C for Java Programmers

George Ferguson

Summer 2016
(Updated Fall 2020)

Contents

1 Introduction **7**

2 Overview of Java and C **9**

2.1 What's The Same? 9

2.2 What's Different? 10

3 Development and Execution **11**

3.1 Development and Execution in Java and in C 11

3.2 Setting Up Your Development Environment 14

3.3 Writing Your First C Program 16

3.4 Compiling Your First C Program 17

4 Basic Expressions and Statements **21**

4.1 Comments 21

4.2 Primitive Types 22

4.3 Producing Output 23

4.4 Operators and Expressions 24

4.5 Variables and Assigment 25

4.6 Arrays 26

4.7 Strings 28

5 Control Flow **31**

5.1 Conditional Statements 31

5.2 Iteration Statements 32

5.3 Other Control Flow Statements 33

6 Functions **35**

6.1 Function Parameters 36

6.2 Function Declarations 37

7 Structured Types **39**

8 Memory Management **43**

8.1 Variables, Addresses, and Pointers 43

8.2 Passing Parameters by Reference 46

8.3 Memory Allocation 48

8.4 Dynamic Memory Allocation in Java 49

8.5 Dynamic Memory Allocation in C 50

8.6 Dynamic Arrays 53

8.7 Dynamic Data Structures 56

8.8 Function Pointers 63

9 Defining New Types **67**

10 Sharing Code: Files and Libraries **71**

10.1 The C Preprocessor 71

10.2 Separate Compilation, Libraries, and Linking 73

10.3 Standard System Libraries 74

10.4 Project Development 75

11 Building Larger C Programs **79**

12 Debugging a C Program **83**

12.1 Debuggers 84

12.2 Compiler Options 84

12.3 valgrind 85

13 Final Thoughts **87**

14 References **89**

Chapter 1

Introduction

When I teach introductory programming, I tell students that they are learning a foreign language: the language understood by the computer. The purpose of programming is to translate your ideas about how to solve a problem into a language that the computer understands so that it can follow your instructions.

You are a Java programmer. You are already fluent in the Java programming language. But now you find that you need to learn a new programming language, namely the language called "C." This is just like learning a second (human) language. As I'm sure you know, some human languages are more similar than others. If you know Spanish, you can learn French or Italian relatively easily. Many of the constructions are the same, although there are some important differences in the details. On the other hand, if you know English, it's of relatively little use to you in learning Chinese or Japanese—they don't even use the same characters!

Luckily for you, Java and C are closely related. In fact, Java was developed by starting with C and adding features designed to help programmers develop complex programs more quickly and with fewer errors. Thus you will have no problem understanding the high-level structure of a C program. There are important differences that we will point out, starting in the next sections, but it really is an easy conceptual transition.

Keep in mind as you learn C that you are stepping back in the history of programming. C was developed in the early 1970's when computers were much simpler (and less powerful) than today. Java appeared in the mid-1990's and has been evolving and expanding ever since. A fundamental thing to realize is that C provides *much less support* to the programmer. It's much easier to make mistakes and often harder to figure out how to fix them.

So why would anyone use C? There are a couple of reasons. First, because as you will see in Section 3, a C program runs in a much smaller memory footprint. This makes C the choice for embedded systems and other environments where memory is at a premium. Second, because C programs run with less support, they may also run faster. Although higher-level languages like Java and C# have gotten faster, it is still probably the case that tightly coded C is as fast as you can get without writing assembly code, which is not only much harder but by definition not portable across platforms. If speed is important (and for many, possibly even most, programs, it is *not*), C is often the choice. A third reason to use C is that it is kind of the universal interchange language. Many other languages interoperate with C, allowing you to connect components written in different languages into one application using C. Finally, because C is more minimalist than many of its successors, it forces the programmer to confront some illuminating design and implementation questions and really think about what their code is doing.

Please note that this guide is not a definitive manual for the C programming language. For that, you should get a copy of *The C Programming Language, Second Edition* by Brian Kernighan and Dennis Ritchie. I will refer to this as "*K&R*" in the rest of this document. Not only is it the definitive specification of the language, it's one of the clearest, most useful books you will ever read. You will learn things about programming and programming languages that you can apply to any language, including Java. Knowing C will make you a better Java programmer, as well as having an additional very useful tool in your programming toolbox.

Chapter 2

Overview of Java and C

Let's start with a quick overview of the similarities and differences between Java and C.

2.1 What's The Same?

Since Java is derived from C, you will find many things that are familiar:

- Values, types (more or less), literals, expressions
- Variables (more or less)
- Conditionals: `if`, `switch`
- Iteration: `while`, `for`, `do-while`, but not `for`-in-collection ("colon" syntax)
- Call-return (methods in Java, functions in C): parameters/arguments, return values
- Arrays (with one big difference)
- Primitive and reference types
- Typecasts
- Libraries extend the core language (although the mechanisms differ somewhat)

2.2 What's Different?

On the other hand, C differs from Java in some important ways. This section gives you a quick heads-up on the most important of these.

- No classes or objects: C is not an object-oriented language, although it has structured types (`struct` and `union`) and there are ways of doing things like inheritance, abstraction, and composition. C also has a mechanism for extending its type system (`typedef`).
- Arrays are simpler: there is no bounds checking (your program just dies if you access a bad index), and arrays don't even know their own size!
- Strings are much more limited, although the C standard library helps.
- No collections (lists, hashtables, *etc.*), exceptions, or generics.
- No memory management: You must explicitly allocate (`malloc`) and release (`free`) memory to use dynamic data structures like lists, trees, and so on. C does almost nothing to prevent you from messing up the memory used by your program.
- Pointer arithmetic: You can, and often have to, do arithmetic on addresses of items in memory (called pointers). C allows you to change the contents of memory almost arbitrarily. This is powerful but also dangerous magic.

Bottom line: When you program in C you are closer to the machine. This gives you more flexibility but less protection. The next section explains this in more detail by comparing the development and execution models of Java and C.

Chapter 3

Development and Execution

3.1 Development and Execution in Java and in C

Figure 3.1 (page 12) shows the basic components involving in developing and executing a Java program. You should already be familiar with this process, so I will describe it only briefly.

- You write your program (source code) as a text file using the Java programming language. The name of your source file ends in "`.java`".
- You compile your source code using the `javac` command ("Java compiler") to create a Java class file containing "bytecode" (`.class` file).
- Bytecode does not use the native instruction set of any real computer. Instead, it uses an abstract instruction set designed for an abstract computer called the "Java Virtual Machine" or "JVM."
- To run your program, you run the `java` command, which simulates the JVM running on your actual computer. You tell it to load and run your class file (bytecode), which results in your code being executed.
- That is, you have a "real" program, `java`, running on your computer, and it is simulating the execution of the bytecode of your program on the "not-real" JVM. This is slower than running native code directly on the computer, but also safer and better for development.

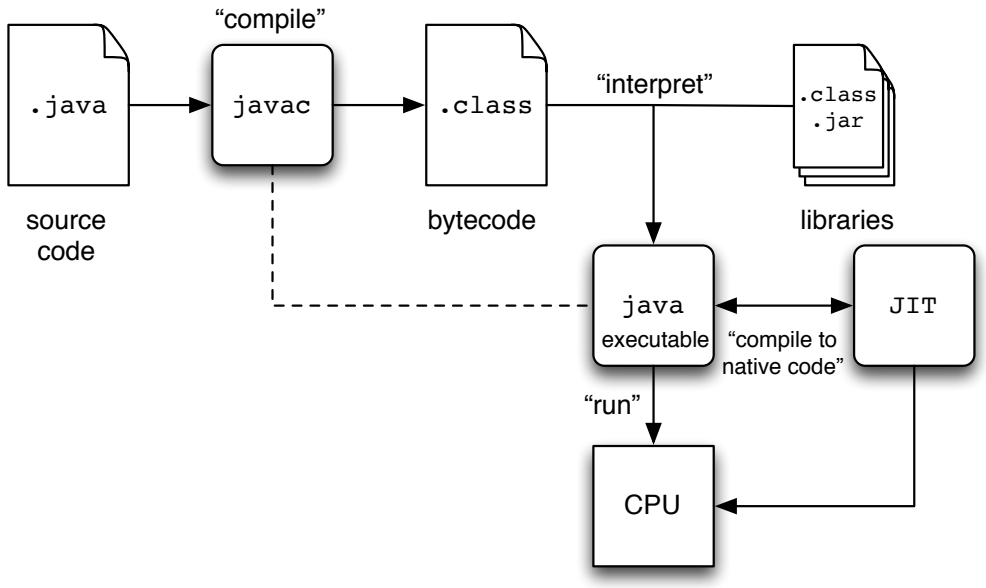

Figure 3.1: Java Development and Execution Environment

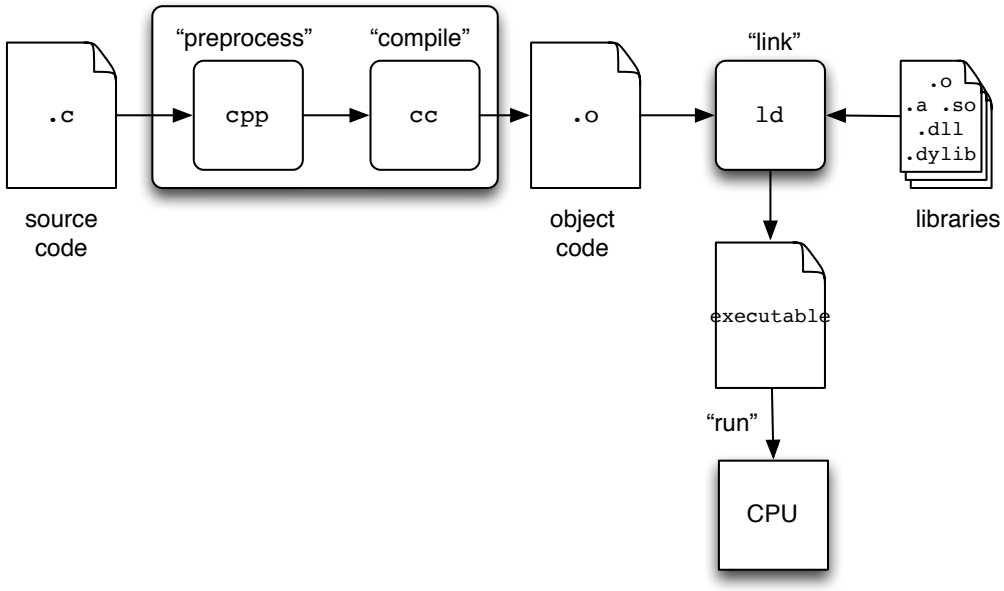

Figure 3.2: C Development and Execution Environment

- Java programs can use other classes and libraries of other classes (`jars`) to provide additional functionality. In fact, most of Java is library classes.
- To partially compensate for the slowdown due to interpreting bytecode, Java includes a feature called a "Just In Time" or "JIT" compiler. This part of the `java` runtime system detects pieces of Java code that are executed frequently and compiles them into the real instruction set of the host computer (native code). This is faster than interpreted bytecode, but usually not quite as fast a native code.
- Finally, note that the Java compiler, `javac`, is itself a Java program, written in Java! That is, running `javac` means running `java` (the Java runtime) and telling it to run the class for the Java compiler. I'll let you think about how the first Java compiler was written...

By comparison, Figure 3.2 shows a similar component diagram for development and execution in C. I'll point out the main differences briefly.

- In C, you write your program as a text file using the C programming language. The name of your source file ends in "`.c`".
- You then compile your code into processor-specific object code ("`.o`") using the `cc` command (for "C compiler").
- The C compiler actually consists of two phases. The first is a "pre-processing" phase which can transform the text of your program in various useful ways. Then the compiler phase translates the resulting C program text into native object code. Although it is technically possible to run just one or the other phase of the C compiler, you will never do that in practice.
- Before your code can be run, it needs to be "linked" with any libraries that it uses, including the C runtime. The result is an executable file containing native object code.
- You can run your executable directly from the command line. If it crashes, your program dies. Sometimes this happens without even a message unless you are using a debugger. In the not-so-good old days, you might even crash the entire machine.
- A compiled C program runs at full speed on the processor for which it was compiled.

