C for Java Programmers: A Primer

Charlie McDowell

University of California, Santa Cruz

Preface

This primer is designed to be used as a quick introduction to C for programmers already familiar with Java. It is not a replacement for a reference book on C, but is instead a supplement. For the programmer already familiar with Java, the typical book on C requires the reader to wade through many details of already familiar material. In this primer, we quickly present the main concepts needed to begin writing serious programs in C, highlighting the differences between C and Java.

This primer was motivated by the relatively new arrival of students that are learning Java as a first programming language. Many of these students will also need to learn C. This primer was designed to ease the transition from Java to C.

Chapter 1

Introduction

For the language features that are common to both C and Java, the two languages usually have similar if not identical syntax. One of the major differences between the two languages is that C is not objected oriented. This means that a program is not a collection of classes. Instead, a C program is a collection of functions. *Functions* in C are very similar to static methods in Java. In this chapter we introduce the basic structure of a C program, including built-in types, variable declarations, simple output, conditionals, and loops. As you will see, most of these features are nearly identical to their counterparts in Java.

Let’s begin with the classic “Hello, world!” program.

```
/* hello.c
 * Author - Jane Programmer
 * Purpose - print hello
 */
#include <stdio.h>

int main(void) {
  printf("Hello, world!\n");
  return 0;
}
```

Dissection of hello.c

-
  ```
  /* hello.c
   * Author - Jane Programmer
   * Purpose - print hello
   */
  ```

This is a comment, just as in Java. Most C compilers also allow the `//` single line comment style.

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

This is similar in purpose to the import in Java, however, the effect is somewhat different. The Java import statement tells the Java compiler where to look for additional classes. The `#include`, is called a compiler directive. It tells the compiler to insert the contents of the named file, `stdio.h` in this case, directly into the program before doing the main phase of compilation. In this example the `#include` is used to include information about the standard library routine `printf()`. This will be discussed in more detail when we discuss functions and parameters.

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

This declares `main()` to be a function that returns an `int` and has no parameters. The keyword `void` is optional, and is used to emphasize that no parameters are expected. The body of the function is defined between braces, just as in Java. Many programmers place this opening brace on a line by itself. Either style is fine, however, you should try to be consistent.

-
  ```
  printf("Hello, world!\n");
  ```

As used here, `printf()` is similar to `System.out.print()` in Java. We will discuss `printf()` in more detail later.

-
  ```
  return 0;
  ```

By convention, a return value of 0 from `main()` indicates normal completion to the operating system. If the program is terminating abnormally, a non-zero return value should be used. This error value can be used by the operating system.

To compile this program from a command line use:

```
os-prompt>gcc hello.c -o hello
os-prompt>
```

Depending upon the configuration of your computer, instead of the command `gcc`, you may need to use `cc` or some other command to compile the program. This will create a file *hello*, which can be executed by simply typing the name of the file.

```
os-prompt>hello
Hello, world!
os-prompt>
```

1.1 Built-in types, local variables, loops and conditionals.

Most Java primitive types have equivalent types in C. The only exception is the Java type `boolean`. C does not contain a boolean type. Instead, C uses integers to represent boolean values, interpreting the integer value zero as false and any other integer value as true.

Trouble spot: *C has no type* boolean. *Use* `int` *instead, interpreting non-zero as true and zero as false.*

Local variables for the built-in types in C are declared just as in Java. The same initialization syntax is also allowed. The one difference is that variables must be declared at the beginning of a block or compound statement.

Trouble spot: *All variable declarations in a block must appear before the first executable statement.*

C includes most of the same numeric, relational and boolean operators as Java, summarized in the following table.

Arithmetic operators	`+`, `-`, `*`, `/`, `%`
Relational operators	<, <=, >, >=, ==, `!=`
Boolean operators	&&, `||`, `!`
Bitwise operators	&, `|`, ^, `!`, <<, >>

The boolean operators operate on integer values, treating zero as false and non-zero as true. A boolean operator always results in a value of 0 or 1. The division operator, /, is the same as in Java. In particular, integer division truncates towards zero without any warning. Also, as in Java, the boolean operators are short-circuit operators, only evaluating their second operand if necessary.

The syntax for `if-else` statements, `for` loops, and `while` loops is identical to that of Java. The following program demonstrates the substantial syntactic similarity between C and Java.

```
/* basicSyntax.c
 * Author - Charlie McDowell
 * Purpose - demonstrate local variables,
 * built-in types, if-else, and loops
 */
#include <stdio.h>

int main(void) {
  int num1 = 4, num2 = 10, i;
  double ratio;

  for(i = 1; i <= num1; i++)
    printf("%d ",i);
  printf("\n");

  while(num1 > 0) {
    printf("%d ",num1);
    num1--;
  }
  printf("\n");

  if(num1 != 0)
    ratio = num2 / num1;
  else
    ratio = num2 / 3.0;
  printf("%f\n",ratio);

  return 0;
}
```

Dissection of basicSyntax.c

-
  ```
  int num1 = 4, num2 = 10, i;
  double ratio;
  ```

Here we declare three integer variables of type `int`, and one floating point variable of type `double`. Unlike Java, these types do not have a fixed size, however, in general they will be the same as the Java types. Two of the variables have been initialized, the others contain undefined data. Unlike Java, C will not give an error if you attempt to use a local variable that has not been initialized. The C program will simply use whatever value is stored in the memory location for the variable. That value could be anything.

Trouble spot: *C compilers will not normally warn you about uninitialized variables.*

-
  ```
  for(i = 1; i <= num1; i++)
    printf("%d ",i);
  printf("\n");
  ```

The syntax for the `for` loop in C is nearly identical to that in Java. The one difference is that the loop index variable cannot be declared as part of the `for` loop. In Java, you could write:

```
for(int i = 1; i <= num1; i++)...
```

This is a syntax error in C. As shown in this example, Java includes the increment operator. It can be used either postfix or prefix, and has the exact same behavior as in Java. The `for` loop above uses an additional feature of the `printf()` function. The string passed to a `printf()` can contain special formatting characters which are always preceded by the character percent. The sequence `%d`, says that the next parameter to the `printf()` should be converted from an integer to a string and inserted at this point. The space after the `%d` is used to separate the numbers in the output. The last `printf()` above simply prints a newline.

-
  ```
  while(num1 > 0) {
    printf("%d ",num1);
    num1--;
  }
  printf("\n");
  ```

The syntax for the `while` loop in C is identical to that in Java. As shown here, Java also includes the decrement operator `--`, which can be used as either prefix or postfix, having the exact same behavior as in Java.

-
  ```
  if(num1 != 0)
    ratio = num2 / num1;
  else
    ratio = num2 / 3.0;
  printf("%f\n", ratio);
  ```

The syntax for `if` statements and `if-else` statements is identical to that in Java. There is one important difference because C does not contain a boolean type. The result of the relational operators such as `>`, `<=` and `==` is a value of type `int`. The value will be 1 if the comparison is true and 0 if it is false. This will become important later, because many C programmers use this fact. For example, the above `if-else` statement could be rewritten equivalently as:

```
if(num1)
  ratio = num2 / num1;
else
  ratio = num2 / 3.0;
```

The format string `%f` in a `printf()` statement indicates that the next parameter to the `printf()` should be converted from a floating point value to a string and inserted in the output string at that point.

