel while the program was running, and total elapsed time, it also gives us other information about resources used by the program. If we just want to see the timing information, we can provide `/usr/bin/time` with a format string indicating what information we want in what format:

```
$ fmt='real    %E\nuser    %Us\nsys     %Ss'
$ /usr/bin/time -f "$fmt" wc verylargefile
 12480100  12480100 111148500 verylargefile
real    0:01.58
user    1.54s
sys     0.01s
```

This is certainly more tractable; fortunately, `/usr/bin/time` looks for the `TIME` environment variable; if found, it uses that as its format string.

```
$ export TIME='real    %E\nuser    %Us\nsys     %Ss'
$ /usr/bin/time wc verylargefile
 12480100  12480100 111148500 verylargefile
real    0:01.58
user    1.54s
sys     0.01s
```

Why do we keep specifying the full pathname to `/usr/bin/time`? `bash` has a built-in `time` command, which looks like this:

```
$ time wc verylargefile
 12480100  12480100 111148500 verylargefile

real    0m1.618s
user    0m1.531s
sys     0m0.062s
```

```
$
```

While this looks like exactly what we want, we will see that shell built-ins do not respect some of the I/O redirection commands that we wish to use. When the shell is resolving a command, if it finds a built-in, it will *not* search for the command along the search path. This is unfortunate when the program we want to run has an identical name to a built-in. Fortunately, we can indicate to the shell that it should *not* look for a built-in simply by prefixing a backslash to the name of the command, as in

```
$ \time wc verylargefile
 12480100  12480100 111148500 verylargefile
real    0:01.58
user    1.53s
sys     0.03s
$
```

Now let's get back to I/O redirection. `/usr/bin/time` writes the timing and other resource usage information on standard error. How might we cause that information to be stored in a file?

```
$ \time wc verylargefile
 12480100  12480100 111148500 verylargefile
real    0:01.58
user    1.53s
sys     0.03s
$ \time wc verylargefile >wc.out
real    0:01.61
user    1.57s
sys     0.01s
$ \time wc verylargefile >wc.out 2>time.out
$ cat time.out
real    0:01.58
user    1.53s
sys     0.03s
```

As we saw in chapter 2, the construction `2>`*filename* (no spaces are allowed between the `2` and the `>`) directs the standard error output into the file.

Note that standard output or standard error output from shell built-in's, like `time`, do not abide by the output redirection commands. For example,

```
$ time wc verylargefile >wc.out 2>&1

real    0m1.618s
user    0m1.531s
sys     0m0.062s
$
```

We see that despite indicating that standard error should be merged with standard output on `wc.out`, the built-in `time` still wrote the timing information on the shell's standard error, the terminal. This is why we prefer to use `/usr/bin/time`, since it is a separate process and abides by any output redirection that has been stated.

If we are going to use `/usr/bin/time` to perform comparative measurements of an algorithm with different input sizes, then reporting the user and system times would appear to be superfluous. Thus, specifying `TIME="%e"` would appear to be most appropriate, as it will cause `/usr/bin/time` to report the elapsed time in seconds on standard error output. Thus, if you had a number of files with different numbers of lines, each line consisting of a single word and a newline, one could capture the performance of `wc` as a function of number of lines using the following `for` loop in `bash`:

```
$ export TIME="%e"
$ ls file*
f10 f100 f1000 f10000 f100000, f1000000 f10000000
$ for n in 10 100 1000 10000 100000 1000000 10000000; do
>     echo -n "$n," 1>&2
>     \time wc "f$n" >/dev/null
> done 2>time.out
$ cat time.out
10,0.01
100,0.01
1000,0.01
10000,0.01
100000,0.02
1000000,0.13
10000000,1.27
```

There are several things about this `for` loop that may seem new to you.

- The invocation of `ls` is simply to show you the filenames with different numbers of lines.
- We are used to seeing the `"$ "` prompt from `bash`; when `bash` is reading your command *and* it is obvious that the command is continuing on the next line of input, the secondary prompt, `"> "` is issued on the line. These two prompts are actually defined as environment variables, `PS1` and `PS2`.
- We have used the `-n` flag to `echo`, which tells it to *not* output a newline after outputting its arguments; we have also used `1>&2` to force `echo`'s output onto standard error.
- While it is not new, we used variable expansion within a quoted string (`"f$n"`) to generate the appropriate filename; note that variable expansion *does not* work if you use apostrophes for your quoted string.
- We redirected standard output for `time`, and for `wc` since it is a child process, to `/dev/null`. `/dev/null` is a pseudo-device in Linux that simply acts as a sink for information written to it. By doing this redirection, we prevent huge amounts of

data being written to the terminal, which could skew the elapsed runtime for `wc`.
- And, finally, we redirected standard error output for the entire `for` loop.

Note that the data is written in the form of comma-separated values, which can be processed by a large number of statistical and graphical packages.

What have we determined from this experiment?

- `/usr/bin/time` enables us to measure the performance of a program.
- Through use of the `TIME` environment variable we can tailor the output from `time`.
- Exploitation of features in `bash` and `echo` enable us to output the performance data in any way we would like.
- 1 second granularity (well, yes, it does give us hundredths of a second) does not give us much insight into the performance if we do not have enough data to keep the processor busy for a significant amount of time (cf. the performance numbers for 10, 100, 1000, and 10000 lines are all 0.01 seconds).
- The elapsed time reported by `time` also includes the time to start up the program, and to tear down the program; thus the number reported is actually something like $overhead + f(n)$.

6.1.2 Measurement via source code instrumentation

If your performance analysis needs require elimination of the process overhead, and better precision than hundredths of a second, then you will need to insert instrumentation in your source code. Doing so is straightforward.

Linux provides a system call, `gettimeofday()`, which returns the current time of day as a number of seconds and microseconds since a fixed time in the past. The following code shows how one would obtain the current time of day using this function.

```
#include <sys/time.h>

struct timeval tm;

(void)gettimeofday(&tm, NULL);
```

Armed with this function, we can now provide the instrumentation needed to time a function. Suppose the function to be timed is `f()`. Then the following code segment will display the elapsed time in milliseconds.

```
#include <sys/time.h>
#include <stdio.h>

struct timeval t1, t2;
long long musecs;
```

```
(void)gettimeofday(&t1, NULL);
f(); /* invoke the function to be timed */
(void)gettimeofday(&t2, NULL);
musecs = 1000000 * (t2.tv_sec - t1.tv_sec) + (t2.tv_usec - t1.tv_usec);
fprintf(stderr, "%Ld.%03d\n", musecs / 1000, (int)(musecs % 1000));
```

As with our use of the `for` loop earlier, we are exploiting some previously unexercised features.

- We declare a `long long` integer for the number of microseconds; since there are `1,000,000` microseconds in a second, and a 32-bit, unsigned integer can only count up to 4 billion, we would overflow `musecs` after `~4000` seconds; this is not very much time, so we are better off guaranteeing that we have 64 bits of precision.
- You might think you need some conditional logic to handle the computation of `musecs` if `t2.tv_usec < t1.tv_usec`. This is not the case, as the first part of the computation has over counted based upon the difference of the `tv_sec` members, and the negative number that results from the difference of the `tv_usec` members merely compensates for that.
- The format string to `fprintf()` is probably more complex than anything you have written to date. Let's deconstruct it to be sure that you understand what it implies.
  - `%Ld` - indicates that the argument to be formatted is a `long long` integer; that is what we get by dividing `musecs` by `1000`.
  - `%03d` - indicates that the argument to be formatted is an `int`, and that it should be formatted in an output field 3 characters wide, and use leading `0`'s if needed to pad out to 3 characters wide; this gives us a decimal expression for the number of milliseconds.
  - Note the cast to type `int` of the modulus operation on `musecs`; we know that the value will be in the range `[0 .. 1000)`, so this fits in an `int`, and guarantees that it fits in a 3 character wide field.

6.2 Is measuring running time sufficient?

While useful, there are limitations to using experimentally measured running times to characterize algorithms:

- Just as with using test cases to test the correctness of your code, the performance characterization is only as good as the completeness of the set of test inputs. The most important test input (i.e., one that causes worst case behavior) may have been omitted.
- Since the measured runtimes depend upon the hardware and software environment of the system upon which the measurements are performed, we will be unable to

compare performance measured on one system with that measured on a different system.
- Finally, a complete implementation of the algorithm or data structure method is required to measure the performance experimentally.

The computer science community has devised a standard approach to analyzing the performance of an algorithm or data structure method that does not depend upon performing experiments. This approach associates with each algorithm a function, $f(n)$, that characterizes the running time of the algorithm as a function of the input size, n. For example, the running time for one algorithm may take twice as long when it processes twice as many items; for such an algorithm, we say "the algorithm runs in time proportional to n", or is "linear in n". The running time for another algorithm may take 4 times as long when the number of items doubles; in this case, we say "the algorithm runs in time proportional to n^2", or is "quadratic in n".

What does it mean to say that "the algorithm runs in time proportional to n"? If we were to experimentally measure the performance of an implementation of the algorithm on *any* input of size n, the measured performance would never exceed kn, where k is a constant that depends upon the hardware and software environment upon which the experiments were conducted.

More importantly, if we have two implementations of an algorithm, one of which is linear in n, and the other is quadratic in n, you would prefer the first to the second, since the linear algorithm will perform much better as the number of items grows larger.

6.3 The most prevalent performance functions, $f(n)$

There are a large number of functions, $f(n)$, that have been found to characterize algorithms. In this section, we discuss only the most prevalent functions that we use to characterize the algorithms and data structures in the remainder of the text book.

6.3.1 Constant in n

There are actually some algorithms and data structure methods where the running time does not depend upon the number of items, n, at all. In this case, we write the function as

$$f(n) = k,$$

where k is a fixed constant. Let's look at a simple example.

```
struct array {
    long size;
    int *buffer;
};
```

```
long arraySize(struct array *a) {
    return a->size;
}

int main(int argc, char *argv[]) {
    int buf[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    struct array myArray = { 10, buf};
    long theSize = arraySize(&myArray);
    return 0;
}
```

Everytime this function is called, it simply returns the `size` member of the argument. Even though a `struct array` could contain arrays of varying sizes, the time required to compute the return for `arraySize()` is constant.

6.3.2 Linear in n

We have already introduced this type of function,

$$f(n) = n.$$

We will encounter this type of function whenever an algorithm or method has to perform a single operation on each of n items. Let's look at a simple example.

```
double sumOfSquares(double theArray[], long size) {
    double answer = 0.0;
    int i;

    for (i = 0; i < size; i++)
        answer += theArray[i] * theArray[i];
    return answer;
}
```

For each element of the array, this function generates the square of that element, and adds this value to `answer`. Thus, we will perform the loop `size` times; the running time will be some constant cost for function entry and return plus another constant times the number of elements in the array - i.e.,

$$k_{overhead} + k_{product+addition} n,$$

which shows that the running time is linear in the number of items.

6.3.3 Quadratic in n

We've previously touched on this type of function, as well,

$$f(n) = n^2.$$

We often encounter this type of function when an algorithm or method has nested loops. Consider the following example.

```
#include <stdio.h>

void bubble(int theArray[], long size) {
    long i, j;

    for (i = 0; i < size - 1; i++)
        for (j = 0; j < size - i - 1; j++)
            if (theArray[j] > theArray[j+1]) {
                int t = theArray[j];
                theArray[j] = theArray[j+1];
                theArray[j+1] = t;
            }
}

void print(int theArray[], long size) {
    long i;

    for (i = 0; i < size; i++)
        printf("%s%d", ((i == 0) ? "" : ","), theArray[i]);
    printf("\n");
}

int main(int argc, char *argv[]) {
    int theArray[] = {0, 9, 1, 8, 2, 7, 3, 6, 4, 5};

    print(theArray, 10);
    bubble(theArray, 10);
    print(theArray, 10);
}
```

If we compile, link, and execute this program, the following output results:

```
0,9,1,8,2,7,3,6,4,5
0,1,2,3,4,5,6,7,8,9
```

In other words, `bubble()` sorts the array of integers in place. We will discuss sorting algorithms in a later chapter, but let's look at the complexity for `bubble()`:

- The outer loop is executed `size-1` times.
- For each value of `i` in the outer loop, we execute the inner loop `size-i-1` times.

- For each value of `j` in the inner loop, we compare two adjacent values in the array, and if the `j`th value is greater than the `j+1`st value, we swap the values. In this way, the largest item "bubbles" to the end of the array on each iteration of the outer loop.

The running time for the outer loop is $O(n)$, since it depends upon `size`. The running time for the inner loop is also $O(n)$, since it too depends upon `size`. Thus, the running time for `bubble()` is $O(n^2)$.

6.3.4 Logarithmic in n

There are a number of algorithms for which the performance varies as the logarithm of n:

$$f(n) = \log_b n,$$

where $b > 1$ is a constant, and is called the base. $\log_b n$ grows much more slowly than n. In computer science, b is usually 2.

Suppose that you are storing a number of integers, with no duplicates, in an array. One of the functions that you would typically need is to determine if a particular integer of interest is in the array. The signature for this function is

```
int contains(int theArray[], long size, int value);
```

An implementation of `contains()` that makes no assumptions about how the elements of the array are ordered would look like this.

```
int contains(int theArray[], long size, int value) {
    long i;

    for (i = 0; i < size; i++)
        if (theArray[i] == value)
            return 1;
    return 0;
}
```

Since no assumptions are made regarding the ordering of elements in the array, we have to do a linear search to see if the array contains `value`. By now, it should be clear that this implementation has complexity $O(n)$.

If we guaranteed that the array was sorted at all times, can we do better? You were most likely introduced to the notion of binary search in your previous computer science courses. The following is an implementation of `contains()` that assumes that the array is sorted.

```
int contains(int theArray[], long size, int value) {
    long lBound = 0, uBound = size - 1, midPoint;

    while (lBound <= uBound) {
        midPoint = lBound + (uBound - lBound) / 2
        if (theArray[midPoint] == value)
            return 1;
        if (theArray[midPoint] < value)
            lBound = midPoint + 1;
        else
            uBound = midPoint - 1;
    }
    return 0;
}
```

Binary search works on the principle of divide and conquer; at each iteration through the array, the number of items under consideration is $\frac{1}{2}$ the number on the previous iteration. If `value` is not contained in the array, then we will perform $\log n$ iterations before returning 0.

6.3.5 Loglinear in n

There are a number of algorithms for which a straightforward implementation is quadratic in n, but for which one can provide an implementation exhibiting

$$f(n) = n \log_b n$$

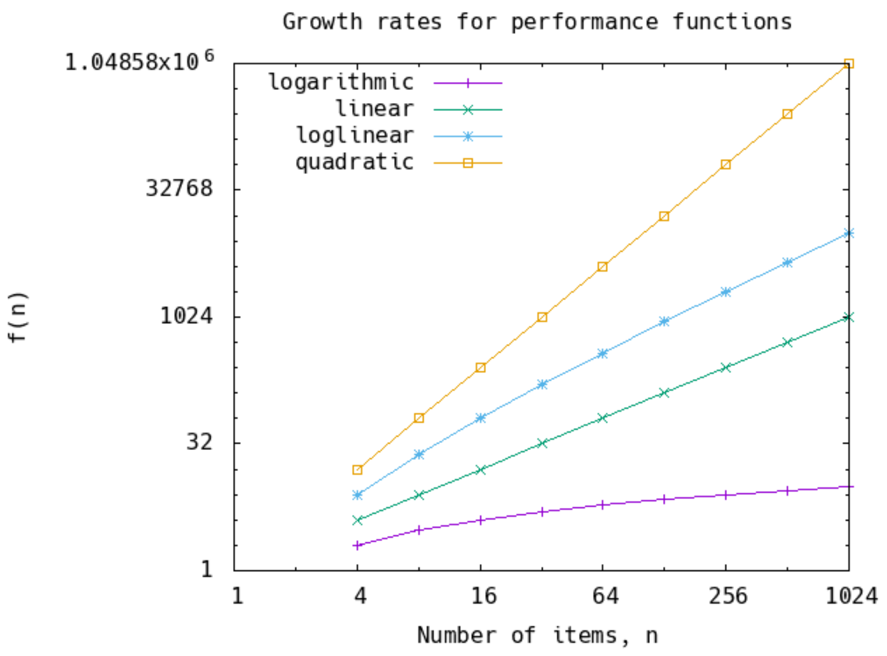
performance behavior. This function grows a little faster than linear, but much more slowly than quadratic. We will encounter this when discussing sorting algorithms, and examples will be provided at that time.

6.3.6 Summary of these performance functions

If we assume that $b = 2$ for logarithmic and loglinear, let's see how these functions grow as a function of n.

n	$\log n$	n	$n \log n$	n^2
4	2	4	8	16
8	3	8	24	64
16	4	16	64	256
32	5	32	160	1,024
64	6	64	384	4,096
128	7	128	896	16,384
256	8	256	2,048	65,536
512	9	512	4,608	262,144
1,024	10	1,024	10,240	1,048,576

The growth rates of these functions are shown graphically below.



Note that this is a log-log graph.

From this data, it should be clear that if one has four different implementations of an algorithm, one being logarithmic, one linear, one loglinear, and one quadratic, then the logarithmic implementation is preferred over the linear implementation, linear preferred over loglinear, and loglinear preferred over the quadratic. This "preference" is transitive, as well.

6.4 Asymptotic notation - the big O

We could spend a significant amount of time determining the running time cost for each basic operation of an algorithm or method. It should come as no surprise that if the running time grows very rapidly, knowing how many instructions one performs for each element simply determines a constant that multiplies the rapidly growing running time function; that function will dominate the runtime performance. Thus, if we can characterize how an algorithm or method relates to our basis set of running time functions, we will understand its asymptotic behavior - i.e., for large n.

Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends toward a particular value or infinity. It is a member of a family of notations invented by Paul Bachmann, Edmund Landau, and others, collectively called *Bachmann-Landau notation*.

For our purposes, big O notation is used to classify algorithms or methods according to how the running time grows as as the input size grows. Thus, big O notation characterizes functions according to their growth rates - different functions with the same growth rate may be represented using the same O notation.

The letter O is used because the growth rate of a function is also referred to as *order of the function*.

Let $f(n)$ and $g(n)$ be functions mapping non-negative integers to real numbers - i.e., number of input items to running time. We say that $f(n)$ is $O(g(n))$ if there is a real constant $k > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq kg(n), for\ n \geq n_0.$$

In typical usage, the formal definition of O notation is not used directly; rather, the O notation for a function $f(n)$ is derived by the following simplification rules:

1. if $f(n)$ is a sum of several terms, if there is one with the largest growth rate, it can be kept, and all others omitted; and
2. if $f(n)$ is a product of several factors, any constants (i.e., terms in the product that do not depend upon n) can be omitted.

Let's look at a number of examples.

- If $f(n) = 5n - 3$, it is $O(n)$ - i.e., its growth is linear. — By rule 1 above, $5n - 3$ reduces to $5n$; by rule 2 above, $5n$ reduces to n; thus, $f(n)$ is $O(n)$.
- If $f(n) = 8n^4 - 4n^2 + 42$, it is $O(n^4)$ - i.e., its growth is quartic. — By rule 1 above, $8n^4 - 4n^2 + 42$ reduces to $8n^4$; by rule 2 above, $8n^4$ reduces to n^4; thus, $f(n)$ is $O(n^4)$.
- If $f(n) = 11n \log n + 23n - 5$, it is $O(n \log n)$ - i.e., its growth is loglinear. — By rule 1 above, $11n \log n + 23n - 5$ reduces to $11n \log n$; by rule 2 above, $11n \log n$ reduces to $n \log n$; thus, $f(n)$ is $O(n \log n)$.

Chapter 7

ArrayLists

We have already seen that C supports arrays of other data types. Once must declare the type of element that can be stored at each index in an array, as well as declaring the number of elements that can be stored in an array. There is no ability at runtime to query an array for its size, nor can the array grow/shrink as one uses it. Thus, we can see that, while one can create arrays, the programmer must remember several pieces of management information in order to be able to effectively use the array.

An *ArrayList* ADT supports dynamic arrays that can grow as needed. We will first implement an ArrayList of integers; this will enable us to become more familiar with building a collection class ADT. We will also discuss the complexity of each of the methods. After we have a working implementation of an ArrayList of integers, we will show the changes necessary to support an ArrayList of arbitrary types.

7.1 Methods on an ArrayList

Obviously, we should be able to retrieve an element at a legal index into the array, and should be able to store an element at a legal index; otherwise, an ArrayList would not be a superset of a C array. Here are some other methods that would be useful in an ArrayList; we would like:

- a constructor in order to create a new ArrayList;
- a method to be able to destroy an ArrayList after we are finished with it;
- a method that enables us to append a new element to the ArrayList; if there is no more room in the ArrayList for the new element, we would like it to grow to accommodate the new element; appending the element has the side effect of incrementing the length of the ArrayList;
- a method to clear all elements from the ArrayList, thus making it empty, just as it is after invocation of the constructor;
- a method to retrieve the element at a particular index (indexing is 0 .. length-1);

- a method to insert an element at a particular index; all current elements from that index onwards are shifted one position to the right; if there is no more room in the ArrayList for the new element, we would like it to grow to accommodate the new element; inserting the element has the side effect of incrementing the length of the ArrayList;
- a method to remove the element at a particular index; all current elements beyond that index are shifted one position to the left; removing the element has the side effect of decrementing the length of the ArrayList;
- a method to set the element at a particular index; and
- a method to return the current number of elements in the ArrayList.

A word about "growing the ArrayList to accommodate new elements" - this is likely to require that a new call to `malloc()` or `realloc()` is made to increase the size of a buffer managed by the ArrayList; this means that the perceived running time for a call to the append method, for example, can be different from one call to the next, since it will be slower if the method invocation causes a call to `realloc()` as compared to an invocation when there is a free slot in the buffer. This difference in running time could be crucial if an ArrayList is being used by code that is performing some soft realtime processing (e.g., reading out an mp3-encoded track). Thus, an ArrayList ADT typically includes the following additional methods:

- the constructor enables the user to specify an initial capacity when creating a new ArrayList;
- a method to ensure that an ArrayList has capacity to accept a known number of elements; if not, it is grown by this call to avoid dynamic growth during calls to append or insert new elements; and
- a method to trim excess capacity from the ArrayList.

Finally, we mentioned earlier in the chapter on ADTs that there are a standard set of methods that we want all collection class ADTs to support:

- a method that returns true if an ArrayList is empty; and
- a method to return an Iterator over the ArrayList.

Normally, there is an additional method that will return a dynamically-allocated array pointing to each element in the collection class; this method also returns the number of elements in that array. The programmer can then linearly proceed through all of the elements of the array, most likely using a `for` loop. The programmer must then remember to free the array that was returned. Since an ArrayList already provides this functionality, this method is not truly needed, although it will provide more rapid access to the elements than calls to the method for getting an element. Our interface definition will include this method for consistency across collection class ADTs.

7.2 An ArrayList of long integers

First let's look at an ArrayList of long integers. The next subsection presents the header file for the ArrayListLong ADT interface. The subsequent subsection presents an implementation for this interface. The final subsection discusses the running time complexity for each method.

7.2.1 The header file for the ArrayListLong ADT

This section provides the header file for the ArrayListLong ADT interface. First we present the entire header file; each method is then discussed separately.

```
#ifndef _ARRAYLISTLONG_H_
#define _ARRAYLISTLONG_H_
#include "iterator.h"                      /* needed for factory method */
typedef struct arraylistlong ArrayListLong;  /* forward reference */
const ArrayListLong *ArrayListLong_create(long capacity);
struct arraylistlong {
    void *self;
    void (*destroy)(const ArrayListLong *al);
    int (*add)(const ArrayListLong *al, long element);
    void (*clear)(const ArrayListLong *al);
    int (*ensureCapacity)(const ArrayListLong *al, long minCapacity);
    int (*get)(const ArrayListLong *al, long index, long *element);
    int (*insert)(const ArrayListLong *al, long index, long element);
    int (*isEmpty)(const ArrayListLong *al);
    int (*remove)(const ArrayListLong *al, long index);
    int (*set)(const ArrayListLong *al, long index, long element);
    long (*size)(const ArrayListLong *al);
    long *(*toArray)(const ArrayListLong *al, long *len);
    int (*trimToSize)(const ArrayListLong *al);
    const Iterator *(*itCreate)(const ArrayListLong *al);
};
#endif /* _ARRAYLISTLONG_H_ */
```

The following discuss each function/method in terms of functionality provided and constraints on function arguments.

- `const ArrayListLong *ArrayListLong_create(long capacity);`
  This creates an ArrayList of long integers; if `capacity > 0`, the initial capacity of the list is `capacity`, otherwise a default capacity is used.

- `void (*destroy)(const ArrayListLong *al);`
  Destroys the ArrayList of long integers.

- `int (*add)(const ArrayListLong *al, long element);`
  Appends `element` to the ArrayList of long integers; dynamically grows the list if

more capacity is needed; increments the length of the list by 1; returns 1 if successful, 0 if there were errors (could not allocate more space).

- `void (*clear)(const ArrayListLong *al);`
  Clears all elements from the ArrayList of long integers; upon return, the list is empty.
- `int (*ensureCapacity)(const ArrayListLong *al, long minCapacity);`
  Ensure that the ArrayList of long integers can hold at least `minCapacity` elements; if not, the buffer is grown to handle that many new elements; returns 1 if successful, 0 if not (`realloc` failure).
- `int (*get)(const ArrayListLong *al, long index, long *element);`
  Returns the element at the specified `index` in the ArrayList of long integers; the value is returned in `*element`. The method returns 1 if successful, 0 if no element at that position.
- `int (*insert)(const ArrayListLong *al, long index, long element);`
  Inserts `element` at the specified `index` in the ArrayList of integers; all current elements from `index` are shifted one position to the right; dynamically grows the list if more capacity is needed; increments the length of the list by 1; returns 1 if successful, 0 if there were errors (could not allocate more space).
- `int (*isEmpty)(const ArrayListLong *al);`
  Returns 1 if ArrayList of long integers is empty, 0 if it is not.
- `int (*remove)(const ArrayListLong *al, long index);`
  Removes the element from `index`; all other elements from `[index + 1, size)` are shifted down one position. Returns 1 if successful, 0 if the position at `index` was unoccupied.
- `int (*set)(const ArrayListLong *al, long index, long element);`
  Replaces the element at `index` with `element`. Returns 1 if successful, 0 if the position at `index` was unoccupied.
- `long (*size)(const ArrayListLong *al);`
  Returns the number of elements in the ArrayList of long integers.
- `long *(*toArray)(const ArrayListLong *al, long *len);`
  Returns an array containing all of the elements of the ArrayList of long integers in the proper sequence `[0..size-1]`; returns the length of the array in `*len`. Returns NULL if unable to allocate array of integers; the caller is responsible for freeing the array when finished with it.
- `int (*trimToSize)(const ArrayListLong *al);`
  Trims the capacity of the ArrayList of long integers to be the current size. Returns 1 if successful, 0 if failure (allocation failure).
- `const Iterator *(*itCreate)(const ArrayListLong *al);`
  Creates a generic iterator to the ArrayList of long integers. Returns a pointer to the Iterator or NULL if failure.

7.2.2 The implementation file for the ArrayListLong ADT

The implementation for an ArrayList of long integers has several similarities to our implementation of the `StringADT`, minus the many string-specific methods. This is because the `StringADT` implemented a sequence of characters, while `ArrayListLong` implements a sequence of long integers. As a result of both ADTs implementing a sequence, is it not unexpected that there will be similarities between the common portions of the implementation.

As with the `StringADT`, we will break up the implementation into bitesize pieces.

7.2.2.1 The preliminaries

```
/* implementation for array list of long integers */

#include "arraylistlong.h"
#include <stdlib.h>

#define DEFAULT_CAPACITY 10L

typedef struct al_data {
    long capacity;
    long size;
    long *theArray;
} AlData;
```

Obviously, we must `#include` the header file to make sure that we define things correctly. We establish the default capacity as `10`. We also define the structure for the instance-specific data; it consists of the size of the allocated array (`capacity`), the number of elements that are stored in the array (`size`), and a pointer to the array allocated on the heap (`theArray`).

7.2.2.2 The `destroy()` and `clear()` methods

```
static void al_destroy(const ArrayListLong *al) {
    AlData *ald = (AlData *)(al->self);
    free(ald->theArray);        /* free array of integers */
    free(ald);                  /* free AlData structure */
    free((void *)al);           /* we free the dispatch table */
}

static void al_clear(const ArrayListLong *al) {
    AlData *ald = (AlData *)(al->self);
    ald->size = 0L;
}
```

The `destroy()` method first frees up the array of integers that was allocated from the heap. It then frees up the structure that contained the instance-specific data. Finally, it frees up the dispatch table.

The `clear()` method simply sets the `size` member of the instance- specific structure to 0, indicating that the ArrayList of integers is empty.