In general, the key difference between Java and C is that in C you are running much closer to the machine. When computers were less powerful and had less memory, it was vital to squeeze out every drop of performance in order to do even simple computing tasks. The history of programming languages is a continual evolution from low-level object code and assembly language to higher-level languages like Fortran, Algol, Lisp, Smalltalk, Java, Javascript, and all the other modern languages. C sits somewhere in the middle, not as low-level as assembly language, but without too many of the powerful (but possibly slow) features of higher-level languages. Interestingly, C was actually predated by some higher-level languages. And some of the earliest languages had features that have only recently been rediscovered.

But as a Java programmer coming over to C, your development and execution process should be quite familiar. You write your code as one or more text files. You compile your code until the compiler is happy with it. There are IDEs for C as for Java should you want to use them (see next section). There are libraries that make your life easier, and you can develop your own libraries and reusable code. Then you run your program and see if it works. If you need to debug it, there are debugging tools for C as for Java. In the case of C, these can catch fatal errors and allow you to post-mortem the execution rather than simply crashing.

3.2 Setting Up Your Development Environment

Development environments, whether for Java or for C, are a deeply religious topic. You can opt to use an "Integrated Development Environment" (IDE), like Eclipse (cross-platform), Visual Studio (Windows), or XCode (Mac). Or you can work like the pioneers did, using separate tools that they could combine in ingenious ways. For simple projects, an IDE is fine. For complicated projects that include some amount of automated compilation or testing, it helps to know how things work under the hood so that you can build your own tools.

For tips on setting up Eclipse for C development, see my document Using Eclipse for C Programming.

Most Computer Science students should become comfortable working from the terminal (command shell). If that's not yet you, then this is your chance to learn something new. Here are some things to look for:

- You'll need an editor for writing your C programs: vim, Emacs, NotePad, TextEdit, BBEdit, ... Text editors are another subject on which programmers have deeply-held religious views.

- You'll need a C compiler and linker. The standard open-source tool is the GNU Project's `gcc` suite.
  - On Linux distributions, this is standard or can be easily installed using a package manager.
  - On Macs, you need the free Apple Developer Tools, which include and extend the GNU tools. Apple originally used `gcc`, but now prefers a compiler suite called `clang`/`llvm` (although the name `gcc` still works).
  - On Windows, it is possible to get `gcc` working, but it's not for the faint of heart. Microsoft expects its developers to use Visual Studio. Check out the MinGW or Cygwin projects if you want an alternative.

- You'll probably want a debugger. I'm a big believer in print statements for debugging, but sometimes you have to look at a program as it breaks. The GNU debugger is `gdb`, for `llvm` the debugger is `lldb`, and on Windows it's Visual Studio or `gdb`.

- For anything but the simplest projects, you will need some kind of build system to automate compilation and testing. The classic tool is `make`, and it is still frequently used for C development. More modern (but also way more complicated) tools include `ant` and `gradle`, both of which are more commonly used for Java development than for C.

Finally, whether you use an IDE or a separate toolchain, you should learn to love some kind of version control system. The original tools on UNIX were SCCS ("Source Code Control System") and CVS ("Concurrent Versions System"). These were replaced by Subversion (`svn`), which is still very popular. More modern version control systems include Git, Mercurial (`hg`), and Bazaar (`bzr`). Note that you don't have to store your files "in the cloud" (*e.g.*, on GitHub) to benefit from a version control system. You can use it locally to track changes, make branches, and revert to a prior state if you break your code five minutes before it's due. Of course, you should also be sure to backup your files, but you already know that...

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

int main(int argc, char *argv[]) {
    printf("Hello world!\n");
}
```

Figure 3.3: "Hello World" Program in C

3.3 Writing Your First C Program

Figure 3.3 shows the classic "Hello World" program which, by the way, first appeared to my knowledge in *The C Programming Language*. Let's break it down by comparing it to the Java equivalent with which I'm sure you're already familiar (otherwise check out "Hello World" Program at Wikipedia).

First the comment at the top: block comment just like in Java.

Next: `#include`: Semantically, this is like an `import` statement in Java, making some library functionality available to your program. In C, it is implemented quite differently, as described more fully in Section 10. In this case, we are including the definitions of the C "Standard Input/Output" (`stdio`) library so that we can use the `printf` function to produce output later in the program.

Now, notice that there is no "main class." Indeed, there are no classes at all in C. Instead, functions are defined at the toplevel of the file, as can be done in Python, and are visible to any following code.

The next block of code is the definition of the function `main`. Just like in Java, each program needs a `main` function and this is the function that is called when the program starts (Java inherited this from C, as a matter of fact).

The function definition should look familar. It says that `main` is a function that returns an `int` (Java uses a `static void` method) and takes two parameters. In Java you get an array of `String`s passed to your program containing any arguments specified on the command-line. In C, as you'll see in Section 4.6, arrays don't know their length, so the first parameter is the number of arguments

(traditionally named `argc` for "argument count"). Strings are defined as `char*` ("pointer to `char`", see Section 4.7), and the parameter `argv` (for "argument vector") is an array of these. So this looks a bit different but it means exactly the same thing as in Java.

And finally, the one-line body of the `main` function in curly brackets. Instead of calling `System.out.println` (or `format`), we are calling the function `printf`, which is part of the C Standard input/output (I/O) library and defined in `stdio.h`. But the function call itself is the same in C as in Java: the name of the function, followed by comma-separated values for the parameters in parentheses.

There is only one argument in this call to `printf`: a string literal containing the text we want to print. String literals use double quotes just like Java. The "`\n`" at end of the string might be new to you. It indicates a "newline" character, since `printf` in C behaves like `print` in Java and does not print a newline the way `println` does. There is no `println` in C. Furthermore, `printf` in C can take additional parameters, and substitutes them for placeholders in the first string parameter, just like Java's `format` methods. You will use this often since, unlike Java, there is no string concatenation (`+` for strings) and there are no `toString` methods.

And finally, a semi-colon at the end of every statement, just like Java.

That's the whole program. Just like Java, the program exits (stops running) when the `main` function is done. That is, just like Java unless you're using multithreading either explicitly or implicitly, as with Swing graphics. And those things are not so easy to do in C.

This C program should be understandable to any Java programmer. The family resemblance between the languages is very strong. Yes there are some important differences, but you are not starting from scratch.

3.4 Compiling Your First C Program

So you've written your first C program, now you want to compile and run it. If you're using an IDE, it may have already done part of this for you. Different IDEs work differently. Some compile constantly in the background to warn you of errors; with others you need to tell it to compile. Either way, you edit your

source code until it compiles successfully.

If you are not using an IDE, you need to be comfortable working with a terminal window and command interpreter (or "shell"). To compile your C program, you need to invoke the C compiler and tell it to compile your file. The simplest form of this is something like the following, which you would type at your command shell prompt:

```
cc hello.c
```

This assumes that your source file is named "`hello.c`" and that it is in the "current working directory" of the terminal (shell). It also assumes that your C compiler is named "`cc`", which is traditional. You might need to try `gcc` or `clang` or whatever is right for your platform and development environment.

When you run the C compiler, you will either get error messages or perhaps no output. If there is no output, you should be left with an executable program file in the current working directory. For historical reasons, this file is called "`a.out`" regardless of what the source file was named. That isn't usually so helpful, so you can tell the C compiler to call the executable something else with the "`-o`" option:

```
cc -o hello hello.c
```

Now your executable will be called "`hello`" (on Windows it's traditional to name executables with names that end in "`.exe`", but not on Unix-like platforms). You should see your new file if you type `ls` on Unix platforms, `dir` on Windows, or find it in a window showing the directory (folder) with your source file in it.

Now you need to run your program. With an IDE, this usually means some kind of "Go" button. Without an IDE, you need to tell your terminal command interpreter (shell) to run the newly-created executable file. For reasons that don't concern us now, you probably can't just type its name the way you did the `cc` command. Instead you probably have to give the pathname of the executable file. The easiest way to do this is:

```
./hello
```

The "dot" ("`.`") means "the current working directory, whatever it is," so this pathname refers to the file named "`hello`" in the current working directory.

Either way, your program should run and you should see its output printed either to the console of your IDE or to your terminal. Congratulations! You are now a C programmer.

The rest of this document builds on this basic program. You can use it as a template for trying the code examples. Try different variations and find out what works and what doesn't. There's only one way to learn a foreign language, and that's to get out there and use it to communicate. For learning programming languages, this means getting onto a computer and *writing programs*.

Chapter 4

Basic Expressions and Statements

Let's dive a bit deeper into the C programming language, emphasizing similarities to and differences from Java. As always, you'll want to consult *K&R* for the final word on C.

As you already know from your prior programming experience:

- *Comments* are a crucial part of any program.
- All computer programs compute *values*.
- Values come in several *types*.
- Values can be written as *literals* ("literally this value") or as *expressions* that use *operators* to combine values into new values.

Java's comments, values, types, and expressions are derived from C's, so they are very similar. Let's take a look at each of these in a bit more detail.

4.1 Comments

The syntax for comments in Java was taken from that for C. Text enclosed between /* and */ is ignored by the compiler. Modern C compilers also understand the // syntax that was introduced in C++ and is also used in Java.

4.2 Primitive Types

Table 4.1 shows the primitive types in both Java and C. As you can see, all the basic numeric types in Java are inherited from C.

Java	**C**
`int`	`int`
`short`	`short`
`long`	`long`
`float`	`float`
`double`	`double`
`char`	`char` (see text)
`byte`	N/A
`boolean`	N/A

Table 4.1: Comparison of Java and C primitive types

Historically these types were specified with somewhat less precision than in Java. For example, `int` was whatever was most natural for a given machine, and could be 16 bits, 32 bits, or something else. I'm not going to dwell on these details. If you need to know how many bits are in an `int`, you can lookup how to do that.

Two of Java's primitive types do not exist in C: `byte` and `boolean`. That's ok. Most people don't use `byte` anyway. In C, use an `int` (or a `short` if you're really concerned about memory usage) and just don't assign it a value less than -128 or greater than 127. I told you C gave you less protection than Java.

Now `boolean`s are genuinely useful, but if you think about it, they simply represent a true-false, yes-no, 0-1 value. So again, in C, use an `int`. C uses the convention that 0 means "false" and *any non-zero value* means "true." Boolean variables and functions that return Boolean values are traditionally declared `int`. The `stdbool` library provides support for more explicit booleans in modern C compilers.

In Java, a `char` represents a 16-bit Unicode character (a "code point"). In C, a `char` is an 8-bit byte representing an ASCII character. The difference matters if you're working with modern text sources which can include foreign characters, accents, math symbols, and, yes, emoji. There is some library support for handling unicode in C, but it is not a strength of the language. On the other hand, a C `char` is exactly a byte, so you could use it for that if you wanted (there are some signed/unsigned math considerations, and C has `unsigned` numeric types, including `unsigned char`).

Literals for primitive types are just like in Java. Numeric representations for integer and floating point numbers. Character (`char`) literals in single-quotes. String literals enclosed in double-quotes. Inside a character or string literal, use backslash for escapes, including `\'`, `\"`, and `\n`.

4.3 Producing Output

We've already seen the `printf` method to produce output in our "Hello World" program. In that example, we passed in the string that we wanted to print. This makes it seem like the Java method `println` from the `java.io.PrintStream` class. In fact though, it's more like the Java method `format` which uses the class `java.util.Formatter`. Python 2 (and 3) has the "`%`" operator, and Python 3 has the function `format`. All of these are derived from C's `printf`.

The `printf` function always takes at least one parameter. The first argument to `printf` is called the *format string* because it controls the format of the output. Any character in the format string other than a percent sign ("`%`") or backslash ("`\`") is simply printed as is.

We've already seen that the backslash ("`\`") is used to introduce characters that can't be easily typed in a string, such as the newline `\n`. A backslash in front of a percent sign "escapes" its special meaning (described next), meaning that `\%` prints as just "*%*".

An unescaped percent sign indicates that the value of the next parameter to the function call is to be converted to text and printed at that point in the output. Unlike in Java, there is no automatic conversion of values to strings. Instead, the character following the percent sign says what type of thing it is. For example "`%d`" indicates an integer ("decimal") number, "`%f`" a floating-point number, and

Arithmetic	+, −, *, /, %
Comparison	==, !=, <, <=, >, >=
Logical	!, &&, ||
Increment/Decrement	++, −− (pre- and post-fix)
Bitwise	&, |, ^, <<, >>, ~

Figure 4.1: Basic C operators

"%s" a string. We will see examples of these when we look at expressions and variables.

Another thing we will see shortly is that there is no string concatention in C. If you are used to writing `println` statements using "+", you will need to break that habit and get used to specifying a format string and the arguments that fill the % specifiers.

FYI: Wikipedia has an extensive description of printf format strings.

4.4 Operators and Expressions

Most of the operators in Java were also derived from C, so they will be familiar to you. The basic C operators are shown in Figure 4.1. Note that Java took "==" for "equals" comparison from C. Pascal (which came after C), used ":=" for assigment and plain "=" for "equals", which actually makes more sense, but unfortunately never caught on.

Precedence rules (order of operations) for the operators are as in Java, and parentheses can be used to group sub-expressions.

As noted above regarding the Java `boolean` type, comparison and logical operators in C use the assumption that 0 means "false" and any non-zero value means "true." You can check the examples shown in Figure 4.2 in your own program.