All of the rules about using braces to place blocks of statements in the branches of an `if-else` statement are the same as in Java. Also, nesting of `if-else` statements is the same in C as in Java.

Trouble spot: *In Java the following is a syntax error:*

```
    if(num1 = 0) ...
```

The expression `num1 = 0` assigns zero to `num1` and the value of the expression is 0. Zero is interpreted as false and the false branch of the `if` statement will be taken. In Java this is a syntax error because the expression does not result in a boolean value. This is a particularly insidious error in C because it is so close to what was probably intended:

```
    if(num1 == 0) ...
```

One approach used by many programmers to help catch such typing errors, is to make a habit of writing

```
   if(0 == num1) ...
```

instead. Then if the `==` is mistyped as `=`, a compile time error will be generated.

1.2 printf()

The standard output function in C is `printf()`. The name `printf` is derived from the fact that this function allows for formatted printing. The function `printf()` can accept any number of arguments. This looks a bit like function overloading, but C does not support function overloading. Instead, the function `printf()` is designed to accept an arbitrary number of arguments.

The first argument to `printf()` is always a format string. The format string is then followed by zero or more additional parameters that contain the values that are to be printed, as part of the output string.

The format string contains both literal characters to be printed, and formatting characters. A sequence of formatting characters always begins with the special character `%` and ends with a conversion character. Between the `%` and the conversion character can be additional formatting characters. The following example shows how literal characters and several conversion specifications can be combined in a single `printf()`.

```
/* printfExample.c
 * Author - Charlie McDowell
 * Purpose - simple use of printf
 */
#include <stdio.h>

int main(void) {
  int intVal = 10;
  double doubleVal = 1.234;

  printf("intVal = %d, doubleVal = %f\n", intVal,
         doubleVal);

  return 0;
}
```

The output of this program is:

```
intVal = 10, doubleVal = 1.234
```

Notice how the values of the two variables were inserted into the output at the points where the format conversion strings `%d` and `%f` appeared. The following table lists all of the conversion characters for `printf()`.

conversion character	*argument*	*description*
c	char	Print a single character.
d or i	int	Print an integer.
u	int	Print as an unsigned integer.
o	int	Print an integer in octal.
x or X	int	Print an integer in hexadecimal.
e or E	float or double	Print in scientific notation.
f	float or double	Print a floating point value.
g or G	float or double	Same as e, E, or f, whichever uses fewest characters.
s	char*	Print a string.
p	void*	Print a pointer's value in hexadecimal.
n	int*	Nothing is printed, instead store into the argument the number of characters printed so far.
%	none	Print a single %, there is no corresponding argument.

The basic formatting specified by the conversion characters shown in the previous table can be modified with additional formatting characters inserted between the % signaling a format string and the conversion character which ends the format string. This additional formatting information may include the following, in order: flags, field width, precision, and length modifier, as described below.

- flags - There are several optional flags that can be used to control left/right justification, fill character and related formatting information. They are described in the following table.

flag	*description*
`-`	The output string will be left-justified in the field. The default is right-justification.
`+`	For conversions that might generate a minus sign for negative values, the + is normally omitted for positive values. With this flag, the + is included.
space	Inserts a space in front of non-negative numeric output when using a conversion that might generate a minus sign. This is similar to the `+` flag. It is used to align the first digit when the output includes both negative and positive values. If both the `+` and space flags are included the space flag is ignored.
`#`	The effect of this flag depends upon the conversion character. With conversion character `o`, a zero is printed at the front of the output string. The leading zero indicates to the reader that this is in octal format. With conversion characters `x` or `X`, the string `0x` or `0X` is printed at the front of the output string indicating hexadecimal to the reader. With conversion characters `g` or `G`, trailing zeros are printed. With conversion characters `e`, `E`, `f`, `g`, or `G`, a decimal point is always printed.
`0`	Numeric conversions will use leading zeros instead of spaces to pad the output to fill the specified field.

- field width - Each converted argument is printed in a field. By default, the field is just wide enough to hold the converted value. When a field width is specified, the output is padded on the left with zeros or spaces, or on the right with spaces, depending on the flags. If the field width is too small, it is extended to the width of the converted argument.
- precision - A precision is specified by a period followed by a nonnegative integer. The precision is interpreted differently for different conversion characters,

as shown in the following table.

Conversion characters	*Effect of precision specification*
`d`, `i`, `o`, `u`, `x`, `X`	The precision specifies the minimum number of digits that will be printed. Leading zeros will be printed as necessary.
`e`, `E`, `f`	The precision specifies the number of digits that will be printed to the right of the decimal point.
`g`, `G`	The precision specifies the maximum number of significant digits that will be printed.
`s`	The precision specifies the maximum number of characters that will be printed from the string.

- length modifier - An l (that is the letter el) placed just before a d, i, o, u, x, or X conversion character indicates that the corresponding argument to be converted is a `long int`. Similarly an h placed before those same conversion characters indicates that the argument is a short int. An L placed before an e, E, f, g, or G indicates that the argument is a `long double`.