7.2.2.3 The `add()`, `insert()`, `remove()`, and `set()` methods

```
static int al_add(const ArrayListLong *al, long element) {
    AlData *ald = (AlData *)(al->self);
    int status = 1;

    if (ald->capacity <= ald->size) { /* need to reallocate */
        size_t nbytes = 2 * ald->capacity * sizeof(long);
        long *tmp = (long *)realloc(ald->theArray, nbytes);
        if (tmp == NULL)
            status = 0; /* allocation failure */
        else {
            ald->theArray = tmp;
            ald->capacity *= 2;
        }
    }
    if (status)
        ald->theArray[ald->size++] = element;
    return status;
}

static int al_insert(const ArrayListLong *al, long index, long element) {
    AlData *ald = (AlData *)(al->self);
    int status = 1;

    if (index > ald->size)
        return 0;                       /* 0 <= index <= size */
    if (ald->capacity <= ald->size) {   /* need to reallocate */
        size_t nbytes = 2 * ald->capacity * sizeof(long);
        long *tmp = (long *)realloc(ald->theArray, nbytes);
        if (tmp == NULL)
            status = 0; /* allocation failure */
        else {
            ald->theArray = tmp;
            ald->capacity *= 2;
        }
    }
    if (status) {
        long j;
        for (j = ald->size; j > index; j--) /* slide items up */
            ald->theArray[j] = ald->theArray[j-1];
        ald->theArray[index] = element;
        ald->size++;
    }
```

```
        return status;
    }

    static int al_remove(const ArrayListLong *al, long index) {
        AlData *ald = (AlData *)(al->self);
        int status = 0;
        long j;

        if (index >= 0L && index < ald->size) {
            for (j = index + 1; j < ald->size; j++)
                ald->theArray[index++] = ald->theArray[j];
            ald->size--;
            status = 1;
        }
        return status;
    }

    static int al_set(const ArrayListLong *al, long index, long element) {
        AlData *ald = (AlData *)(al->self);
        int status = 0;

        if (index >= 0L && index < ald->size) {
            ald->theArray[index] = element;
            status = 1;
        }
        return status;
    }
```

These are the mutator methods on an ArrayList of integers. Both `add()` and `insert()` must check to see if the buffer of integers has been exhausted, and if so, invoke `realloc()` to allocate a larger buffer; at each reallocation, the capacity of the buffer is doubled. `insert()`, `remove()`, and `set()` much check to see if the index that was passed in the argument list is legal (0 <= `i` < `size` for `remove()` and `set()`, and 0 <= `i` <= `size` for `insert()`). For `insert()`, we need to move all of the data in `[i, size)` up one location in order to insert `element` into the `i`th location. For `remove()`, we need to move all of the data in `(i, size)` down one location.

7.2.2.4 The `get()`, `isEmpty()`, `size()`, `ensureCapacity()`, and `trimToSize()` methods

```
    int al_get(const ArrayListLong *al, long index, long *element) {
        AlData *ald = (AlData *)(al->self);
        int status = 0;

        if (index >= 0L && index < ald->size) {
            *element = ald->theArray[index];
            status = 1;
        }
        return status;
    }

    static int al_isEmpty(const ArrayListLong *al) {
        AlData *ald = (AlData *)(al->self);
        return (ald->size == 0L);
    }

    static long al_size(const ArrayListLong *al) {
        AlData *ald = (AlData *)(al->self);
        return ald->size;
    }

    static int al_ensureCapacity(const ArrayListLong *al, long minCapacity) {
        AlData *ald = (AlData *)(al->self);
        int status = 1;

        if (ald->capacity < minCapacity) {     /* must extend */
            long *tmp = (long *)realloc(ald->theArray, minCapacity * sizeof(long));
            if (tmp == NULL)
                status = 0;                    /* allocation failure */
            else {
                ald->theArray = tmp;
                ald->capacity = minCapacity;
            }
        }
        return status;
    }

    static int al_trimToSize(const ArrayListLong *al) {
        AlData *ald = (AlData *)al->self;
        int status = 0;

        long *tmp = (long *)realloc(ald->theArray, ald->size * sizeof(long));
        if (tmp != NULL) {
            status = 1;
            ald->theArray = tmp;
            ald->capacity = ald->size;
        }
        return status;
    }
```

`isEmpty()` and `size()` simply return a value based upon the `size` member of the instance-specific struct. `get()` simply checks that the specified index is legal, and if so, returns the value stored there in `*element`, returning 1/0 if the index was legal/illegal. `ensureCapacity()` and `trimToSize()` simply use `realloc()` appropriately to provide the requested functionality, returning 1 if the reallocation was successful, 0 if not.

7.2.2.5 The `toArray()` and `itCreate()` methods

```
    /*
```

```
 * local function that duplicates the array of long integers on the heap
 *
 * returns pointer to duplicate array or NULL if malloc failure
 */
static long *arraydupl(AlData *ald) {
    long *tmp = NULL;
    if (ald->size > 0L) {
        size_t nbytes = ald->size * sizeof(long);
        tmp = (long *)malloc(nbytes);
        if (tmp != NULL) {
            long i;

            for (i = 0; i < ald->size; i++)
                tmp[i] = ald->theArray[i];
        }
    }
    return tmp;
}

static long *al_toArray(const ArrayListLong *al, long *len) {
    AlData *ald = (AlData *)al->self;
    long *tmp = arraydupl(ald);

    if (tmp != NULL)
        *len = ald->size;
    return tmp;
}

static const Iterator *al_itCreate(const ArrayListLong *al) {
    AlData *ald = (AlData *)al->self;
    const Iterator *it = NULL;
    long *tmp = arraydupl(ald);

    if (tmp != NULL) {
        it = Iterator_create(ald->size, (void **)tmp);
        if (it == NULL)
            free(tmp);
    }
    return it;
}
```

Both `toArray()` and `itCreate()` need to allocate an array of integers on the heap, so we first define a helper function, `arraydupl()`, that performs this function, returning the allocated array of integers as its function value. The two methods each invoke this helper function; if that call is successful, each then proceeds to complete its functionality.

`itCreate()` invokes the `Iterator` constructor to obtain an interator; since the generic iterator is defined to take an array of `void *` pointers, we have to cast the pointer to the integer array to `void **` when we call the constructor.

7.2.2.6 The `ArrayListLong` constructor

```
static ArrayListLong template = {NULL, al_destroy, al_add, al_clear,
                             al_ensureCapacity, al_get, al_insert, al_isEmpty,
                             al_remove, al_set, al_size, al_toArray,
                             al_trimToSize, al_itCreate};

const ArrayListLong *ArrayListLong_create(long capacity) {
    ArrayListLong *al = (ArrayListLong *)malloc(sizeof(ArrayListLong));

    if (al != NULL) {
        AlData *ald = (AlData *)malloc(sizeof(AlData));
        if (ald != NULL) {
            long cap;
            long *array = NULL;

            cap = (capacity <= 0) ? DEFAULT_CAPACITY : capacity;
            array = (long *) malloc(cap * sizeof(long));
            if (array != NULL) {
                ald->capacity = cap;
                ald->size = 0L;
                ald->theArray = array;
                *al = template;
                al->self = ald;
            } else {
                free(ald);
                free(al);
                al = NULL;
            }
        } else {
            free(al);
            al = NULL;
        }
    }
    return al;
}
```

As with our `StringADT`, we create a static template of the dispatch table so we can initialize the dispatch table by copying the template `struct` for each new `ArrayListLong` instance. The logic in the constructor is straightforward:

1. allocate a new dispatch table;
2. if that was successful, allocate the instance-specific data structure;
3. if that was successful, allocate the initial long integer array from the heap; its capacity is either the value passed in the argument to the constructor, or the default capacity;
4. initialize the members of the instance-specific data structure, and the dispatch table.

If all heap allocations were successful, return the pointer to the dispatch table. If any of the heap allocations failed, free any previous allocations that were successful, and return

NULL.

7.2.3 The complexity of each of the methods

Now let's characterize the running time complexity of each of the methods. In the characterizations below, we will use n to mean the number of elements in the array; this is stored in the `size` member of the instance-specific data structure for the array list.

Let's assume that the initial capacity of the array list is large enough so that the neither `add()` nor `insert()` cause the array to have to resize. Under this assumption, the complexity of each method is as follows.

7.2.3.1 `add()`

Since the capacity is always sufficient, the code simply stores the element in the next free location, then increments the next free location index. This is independent of the number of items in the array, so this method is $O(1)$ complexity.

7.2.3.2 `insert()`

Since the capacity is always sufficient, and assuming that a legal index has been provided, all of the elements from [index, size) must be moved up one index in the array; for an arbitrary index, this means that we move some fraction of the array, thus this activity is $O(n)$. After we have moved the fraction of the array, there is a fixed cost to store the element at the now vacated index; this is $O(1)$. Thus, the overall complexity if $O(n)$.

7.2.3.3 `remove()`

Assuming that a legal index has been provided, all of the elements from (index, size) must be moved down one index in the array; for an arbitrary index, this means that we move some fraction of the array, thus this activity is $O(n)$.

7.2.3.4 `set()`

Assuming that a legal index has been provided, the code simply stores the element at that index. This is independent of the number of items in the array, so this method is $O(1)$ complexity.

7.2.3.5 `get()`

Assuming that a legal index has been provided, the code simply stores the value at that index into the integer pointer passed as an argument. This is independent of the number of items in the array, so this method is $O(1)$ complexity.

7.2.3.6 `isEmpty()`

The code simply compares the current size against 0; this is independent of the number of items in the array, so this method is $O(1)$ complexity.

7.2.3.7 `size()`

The code simply returns the current size; this is independent of the number of items in the array, so this method is $O(1)$ complexity.

7.2.3.8 `toArray()`

We have no insight into the implementation of `malloc()`, so we do not know its complexity as a function of the number of items in the array. Most heap allocation systems are extremely efficient at finding a memory block of the required size; buddy heaps have a complexity of $O(\log m)$, where m is the number of memory blocks that it uses. It is impossible to correlate m with n, thus for for `malloc()` and `free()` we will assume a complexity of $O(1)$. Once the allocation has been made, the code in `arraydupl()` copies each of the integers into the new array. Thus, this aspect of the code is linearly dependent upon the number of items in the array, $O(n)$. Thus, the overall complexity is $O(n)$.

7.2.3.9 `itCreate()`

As for `toArray()` above, since it calls `arraydupl()` to duplicate the integer array on the heap, the running time complexity for this method is $O(n)$.

7.2.3.10 `ensureCapacity()`, `trimToSize()`, and `add()`/`insert()` if they cause the buffer to grow

`realloc()` must copy the current contents of the buffer into the newly-allocated buffer; since the number of bytes to copy is a multiple of the number of integers in the buffer, `realloc()` is $O(n)$. Thus, any method that must perform a `realloc()` will have an $O(n)$ component to its running time complexity. This indicates that `add()`, which has an $O(1)$ complexity in the absence of a call to `realloc()`, has a worst case complexity of $O(n)$ if the buffer has to grow. `insert()` was already $O(n)$, so the complexity does not change if the buffer has to grow.

7.2.3.11 `clear()`

This code simply assigns 0 to the current size of the array list; as such, the method is $O(1)$.

7.2.3.12 `destroy()`

As we stated above, we assume `free()` is $O(1)$, so the complexity of this method is $O(1)$.

7.2.3.13 `ArrayListLong_create()`

This code makes three calls to `malloc()`, which we have asserted is $O(1)$. Thus, the complexity for the constructor is $O(1)$.

7.3 A generic ArrayList

While the preceding ADT provides a very functional interface to an ArrayList of long integers, would we have to implement another ADT for an ArrayList of doubles? Or an ArrayList of structures? Or an ArrayList of ArrayLists? We need a way to make our data structures generic.

You will recall that all pointer types in C can be cast to `void *` and back again without loss of information. In fact, the implementation for the StringADT in chapter 4 and the ArrayListLong ADT have taken advantage of this with respect to the `self` member in the dispatch table; it is declared as a `void *`, yet in the constructor a legitimate pointer is assigned to `self`. In each of the methods, the first statement is a cast to convert `self` to a pointer to the instance-specific data type. We will take advantage of `void *` to make our container data structures generic.

If we review the functions of an ArrayList from Section 7.1, we can see that the functionality provided should work for any types of items. Obviously, methods to add or retrieve items from the ArrayList (`add()`, `get()`, `insert()`, and `set()`) will need to have different signatures than those in the ArrayListLong ADT. Additionally, we expect that the data stored in an ArrayList instance will be allocated from the heap, so we will also change the signatures for `destroy()`, `clear()`, `remove()`, and `set()` to assist the programmer in freeing up heap memory.

7.3.1 Header file for a generic ArrayList

```
#ifndef _ARRAYLIST_H_
#define _ARRAYLIST_H_

/* BSD header removed to save space */

#include "iterator.h"                  /* needed for factory method */

/* interface definition for generic arraylist implementation
 *
 * patterned roughly after Java 6 ArrayList generic class
 */

typedef struct arraylist ArrayList;     /* forward reference */

/* create an arraylist with the specified capacity; if capacity == 0, a
 * default initial capacity (10 elements) is used
 *
 * returns a pointer to the array list, or NULL if there are malloc() errors
 */
const ArrayList *ArrayList_create(long capacity);

/* now define struct arraylist */
struct arraylist {
/* the private data of the array list */
    void *self;

/* destroys the arraylist; for each occupied index, if freeFxn != NULL,
 * it is invoked on the element at that position; the storage associated with
 * the arraylist is then returned to the heap
 */
    void (*destroy)(const ArrayList *al, void (*freeFxn)(void *element));

/* appends `element' to the arraylist; if no more room in the list, it is
 * dynamically resized
 *
 * returns 1 if successful, 0 if unsuccessful (malloc errors)
 */
    int (*add)(const ArrayList *al, void *element);

/* clears all elements from the arraylist; for each occupied index,
 * if freeFxn != NULL, it is invoked on the element at that position;
 * any storage associated with the element in the arraylist is then
 * returned to the heap
 *
 * upon return, the arraylist will be empty
 */
    void (*clear)(const ArrayList *al, void (*freeFxn)(void *element));

/* ensures that the arraylist can hold at least `minCapacity' elements
 *
 * returns 1 if successful, 0 if unsuccessful (malloc failure)
 */
    int (*ensureCapacity)(const ArrayList *al, long minCapacity);

/* returns the element at the specified position in this list in `*element'
 *
 * returns 1 if successful, 0 if no element at that position
 */
    int (*get)(const ArrayList *al, long index, void **element);

/* inserts `element' at the specified position in the arraylist;
 * all elements from `i' onwards are shifted one position to the right;
 * if no more room in the list, it is dynamically resized;
 * if the current size of the list is N, legal values of i are in the
 * interval [0, N]
 *
 * returns 1 if successful, 0 if unsuccessful (malloc errors)
 */
    int (*insert)(const ArrayList *al, long index, void *element);

/* returns 1 if arraylist is empty, 0 if it is not */
    int (*isEmpty)(const ArrayList *al);

/* removes the `i'th element from the list, returns the value that
 * occupied that position in `*element'; all elements from [i+1, size-1] are
 * shifted down one position
 *
 * returns 1 if successful, 0 if `i'th position was not occupied
 */
    int (*remove)(const ArrayList *al, long index, void **element);

/* relaces the `i'th element of the arraylist with `element';
 * returns the value that previously occupied that position in `*previous'
 *
 * returns 1 if successful
 * returns 0 if `i'th position not currently occupied
 */
    int (*set)(const ArrayList *al, long index, void *element, void **previous);

/* returns the number of elements in the arraylist */
    long (*size)(const ArrayList *al);

/* returns an array containing all of the elements of the list in
 * proper sequence (from first to last element); returns the length of
 * the list in `*len'
 *
 * returns pointer to void * array of elements, or NULL if malloc failure
 *
 * NB - the caller is responsible for freeing the void * array when finished
 *      with it
 */
    void **(*toArray)(const ArrayList *al, long *len);
```

```
/* trims the capacity of the arraylist to be the list's current size
 *
 * returns 1 if successful, 0 if failure (malloc errors)
 */
    int (*trimToSize)(const ArrayList *al);

/* create generic iterator to this arraylist
 *
 * returns pointer to the Iterator or NULL if failure
 */
    const Iterator *(*itCreate)(const ArrayList *al);
};

#endif /* _ARRAYLIST_H_ */
```

Changes to the ArrayListLong interface are as follows:

- `void (*destroy)(const ArrayList *al, void (*freeFxn)(void *element));`
  `void (*clear)(const ArrayList *al, void (*freeFxn)(void *element));`
  These two methods give the caller the opportunity to provide a function pointer to free up any heap-allocated storage before the ArrayList is destroyed or cleared. If the caller does not want this feature, simply pass `NULL` as the second parameter to these calls. If the second parameter is *not* `NULL`, then this function is invoked on each element in the array before `destroy()` returns its heap-allocated storage, and on each element in the array before `clear()` indicates that the ArrayList is empty.

- `int (*add)(const ArrayList *al, void *element);`
  `int (*get)(const ArrayList *al, long index, void **element);`
  `int (*insert)(const ArrayList *al, long index, void *element);`
  `int (*remove)(const ArrayList *al, long index, void **element);`
  `int (*set)(const ArrayList *al, long index, void *element, void **previous);`
  `element` is everywhere declared a `void *`. For both `remove()` and `set()`, an additional argument is provided for the implementation to return the element that was deleted/replaced.

- `void **(*toArray)(const ArrayList *al, long *len);`
  `toArray()` returns an array of `void *` pointers.

- `const Iterator *(*itCreate)(const ArrayList *al);`
  `itCreate()` creates a generic iterator over the ArrayList.

7.3.2 Implementation for the generic ArrayList

```
/* BSD Header removed to save space */

/*
 * implementation for generic array list
 */

#include "arraylist.h"
#include <stdlib.h>

#define DEFAULT_CAPACITY 10L

typedef struct al_data {
    long capacity;
    long size;
    void **theArray;
} AlData;

/*
 * traverses arraylist, calling freeFxn on each element
 */
static void purge(AlData *ald, void (*freeFxn)(void *element)) {

    if (freeFxn != NULL) {
        long i;

        for (i = 0L; i < ald->size; i++) {
            (*freeFxn)(ald->theArray[i]); /* user frees element storage */
            ald->theArray[i] = NULL;
        }
    }
}

static void al_destroy(const ArrayList *al, void (*freeFxn)(void *element)) {
    AlData *ald = (AlData *)(al->self);
    purge(ald, freeFxn);
    free(ald->theArray);   /* we free array of pointers */
    free(ald);
    free((void *)al);   /* we free the ArrayList struct */
}

static int al_add(const ArrayList *al, void *element) {
    AlData *ald = (AlData *)(al->self);
    int status = 1;

    if (ald->capacity <= ald->size) { /* need to reallocate */
        size_t nbytes = 2 * ald->capacity * sizeof(void *);
        void **tmp = (void **)realloc(ald->theArray, nbytes);
        if (tmp == NULL)
            status = 0; /* allocation failure */
        else {
            ald->theArray = tmp;
            ald->capacity *= 2;
        }
    }
    if (status)
        ald->theArray[ald->size++] = element;
    return status;
}

static void al_clear(const ArrayList *al, void (*freeFxn)(void *element)) {
    AlData *ald = (AlData *)(al->self);
    purge(ald, freeFxn);
    ald->size = 0L;
}

static int al_ensureCapacity(const ArrayList *al, long minCapacity) {
    AlData *ald = (AlData *)(al->self);
    int status = 1;

    if (ald->capacity < minCapacity) { /* must extend */
        void **tmp = (void **)realloc(ald->theArray, minCapacity * sizeof(void *));
        if (tmp == NULL)
            status = 0; /* allocation failure */
        else {
            ald->theArray = tmp;
            ald->capacity = minCapacity;
        }
    }
    return status;
}

int al_get(const ArrayList *al, long index, void **element) {
    AlData *ald = (AlData *)(al->self);
    int status = 0;

    if (index >= 0L && index < ald->size) {
        *element = ald->theArray[index];
        status = 1;
    }
    return status;
}

static int al_insert(const ArrayList *al, long index, void *element) {
    AlData *ald = (AlData *)(al->self);
    int status = 1;

    if (index > ald->size)
        return 0; /* 0 <= index <= size */
    if (ald->capacity <= ald->size) { /* need to reallocate */
        size_t nbytes = 2 * ald->capacity * sizeof(void *);
        void **tmp = (void **)realloc(ald->theArray, nbytes);
        if (tmp == NULL)
            status = 0; /* allocation failure */
        else {
            ald->theArray = tmp;
            ald->capacity *= 2;
        }
    }
    if (status) {
        long j;
        for (j = ald->size; j > index; j--) /* slide items up */
            ald->theArray[j] = ald->theArray[j-1];
        ald->theArray[index] = element;
        ald->size++;
    }
    return status;
}

static int al_isEmpty(const ArrayList *al) {
    AlData *ald = (AlData *)(al->self);
    return (ald->size == 0L);
}

static int al_remove(const ArrayList *al, long index, void **element) {
    AlData *ald = (AlData *)(al->self);
    int status = 0;
    long j;

    if (index >= 0L && index < ald->size) {
        *element = ald->theArray[index];
        for (j = index + 1; j < ald->size; j++)
            ald->theArray[index++] = ald->theArray[j];
        ald->size--;
        status = 1;
    }
    return status;
}

static int al_set(const ArrayList *al, long index, void *element, void **previous) {
    AlData *ald = (AlData *)(al->self);
    int status = 0;

    if (index >= 0L && index < ald->size) {
        *previous = ald->theArray[index];
        ald->theArray[index] = element;
        status = 1;
    }
    return status;
}

static long al_size(const ArrayList *al) {
    AlData *ald = (AlData *)(al->self);
    return ald->size;
}

/*
 * local function that duplicates the array of void * pointers on the heap
 *
 * returns pointer to duplicate array or NULL if malloc failure
 */
```

```
static void **arraydupl(AlData *ald) {
    void **tmp = NULL;
    if (ald->size > 0L) {
        size_t nbytes = ald->size * sizeof(void *);
        tmp = (void **)malloc(nbytes);
        if (tmp != NULL) {
            long i;

            for (i = 0; i < ald->size; i++)
                tmp[i] = ald->theArray[i];
        }
    }
    return tmp;
}

static void **al_toArray(const ArrayList *al, long *len) {
    AlData *ald = (AlData *)al->self;
    void **tmp = arraydupl(ald);

    if (tmp != NULL)
        *len = ald->size;
    return tmp;
}

static int al_trimToSize(const ArrayList *al) {
    AlData *ald = (AlData *)al->self;
    int status = 0;

    void **tmp = (void **)realloc(ald->theArray, ald->size * sizeof(void *));
    if (tmp != NULL) {
        status = 1;
        ald->theArray = tmp;
        ald->capacity = ald->size;
    }
    return status;
}

static const Iterator *al_itCreate(const ArrayList *al) {
    AlData *ald = (AlData *)al->self;
    const Iterator *it = NULL;
    void **tmp = arraydupl(ald);

    if (tmp != NULL) {
        it = Iterator_create(ald->size, tmp);
        if (it == NULL)
            free(tmp);
    }
    return it;
}

static ArrayList template = {NULL, al_destroy, al_add, al_clear,
                             al_ensureCapacity, al_get, al_insert, al_isEmpty,
                              al_remove, al_set, al_size, al_toArray,
                              al_trimToSize, al_itCreate};

const ArrayList *ArrayList_create(long capacity) {
    ArrayList *al = (ArrayList *)malloc(sizeof(ArrayList));

    if (al != NULL) {
        AlData *ald = (AlData *)malloc(sizeof(AlData));
        if (ald != NULL) {
            long cap;
            void **array = NULL;

            cap = (capacity <= 0) ? DEFAULT_CAPACITY : capacity;
            array = (void **) malloc(cap * sizeof(void *));
            if (array != NULL) {
                ald->capacity = cap;
                ald->size = 0L;
                ald->theArray = array;
                *al = template;
                al->self = ald;
            } else {
                free(ald);
                free(al);
                al = NULL;
            }
        } else {
            free(al);
            al = NULL;
        }
    }
    return al;
}
```

7.3.3 Example program using the generic ArrayList

How might one use the ArrayList in anger? Here is a simple program that reads each line of input from `stdin`, duplicates each line on the heap, and appends the line to an ArrayList. The program can do anything it likes with the ArrayList before it destroys it. As you can see, it takes advantage of the 2nd argument to allow `destroy()` to return the allocated lines on the heap.

```
#include "arraylist.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define UNUSED __attribute__((unused))

int main(UNUSED int argc, UNUSED char *argv[]) {
    const ArrayList *al;
    char buf[4096];
    long index;

    if ((al = ArrayList_create(0L)) == NULL) {
        fprintf(stderr, "Unable to create ArrayList to hold input lines\n");
        return 1;
    }
    while (fgets(buf, sizeof buf, stdin) != NULL) {
        char *sp = strdup(buf);         /* make a copy on the heap */
        if (!al->add(al, sp)) {
            fprintf(stderr, "Unable to append line to ArrayList: %s", sp);
            al->destroy(al, free);
            return 1;
        }
    }
    for (index = 0; index < al->size(al); index++) {
        char *sp;
        if (!al->get(al, index, (void **)&sp)) {
            fprintf(stderr, "Unable to retrieve line %ld\n", index);
            al->destroy(al, free);
            return 1;
        }
        printf("%s", sp);
    }
    al->destroy(al, free);
    return 0;
}
```

This particular version of the program simply accesses each element in the ArrayList and prints it to standard output. After compiling this and linking it with `arraylist.o` and `iterator.o`, you can test this using commands of the form:

```
$ ./alfile <alfile.c | diff - alfile.c
$ valgrind ./alfile <alfile.c >/dev/null
```

If the program is working correctly, the first command should yield no output, and the second should indicate that there are absolutely no memory leaks.

Chapter 8

Stacks

A *stack* is an object container that conforms to the *last in, first out* discipline: one may *push* an element onto the stack, and one may *pop* the most recently added element from the stack; usually, one may also *peek* at the most recently added element without removing it from the stack.

Why call it a stack? Consider the analogy of a set of rectangular physical objects "stacked" on top of each other; it is easy to add ("push") another such object on the top of the stack, and it is easy to remove ("pop") the top element from the stack. Attempting to remove an element that is not at the top of the stack is much more difficult.

A stack can have bounded capacity; if a `push()` method is invoked when the stack is full, an exception is raised; as you have probably already deduced, an exception is raised if a `pop()` or a `peek()` operation is invoked on an empty stack, as well. It is often more useful to the programmer if the stack has unbounded capacity; of course, it's not truly unbounded, simply an attempt is made to grow the stack when is it full, much as we did with our ArrayList implementation.

8.1 A stack of long integers

As we did with ArrayLists, first let's look at how one would define and implement a stack of long integers. We will first implement it as a bounded stack, then show the changes required to make it unbounded.

8.1.1 The interface specification

```
#ifndef _STACKLONG_H_
#define _STACKLONG_H_

typedef struct stacklong StackLong;            /* forward reference */
```

```
const StackLong *StackLong_create(long capacity);
struct stacklong {
    void *self;
    void (*destroy)(const StackLong *st);
    void (*clear)(const StackLong *st);
    int (*push)(const StackLong *st, long element);
    int (*pop)(const StackLong *st, long *element);
    int (*peek)(const StackLong *st, long *element);
    long (*size)(const StackLong *st);
    int (*isEmpty)(const StackLong *st);
};

#endif /* _STACKLONG_H_ */
```

By now, the general structure of the interface should be familiar. When we are first introducing a new data structure focused on long integers, we will dispense with the `toArray()` and `itCreate()` methods to simplify the presentation. We will reintroduce them when we introduce the generic version.

The semantics of the constructor and the methods are as follows:

- `const StackLong *StackLong_create(long capacity);`
  create a new instance of a Stack of long integers that can hold `capacity` elements; returns a pointer to the dispatch table, or `NULL` if there were heap-allocation errors;
- `void (*destroy)(const StackLong *st);`
  destroy the Stack of long integers; all heap-allocated memory is returned;
- `void (*clear)(const StackLong *st);`
  purge all elements from the stack; upon return, the stack is empty;
- `int (*push)(const StackLong *st, long element);`
  push `element` onto the stack; if the stack cannot accommodate the new element (it is full), the function return is 0; otherwise, it is 1;
- `int (*pop)(const StackLong *st, long *element);`
  remove the element at the top of the stack, returning it in `*element`; if the stack has no elements, the function return is 0; otherwise, it is 1;
- `int (*peek)(const StackLong *st, long *element);`
  return the element at the top of the stack in `*element` *without* removing it from the stack; if the stack has no elements, the function return is 0; otherwise, it is 1;
- `long (*size)(const StackLong *st);`
  return the number of elements in the stack; and
- `int (*isEmpty)(const StackLong *st);`
  return 1 if the stack is empty, 0 otherwise.

8.1.2 The implementation

8.1.2.1 The preliminaries

As with the ArrayList implementation, the secret to an efficient and successful implementation is in the choice of the instance-specific data structure.

```
#include "stacklong.h"
#include <stdlib.h>

typedef struct st_data {
    long capacity;
    long next;
    long *theArray;
} StData;
```

`next` is an index into `theArray[]` where the next `push()` will store the element. The top element on the stack is always found at `next-1`. `capacity` indicates the number of elements that can be pushed onto the stack before overflow would occur (remember, this is a bounded stack).