```
printf("%d\n", 3 < 2);           // 0
printf("%d\n", 2 < 3);           // 1
printf("%d\n", 1 && 0);          // 0
printf("%d\n", 1 || 0);          // 1
printf("%d\n", 0 && 123);        // 0
printf("%d\n", 0 || 123);        // 1
printf("%d\n", 1 < 2 && 2 < 3);  // 1
```

Figure 4.2: Comparison and logical operators in C and the zero/non-zero convention.

There is one important difference regarding Java and C operators: there is no string concatenation using the + operator. Strings are not primitive types in C or in Java. We discuss this further in Section 4.7.

4.5 Variables and Assigment

As in Java, variables in C must be declared before they are used. A variable declaration includes declaring the type of the variable:

```
int x;
float f;
```

Historically, variables could only be declared in certain contexts, such as at the start of a function. This has gradually been relaxed and is now very similar to Java.

Also as in Java, in C you assign a value to a variable in an assignment statement using "=" (single equals sign):

```
x = 1;
f = 3.14159;
```

Of course the value assigned to the variable need not be a literal, as above, but can be any expression that computes a value of the appropriate type. There are also the shortcut assignment statements +=, −=, and so on, just like in Java.

Finally, again as in Java, in C you may combine declaration and initialization in one statement:

```
int x = 1;
float f = 3.14159;
```

The value must be "compile-time evaluable" meaning either a literal or an expression that uses only values known at compile-time (including literals).

In Java, you can declare a variable `final`, meaning that its value cannot change after it has been initialized. This allows the Java compiler to optimize the use of the variable and it will flag as an error any attempt to change a `final` variable. In C, you can declare a variable `const` (for "constant"). This has the same meaning, and the C compiler will detect any attempt to change a `const` variable. Unfortunately, it cannot prevent you from doing it in all cases. *K&R* (p. 40) says that "the result is implementation-dependent if an attempt is made to change a `const`." So don't do that.

4.6 Arrays

Java took its array concept from C, so both the syntax and the semantics of arrays are very similar in the two languages, although there are a couple of key differences.

In both languages, square brackets (`[]`) indicate arrays. In Java, you declare an array variable by adding `[]` to a type:

```
int[] numbers;
char[] letters;
```

Then, separately or as the initialization of the variable, you allocate the storage for an array by calling `new` on a type and specifying the size in square brackets:

```
numbers = new int[3];
letters = new char[5];
```

Declaration and allocation: two separate concepts, two separate steps in Java, although you can do them in one statement.

In C, you put the square brackets after the variable name and you have to know the size of the array at compile-time:

```
int numbers[3];
char letters[5];
```

This not only declares the variable to be of an array type, but also allocates the space to store the given number of elements. If you need to create an array whose size you don't know at compile-time, or if you don't want to allocate it at compile-time for some other reason, you will need to allocate it dynamically, as discussed in Section 8.

In both languages, an array can be initialized by providing the necessary number of compile-time evaluable expressions in curly brackets. In C, this looks like:

```
int numbers[3] = { 1, 2, 3 };
char letters[5] = { 'a', 'b', 'c', 'd', 'z' };
```

As in Java, the C compiler is smart enough to figure out the size of the array for you if you provide an initializer, so these would normally be written as:

```
int numbers[] = { 1, 2, 3 };
char letters[] = { 'a', 'b', 'c', 'd', 'z' };
```

If not explicitly initialized, the value of array elements is typically 0 (but you probably shouldn't count on that).

To specify an array type for a function parameter (see Section 6), in both languages, you can use the array type without its size:

```
int main(int argc, char* argv[]) {
    ...
}
```

Once you have an array variable, you get or set its elements using the same square-bracket syntax as Java (the first element of an array is at index 0 in both languages):

```
printf("%d\n", numbers[0]);     // 1
printf("%c\n", letters[1]);     // b
numbers[1] = 99;                // { 1, 99, 3 }
numbers[2] += 1;                // { 1, 99, 4 }
```

C also has multi-dimensional arrays. The details are a bit different from Java—in C they are rectangular arrays of the element type, whereas in Java they are arrays of references to other arrays. In both languages you use multiple indexes to access the elements (*e.g.*, `a[i][j]`).

Like in Java, arrays in C are *reference types*. That is, the value of an array variable is a reference to where the element values are stored. In fact, as we will see, the value of an array variable in C is the *address* in memory of the first element in the array, with the rest of the elements occupying the subsequent contiguous memory locations. Arrays and other reference types are often called *pointers* in C since they "point" to the data in memory.

Like in Java, the length of an array cannot be changed after it has been created. But unlike in Java, *there is no way to get the length of an array*. Seriously. Either you keep track of how many elements are in it, or you mark the last entry in some way. This is one of the most common sources of bugs, crashes, and security holes in C programs, which is why Java looks after it for you (at some cost, of course).

Finally, there is no built-in support for printing the contents of arrays. You will need to write a function (Section 6) that takes an array (and its length or other way of knowing which element is last) and iterates over the elements (Section 5.2) printing them out. This is a good simple exercise in C programming. Spoiler: It will look almost identical to its Java equivalent.

4.7 Strings

In Java, strings (sequences of characters) are full-featured objects, with constructors and methods and runtime checking and all that. Java also provides special support for strings through the string literal syntax and the overloading of + to mean string concatention.

In C, strings are simply arrays of `chars`. That's it. The following allocates a string that can hold 32 characters:

```
char name[32];
```

You can use the string literal syntax to initialize a character array, and if you do, the C compiler is smart enough to figure out the length for itself:

```
char name[] = "George";
```

In Java, strings are immutable (cannot be changed), although you can create new strings from them or pieces of them. In C, you can get *and set* individual elements of character arrays, so they are most definitely mutable. Whether this is a good thing or not depends on the application.

Since there is no way to get the length of an array from the array, it also isn't possible to get the length of a string from the character array, or to know when you've reached the end of the string (the last character in the array). By convention (that is, since the dawn of C), the last character in a string is indicated by the value 0 (the ASCII character `NUL`). This was nice because it meant that a newly-allocated but uninitialized character array was treated as the empty string because its first element is 0 (indeed, all its elements are zero). This convention means that a character array holding a string must always be one element longer than the number of characters in the string, in order to hold the `NUL` character at the end.

Many of the methods of Java's `String` class are provided by functions in the C standard library (see Section 10). Almost all of these methods rely on the `NUL`-terminated string convention. Furthermore, when a character array is initialized from a string literal, the compiler allocates one element more than the number of characters in the literal and puts a `NUL` (0) in there to terminate the string. If you construct strings and manipulate character arrays manually, you need to look after preserving the `NUL` at the end of the string.

Messing with strings and messing up the `NUL` at the end of them is the cause of many bugs and security holes in C programs which, again, is why Java does significantly more for you with strings (at some cost).

Chapter 5

Control Flow

So far we've seen enough to write basic *straight-line* programs in C. Next let's look at control statements. The good news is that they're almost exactly the same as in Java.

5.1 Conditional Statements

The basic conditional statement is the `if` statement:

```
if (x != 0) {
    // Do something...
}
```

As in Java, the condition must be in parentheses and the "then" clause (nested statement or block) is executed if the condition is true (that is, evaluates to a non-zero value; see Section 4.4). Because there is no true boolean type, only the zero/non-zero convention, you will often see conditionals like the following, which tests whether the variable `x` is non-zero:

```
if (x) {
    // Do something...
}
```

Also as in Java, the "then" clause can be a single statement or a compound statement (block) in curly brackets. In both languages, I highly recommend using

curly brackets for all control statements. Lookup the "dangling else" problem if you don't understand why this is good practice.

Multi-conditional statements in C are identical to Java's:

```
if (x < 0) {
    // Do something when x is negative
} else if (x > 0) {
    // Do something else when x is positive
} else {
    // Otherwise do something when x is 0
}
```

Java also inherited its multi-way conditional `switch` statement from C, so they are identical:

```
switch (i) {
case 0: // Do something
    break;
case 1: // Do something
    break;
// ....
default:
    // Do something
}
```

In C, execution falls through into subsequent cases unless prevented by an explicit `break` statement. This is because the `switch` statement was originally designed to mimic a common assembly language programming idiom. Unfortunately, Java kept this behavior, but at least you're already used to it (Swift has finally broken with the past on this point).

5.2 Iteration Statements

Java's iteration statements are derived from C's. The `while` statement, including the `do-while` form, is identical (with C using the zero/non-zero convention to represent false and true in the condition).

```
while (x != 0) {      // or perhaps ``while (x)''
    // Do something
}
```

The "original" form of the `for` loop is identical in C and in Java:

```
for (int i=0; i < 10; i++) {
    // Do something
}
```

Historically you couldn't declare the variable inside the `for` statement, you had to do it at the start of the enclosing block. But now life is easier, and identical to Java, although of course you can declare the loop variable earlier if you need to.

C does not provide any kind of built-in Collection classes (see below). Thus the "colon" form of the `for` loop ("for element in collection") is not available. To iterate over the elements of an array, use the array index as loop variable. Iterating over the elements of a list or other dynamic collection is part of the fun of programming in C (see Section 8.7).

5.3 Other Control Flow Statements

C has `break` and `continue` statments for use in iteration constructs. These behave the same as in Java, terminating the loop or moving immediately to the next iteration of the loop, respectively.

C's `return` statement is identical to Java's and used to return a value from a function (see Section 6).

C also provides an arbitrary branching construct, leftover from the early days of assembly language and other "unstructured" programming. Here's what *K&R* has to say about it:

> "C provides the infinitely-abusable `goto` statement, and labels to branch to. Formally, the `goto` is never necessary, and in practice it is almost always easy to write code without it." (p. 65)

You will almost certainly not need to use `goto` statements in your own programs, and you are unlikely to even come across them in code written since the 1990's. Nonetheless, the history of "structured" programming and Edsger Dijkstra's famous letter to *Communications of the ACM* entitled "Go-to statement considered harmful" make good reading for a student of Computer Science.

Chapter 6

Functions

All but the earliest programming languages had the idea of *subroutines*: reusable pieces of code that do something or compute something (or both). In Java, subroutines are called *methods* and are always part of a class. In C, they are called *functions*. In Python, you can have both standalone functions and methods in classes. In C there are only standalone functions.

I assume that you understand how functions and methods work: calling a function and returning from it ("call-return" flow of control), passing values to functions (parameter binding), and returning values from functions.

As we saw in the "Hello World" program definition of the `main` function, a function definition in C is very similar to a method definition in Java: return type, function name, parameters in parentheses separated by commas, body of the function in a block. C supports `void` functions that don't return a value, just like Java. Here's another example of a function definition. This one doubles and returns the integer value that it is given:

```
int times2(int x) {
    return x * 2;
}
```

6.1 Function Parameters

As in Java, parameters are passed by value. Consider the following code:

```
int double_it(int x) {
    x = 2 * x;
    return x;
}

int main(int argc, char* argv[]) {
    int i = 10;
    int j = double_it(i);
    printf("i=%d, j=%d\n", i, j);
}
```

The value of `i` is copied to the parameter `x` when function `double_it` is called. Thus the assignment to `x` within the function does not affect the value of `i`. You should already be comfortable with this from Java.

C does provide a form of parameter passing that allows a function to change variables passed in as parameters ("pass-by-reference", called an `inout` parameter in Swift). In C, this is better thought of as passing a pointer (reference) to the original variable, so we will describe it further in Section 8 when we discuss pointers.

Historically only primitive values (including pointers) could be passed to functions. More modern C compilers also allow structured types to be passed although it is much more common to pass a pointer to the structured object.

Function parameters may be marked `const`, meaning that the function promises not to change them. This is only really relevant for reference types (pointers). And anyway, *K&R* says it isn't always enforced. But you will often see this done in library functions to make clear to programmers which things may or may not be changed by the function call.

6.2 Function Declarations

Once you start writing in functions in C, you may be surprised to discover that they must be declared before they can be used. In Java, the order that methods are defined in a class does not matter. The Java compiler first scans your class definition for any method definitions. It then makes a *second pass* to compile the bodies of the methods. Since it already knows about all the methods in the class, it can compile a call to method B in some other method A even if A is defined earlier in the class.

In C, no such luck. If you need to define a function after its first use, you must use a *forward declaration*. This declaration tells the compiler what arguments the function takes and what it returns—in other words, everything the compiler needs to know to make sure that you're using the function corrrectly. Here's an example:

```
int f(int x);
...
void g(int x, int y) {
    int z = f(y);
    ...
}
...
int f(int x) {
    ...
}
```

The first line is a forward declaration of the function `f` which says that it takes a single `int` parameter and returns an `int` value. We can then use (call) `f` in the body of function `g` even though `f` hasn't been defined. And then at some point elsewhere in the program, we need to define `f` and its definition must match its declaration or the compiler will complain. Try it.

Forward declarations are also used for declaring functions defined in other files or libraries. See Section 10 for more on that.

Chapter 7

Structured Types

We've said several times already that C does not have classes and objects. Instead, there are two ways of defining *structured* types—types that are composed of other types. Of these, one is a thousand times more common that the other, but the other is interesting from a historical perspective.

By far the most common form of structured type is called a "structure" or `struct`:

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

This means that every instance of the type `struct point` includes two *members*: `x` and `y`, both `int`s in this example. A `struct` is therefore like the part of a Java class definition that specifies the instance variables for the class.

Incidentally, the name `point` is not itself a type. Rather, it is called a "tag," which combines with the keyword `struct` to specify the type, as in `struct point`. This is discussed further in Section 9.

As with Java instance variables, you access the members of a `struct` using "dot" syntax:

```
struct point origin;
origin.x = 0;
origin.y = 0;
```

Structured types can be composed just like in Java:

```
struct line {
    struct point start;
    struct point end;
}
```

As with Java, you can chain the accessors using dot syntax:

```
struct line line1;
line1.start.x = 1;
line1.start.y = 2;
line1.end.x = 3;
line1.end.y = 4;
```

Note that there are no "getters" and "setters" since there are no classes, although it is possible to define functions to do these things; see Section 8.

In C, `structs` are the building blocks of data structures, just like classes are in Java. Any time you think "this should be a class," use a `struct`. We'll see more of how `structs` work like classes when we look at dynamic data structures in Section 8.7.

The much less common C structured type is called a `union`:

```
union value {
    int i;
    float f;
};
```

The difference is that the members of a union *share the same memory*. That is, in our example, allocating a `union value` allocates enough memory for either an `int` or a `float`, whichever is bigger. The choice of which member to access determines how the value (the bits) stored in the `union` is interpreted:

```
union value v;
v.i = 123;              // Set v to bits of integer 123
printf("%f\n", v.f);    // Interpret bits of 123 as float
```

There are some `unions` used in common Unix system libraries, and they can be used to implement some features of object-oriented programming. But you will rarely need to use them yourself outside of low-level systems programming.

Chapter 8

Memory Management

With functions and structured types in hand, it's time to deal with one of the few really significant differences between C and Java. This is the question of how C programs manage the memory that they use to store values. This includes reference types, also known as "pointers," and how you use these to develop complex, dynamic data structures. There are several aspects of this, all intertwined, and I will do my best to spell them out for you. But be warned: this is where you will need to spend time becoming familar with C's practices and idioms.

8.1 Variables, Addresses, and Pointers

Let's start at the beginning. You already know that a variable is a name associated with a piece of memory somewhere in your program's total memory space. Assigning a value to a variable copies that value to the place in memory associated with the variable. Using the value of variable retrieves the value from the piece of memory. In fact, named variables were developed to free programmers from needing to keep track of all the chunks of memory their programs were using to store values.

In Java, you cannot do anything to a variable other than get or set its value. The details of storage in memory are completely hidden from you. In C, however, you can retrieve the address of the location in memory where the variable is stored. This is called a "pointer." If the variable is, say, an `int`, then the type of the address is "pointer to `int`", written "`int*`" (and sometimes pronounced "`int` pointer").

You use the "&" (ampersand) or "address-of" operator to get the address of a variable. For example:

```
int i = 123;
int* ip = &i;
```

Here the first variable is an `int`, the second one is a pointer to an `int`. By the way, you can write `int*` as the type, which is more readable, or do it the *K&R* way, with the `*` on the variable name separated by whitespace (as in "`int *ip`").

When the C compiler compiles the first line of this code, it produces machine instructions to find and use an `int`-sized chunk of memory to store the value of the variable `i` (more on this below). It also produces code to store the representation of 123 as an `int` in that chunk of memory.

For the second line, the C compiler produces code to find and use a chunk of memory suitable for holding the address of a chunk of memory. Exactly how big this is depends on the platform, but 32 and now 64 bits (four and eight bytes, respectively) are common. The initialization code for initializing the variable `ip` ("`i` pointer") copies the address of the memory chunk storing `i` into the memory chunk storing `ip`. Figure 8.1 illustrates this situation. Note that the value of the pointer variable (1108) is the address in memory of the original variable.

Now that we can get a pointer to a variable, what can we do with it? The main operations on pointers are getting and setting the value "pointed to," meaning the value stored at the address given by the value of the pointer variable. This is called *dereferencing* the pointer. A pointer variable is a reference to another value; dereferencing the pointer gets you that other value.

The dereferencing operator in C is the same "`*`" (asterisk) character used to indicate pointer types. This can get confusing, but you just have to read carefully and keep all the "`*`"s straight. For example:

```
int i = 123;
int* ip = &i;
printf("%d, %d\n", i, *ip);
```

The `printf` statement will print two values. The first value is the integer value of the variable `i`. You know that it will be 123. The second is the integer value

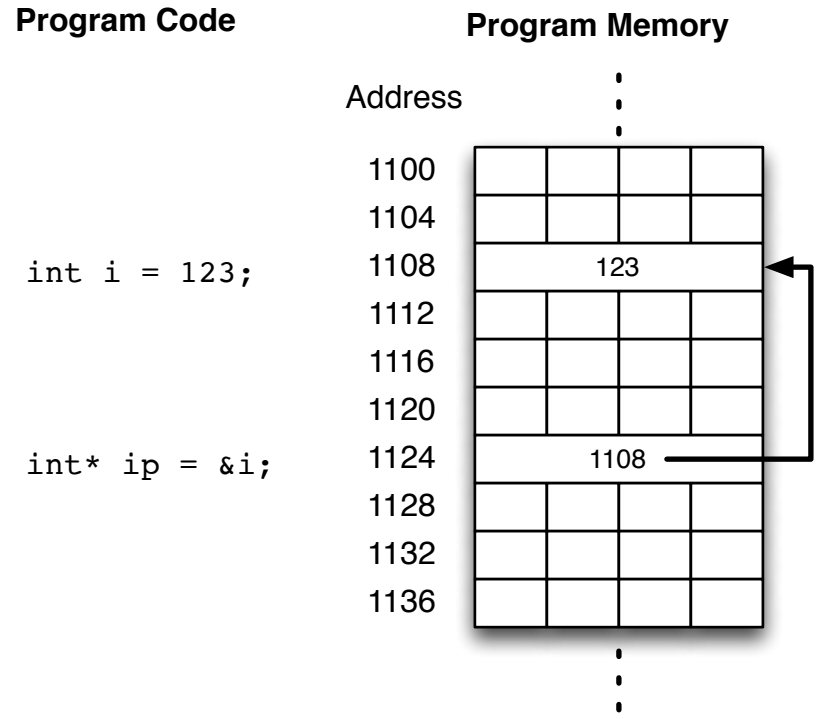

Figure 8.1: Variable and pointer in memory

resulting from deferencing the `int` pointer `ip`, written `*ip`. Since we set the value of the pointer `ip` to be the address of `i`, this will also be 123. The `*` operator on a variable being used in an expression (an "*rvalue*") dereferences the pointer and returns the pointed-to value.

Now add the following line and print again:

```
*ip = 987;
printf("%d, %d\n", i, *ip);
```

The `*` operator on a variable being assigned to (an "*lvalue*") says assign not to the pointer variable itself, but to the location pointed to by the pointer. Since `ip` is the address of `i`, this puts 987 into the memory locations storing the value of `i`. Thus it changes the value of `i` without assigning to that variable directly! Both values will print as 987.

There are many legitimate uses of pointers, which is why they are part of the language. We'll see examples of these uses shortly. But clearly pointers are potentially very dangerous since they let you change memory in non-obvious ways. If the pointer `ip` had somehow been reassigned to point to somewhere other than the location of `i`, then assigning a new value to `*ip` would not change the value of `i`. This would probably be a mistake given the variables' names, but the C compiler can't tell and so can't help you avoid it.

8.2 Passing Parameters by Reference

In Section 6, we described how parameters are passed by value into functions. But what if you want to pass a variable to a function and allow the function to change the value of the variable? You should be able to figure how to do this using pointers.

Suppose we want to write a function that takes an integer variable as parameter. It first tests whether the variable is greater than or equal to zero. If it is, the function should decrement the value of the variable and return true. But if the variable is less than or equal to zero already, it should do nothing and return false. This is a classic example of a function that wants to return more than one value. In this case, the two values are the value of the counter being decremented and the boolean result of whether or not it was decremented.

Think about how you would do this, then look at my code:

```
int test_and_dec(int* ip) {
    if (*ip >= 0) {
        *ip -= 1;
        return 1;
    } else {
        return 0;
    }
}
```

The key is that we define the parameter `ip` to be of type `int*` rather than `int`. To access the value, we can deference the pointer, but we can also change the value of the variable by storing into the dereferenced pointer.

Somewhere else in our program (perhaps in `main`), we can use this function as follows:

```
int i = 123;
...
if (test_and_dec(&i)) {
    // i is now one less than before, possibly 0
} else {
    // i was already <= 0
}
```

The address of the `int` variable `i` matches the type of the `int*` function parameter `ip`. When the function deferences the pointer and decrements it, this changes the value pointed to, namely the value of `i`. And this change (called a *side-effect*) is separate from the value returned by the function.

Another example of using pass-by-reference parameters is C's `scanf` function, used to read input from the terminal. Like `printf`, `scanf` takes a format (or control) string that in this case specifies how to read the input, followed by a series of parameters. For `printf`, the additional parameters are generally variables whose values are printed. For `scanf`, they need to be references to variables so that `scanf` can set their values as it reads and parses the input. For example:

```
int i;
float f;
char name[50];
scanf("%d %f %s", &i, &f, name);
```

The control string tells scanf to read an integer (`%d`), a floating-point value (`%f`), and an array of characters (`%s`), each separated by whitespace. Since `int`s and `float`s are primitive types, we must pass a reference to them. As we will see in the next section, arrays are reference types, so we can pass the array variable `name` directly and `scanf` will change the contents of the array rather than the value of the variable itself.

There is a lot more one could say about `scanf`. For example, it will read characters up to a newline when reading a `%s` value. But what if there are more than 50 characters (or whatever the length of the array is)? The answer is a "buffer overflow" and the source of many dangerous and pernicious bugs. It is ok for simple,

non-critical input. But real programs that do serious input parsing generally write their own code using more primitive functions.

We will see further examples of passing parameters by reference when we look at dynamic data structures and pointers to structured types.

8.3 Memory Allocation

Now let's look at how C uses pointers for dynamic memory allocation where you don't know at compile-time exactly what memory you will need. Classic examples of this are dynamic data structures like lists, trees, and other "collections," as well as objects in object-oriented languages like Java. We'll get to those shortly. But first, we need to understand a bit more about memory allocation in general.

In C, as in Java (although you don't often think about it in Java), there are three ways for your program to allocate memory to store values:

1. Automatic: This is the easiest to understand. When you define a local variable or function parameter (which is effectively a local variable), the compiler arranges for the necessary memory to be allocated and initialized when the enclosing scope is entered.

2. Static: In C, memory for static variables is allocated once when the program starts. This is similar to class variables in Java, which is why Java uses the `static` keyword borrowed from C. In C, all global variables are static in term of memory allocation, and other variables can be explicitly marked `static` if needed (this is fairly rare and often indicates a bad design).

3. Dynamic: Both automatic and static variables are defined at the time the program is compiled. If you don't know until run-time that you need to store something, or what or how many you need to store, then you need to use dynamic allocation to allocate memory on the fly. This is also called "allocating from the heap," as opposed to "allocating on the stack," which is used for automatic variables where the memory is part of the "stack frames" that are pushed and popped as functions are called and return.

Most of the rest of this section is concerned with dynamic memory allocation.

It is worth making one note about the different allocation classes and the relationship to pointers. The lifetime of a static variable is the lifetime of the program. So it is always safe to take its address and use it as a pointer. The lifetime of an automatic variable, on the other hand, is the enclosing scope (typically a block or function definition). When an automatic variable goes out of scope, the memory it uses is reclaimed by the memory system. If you have a pointer to an automatic variable, you may inadvertently dereference it after it is no longer valid. This will cause a nasty crash if you're lucky, or it may not crash but will just use whatever bits happen to be at that address now. Yowza! Dynamically-allocated memory has to be managed explicitly by the programmer, as we shall see next.

8.4 Dynamic Memory Allocation in Java

Before we look at dynamic memory allocation in C, let's look at how it works in Java. In Java, if you want to create a new instance of a class, you use `new` to invoke the class' constructor:

```
Person p;
...
p = new Person("Alan", "Turing");
```

You should know by now that doing this first allocates a chunk of memory of a size suitable for storing a `Person`, as determined by its class definition. That chunk of memory is passed to the class constructor along with any arguments, and the constructor is responsible for initializing the newly-allocated chunk of memory (which is why it is called `init` in some languages, such as Smalltalk, Objective-C, and Swift).

In Java, the variable `p` contains a reference to the object, or if you think about it, the "address" of the chunk of memory storing the `Person` instance. I put "address" in scare quotes because it may not be exactly the in-memory address since Java runs in a virtual machine environment (see Section 3), but it means the same thing. This is why we say that, in Java, classes define reference types.

Finally, in Java, when there are no more active references to an object, the chunk of memory containing the object is returned to the system for reuse. This is called *garbage collection*. Simple cases are easy for the Java system to handle. In the

example above, if we never assign the value of `p` to any other variable, then when `p` goes out of scope there can be no other references to the `Person` instance so it can be garbage-collected. But what if you assign the value to another variable with a different scope? Or what if you return it as the value of a method? Tricky. But Java looks after it for you, at some cost.[1]

8.5 Dynamic Memory Allocation in C

In C, you have all the same pieces but you have to look after almost everything by yourself. Sorry. But as a Java programmer, you don't need to be intimidated. Everything you already know about dynamic data structures still applies, you just have to spell things out somewhat painfully. And you can appreciate why languages like C++ and Java were developed to spare programmers all that spelling out over and over again.

For starters, in C, there are no classes so there are no constructors. Instead there is a single library function, `malloc`, for memory allocation. You give it a size in bytes, and it returns a pointer to a chunk of memory of that size. For example:

```
#include <stdlib.h>
...
void* p = malloc(100);
```

Note that `malloc` returns a value of type `void*`. This is C's somewhat strange way of saying "pointer to unknown type." After all, it doesn't know what you plan to do with that chunk of memory.

If `malloc` fails, usually because your program's memory space is exhausted, it returns the special value `NULL`. Like an `OutOfMemory` execption in Java, there

[1] In fact, adding garbage collection to Java was one of the most dramatic design decisions made by the original Java designers. Other languages, like Lisp, had had garbage collection for decades, but it was still considered too expensive for "real" code. Java's design decision emphasized that the true cost of almost all software is in the development time, not the run time, and garbage collection makes the developer's job much, much easier (at some runtime cost). Apple's Objective-C and Swift languages use something called "Automatic Reference Counting," which is less powerful but also less expensive than true garbage collection. Cleaning up allocated memory remains a challenge for large, long-running platforms.

is usually not much you can do about this situation. For simple programs, it almost always implies a programming error and terminating the program is the only reasonable thing to do. For complex, long-running software, you might need to be more careful. Either way you should always test that the return value of `malloc` is not `NULL`, although I will not do this in my examples to save space.

But if `malloc` doesn't return `NULL`, then we have a pointer to a chunk of bytes. What can we do with it? The answer is: not much. The C compiler will not let you dereference a `void` pointer because it doesn't know what type is on the other end of the pointer.

The solution is that you can typecast (or just "cast") the pointer to a different type. As in Java, a cast is you telling the compiler "I know what type this thing is—trust me." In Java, casts are almost never necessary and usually indicate lazy programming and poor design decisions. But in C with dynamic memory allocation, they are always necessary. For example:

```
void* p = malloc(100);
int* ip = (int*)p;      // <- cast is here
*ip = 98765;
printf("%d\n", *ip);
```

Here we declare a pointer to `int` and initialize it with the address of our chunk of memory, which is the value of the variable `p`. To satisfy the compiler's type checking, we need to cast the value of `p`, which is a `void*`, to an `int*`. Once we've done that, we can dereference our `int` pointer and store an `int` value at that location in memory. When we dereference and retrieve the value at the location stored in `ip`, it is an `int`.

Typically the two steps, allocation and casting, are done together in one very common idiom:

```
int* ip = (int*)malloc(100);
```

You should confirm that this behaves the same as the first two lines of the previous program.

Now you may have noticed something a bit funny with my example. I allocated 100 bytes of memory, and then used it to store an `int`. But `int`s are generally four or eight bytes these days. What gives?

Well, in one sense nothing gives. You asked for 100 bytes and `malloc` gave them to you. If you only want to use four or eight bytes of it, the processor and the C runtime don't care.

While this is true, it's not really ok to be deliberately wasteful and furthermore it's bad programming practice. If you want to allocate an `int` dynamically, your code should make it clear that that's what's happening. So how do we know how many bytes we need to store an `int`? It turns out that C has a built-in operator for just this purpose:

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

The `sizeof` operator takes a type and returns the number of bytes required to store a value of that type. Just what we need. The C compiler is happy and our code is nice and clear. We can allocate memory to store any type of value and get a properly-typed pointer to access it. You will use this idiom often.

Now, the memory available to your program is finite. Just as with garbage collection in Java, it makes sense to return memory to the system when you are done with it. Unfortunately, in C it is all up to you to determine when a chunk of memory is no longer needed. And when it is, you call the standard library function `free` and hand it the pointer to the chunk you want to return:

```
int* ip = (int*)malloc(sizeof(int));
...
free(ip);
```

Note that the value of the variable `ip` is not changed (it is passed by value into `free` after all). Yes, the system has made note that the chunk of memory at that address is now available for reuse, but the value of the variable `ip` is still the address of the chunk. If you try to deference the pointer after freeing it, one of three things will happen:

1. Your program will crash with some kind of memory error; or
2. The dereference may find a value, but it isn't the value you expected since the chunk of memory has been reused by a subsequent call to `malloc`; or
3. Nothing funny will happen and you will happily get the value you expected to find on the other end of the pointer.

Assuming that you were really done with the chunk of memory, the first alternative is in fact best. Your program has a bug and you need to fix it. The second situation typically happens when the pointer has also been stored in some other variable with a longer lifetime, and the code that uses that variable is not aware that it has been freed. The third situation may happen if you access the pointer very soon after freeing it, and in particular before your program allocates any more memory (including in library functions, which may allocate memory to do their job).

To help avoid these so-called "dangling pointer" bugs, set the value of the pointer to `NULL` immediately after freeing it.

```
int* ip = (int*)malloc(sizeof(int));
...
free(ip);
ip = NULL;
```

Attempting to dereference a `NULL` pointer will cause a crash (usually a "segmentation violation," `SIGSEGV`, or a "bus error," `SIGBUS`). This is your cue that your program needs fixing. This technique certainly does not prevent all problems with using freed memory, but it can be generalized to more complex situations where pointers are shared within a program.

8.6 Dynamic Arrays

Now that we can allocate and free memory, let's look at the relationship between pointers and arrays in C. This is crucial to understanding how strings work, as well as more generally how C handles memory management.

We already saw in Section 4.6 that in Java, you allocate the storage for an array by calling `new` on an array type:

```
numbers = new int[3];
letters = new char[5];
```

You can do this any time in your program, not just when a variable is first initialized as part of its declaration. Thus, for example, it's easy to read an integer from the user or from a file and then allocate an array of that size.

In C, you use a variant of `malloc` called `calloc` to allocate a chunk of memory suitable for storing some number of elements of a given size contiguously:

```
int num_numbers, num_letters;
...
int* numbers = (int*)calloc(num_numbers, sizeof(int));
char* letters = (char*)calloc(num_letters, sizeof(char));
```

We've used the typecast and `sizeof` idiom just like for `malloc`.

But wait a minute. Here we have the type `int*`, but previously we said that an array of `int`s had type `int[]`. The fact is that, in C, the value of an array variable is the address of its first element. Thus every `int[]` variable is a `int*` variable. By and large, you can use an array-of-type variable anywhere that you can use a pointer-to-type variable, and vice-versa (there are some restrictions).

So for example, given the above code, you can do the following to access the elements of the array just like our array variable examples:

```
numbers[0] = 99;
letters[0] = 'x';
...
printf("%d\n", numbers[0]);
printf("%c\n", letters[0]);
```

You can also pass an array variable as value of a pointer parameter, and vice-versa:

```
int foo(int* numbers) {
    return numbers[0];
}

char bar(char letters[]) {
    return letters[0];
}

int main(int argc, char* argv[]) {
    int numbers[] = { 1, 2, 3 };
    char* letters = (char*)calloc(5, sizeof(char));
    letters[0] = 'x'; // ...
    foo(numbers);
    bar(letters);
}
```

One last thing: strings and arrays of strings. We already know that, in C, strings are simply arrays of `char`s. You can allocate a string dynamically just like any array:

```
char *str = (char*)calloc(length, sizeof(char));
```

This explains why we said at the very outset that the "string" type was `char*` in C. And remember that when you allocate strings dynamically, it is up to you to ensure that your string ends with a `NUL` (value zero) character.

If a `char*` is a string, then what's the type of an array of strings? Well, it's an array, each of whose elements is a `char*`. Since arrays are themselves just pointers, an array of strings is type `char**`. That is, it's a pointer to pointer to `char`, or pointer to string. To allocate an array of strings dynamically, you have to allocate the array, and then the individual strings that are the contents of the array:

```
char **s_array = (char**)calloc(array_len, sizeof(char*));
for (int i=0; i < length; i++) {
    s_array[i] = (char*)calloc(this_str_len, sizeof(char));
}
```

This is a very common idiom, and in fact you need to do this for any dynamically-allocated array containing elements of a reference (non-primitive) type, not just arrays of strings. Don't forget that you eventually need to `free` all these dynamically allocated strings, not just the main array. Of course this is true for dynamically allocated arrays of any type of non-primitive element.

This "pointer-to-pointer" interpretation explains why you will often see the `main` function declared as:

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

This is equivalent to using `char* argv[]`, although less clear in my opinion.

My recommendation is that you use the `[]` form when dealing with true arrays, and save the `*` form for general dynamically-allocated objects, as we'll see next. The exception is strings, which are almost always referred to as `char*`. It's all just bits and bytes in memory anyway...

8.7 Dynamic Data Structures

We're now comfortable with the general idea of pointers, of dereferencing pointers to get and set the pointed-to value, and of arrays as more or less synonymous with pointers to continguously-allocated sets of elements. Let's look at dynamically allocating structured types, which is as close as you get in C to creating objects.

Suppose that we have the definition of a structured type representing points in two dimensional space from before:

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

To dynamically allocate an instance of this structure, we just use the standard `malloc` idiom,[2] only with the type `struct point`:

```
struct point* p = (struct point*)malloc(sizeof(struct point));
```

To access the members of our newly allocated structure, we need to dereference the pointer to get to the `struct` and then pick out the member. You might try the following:

```
*p.x = 1;
*p.y = 99;
```

Unfortunately, because of the precedence of the dereference (`*`) and member selection (`.`) operators, you will get a compiler error. You can fix it with some parentheses:

[2] If this idiom seems like repetitive boilerplate for allocating structured objects, you would be right. Now you know why C++ was created (and before that, people used C preprocessor macros for this, which you can check out).

Also, be careful not to write `sizeof(struct point *)`. That will allocate only enough space for a pointer (probably four or eight bytes), but tell the compiler that it is a pointer to a struct, which is probably larger. This mistake can be very tricky to track down.