To summarize, if we let f stand for flags, w for width of the field, p for precision, l for length modifier and c for conversion character, we see that a conversion string has the general form:

```
%fw.plc
```

where only the % and the conversion character represented by c are required. The following example demonstrates a few of the many formatting possibilities.

```
/* printf.c
 * Author - Charlie McDowell
 * Purpose - demonstrate a few formatting possibilities
 */
#include <stdio.h>

int main(void) {
  int anInt = 123;
  float aFloat = 1.234567;

  printf("1234567890,1234567890\n");
  printf("%+10.4d,%-+10.4d\n", anInt, anInt);
  printf("%010d,%#x\n", anInt, anInt);
  printf("%-10.4f,%-+10.7f\n", aFloat, aFloat);
  return 0;
}
```

The output of this example is:

```
1234567890,1234567890
     +0123,+0123
0000000123,0x7b
1.2346    ,+1.2345670
```

1.3 scanf()

The function `scanf()` is similar in form to `printf()` but it is used for input instead of for output. Here is a simple example that uses `scanf()` to read an integer, and a floating point value from the standard input device, usually the keyboard.

```
/* scanfExample.c
 * Author - Charlie McDowell
 * Purpose - simple use of scanf
 */
#include <stdio.h>

int main(void) {
  int intVal;
  float floatVal;

  scanf("%d", &intVal);
  scanf("%f", &floatVal);

  printf("intVal = %d, floatVal = %f\n", intVal,
         floatVal);

  return 0;
}
```

The format strings for `scanf()` are very similar to those for `printf()`. In the example above we use `%d` to indicate that an `int` is to be read from the standard input device and `%f` to indicate that a `float` is to be read. These input conversion strings will automatically skip over any white space characters. The white space characters are newline, space and tab. Notice that the variables into which the data is to be stored are preceded by the & character. The & is the “address of” operator. It will be explained in more detail later. It must always be used when reading built-in types into simple variables. The function `scanf()` needs to know *where* to store the values read, hence the function is passed the “address of” the variables where the values are to be stored. It is not possible in Java to write a function like `scanf()` that has multiple output parameters that are primitive types.

Trouble spot: *If you forget the & in front of one of the parameters to scanf() your program will most likely abort with some obscure message. On Unix systems the message will probably be something about a segmentation fault or bus error.*

The & in front of a variable or expression of type T changes the type of the expression from the type T to type T*. The type T* is a pointer to a value of type T. We will explain this in Chapter 2. In the following table when you see that the argument must be type T*, then that is an indication that you can place a variable of type T, preceded by & in the corresponding argument position for `scanf()`. The following table lists the conversion characters for `scanf()`..

conversion character	*argument*	*description*
c	char*	Read a single character.
d	int*	Read a decimal integer.
i	int*	Read an integer in decimal, octal, or hexadecimal. Octal is indicated by a leading 0, and hexadecimal by a leading 0x.
u	unsigned*	Read an unsigned decimal integer.
o	unsigned*	Read an octal integer. No leading 0 is required.
x or X	int	Read a hexadecimal integer. No leading 0x or 0X is required.
e, E, f, g, G	float*	Read a floating point number.
s	char*	Read a sequence of non-white characters. (Do not use this with &c, where c is a simple `char` variable. See Chapter 4.)
p	void**	Read a pointer's value in the same format as that created with p in `printf()`.
n	int*	Store into the argument the number of characters read so far.
%	none	Skip single % in the input.
[...]	char*	Read characters as in conversion character s, except that only characters specified by the scan set between the brackets are read. Any others terminate the input for this argument. The scan set is discussed further in Chapter 4.

As with `printf()`, the basic formatting specified by the conversion characters shown in the previous table can be modified with additional formatting characters inserted between the `%` signaling a format string and the conversion character which ends the format string. This additional formatting information may include the following flags, in order: assignment suppression (`*`), field width (an integer), and length modifier (one of `h`, `l`, or `L`), as described below.

flag	*description*
*	The read value is discarded. There is no corresponding argument.
an integer	This specifies the maximum number of characters that will be scanned for the current argument.
h	The length modifier h can precede a d, i, o, u, x, or X conversion character. It indicates that the corresponding argument is a `short int*`, or `unsigned short int*`.
l	Same as h but the corresponding argument is a `long int*`, or `unsigned long int*`. In addition l can precede e, E, f, g, or G in which case the argument is a `double*`.
L	The length modifier L can precede e, E, f, g, or G. The corresponding argument should be a `long double*`.

Summary

- A C program is a collection of functions. Functions are very similar to static methods in Java. C program execution begins in the function `main()`.
- Comments are indicated with `//` or `/* */`, just as in Java.
- The standard C types include the Java primitive types except for boolean.
- C represents boolean values as integers. Zero is interpreted as false, non-zero is interpreted as true.
- In general, the built-in operators for the standard C types are the same as those for the corresponding Java primitive types.
- Expressions involving relational operators evaluate to one for true and zero for false. The boolean operators operate on integers treating any non-zero value as true.

- The syntax of both `for` and `while` loops is the same as in Java.
- The syntax of `if-else` statements is the same, except that any integer expression can be used as the conditional expression.
- The standard C print function is `printf()`.
- The standard C input function is `scanf()`.

Exercises

1 Write a program that reads in two integers for the width and height of a rectangle, then prints the area of the rectangle. A Java solution is the program SimpleInput.java from Section 2.6 of JBD.

2 Write a program to compute the area of a circle given its radius. Let `radius` be a variable of type `double` and use `scanf()` to read in its value. Be sure that the output is understandable. If you include the header file *math.h* using `#include <math.h>`, then you can use the predefined constant `M_PI` to get the value of π.

3 Write a program that asks for the number of quarters, dimes, nickels, and pennies you have. Then compute the total value of your change and print the number of dollars and the remaining cents. The output format should be `$X.YY`.

4 Write a program that prompts for the length of three line segments as integers. If the three lines could form a triangle, the program prints “Is a triangle.” Otherwise, it prints “Is not a triangle.” Recall that the sum of the lengths of *any* two sides of a triangle must be greater than the length of the third side. For example, 20, 5, and 10 can’t be the lengths of the sides of a triangle because 5 + 10 is not greater than 20.

5 Write a program that will print out a box drawn with asterisks, as shown

```
*****
*   *
*   *
*****
```

Use a loop so that you can easily draw a larger box. Modify the program to read in a number from the user specifying how many asterisks high and wide the box should be.

6 Write a program that reads in numbers until the same number is typed twice in a row. Modify it to go until three in a row are typed. Modify it so that it first asks for "how many in a row should I wait for?" and then it goes until some number is typed that many times. For example, for two in a row, if the user typed "1 2 5 3 4 5 7" the program would still be looking for two in a row. The number 5 had been typed twice, but not in a row. If the user then typed 7, that would terminate the program because two 7s were typed, one directly after the other.

7 Write a program that prints all the prime numbers in 2 through 100. A prime number is an integer that is greater than 1 and is divisible only by 1 and itself. For example, 2 is the only even prime. Why?

Pseudocode for finding primes

```
for n = 2 until 100
  for i = 2 until the square root of n
    if n % i == 0 the number is divisible by i
      otherwise n is prime
```

Can you explain or prove why the inner-loop test only needs to go up to the square root of *n*?