8.1.2.2 The constructor

This looks very similar to the constructor for ArrayListLong. Allocate a dispatch table from the heap, allocate an instance-specific data structure from the heap, allocate an array of longs from the heap. If all allocations are successful, initialize the instance-specific data structure and dispatch table, and return to the caller. If any heap-allocation failures, free up any heap-allocations prior to the failure and return `NULL`.

```
static StackLong template = {
    NULL, st_destroy, st_clear, st_push, st_pop, st_peek, st_size, st_isEmpty
};

const StackLong *StackLong_create(long capacity) {
    StackLong *st = (StackLong *)malloc(sizeof(StackLong));

    if (st != NULL) {
        StData *std = (StData *)malloc(sizeof(StData));

        if (std != NULL) {
            long *array = (long *)malloc(capacity * sizeof(long));;

            if (array != NULL) {
                std->capacity = capacity;
                std->next = 0L;
                std->theArray = array;
                *st = template;
                st->self = std;
            } else {
                free(std);
                free(st);
                st = NULL;
            }
        } else {
            free(st);
            st = NULL;
        }
    }
    return st;
}
```

8.1.2.3 destroy() and clear()

`destroy()` frees the heap-allocated memory in the reverse order to the constructor. `clear()` simply resets `next` to 0, indicating that the stack is empty.

```
static void st_destroy(const StackLong *st) {
    StData *std = (StData *)st->self;

    free(std->theArray);        /* free array of longs */
    free(std);                  /* free structure with instance data */
    free((void *)st);           /* free dispatch table */
}

static void st_clear(const StackLong *st) {
    StData *std = (StData *)st->self;

    std->next = 0L;
}
```

8.1.2.4 push(), pop(), and peek()

`push()` checks to see if there is room on the stack for another element; if so, it stores it into `theArray[next]`, increments `next`, and returns 1 as the function value; if not, it returns 0 as the function value.

`pop()` checks to see if `next > 0`; if so, the value of `theArray[next-1]` is stored into `*element`, `next` is decremented, and 1 is returned as the function value; if not, 0 is returned as the function value.

`peek()` checks to see if `next > 0`; if so, the value of `theArray[next-1]` is stored into `*element` and 1 is returned as the function value; if not, 0 is returned as the function value.

```
static int st_push(const StackLong *st, long element) {
    StData *std = (StData *)st->self;
    int status = (std->next < std->capacity);
    if (status)
        std->theArray[std->next++] = element;
    return status;
}

static int st_pop(const StackLong *st, long *element) {
    StData *std = (StData *)st->self;
    int status = 0;

    if (std->next > 0L) {
        *element = std->theArray[--std->next];
        status = 1;
    }
    return status;
}

static int st_peek(const StackLong *st, long *element) {
    StData *std = (StData *)st->self;
    int status = 0;

    if (std->next > 0L) {
        *element = std->theArray[std->next - 1];
        status = 1;
    }
    return status;
}
```

8.1.2.5 size() and isEmpty()

Since arrays in C are 0-based, the call to `size()` simply returns the current value of `next`. Likewise, `isEmpty()` simply returns `next == 0L`.

```
static long st_size(const StackLong *st) {
    StData *std = (StData *)st->self;

    return std->next;
}

static int st_isEmpty(const StackLong *st) {
    StData *std = (StData *)st->self;

    return (std->next == 0L);
}
```

8.1.3 Complexity of the constructor and methods

As we did with ArrayLists, let's estimate the complexity of the constructor and each of the methods.

- `StackLong_create()` - when this function is called, there are no elements in the Stack; we perform three calls to `malloc()`, which as we have discussed in the ArrayList section, we consider $O(1)$ for the purposes of this discussion. Thus, the complexity for the constructor is $O(1)$.
- `push()` - this method simply checks to see that `next < capacity`, and if true, stores the element in the array at that index and increments `next`. This is independent of the number of elements currently in the stack, thus the complexity for `push()` is $O(1)$.
- `pop()` - this method simply checks to see that `next > 0`, and if true, stores the element at `next-1` in the pointer argument and decrements `next`. This is independent of the number of elements currently in the stack, thus the complexity for `pop()` is $O(1)$.
- `peek()` - this method simply checks to see that `next > 0`, and if true, stores the element at `next-1` in the pointer argument. This is independent of the number of elements currently in the stack, thus the complexity for `peek()` is $O(1)$.
- `destroy()` - when this function is called, we perform three calls to `free()`, which as we have discussed in the ArrayList section, we consider $O(1)$ for the purposes of this discussion. Thus, the complexity for `destroy()` is $O(1)$.
- `clear()` - this function simply sets `next` to 0. This is independent of the number of elements currently in the stack, thus the complexity for `clear()` is $O(1)$.
- `size()` - this function simply returns the value of `next`. This is independent of the number of elements currently in the stack, thus the complexity for `size()` is $O(1)$.
- `isEmpty()` - this function simply returns the value of `next == 0L`. This is independent of the number of elements currently in the stack, thus the complexity for `size()` is $O(1)$.

From this discussion, we can see that for a bounded Stack of long integers, the constructor and *all* of the methods are $O(1)$. Thus, if your problem needs a container exhibiting *last in, first out* functionality, you cannot do better than a stack that is implemented using an array.

8.1.4 Changes needed to make the stack unbounded

8.1.4.1 The interface

No change is required to the signatures for the constructor or the methods, although the semantics for the value of `capacity` in the constructor is slightly different. If `capacity == 0L`, this tells the implementation to use a default stack size for the initial array. This default capacity should be `#define`'d in the header file, as in

```
#define DEFAULT_CAPACITY 50
```

8.1.4.2 The instance-specific data structure

The array of long integers will need to be made larger if `push()` is invoked when the current stack is already full. The implementation can choose to simply double the size of the array whenever this occurs - this leads to exponential growth of the array. Alternatively, the implementation can choose to grow the array by a constant increment each time, leading to linear growth of the array. If linear growth is chosen, the increment can be a defined constant for all stacks, or it can be a stack-specific value, usually dependent upon the initial capacity specified in the constructor.

If a stack-specific linear growth increment is to be used, then an additional member must be added to `StData`, typically called `increment`. It will look as follows:

```
typedef struct st_data {
    long capacity;
    long next;
    long increment;
    long *theArray;
} StData;
```

The changes below will assume stack-specific linear growth using this structure for the instance-specific data.

8.1.4.3 `destroy()`, `clear()`, `pop()`, `peek()`, `size()`, and `isEmpty()`

No change required, since none of these methods can cause the array of longs to have to grow.

8.1.4.4 The constructor

After successful allocation of the instance-specific struct, we need to 1) determine the initial capacity of the stack, and 2) place a value in the `increment` member of that struct. The following code replaces the lines from "`if(std != NULL) {`" up to the first line consisting of "`} else {`".

```
if (std != NULL) {
    long cap;
    long *array = NULL;

    cap = (capacity <= 0L) ? DEFAULT_CAPACITY : capacity;
```

```
        array = (long *)malloc(cap * sizeof(long));
        if (array != NULL) {
            std->capacity = cap;
            std->increment = cap;
            std->next = 0L;
            std->theArray = array;
            *st = template;
            std->self = std;
        } else {
```

8.1.4.5 push()

The implementation for `push()` shown above does nothing if `next >= capacity`. The following code will grow the array when it is full, and should simply be inserted above the "`if (status)`" line in the previous implementation.

```
    if (! status) {          /* need to reallocate */
        size_t nbytes = (std->capacity + std->increment) * sizeof(long);
        long *tmp = (long *)realloc(std->theArray, nbytes);

        if (tmp != NULL) {
            status = 1;
            std->theArray = tmp;
            std->capacity += std->increment;
        }
    }
```

8.2 A generic stack

The following show the header and implementation files for a stack in which `void *` elements are stored. We have also included the `toArray()` and `itCreate()` methods, along with the `freeFxn()` 2nd arguments for `destroy()` and `clear()`. The path from `StackLong` to `Stack` should now be understandable, so no additional discussion will be provided.

8.2.0.1 Interface for a generic stack

```
#ifndef _STACK_H_
#define _STACK_H_

/* BSD header removed to save space */
```

```
/*
 * interface definition for generic stack
 *
 * patterned roughly after Java 6 Stack interface
 */

#include "iterator.h"                  /* needed for factory method */

#define DEFAULT_CAPACITY 50L

typedef struct stack Stack;            /* forward reference */

/*
 * create a stack with the specified capacity; if capacity == 0L, a
 * default initial capacity (50 elements) is used
 *
 * returns a pointer to the stack, or NULL if there are malloc() errors
 */
const Stack *Stack_create(long capacity);

/*
 * now define struct stack
 */
struct stack {
/*
 * the private data of the stack
 */
    void *self;

/*
 * destroys the stack; for each occupied position, if freeFxn != NULL,
 * it is invoked on the element at that position; the storage associated with
 * the stack is then returned to the heap
 */
    void (*destroy)(const Stack *st, void (*freeFxn)(void *element));

/*
 * clears all elements from the stack; for each occupied position,
 * if freeFxn != NULL, it is invoked on the element at that position;
 * the stack is then re-initialized
 *
 * upon return, the stack is empty
 */
    void (*clear)(const Stack *st, void (*freeFxn)(void *element));

/*
 * pushes `element' onto the stack; if no more room in the stack, it is
 * dynamically resized
 *
 * returns 1 if successful, 0 if unsuccessful (malloc errors)
 */
```

```
    int (*push)(const Stack *st, void *element);

/*
 * pops the element at the top of the stack into `*element'
 *
 * returns 1 if successful, 0 if stack was empty
 */
    int (*pop)(const Stack *st, void **element);

/*
 * peeks at the top element of the stack without removing it;
 * returned in `*element'
 *
 * return 1 if successful, 0 if stack was empty
 */
    int (*peek)(const Stack *st, void **element);

/*
 * returns the number of elements in the stack
 */
    long (*size)(const Stack *st);

/*
 * returns true if the stack is empty, false if not
 */
    int (*isEmpty)(const Stack *st);

/*
 * returns an array containing all of the elements of the stack in
 * proper sequence (from top to bottom element); returns the length of the
 * array in `*len'
 *
 * returns pointer to void * array of elements, or NULL if malloc failure
 *
 * The array of void * pointers is allocated on the heap, so must be returned
 * by a call to free() when the caller has finished using it.
 */
    void **(*toArray)(const Stack *st, long *len);

/*
 * creates generic iterator to this stack;
 * successive next calls return elements in the proper sequence (top to bottom)
 *
 * returns pointer to the Iterator or NULL if malloc failure
 */
    const Iterator *(*itCreate)(const Stack *st);
};

#endif /* _STACK_H_ */
```

8.2.0.2 Implementation for a generic stack

```
/* BSD header removed to save space */

/*
 * implementation for generic stack
 */

#include "stack.h"
#include <stdlib.h>

typedef struct st_data {
    long capacity;
    long increment;
    long next;
    void **theArray;
} StData;

/*
 * local function - traverses stack, applying user-supplied function
 * to each element; if freeFxn is NULL, nothing is done
 */

static void purge(StData *std, void (*freeFxn)(void*)) {
    if (freeFxn != NULL) {
        long i;

        for (i = 0L; i < std->next; i++)
            (*freeFxn)(std->theArray[i]);       /* user frees elem storage */
    }
}

static void st_destroy(const Stack *st, void (*freeFxn)(void *element)) {
    StData *std = (StData *)st->self;

    purge(std, freeFxn);
    free(std->theArray);                /* free array of pointers */
    free(std);                          /* free structure with instance data */
    free((void *)st);                   /* free dispatch table */
}

static void st_clear(const Stack *st, void (*freeFxn)(void *element)) {
    StData *std = (StData *)st->self;

    purge(std, freeFxn);
    std->next = 0L;
}

static int st_push(const Stack *st, void *element) {
    StData *std = (StData *)st->self;
    int status = (std->next < std->capacity);
    if (! status) {    /* need to reallocate */
        size_t nbytes = (std->capacity + std->increment) * sizeof(void *);
        void **tmp = (void **)realloc(std->theArray, nbytes);

        if (tmp != NULL) {
            status = 1; /* allocation failure */
            std->theArray = tmp;
            std->capacity += std->increment;
        }
    }
    if (status)
        std->theArray[std->next++] = element;
    return status;
}

static int st_pop(const Stack *st, void **element) {
    StData *std = (StData *)st->self;
    int status = 0;

    if (std->next > 0L) {
        *element = std->theArray[--std->next];
        status = 1;
    }
    return status;
}

static int st_peek(const Stack *st, void **element) {
    StData *std = (StData *)st->self;
    int status = 0;

    if (std->next > 0L) {
        *element = std->theArray[std->next - 1];
        status = 1;
    }
    return status;
}

static long st_size(const Stack *st) {
    StData *std = (StData *)st->self;

    return std->next;
}

static int st_isEmpty(const Stack *st) {
    StData *std = (StData *)st->self;

    return (std->next == 0L);
}

/*
 * local function - duplicates array of void * pointers on the heap
 *
 * returns pointers to duplicate array or NULL if malloc failure
 */
static void **arrayDupl(StData *std) {
    void **tmp = NULL;

    if (std->next > 0L) {
        size_t nbytes = std->next * sizeof(void *);
        tmp = (void **)malloc(nbytes);
        if (tmp != NULL) {
            long i;

            for (i = 0L; i < std->next; i++)
                tmp[i] = std->theArray[i];
        }
    }
    return tmp;
}

static void **st_toArray(const Stack *st, long *len) {
    StData *std = (StData *)st->self;
    void **tmp = arrayDupl(std);

    if (tmp != NULL)
        *len = std->next;
    return tmp;
}

static const Iterator *st_itCreate(const Stack *st) {
    StData *std = (StData *)st->self;
    const Iterator *it = NULL;
    void **tmp = arrayDupl(std);

    if (tmp != NULL) {
        it = Iterator_create(std->next, tmp);
        if (it == NULL)
            free(tmp);
    }
    return it;
}

static Stack template = {
    NULL, st_destroy, st_clear, st_push, st_pop, st_peek, st_size,
    st_isEmpty, st_toArray, st_itCreate
};

const Stack *Stack_create(long capacity) {
    Stack *st = (Stack *)malloc(sizeof(Stack));

    if (st != NULL) {
        StData *std = (StData *)malloc(sizeof(StData));

        if (std != NULL) {
            long cap;
            void **array = NULL;

            cap = (capacity <= 0L) ? DEFAULT_CAPACITY : capacity;
            array = (void **)malloc(cap * sizeof(void *));
            if (array != NULL) {
                std->capacity = cap;
                std->increment = cap;
                std->next = 0L;
                std->theArray = array;
                *st = template;
                st->self = std;
            } else {
                free(std);
                free(st);
                st = NULL;
            }
        } else {
            free(st);
            st = NULL;
        }
    }
    return st;
}
```

Chapter 9

Queues

A *queue* is an object container that conforms to the *first in, first out* discipline: there is a notion of the head and tail of the queue, elements may be added to the tail of the queue (*enqueue*), and removed from the head of the queue (*dequeue*). Once a new element is added to the queue, all elements that were added before it must be removed before the new element is removed. There is also usually a *front* method, returning the element at the head of the queue without dequeuing it.

Why call it a queue? Whenever there is a fixed number of points of service, and more requests needing service than there are points of service, one will typically queue. Examples abound in every day life: at the DMV, at the bakery, buying tickets to see a movie, ... Individuals line up waiting their turn to be served, in the order in which they entered the queue. A queue data structure maintains this arrival sequence, by forcing additions at the tail of the queue, and removals from the head of the queue.

A queue can have bounded capacity; if a `enqueue()` method is invoked when the queue is full, an exception is raised; as with a stack, an exception is raised if a `dequeue()` or a `front()` method is invoked on an empty queue, as well. It is often more useful to the programmer if the queue has unbounded capacity; of course, it's not truly unbounded, simply an attempt is made to grow the queue when is it full, much as we did with our Stack implementation.

In this section, we will show how to create a queue using an array. It is also common to implement a queue using a linked list, which we will explore in a later chapter.

9.1 A queue of long integers

As we did with Stacks, first let's look at how one would define and implement a queue of long integers. We will first implement it as a bounded queue, then show the changes required to make it unbounded.

How are we going to use an array to implement our queue? We could structure our `dequeue()` method so that we always return the element at index 0; if we did so, after removing the value at index 0, we would have to copy each of the other elements down one position. As we saw in the section on ArrayLists, such a method will have $O(n)$ running time complexity. Since a Queue has localized addition and removal (as did a Stack), we would like an implementation that exhibited $O(1)$ complexity on removal. How might we do this?

We can avoid moving the other elements of the queue if we keep track of two indices into the array: `in`, which is the next index into which an element can be stored (therefore, is the "tail" of the queue); and `out`, which is the next index from which an element can be retrieved (therefore, is the "head" of the queue). Obviously, whenever we `enqueue()` a new element, we need to increment `in`. What happens if we reach the end of the array?

If we treat the array as a circular buffer, then after storing a new element at the last index in the array, instead of simply incrementing `in`, we increment *modulo* the size of the array. That is, if `N` is the size of the array, when we invoke `enqueue()`, we increment `in` as

```
in = (in + 1) % N;
```

We would do a similar increment for `out` after a successful `dequeue()`. We also have to keep track of the number of elements in the queue to prevent one index from overtaking the other. We shall see this in the implementation.

9.1.1 The interface specification

```
#ifndef _QUEUELONG_H_
#define _QUEUELONG_H_

typedef struct queuelong QueueLong;        /* forward reference */
const QueueLong *QueueLong_create(long capacity);
struct queuelong {
    void *self;
    void (*destroy)(const QueueLong *ql);
    void (*clear)(const QueueLong *ql);
    int (*enqueue)(const QueueLong *ql, long element);
    int (*front)(const QueueLong *ql, long *element);
    int (*dequeue)(const QueueLong *ql, long *element);
    long (*size)(const QueueLong *ql);
    int (*isEmpty)(const QueueLong *ql);
};

#endif /* _QUEUELONG_H_ */
```

The semantics of the constructor and the methods are as follows:

- `const QueueLong *QueueLong_create(long capacity);`

create a new instance of a Queue of long integers that can hold `capacity` elements; returns a pointer to the dispatch table, or `NULL` if there were heap-allocation errors;

- `void (*destroy)(const QueueLong *ql);`
  destroy the Queue of long integers; all heap-allocated memory is returned;
- `void (*clear)(const QueueLong *ql);`
  purge all elements from the Queue; upon return, the queue is empty;
- `int (*enqueue)(const QueueLong *ql, long element);`
  add `element` to the tail of the queue; if the queue cannot accommodate the new element (it is full), the method returns 0; otherwise, it returns 1;
- `int (*front)(const QueueLong *ql, long *element);`
  return the element at the head of the queue in `*element` *without* removing it from the queue; if the queue has no elements, the method returns 0; otherwise it returns 1;
- `int (*dequeue)(const QueueLong *ql, long *element);`
  remove the element at the head of the queue, returning it in `*element`; if the queue has no elements, the method returns 0; otherwise, it returns 1;
- `long (*size)(const QueueLong *ql);`
  return the number of elements in the queue; and
- `long (*isEmpty)(const QueueLong *ql);`
  return 1 if the stack is empty, 0 otherwise.

9.1.2 The implementation

9.1.2.1 The preliminaries

As with the Stack implementation, the secret to an efficient and successful implementation is the choice of the instance-specific data structure.

```
#include "queuelong.h"
#include <stdlib.h>

typedef struct ql_data {
    long count;      /* number of elements in Q */
    long size;       /* size of array for elements */
    int in;          /* array index for next enqueue */
    int out;         /* array index for next dequeue or front */
    long *buffer;    /* array holding queue elements */
} QlData;
```

`count` is the number of elements in the queue, while `size` is the size of the array holding the queue; if `count == size`, then the queue is full (this is a bounded queue); if `count == 0`, the queue is empty. As we discussed earlier, `in` is the index in which the next `enqueue()` will store the element, and `out` is the index from which the next `dequeue()` or `front()` will retrieve an element.

9.1.2.2 The constructor

This looks very similar to the constructor for StackLong. Allocate a dispatch table from the heap, allocate an instance-specific data structure from the heap, allocate an array of longs from the heap. If all allocations are successful, initialize the instance-specific data structure and dispatch table, and return to the caller. If any heap-allocation failures, free up any heap-allocations prior to the failure and return `NULL`.

```
static QueueLong template = {
    NULL, ql_destroy, ql_clear, ql_enqueue, ql_front, ql_dequeue, ql_size,
    ql_isEmpty
};

const QueueLong *QueueLong_create(long capacity) {
    QueueLong *ql = (QueueLong *)malloc(sizeof(QueueLong));

    if (ql != NULL) {
        QlData *qld = (QlData *)malloc(sizeof(QlData));

        if (qld != NULL) {
            long *tmp;

            tmp = (long *)malloc(capacity * sizeof(long));
            if (tmp != NULL) {
                qld->count = 0;
                qld->size = capacity;
                qld->in = 0;
                qld->out = 0;
                qld->buffer = tmp;
                *ql = template;
                ql->self = qld;
            } else {
                free(qld);
                free(ql);
                ql = NULL;
            }
        } else {
            free(ql);
            ql = NULL;
        }
    }
    return ql;
}
```

Note that we initialize `in` and `out` to `0`. This can certainly not be a legal situation, or can it? If the queue is full, or the queue is empty, these two indices can be equal (you should work this through yourself). Thus, as we shall see below, `enqueue()`, `dequeue()`, and `front()` must check for full/empty *before* manipulating the indices.

9.1.2.3 `destroy()` and `clear()`

`destroy()` frees the heap-allocated memory in reverse order to the constructor. `clear()` simply resets `in` and `out` to `0`, indicating that the queue is empty.

```
static void ql_destroy(const QueueLong *ql) {
    QlData *qld = (QlData *)ql->self;

    free(qld->buffer);
    free(qld);
    free((void *)ql);
}

static void ql_clear(const QueueLong *ql) {
    QlData *qld = (QlData *)ql->self;

    qld->count = 0;
    qld->in = 0;
    qld->out = 0;
}
```

9.1.2.4 `enqueue()`

`enqueue()` first checks to see if `count < size`; if not, the queue is full, and it returns `0`, indicating a failure to queue `element`. Otherwise, it stores `element` in `buffer[in]`, increments `in` modulo `size`, increments `count`, and returns 1, indicating that `element` was successfully queued.

```
static int ql_enqueue(const QueueLong *ql, long element) {
    QlData *qld = (QlData *)ql->self;
    int status = (qld->count < qld->size);
    int i;

    if (status) {
        i = qld->in;
        qld->buffer[i] = element;
        qld->in = (i + 1) % qld->size;
        qld->count++;
    }
    return status;
}
```

9.1.2.5 `front()` and `dequeue()`

The logic for these two methods is identical, except that `front()` does not increment `out` or decrement `count`. Thus, we have defined a helper function, `retrieve()`, that retrieves the element at the head of the queue (if there is one), and then conditionally manipulates `out` and `count`. The method implementations then simply become appropriate invocations of `retrieve()`.

```
/* helper function used by dequeue() and front() */
static int retrieve(QlData *qld, long *element, int ifRemove) {
    int i;

    if (qld->count <= 0)
        return 0;
    i = qld->out;
    *element = qld->buffer[i];
    if (ifRemove) {
        qld->out = (i + 1) % qld->size;
        qld->count--;
    }
    return 1;
}

static int ql_front(const QueueLong *ql, long *element) {
    QlData *qld = (QlData *)ql->self;
    return retrieve(qld, element, 0);
}

static int ql_dequeue(const QueueLong *ql, long *element) {
    QlData *qld = (QlData *)ql->self;
    return retrieve(qld, element, 1);
}
```

9.1.2.6 size() and isEmpty()

These methods perform the obvious manipulations of `count`, returning the result to the caller.

```
static long ql_size(const QueueLong *ql) {
    QlData *qld = (QlData *)ql->self;
    return qld->count;
}

static int ql_isEmpty(const QueueLong *ql) {
    QlData *qld = (QlData *)ql->self;
    return (qld->count == 0L);
}
```

9.1.3 Complexity of the constructor and methods

As we did with Stacks, let's estimate the complexity of the constructor and each of the methods.

- `QueueLong_create()` - when this function is called, there are no elements in the Queue; we perform three calls to `malloc()`, which as we have discussed in the ArrayList section, we consider $O(1)$ for the purposes of this discussion. Thus, the complexity for the constructor is $O(1)$.
- `enqueue()` - this method simply checks to see if `count < size`, and if true, stores the element in the array at `in`, increments `in` modulo `size`, and increments `count`. This is independent of the number of elements in the queue, thus the complexity for `enqueue()` is $O(1)$.
- `dequeue()` - this method checks to see if `count > 0`, and if true, retrieves the element in the array at `out`, increments `out` modulo `size`, and decrements `count`. This is independent of the number of elements in the queue, thus the complexity for `dequeue()` is $O(1)$.
- `front()` - this method checks to see if `count > 0`, and if true, retrieves the element in the array at `out`. This is independent of the number of elements in the queue, thus the complexity for `front()` is $O(1)$.
- `destroy()` - when this function is called, we perform three calls to `free()`, which as we have discussed in the ArrayList section, we consider $O(1)$ for the purposes of this discussion. Thus, the complexity for `destroy()` is $O(1)$.
- `clear()` - this function simply sets `count`, `in`, and `out` to 0. This is independent of the number of elements currently in the queue, thus the complexity for `clear()` is $O(1)$.
- `size()` - this function simply returns the value of `count`. This is independent of the number of elements currently in the queue, thus the complexity for `size()` is $O(1)$.
- `isEmpty()` - this function simply returns the value of `count == 0L`. This is independent of the number of elements currently in the queue, thus the complexity for `size()` is $O(1)$.

From this discussion, we can see that for a bounded Queue of long integers, the constructor and *all* of the methods are $O(1)$. Thus, if your problem needs a container exhibiting *first in, first out* functionality, you cannot do better than a queue that is implemented using an array.

9.1.4 Changes needed to make the queue unbounded

9.1.4.1 The interface

No change is required to the signatures for the constructor or the methods, although the semantics for the value of `capacity` in the constructor is slightly different. If `capacity == 0L`, this tells the implementation to use a default queue size for the initial array. This default capacity should be `#define`'d in the header file, as in

```
#define DEFAULT_CAPACITY 50
```

9.1.4.2 The instance-specific data structure

The array of long integers will need to be made larger if `enqueue()` is invoked when the current queue is already full. The implementation can choose to simply double the size of the array whenever this occurs - this leads to exponential growth of the array. Alternatively, the implementation can choose to grow the array by a constant increment each time, leading to linear growth of the array. If linear growth is chosen, the increment can be a defined constant for all queues, or it can be a queue-specific value, usually dependent upon the initial capacity specified in the constructor.

If a queue-specific linear growth increment is to be used, then an additional member must be added to `QlData`, typically called `increment`. The data structure will look as follows:

```
typedef struct ql_data {
    long count;
    long size;
    long increment;
    long in;
    long out;
    long *buffer;
} QlData;
```

The changes below will assume queue-specific linear growth using this structure for the instance-specific data.

9.1.4.3 `destroy()`, `clear()`, `dequeue()`, `front()`, `size()`, and `isEmpty()`

No change required, since none of these methods can cause the array of longs to have to grow.

9.1.4.4 The constructor

After successful allocation of the instance-specific struct, we need to 1) determine the initial capacity of the queue, and 2) place a value in the `increment` member of that struct. The following code replaces the lines from “`if(qld != NULL) {`” up to the first line consisting of “`} else {`”.

```
if (qld != NULL) {
    long cap;
    long *tmp = NULL;
```

```
        cap = (capacity <= 0L) ? DEFAULT_CAPACITY : capacity;
        tmp = (long *)malloc(cap * sizeof(long));
        if (tmp != NULL) {
            qld->count = 0;
            qld->size = cap;
            qld->increment = cap;
            qld->in = 0;
            qld->out = 0;
            qld->buffer = tmp;
            *ql = template;
            ql->self = qld;
        } else {
```

9.1.4.5 enqueue()

The implementation for `enqueue()` shown above does nothing if `count >= size`. The following code will grow the array when it is full, and should simply be inserted above the "`if (status)`" line in the previous implementation.

```
    if (! status) {          /* need to reallocate */
        size_t nbytes = (qld->size + qld->increment) * sizeof(long);
        long *tmp = (long *)realloc(qld->buffer, nbytes);

        if (tmp != NULL) {
            long n = qld->count, i, j;
            status = 1;
            for (i = qld->out, j = 0; n > 0; i = (i + 1) % qld->size, j++) {
                tmp[j] = qld->buffer[i];
                n--;
            }
            free(qld->buffer);
            qld->buffer = tmp;
            qld->size += qld->increment;
            qld->out = 0L;
            qld->in = qld->count;
        }
    }
```

Note that on the occasion when an `enqueue()` invocation causes the queue to be expanded, the running time complexity of that call will be $O(n)$ - why is this so? Since we are using the array as a circular buffer, we may have wrapped around to the beginning of the array; when we grow the array, we need to move the queue of items to the beginning of the larger array in order to continue to use it in this circular fashion.

This is an example where the running time complexity of a method is different for the average case ($O(1)$ for `enqueue()` when not growing the array) and for the worst case ($O(n)$ when we have to grow the array). As long as we do not have to grow the array very often, `enqueue()` can be considered as $O(1)$. After we have introduced linked lists in Chapter 10, we will show an implementation of Queues in which the worst case running time complexity for `enqueue()` is $O(1)$, as well.

9.2 A generic queue

The following show the header and implementation files for an unbounded queue in which `void *` elements are queued. We have also included the `toArray()` and `itCreate()` methods, along with the `freeFxn()` 2nd arguments for `destroy()` and `clear()`. The path from QueueLong to Queue should now be understandable, so no additional discussion will be provided.