```
(*p).x = 1;
(*p).y = 99;
```

First dereference the variable. Then, with the value you have there (which is the `struct` itself), select the `x` or `y` member. This is a bit unwieldy and so incredibly common that there is a special syntax for it in C:

```
p->x = 1;
p->y = 99;
```

It means exactly what it says: follow the pointer which is the value of the pointer variable `p`, and then access the appropriate member from the pointed-to structure.

Now, what about our `struct line` example from before, which used two `struct points` as its members?

```
struct line {
    struct point start;
    struct point end;
}
```

If we use our standard idiom, we will get a pointer to a `struct line`:

```
struct line* line1 = (struct line*)malloc(sizeof(struct line));
```

Each instance of a `struct line` contains within it, that is, within the block of memory allocated for it, the memory for two `struct point` instances. So to access the members of the points, we must first derefence the pointer and then chain the member accessors:

```
line1->start.x = 667;
line1->start.y = 668;
line2->start.x = -123;
line2->start.y = 0;
```

This situation is illustrated in Figure 8.2.

Or we could change our definition of a `struct line` so that rather than having `struct points` as members, it has pointers to `struct points` as members:

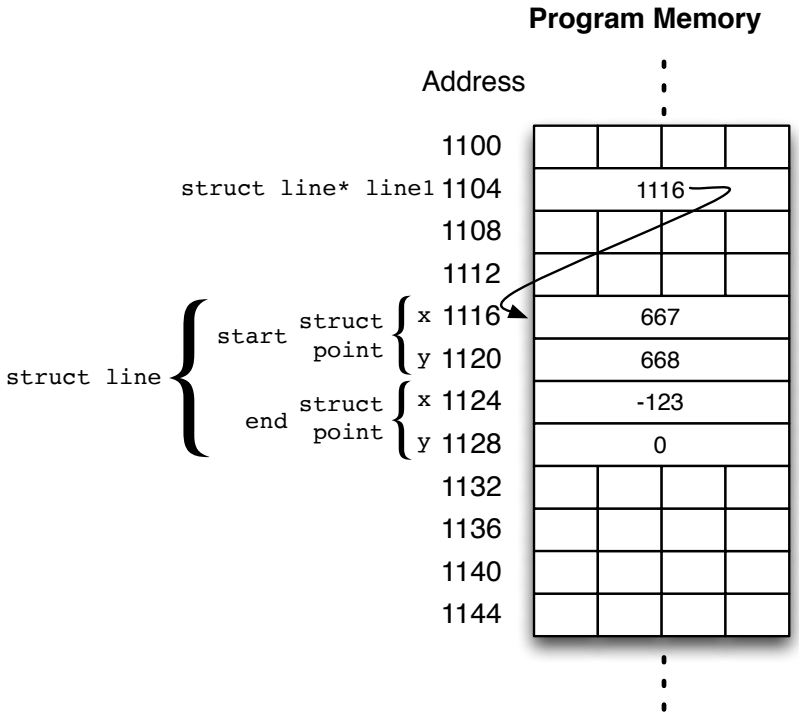

Figure 8.2: Dynamically allocated `struct line`

```
struct line {
    struct point *start;
    struct point *end;
}
```

And then, in order to allocate a `struct line`, we have to also allocate the two `struct points` and save them in the `struct line`:

```
struct line* line1 = (struct line*)malloc(sizeof(struct line));
line1->start = (struct point*)malloc(sizeof(struct point));
line1->end = (struct point*)malloc(sizeof(struct point));
```

Why would you do this? Well, perhaps the points are really the important things, for example in a graphical drawing program. The lines are defined in terms of the points, but if a point changes location, all the lines that include that point ought to change also. If there's only one instance of the point and all the lines that include it refer to that instance, then this is easy. If each line has its own instances of its points, then it's probably harder. Of course you'd have the same design issue if you were using Java, so let's get back to C.

Now that a `struct line` stores references to `struct points`, we need to chain through the pointers to access the members of the `struct points`:

```
line1->start->x = 667;
line1->start->y = 668;
line2->start->x = -123;
line2->start->y = 0;
```

Figure 8.3 illustrates the contents of memory after this code has run. You should be able to understand why the contents of memory are what they are and how to follow the pointers to the various elements of the data structure.

Don't forget that you need to `free` the reference type members of a dynamically allocated `struct` before you free the `struct` itself:

```
free(line1->start);
free(line1->end);
free(line1);
```

You can use references to structured types to implement getters and getters in C.

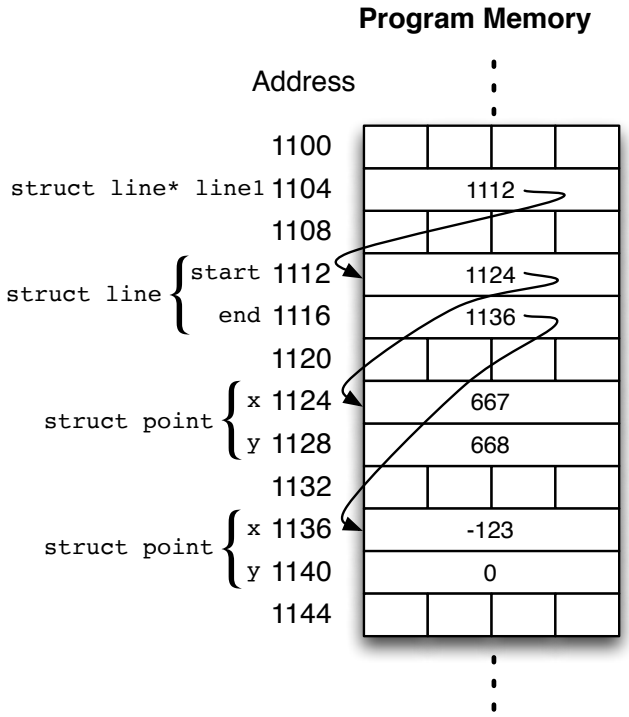

Figure 8.3: Dynamically allocated `struct line` with pointer members

For example:

```
int point_get_x(struct point* p) {
    return p->x;
}

void point_set_x(struct point* p, int x) {
    p->x = x;
}
```

The pointer itself, `p`, is passed by value, so its value cannot be changed in the function. However by dereferencing the pointer the function can change the members of the pointed-to `struct`. Very neat.

You can't (easily) make the members of a `struct` private or protected like in Java, but you can use functions like we've just seen to clarify your code. Note that you probably need to include the type of structure in the name of the function. Why? Because you may well have several structures with a member named `x` but in C you can only have one function named `get_x` and one named `set_x`.

Finally, you can define structured types that contain members that are references (pointers) to that same type. Just like in Java, these recursive types are used in many common data structures including lists and trees. For example, here's an example of simple linked list of integers with three elements:

```
struct node {
    int value;
    struct node* next;
};

struct node* head = (struct node*)malloc(sizeof(struct node));
head->value = 0;
head->next = (struct node*)malloc(sizeof(struct node));
head->next->value = 99;
head->next->next = (struct node*)malloc(sizeof(struct node));
head->next->next->value = -13;
head->next->next->next = NULL;
```

The memory layout for this data structure is illustrated in Figure 8.4.

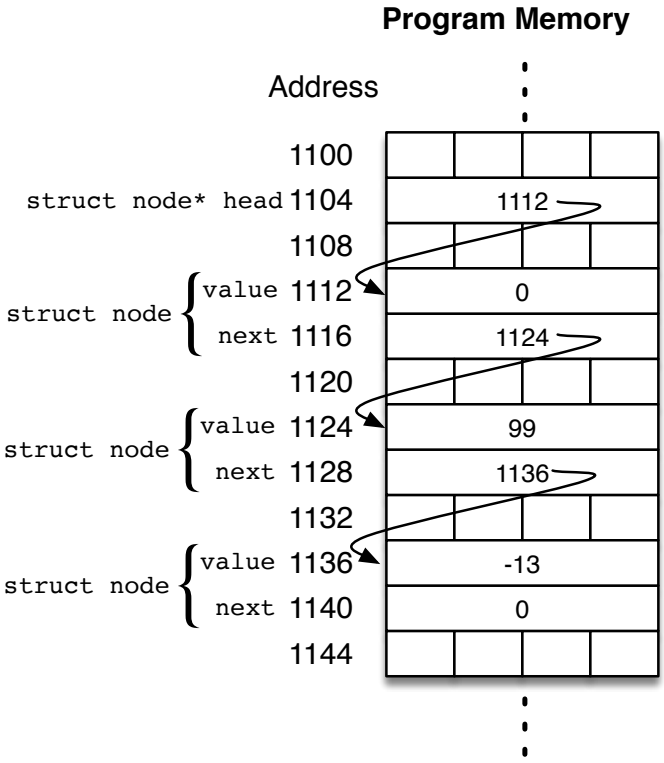

Figure 8.4: Dynamically allocated linked list

Traversing a dynamic data structure like this can be done quite elegantly in C, as in Java:

```
for (struct node *p=head; p != NULL; p=p->next) {
    printf("%d\n", p->value);
}
```

You could use similar structures and pointer traversal code to implement doubly-linked lists, trees, graphs, and so on.

In summary, creating dynamic data structures in C is no harder than creating them in Java once you master the basic idioms that we've described in this section. Freeing them, however, is a bit more challenging, and of course you don't have to do that in Java since the garbage collector will do it for you (at some cost).

8.8 Function Pointers

Thus far we have seen pointers to data: chunks of memory allocated on the stack or from the heap. C is interesting in that it also allows you to have pointers to functions, which are executable code, not data. This allows you to save references to functions in variables and pass them as parameters to functions. C was not the first language to support these features, and certainly it was a common assembly language idiom. But it is powerful and you will see it quite often in more sophisticated code, so it is worth understanding.

In C, although a function is not a variable, you can get a pointer to a function by using the "address-of" (`&`) operator on a function name just like you get a pointer to a variable by using `&` on a variable. And in fact, since a function name that is not followed by parentheses and arguments is not a function call, the bare function name can be used without the `&` operator.

For example, consider the following simple function that doubles and returns the value of its one `int` parameter:

```
int times2(int x) {
    return x * 2;
}
```

The name `times2` (or the expression `&times2`) can be used to get a pointer to this function and assign it to a variable. The type of such a pointer is "pointer to a function that takes one `int` parameter and returns an `int`." This is written as:

```
int (*fp)(int) = times2;    // or &times2
```

This defines the variable `fp` to be a pointer to a function of one `int` returning `int`. The extra parentheses around `*fp` are required to get the "pointer to function" meaning right.

Now that we have a function pointer, there is really only one thing we can do with it. We can call the function, or more precisely, we can apply it to an `int` and get an `int` back. To do this, we dereference the function pointer to get "the function being pointed to," pass it its parameter(s), and use its result:

```
int i = 999;
int j = (*fp)(i);     // 1998
```

As with pointers to structures, the extra parentheses are needed to force the dereferencing before the function application.

In this example, there is no reason to use the variable `fp` since we could just as easily have called `times2` directly in our code. Let's see a quick example that requires the use of function pointers.

First, suppose we have a function that will take an array of `int`s and double each of the elements using our `times2` function. Think about how you would do this, then look at my code:

```
void array_double(int a[], int alen) {
    for (int i=0; i < alen; i++) {
        a[i] = times2(a[i]);
    }
}
```

Not rocket science. Note that we have to pass the length of the array since in C arrays don't know their own length (see Section 4.6). The cool thing is that by using function pointers we can generalize this function so that it can apply *any* function of one `int` argument to each element of the array.

For example, suppose we also have the following simple function:

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

Not very exciting, but it's just an example. To generalize our array function, we need to do two things:

1. The array function needs a parameter telling it which `int` function to use; and
2. We need to be able to pass either `times2` or `square` as the function to use.

For the array parameter, it's the same type as our variable example earlier and it gets used (applied) the same way also:

```
void array_apply(int a[], int alen, int (*fp)(int)) {
    for (int i=0; i < alen; i++) {
        a[i] = (*fp)(a[i]);
    }
}
```

To use this array function, we can pass any "pointer to function that takes one `int` parameter and returns `int`" as the third parameter. For example:

```
int numbers[] = { 1, 2, 3 };
array_apply(numbers, 3, times2);    // 2, 4, 6
array_apply(numbers, 3, square);    // 4, 16, 36
```

You can use your array printing function described in Section 4.6 to verify this code for yourself. The values shown assume that you first do `times2` and then do `square`. You should try other arrangements to confirm your understanding.

K&R includes a more complicated example of passing a comparison function to a `sort` function. You can compare this to Java's use of interfaces (for example, `Comparable`) and Python's function objects (for example, the `key` parameter to `list.sort`).

Finally, one other sophisticated use of function pointers is to store them as members of a `struct`. For example:

```
struct dog {
    char *name;
    void (*bark)();
};

void arf() { printf("arf\n"); }
void woof() { printf("woof\n"); }

// ...

struct dog *fido = (struct dog *)malloc(sizeof(struct dog));
fido->name = "fido";
fido->bark = arf;
// ...
fido->bark();      // prints "arf"
```

Just like `structs` in C are the ancestors of objects in object-oriented languages like Java, function pointers in `structs` are the ancestors of methods. There are differences. For example, each instance has a separate reference to the "method" (function), and instances can change the value of these function pointers, which is equivalent to redefining the method for that instance. And each instance can do that separately from the other instances of its "class" (which is possible in some languages, for example Javascript, but not in Java). And so on. C clearly isn't an object-oriented language, but you can see the way object-oriented capabilities could be implemented using pointers, both dynamically allocated data and function pointers.

Chapter 9

Defining New Types

In Java and other object-oriented languages, when you define a class it becomes a type that you can use for variables and function declarations, thereby extending the type system. C has a different mechanism that you will certainly see used and may want to use yourself. It's called `typedef`, and it allows you to define a new type name. For example:

```
typedef int Distance;
```

This makes `Distance` a synonym for `int`. Note the somewhat awkward order of the original and new types (think of `Distance` as the "variable" being defined to be an `int`).

You can then use the new type name anywhere you could use an `int`. For example:

```
Distance d;
Distance *dp;
Distance distances[100];
```

One common usage, from *K&R*, is:

```
typedef char* String;
```

Defining more specific type names can make your code clearer. Rather than a bunch of `float`s, you can have `Distance`s, `Area`s, `Volume`s, and so on. But

those are all primitive types under the hood, and most programmers use variable names to make their intent clear. C programmers don't generally appreciate seeing `String` rather than `char*`, especially since you don't have nearly the same level of builtin support for strings in C as in other languages.

It is quite a bit more useful when you use `typedef` with structured types (`structs`). For example, using our dynamically allocated linked list example from the last section, we could write:

```
typedef struct node {
    int value;
    struct node* next;
} Node;
```

And then we can use the type `Node` rather than `struct node`. For example:

```
Node* head = (Node*)malloc(sizeof(Node));
```

This idiom makes C structured types look more like object classes and makes your code significantly more readable.

You could also `typedef` the pointer to a `struct`, as in:

```
typedef struct node* Node;
struct node {
    int value;
    Node next;
};
// ...
Node head = (Node)malloc(sizeof(struct node));
```

The first declaration uses what's called a partial structure definition. It says that `struct node` is a kind of `struct`, which is enough to define a pointer to it. Then the full structure definition follows, and can use the `Node` type.[1] This is pretty close to looking like Java.

[1]Note that you have to use `struct node`, the structure type, in the call to `malloc`. If you use `Node`, that's a pointer type, and you'll only allocate enough memory for a pointer, not for the structure (see Section 8.7).

Finally, *K&R* has a nice example of using `typedef` with function pointer types. Recall from the previous section that these are awkward things like "pointer to function of one `int` returning `int`." You can define a type for such pointers as follows:

```
typedef int (*PFII)(int);
PFII the_func = times2;
```

Here `PFII` stands for "`P`ointer to `F`unction", then the first "`I`" for the `int` argument and the second "`I`" for the `int` return value. Another example, adapted from *K&R*, is a pointer to a function of two strings (`char*`) that returns `int`:

```
typedef int (*PFSSI)(char *, char*);
PFSSI strcmp, numcmp;
```

This form of `typedef` is often used for the types of callback functions, whose meaning might otherwise get lost in all the parentheses and asterisks.

Chapter 10

Sharing Code: Files and Libraries

In our "Hello World" program, the first line after the comment was

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

We commented that this was similar to an `import` statement in Java in that it made some library functionality available to your program. Now it's time to explain this in more detail, since it is used in every program and is crucial when developing larger projects.

10.1 The C Preprocessor

The `#include` statement, and indeed any statement that starts with "#", is a *preprocessor directive*. Recall from Section 3 and Figure 3.2 that the C preprocessor processes your source code before it is seen by the C compiler. There are many preprocessor directives but I will discuss only two them in this document: `#include` and `#define`.

When the C preprocessor sees a `#include` directive, it will process the entire contents of the given file at that point in processing your source code. Thus the C compiler will see whatever of your program came before `#include`, then contents of the included file, then whatever of yours came after. If the given

filename is enclosed in regular double-quotes, then the preprocessor looks in the current directory for the file. If the filename is enclosed in angle brackets, as in our `stdio` example, then the preprocessor looks for it in the standard system directories.

In our example, we want to use the function `printf` from the `stdio` (Standard input/output) library. We could `#include` the source code for the function itself. The C compiler would compile it in addition to compiling our code. However it doesn't really make sense for us to recompile `printf` every time we compile my program since we know `printf` hasn't changed. In the first place, it might be large and slow to compile. Its code might be complicated, including variables and other functions. This code might conflict with our own code when literally included by the preprocessor. And finally, someone might want to provide access to a library function for use without making its source code available.

So what happens instead is that the included file typically contains only the *declarations* of the variables and functions provided by the library. This is enough for the C compiler to check that you are using them properly, for example, that arguments are of the correct types and that return values are being used correctly. By convention, these definitions are provided in a file whose name ends with "`.h`", called a "header" file. The actual source code for the *implementation* of the library functions is in a separate C source file, which incidentally also `#includes` the header file to keep its declarations in sync.

The following are typical contents of header files:

- Function declarations, but not their implementations. This is a bit like an interface declaration in Java:

```
int printf(char *, ...);
int strcmp(char *, char *);
```

- Variable declarations marked with the `extern` keyword:

```
extern int errno;
extern FILE *stdin, *stdout, *stderr;
```

  The `extern` keyword tells the compiler not to generate code to allocate space for storing the variable. Rather, the name will be "linked" with its storage later (see below).

- Macros defined using the `#define` directive. Simple macros simply tell the preprocessor to replace a name with some other text. For example:

```
#define MAXLEN 255
// ...
char buffer[MAXLEN];
```

  While this example could also be done with a `const int` variable, much more complicated macros are possible, including macros that take arguments. Also note that because the preprocessor literally replaces occurrences of `MAXLEN` with `255`, there is no variable with that name and no storage allocated for it. It literally doesn't exist at run-time.

- Other `#include` directives, which are themselves included inline before the preprocessor resumes processing the original header file.

10.2 Separate Compilation, Libraries, and Linking

So you have your program source code, say `hello.c`, and you understand that it uses the preprocessor `#include` directive to include definitions of library functions like `printf`. This is enough for the C compiler to compile your C code to object code (executable instructions). But before the program can be run, it needs the object code for `printf`. This will have been separately compiled from the `printf` source code and is stored in a system library.

Recall from Section 3 and Figure 3.2 that the process of combining several object code files into an executable program is called *linking* and is performed by the system *linker*. For historical reasons, the linker is usually named `ld`, which comes from "loader," which meant loading a program into memory for execution. Usually the linker is invoked automatically by the C compiler `cc` to put together the final program, although some *dynamic linking* typically occurs when the program is run.

The details vary, but one of the main jobs of the linker is to resolve references to variables and functions defined in header files. That is, for every use of `printf`

in your program, the linker must substitute code to invoke the actual implementation of the function in the library's object code. Most C compiler toolchains automatically link your code against the C standard library. For other libraries, you may need to tell the linker which library to use and possibly where to find it.

10.3 Standard System Libraries

The following is a very brief description of some of the more commonly-used C standard library functions, grouped by header file. You can find more documentation online or often by using the `man` command from your command shell (terminal).

- `stdio.h`: The Standard input/output (I/O) library, provides the `printf` and `scanf` families of functions as well as the `FILE` type and functions that involve files such as `fopen`, `fclose`, `fread`, and `fwrite`.
- `stdlib.h`: The C Standard Library, including `malloc`, `calloc`, and `free`, and other very fundamental functions like `exit` and `abort`. Also includes random number functions `random` and `srandom` (now preferred over `rand` and `srand`).
- `string.h`: String functions such as `strlen`, `strcmp`, and `strcpy`. Be sure that you understand how these work based on C's convention that strings are simply arrays of `char` with a `NUL` character indicating the end of the string.
- `math.h`: Math functions corresponding to Java's `Math` class: `sin`, `cos`, `tan`, `log`, `pow`, and so on. Now you know where Java got its math functions.

As we have already mentioned, there is no standard library support for any of what Java calls "Collections": lists, trees, sets, hashtables, and so on. Luckily these are easy enough to build yourself if you need them.

Neither does C support "generic" functions, which can be applied to different types of arguments in a type-safe way. You can accomplish some of the same things using macros if you have to, which after all is how C++ got started. These days there are probably better tools for that job than C.

10.4 Project Development

Separate complication and linking aren't just for system libraries. You can use them yourself for larger programs and projects.

You can break your code into separate files that are re-compiled only when they change. Back in the day when there was no such thing as an IDE and compiling a file took significant time, this was absolutely necessary. These days, IDEs often look after this for you and anyway, the compiler is now generally pretty fast. Still, large projects such as you may find in the open source community may involve hundreds of thousands of lines of code. These will be broken into separate files to speed compilation and allow different developers to work on different parts of the codebase. By the way, this history is the reason Java requires each public class to be defined in its own source file (other object-oriented languages do not require this, and some people hate it).

If your different source files need to share definitions, you can write header files that can be included where necessary. If you develop useful macros, data structures, and functions, you can share them between projects. You can even build your own libraries and link against them if you like.

Here's a quick example of this, for a data strucure representing a point in 2D space. I assume that you've read Section 8.7 on dynamic data structures and Chapter 9 on defining new types using `typedef`.

The header file `Point.h` defines the type `Point` and the functions related to points:

```
/* File: Point.h */

typedef struct point *Point;

extern Point new_Point(int x, int y);

extern int Point_getX(Point this);
extern int Point_getY(Point this);
```

The implementation file `Point.c` provides the definition of the `struct point` behind the type `Point`, and the implementations of the functions related to points. Note that it includes `Point.h` since it needs the definition of type `Point` and it needs to be consistent with the function definitions:

```
/* File: Point.c */

#include <stdlib.h>
#include "Point.h"

struct point {
    int x;
    int y;
};

Point new_Point(int x, int y) {
    /* Note: Do not use sizeof(Point) for the size
     * requested from malloc. The type 'Point' is a
     * pointer, you need space for a 'struct point'.
     */
    Point this = (Point)malloc(sizeof(struct point));
    this->x = x;
    this->y = y;
    return this;
}

int Point_getX(Point this) {
    return this->x;
}

int Point_getY(Point this) {
    return this->y;
}
```

Finally, our program that uses points is in `pointprog.c`. It has the code for a `main` function that allocates an instance of a point, gets its properties using its accessors, and prints them out. Note that it also needs to include `Point.h` (but not `Point.c`—it just needs the definitions, not the implementations).