8 Write a program that generates an approximation of the real number *e*. Use the formula

$$e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{k!} + \ldots$$

where $k!$ means k factorial = 1 * 2 * . . . *k. Keep track of term $1/k!$ by using a `double`. Each iteration should use the previous value of this term to compute the next term, as in

$$T_{k+1} = T_k \times \frac{1}{k+1}$$

Run the computation for 20 terms, printing the answer after each new term is computed

Chapter 2

Passing parameters to functions

Java and C both pass parameters using pass-by-value. This means that the value of the actual parameter is copied and stored in the corresponding formal parameter. Because the value of the actual parameter is copied, the value stored in the actual parameter cannot be modified by the function.

In Java you can get several results from a function by passing references to several objects and then modifying the objects in the function. In C the same thing is accomplished by passing pointers.

2.1 Declaring functions

Below is a simple example of parameter passing and a program with two functions, `main()` and `min()`.

```
/* min.c
 * Author - Charlie McDowell
 * Purpose - simple function definition/call
 */
#include <stdio.h>

int min(int x, int y) {
  if(x < y)
    return x;
  else
    return y;
}

int main(void) {
  int num1, num2;

  printf("Enter two integers.\n");
  scanf("%d%d", &num1, &num2);

  printf("The smaller is %d\n", min(num1, num2));
  return 0;
}
```

The function `min()` is essentially identical to its counterpart in Java. The only difference is that there is no keyword static. In the example above, we have placed the definition of `min()` before its use in `main()`. C requires that the signature of a function be known before the first call to the function. Recall that the signature is the name of the function, along with its return type and the type and number of the formal parameters. Stylistically many programmers prefer to have `main()` be the first function instead of the last. When you wish to call a function that has not yet been defined, or one that is defined in another file (see Section 6.2), you must precede the call with a function prototype. A function prototype is like a function definition with no body. Below is the above program rearranged with `main()` first, by using a function prototype.

```
/* min2.c
 * Author - Charlie McDowell
 * Purpose - simple function definition/call
 *    using a prototype to allow main() to be first
 */
#include <stdio.h>

int min(int x, int y);

int main(void) {
  int num1, num2;

  printf("Enter two integers.\n");
  scanf("%d%d", &num1, &num2);

  printf("The smaller is %d\n", min(num1, num2));
  return 0;
}

int min(int x, int y) {
  if(x < y)
    return x;
  else
    return y;
}
```

2.2 Pointers

One of the most important differences between C and Java is C's ability to manipulate pointers to data. In Java, it is not possible to create a pointer or reference to a primitive type. Java only allows pointers to objects and these pointers can never be changed other than assigning one pointer to another. To emphasize this distinction, in Java pointers to objects are called references.

In C you can create a pointer to any variable. This is done using the address-of operator &, as shown briefly in the example with `scanf()` in Section 1.3. Once you have a pointer to a variable (actually a pointer to some arbitrary memory location), you obtain the value that the pointer points to using the indirection operator *.

The following example introduces these two operators. Notice that the character * is used both to access the location pointed to by a pointer, and to declare that a variable is actually a pointer. Prepending a variable with a * in a declaration statement, declares that the variable will be a pointer to the indicated type instead of a regular variable of that type. This can be seen in the declaration of `pointerOne` and `pointerTwo` below. Prepending a variable with a * in an expression says don't use the value stored in the variable, instead use the value pointed to by the address stored in the variable. This can be seen in the first assignment to `y` in the example below.

```
/* pointer.c
 * Author - Charlie McDowell
 * Purpose - introduce pointers
 */
#include <stdio.h>

int main(void) {
  int x = 123, y;
  int *pointerOne, *pointerTwo;

  pointerOne = &x;
  y = *pointerOne;
  printf("x = %d, y = %d\n", x, y);

  pointerTwo = pointerOne;
  *pointerTwo = 99;
  printf("x = %d, y = %d\n", x, y);

  pointerTwo = &y;
  printf("*pointerOne = %d, *pointerTwo = %d\n",
         *pointerOne, *pointerTwo);

  printf("pointerOne = %u, pointerTwo = %u\n",
         pointerOne, pointerTwo);

  return 0;
}
```

Here is the output from the above program. If you run it on your system you may get different values for the numbers in the last line of output.

```
x = 123, y = 123
x = 99, y = 123
*pointerOne = 99, *pointerTwo = 123
pointerOne = 4026529588, pointerTwo = 4026529584
```

Dissection of pointer.c

-
  ```
  int x = 123, y;
  ```

Here we declare two simple variables. We will manipulate the values stored in these variables using pointers.

-
  ```
  int *pointerOne, *pointerTwo;
  ```

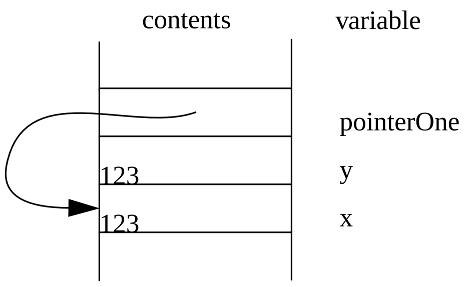
Here we declare two variables that are pointers to integers. Notice the * preceding each variable.

-
  ```
  pointerOne = &x;
  ```

Here we are storing the address of variable `x` (remember & is the address of operator) into the variable `pointerOne`. The variable `pointerOne` is now pointing at the same memory location as that referred to by the variable `x`.

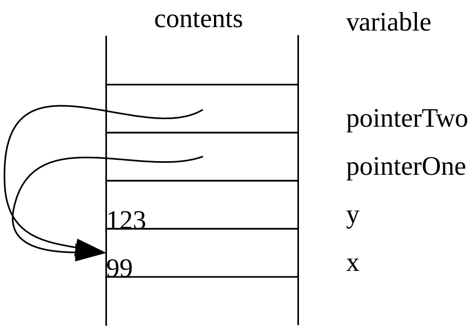
-
  ```
  y = *pointerOne;
  printf("x = %d, y = %d\n", x, y);
  ```

The assignment stores the value pointed to by `pointerOne`, into the variable `y`. Here the `*` is the indirection operator. It says that `pointerOne` does not directly refer to the value of interest, but refers to it indirectly, i.e. it points to the value. The result is that the value stored in variable `x` is copied into variable `y` as shown in the following figure. In the figure, variables that contain pointers are depicted as arrows leading to the memory that the pointer points to.



-
```
pointerTwo = pointerOne;
*pointerTwo = 99;
printf("x = %d, y = %d\n", x, y);
```

In this sequence, we first assign `pointerTwo` to point to the same location as `pointerOne`. Then we use `*pointerTwo` as the destination of an assignment. This says store into the location pointed to by `pointerTwo`. From the previous statements we can determine that `pointerTwo` is pointing to the variable `x` (as is `pointerOne`). The following figure shows the result.