9.2.0.1 Interface for a generic queue

```
#ifndef _QUEUE_H_
#define _QUEUE_H_

/* BSD header removed to conserve space */

/*
 * interface definition for generic FIFO queue
 *
 * patterned roughly after Java 6 Queue interface
 */

#include "iterator.h"              /* needed for factory method */

#define DEFAULT_CAPACITY 50L

typedef struct queue Queue;        /* forward reference */

/*
 * create a queue; if capacity is 0L, give it a default capacity (50L)
 *
 * returns a pointer to the queue, or NULL if there are malloc() errors
 */
const Queue *Queue_create(long capacity);

/*
 * now define struct queue
 */
struct queue {
/*
 * the private data of the queue
 */
    void *self;

/*
 * destroys the queue; for each element, if freeFxn != NULL,
 * invokes freeFxn on the element; then returns any queue structure
 * associated with the element; finally, deletes any remaining structures
 * associated with the queue
 */
    void (*destroy)(const Queue *q, void (*freeFxn)(void *element));

/*
 * clears the queue; for each element, if freeFxn != NULL, invokes
 * freeFxn on the element; then returns any queue structure associated with
 * the element
 *
 * upon return, the queue is empty
 */
    void (*clear)(const Queue *q, void (*freeFxn)(void *element));

/*
 * appends `element' to the end of the queue
 *
 * returns 1 if successful, 0 if unsuccesful (queue is full)
 */
    int (*enqueue)(const Queue *q, void *element);

/*
 * retrieves, but does not remove, the head of the queue, returning that
 * element in `*element'
 *
 * returns 1 if successful, 0 if unsuccessful (queue is empty)
 */
    int (*front)(const Queue *q, void **element);

/*
 * Retrieves, and removes, the head of the queue, returning that
 * element in `*element'
 *
 * return 1 if successful, 0 if not (queue is empty)
 */
    int (*dequeue)(const Queue *q, void **element);

/*
 * returns the number of elements in the queue
 */
    long (*size)(const Queue *q);

/*
 * returns true if the queue is empty, false if not
 */
    int (*isEmpty)(const Queue *q);
```

```
/*
 * returns an array containing all of the elements of the queue in
 * proper sequence (from first to last element); returns the length of the
 * queue in `*len'
 *
 * returns pointer to void * array of elements, or NULL if malloc failure
 *
 * NB - it is the caller's responsibility to free the void * array when
 *      finished with it
 */
    void **(*toArray)(const Queue *q, long *len);

/*
 * creates an iterator for running through the queue
 *
 * returns pointer to the Iterator or NULL
 */
    const Iterator *(*itCreate)(const Queue *q);
};

#endif /* _QUEUE_H_ */
```

9.2.0.2 Implementation for a generic queue

```
/* BSD header removed to conserve space */

/*
 * implementation for generic FIFO queue
 */

#include "queue.h"
#include <stdlib.h>

typedef struct q_data {
    long count;
    long size;
    long increment;
    int in;
    int out;
    void **buffer;
} QData;

static void purge(QData *qd, void (*freeFxn)(void *element)) {
    if (freeFxn != NULL) {
        int i, n;

        for (i = qd->out, n = qd->count; n > 0; i = (i + 1) % qd->size, n--)
            (*freeFxn)(qd->buffer[i]);
    }
}
```

```
static void q_destroy(const Queue *q, void (*freeFxn)(void *element)) {
    QData *qd = (QData *)q->self;
    purge(qd, freeFxn);
    free(qd->buffer);
    free(qd);
    free((void *)q);
}

static void q_clear(const Queue *q, void (*freeFxn)(void *element)) {
    QData *qd = (QData *)q->self;
    int i;

    purge(qd, freeFxn);
    for (i = 0; i < qd->size; i++)
        qd->buffer[i] = NULL;
    qd->count = 0;
    qd->in = 0;
    qd->out = 0;
}

static int q_enqueue(const Queue *q, void *element) {
    QData *qd = (QData *)q->self;
    int status = (qd->count < qd->size);

    if (! status) {          /* need to reallocate */
        size_t nbytes = (qd->size + qd->increment) * sizeof(void *);
        void **tmp = (void **)malloc(nbytes);

        if (tmp != NULL) {
            long n = qd->count, i, j;
            status = 1;

            for (i = qd->out, j = 0; n > 0; i = (i + 1) % qd->size, j++) {
                tmp[j] = qd->buffer[i];
                n--;
            }
            free(qd->buffer);
            qd->buffer = tmp;
            qd->size += qd->increment;
            qd->out = 0L;
            qd->in = qd->count;
        }
    }
    if (status) {
        int i = qd->in;
        qd->buffer[i] = element;
        qd->in = (i + 1) % qd->size;
        qd->count++;
    }
    return status;
}
```

```
static int retrieve(QData *qd, void **element, int ifRemove) {
    int i;

    if (qd->count <= 0)
        return 0;
    i = qd->out;
    *element = qd->buffer[i];
    if (ifRemove) {
        qd->out = (i + 1) % qd->size;
        qd->count--;
    }
    return 1;
}

static int q_front(const Queue *q, void **element) {
    QData *qd = (QData *)q->self;
    return retrieve(qd, element, 0);
}

static int q_dequeue(const Queue *q, void **element) {
    QData *qd = (QData *)q->self;
    return retrieve(qd, element, 1);
}

static long q_size(const Queue *q) {
    QData *qd = (QData *)q->self;
    return qd->count;
}

static int q_isEmpty(const Queue *q) {
    QData *qd = (QData *)q->self;
    return (qd->count == 0L);
}

static void **toArray(QData *qd) {
    void **tmp = NULL;

    if (qd->count > 0L) {
        tmp = (void **)malloc(qd->count * sizeof(void *));
        if (tmp != NULL) {
            int i, j, n;

            n = qd->count;
            for (i = qd->out, j = 0; n > 0; i = (i+1) % qd->size, j++, n--) {
                tmp[j] = qd->buffer[i];
            }
        }
    }
    return tmp;
}
```

```
static void **q_toArray(const Queue *q, long *len) {
    QData *qd = (QData *)q->self;
    void **tmp = toArray(qd);

    if (tmp != NULL)
        *len = qd->count;
    return tmp;
}

static const Iterator *q_itCreate(const Queue *q) {
    QData *qd = (QData *)q->self;
    const Iterator *it = NULL;
    void **tmp = toArray(qd);

    if (tmp != NULL) {
        it = Iterator_create(qd->count, tmp);
        if (it == NULL)
            free(tmp);
    }
    return it;
}

static Queue template = {
    NULL, q_destroy, q_clear, q_enqueue, q_front, q_dequeue, q_size,
    q_isEmpty, q_toArray, q_itCreate
};

const Queue *Queue_create(long capacity) {
    Queue *q = (Queue *)malloc(sizeof(Queue));

    if (q != NULL) {
        QData *qd = (QData *)malloc(sizeof(QData));

        if (qd != NULL) {
            long cap;
            void **tmp;

            cap = (capacity <= 0L) ? DEFAULT_CAPACITY : capacity;
            tmp = (void **)malloc(cap * sizeof(void *));
            if (tmp != NULL) {
                qd->count = 0;
                qd->size = cap;
                qd->increment = cap;
                qd->in = 0;
                qd->out = 0;
                qd->buffer = tmp;
                *q = template;
                q->self = qd;
            } else {
                free(qd);
                free(q);
                q = NULL;
            }
        } else {
            free(q);
            q = NULL;
        }
    }
    return q;
}
```

Chapter 10

Linked lists

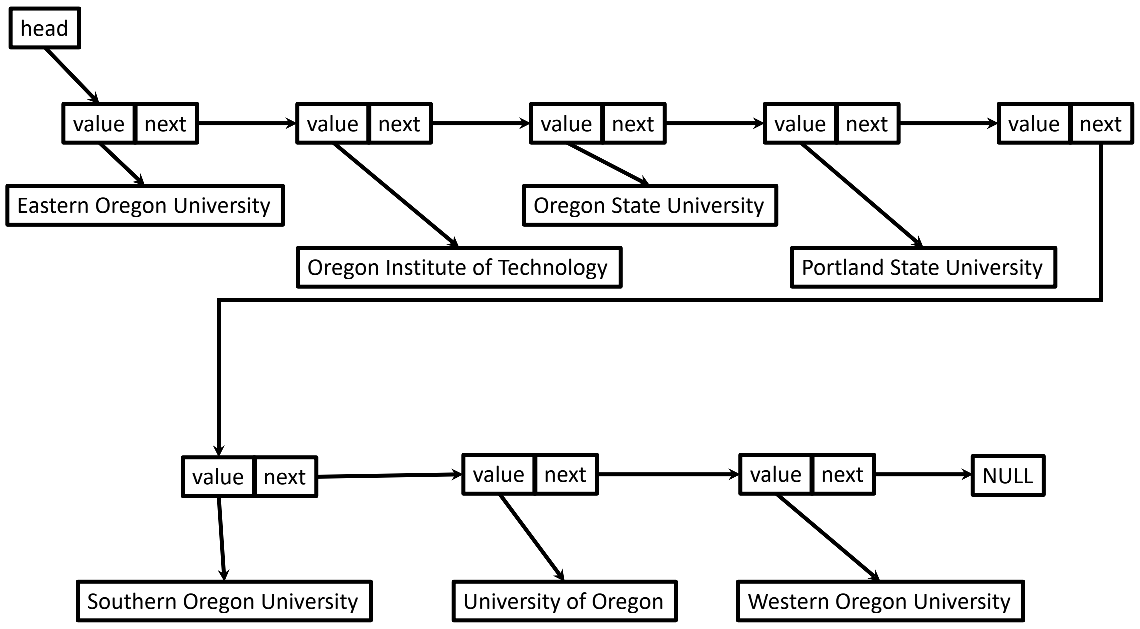
A *linked list* is linear collection of data elements, in which the linear order is *not* given by their physical placement in memory; instead, each *node* in the list points at the next element in the list. Such a structure enables insertion or removal of elements *without* reallocation or reorganization of the entire structure, since the data items do not need to be stored contiguously in memory.

It is important to note that while a linked list is an implementation of a sequence, it does not enable random access to the data, unlike an array.

10.1 General structure

Each node in a linked list consists of a pointer to the next node in the list (or `NULL` if the current node is the last one in the list) and a the value that the node contains. The value might be a a basic numeric value, such as 42, or more often, is a pointer to the value (for example, a pointer to a character string or a structure). We also require a pointer to the first element in the list, commonly called the *listhead*.

For example, suppose we wanted to have a linked list of the seven public universities in Oregon. A picture of this list is shown below.


head
value
next
Eastern Oregon University
Oregon Institute of Technology
Oregon State University
Portland State University
Southern Oregon University
University of Oregon
Western Oregon University
NULL

This is an example of a *singly linked* list, since there are only links in the forward direction along the list. Later in this chapter we will discuss doubly-linked lists.

For this linked list, each *node* in the list is a struct with the following members:

```
struct node {
    struct node *next;
    char *value;
};
```

and the variable `head` has the following declaration:

```
struct node *head;
```

The strings for the universities have been allocated on the heap.

What are some things that we can do with such a list? Suppose that we want to print out the names of the universities, one per line. The following code would do the trick:

```
struct node *p;

for (p = head; p != NULL; p = p->next)
    printf("%s\n", p->value);
```

Now suppose that we wish to search the linked list to see if a particular university is contained in the list. Writing a function to search such a list is straightforward:

```
int search(char *university, struct node *theList) {
    struct node *p;

    for (p = theList; p != NULL; p = p->next)
        if (strcmp(p->value, university) == 0)
            return 1;
    return 0;
}
```

Now let's search the list for "Medford College".

```
if (! search("Medford College", head))
    fprintf(stderr, "Medford College not found in list of Universities\n");
```

10.1.1 Additions/deletions to singly linked lists

Insertion or deletion at the head of a linked list can be done in $O(1)$ running time complexity. Suppose that we wish to add `"Oregon Health Science University"` to the front of the list. First we would have to create a new node using `malloc()`. Then we would have to make a copy of the string on the heap, using `strdup()`. Then we have the new node point to the current node at the head of the list, and then have the `head` variable point at this new node.

```
struct node *new = (struct node *)malloc(sizeof(struct node));
new->value = strdup("Oregon Health Science University");
new->next = head;
head = new;
```

We can see that the amount of work that we have to do is independent of the number of nodes currently in the list, thus $O(1)$. Note that this works correctly even if `head == NULL`.

Deletion of the node at the head of the list is similarly straightforward:

```
struct node *p = head;  /* p points to head node */
if (p != NULL) {        /* check that list was not empty */
    head = p->next;     /* head now points to 2nd node */
    free(p->value);     /* free the string */
    free(p);            /* free the node */
}
```

Here we have to explicitly check that the list was not empty; otherwise, we would have attempted to dereference a `NULL` pointer, which would cause our process to terminate with a segmentation violation.

How about additions at the end of the list? Absent any other metadata about the list, our only choice would be to write a loop to "chase pointers" until we found the last node, and then append a new node to form the new tail of the list. The code would be similar to the following:

```
struct node *new = (struct node *)malloc(sizeof(struct node));
new->value = strdup("Oregon Health Science University");
new->next = NULL;
struct node *p = head;

if (p != NULL) {
    while (p->next != NULL)
        p = p->next;
}
if (p != NULL)
    p->next = new;
else
    head = new;
```

This is significantly more complex than addition at the head. We still have to allocate a new node and the string for its value; note that we initialize the `next` field to `NULL`, since that is the required value in that field for the node at the tail of the list. We then have to find the current tail node, which is the next 4 lines of code; note that we have to conditionalize entering the while loop on the list being non-empty. Finally, we have to take different actions depending upon the loop being empty before the addition or not.

The while loop to find the tail node has to touch all of the nodes in the list, thus has an $O(n)$ running time complexity. We can reduce this complexity for additions at the tail by keeping one additional pointer for the list, a pointer to the tail node. Suppose we have a non-empty list, and that `head` points at the head node, and that `tail` points at the tail node. The logic required to add a new node at the tail becomes

```
struct node *new = (struct node *)malloc(sizeof(struct node));
new->value = strdup("Oregon Health Science University");
new->next = NULL;
tail->next = new;  /* previous tail node now points to new tail */
tail = new;        /* tail points to new tail */
```

From our experience to date, since this code does not depend at all on the number of nodes in the list, we can conclude that the running time complexity is $O(1)$.

The previous code assumed that the list was non-empty. If the list was empty, both `head` and `tail` would be `NULL`, and our code for additions at the tail becomes a little more complex to handle the empty list case.

```
struct node *new = (struct node *)malloc(sizeof(struct node));
new->value = strdup("Oregon Health Science University");
new->next = NULL;
if (head == NULL)  /* empty list */
    head = new;
else
    tail->next = new;
tail = new;        /* tail points to new tail */
```

Deletions of nodes anywhere in the list but at the head are $O(n)$ complexity. This is because we need to have a pointer to the node that precedes the node to be deleted. Since the list is singly linked, this means that we must chase pointers through the list, remembering the previous node as well as the current node.

10.2 Alternative implementations for Stacks and Queues

10.2.1 Stack implementation using a singly linked list

As we saw earlier in this chapter, Stacks follow a last in, first out policy. We implemented a stack using an array, with an integer variable that always represented the next free array index.

We could also implement the Stack interface using a singly linked list, where a `push()` would generate a node addition at the head, and a `pop()` would generate a node removal at the head; obviously, a `peek()` simply returns the value of the head node, without removing the node from the linked list.

Here is a revised implementation of the Stack interface using a singly linked list.

10.2.1.1 The preliminaries

Since we have switched to a linked list for the stack implementation, we need to define the structure of a linked list node - `struct node` or its typedef, `Node`. The instance-specific data has become simpler, since all we need to keep track of is the number of nodes in the list, `size`, and a pointer to the head of the list, `head`.

```
/* BSD header removed to save space */

/*
 * linked list implementation for generic stack
 */
```

```
#include "stack.h"
#include <stdlib.h>

typedef struct node {      /* node in linked list */
    struct node *next;
    void *value;
} Node;

typedef struct st_data {
    long size;    /* number of nodes in list */
    Node *head;   /* head of list */
} StData;
```

10.2.1.2 destroy() and clear()

The `purge()` function that applies the user-supplied free function to elements in the stack had to be rewritten to navigate the linked list, instead of indexing through the array. We have also added another local function, `freeList()`, which frees the nodes in the list. Note the body of the while loop in `freeList()`, where we save the next node in another pointer variable, then free the current node, then set that pointer to the saved value. This is because you should *never* attempt to access heap memory after it has been freed. Finally, both `destroy()` and `clear()` have called `freeList()` as part of their operation to return the nodes to the heap.

```
/*
 * local function - traverses stack, applying user-supplied function
 * to each element; if freeFxn is NULL, nothing is done
 */

static void purge(StData *std, void (*freeFxn)(void*)) {
    if (freeFxn != NULL) {
        Node *p;

        for (p = std->head; p != NULL; p = p->next)
            (*freeFxn)(p->value);       /* user frees elem storage */
    }
}

/*
 * local function - frees Node's associated with the stack; set head to NULL
 */
static void freeList(StData *std) {
    Node *p = std->head;

    while (p != NULL) {
        Node *q = p->next;
        free(p);
        p = q;
    }
```

```
        std->head = NULL;
    }

    static void st_destroy(const Stack *st, void (*freeFxn)(void *element)) {
        StData *std = (StData *)st->self;

        purge(std, freeFxn);
        freeList(std);                  /* free linked list */
        free(std);                      /* free structure with instance data */
        free((void *)st);               /* free dispatch table */
    }

    static void st_clear(const Stack *st, void (*freeFxn)(void *element)) {
        StData *std = (StData *)st->self;

        purge(std, freeFxn);
        freeList(std);
        std->size = 0L;
    }
```

10.2.1.3 push(), pop(), and peek()

push() has to allocate a new node from the heap; if this fails, a value of 0 is returned to the user. Otherwise, the element is stored in the node, the node is linked in at the head of the linked list, and the size of the list is incremented.

pop() copies the value of the head node into *element, removes head node from the list, frees the node, and decrements the size of the list. peek() simply copies the value of the head node into *element.

```
    static int st_push(const Stack *st, void *element) {
        StData *std = (StData *)st->self;
        Node *new = (Node *)malloc(sizeof(Node));  /* need a new node */

        if (new == NULL)
            return 0;
        new->value = element;
        new->next = std->head;
        std->head = new;
        std->size++;
        return 1;
    }

    static int st_pop(const Stack *st, void **element) {
        StData *std = (StData *)st->self;
        int status = 0;

        if (std->size > 0L) {
            Node *p = std->head;
            *element = p->value;
            std->head = p->next;
            std->size--;
            free(p);                /* free the node */
            status = 1;
        }
        return status;
    }

    static int st_peek(const Stack *st, void **element) {
        StData *std = (StData *)st->self;
        int status = 0;

        if (std->size > 0L) {
            *element = std->head->value;
            status = 1;
        }
        return status;
    }
```

10.2.1.4 size() and isEmpty()

The logic here is identical to that used in the array implementation.

```
    static long st_size(const Stack *st) {
        StData *std = (StData *)st->self;

        return std->size;
    }

    static int st_isEmpty(const Stack *st) {
        StData *std = (StData *)st->self;

        return (std->size == 0L);
    }
```

10.2.1.5 toArray() and itCreate()

We have changed the local function from `arrayDupl()` to `genArray()`, since in this implementation we do not have an array to duplicate. Most of the logic in `genArray()` is the same, except for the code that copies the elements into the array; here we have to navigate the linked list, whereas the array-based implementation used indexing of the array. The logic for the methods is identical to the array-based implementation.

```
    /*
     * local function - generates array of void * pointers on the heap
     *
     * returns pointers to array or NULL if malloc failure
     */
    static void **genArray(StData *std) {
```

```
    void **tmp = NULL;

    if (std->size > 0L) {
        size_t nbytes = std->size * sizeof(void *);
        tmp = (void **)malloc(nbytes);
        if (tmp != NULL) {
            Node *p;
            long i = 0;

            for (p = std->head; p != NULL; p = p->next)
                tmp[i++] = p->value;
        }
    }
    return tmp;
}

static void **st_toArray(const Stack *st, long *len) {
    StData *std = (StData *)st->self;
    void **tmp = genArray(std);

    if (tmp != NULL)
        *len = std->size;
    return tmp;
}

static const Iterator *st_itCreate(const Stack *st) {
    StData *std = (StData *)st->self;
    const Iterator *it = NULL;
    void **tmp = genArray(std);

    if (tmp != NULL) {
        it = Iterator_create(std->size, tmp);
        if (it == NULL)
            free(tmp);
    }
    return it;
}
```

10.2.1.6 The constructor

This is simpler than the constructor for the array-based implementation. There is one less heap-allocation required, and the number of fields in the instance-specific data structure is smaller. Note that we have applied the *unused* attribute to the constructor argument, as `capacity` no longer has any meaning in this implementation. Flagging it as unused guarantees that `gcc` will not issue warnings to us when the code is compiled.

```
static Stack template = {
    NULL, st_destroy, st_clear, st_push, st_pop, st_peek, st_size,
    st_isEmpty, st_toArray, st_itCreate
};
```

```
#define UNUSED __attribute__((unused))
const Stack *Stack_create(UNUSED long capacity) {  /* ignore capacity */
    Stack *st = (Stack *)malloc(sizeof(Stack));

    if (st != NULL) {
        StData *std = (StData *)malloc(sizeof(StData));

        if (std != NULL) {
            std->size = 0L;
            std->head = NULL;
            *st = template;
            st->self = std;
        } else {
            free(st);
            st = NULL;
        }
    }
    return st;
}
```

10.2.2 Queue implementation using a singly linked list

Queues follow a first in, first out policy. We previously implemented a queue using an array, with two integer variables that represented the index into the array for the next `enqueue()` and the next `dequeue()` or `front()`.

We can also implement the Queue interface using a singly linked list, where an `enqueue()` generates a node addition at the tail, and a `dequeue()` generates node removal at the head; a `front()` simply returns the value of the head node, without removing the node from the linked list.

Here is a revised implementation of the Queue interface using a singly linked list.

10.2.2.1 The preliminaries

We have to define the structure for a linked list node, `struct node` or its typedef, `Node`. The instance-specific data structure is simpler than the array-based implementation, as all we need to keep track of are the head and tail of the list and the number of nodes in the list.

```
/* BSD Header removed to conserve space */

/*
 * linked list implementation for generic unbounded FIFO queue
 */

#include "queue.h"
```

```
#include <stdlib.h>

typedef struct node {
    struct node *next;
    void *value;
} Node;

typedef struct uq_data {
    long size;
    Node *head;
    Node *tail;
} UqData;
```

10.2.2.2 `destroy()` and `clear()`

`purge()` navigates the linked list of nodes instead of indexing into an array. We have also added `freeList()` to free the nodes associated with the queue. `destroy()` and `clear()` invoke `freeList()` after invoking `purge()`.

```
/*
 * local function - traverses queue, applying user-supplied function
 * to each element; if freeFxn is NULL, nothing is done
 */
static void purge(UqData *uqd, void (*freeFxn)(void*)) {
    if (freeFxn != NULL) {
        Node *p;

        for (p = uqd->head; p != NULL; p = p->next)
            (*freeFxn)(p->value);       /* user frees elem storage */
    }
}

/*
 * local function - frees Node's associated with the queue; sets head
 * and tail to NULL
 */
static void freeList(UqData *uqd) {
    Node *p = uqd->head;

    while (p != NULL) {
        Node *q = p->next;
        free(p);
        p = q;
    }
    uqd->head = NULL;
    uqd->tail = NULL;
}

static void uq_destroy(const Queue *uq, void (*freeFxn)(void *element)) {
    UqData *uqd = (UqData *)uq->self;
```

```
        purge(uqd, freeFxn);
        freeList(uqd);
        free(uqd);
        free((void *)uq);
    }

    static void uq_clear(const Queue *uq, void (*freeFxn)(void *element)) {
        UqData *uqd = (UqData *)uq->self;

        purge(uqd, freeFxn);
        freeList(uqd);
        uqd->size = 0L;
    }
```

10.2.2.3 `enqueue()`, `front()`, and `dequeue()`

`enqueue()` has to allocate a new node on the heap; if that fails, the function returns 0 to indicate failure to enqueue the element. If it is successful, `element` is copied into the node, and its `next` field is set to NULL. It is then placed at the tail of the queue. Note the five lines:

```
    if (uqd->size > 0L)
        uqd->tail->next = new;
    else
        uqd->head = new;
    uqd->tail = new;
```

These guarantee that the head and tail pointers are correct independent of the value of `size`. This pattern is a useful thing to memorize if you are going to create linked lists in your programs.

The code for `front()` and `dequeue()` look similar to the code for `peek()` and `pop()` for the linked list implementation of the stack. The two additional lines in `dequeue()`:

```
    if ((uqd->size--) == 0L)
        uqd->tail= NULL;
```

guarantee that the tail pointer is correct if we have dequeue'd the only element in a singleton list.

```
    static int uq_enqueue(const Queue *uq, void *element) {
        UqData *uqd = (UqData *)uq->self;
        Node *new = (Node *)malloc(sizeof(Node));

        if (new == NULL)
            return 0;
        new->value = element;
        new->next = NULL;
        if (uqd->size > 0L)
            uqd->tail->next = new;
        else
            uqd->head = new;
        uqd->tail = new;
        uqd->size++;
        return 1;
    }

    static int uq_front(const Queue *uq, void **element) {
        UqData *uqd = (UqData *)uq->self;
        int result = 0;

        if (uqd->size > 0L) {
            *element = uqd->head->value;
            result = 1;
        }
        return result;
    }

    static int uq_dequeue(const Queue *uq, void **element) {
        UqData *uqd = (UqData *)uq->self;
        int result = 0;

        if (uqd->size > 0L) {
            Node *p = uqd->head;
            *element = p->value;
            uqd->head = p->next;
            if ((uqd->size--) == 0L)
                uqd->tail = NULL;
            free(p);
            result = 1;
        }
        return result;
    }
```

Note that this implementation using a singly-linked list has a worst case running time complexity for `enqueue()` of $O(1)$, unlike the array-based implementation. As a result, unbounded queues are almost always implemented using a singly-linked list.

10.2.2.4 size() and isEmpty()

The logic here is identical to that used in the array implementation.

```
    static long uq_size(const Queue *uq) {
        UqData *uqd = (UqData *)uq->self;

        return uqd->size;
```

```
}

static int uq_isEmpty(const Queue *uq) {
    UqData *uqd = (UqData *)uq->self;

    return (uqd->size == 0L);
}
```

10.2.2.5 toArray() and itCreate()

`genArray()` instead of `arrayDupl()`, as for stacks. Most of logic is the same, except for having to navigate the linked list to populate the array. The logic for the methods is identical to the array-based implementation.

```
/*
 * local function - generates array of void * pointers on the heap
 *
 * returns pointer to array or NULL if malloc failure
 */
static void **genArray(UqData *uqd) {
    void **tmp = NULL;

    if (uqd->size > 0L) {
        size_t nbytes = uqd->size * sizeof(void *);
        tmp = (void **)malloc(nbytes);
        if (tmp != NULL) {
            Node *p;
            long i = 0;

            for (p = uqd->head; p != NULL; p = p->next)
                tmp[i++] = p->value;
        }
    }
    return tmp;
}

static void **uq_toArray(const Queue *uq, long *len) {
    UqData *uqd = (UqData *)uq->self;
    void **tmp = genArray(uqd);

    if (tmp != NULL)
        *len = uqd->size;
    return tmp;
}

static const Iterator *uq_itCreate(const Queue *uq) {
    UqData *uqd = (UqData *)uq->self;
    const Iterator *it = NULL;
    void **tmp = genArray(uqd);
    if (tmp != NULL) {
        it = Iterator_create(uqd->size, tmp);
        if (it == NULL)
            free(tmp);
    }
    return it;
}
```

10.2.2.6 The constructor

Again, this is simpler than the array-based implementation, since we do not have to allocate an array from the heap, and the number of fields in the instance-specific data structure is smaller. Again, we have applied the *unused* atribute to the constructor argument, as `capacity` no longer has any meaning in this implementation.

```
static Queue template = {
    NULL, uq_destroy, uq_clear, uq_enqueue, uq_front, uq_dequeue, uq_size,
    uq_isEmpty, uq_toArray, uq_itCreate
};

#define UNUSED __attribute__((unused))
const Queue *Queue_create(UNUSED long capacity) {
    Queue *uq = (Queue *)malloc(sizeof(Queue));

    if (uq != NULL) {
        UqData *uqd = (UqData *)malloc(sizeof(UqData));

        if (uqd != NULL) {
            uqd->size = 0L;
            uqd->head = NULL;
            uqd->tail = NULL;
            *uq = template;
            uq->self = uqd;
        } else {
            free(uq);
            uq = NULL;
        }
    }
    return uq;
}
```

Chapter 11

Deques - double-ended queues

The singly-linked list implementation of unbounded queues gives us $O(1)$ running time complexity for insertions at the head and tail, and removals at the head; if we wish to remove at the tail, the complexity is $O(n)$. This chapter is concerned with double-ended queues, or *deques* (pronounced "decks"). Deques support $O(1)$ complexity insertion and deletion at both the head and tail of the queue.