```
/* File: pointprog.c */

#include <stdio.h>
#include "Point.h"

int main(int argc, char* argv[]) {
    Point p = new_Point(123, 456);
    int px = Point_getX(p);
    int py = Point_getY(p);
    printf("px=%d, py=%d\n", px, py);
}
```

Any source file that needs to know about points must include `Point.h`. But `Point.c` only needs to be compiled once.

The following command will compile both of the C implementation files and link them together into an executable named `pointprog`:

```
    gcc -o pointprog Point.c pointprog.c
```

Run the executable and it will print out the coordinates of the point that it created.

The next section describes some tools for automating separate compilation and linking the resulting object code files into an executable.

Chapter 11

Building Larger C Programs

This section describes very briefly how to use the `make` command to automate the compilation and linking of your programs. Even if you prefer to use an IDE, I think it's useful to know how `make` works since it shows you what's going on the under the hood.

The `make` command looks for a file named `Makefile` in the current directory. This `Makefile` provides variable definitions and rules for "making" targets from their dependencies. For C program development, this is primarily making executable programs from C program source files.

The key to using `make` is knowing what to put in your `Makefile`. Here's a simple example:

```
hello: hello.c
        cc -o hello hello.c
```

This is an example of a single `make` rule. It says that the target `hello`, which is the name of our executable program, depends on the file `hello.c`. If the target does not exist or is older than any of its dependencies, then `make` will use the given command(s) to try to rebuild it. So in this case, `make` will run the `cc` ("compile C") command to recompile `hello.c` whenever necessary.

A couple of things to note:

- `make` will test that dependencies are up to date and attempt to re-make them if neccessary before deciding about a target. In this example, the ex-

ecutable file `hello` depends on the source file `hello.c`, but `hello.c` itself doesn't depend on anything.

- The lines of the rule after the first one are the body of the rule. These are the commands that make should invoke to remake the target. They are indented with a single `TAB` character. The rule ends at the first non-indented line.
- In the example, note the use of the `-o` option to set the name of the resulting executable.

When you invoke `make`, it will try to build the first target defined in the `Makefile`, or you can tell `make` which target to build on the command-line. To see what `make` will do without actually doing it, you can add the option `-n` before any target names.

`make` comes with many built-in ("implicit") rules. In particular, it knows how to make a `.o` file from the corresponding `.c` file and it even knows how to make a program that comes from a single correspondingly-named `.c` file. This means that our `Makefile` really only has to say what the target is:

```
hello:
```

Try it. By default, `make` prints the commands as it executes them, and you will see that it executes some default rule involving `cc` to re-compile your program (or it will report that it is up-to-date, in which case delete the executable and try again).

Let's see a slightly more complicated example. Suppose you've split your source code into two C files, `scanner.c` and `util.c`. We want to build these into an executable named `scanner`.

```
PROG = scanner
SRCS = scanner.c util.c
OBJS = ${SRCS:%.c=%.o}

$(PROG): $(OBJS)
        $(CC) -o $(PROG) $(OBJS)
```

This example illustrates several additional features of `make`:

- Variables: `PROG`, `SRCS`, and `OBJS`. Set a `make` variable with =. Get its value using a `$` and the name in parentheses (or curly brackets). All `make` variables are string-valued.
- A substitution pattern which initializes the value of the variable `OBJS` to be the value of the variable `SRCS` with any occurrence of "`.c`" replaced by "`.o`". There are many other ways to manipulate text in `make`.
- The variables are used in the `Makefile`'s one rule. Variable names must be in parentheses or curly backets if they are longer than one character. So this rule expands to:

```
scanner: scanner.o util.o
        $(CC) -o scanner scanner.o util.o
```

- Built-in variables, like `$(CC)`, which are pre-defined to reasonable values for the platform, or you can redefine them.

This `Makefile` therefore says that the executable `scanner` depends on two object files: `scanner.o` and `util.o`. `make` already knows how to build those two files from their corresponding `.c` files, so it will do that first if necessary. Then it will link the two object files using the `cc` command to produce the executable `scanner`.

This is will leave the object files lying around. If you don't like that, add another rule to the `Makefile`:

```
clean:
        -rm $(OBJS)
```

This says that the target "`clean`" depends on nothing, so `make` will always run the rule's body in order to build the target. The body is the `rm` command to remove the object files. The "`-`" in front of the command tells `make` not to exit if the command fails, which it might if, for example, the object files haven't been made yet. To run this, just type "`make clean`" (or "`make -n clean`" to check your `Makefile`).

There is much, much more to `make`. See the manpage or, probably better, the online GNU Project document for it: https://www.gnu.org/software/make/.

Chapter 12

Debugging a C Program

Debugging is a crucial skill for any programmer. Most aspects are not specific to any programming language (for example, the practice of test-driven development). In this section I'll mention just a few tips about debugging C programs.

I'm a big believer in "print statement" debugging. That is, print the values you have and maybe what you expect and watch the output go by. This is generally easier and more informative than working with a debugger, but opinions differ.

To print out a pointer, modern C compilers provide the "`p`" format specifier for `printf`:

```
printf("pointer is %p\n", ptr);
```

The traditional way of doing this was to print the pointer in "long hexadecimal" (base 16) prefixed by "0x" (which is traditional but optional):

```
printf("pointer is 0x%lx\n", ptr);
```

You can use decimal if you like (format letter `d` rather than `x`), but hex is shorter and usually all you want to know is whether two pointers are the same or not. With modern C compilers and restrictive settings, you may need to cast the pointers you are printing, or just use "`%p`".

12.1 Debuggers

If you need a debugger and are not using an IDE, you would use `gdb` to go with `gcc` (and `lldb` to go with `llvm`/`clang`). The debugger generally has a complex command shell syntax. Some useful commands for you to investigate are `run`, `break`, `backtrace` (`bt`), `step`, `continue` (`c`), and `quit`.

Remember that there are no exceptions in C. The analogue (sort of) is something called "signals." Especially for beginners, most signals that your program might receive are due to fatal errors. You can register signal handlers and this can sometimes be useful or even necessary. But I don't think you'll run into it until you're a C expert. A debugger will catch the signal and let you investigate the state of your program.

Full documentation of `gdb` is at the GNU Project website: https://www.gnu.org/software/gdb/. For `lldb`, it's at the `llvm` website: http://lldb.llvm.org.

You usually need to give the `-g` option to the compiler when compiling in order for it to include the symbol information required for symbolic debugging. If you get no variable names in the output, just octal or hex addresses, that's probably the problem.

12.2 Compiler Options

While I'm mentioning compiler options, let me suggest that you also always use the options "`-Wall -Werror`" with `gcc` or `clang`. These mean, respectively, to report all possible warnings, and to make all warnings into errors so that the program will not compile with any warnings. At least for beginning C programmers, warnings indicate that you are doing something that is not recommended but technically not forbidden (usually for historical reasons). You shouldn't have to do that. I'll also mention the option "`-std=c99`" which causes the compiler to use the C99 language standard. I strongly recommend that you stick with the C99 core as you're getting started. There are other possibilities that you can investigate as your C skills improve.

12.3 valgrind

The biggest source of problems for Java programmers programming in C, and indeed for C programmers in general, is memory management. As you program in C, you quickly appreciate why more recent languages include some form of automated memory management. But in C, you're on your own. I have two suggestions.

First: Design your dynamic data structures carefully. As a Java programmer, you understand about things like constructors, getters and setters, and the safety of a strongly typed language. So write your C code that way. For a structured type (a `struct`) named "`Thing`," have a "constructor" function `new_Thing`, a "destructor" `free_Thing`, use getters and setters (*e.g.*, `Thing_get_x`, `Thing_set_x`), and avoid casts outside constructors unless absolutely necessary (*e.g.*, to mimic generics). Together with strong compiler options, this will prevent many problems and will help you track down the problems that sneak through.

Second: Find (or install) and use the `valgrind` program on your system. Among other things, `valgrind` tools "can automatically detect many memory management and threading bugs, and profile your programs in detail." Valgrind will swap out your platform's implementation of `malloc`, `free`, and the rest and substitute its own versions. These run much more slowly than the originals, but they track every piece of memory that you allocate and/or free. You get a report including memory leaks (memory that you allocated but forgot to free), as well as places where you wrote to memory improperly (for example, off the end of an array). You then fix those issues until `valgrind` reports no errors. You can find valgrind at `valgrind.org`. Use it.

Chapter 13

Final Thoughts

We've covered a lot of ground. Let me leave you with a few thoughts.

First: Don't freak out. C is really very similar to Java.

Ok. You're not freaking out. Where to start? You should start any project with a design phase. What information does my program need to keep track of? What kinds of things is it dealing with? How are those things related?

As an example that I use in my intro programming classes, think about a program that lets a user manage their financial information. What does it need to keep track of? Answer: how much money you have (or owe) and where it is. So I would probably start with financial accounts. There are different kinds of accounts, for example bank accounts, credit card accounts, brokerage accounts, and so on. Sounds like the start of a class hierarchy. Or maybe just do bank accounts first and make notes for later generalization. What else? People hold (own) the accounts. The accounts are with institutions like banks and brokerages. The accounts have transactions (of various types), they have a balance (which is a function of the transactions), and so on. You get the idea. You're designing the model of the world that you will build inside the computer so that it can solve problems for you.

In Java, you would take your design and write some classes. Classes define reference types and values of the type refer to instances of the class. In C, you will write some structures, and pointers to instances of your structures are references.

In place of constructors, write a function to allocate, initialize, and return an instance of each kind of structure. Don't call `malloc` or `calloc` outside of these

"constructors."

Next, write functions that get and set the members of the structure (properties of the object). These functions should be named as described earlier in this document to avoid conflicts over member names. These will be quite repetitive, just like Java getters and setters. Of course you don't have to have getters and setters (just like Java), but it does help you focus and allows you to change the implementation of the data structure later without having to change the code that uses it.

That will actually get you pretty far. Now you have to write the code that actually solves whatever problem it is you're working on. But there's nothing specific to C about that. In fact, you could do it in Java and almost copy the code into your C program with only fairly minor changes.

One last suggestion: don't call `free` unless or until you have to. If you don't call `free`, you can't have dangling pointer bugs. Of course you could still have bugs with your pointers. But these are the exact same bugs that would show up in Java as `NullPointerException`s or incorrect performance. For many small, course-sized projects, you may not ever run out of memory (except perhaps for AI classes). Of course as a Computer Scientist you should probably understand how to manage the memory used by your programs. But it wouldn't be the first thing I worried about on a project.

Bottom line: Start with a design. Build something simple that works and incrementally make it better. Just like you would if you were programming in Java...

Chapter 14

References

Kernighan, Brian W., and Ritchie, Dennis M. (1988). *The C Programming Language, 2nd Edition.* Prentice Hall. ISBN: 978-0131103627

https://en.wikipedia.org/wiki/C_(programming_language)

https://en.wikipedia.org/wiki/Java_(programming_language)