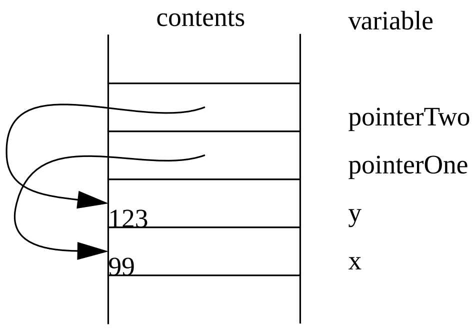


-
```
pointerTwo = &y;
printf("*pointerOne = %d, *pointerTwo = %d\n",
        *pointerOne, *pointerTwo);
printf("pointerOne = %u, pointerTwo = %u\n",
        pointerOne, pointerTwo);
```

We now change `pointerTwo` to point to y. Then we print both the values *stored in* the two pointers and the values *pointed to* by the two pointers. Looking at the output, it may seem strange that the addresses of `x` and `y` are such large numbers for such a small program. This is because most operating systems today use what is called virtual memory. Virtual memory allows a program to use some very small addresses and some very big addresses without actually requiring physical computer memory for all of the addresses in between. It is common for the memory allocated for local variables to start at the upper end of the usable virtual address space, hence the very large addresses in this example. Below is a figure showing the layout of the variables in the above example at the end of the program.

address	contents	variable
4026529576	4026529584	pointerTwo
4026529580	4026529588	pointerOne
4026529584	123	y
4026529588	99	x

In general we do not know or care about the actual address values. We only care about what locations pointers are pointing to. For this reason, it is generally more useful to depict pointer values as arrows pointing to the appropriate location as we did in the earlier figures. Using this notation, the above figure would become:



2.3 Call-By-Reference

In Java, if what is being passed is actually a reference to an object, then the instance variables inside of the referenced object can be changed, but the actual parameter referencing the object is not changed. In C, although parameters are always passed using pass-by-value, because the value that is passed can be an address, C can get the effect of pass-by-reference. Consider the following example:

```
/* swap.c
 * Author - Charlie McDowell
 * Purpose - call-by-reference
 */
#include <stdio.h>

void swap(int *x_p, int *y_p) {
  int temp;

  temp = *x_p;
  *x_p = *y_p;
  *y_p = temp;
}

int main(void) {
  int num1, num2;

  printf("Enter two integers.\n");
  scanf("%d%d", &num1, &num2);
  printf("num1 = %d, num2 = %d\n", num1, num2);

  swap(&num1, &num2);

  printf("num1 = %d, num2 = %d\n", num1, num2);
  return 0;
}
```

The output of this program with the input 99 and 123 is

```
Enter two integers.
99 123
num1 = 99, num2 = 123
num1 = 123, num2 = 99
```

Dissection of swap.c

- `void swap(int *x_p, int *y_p)`

The function `swap()` does not return a value, indicated by the return type of `void`. The function expects two parameters, each of which is a pointer to an `int`. This effectively makes `swap()` a function with two `int` parameters that are passed, call-by-reference. A good programming style is to always append `_p` to the end of variables that are actually pointers. This can help you remember that you need to use the indirection operator * when accessing the variable, unless you actually want the address.

-
  ```
  int temp;

  temp = *x_p;
  ```

The value pointed to by `x_p`, is stored in `temp`. The expression `*x_p`, represents the value stored in the location pointed to by `x_p`. See the figure below.

- `*x_p = *y_p;`

Store the value stored in the location pointed to by `y_p`, into the location pointed to by `x_p`.

- `*y_p = temp;`

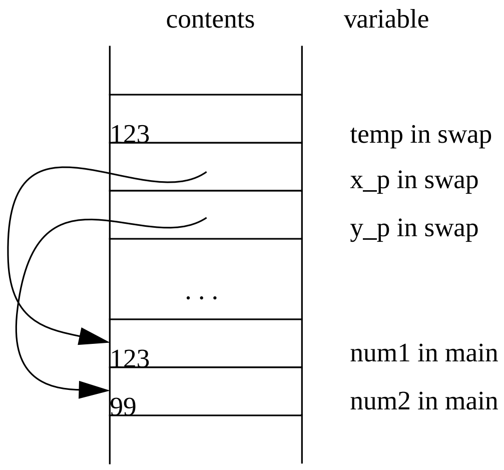
Store the value stored in the variable `temp`, into the location pointed to by `y_p`.

```
•   int main(void) {
      int num1, num2;

      printf("Enter two integers.\n");
      scanf("%d%d", &num1, &num2);
      printf("num1 = %d, num2 = %d\n", num1, num2);

      swap(&num1, &num2);
```

In `main()` when we call `swap()` we need to pass pointers to integers, not integers. The address-of operator, &, does the job. We are passing pointers to the locations storing the values for `num1` and `num2`. The result is that `num1`, and `num2` are output parameters. Their values are changed by the execution of `swap()`.

Notice that the call to `scanf()` also uses the address-of operator, &, with all but the first parameter. This is for the same reason that we used & with `swap()`. The `scanf()` function needs to be able to change the values stored in `num1` and `num2`. Without the &, we would be passing the values stored in `num1` and `num2`, and `scanf()` would not be able to change them - remember, C uses pass-by-value. By passing an address as the value, we get the effect of pass-by-reference, and the ability to have output parameters.

The following figure shows the memory contents just before the return from `swap()`.



Trouble spot: *A major source of errors in C programs, is the failure to properly use * and &.*

For example, if we forgot to use the * in the assignment to `temp` in `swap()` like this:

```
   temp = x_p;
```

we would be attempting to store not the `int` value pointed to by `x_p` (`num1` in our example), but would instead be storing the contents of `x_p` (the actual address of `num1`) into `temp`. Fortunately, any good C compiler should generate a warning message in this situation. In the case of `scanf()`, the problem is a bit different. If you forget to use the & with one of the variables passed to `scanf()`, you may only get a warning if you ask the compiler to be very strict. For most C compilers you can do this by using the flag `-Wall`, which means generate all possible warning message. We highly recommend that you always use this flag. The reason `scanf()` is a bit different is because `scanf()` uses a special C notation that indicates it doesn't really know what type of parameter will be passed except for the first, which is the format string. The contents of the format string determine what are appropriate types for the other parameters.