11.1 The interface for the Deque ADT

A deque supports insertion and removal from the head and tail of the queue. The interface of a Deque is shown below.

```
#ifndef _DEQUE_H_
#define _DEQUE_H_

#include "iterator.h"                  /* needed for factory method */

typedef struct deque Deque;            /* forward reference */

const Deque *Deque_create(void);

struct deque {
    void *self;
    void (*destroy)(const Deque *d, void (*freeFxn)(void *element));
    void (*clear)(const Deque *d, void (*freeFxn)(void *element));
    int (*insertFirst)(const Deque *d, void *element);
    int (*insertLast)(const Deque *d, void *element);
    int (*first)(const Deque *d, void **element);
    int (*last)(const Deque *d, void **element);
    int (*removeFirst)(const Deque *d, void **element);
    int (*removeLast)(const Deque *d, void **element);
    long (*size)(const Deque *d);
    int (*isEmpty)(const Deque *d);
    void **(*toArray)(const Deque *d, long *len);
    const Iterator *(*itCreate)(const Deque *d);
};

#endif /* _DEQUE_H_ */
```

The semantics of the constructor and the methods are as follows:

- `const Deque *Deque_create(void);`
  create a new instance of a deque; returns a pointer to the dispatch table, or `NULL` if there were heap-allocation errors;
- `void (*destroy)(const Deque *d, void (*freeFxn)(void *element));`
  destroy the deque; if `freeFxn != NULL`, this function is invoked on each element in the deque; then all heap-allocated memory associated with the deque is freed;
- `void (*clear)(const Deque *d, void(*freeFxn)(void * element));`
  purge all elements from the deque; if `freeFxn != NULL`, this function is invoked on each element in the deque; then heap-allocated memory associated with the the elements in the deque is freed; upon return, the deque is empty;
- `int (*insertFirst)(const Deque *d, void *element);`
  insert `element` at the head of the deque; if successful, returns 1; if not (heap-allocation failure), returns 0;
- `int (*insertLast)(const Deque *d, void *element);`
  insert `element` at the tail of the deque; if successful, returns 1; if not (heap-allocation failure), returns 0;
- `int (*first)(const Deque *d, void **element);`
  return the element at the head of the deque in `*element` *without* removing it from the deque; if the deque has no elements, the function return is 0; otherwise, it is 1;
- `int (*last)(const Deque *d, void **element);`
  return the element at the tail of the deque in `*element` *without* removing it from the deque; if the deque has no elements, the function return is 0; otherwise, it is 1;
- `int (*removeFirst)(const Deque *d, void **element);`
  remove the element at the head of the deque, returning it in `*element`; if the deque has no elements, the function return is 0; otherwise, it is 1;
- `int (*removeLast)(const Deque *d, void **element);`
  remove the element at the tail of the deque, returning it in `*element`; if the deque has no elements, the function return is 0; otherwise, it is 1;
- `long (*size)(const Deque *d);`
  return the number of elements in the deque;
- `int (*isEmpty)(const Deque *d);`
  return 1 if the deque is empty, 0 otherwise;

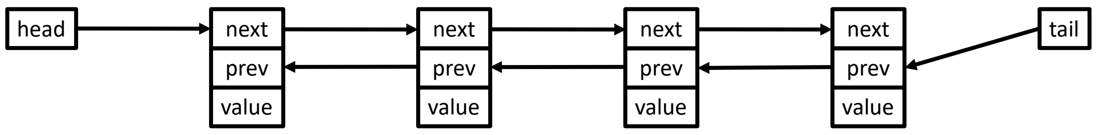
- `void **(*toArray)(const Deque *d, long *len);`
  returns an array containing all of the elements of the deque in proper sequence (from first to last element); returns the length of the array in `*len`; function return is pointer to the array of elements, or NULL if heap-allocation failure; the programmer must `free()` the array when it is no longer needed; and
- `const Iterator *(*itCreate)(const Deque *d);`
  creates an iterator for running through the elements in the deque; returns a pointer to the Iterator or NULL if there were heap-allocation problems.

11.2 The implementation for a generic Deque

We have already seen that when using a singly-linked list to implement a Queue that we need conditional code to handle the case when a removal leaves the Queue empty. Also, if you think about it, it will be difficult to provide an $O(1)$ method for removing the last element in the Queue; this is because we need to know the element just before the last element in order to leave the Queue in a known state.

We could implement the Deque using an array, and this is one of the programming exercises at the end of section. Here, we will discuss the most common way to implement a Deque, using a doubly linked list with a *sentinel*.

11.2.1 What is a doubly linked list?



This figure shows how a doubly-linked list is structured. We still require a pointer to the head of the list, and another pointer to the tail of the list. Each node in the list has two pointers, `next` points to the next node in the list, and `prev` points to the previous node in the list. The last node in the list will have `next == NULL`, and the first node in the list will have `prev == NULL`. In essence, we have two singly linked lists combined into the doubly linked list.

Recall that we already had $O(1)$ removal of the node at the head of a Queue, and with the `tail` pointer, we had $O(1)$ insertion at the tail of a Queue. We already knew from our Stack implementation that we had $O(1)$ insertion at the head of the list. Thus, we need to show that a doubly-lined list exhibits $O(1)$ removal of the node at the tail of a Deque.

The following code fragment shows how we would remove the tail node

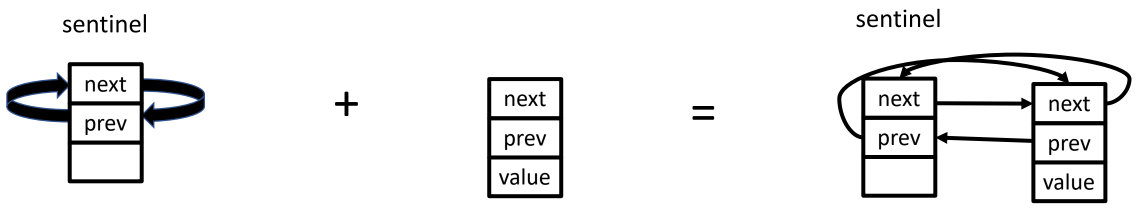
```
LLNode *p, *tail;

p = tail;                /* p now points to the tail node */
p->prev->next = p->next; /* p's predecessor now points to NULL */
tail = p->prev;          /* tail now points to p's predecessor */
```

None of these actions have anything to do with the number of nodes in the list, therefore this code fragment has $O(1)$ running time complexity.

This code fragment assumed that this was not a singleton doubly-linked list; this is implicit in assigning a value to `p->prev->next`; if there was only one node in the list, then `p->prev` would be NULL, and attempting to dereference it would lead to a segmentation violation, causing your program to abort. In general, the code to handle removing the last node in the doubly-linked list, or inserting into a singleton doubly-linked list, can be quite complex.

An elegant way to finesse this situation is to create a *sentinel* node. An *empty* doubly-linked list has a single node, the *sentinel*; the `next` field in the sentinel points at itself, the `prev` field points at itself, and its `value` field is not used. Why would this eliminate all of the conditional code?

Consider that we have an empty doubly-linked list and wish to add a node to it. The following figure shows graphically what will happen.



It may be a little easier to understand how we go from left hand side to the right hand side with some code:

```
typedef struct node {
    struct node *next;
    struct node *prev;
    void *value;
} Node;

Node sentinel = {&sentinel, &sentinel, NULL};
Node *p;     /* assume that p points to the node to be added */

p->next = &sentinel;
```

```
p->prev = &sentinel;
sentinel->prev = p;
sentinel->next = p;
```

This is just a special case of the following: on insertions, if we have pointers to the two adjacent nodes between which the insertion should occur, then the insertion action takes place as:

```
void link(Node *before, Node *p, Node *after) {
    p->next = after;
    p->prev = before;
    after->prev = p;
    before->next = p;
}
```

To insert at the head,

```
link(&sentinel, p, sentinel.next);
```

while to insert at the tail

```
link(sentinel.prev, p, &sentinel);
```

Removals are equally as straightforward. In general, given a pointer to a node in the doubly-linked list, we can remove it from the list as follows:

```
void unlink(Node *p) {
    p->prev->next = p->next;
    p->next->prev = p->prev;
}
```

To remove the tail node,

```
unlink(sentinel.prev);
```

while to remove the head node,

```
link(sentinel.next);
```

11.2.2 The Deque implementation using a doubly-linked list

11.2.2.1 The preliminaries

The preliminaries have to include the header file, define the node structure for the linked list, and the instance-specific data structure for the deque. The sentinel is contained in the instance-specific data structure; since the code will have many situations where it needs the *address* of the sentinel in that data structure, we define a macro, `SENTINEL`, which returns that address.

```
/* BSD Header removed to conserve space */

/*
 * implementation for generic deque using a linked list with a sentinel
 */

#include "deque.h"
#include <stdlib.h>

#define SENTINEL(p) (&(p)->sentinel)

typedef struct llnode {
    struct llnode *next;
    struct llnode *prev;
    void *element;
} LLNode;

typedef struct d_data {
    long size;
    LLNode sentinel;
} DData;
```

11.2.2.2 `destroy()` and `clear()`

We define two helper functions: `purge()`, which calls the user-supplied `freeFxn()` (if non-NULL) on each element; and `freeList()`, which frees each of the linked list nodes. The code for the two methods is similar to that seen for all of the other ADTs, except for the additional call to `freeList()`.

```
/*
 * traverses linked list, calling freeFxn on each element
 */
static void purge(DData *dd, void (*freeFxn)(void *element)) {
    if (freeFxn != NULL) {
        LLNode *p;

        for (p = dd->sentinel.next; p != SENTINEL(dd); p = p->next)
            (*freeFxn)(p->element);
    }
}

/*
 * frees the nodes from the doubly-linked list
 */
static void freeList(DData *dd) {
    LLNode *p = dd->sentinel.next;

    while (p != SENTINEL(dd)) {
        LLNode *q = p->next;
        free(p);
        p = q;
    }
}

static void d_destroy(const Deque *d, void (*freeFxn)(void *element)) {
    DData *dd = (DData *)d->self;
    purge(dd, freeFxn);
    freeList(dd);
    free(dd);
    free((void *)d);
}

static void d_clear(const Deque *d, void (*freeFxn)(void *element)) {
    DData *dd = (DData *)d->self;
    purge(dd, freeFxn);
    freeList(dd);
    dd->size = 0L;
    dd->sentinel.next = SENTINEL(dd);
    dd->sentinel.prev = SENTINEL(dd);
}
```

11.2.2.3 `insertFirst()` and `insertLast()`

We define the function `link()` which does the appropriate link modifications to insert a note between two other nodes (as discussed above). The code for the two methods is then a straightforward application of this function on the appropriate side of the sentinel.

```
/*
 * link `p' between `before' and `after'
 * must work correctly if `before' and `after' are the same node
 * (i.e. the sentinel)
 */
static void link(LLNode *before, LLNode *p, LLNode *after) {
    p->next = after;
    p->prev = before;
    after->prev = p;
    before->next = p;
}
```

```
static int d_insertFirst(const Deque *d, void *element) {
    DData *dd = (DData *)d->self;
    int status = 0;
    LLNode *p = (LLNode *)malloc(sizeof(LLNode));

    if (p != NULL) {
        p->element = element;
        status = 1;
        link(SENTINEL(dd), p, SENTINEL(dd)->next);
        dd->size++;
    }
    return status;
}

static int d_insertLast(const Deque *d, void *element) {
    DData *dd = (DData *)d->self;
    int status = 0;
    LLNode *p = (LLNode *)malloc(sizeof(LLNode));

    if (p != NULL) {
        p->element = element;
        status = 1;
        link(SENTINEL(dd)->prev, p, SENTINEL(dd));
        dd->size++;
    }
    return status;
}
```

11.2.2.4 `first()`, `last()`, `removeFirst()`, and `removeLast()`

The code for `first()` and `last()` are quite straightforward, simply returning the elements on the appropriate side of the sentinel. We define the `unlink()` function (as discussed above), and then use it in the `remove*()` methods on the element on the appropriate side of the sentinel.

```
static int d_first(const Deque *d, void **element) {
    DData *dd = (DData *)d->self;
    int status = 0;
    LLNode *p = SENTINEL(dd)->next;

    if (p != SENTINEL(dd)) {
        status = 1;
        *element = p->element;
    }
    return status;
}

static int d_last(const Deque *d, void **element) {
    DData *dd = (DData *)d->self;
    int status = 0;
    LLNode *p = SENTINEL(dd)->prev;

    if (p != SENTINEL(dd)) {
        status = 1;
        *element = p->element;
    }
    return status;
}

/*
 * unlinks the LLNode from the doubly-linked list
 */
static void unlink(LLNode *p) {
    p->prev->next = p->next;
    p->next->prev = p->prev;
}

static int d_removeFirst(const Deque *d, void **element) {
    DData *dd = (DData *)d->self;
    int status = 0;
    LLNode *p = SENTINEL(dd)->next;

    if (p != SENTINEL(dd)) {
        status = 1;
        *element = p->element;
        unlink(p);
        free(p);
        dd->size--;
    }
    return status;
}

static int d_removeLast(const Deque *d, void **element) {
    DData *dd = (DData *)d->self;
    int status = 0;
    LLNode *p = SENTINEL(dd)->prev;

    if (p != SENTINEL(dd)) {
        status = 1;
        *element = p->element;
        unlink(p);
        free(p);
        dd->size--;
    }
    return status;
}
```

11.2.2.5 `size()` and `isEmpty()`

The obvious logic, just as for all of the other ADTs.

```
static long d_size(const Deque *d) {
    DData *dd = (DData *)d->self;
    return dd->size;
}

static int d_isEmpty(const Deque *d) {
    DData *dd = (DData *)d->self;
    return (dd->size == 0L);
}
```

11.2.2.6 toArray() and itCreate()

Again, this code should be obvious given the other ADTs that you have seen.

```
/*
 * local function to generate array of element values on the heap
 *
 * returns pointer to array or NULL if malloc failure
 */
static void **genArray(DData *dd) {
    void **tmp = NULL;
    if (dd->size > 0L) {
        size_t nbytes = dd->size * sizeof(void *);
        tmp = (void **)malloc(nbytes);
        if (tmp != NULL) {
            long i;
            LLNode *p;
            for (i = 0, p = SENTINEL(dd)->next; i < dd->size; i++, p = p->next)
                tmp[i] = p->element;
        }
    }
    return tmp;
}

static void **d_toArray(const Deque *d, long *len) {
    DData *dd = (DData *)d->self;
    void **tmp = genArray(dd);

    if (tmp != NULL)
        *len = dd->size;
    return tmp;
}

static const Iterator *d_itCreate(const Deque *d) {
    DData *dd = (DData *)d->self;
    const Iterator *it = NULL;
    void **tmp = genArray(dd);

    if (tmp != NULL) {
        it = Iterator_create(dd->size, tmp);
        if (it == NULL)
```

```
            free(tmp);
        }
        return it;
    }
```

11.2.2.7 The constructor

The implementation performs the usual heap allocations. As we discussed earlier, and empty doubly linked list is indicated by just the sentinel, with both its `next` and `prev` fields pointing to itself.

```
    static Deque template = {
        NULL, d_destroy, d_clear, d_insertFirst, d_insertLast, d_first, d_last,
        d_removeFirst, d_removeLast, d_size, d_isEmpty, d_toArray, d_itCreate
    };

    const Deque *Deque_create(void) {
        Deque *d = (Deque *)malloc(sizeof(Deque));

        if (d != NULL) {
            DData *dd = (DData *)malloc(sizeof(DData));
            if (dd != NULL) {
                dd->size = 0L;
                dd->sentinel.next = SENTINEL(dd);
                dd->sentinel.prev = SENTINEL(dd);
                *d = template;
                d->self = dd;
            } else {
                free(d);
                d = NULL;
            }
        }
        return d;
    }
```

Chapter 12

Priority queues

A *priority queue* is an abstract data type for storing a collection of prioritized elements that support arbitrary element insertion, but only supports removal of elements in order of priority. The priority associated with an element is usually a numerical value, with the smallest numerical value connoting highest priority. We will first show different implementations when the priority is an unsigned long; at the end, we will generalize to arbitrary priorities that support a comparator function.

12.1 Priorities and total order relations

A priority queue needs a comparator rule (denoted by $\leq$ below) between priorities (denoted by p_i below) that defines a *total order* relation; this means that it is defined for every pair of priorities, and it must satisfy three properties:

- **reflexive**: $p \leq p$;
- **antisymmetric**: $if\ p_1 \leq p_2\ and\ p_2 \leq p_1, then\ p_1 = p_2$;
- **transitive**: $if\ p_1 \leq p_2\ and\ p_2 \leq p_3,\ then\ p_1 \leq p_3$.

Such a rule defines a linear ordering relationship among a set of priorities; from this we can deduce that if a collection of elements has a total order defined for it, then there is a well-defined notion of a *smallest* priority, $p_{min} \leq p$, for any priority p in the collection. It should be clear that the normal numeric semantics of $\leq$ provide a total order over integers.

12.2 The interface for an integer priority queue ADT

A priority queue is a container of elements, each having an associated priority that is provided at the time that the element is inserted. We will first focus on numeric priorities; later in the chapter, we will generalize to other priority data types.

```
#ifndef _PRIOQUEUELONG_H_
#define _PRIOQUEUELONG_H_

typedef struct prioqueuelong PrioQueueLong;        /* forward reference */
const PrioQueueLong *PrioQueueLong_create(void);
struct prioqueuelong {
    void *self;
    void (*destroy)(const PrioQueueLong *pq, void(*freeFxn)(void *e));
    void (*clear)(const PrioQueueLong *pq, void (*freeFxn)(void *e));
    int (*insert)(const PrioQueueLong *pq, long priority, void *element);
    int (*min)(const PrioQueueLong *pq, void **element);
    int (*removeMin)(const PrioQueueLong *pq, void **element);
    long (*size)(const PrioQueueLong *pq);
    int (*isEmpty)(const PrioQueueLong *pq);
};

#endif /* _PRIOQUEUELONG_H_ */
```

The semantics of the constructor and the methods are as follows:

- `const PrioQueueLong *PrioQueueLong_create(void);`
  create a new instance of a priority queue using longs as a priority; returns a pointer to the dispatch table, or `NULL` if there were heap-allocation errors;
- `void (*destroy)(const PrioQueueLong *pq, void (*freeFxn)(void *e));`

- `const PrioQueueLong *PrioQueueLong_create(void);`
  create a new instance of a priority queue using longs as a priority; returns a pointer to the dispatch table, or `NULL` if there were heap-allocation errors;

- `void (*destroy)(const PrioQueueLong *pq, void (*freeFxn)(void *e));`
  destroy the priority queue; if `freeFxn != NULL`, this function is invoked on each element in the priority queue; then all heap-allocated memory associated with the priority queue is freed;
- `void (*clear)(const PrioQueueLong *pq, void (*freeFxn)(void *e));`
  purge all elements from the priority queue; if `freeFxn != NULL`, this function is invoked on each element in the priority queue; then heap-allocated memory associated with the entries in the priority queue is freed; upon return, the priority queue is empty;
- `int (*insert)(const PrioQueueLong *pq, long priority, void *element);`
  insert `element` in the priority queue at the location appropriate to `priority`; if other elements at that priority are already in the queue, this element should be placed after the last such element (FIFO within a priority); returns 0 if there are heap-allocation problems, 1 otherwise;
- `int (*min)(const PrioQueueLong *pq, void **element);`
  returns the `element` with the smallest `priority` in `*element`; function return is 1 if successful, 0 if priority queue is empty;
- `int (*removeMin)(const PrioQueueLong *pq, void **element);`
  returns the `element` with the smallest `priority` in `*element` and removes that element from the priority queue; function return is 1 if successful, 0 if priority queue

is empty;

- `long (*size)(const PrioQueueLong *pq);`
  returns the number of elements in the priority queue;
- `int (*isEmpty)(const PrioQueueLong *pq);`
  returns 1 if the priority queue is empty, 0 otherwise.

12.3 Implementations for an integer priority queue ADT

There are several possible implementation strategies for a priority queue:

1. implement as an unordered linked list - we always insert a new element at the end of the linked list, but in order to implement `min()` and `removeMin()` we have to scan the entire list to find the smallest priority value; this means that insertions are $O(1)$, but removal is $O(n)$; not a particularly elegant solution, and the worst running time complexity is during removal; we will *not* show the implementation for this approach;
2. implement as an ordered linked list - we always insert a new element at the appropriate place in the list such that the comparator rule holds between adjacent items in the list; if there are already elements in the list with the specified priority, the new element is inserted after those elements (FIFO within a particular priority); removal is always to return the element at the head of the list; thus, insertion is $O(n)$ and removal is $O(1)$; this is a slightly more elegant solution, with worst running time complexity for insertions; we will show this implementation below;
3. if there are a bounded number of priorities, we can implement the priority queue as a set of FIFO queues, one for each priority; insertions are now $O(1)$, and removals are $O(P)$ where P is the number of priorities; for example, the Linux process scheduler supports 140 different process priorities [0..199]; since the number of priorities never changes, this means that removals are also $O(1)$; we will show changes to the ordered linked list implementation for this approach; it *will* require that we modify the function signature for the constructor, since we need to indicate the total number of priority values;
4. if there is no bound for the number of priorities, but you would like both insertions and removals to have better running time complexity than $O(n)$, we can use a heap to implement the priority queue; for this implementation, the running time complexity will be $O(\log n)$; we will introduce heaps below, and a full implementation of a priority queue using a heap.

12.3.1 Implementation as an ordered linked list

12.3.1.1 The preliminaries

Standard set of includes, macro to obtain address of sentinel node, typedef for linked list node, and definition of the instance-specific data structure.

```
/*
 * implementation for generic priority queue, where priorities are longs
 * implemented as a doubly-linked list
 */

#include "prioqueuelong.h"
#include <stdlib.h>

#define SENTINEL(p) (&(p)->sentinel)

typedef struct llnode {
    struct llnode *next;
    struct llnode *prev;
    long priority;
    void *element;
} LLNode;

typedef struct pq_data {
    long size;
    LLNode sentinel;
} PqData;
```

12.3.1.2 destroy() and clear()

Two helper functions, one to apply `freeFxn()` to each element, the other to free the nodes from the list.

```
/*
 * traverses linked list, calling freeFxn on each element
 */
static void purge(PqData *pqd, void (*freeFxn)(void *e)) {
    if (freeFxn != NULL) {
        LLNode *p;

        for (p = pqd->sentinel.next; p != SENTINEL(pqd); p = p->next)
            (*freeFxn)(p->element);
    }
}

/*
 * frees the nodes from the doubly-linked list
 */
static void freeList(PqData *pqd) {
    LLNode *p = pqd->sentinel.next;

    while (p != SENTINEL(pqd)) {
        LLNode *q = p->next;
        free(p);
        p = q;
    }
}
```

```
static void pq_destroy(const PrioQueueLong *pq, void (*freeFxn)(void *e)) {
    PqData *pqd = (PqData *)pq->self;
    purge(pqd, freeFxn);
    freeList(pqd);
    free(pqd);
    free((void *)pq);
}

static void pq_clear(const PrioQueueLong *pq, void (*freeFxn)(void *e)) {
    PqData *pqd = (PqData *)pq->self;
    purge(pqd, freeFxn);
    freeList(pqd);
    pqd->size = 0L;
    pqd->sentinel.next = SENTINEL(pqd);
    pqd->sentinel.prev = SENTINEL(pqd);
}
```

12.3.1.3 `insert()`

A helper function to link a new node into the doubly-linked list is defined. The method needs to traverse the entire list each time looking for the correct pair of nodes between which the new node must be inserted.

```
/*
 * link `p` between `before' and `after'
 * must work correctly if `before' and `after' are the same node
 * (i.e. the sentinel)
 */
static void link(LLNode *before, LLNode *p, LLNode *after) {
    p->next = after;
    p->prev = before;
    after->prev = p;
    before->next = p;
}

static int pq_insert(const PrioQueueLong *pq, long priority, void *element) {
    PqData *pqd = (PqData *)pq->self;
    int status = 0;
    LLNode *p = (LLNode *)malloc(sizeof(LLNode));

    if (status) {
        LLNode *q = pqd->sentinel.next;
        while (q != SENTINEL(pqd) && q->priority <= priority)
            q = q->next;
        /* q either points to the sentinel or to the first entry for which
           the priority is larger than our insertion */
        p->priority = priority;
        p->element = element;
        link(q->prev, p, q);
```

```
            pqd->size++;
        }
        return status;
    }
```

12.3.1.4 min() and removeMin()

`min()` is straightforward. A helper function to unlink a node from the doubly-linked list is defined. With that helper function, `removeMin()` is also straightforward.

```
    static int pq_min(const PrioQueueLong *pq, void **element) {
        PqData *pqd = (PqData *)pq->self;
        int status = 0;
        LLNode *p = SENTINEL(pqd)->next;

        if (p != SENTINEL(pqd)) {
            status = 1;
            *element = p->element;
        }
        return status;
    }

    /*
     * unlinks the LLNode from the doubly-linked list
     */
    static void unlink(LLNode *p) {
        p->prev->next = p->next;
        p->next->prev = p->prev;
    }

    static int pq_removeMin(const PrioQueueLong *pq, void **element) {
        PqData *pqd = (PqData *)pq->self;
        int status = 0;
        LLNode *p = SENTINEL(pqd)->next;

        if (p != SENTINEL(pqd)) {
            status = 1;
            *element = p->element;
            unlink(p);
            free(p);
            pqd->size--;
        }
        return status;
    }
```

12.3.1.5 size() and isEmpty()

The identical implementation to all of our other ADTs.

```
static long pq_size(const PrioQueueLong *pq) {
    PqData *pqd = (PqData *)pq->self;
    return pqd->size;
}

static int pq_isEmpty(const PrioQueueLong *pq) {
    PqData *pqd = (PqData *)pq->self;
    return (pqd->size == 0L);
}
```

12.3.1.6 The constructor

Again, there should be no surprises here.

```
static PrioQueueLong template = {
    NULL, pq_destroy, pq_clear, pq_insert, pq_min, pq_removeMin, pq_size,
    pq_isEmpty
};

const PrioQueueLong *PrioQueueLong_create(void) {
    PrioQueueLong *pq = (PrioQueueLong *)malloc(sizeof(PrioQueueLong));

    if (pq != NULL) {
        PqData *pqd = (PqData *)malloc(sizeof(PqData));

        if (pqd != NULL) {
            pqd->size = 0L;
            pqd->sentinel.next = SENTINEL(pqd);
            pqd->sentinel.prev = SENTINEL(pqd);
            *pq = template;
            pq->self = pqd;
        } else {
            free(pq);
            pq = NULL;
        }
    }
    return pq;
}
```

12.3.2 Implementation as a set of FIFO linked lists

As alluded to above, we need to modify the signature of the constructor as follows:

```
const PrioQueueLong *PrioQueueLong_create(long minP, long maxP);
```

which indicates that elements associated with priorities in the range $[minP, maxP]$ will be inserted. We reproduce here the entire source file, with changes from the ordered list implementation highlighted.

```
/*
 * implementation for generic priority queue, where priorities are longs
 * implemented as an array of FIFO linked lists
 */

#include "prioqueuelongbounded.h"     has modified constructor signature
#include <stdlib.h>

typedef struct llnode {
    struct llnode *next;
    long priority;     removed prev pointer
    void *element;
} LLNode;

typedef struct listhead {     listhead structure
    LLNode *head;
    LLNode *tail;
} ListHead;

typedef struct pq_data {     most of this structure is different
    long size;
    long minPrio;
    long maxPrio;
    long nPrios;
    ListHead *listheads;
} PqData;

/*     code is different reflecting new structure
 * traverses linked lists, calling freeFxn on each element
 */
static void purge(PqData *pqd, void (*freeFxn)(void *e)) {
    if (freeFxn != NULL) {
        LLNode *p;
        long i;

        for (i = 0; i < pqd->nPrios; i++)
            for (p = pqd->listheads[i].head; p != NULL; p = p->next)
                (*freeFxn)(p->element);
    }
}

/*     code is different reflecting new structure
 * frees the nodes from the linked lists
 */
static void freeList(PqData *pqd) {
    long i;
    for (i = 0; i < pqd->nPrios; i++) {
        LLNode *p = pqd->listheads[i].head;
        while (p != NULL) {
            LLNode *q = p->next;
            free(p);
            p = q;
        }
        pqd->listheads[i].head = NULL;
        pqd->listheads[i].tail = NULL;
    }
}

static void pq_destroy(const PrioQueueLong *pq, void (*freeFxn)(void *e)) {
    PqData *pqd = (PqData *)pq->self;
    purge(pqd, freeFxn);
    freeList(pqd);
    free(pqd);
    free((void *)pq);
}

static void pq_clear(const PrioQueueLong *pq, void (*freeFxn)(void *e)) {
    PqData *pqd = (PqData *)pq->self;
    purge(pqd, freeFxn);
    freeList(pqd);
    pqd->size = 0L;
}
```

Most of this code is different, again reflecting the different structure

```
static int pq_insert(const PrioQueueLong *pq, long priority, void *element) {
    PqData *pqd = (PqData *)pq->self;
    int status = 0;
    long i = priority - pqd->minPrio;

    if (i < pqd->nPrios) {
        LLNode *p = (LLNode *)malloc(sizeof(LLNode));

        if (p != NULL) {
            if (pqd->listheads[i].head == NULL)
                pqd->listheads[i].head = p;
            else
                (pqd->listheads[i].tail)->next = p;
            pqd->listheads[i].tail = p;
            p->priority = priority;
            p->element = element;
            p->next = NULL;
            pqd->size++;
            status = 1;
        }
    }
    return status;
}

/*    New helper function
 * helper function to find the first non-empty priority list;
 * returns number of priorities if all lists are empty
 */
static long findFirst(PqData *pqd) {
    long i;

    for (i = 0; i < pqd->nPrios; i++)
        if (pqd->listheads[i].head != NULL)
            break;
    return i;
}
```

Most of this code is different, again reflecting the different structure

```
static int pq_min(const PrioQueueLong *pq, void **element) {
    PqData *pqd = (PqData *)pq->self;
    int status = 0;
    long i = findFirst(pqd);

    if (i < pqd->nPrios) {
        status = 1;
        *element = (pqd->listheads[i].head)->element;
    }
    return status;
}
```

Most of this code is different, again reflecting the different structure

```
static int pq_removeMin(const PrioQueueLong *pq, void **element) {
    PqData *pqd = (PqData *)pq->self;
    int status = 0;
    long i = findFirst(pqd);

    if (i < pqd->nPrios) {
        LLNode *p = pqd->listheads[i].head;
        status = 1;
        *element = p->element;
        if ((pqd->listheads[i].head = p->next) == NULL)
            pqd->listheads[i].tail = NULL;
        free(p);
        pqd->size--;
    }
    return status;
}

static long pq_size(const PrioQueueLong *pq) {
    PqData *pqd = (PqData *)pq->self;
    return pqd->size;
}

static int pq_isEmpty(const PrioQueueLong *pq) {
    PqData *pqd = (PqData *)pq->self;
    return (pqd->size == 0L);
}

static PrioQueueLong template = {
    NULL, pq_destroy, pq_clear, pq_insert, pq_min, pq_removeMin, pq_size,
    pq_isEmpty
};

const PrioQueueLong *PrioQueueLong_create(long minP, long maxP) {
    PrioQueueLong *pq = (PrioQueueLong *)malloc(sizeof(PrioQueueLong));

    if (pq != NULL) {
        PqData *pqd = (PqData *)malloc(sizeof(PqData));

        if (pqd != NULL) {
            long nPrios = maxP - minP + 1;
            ListHead *p = (ListHead *)malloc(nPrios * sizeof(ListHead));
```

Had to allocate array of listheads

```
            if (p != NULL) {
                long i;
```

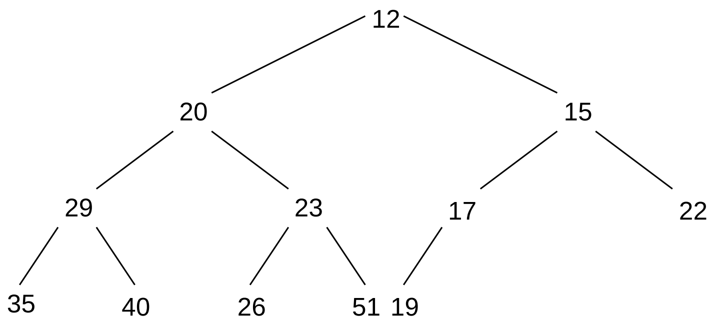
Appropriately initialize PqData, especially initialize listheads

```
                pqd->minPrio = minP;
                pqd->maxPrio = maxP;
                pqd->nPrios = nPrios;
                pqd->size = 0L;
                pqd->listheads = p;
                for (i = 0; i < nPrios; i++) {
                    pqd->listheads[i].head = NULL;
                    pqd->listheads[i].tail = NULL;
                }
                *pq = template;
                pq->self = pqd;
            } else {
                free(pqd);
                free(pq);
                pq = NULL;
            }
        } else {
            free(pq);
            pq = NULL;
        }
    }
    return pq;
}
```

12.3.3 Implementation using a heap

12.3.3.1 What is a heap?

A heap is a data structure for representing a collection of items - just what we need for a priority queue. We will be using a heap of long integers; elements of a heap can be any ordered type. For example, here is a heap of 12 integers:

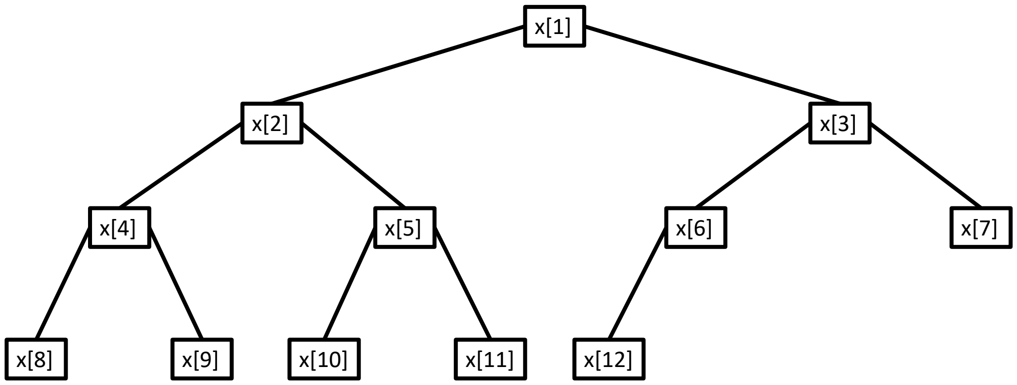


This is an example of a binary tree - i.e., each node has 0, 1, or 2 children. This binary tree is a heap by virtue of two properties:

- *order* - the value at any node is less than or equal to the values of the node's children; this implies that the least element in the set of numbers is at the root of the tree (12 in this case), but it does *not* say anything about the relative order of the left and right children; and
- *shape* - a heap has its terminal nodes on at most two levels, with those on the bottom level as far left as possible; if it contains N nodes, no node is at a distance of more than $\log_2 N$ levels from the root.

These two properties are restrictive enough to enable us to find the minimum element in a bag of numbers, but lax enough so that we can efficiently reorganize the structure after inserting or deleting an element.

Despite a heap being a binary tree, there is a *very* efficient way to represent the heap without using pointers - this is because a heap possesses the *shape* property. A 12-element tree with the shape property is represented in a 12-element array $x[1]...x[12]$ as:



Note that heaps use a 1-based array (we will see why in a few lines); the easiest approach in C is to declare $x[N+1]$ and not use element $x[0]$. In this implicit representation of a binary tree with the *shape* property, the root is $x[1]$, its two children are $x[2]$ and $x[3]$, etc. The typical functions on a tree become:

- root = 1
- value(i) = x[i]
- leftchild(i) = 2*i
- rightchild(i) = 2*i + 1
- parent(i) = i / 2 (integer division)

A C declaration for the 12-element tree shown above would be

```
int x[12+1] = {0, 12, 20, 15, 29, 23, 17, 22, 35, 40, 26, 51, 19};
```

Because the shape property is guaranteed by the representation, the name *heap* will mean that the value of any node is greater than or equal to the value of its parent. Phrased precisely, the array $x[1]...x[N]$ has the heap property if

$$\forall_{2 \le i \le N}\ x[i/2] \le x[i].$$

This works correctly since integer division rounds down, so 4/2 and 5/2 both yield a parent of 2. We will need to be able to talk about a subarray $x[1]...x[u]$ having the *heap* property, so we will mathematically define $heap(l,u)$ as

$$\forall_{2l \leq i \leq u}\ x[i/2] \leq x[i].$$

12.3.3.2 Two critical functions

When we add another element to a heap, assuming that the heap currently occupies $x[1]...x[N-1]$, the easiest thing to do is to place the new element in $x[N]$. Unfortunately, $heap(1, N)$ will most likely not be true. Therefore, we need an efficient algorithm to modify $x[1]...x[N]$ such that $heap(1, N)$ is true. The function that re-establishes the *heap* property is called *siftup*.

The code for `siftup()` is as follows:

```
void siftup(int x[], int n) {
/*
 * preconditions: n > 0 && heap(1, n-1)
 * postcondition: heap(1,n)
 */
    int p, i = n;

    while (i > 1) {
        int tmp = x[p];
        p = i/2;
        if (x[p] <= x[i])
            break;
        x[p] = x[i];
        x[i] = tmp;
        i = p;
    }
}
```

If $heap(1, N)$ is true, and we remove $x[1]$, we would like to move the $N-1$ remaining elements into $x[1]...x[N-1]$, but we need to re-establish $heap(1, N-1)$. The easiest way to do this is to place $x[N]$ into $x[1]$, and then re-establish $heap(1, N-1)$. The function *siftdown* re-establishes our heap property.

```
void siftdown(int x[], int n) {
/*
 * preconditions: heap(2,n) && n >= 0
 * postcondition: heap(1, n-1)
 */
   int c, i;

    i = 1;
    for (;;) {
        int tmp = x[i];
        c = 2 * i;
        if (c > n)
            break;
        if ((c+1) <= n && x[c+1] < x[c])
            c++;
        if (tmp <= x[c])
            break;
        x[i] = x[c]
        x[c] = tmp;
        i = c;
    }
}
```

12.3.3.3 The implementation

We are now in a position to implement the priority queue using a heap. We can revert back to our original function signature for the constructor, as we can grow the array beyond our original default size when needed. As with the Stack implementation, we will double the size of the array whenever growth is needed.

```
/*
 * implementation for generic priority queue, where priorities are longs
 * implemented using a heap that expands when needed
 */

#include "prioqueuelong.h"
#include <stdlib.h>

#define DEFAULT_HEAP_SIZE 25

typedef struct heapnode {
    long priority;
    void *element;
} HeapNode;

typedef struct pq_data {
    long last;
    long size;
    HeapNode *heap;
} PqData;

/*
 * traverses the heap, calling freeFxn on each element
 */
static void purge(PqData *pqd, void (*freeFxn)(void *e)) {
    if (freeFxn != NULL) {
        long i;
        for (i = 1; i <= pqd->last; i++)
            (*freeFxn)(pqd->heap[i].element);
    }
}

static void pq_destroy(const PrioQueueLong *pq, void (*freeFxn)(void *e)) {
    PqData *pqd = (PqData *)pq->self;
    purge(pqd, freeFxn);
    free(pqd->heap);
    free(pqd);
    free((void *)pq);
}

static void pq_clear(const PrioQueueLong *pq, void (*freeFxn)(void *e)) {
    PqData *pqd = (PqData *)pq->self;
    purge(pqd, freeFxn);
    pqd->last = 0L;
}

/*
 *  the siftup function restores the heap property after adding a new element
 *  preconditions: last > 0 && heap(1,last-1)
 *  postcondition: heap(1,last)
 */
static void siftup(PqData *pqd) {
    int p, i = pqd->last;

    while (i > 1) {
        HeapNode hn;
        p = i / 2;
        if (pqd->heap[p].priority <= pqd->heap[i].priority)
            break;
        hn = pqd->heap[p];
        pqd->heap[p] = pqd->heap[i];
        pqd->heap[i] = hn;
        i = p;
    }
}

static int pq_insert(const PrioQueueLong *pq, long priority, void *element) {
    PqData *pqd = (PqData *)pq->self;
    long i = pqd->last + 1;
    int status = (i < pqd->size);

    if (! status) {       /* need to resize the array */
        size_t nbytes = (2 * pqd->size) * sizeof(HeapNode);
        HeapNode *tmp = (HeapNode *)realloc(pqd->heap, nbytes);

        if (tmp != NULL) {
            status = 1;
            pqd->heap = tmp;
            pqd->size *= 2;
        }
    }
    if (status) {
        pqd->heap[i].priority = priority;
        pqd->heap[i].element = element;
        pqd->last = i;
        siftup(pqd);
    }
    return status;
}

static int pq_min(const PrioQueueLong *pq, void **element) {
    PqData *pqd = (PqData *)pq->self;
    int status = 0;

    if (pqd->last > 0L) {
        status = 1;
        *element = (pqd->heap[1].element);
    }
    return status;
}

/*
 *  the siftdown function restores the heap property after removing
 *  the top element, and replacing it by the previous last element
 *  preconditions: heap(2,last) && last >= 0
 *  postcondition: heap(1,last-1)
 */
static void siftdown(PqData *pqd) {
    int c, i;

    i = 1;
    for(;;) {
        HeapNode hn;
        c = 2 * i;
        if (c > pqd->last)
            break;
        if ((c+1) <= pqd->last && pqd->heap[c+1].priority < pqd->heap[c].priority)
            c++;
        if (pqd->heap[i].priority <= pqd->heap[c].priority)
            break;
        hn = pqd->heap[i];
        pqd->heap[i] = pqd->heap[c];
        pqd->heap[c] = hn;
        i = c;
    }
}

static int pq_removeMin(const PrioQueueLong *pq, void **element) {
    PqData *pqd = (PqData *)pq->self;
    int status = 0;
    if (pqd->last > 0L) {
        status = 1;
        *element = (pqd->heap[1].element);
        pqd->heap[1] = pqd->heap[pqd->last];
        pqd->last--;
        siftdown(pqd);
    }
    return status;
}

static long pq_size(const PrioQueueLong *pq) {
    PqData *pqd = (PqData *)pq->self;
    return pqd->last;
}

static int pq_isEmpty(const PrioQueueLong *pq) {
    PqData *pqd = (PqData *)pq->self;
    return (pqd->last == 0L);
}

static PrioQueueLong template = {
    NULL, pq_destroy, pq_clear, pq_insert, pq_min, pq_removeMin, pq_size,
    pq_isEmpty
};

const PrioQueueLong *PrioQueueLong_create(void) {
    PrioQueueLong *pq = (PrioQueueLong *)malloc(sizeof(PrioQueueLong));

    if (pq != NULL) {
        PqData *pqd = (PqData *)malloc(sizeof(PqData));

        if (pqd != NULL) {
            HeapNode *p = (HeapNode *)malloc(DEFAULT_HEAP_SIZE * sizeof(HeapNode));

            if (p != NULL) {
                pqd->size = DEFAULT_HEAP_SIZE;
                pqd->last = 0L;
                pqd->heap = p;
                *pq = template;
                pq->self = pqd;
            } else {
                free(pqd);
                free(pq);
                pq = NULL;
            }
        } else {
            free(pq);
            pq = NULL;
        }
    }
    return pq;
}
```

Chapter 13

Searching and sorting

In the data structures that we have covered so far, the containers were structured in such a way that we did not have to search for the items to return. We had methods such as `pop()` for a Stack, `dequeue()` for a Queue, and `removeFirst()` and `removeLast()` for a Deque. Searching for an item in a container is a very common operation when constructing an application. This chapter will first remind you of two different types of searching that you will have learned in your previous programming experience: linear and binary search. It will then focus on different types of sorting algorithms and their running time complexities that can be used to support binary search.

13.1 Searching

The basic structured data type provided by C is the array. An array has a declared type and size; the cells that make up the array are allocated in contiguous memory locations. The size of an array is not part of the array, so one must remember the size using another variable. We have seen examples of this for the array-based Stack and Queue implementations.

For the next two subsections, we will assume the following declarations:

```
#define N 32
void *a[N];          /* the array of values in which we search */
void *x;             /* the value for which we are searching */
int size = N;        /* legal indices are [0..N-1] */
/*
 * the following function compares two values, returning <0 | 0 | >0
 * if v1 < v2 | v1 == v2 | v1 > v2
 */
int cmp(void *v1, void *v2);
```

13.1.1 Linear search

If we do not have any information about how the values are stored in the array, we have to resort to *linear search*. Given `x`, we wish to search the array `a[]` to see if it is contained therein. We will start searching at the beginning of `a[]`.

There are two conditions that terminate the search:

1. the element is found - i.e., `a[i] == x` for some `0 <= i < N`; or
2. the entire array has been searched, and no match was found.

We will write a function that searches the array for a value, returning the index in `a` at which `x` was found; if it is not found, we return `-1`.

```
int linearSearch(void *a[], int size, void *x) {
    int i;
    int ans = -1;     /* assume failure */

    for (i = 0; i < size; i++)
        if (cmp(a[i], x) == 0) {
            ans = i;
            break;
        }
    return ans;
}
```

Does this return the correct function values? There are two ways that we end up at the `return ans;` statement:

1. we have encountered an array element `a[i]` that is equal to `x` (as determined by the function `cmp()`); when this occurs, we assign the value of `i` to `ans` and break out of the `for` loop; or
2. `i >= size`, in which case we do not execute the loop; since we initialized `ans` to -1, this is the result that will be returned.

It should be clear that this function has $O(N)$ running time complexity since it has to process some portion of the array; in the worst case, it has to invoke `cmp` on all of the items in the array; in the average case, when searching for an item that *is* in the array, one will invoke `cmp` on half of the items. Thus the complexity of the function depends upon the number of items in the array, yielding $O(N)$ running time complexity.

13.1.2 Binary search

The only way we will be able to improve on the running time complexity is if there is more information about the data to be searched. It is well known that a search can be made much more efficient if the data to be searched is ordered. For example, consider how you use an index in a book; the index is sorted alphabetically, so you can quickly zero in on the topic of interest, and from there go to the pages upon which that topic is discussed. In particular, you do not need to linearly search through the index.

So let's assume that our array, `a[]`, is ordered by the comparator function `cmp()`. That is:

$$\forall_{1 \leq k < N} \; cmp(a[k-1], a[k]) \leq 0$$

Since the array is ordered, we can use a *divide and conquer* approach:

- select a value at random, say `a[m]` for some legal index `m`;
- compare `x` with the selected array element;
- if they are equal, terminate the search;
- if $cmp(a[m], x) < 0$, it means that array elements with indices less than or equal to `m` can be eliminated from the search, and repeat the process with the indices greater than `m`;
- if $cmp(a[m], x) > 0$, then we eliminate indices greater than or equal to `m` from the search, and repeat the process with indices less than `m`;
- since the range of indices that we process reduces each time, we eventually either find the element, or we have no more indices to check.

Let's write a function that searches the array using this algorithm.

```
int binarySearch(void *a[], int size, void *x) {
    int left = 0, right = size - 1;
    int ans = -1;                    /* assume failure */

    while (left <= right) {
        int m = (left + right) / 2;  /* take the midpoint */
        result = cmp(a[m], x);

        if (result == 0) {
            ans = m;
            break;
        } else if (result < 0)
            left = m + 1;
        else
            right = m - 1;
    }
    return ans;
    }
```

How does this work? Let's assume the worst case, that `x` is not in `a[]`; we will assume that `x < a[0]`, but the complexity is the same if $a[0] \leq x \leq a[N-1]$ and `x` is not in the array. Let's track the values of `left` and `right` through the iterations of the while loop.

iteration	left	right	m
1	0	31	15
2	0	14	7
3	0	6	3
4	0	2	1
5	0	0	0
6	0	-1	

By selecting the midpoint at each iteration, we are reducing the number of elements to search by $\frac{1}{2}$. Such a reduction at each iteration means that we will perform $\log_2 N$ iterations in the worst case. In our case, we had 32 elements in the array, and $\log_2 32$ is 5; in fact, we performed exactly 5 iterations, since the 6th iteration listed above actually fails the `left <= right` condition of the while loop.

We already know from Chapter 6 that $O(\log n)$ grows much more slowly than $O(n)$, so binary search is preferred over linear search if the array being searched is ordered.

13.2 Sorting

Since searching over a sorted array is so much more scalable than over an unsorted array, we now need to discuss various ways to sort an array to enable this more scalable behavior. You may remember that the *list* datatype in Python has a `sort()` method, which will sort the list in place. There is also a `sorted()` function which will produce a sorted list from any argument that supports iteration. We will be focussing on a C function that is more like the `sorted()` approach.

The basic problem we are trying to solve is as follows:

- we have an array of items that need to be ordered;
- we assume that all of the elements fit into the random access memory of the computer; if they do not, then one must resort to external sorting algorithms, which is beyond the scope of this book;
- we have a comparator function defined on the items that determines the appropriate ordering of the array elements;
- the array elements may already be ordered, or may be in random order.

Given this set of requirements, we will first look at algorithms that exhibit $O(n^2)$

complexity. Then we will move on to algorithms that exhibit better than $O(n^2)$ complexity.

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array. There are some applications for which a stable sorting algorithm is required - e.g., you have multi-column records, and you first sort by column 2, then by column 1, and you want all records that have identical values in column 1 to remain sorted by column 2. We will note those algorithms that are stable by nature.

For all of the sort algorithms described in the subsequent subsections, the function prototype for our function is:

```
void sort(void *a[], long size, int (*cmp)(void *v1, void *v2));
```

We will link our sort implementations with the following main program to test it out.

```
#include "sort.h"
#include <stdio.h>
#include <stdlib.h>

void display(void *a[], int size) {
    int i;
    for (i = 0; i < size; i++) {
        long v = (long)a[i];
        printf("%3ld", v);
    }
    printf("\n");
}

int cmp(void *v1, void *v2) {
    long l1 = (long)v1;
    long l2 = (long)v2;
    return (l1 - l2);
}

int main(int argc, char *argv[]) {
    void *a[25];
    int i, j;

    for (i = 1, j = 0; i < argc; i++) {
        long v = atoi(argv[i]);
        a[j++] = (void *)v;
    }
    display(a, j);
    sort(a, (long)j, cmp);
    display(a, j);
    return 0;
}
```

We link `sortmain.o` with the object file from our sort algorithm, and invoke the resulting executable as

```
$ ./program 23 40 15 73 97 11 2 87
```

with the resulting output being

```
 23 40 15 73 97 11  2 87
  2 11 15 23 40 73 87 97
```

13.2.1 Sorting algorithms with *O(n²)* complexity

Sorting methods that sort items *in situ* can be classified into three principal categories according to their underlying method:

1. sort by insertion;
2. sort by selection; or
3. sort by exchange.

13.2.1.1 Insertion sort

Insertion sort is typically how bridge players structure their hands after they are dealt. The items are divided into a destination sequence, `a[0]...a[i-1]`, and a source sequence, `a[i]...a[N-1]`. We start with `i = 1`, yielding a destination sequence consisting of `a[0]` (since a singleton is obviously sorted), and a source sequence of `a[1]...a[N-1]`. At each step, the $\texttt{i}^{\text{th}}$ element of the source sequence is transferred into the appropriate place in the destination sequence, and then `i` is incremented. These steps are repeated until the source sequence is empty, at which point the destination sequence is the entire sorted array.

Here is a C implementation of insertion sort.

```
#include "sort.h"

void sort(void *a[], long size, int (*cmp)(void *v1, void *v2)) {
    long i;

    for (i = 1; i < size; i++) {
        void *x = a[i];
        long j;

        for (j = i - 1; j >= 0 && (*cmp)(a[j], x) > 0; j--)
            a[j+1] = a[j];
        a[j+1] = x;
    }
}
```

Lets look at how the array changes with each iteration over `i` in the insertion sort function.

Input	23	40	15	73	97	11	2	87
i = 1	23	40	15	73	97	11	2	87
i = 2	15	23	40	73	97	11	2	87
i = 3	15	23	40	73	97	11	2	87
i = 4	15	23	40	73	97	11	2	87
i = 5	11	15	23	40	73	97	2	87
i = 6	2	11	15	23	40	73	97	87
i = 7	2	11	15	23	40	73	87	97

Each iteration over `j` checks that `a[j] > x`; thus, if we have duplicates in the initial array, we guarantee that the second identical value will not overtake the first. Thus, insertion sort is stable.

The complexity for insertion sort is obviously $O(n^2)$, since the outer for loop (over `i`) is executed `n - 1` times. The inner for loop (over `j`) is executed `i` times; since `i` is a fraction of `n`, this means that we have $O(n^2)$ comparisons and moves.

13.2.1.2 Selection sort

Selection sort is based upon the following approach:

1. select the array element that has the smallest value;
2. exchange it with `a[0]`;
3. repeat with the remaining `n-1` items, then `n-2` items, until only one item, which is the largest, is left.

In some sense, selection is the opposite of insertion; insertion sort considers in each step only the *one* next item of the source sequence and *all* the items of the destination sequence; selection sort considers *all* items of the source sequence to find the one with the smallest value, and then deposits it as the *one* next item of the destination sequence.

Here is a C implementation of selection sort.

```
#include "sort.h"

void sort(void *a[], long size, int (*cmp)(void *v1, void *v2)) {
    long i;

    for (i = 0; i < size - 1; i++) {
        long k = i;
        long j;

        for (j = i + 1; j < size; j++)
            if ((*cmp)(a[j], a[k]) < 0)
```

```
                k = j;
            if (k != i) {
                void *x = a[i];
                a[i] = a[k];
                a[k] = x;
            }
        }
    }
```

Lets look at how the array changes with each iteration over `i` in the selection sort function.

Input	23	40	15	73	97	11	2	87
i = 0	2	40	15	73	97	11	23	87
i = 1	2	11	15	73	97	40	23	87
i = 2	2	11	15	73	97	40	23	87
i = 3	2	11	15	23	97	40	73	87
i = 4	2	11	15	23	40	97	73	87
i = 5	2	11	15	23	40	73	97	87
i = 6	2	11	15	23	40	73	87	97

Selection sort is *not* stable, since at each step it swaps the current element with the minimum element in the source sequence. If the source sequence had a duplicate of the current element at a location before the index where the minimum element was found, it has inverted the order of the two identical elements, and is thus not stable.

The complexity is again $O(n^2)$, since we again have an outer loop that is performed `n-1` times, and an inner loop that is performed a fraction of `n` times, thus yielding $O(n^2)$.

13.2.1.3 Exchange sort

While both insertion and selection sorts performed the exchange of two items, the exchange was not the dominant characteristic of the algorithm. Exchange sort is based on the principle of comparing and exchanging pairs of adjacent items until all items are sorted.

As in selection sort, we make repeated passes over the array, each time "bubbling" the least item of the remaining set to the left end of the subarray under consideration. This method is widely known as *Bubblesort*.

Here is a C implementation of Bubblesort/exchange sort.

```
#include "sort.h"

void sort(void *a[], long size, int (*cmp)(void *v1, void *v2)) {
    long i;

    for (i = 1; i < size; i++) {
        long j;

        for (j = size - 1; j >= i; j--)
            if ((*cmp)(a[j-1], a[j]) > 0) {
                void *x = a[j-1];
                a[j-1] = a[j];
                a[j] = x;
            }
    }
}
```

Let's look at how the array changes with each iteration over `i` in the exchange sort function.

Input	23	40	15	73	97	11	2	87
i = 1	2	23	40	15	73	97	11	87
i = 2	2	11	23	40	15	73	97	87
i = 3	2	11	15	23	40	73	87	97
i = 4	2	11	15	23	40	73	87	97
i = 5	2	11	15	23	40	73	87	97
i = 6	2	11	15	23	40	73	87	97
i = 7	2	11	15	23	40	73	87	97

Note that the order of the array elements did not change for the last four iterations for the outer loop. Thus, we can improve the implementation by noting in each pass whether any exchanges were made; if not, the algorithm can terminate. This improvement is shown in the following implementation.

```
#include "sort.h"

void sort(void *a[], long size, int (*cmp)(void *v1, void *v2)) {
    long i;
    int nexchange = 1;        /* guarantee that we perform first iteration */

    for (i = 1; i < size && nexchange > 0; i++) {
        long j;

        for (j = size - 1, nexchange = 0; j >= i; j--)
            if ((*cmp)(a[j-1], a[j]) > 0) {
                void *x = a[j-1];
                a[j-1] = a[j];
                a[j] = x;
                nexchange++;
            }
    }
}
```

With these changes, we guarantee that we perform the minimum number of outer loop iterations.

Exchange sort *is* stable, since the pairwise exchanges is driven by $a[j-1] > a[j]$; thus, items with the same values will cluster together in the same order in which they appeared in the input array.

The complexity is $O(n^2)$, since the outer loop is performed `n - 1` times, and the inner loop is performed a fraction of `n` times. The improvement obtained by tracking the number of exchanges in each iteration generally improves the performance, but the complexity remains $O(n^2)$.

There is a particular asymmetry to the exchange sort; if there is a small value at the end of the input array, it will bubble into its correct position in a single pass; if there is a very large value at the beginning of the input array, it will only move one location toward its correct position with each iteration. For example, the array

```
11 15 23 40 73 87 97 2
```

is sorted by our improved exchange sort in a single pass, but the array

```
97 2 11 15 23 40 73 87
```

requires 7 passes to sort the array.

Thus, another improvement to exchange sort can be obtained if we alternate the direction of consecutive passes. This is left as a programming exercise at the end of the chapter.

13.2.2 Sorting algorithms with better than *O(n²)* complexity

As promised, this section will cover versions of insertion, selection, and exchange sort that exhibit better running time complexity than the simple sort algorithms shown previously.

13.2.2.1 Shell sort

The insertion sort presented in Section 13.2.1.1 forces an $O(n^2)$ complexity since you have to shuffle through all of the elements of the destination sequence to find the right place for the next item from the source sequence. This variation of insertion sort requires that we perform insertion sort with different sized strides, starting with a large stride, then processing the array with a smaller stride, ..., until we are left with a stride of 1. While it is not obvious that this will improve the running time complexity, it is known to yield a complexity of $O(n^{1.2})$.

Let's try it on our standard eight item sequence of integers, with strides of 4, 2, and 1.

Input	23	40	15	73	97	11	2	87
after 4-sort	23	11	2	73	97	40	15	87
after 2-sort	2	11	15	40	23	73	97	87
after 1-sort	2	11	15	23	40	73	87	97

It may seem counterintuitive that several sorting passes, each of which involves all items, reduces the complexity. Note that each sorting step over a chain either involves relatively few items or the items are already quite well-ordered and relatively few rearrangements are required.