Summary

- Values of built-in types are passed by-value, just like Java primitive types.
- Function syntax is essentially the same as a static method in Java.
- C requires a function prototype if a function call appears in a file before the definition of the function.
- In addition to passing values, C can pass the address of any variable, achieving an effect similar to passing references to objects in Java.

- The expression `&x` is the address of `x`; a reference to the location containing the value of x. All non-primitive variables in Java are references. C allows for pointers (references) to both built-in types such as int, and user defined types.

- The expression `*x` evaluates not to the value in variable `x`, but to the value in the memory location whose address is in variable `x`. In this case, x is like a Java reference variable.

- Using the address-of operator (&) and the indirection operator (*), C can pass even built-in types such as int, using pass-by-reference. This makes it possible to have multiple output parameters that are values of built-in types. This is not possible in Java without using wrapper classes.

Exercises

1 Write a program to play the game of Twenty-One Pickup. This is a two-player game that starts with a pile of 21 stones. Each player takes turns removing 1, 2, or 3 stones from the pile. The player that removes the last stone wins. The program will play against a person and the person always gets to move first. The program should allow the person to play a series of games, keeping track of the number of games won by the person. A Java solution to this problem is presented in Section 4.9 of JBD.

2 Modify Exercise 5, on page 17, in Chapter 1, to have a function `drawBox()` that takes the width and height of the box in asterisks as formal parameters. Modify the program further to include three functions: `drawLine()`, `drawSides()`, and `drawBox()`. The function `drawLine()` will take two parameters: the length of the line and a string that will be printed length times to draw the line. The function `drawSides()` will take three parameters: the height of the sides, the width of the box that will be formed (both lines must be drawn at the same time), and a string used to represent a side. The function `drawBox()` will take three parameters: the width and height of the box and a

string used to represent the sides. For example, if called as `drawBox(5,4,"CisFun")`, the output would be

```
CisFunCisFunCisFunCisFunCisFun
CisFun                  CisFun
CisFun                  CisFun
CisFunCisFunCisFunCisFunCisFun
```

3 Write a function that can be used to sort three integers. The function will need to have three parameters that are pointers to integers. When the function returns, the three parameters should be in ascending order. Provide a program to test your function by asking the user to enter three integers, pass those integers to the function and then print them after the function returns.

4 Write a program to print the values of $F(x) = x^2 - 2$ for the range $0 < x < 10$. Print the values in this range at a step size of 0.01. Also find the largest and smallest value of *F(x)* in this interval.

5 Write a program that allows the user to play the game of Craps, which is played with two dice. A simple version of the game between one player and "the house" can be described as follows.

1 The player bets some amount of money.

2 The player throws the dice.

3 If the dice total 2 or 12 the player loses the bet and play starts again at step 1.

4 If the dice total 7 or 11 the player wins the amount of the bet from the house and play starts again at step 1.

5 If the dice total any other value, this value is called the point.

6 The player continues to roll the dice until they total either 7 or the point.

7 If the dice total is 7 the player loses the bet; otherwise, the player has made the point and wins the amount of the bet from the house. In either case, play starts again at step 1.

Play continues until the player indicates that he or she wants to quit or until the player runs out of money. Before you begin to write the code for this program, you should develop a design. Convert the description of play to pseudocode and identify the primary methods that you'll need. You may even need to refine the specification some more first. For example, how much money does the player start with?

To simulate the roll of the dice, place `#include <stdlib.h>` at the top of your source file, then the following expression

```
rand()%6 + 1
```

will evaluate to a random integer of 1 through 6. You must simulate rolling each die separately. Generating a random number in the range 2 through 12 isn't sufficient.

6 Write a program to compute the probability that we can toss some number, n, heads in a row using a coin. Compute the probability by simulating the coin toss with a random number. A Java solution to this problem is presented in Section 4.7 of JBD. The function `rand()` mentioned in the previous exercise returns a random integer in the rand `0` to `RAND_MAX`, which is defined in `stdlib.h`. You can simulate a coin toss by treating a return value from rand() that is less than `RAND_MAX/2` as a head and treating any other value as a tail.

Chapter 3

Arrays

Syntactically, arrays in C appear very similar to arrays in Java (see JBD Chapter 5), however, they are in fact quite different. In Java, all arrays are created dynamically using the `new` operator. In C, arrays can be created both dynamically, and statically.

3.1 One dimensional arrays

Let's first look at statically allocated arrays. The following statement declares and allocates an array of 10 integers. Notice the placement of the brackets after the variable name, not after the element type, `int`.

```
int smallArray[10];
```

The elements of this array can be accessed using array indexing, just as in Java (see JBD Section 5.1). For example, the following code fragment will print the elements of the above array.

```
int i;

for(i = 0; i < 10; i++)
  printf("%d\n",smallArray[i]);
```

There are several important differences between the above example, and its Java counterpart. First, C does not check to make sure that the array index is within the bounds of the array. The following will compile and run, giving unpredictable results.

```
/* arrayBounds.c
 * Author - Charlie McDowell
 * Purpose - exceed array bounds without an error
 */
#include <stdio.h>

int main(void)
{
  int smallArray[10];
  int i;

  for(i = 0; i < 10; i++)
    smallArray[i] = i*10;

  for(i = 0; i <= 10; i++)
    printf("%d\n",smallArray[i]);

  return 0;
}
```

If you look closely, you will notice that the second loop executes eleven times with i ranging from 0 to 10 inclusive, but the bounds of the array are 0 to 9 inclusive. Arrays in C do not carry around their length as they do in Java.

Another difference between arrays in C and arrays in Java, is that you cannot reassign an array variable to point to a new array. This also implies that you cannot change the size of an array. There is no way in the example above to change `smallArray` to refer to a larger array, after the initial declaration. As we will show below, you can dynamically create arrays and use variables to reference them, but those variables are actually pointers, not arrays.

Statically allocated arrays in C can also be created using array initializers which are syntactically identical to array initializers in Java. The following statement creates a static array of five integers, initialized to the values 100 through 500.

```
int x[] = { 100, 200, 300, 400, 500 };
```

This statement is legal in both C and Java. Although the preferred Java syntax places the brackets with the type int, as in

```
int[] x = { 100, 200, 300, 400, 500 }; // not legal C
```

3.2 Multi-dimensional arrays

Multi-dimensional arrays are created using multiple bracket pairs. The following creates a two-dimensional array of integers.

```
int matrix[10][20];
```

The above declaration allocates storage for 200 integers. The elements of this array can be accessed just as in Java (see JBD Section 5.9). The following will initialize the entire array to 0.

```
for(i = 0; i < 10; i++)
  for(j = 0; j < 20; j++)
    matrix[i][j] = 0;
```

Unlike Java, an array declared as a local variable will not be initialized to zero.