While it is clear from the table above that this works when the set of strides are decreasing powers of two, it works even better if the strides related in this way. Knuth suggests two reasonable choices for the increments $(h_1, h_2, ..., h_t)$ to be used (with the order reversed):

1, 4, 13, 40, 121, ... i.e., $h_{k-1} = 3h_k + 1$, $h_t = 1$, and $t = \lfloor \log_3 n \rfloor - 1$

1, 3, 7, 15, 31, ... i.e., $h_{k-1} = 2h_k + 1$, $h_t = 1$, and $t = \lfloor \log_2 n \rfloor - 1$

We will use `malloc()` to allocate the appropriate number of strides for the second of the choices above. Here is an implementation of Shell sort.

```
#include "sort.h"
#include <stdlib.h>

static long logb2(long size) {
    long power = 1;         /* overcompensate by 1 */
    unsigned long mask = 1;

    while ((unsigned long)size > mask) {
        mask *= 2;
        power++;
    }
    return power;
}

void sort(void *a[], long size, int (*cmp)(void *v1, void *v2)) {
    long t = logb2(size) - 1;
    long *h = (long *)malloc(t*sizeof(long));
    long i;

    h[t-1] = 1;
    for (i = t - 2; i >= 0; i--)
        h[i] = 2 * h[i+1] + 1;
    for (i = 0; i < t; i++) {
        long stride = h[i];
        long j;
        for (j = stride; j < size; j++) {
            long k;
            for (k = j - stride; k >= 0; k -= stride) {
                if ((*cmp)(a[k+stride], a[k]) >= 0)
```

```
                        break;
                    else {
                        void *tmp = a[k];
                        a[k] = a[k+stride];
                        a[k+stride] = tmp;
                    }
                }
            }
        }
        free(h);
    }
```

13.2.2.2 Heap sort

The fundamental characteristic of our previous selection sort was, at each step, to scan the source sequence to find the minimum value to exchange with the next item in the destination sequence. Finding the minimum value was $O(n)$, and since we had to do that scan for each index in our array, we ended up with an algorithm that was $O(n^2)$ complexity. If we could reduce the complexity for finding the minimum value in the source sequence, we could reduce the complexity of our sort algorithm.

In Section 12.3.3, we discussed a heap structure that enabled us to obtain an element with the minimum value from a bag of numbers in $O(\log n)$ time. If we can figure out how to use the heap structure, we should end up with an algorithm that is $O(n \log n)$ complexity.

The discussion in Section 12.3.3 had two functions, `siftup()` for re-establishing the heap property when adding a new element to the array, and `siftdown()` for re-establishing the heap property after removal of the smallest item. In this case, we are presented with an already-populated array, and must establish the heap property for the items that are already there. The logic is similar, but will require some additional machinations in order to make it work; fortunately, the return for this extra effort, $O(n \log n)$ complexity, makes it worth it.

Recall that our heap properties are:

- *order* - the value at any node is less than or equal to the values of the node's children; and
- *shape* - the terminal nodes are on at most two levels, with those on the bottom level as far left as possible.

The first property guarantees that the minimum element of the heap is always the root; thus, if we can move the items in our array such that the heap properties are satisfied, we have $O(1)$ access to the minimum value.

Also recall that if we stored our `N` data items in an array using indices `1..N`, that given a node at index `i`, its left child is at index `2*i`, its right child is at index `2*i + 1`, and its parent is at index `i/2` (using integer division). We are given an array indexed from

0..N-1, so it will require some additional, straight-forward arithmetic to be able to generate a heap *in situ*.

Suppose that we have an array b[] indexed from 1..N, and consider index m = N/2 + 1, where we are using integer division. b[m]..b[N] satisfy the heap property already, since no two indices k and l exist such that l = 2*k or l = 2*k + 1; these items essentially form the bottom row of the heap.

Now extend the subarray under consideration by one to the left; we need to "sift" this new value into the appropriate place in the subarray b[m-1]..b[N] such that the subarray satisfies the heap properties. This sifting is very much like the siftdown() helper function in the heap-based implementation of our priority queue ADT in Section 12.3.3.2. We continue to extend the array to the left and sift until b[1]..b[N] is a heap.

Here is the sift function to establish the heap property for b[L]..b[R].

```
void sift(void *b[], long L, long R, int (*cmp)(void*,void*)) {
    long i = L, j = 2 * L;
    void *x = b[L];

    if (j < R && (*cmp)(b[j+1], b[j]) < 0)
        j++;
    while (j <= R && (*cmp)(b[j], x) < 0) {
        b[i] = b[j];
        b[j] = x;
        i = j;
        j *= 2;
        if (j < R && (*cmp)(b[j+1], b[j]) < 0)
            j++;
    }
}
```

What do we do after we have the entire array satisfying the heap properties? We know that b[1] is the minimum value of the array. We can, therefore, swap b[1] with b[N], and then call sift() to re-establish the heap properties for b[1]..b[N-1]. We repeat this until we are left with b[1]..b[1], which obviously satisfies the heap properties.

Unfortunately, this leaves us with our array sorted but in the reverse order to what we expected. If we change the direction of the ordering relations in sift(), such that b[1] is the largest item in the heap, we will have sorted our array *in situ* from smallest to largest. This yields the following version of sift() for 1-based arrays.

```
void sift(void *b[], long L, long R, int (*cmp)(void*,void*)) {
    long i = L, j = 2 * L;
    void *x = b[L];

    if (j < R && (*cmp)(b[j], b[j+1]) < 0)
```

```
            j++;
        while (j <= R && (*cmp)(x, b[j]) < 0) {
            b[i] = b[j];
            b[j] = x;
            i = j;
            j *= 2;
            if (j < R && (*cmp)(b[j], b[j+1]) < 0)
                j++;
        }
    }
```

How do we make this work with 0-based arrays? This is actually quite straight-forward. We have the correct logic above if the indices run from `1..N`. Whenever we actually access an element of the array, our index will be too large by `1`. Thus, we define a macro, `ind(val)`, which is invoked whenever we are accessing an element of the array. Here is the final version of heapsort.

```
    #include "sort.h"

    #define ind(val) ((val)-1)    /* macro to compute correct index */

    static void sift(void *a[], long L, long R, int(*cmp)(void*,void*)) {
        long i = L, j = 2 * L;
        void *x = a[ind(L)];

        if (j < R && (*cmp)(a[ind(j)], a[ind(j+1)]) < 0)
            j++;
        while (j <= R && (*cmp)(x, a[ind(j)]) < 0) {
            a[ind(i)] = a[ind(j)];
            a[ind(j)] = x;
            i = j;
            j *= 2;
            if (j < R && (*cmp)(a[ind(j)], a[ind(j+1)]) < 0)
                j++;
        }
    }

    void sort(void *a[], long size, int(*cmp)(void *v1, void *v2)) {
        long L = size / 2 + 1, R = size;

        while (L > 1) {
            L--;
            sift(a, L, R, cmp);
        }
        while (R > 1) {
            void *x = a[ind(1)];
            a[ind(1)] = a[ind(R)];
            a[ind(R)] = x;
            R--;
            sift(a, L, R, cmp);
        }
    }
```

Have we achieved $O(n \log n)$ complexity? The initial construction of the heap *in situ* requires $\frac{n}{2}$ calls to `sift()`; each call to `sift()` requires a maximum of $\log n$ comparisions and swaps; thus, this part of the algorithm is $O(n \log n)$. The ordering of the array in situ requires n swaps and n calls to `sift()`, which again yields $O(n \log n)$. Thus, the entire algorithm is $O(n \log n)$.

As with the basic selection sort, because we swap values over long distances, both in heap construction and in the assembly of the sorted array, we may invert the order of items that have identical keys. As a result, heapsort is *not* a stable sorting algorithm.

13.2.2.3 Quick sort

We have shown advanced sorting methods based upon the principles of insertion and selection, so it is no surprise that the third improved method is based upon the principle of exchange. Bubblesort exchanged adjacent pairs of items, “bubbling” the smallest remaining value in the source sequence into the next location in the destination sequence. We might expect that if we exchanged pairs of items over large distances, we would see a performance improvement.

Consider the following algorithm: pick any item, `x`, at random - we call this item the *pivot*; scan the array from the left until an item `a[i] > x` is found, and then scan from the right until an item `a[j] < x` is found. Now exchange these two items and continue this “scan and swap” process until the two scans meet somewhere in the middle of the array. The array is now partitioned into a left part with keys less than (or equal to) `x`, and a right part with keys greater than (or equal to) `x`. If we change the relations to $\geq$ and $\leq$, then `x` acts as a sentinel terminating the two scans. This leads us to the following function to partition our array in this way.

```
void partition(void *a[], long size, int (*cmp)(void*, void*)) {
    void *x;
    long i = 0, j = size-1;

    /* select the pivot from a[], assigning it to x */
    while (i <= j) {
        while ((*cmp)(a[i], x) < 0)
            i++;
        while ((*cmp)(x, a[j]) < 0)
            j--;
        if (i <= j) {
            void *y = a[i];
            a[i] = a[j];
            a[j] = y;
```

```
                i++;
                j--;
            }
        }
    }
```

So, how do we “select the pivot from `a[]`, assigning it to `x`”? Usually we compute the index for the middle element in the array, and use its value for `x`. This works quite well if the items are randomly ordered in the array. This also works very well for arrays that are initially sorted in decreasing value - $\frac{n}{2}$ pairs are swapped around the middle value, and we are done. One could choose `a[0]` or `a[size-1]` as the pivot value; for random initial order, they work well, but if the array is already sorted, it leads to worst case $O(n^2)$ complexity. We will use the middle element for the *pivot*.

Here is a recursive implementation of Quicksort.

```
#include "sort.h"

static void qsort(void *a[], long L, long R, int(*cmp)(void*, void*)) {
    long i = L, j = R;
    void *x = a[(L+R)/2];

    while (i <= j) {
        while ((*cmp)(a[i], x) < 0)
            i++;
        while ((*cmp)(x, a[j]) < 0)
            j--;
        if (i <= j) {
            void *y = a[i];
            a[i] = a[j];
            a[j] = y;
            i++;
            j--;
        }
    }
    if (L < j)
        qsort(a, L, j, cmp);
    if (i < R)
        qsort(a, i, R, cmp);
}

void sort(void *a[], long size, int (*cmp)(void *v1, void *v2)) {

    qsort(a, 0L, size-1, cmp);
}
```

What is the running time complexity for Quicksort? Since we are doing a “divide and conquer” approach to the exchange of items, as long as we choose a *pivot* that approximately divides the array into two equal sized subarrays at each recursion, we will perform $\log n$ of these recursions. Since we have to compare each item against the pivot, this comparison activity will be $O(n)$. Thus, in the normal case, we expect the complexity to be $O(n \log n)$. Even if the choice of pivot does not lead to equal sized subarrays, as long as the pivot is chosen at random, we still achieve $O(n \log n)$.

What happens if we make a bad choice for the pivot? For example, if we use `a[0]` as the pivot and the array is initially sorted, we are left with a left partition of a single element, and a right partition with `n-1` items. The result is that we will recurse to a depth of `n`, instead of `log\,n`, leading to a worst-case performance of $O(n^2)$.

We might opt for an iterative Quicksort implementation to avoid the call frame overhead of the recursive implementation. In order to solve this iteratively, we must maintain a list of partitioning requests that have yet to be performed. After each step, two partitioning tasks must be performed; only one of them can be performed in the next iteration, so we must store the other partitioning task away on the list. The tasks on the list must be processed in a *last in, first out* sequence - thus, we must implement a stack representing the partitioning tasks remaining to be performed.

How big should this stack be? Given the worst-case performance for a bad choice of the pivot, we have previously argued that the recursion depth would be `n`, which means that we might need a stack of that size. There is a way out of this - if we always stack the larger of the two subarrays, and process the smaller in the next iteration, it can be shown that the stack need never be larger than $\log n$ in size. We will use `malloc()` to allocate the appropriately sized stack in our iterative Quicksort implementation.

```
#include "sort.h"
#include <stdlib.h>
#include <stdio.h>

typedef struct pStruct {
    long left;
    long right;
} PStruct;

static long logb2(long size) {
    long power = 1;         /* overcompensate by 1 */
    unsigned long mask = 1;

    while ((unsigned long)size > mask) {
        mask *= 2;
        power++;
    }
    return power;
}

void sort(void *a[], long size, int (*cmp)(void *v1, void *v2)) {
    PStruct *st = (PStruct *)malloc(logb2(size)*sizeof(PStruct));
    int sp = 0;

    st[sp].left = 0;
    st[sp].right = size - 1;
```

```
        while (sp >= 0) {
            long L = st[sp].left;
            long R = st[sp].right;
            sp--;
            while (L < R) {
                long i = L;
                long j = R;
                void *x = a[(L+R)/2];
                while (i <= j) {
                    while ((*cmp)(a[i], x) < 0)
                        i++;
                    while ((*cmp)(x, a[j]) < 0)
                        j--;
                    if (i <= j) {
                        void *y = a[i];
                        a[i] = a[j];
                        a[j] = y;
                        i++;
                        j--;
                    }
                }
                if ((j - L) < (R - i)) {
                    if (i < R) {         /* stack right partition */
                        sp++;
                        st[sp].left = i;
                        st[sp].right = R;
                    }
                    R = j;               /* continue with left partition */
                } else {
                    if (L < j) {         /* stack left partition */
                        sp++;
                        st[sp].left = L;
                        st[sp].right = j;
                    }
                    L = i;               /* continue with right partition */
                }
            }
        }
        free(st);                        /* return the stack to the heap */
    }
```

13.2.3 Merge sort

In Section 13.2.2, we showed three different advanced sorting algorithms, all of which outperformed the simple sorting algorithms in Section 13.2.1, but *none* of which were stable. There are numerous occasions where a stable sorting algorithm is required, and usually at a scale where $O(n^2)$ simply will not work. This section describes *Mergesort*, an algorithm that yields $O(n \log n)$ complexity, but at the cost of $O(n)$ extra storage locations for a scratch array.

Mergesort is another "divide and conquer" algorithm. The algorithm has three steps:

1. **divide**: if the input size is smaller than a certain threshold, solve the problem directly using a straightforward *stable* method and return the solution; otherwise, divide the input data into two or more disjoint subsets;
2. **recurse**: recursively solve the problem for each of the subsets; and
3. **merge**: merge the solutions to the subproblems into a solution to the original problem.

We have our items in an array, `a[]`, and a comparator function that defines a total order over those items. Let's define a sequence `S` as a sub-array of `a[]`, defined by `low` and `high` indices, and which has `n = high - low + 1` items. To sort the `n` items in `S`, our Mergesort algorithm implements the above three steps as follows:

1. if `n` is `0` or `1`, return `S` immediately, as it is already sorted; otherwise (`n >= 2`), divide `S` into two disjoint subsequences, $\mathtt{S}_{\mathtt{low}}$ and $\mathtt{S}_{\mathtt{high}}$; each subsequence contains ~half of the elements of `S`;
2. recursively sort sequences $\mathtt{S}_{\mathtt{low}}$ and $\mathtt{S}_{\mathtt{high}}$;
3. merge the sorted sequences $\mathtt{S}_{\mathtt{low}}$ and $\mathtt{S}_{\mathtt{high}}$ back into `S`.

In order to perform the merge in step 3 above, we need a scratch array of size `n`; the way that we will handle this in our implementation of Mergesort is to use `malloc()` to allocate a scratch array of the `size` specified in the call to `sort()`; when merging the two sorted subsequences, defined by `L`, `M`, and `H` indices, `scratch[L..H]` is used. Obviously, after the sort is completed, the scratch array is returned to the heap.

Here is a recursive implementation of Mergesort.

```
#include "sort.h"
#include <stdlib.h>

static void **scratch = NULL;

static void merge(void *a[], long L, long M, long H, int (*cmp)(void*,void*)) {
    long l1 = L, l2 = M + 1, i;

    for (i = L; l1 <= M && l2 <= H; i++) {
        if((*cmp)(a[l1], a[l2]) <= 0)
            scratch[i] = a[l1++];
        else
            scratch[i] = a[l2++];
    }
    while (l1 <= M)
        scratch[i++] = a[l1++];
    while (l2 <= H)
        scratch[i++] = a[l2++];
    for (i = L; i <= H; i++)
        a[i] = scratch[i];
}

static void msort(void *a[], long low, long high, int(*cmp)(void*, void*)) {

    if (low < high) {
        long mid = (low + high) / 2;
        msort(a, low, mid, cmp);
        msort(a, mid+1, high, cmp);
        merge(a, low, mid, high, cmp);
    }
}

void sort(void *a[], long size, int (*cmp)(void *v1, void *v2)) {
    scratch = (void **)malloc(size*sizeof(void *));
    msort(a, 0L, size-1, cmp);
    free(scratch);
}
```

Note that our interpretation of the **divide** step is to keep dividing until the size of the sequence is 0 or 1. A standard adaptation of Mergesort is to define a threshold size for the sequence, say 7, and to use insertion sort to sort the sequence in place if the size of the sequence is less than that threshold. The following code shows the modified `msort()` along with the definition of the threshold and the code for insertion sort (as already seen in Section 13.2.1.1 but with the name changed to `isort()`.

```
static void isort(void *a[], long size, int (*cmp)(void *v1, void *v2)) {
    long i;

    for (i = 1; i < size; i++) {
        void *x = a[i];
        long j;

        for (j = i - 1; j >= 0 && (*cmp)(a[j], x) > 0; j--)
            a[j+1] = a[j];
        a[j+1] = x;
    }
}

#define THRESHOLD 7
static void msort(void *a[], long low, long high, int(*cmp)(void*, void*)) {
    long size = high - low + 1;

    if (size > THRESHOLD) {
        long mid = (low + high) / 2;
        msort(a, low, mid, cmp);
        msort(a, mid+1, high, cmp);
        merge(a, low, mid, high, cmp);
    } else if (size > 1) {
        isort(a+low, size, cmp);
    }
}
```

Chapter 14

Maps

A *map* stores (key, value) pairs, typically called *entries*. Each key in a map is required to be unique. Thus, the key in an entry is a unique identifier for the entry.

The most general type of map allows the keys and the values to be any type of data. A function must exist to enable the Map implementation to determine if two key instances are equal in order to be able to support the uniqueness criterion stated above.

We will focus on maps for which the keys are C character strings, as these are very common. At the end of the chapter we will show the changes that are required to support maps for which the keys can be any data type.

14.1 The Map interface

A map is a container for (key, value) pairs. The Map interface is defined as follows:

```
#ifndef _MAP_H_
#define _MAP_H_

#include "iterator.h"               /* needed for factory method */

typedef struct map Map;             /* forward reference */
typedef struct mentry MEntry;       /* opaque type definition */

const Map *Map_create(void);
struct map {
    void *self;
    void (*destroy)(const Map *m, void (*freeFxn)(void *value));
    void (*clear)(const Map *m, void (*freeFxn)(void *value));
    int (*containsKey)(const Map *m, char *key);
    int (*get)(const Map *m, char *key, void **value);
    int (*put)(const Map *m, char *key, void *value, void **previous);
    int (*putUnique)(const Map *m, char *key, void *value);
    int (*remove)(const Map *m, char *key, void **value);
    long (*size)(const Map *m);
    int (*isEmpty)(const Map *m);
    char **(*keyArray)(const Map *m, long *len);
    MEntry **(*entryArray)(const Map *m, long *len);
    const Iterator *(*itCreate)(const Map *m);
};

char *mentry_key(MEntry *me);
void *mentry_value(MEntry *me);

#endif /* _MAP_H_ */
```

The semantics of the constructor and the methods are as follows:

- `const Map *Map_create(void);`
  create a new instance of a map; returns a pointer to the dispatch table, or `NULL` if there were heap-allocation errors;
- `void (*destroy)(const Map *m, void (*freeFxn)(void *value));`
  destroy the map; if `freeFxn != NULL`, this function is invoked on each value in the map; then all heap-allocated memory associated with the map is freed;
- `void (*clear)(const Map *m, void(*freeFxn)(void * value));`
  purge all entries from the map; if `freeFxn != NULL`, this function is invoked on each value in the map; then heap-allocated memory associated with the the entries in the map is freed; upon return, the map is empty;
- `int (*containsKey)(const Map *m, char *key);`
  returns 1 if the map has an entry for `key`, 0 otherwise;
- `int (*get)(const Map *m, char *key, void **value);`
  returns the value associated with `key` in `*value`; if an entry was found, returns 1; returns 0 if no mapping for `key`;
- `int (*put)(const Map *m, char *key, void *value, void **previous);`
  associates `value` with `key`; if this replaces an existing mapping, the previous value is returned in `*previous`; otherwise, returns `NULL` in `*previous`; returns 1 if successful, 0 if not;
- `int (*putUnique)(const Map *m, char *key, void *value);`
  associates `value` with `key`; fails if `key` is already present; returns 1 if successful, 0 if not;
- `int (*remove)(const Map *m, char *key, void **value);`
  removes the entry associated with `key` if one exists; returns value associated with `key` in `*value`; returns 1 if successful, 0 if no entry associated with `key`;
- `long (*size)(const Map *m);`
  return the number of entries in the map;

- `int (*isEmpty)(const Map *m);`
  return 1 if the map is empty, 0 otherwise;

- `char **(*keyArray)(const Map *m, long *len);`
  returns an array containing all of the keys in the map in an arbitrary order; returns the length of the array in `*len`; returns a pointer to the `char *` array of keys, or `NULL` if a heap-allocation failure occurred; **N.B.** the caller is responsible for freeing the `char *` array when finished;

- `MEntry **(*entryArray)(const Map *m, long *len);`
  returns an array containing all of the entries in the map in arbitrary order; returns the length of the array in `*len`; returns a pointer to `MEntry *` array of elements, or `NULL` if a heap-allocation failure occurred; **N.B.** the caller is responsible for freeing the `MEntry *` array when finished; the `key` and `value` of each entry can be obtained by invoking `mentry_key()` and `mentry_value()` on each `MEntry *`;

- `const Iterator *(*itCreate)(const Map *m);`
  creates an iterator for running through the entries in the map; returns a pointer to the Iterator or NULL if there were heap-allocation problems; each invocation of `next()` on the iterator will yield a `MEntry *`, for which the caller should use `mentry_key()` and `mentry_value()` as needed.

14.2 A linked list-based Map implementation

A simplistic approach to implementing a Map would be to store the entries in a linked list. Since the interface supports removal of an entry corresponding to an arbitrary `key`, it would be wise to use a doubly-linked list. Let's look at how we would implement `containsKey()`, `get()`, `put()`, `putUnique()`, and `remove()`.

14.2.1 The necessary data structures and a helper function

First, we need to define `struct mentry` from the header file. Then we need to define the node structure in the doubly-linked list. Finally, we need to define the instance-specific data structure. These are shown below.

In addition, we will define a helper function that scans the doubly-linked list for an entry matching `key`; if found, it returns its `Node *` address; if not found, it returns `NULL`.

```
struct mentry {         /* the representation of an entry */
    char *key;
    void *value;
};
typedef struct node {  /* a node in the DLL */
    struct node *next;
    struct node *prev;
    struct mentry entry;
} Node;
typedef struct mdata { /* instance-specific data structure */
    long size;
    Node sentinel;
} MData;

static Node *findNode(MData *md, char *key) {
    Node *p;

    for (p = md->sentinel.next; p != &(md->sentinel); p = p->next)
        if (strcmp(key, p->entry.key) == 0)
            return p;
    return NULL;
}
```

14.2.2 `containsKey()`

This method is particularly trivial, given that we have defined `findNode()`.

```
static int m_containsKey(const Map *m, char *key) {
    MData *md = (MData *)m->self;
    return (findNode(md, key) != NULL);
}
```

14.2.3 `get()`

Armed with `findNode()` defined above, the code for the `get()` method is straightforward.

```
static int m_get(const Map *m, char *key, void **value) {
    MData *md = (MData *)m->self;
    Node *p = findNode(md, key);

    if (p == NULL)
        return 0;
    else {
        *value = p->entry.value;
        return 1;
    }
}
```

14.2.4 put() and putUnique()

These methods differ only in the action they take if a call to `findNode(key)` returns a non-NULL value; for `putUnique()`, it returns a failure indication; for `put()`, it proceeds to replace the current value with the new one. Therefore, we will define a helper function that encapsulates the common logic. We also use the `link()` helper function from the Deque implementation.

```
static int putEntry(MData *md, char *key, void *val, void **prev, int uniq) {
    Node *p = findNode(md, key);
    status = 0;           /* assume failure */

    if (p != NULL) {
        if (! unique) {
            status = 1;
            *prev = p->entry.value;
            p->entry.value = val;
        }
    } else {
        p = (Node *)malloc(sizeof(Node));
        if (p != NULL) {
            char *s = strdup(key);
            if (s != NULL) {
                status = 1;
                p->entry.key = s;
                p->entry.value = val;
                *prev = NULL;
                link(md->sentinel.prev, p, &(md->sentinel)); /* from Deque */
            } else {
                free(p);
            }
        }
    }
    return status;
}

static int m_put(const Map *m, char *key, void *value, void **previous) {
    MData *md = (MData *)m->self;
    return putEntry(md, key, value, previous, 0);
}

static int m_putUnique(const Map *m, char *key, void *value) {
    MData *md = (MData *)m->self;
    return putEntry(md, key, value, previous, 1);
}
```

Note that when we add a new node to the list that we make a copy of the key on the heap using `strdup()`; you might think that we could just have our entry point to the `key` passed to us by the caller. The caller is under no obligation to maintain that key value at that address over time; in order to implement the map semantics, we must make a copy of the `key`.

14.2.5 `remove()`

Armed with `findNode()`, the implementation of this method is straightforward. Note that we use the `unlink()` function from the Deque implementation here, as well.

```
int m_remove(const Map *m, char *key, void **value) {
    MData *md = (MData *)m->self;
    Node *p = find Node(m, key);
    int status = 0;

    if (p != NULL) {
        status = 1;
        *value = p->entry.value;
        free(p->entry.key);
        unlink(p);
        free(p);
    }
    return status;
}
```

14.2.6 Running time complexity of these methods

Since each of these methods invokes `findNode()`, their running time complexities will be the same as that for `findNode()`. `findNode()` searches through each entry in the list, returning if it finds an entry with that `key`, or going through the entire list to determine that no such entry exists. In either situation, `findNode()` has to touch some fraction of the entire list, and it thus has a running time complexity of $O(n)$. If we wish to store a large number of entries in a Map, it will be very costly in running time.

You might think that if we kept the entries in the list sorted by key that it would affect the running time complexity. Unfortunately, it will have no effect on the complexity, since the nodes are not contiguous in memory; if they were, we could use a binary search, reducing the complexity to $O(\log n)$.

14.3 Hash tables to the rescue

We would like a data structure that can be used to implement a map such that each of the essential Map methods are typically $O(1)$ running time complexity; the worst case complexity should certainly be no worse that for a list-based implementation, $O(n)$. A very efficient way to implement a map is to use a *hash table*.

A hash table consists of two components: an array for organizing the storage of (key, value) pairs, and a *hash function*. The hash function, which we will discuss a little later, maps from a `key` to an integer in the range [0..N-1], where N is the size of the array. For a given `key`, `hash(key)` returns an index into the array at which the `value` associated with that `key` is "stored" - i.e., it is either at that index or nearby.

Two types of organizing arrays are commonly used:

- for *open hashing*, the array contains the (key, value) pairs themselves; and
- for *bucket hashing*, the array contains the header of a LIFO linked list of `value`s associated with `key`s that hash to that index.

In this textbook, we will use bucket hashing; your next course on data structures and algorithms will expose you to open hashing.

14.3.1 The bucket array

The instance-specific data structure for a hash table consists of an integer `N`, which is the size of the array of listheads, and the array of listheads, often called the *bucket array*. Each element of the array can be thought of as a "bucket" containing all (key, value) pairs for which `hash(key)` returns the index for that cell; the number of array elements is sometimes called the *capacity* of the array.

Given a hash function, an insertion into our hash table requires that we

1. compute the index into the bucket array by invoking the hash function on the key;
2. look at each node in the bucket to see if there is already an entry with the specified key;
3. if so, either replace the value (`put()`) or return failure (`putUnique()`);
4. if not, generate a node for insertion at the beginning of the linked list at that index.

`containsKey()` and `get()` require that we

1. compute the index into the bucket array by calling the hash function on the key;
2. look at each node in the bucket to see if there is an entry with the specified key;
3. if so, either return 1 (`containsKey()`) or return the corresponding value (`get()`);

4. if not, return 0.

`remove()` requires that we

1. compute the index into the bucket array by calling the hash function on the key;
2. look at each node in the bucket to see if there is an entry with the specified key; **N.B.** while scanning the linked list of nodes in the bucket, we must remember the previous node in order to be able to remove the matching node from the list;
3. if so, unlink the matching node and free heap-allocated storage;
4. if not, return 0.

Thus, we can see that the running time complexity for these methods depends heavily upon how well the hash function distributes the `key`s over the N buckets. Let's look at some limiting cases.

- Suppose that `hash(key)` always returns the value 0; in this case, our hash table simply degenerates into a single LIFO linked list. We already know that the running time complexity for these five methods is $O(n)$ for such an implementation. Obviously, such a hash function is not a "good" one.
- Suppose that `N > M` (number of items to be stored in the table). A *perfect* hash function would generate no collisions - i.e., each of the buckets would contain 0 or 1 entry. If we had a perfect hash function, the running time complexity for these five methods would be $O(1)$; this is the goal to which we should strive.
- Even if `M > N`, a "good" hash function would evenly distribute the items over the `N` listheads, such that these five methods would display running time complexity of $O(M/N)$.

It is this last type of hash function that we will explore in the next section.

14.3.2 Hash functions

A "good" hash function is a function that:

- when applied to a `key`, returns a number;
- when applied to `key`s that are *equal*, returns the same number for each;
- when applied to `key`s that are *unequal*, is *very unlikely* to return the same number for each.

When two or more unequal `key`s hash to the same value, this is called a *collision*. Since two or more entries can be in a single bucket, it means that we must store the (key, value) pairs in the bucket, and when looking for a particular `key`, we first hash to the bucket, and then compare our `key` with the `key` stored with each entry in the bucket.

Hash functions are often described as having two phases:

1. mapping the `key` to an unsigned integer, and
2. converting that integer to an index in the range `[0..N-1]`.

Typically, the conversion step is achieved by applying the modulus function (`%`) to the unsigned integer from the first step.

Since we are focused on Maps for which the keys are character strings, let's look at some different hash functions and how well they distribute keys over the bucket array. Each of the hash functions will conform to the following signature:

```
long hashx(char *key, long N);
```

14.3.2.1 Summing characters

The simplest approach to a hash function for character strings is to simply sum the values of the ASCII characters in the string.

```
long hash1(char *key, long N) {
    unsigned long sum = 0L;
    char *p;

    for (p = key; *p != '\0'; p++)
        sum += (unsigned long)(*p);
    return (long)(sum % N);
}
```

If `N` is small compared to the sums that are generated, this function should do a reasonable job distributing the keys evenly across the bucket array. If `N` is large compared to the sums that are generated, this function will do a very poor job of key distribution. For example, consider words in a dictionary where all letters are lower case. The ASCII value for `'a'` is 97, and for `'z'` is 122. A dictionary file on Linux contains 124,801 words, with the average word length being 7.91 characters per word. Assuming 8 characters for the sake of simplicity, the sums generated for these words will fall predominantly in the range of 776 .. 976. If we assume even distribution across the buckets, with 124 thousand words we would probably want 200 thousand buckets; this hash function would put most of the words in a very small number of buckets relative to `N`. Thus, this is not a very good hash function.

14.3.2.2 Polynomial hashing

The previous hash function ignores character order. Perhaps we can do a better job if we can somehow take character order into account.

A character string `s` can be viewed as a tuple of the form

$$(s_0, s_1, ..., s_{m-1})$$

where m is the number of characters in the string.

If we pick a positive integer $a \neq 1$, we can compute an integer representing the string using the following polynomial form:

$$s_0 a^{m-1} + s_1 a^{m-2} + \ldots + s_{m-2} a + s_{m-1}.$$

We can obviously expand this polynomial form into:

$$s_{m-1} + a(s_{m-2} + a(s_{m-3} + \ldots + a(s_2 + a(s_1 + a s_0)) \ldots)).$$

Here is an implementation of a hash function using this polynomial form:

```
#define A 31L
long hash2(char *key, long N) {
    unsigned long sum = 0L;
    char *p;

    for (p = key; *p != '\0'; p++)
        sum = A * sum + (unsigned long)(*p);
    return (long)(sum % N);
}
```

If you think about the bit representation of an integer, multiplying by a positive number tends to move the bits to the left - i.e. to higher order bits; in fact, if `A == 32`, this would correspond exactly to shifting `sum` 5 bits left in memory. Thus, multiplication by `A` mixes more of the summation into higher order bits.

Choice of the value for `A` requires some experimentation. We show the number of collisions that occur as a function of the value of `A` for a dictionary file below.

In general, it is a good idea if `A` and `N` are mutually prime, as this improves the distribution of strings over the bucket array. It is common for `N` to be a power of 2; therefore, one should choose `A` to be a prime number. In our implementation of a Map using a hash table, we will also choose `N` to be prime, as well.

14.3.2.3 Cyclic shift hashing

C provides syntactic support for shifting variables by an integer number of bits. Even though we did not cover this in Chapter 4, we will do so now, as the next type of hash function requires this capability.

If n is an integer type, then $n << 2$ shifts the value of n by 2 bits left, equivalent to multiplying n by 4; the vacated low order bits are filled with 0. The two high order bits of n are shifted out of the variable.

$n >> 3$ shifts the value of n by 3 bits to the right, equivalent to division by 8; if n is signed, fill the vacated bits with the sign bit; if unsigned, fill with 0.

$n \mid 0x11$ sets bits 0 and 4 in n to 1, leaving all others alone.

Armed with these bit operations, we can now define a cyclic shift hash function.

```
#define B 5
unsigned long rotate(unsigned long value) {
    static int bits = 8 * sizeof(unsigned long);
    unsigned long ans = value;

    if (B != 0)
        ans = (value << B) | (value >> (bits - B));
    return ans;
}

long hash3(char *key, long N) {
    unsigned long sum = 0L;
    char *p;

    for (p = key; *p != '\0'; p++)
        sum = rotate(sum) + (unsigned long)*p;
    return (long)(sum % N);
}
```

Once again, our hash function has a free paramenter, `B`, which is the number of bits by which the current sum should be rotated; we will need to tune this value. In the next section we show how the number of collisions varies with the value of `B`.

14.3.2.4 Performance of these hash functions

As described earlier, we have a dictionary file that contains 124,801 words, all in lower case. We have tested the three hash functions on the words in that file, noting the following collision statistics: total, minimum, maximum, mean, median, and standard deviation. For `hash2()` and `hash3()`, we quote these statistics for different values of `A` and `B`, respectively.

Function	Total	Minimum	Maximum	Mean	Median	Std Dev
hash1	123,237	0	639	78.8	18	128.3
hash2(A=1)	123,237	0	639	78.8	18	128.3
hash2(A=7)	6,072	0	6	0.0522	0	0.261
hash2(A=19)	806	0	2	0.00650	0	0.0812
hash2(A=29)	642	0	1	0.00517	0	0.0717
hash2(A=31)	642	0	1	0.00517	0	0.0717
hash2(A=37)	642	0	1	0.00517	0	0.0717
hash2(A=73)	642	0	1	0.00517	0	0.0717
hash2(A=129)	642	0	1	0.00517	0	0.0717
hash2(A=257)	642	0	1	0.00517	0	0.0717
hash2(A=511)	642	0	1	0.00517	0	0.0717
hash3(B=0)	123,237	0	639	78.8	18	128.3
hash3(B=1)	91,185	0	51	2.71	0	5.21
hash3(B=3)	4,432	0	5	0.0368	0	0.208
hash3(B=5)	642	0	1	0.00517	0	0.0717
hash3(B=7)	642	0	1	0.00517	0	0.0717
hash3(B=9)	655	0	1	0.00528	0	0.0724
hash3(B=13)	1,114	0	3	0.00901	0	0.0972
hash3(B=17)	702	0	2	0.00566	0	0.0752
hash3(B=21)	7,800	0	5	0.0667	0	0.287
hash3(B=25)	821	0	2	0.00662	0	0.0818
hash3(B=29)	642	0	1	0.00517	0	0.0717

Note that the statistics are the same for `hash1()`, `hash2(A=1)`, and `hash3(B=0)`; this is exactly as expected.

Also note that with a file this large, these hash functions cannot achieve anything less than 642 total collisions. The worst case is for `hash1()`, where the largest bucket has 639 entries, and the average number of collisions is 78.8. It is clear that `hash1()` is a very poor choice for a task such as this.

The variation of `hash2()` collision statistics as a function of `A` shows that for even small values of `A`, the maximum drops quite rapidly. By the time that `A = 29`, we have reached the best collision statistics for this set of data; for values of `A` larger than that value, the overall statistics remain the same.

The variation of `hash3()` collistion statistics as a function of `B` shows that a value of `B = 5` gives us the same overall statistics as `hash2()` with a value of `A = 31`; not particularly surprising, since rotating left by 5 bits is the same as multiplying by 32. For larger `B` values, the overall statistics become ever so slightly worse.

Given these results, we will use a `hash2()` function in our Map implementation, with `A = 31`.

14.4 A hash table-based Map implementation

Before we work our way through a hash table-based Map implementation, we need to revisit the interface, in particular, the constructor.

14.4.1 Revised constructor function signature

As you will recall from section 14.1, the function signature for the constructor was

```
const Map *Map_create(void);
```

This was fine for a linked list-based implementation, as we simply grew the list using heap-allocated storage as we needed it. A hash table, on the other hand, needs to know what size of bucket array to create; it could start with a default size, and then resize when performance required; alternatively, the user could specify a *capacity* argument to the constructor so that the implementation starts out at an appropriate size for the problem at hand. An implementation that does not require that initial capacity can simply ignore it.

Another performance aspect of a hash table is the ratio of the number of entries in the table divided by the number of buckets - this is usually called the *load factor*. It should be apparent that once the load factor gets large, each bucket is a substantial linked list, and the table performance will suffer. We would like to enable the user to specify a load factor in the call to the constructor; if an addition to the table causes the current load factor to exceed that value, the bucket array should be made larger, and the entries should be redistributed in the new, larger array. Again, an implementation that cannot use this information can just ignore it.

As a result, we will modify the signature for the constructor to

```
const Map *Map_create(long capacity, double loadFactor);
```

where `capacity` is the initial number of buckets to create for the underlying hash table; if `capacity == 0L`, then a default capacity will be used. Similarly, `loadFactor` is a floating point number indicating the preferred load factor at which the table should operate; if `loadFactor <= 0.0`, a default preferred value will be used. If an addition causes the current load factor to exceed this preferred value, the bucket array is resized to bring the load factor under the preferred value.

14.4.2 The preliminaries

Besides the usual includes and structure definitions, we also include our version of `hash2()` using `A = 31`.

```
/* BSD Header removed to conserve space */

#include "map.h"
#include <stdlib.h>
#include <string.h>

#define DEFAULT_CAPACITY 16
#define MAX_CAPACITY 134217728L
#define DEFAULT_LOAD_FACTOR 0.75
#define TRIGGER 100 /* number of changes that will trigger a load check */

typedef struct m_data {
    long size;
    long capacity;
    long changes;
    double load;
    double loadFactor;
    double increment;
    MEntry **buckets;
} MData;

struct mentry {
    struct mentry *next;
    char *key;
    void *value;
};

/*
 * generate hash value from key; value returned in range of 0..N-1
 */
#define SHIFT 31L /* should be prime */
static long hash(char *key, long N) {
    unsigned long ans = 0L;
    char *sp;

    for (sp = key; *sp != '\0'; sp++)
        ans = SHIFT * ans + (unsigned long)*sp;
    return (long)(ans % N);
}
```

We define a default capacity as a small value (16). We also define a maximum capacity beyond which the bucket array will not grow. The default load factor implies that we want to keep the total number of entries in the table at 75% of the number of buckets. There is nothing that stops you from setting the target load factor as a number like 2.0, 10.0, or even larger, if you are working with a very big data set.

The members of the instance-specific data structure keep the following information:

- `size`: as with the other data structures that we have implemented, this data member is the number of entries in the map;
- `capacity`: this is the number of buckets in the hash table;

- `changes`: instead of checking the load factor on every insertion, this data member is a counter for the number of changes since the last load factor check; when it reaches `TRIGGER`, the load factor will be checked, the table resized if necessary, and `changes` is reset to 0;
- `load`: this is the current load factor of the table underlying the map;
- `loadFactor`: this is the target load factor for the table;
- `increment`: this is the amount by which the current load factor increases when a new entry is inserted; `load` is incremented by this amount for each new insertion;
- `buckets`: this is the bucket array, an array of pointers to MEntry structures.

14.4.3 `destroy()` and `clear()`

A helper function is provided to apply `freeFxn()` to each entry, and to free the heap-allocated memory associated with each entry. The two methods then use this helper function to implement the required functionality.

```
/*
 * traverses the map, calling freeFxn on each entry
 * then frees storage associated with the key and the MEntry structure
 */
static void purge(MData *md, void (*freeFxn)(void *value)) {
    long i;

    for (i = 0L; i < md->capacity; i++) {
        MEntry *p, *q;
        p = md->buckets[i];
        while (p != NULL) {
            if (freeFxn != NULL)
                (*freeFxn)(p->value);
            q = p->next;
            free(p->key);
            free(p);
            p = q;
        }
        md->buckets[i] = NULL;
    }
}

static void m_destroy(const Map *m, void (*freeFxn)(void *value)) {
    MData *md = (MData *)m->self;
    purge(md, freeFxn);
    free(md->buckets);
    free(md);
    free((void *)m);
}

static void m_clear(const Map *m, void (*freeFxn)(void *value)) {
    MData *md = (MData *)m->self;
    purge(md, freeFxn);
    md->size = 0;
    md->load = 0.0;
    md->changes = 0;
}
```

14.4.4 containsKey() and get()

A helper function is provided to find the entry that corresponds to `key`; if such an entry exists, the bucket index is returned in `*bucket`, and the `MEntry *` for that entry is the function return value. If no such entry exists, `NULL` is the function return value.

```
/*
 * local function to locate key in a map
 *
 * returns pointer to entry, if found, as function value; NULL if not found
 * returns bucket index in `bucket'
 */
static MEntry *findKey(MData *md, char *key, long *bucket) {
    long i = hash(key, md->capacity);
    MEntry *p;

    *bucket = i;
    for (p = md->buckets[i]; p != NULL; p = p->next) {
        if (strcmp(p->key, key) == 0) {
            break;
        }
    }
    return p;
}

static int m_containsKey(const Map *m, char *key) {
    MData *md = (MData *)m->self;
    long bucket;

    return (findKey(md, key, &bucket) != NULL);
}

static int m_get(const Map *m, char *key, void **value) {
    MData *md = (MData *)m->self;
    long i;
    MEntry *p;
    int ans = 0;

    p = findKey(md, key, &i);
    if (p != NULL) {
        ans = 1;
        *value = p->value;
    }
    return ans;
}
```

14.4.5 `put()` and `putUnique()`

Two helper functions are provided to support these methods:

- `resize()` resizes the hash table. It attempts to double the number of buckets; if doing so takes it over `MAX_CAPACITY`, it limits the size to `MAX_CAPACITY`. If the capacity of the table is already at `MAX_CAPACITY`, it does nothing.
- `insertEntry()` inserts a new (key, value) pair into the bucket corresponding to index `i`. In that function, an `MEntry` is allocated, the `key` is duplicated on the heap and stored in the entry, the value is stored in the entry, and the entry is inserted at the head of that bucket (i.e., the bucket is a LIFO linked list). We also increment the size, increment the current load value, and increment the `changes` variable.

The methods each check if `changes > TRIGGER`, and if so, resize the table if `load > loadFactor`. They each attempt to find an entry corresponding to `key`. `putUnique()` returns an error if an entry is found, while `put()` just replaces the value in that entry. If an entry is not found, both methods invoke the insertion helper function.

```
/*
 * helper function that resizes the hash table
 */
static void resize(MData *md) {
    int N;
    MEntry *p, *q, **array;
    long i, j;

    N = 2 * md->capacity;
    if (N > MAX_CAPACITY)
        N = MAX_CAPACITY;
    if (N == md->capacity)
        return;
    array = (MEntry **)malloc(N * sizeof(MEntry *));
    if (array == NULL)
        return;
    for (j = 0; j < N; j++)
        array[j] = NULL;
    /*
     * now redistribute the entries into the new set of buckets
     */
    for (i = 0; i < md->capacity; i++) {
        for (p = md->buckets[i]; p != NULL; p = q) {
            q = p->next;
            j = hash(p->key, N);
            p->next = array[j];
            array[j] = p;
        }
    }
    free(md->buckets);
    md->buckets = array;
    md->capacity = N;
    md->load /= 2.0;
    md->changes = 0;
    md->increment = 1.0 / (double)N;
}

/*
 * helper function to insert new (key, value) into table
 */
static int insertEntry(MData *md, char *key, void *value, long i) {
    MEntry *p = (MEntry *)malloc(sizeof(MEntry));
    int ans = 0;

    if (p != NULL) {
        char *q = strdup(key);
        if (q != NULL) {
            p->key = q;
            p->value = value;
            p->next = md->buckets[i];
            md->buckets[i] = p;
            md->size++;
            md->load += md->increment;
            md->changes++;
            ans = 1;
        } else {
            free(p);
        }
    }
    return ans;
}

static int m_put(const Map *m, char *key, void *value, void **previous) {
    MData *md = (MData *)m->self;
    long i;
    MEntry *p;
    int ans = 0;

    if (md->changes > TRIGGER) {
        md->changes = 0;
        if (md->load > md->loadFactor)
            resize(md);
    }
    p = findKey(md, key, &i);
    if (p != NULL && previous != NULL) {
        *previous = p->value;
        p->value = value;
        ans = 1;
    } else {
        if (previous != NULL)
            *previous = NULL;
        ans = insertEntry(md, key, value, i);
    }
    return ans;
```

```
}

static int m_putUnique(const Map *m, char *key, void *value) {
    MData *md = (MData *)m->self;
    long i;
    MEntry *p;
    int ans = 0;

    if (md->changes > TRIGGER) {
        md->changes = 0;
        if (md->load > md->loadFactor)
            resize(md);
    }
    p = findKey(md, key, &i);
    if (p == NULL) {
        ans = insertEntry(md, key, value, i);
    }
    return ans;
}
```

14.4.6 remove()

The method searches for an entry matching the `key`. If found, it copies the value into `*value`. Then is scans the linked list in that bucket to determine the `MEntry` node that precedes the node being removed; the node is then unlinked from the list. `size` is decremented, `load` is decremented, and `changes` is incremented. Then the key and the entry are returned to the heap.

```
static int m_remove(const Map *m, char *key, void **value) {
    MData *md = (MData *)m->self;
    long i;
    MEntry *entry;
    int ans = 0;

    entry = findKey(md, key, &i);
    if (entry != NULL) {
        MEntry *p, *c;
        *value = entry->value;
        /* determine where the entry lives in the singly linked list */
        for (p = NULL, c = md->buckets[i]; c != entry; p = c, c = c->next)
            ;
        if (p == NULL)
            md->buckets[i] = entry->next;
        else
            p->next = entry->next;
        md->size--;
        md->load -= md->increment;
        md->changes++;
        free(entry->key);
        free(entry);
```

```
            ans = 1;
        }
        return ans;
    }
```

14.4.7 `size()` and `isEmpty()`

These methods do the obvious thing with `size`.

```
    static long m_size(const Map *m) {
        MData *md = (MData *)m->self;
        return md->size;
    }

    static int m_isEmpty(const Map *m) {
        MData *md = (MData *)m->self;
        return (md->size == 0L);
    }
```

14.4.8 `keyArray()`

A helper function allocates an appropriately-sized array of `char *` pointers for the keys that are defined in the map. The method then uses this function to return the array to the caller. **N.B.** the caller must invoke `free()` to return this array to the heap when no longer needed.

```
    /*
     * local function for generating an array of keys from a map
     *
     * returns pointer to the array or NULL if malloc failure
     */
    static char **keys(MData *md) {
        char **tmp = NULL;
        if (md->size > 0L) {
            size_t nbytes = md->size * sizeof(char *);
            tmp = (char **)malloc(nbytes);
            if (tmp != NULL) {
                long i, n = 0L;
                for (i = 0L; i < md->capacity; i++) {
                    MEntry *p = md->buckets[i];
                    while (p != NULL) {
                        tmp[n++] = p->key;
                        p = p->next;
                    }
                }
            }
        }
        return tmp;
    }
```

```
static char **m_keyArray(const Map *m, long *len) {
    MData *md = (MData *)m->self;
    char **tmp = keys(md);

    if (tmp != NULL)
        *len = md->size;
    return tmp;
}
```

14.4.9 `entryArray()` and `itCreate()`

A helper function allocates an appropriately-sized array of `MEntry *` pointers for the entries in the map. The methods then use this function to implement their functionality. **N.B.** the caller must invoke `free()` to return the array from `entryArray()` when no longer needed; the `destroy()` method on the iterator returned by `itCreate()` must be invoked when the iterator is no longer needed.

```
/*
 * local function for generating an array of MEntry * from a map
 *
 * returns pointer to the array or NULL if malloc failure
 */
static MEntry **entries(MData *md) {
    MEntry **tmp = NULL;
    if (md->size > 0L) {
        size_t nbytes = md->size * sizeof(MEntry *);
        tmp = (MEntry **)malloc(nbytes);
        if (tmp != NULL) {
            long i, n = 0L;
            for (i = 0L; i < md->capacity; i++) {
                MEntry *p = md->buckets[i];
                while (p != NULL) {
                    tmp[n++] = p;
                    p = p->next;
                }
            }
        }
    }
    return tmp;
}

static MEntry **m_entryArray(const Map *m, long *len) {
    MData *md = (MData *)m->self;
    MEntry **tmp = entries(md);

    if (tmp != NULL)
        *len = md->size;
    return tmp;
}
```

```
static const Iterator *m_itCreate(const Map *m) {
    MData *md = (MData *)m->self;
    const Iterator *it = NULL;
    void **tmp = (void **)entries(md);

    if (tmp != NULL) {
        it = Iterator_create(md->size, tmp);
        if (it == NULL)
            free(tmp);
    }
    return it;
}
```

14.4.10 The constructor

The usual memory allocations for the dispatch table, the instance-specific data structure, and the bucket array. There is also logic to use defaults for the capacity and load factor if nothing is provided by the caller. If all allocations are successful, then the various members of the instance-specific data structure are filled in.

```
static Map template = {
    NULL, m_destroy, m_clear, m_containsKey, m_get, m_put, m_putUnique,
    m_remove, m_size, m_isEmpty, m_keyArray, m_entryArray, m_itCreate
};

const Map *Map_create(long capacity, double loadFactor) {
    Map *m = (Map *)malloc(sizeof(Map));
    long N;
    double lf;
    MEntry **array;
    long i;

    if (m != NULL) {
        MData *md = (MData *)malloc(sizeof(MData));

        if (md != NULL) {
            N = ((capacity > 0) ? capacity : DEFAULT_CAPACITY);
            if (N > MAX_CAPACITY)
                N = MAX_CAPACITY;
            lf = ((loadFactor > 0.000001) ? loadFactor : DEFAULT_LOAD_FACTOR);
            array = (MEntry **)malloc(N * sizeof(MEntry *));
            if (array != NULL) {
                md->capacity = N;
                md->loadFactor = lf;
                md->size = 0L;
                md->load = 0.0;
                md->changes = 0L;
                md->increment = 1.0 / (double)N;
                md->buckets = array;
                for (i = 0; i < N; i++)
                    array[i] = NULL;
                *m = template;
                m->self = md;
            } else {
                free(md);
                free(m);
                m = NULL;
            }
        } else {
            free(m);
            m = NULL;
        }
    }
    return m;
}
```

14.4.11 Accessor functions to the `MEntry` opaque data type

The return from `entryArray()` has `MEntry *` pointers; this is also what is returned when invoking `next()` on an iterator obtained from `itCreate()`. The header file does not define the structure of an `MEntry`; this is done in the implementation to hide that structure from users. Thus, the header file defines two accessor functions that return the `key` and the `value` from an entry.

```
char *mentry_key(MEntry *me) {
    return me->key;
}

void *mentry_value(MEntry *me) {
    return me->value;
}
```

14.5 Maps with generic keys

While the discussion in the preceding sections has restricted keys to character strings, we have all the tools we need to enable the creation and manipulation of maps that support generic keys. We will have to change several of the signatures in the header file to do so.

First, we will show the complete header file. Then we will discuss the semantics of each of the functions/methods for which the signatures have changed.

14.5.1 The interface

```
#ifndef _MAP_H_
#define _MAP_H_
```

```
#include "iterator.h"              /* needed for factory method */

typedef struct map Map;            /* forward reference */
typedef struct mentry MEntry;      /* opaque type definition */

const Map *Map_create(long capacity, double loadFactor,
                      long (*hash)(void*, long N), int (*cmp)(void*, void*));
struct map {
    void *self;
    void (*destroy)(const Map *m, void (*freeK)(void*), void (*freeV)(void*));
    void (*clear)(const Map *m, void (*freeK)(void*), void (*freeV)(void*));
    int (*containsKey)(const Map *m, void *key);
    int (*get)(const Map *m, void *key, void **value);
    int (*put)(const Map *m, void *key, void *value, void **prevK, void **prevV);
    int (*putUnique)(const Map *m, void *key, void *value);
    int (*remove)(const Map *m, void *key, void **theK, void **theV);
    long (*size)(const Map *m);
    int (*isEmpty)(const Map *m);
    void **(*keyArray)(const Map *m, long *len);
    MEntry **(*entryArray)(const Map *m, long *len);
    const Iterator *(*itCreate)(const Map *m);
};

void *mentry_key(MEntry *me);
void *mentry_value(MEntry *me);

#endif /* _MAP_H_ */
```

14.5.2 The semantics

- `const Map *Map_create(long capacity, double loadFactor,`
  `                      long (*hash)(void*, long N), int(*cmp)(void*, void*));`
  `capacity` and `loadFactor` are exactly as before. The `hash` function pointer is used by the implementation to hash from a `void *key` to an index in the bucket array; it will return a long integer in the range $[0, N)$. The `cmp` function pointer is used by the implementation to compare two `void *key`s for equality; to be consistent with `strcmp()`, the `cmp` function pointer must return a value < 0 if the first argument is less than the second, 0 if they are equal, and a value > 0 if the first argument is greater than the second.

- `void (*destroy)(const Map *m, void (*freeK)(void*), void (*freeV)(void*));`
  `void (*clear)(const Map *m, void (*freeK)(void*), void (*freeV)(void*));`
  Previously, we were able to provide a `freeFxn()` to be used to free the `value` of the (key, value) pair. Since the keys were character strings, the implementation made a copy of the `key` in the hash table, and returned that storage to the heap when `destroy()` or `clear()` was invoked. When using generic keys, the implementation has no way to know how to make a copy of the `key`; therefore, as with the `value` before, the `key` must continue to exist until such time as the entry is `remove()`'d. In

the same way that we had `destroy()` and `clear()` be able to return the storage associated with the value, we would like them to be able to return the storage associated with the key, as well. Thus, two function pointers are required here, with `freeK` being used to free the storage associated with the keys, and with `freeV` being used to free the storage associated with the values.

- `int (*containsKey)(const Map *m, void *key);`
  `int (*get)(const Map *m, void *key, void **value);`
  `int (*putUnique)(const Map *m, void *key, void *value);`
  These method signatures now require the `key` argument to be `void *`. The `key` will be stored in a node in the hash table, and the caller must guarantee that the value of the `key` will be maintained until such time as the (key, value) pair is removed.

- `int (*put)(const Map *m, void *key, void *value, void **prevK, void **prevV);`
  `int (*remove)(const Map *m, void *key, void **theK, void **theV);`
  Both of these methods now require that if a (key, value) pair is replaced or removed, the stored `key` and `value` must be returned to the caller so that the caller can return any heap storage associated with them. If either `void **` argument is NULL, then that value will not be returned.

- `void **(*keyArray)(const Map *m, long *len);`
  The function return of this method is an array of `void *` pointers, not an array of `char *` pointers.

- `void *mentry_key(MEntry *me);`
  The return value of this function is now `void *`, not `char *`.

14.5.3 An example program using the generic Map interface

Suppose that each line of a file consisted of a keyword, a blank separator, and a string value to associate with that keyword. We need to load each (key, value) pair from the file into a map for some subsequent processing. The first subsection below shows how we would do so using a Map interface for which keys are strings. The second subsection shows how the program would change to use the generic Map interface.

14.5.3.1 Using the Map interface for which keys are strings

```
#include "map.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define UNUSED __attribute__((unused))

int main(UNUSED int argc, UNUSED char *argv[]) {
    const Map *m;
    char buf[4096];
    if ((m = Map_create(1024L, 2.0)) == NULL) {
        fprintf(stderr, "Unable to create Map to hold (key, value) pairs\n");
        return 1;
    }
    while (fgets(buf, sizeof buf, stdin) != NULL) {
        int n = strlen(buf) - 1;
        char *sp, *vp;
        if (buf[n] == '\n')
            buf[n] = '\0';          /* overwrite newline character */
        sp = strchr(buf, ' ');      /* find the blank */
        *sp++ = '\0';               /* replace with EOS, sp points to value */
        vp = strdup(sp);            /* make a copy on the heap */
        if (! m->putUnique(m, buf, vp)) {
            fprintf(stderr, "%s is not a unique key\n", buf);
        }
    }
    /* code to use the map */
    m->destroy(m, free);            /* destroy map, each value is freed */
    return 0;
}
```

14.5.3.2 Using the Map interface for generic keys

```
#include "map.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define UNUSED __attribute__((unused))

/*
 * need to define the hash and compare functions used by the generic map
 */
#define SHIFT 31L
long hash(void *value, long N) {
    char *sp = (char *)value;
    unsigned long ans;

    for ( ; *sp != '\0'; sp++)
        ans = SHIFT * ans + (unsigned long)*sp;
    return (long)(ans % N);
}

int cmp(void *v1, void *v2) {
    return strcmp((char *)v1, (char *)v2);
}

int main(UNUSED int argc, UNUSED char *argv[]) {
    const Map *m;
    char buf[4096];
    if ((m = Map_create(1024L, 2.0, hash, cmp)) == NULL) {
        fprintf(stderr, "Unable to create Map to hold (key, value) pairs\n");
        return 1;
    }
    while (fgets(buf, sizeof buf, stdin) != NULL) {
        int n = strlen(buf) - 1;
        char *sp, *vp, *kp;
        if (buf[n] == '\n')
            buf[n] = '\0';          /* overwrite newline character */
        sp = strchr(buf, ' ');      /* find the blank */
        *sp++ = '\0';               /* replace with EOS, sp points to value */
        kp = strdup(buf);           /* make a copy on the heap */
        vp = strdup(sp);            /* make a copy on the heap */
        if (! m->putUnique(m, kp, vp)) {
            fprintf(stderr, "%s is not a unique key\n", buf);
        }
    }
    /* code to use the map */
    m->destroy(m, free, free);   /* destroy map, each key and value is freed */
    return 0;
}
```

Index