Trouble spot: *Most C compilers will not give you any indication if you fail to initialize the values of an array before using them.*

The results will be that you will get unpredictable values when accessing an uninitialized array.

Higher dimensional arrays are created and accessed by adding more pairs of brackets. The following declares a three-dimensional array, and initializes all elements to 1.

```
int space[10][20][30];
int i, j, k;

for(i = 0; i < 10; i++)
  for(j = 0; j < 20; j++)
    for(k = 0; i < 30; k++)
      space[i][j][k] = 1;
```

3.3 Passing arrays as parameters

An array variable in C is actually a pointer to the first element of the array. For an array of integers, it is just like a pointer to an `int`, except that you cannot change where it points. Thinking of the name of an array as a pointer to the first element, is essential for understanding how arrays are passed as parameters. Consider the following example:

```
/* arrayParams.c
 * Author - Charlie McDowell
 * Purpose - passing arrays as parameters
 */
#include <stdio.h>

void fillIt(int someArray[]);

int main(void) {
  int smallArray[10];
  int i;

  fillIt(smallArray);

  for(i = 0; i < 10; i++)
    printf("%d ",smallArray[i]);
  printf("\n");

  return 0;
}

void fillIt(int someArray[]) {
  int i;

  for(i = 0; i < 10; i++)
    someArray[i] = 100 + i;
}
```

The output of this program is

```
100 101 102 103 104 105 106 107 108 109
```

Dissection of arrayParams

-
  ```
  void fillIt(int someArray[]);
  ```

We place a function prototype (see Section 2.1) at the beginning so that the definition of the function `main()` can appear first. An array parameter is declared just like an array variable. The one difference is that for one dimensional arrays, the size of the array is irrelevant and not required. For this reason, the function `fillIt()` will work with any array of integers.

-
  ```
  int main(void) {
    int smallArray[10];
    int i;

    fillIt(smallArray);
  ```

To pass an array as a parameter, just use the name of the array. Remember, C always uses pass-by-value. The value of the expression `smallArray`, is the address of the first element of the array. In this way, the function `fillIt()` will have a pointer to the first element, and can thus change any of the elements of the array.

-
  ```
  void fillIt(int someArray[]) {
    int i;

    for(i = 0; i < 10; i++)
      someArray[i] = 100 + i;
  }
  ```

The preferred way to write a function that expects an array parameter is as shown above. You must remember that the value that is passed in this case is *not* the value of the entire array, but is only the address of the first element. The above declaration could lead you to believe that it is the array that is being passed by-value. This is not the case.

To emphasize the fact that an array parameter is simply a pointer, consider the following variation on the example above. In the example below, `main()` is *identical* to `main()` in *arrayParams.c* above.

```
/* arrayParams2.c
 * Author - Charlie McDowell
 * Purpose - show array is really a pointer
 */
#include <stdio.h>

void fillIt(int *someArray);

int main(void) {
  int smallArray[10];
  int i;

  fillIt(smallArray);

  for(i = 0; i < 10; i++)
    printf("%d ",smallArray[i]);
  printf("\n");

  return 0;
}

void fillIt(int *someArray) {
  int i;

  for(i = 0; i < 10; i++) {
    *someArray = 100 + i;
    someArray++;
  }
}
```

Notice that in this example, `fillIt()` has been declared to have a pointer to `int` as its only parameter, instead of an array of `int`. The call to `fillIt()` remains unchanged. This works because, as we said above, the name of an array, when used as an expression, is simply a pointer to the first element of the array.

The statement in `fillIt()`:

```
*someArray = 100 + i;
```

presents nothing new. It says, store the value `100 + i` into the location pointed to by `someArray`. Initially `someArray` is pointing to the first element of the array `smal-lArray` in `main()`. The next statement is a key feature of C not found in Java. The statement

```
someArray++;
```

is incrementing the value of a pointer. This type of pointer arithmetic is strictly prohibited in Java. In C, this statement causes the address stored in `someArray` to be incremented. Remember `someArray` stores an address which is interpreted as a pointer to some other location storing an integer. Incrementing the address causes `someArray` to point to the next memory location. Because arrays are allocated contiguously, that is, array elements are placed in consecutive memory locations, incrementing a pointer to an array element, causes the pointer to point to the next array element. In this way, the variable `someArray`, points to each of the elements of `smallArray`, as the loop executes.

3.4 Dynamic arrays

In addition to having statically allocated arrays, such as those discussed above, C allows for dynamic allocation of arrays. When using dynamically allocated arrays, the thin disguise of array variables as something other than pointers is completely removed. The following example dynamically creates an array based upon a user specified value.

```
/* dynamicArray.c
 * Author - Charlie McDowell
 * Purpose - create an array dynamically
 */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int *array_p;
  int size, i;

  printf("How many elements will be in the array?\n");
  scanf("%d", &size);

  array_p = calloc(size, sizeof(int));

  for(i = 0; i < size; i++) {
    array_p[i] = 100 + i;
  }

  for(i = 0; i < size; i++)
    printf("%d ", array_p[i]);
  printf("\n");
  return 0;
}
```

In the example above, `array_p` is the “array variable”, although as you can see, it is in fact a pointer to an integer. The array is created by the call to the function `calloc()` which is part of the standard C libraries. The function prototype for `calloc()` is in *stdlib.h* which is included at the beginning of the above program. The function `calloc()` takes two parameters. The first is the number of elements in the array being allocated, and the second is the size of each element.

As you can see from the above example, we can use the array indexing syntax of square brackets, with a pointer to an `int`. In fact, the following two statements are identical for all practical purposes.

```
*array_p = x;
array_p[0] = x;
```

The loop that fills the array above could have been written like this:

```
temp_ptr = array_p;
for(i = 0; i < size; i++) {
  *temp_ptr = 100 + i;
  temp_ptr++;
}
```

assuming `temp_ptr` was declared earlier to be a pointer to an integer.

In Java, dynamically allocated memory (created using `new`) is automatically reclaimed using what is called garbage collection. In C, dynamically allocated memory must be manually reclaimed by the programmer. If an array such as `array_p` above was allocated as part of a larger program, when the storage for the array is no longer needed, the programmer must deallocate the memory using `free()`. You pass `free()` the pointer that was returned from the call to `calloc()`;

```
array_p = calloc(size, sizeof(int));
...
// when done with the array
free(array_p);
```

Failing to free memory can result in what is called a *memory leak*. A long running program the continues to allocate more memory will eventually run out of memory. On the other hand, prematurely freeing memory can result in unpredictable results, when the program continues to try and use memory that has been deallocated using `free()`. A pointer variable that points to memory that has been deallocated is called a *dangling pointer*. Any attempt to use such a pointer is called a dangling pointer error. These types of problems with memory allocation are why languages like Java provide automated memory management with garbage collection.

Summary

- Static arrays in C have no direct counterpart in Java. Memory is allocated implicitly and the size of the array is part of the declaration. For example, int x[10], creates a static array of ten integers. This variable x will always refer to this array of ten integers.
- Array elements are indexed just like arrays in Java.
- Dynamic arrays are allocated using calloc(). The memory returned from calloc() is initialized to contain all zeros.
- Unlike Java, there is no bounds checking on array indicies.
- Array variables are actually just pointers to the first element of the array.
- There is no direct support for determining the length of an arbitrary array.
- Arrays are passed to functions by-reference. Only the address of the first element of the array is passed. This is essentially equivalent to how arrays are passed in Java.

Exercises

1 Write a program based on the *sieve of Eratosthenes* to compute the prime numbers between 2 and 100. A Java solution to this problem can be found in Section 5.8.1 of JBD.

2 Generalize the sieve algorithm in Section 5.8.1, on page 160, of JBD to go from 2 through *n*. In the general case, you need only strike out multiples that are less than or equal to the square root of *n*. You also need only strike out factors for values that have not been set to false. For example, striking out multiples of 2 strikes out `isPrime[4]`. Striking out multiples of 4 won't strike out any elements not already out. Try to improve your solution to the previous exercise with these ideas.

3 Modify your palindrome function from the previous exercise so that blanks and capitals are ignored in the matching process. Under these rules, the following are examples of palindromes.

```
"Huh"     "A man a plan a canal Panama"     "at a"
```

4 A real polynomial $p(x)$ of degree n or less is given by

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

with the coefficients $a_0, a_1, \ldots, a_n$ representing real numbers. If a_n ! = 0, the degree of $p(x)$ is n. Polynomials can be represented in code by an array such as

```
#define N  5 /* N is the max degree */
double  p[N+1];
```

Write a method

```
double evalPoly(double p[], int degree, double x) {
   ...
```

that returns the value of the polynomial `p` evaluated at `x`. Write two versions of the function. The first version should be a straightforward, naive approach. The second version should incorporate Horner's Rule. For fifth-degree polynomials, Horner's Rule is expressed as

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + x(a_4 + x(a_5)))))$$

How many additions and multiplications are used in each of your two versions of the `eval()` function?

5 Write a function that adds two polynomials.

```
// f = g + h;
int addPoly(double f[], int degreeF,
        double g[], int degreeG,
        double h[], int degreeH) {
...
```

If either `degreeG` or `degreeH` is greater than `degreeF` then the function does not modify `f` and the return value is -1, otherwise the sum is placed in `f` and the return value is the degree of the resulting polynomial.

6 Write a program that reads 10 characters into an array. Then have it print out the letters of the array sorted in alphabetic order.

Chapter 4

Strings

Strings in C are nothing more than arrays of characters. Like Java, C provides special syntax for creating string literals. Also, the standard C libraries include many useful functions for operating on strings.

4.1 Strings - arrays of characters

A properly formed string contains a null character as its last character, indicating the end of the string. Remember, arrays in C do not carry around their length so some mechanism had to be created for determining the length of a string. Consider the following example:

```
/* strings.c
 * Author - Charlie McDowell
 * Purpose - show strings as arrays of char
 */
#include <stdio.h>

int main(void) {
  char strOne[] =
    {'s','t','r','i','n','g',' ','o','n','e','\0'};
  char *char_p;
  int i;
```

```
    printf("%s\n", strOne);

    for(i = 0; strOne[i] != '\0'; i++)
      printf("%c,", strOne[i]);
    printf("\n");

    for(char_p = strOne; *char_p != '\0'; char_p++)
      printf("%c,", *char_p);
    printf("\n");

    return 0;
  }
```

The output of this program is

```
string one
s,t,r,i,n,g, ,o,n,e,
s,t,r,i,n,g, ,o,n,e,
```

Dissection of strings.c

-
  ```
  char strOne[] =
      {'s','t','r','i','n','g',' ','o','n','e','\0'};
  ```

The variable `strOne` is an array of characters initialized as shown. The char value `'\0'` is called the null character. It is used as the last character in all properly formed strings. Just as in Java, the backslash is an escape character (see JBD Section 2.9.3). In this example the escape sequence is used to specify the character whose binary representation is zero.

-
  ```
  char *char_p;
  int i;
  ```

These variables will be used later. In Section 3.4 we showed that we could use a pointer to an `int` to access an array of integers. In this program we will use a pointer to a `char` to access the elements of an array of characters.

-
  ```
  printf("%s\n", strOne);
  ```

As shown by this statement, `strOne`, is in fact a string in C. We can print it using `printf()` and the string conversion string `%s`.

-
  ```
  for(i = 0; strOne[i] != '\0'; i++)
    printf("%c,", strOne[i]);
  printf("\n");
  ```

We can also print the characters of the string one at a time, using normal array notation.

-
  ```
  for(char_p = strOne; *char_p != '\0'; char_p++)
    printf("%c,", *char_p);
  printf("\n");
  ```

As we did with the `int` array in Section 3.4, we can also print the characters of the string using pointer notation.

Because strings are used so often, the string literal notation can be used for initializing a string (array of characters). The following two statements are equivalent:

```
char *strOne = "string one";
char strOne[] =
    {'s','t','r','i','n','g',' ','o','n','e','\0'};
```

Notice that the notation, `"string one"`, implicitly includes the null character at the end.

4.2 Manipulating strings

The standard C libraries include a number of functions for manipulating strings. In this section we introduce a few of the most commonly used string functions. The functions we describe are summarized in the following table.

function	*brief description*
strcat(), strncat()	These functions are used to append the contents of one string onto the end of another string. This is called concatenation.
strchr()	This function is used to search for the first occurrence of a particular character in a string.
strcmp(), strncmp()	These functions are used to compare two strings.
strcpy(), strncpy()	These functions are used to make a copy of a string.
strlen()	This function is used to find the length of a string.
strstr()	This function is used to search for the first occurrence of one string as a substring in another string.

Notice that several of the functions come in two forms; the `str...` form and the `strn...` form. The latter form is always safer. These safer functions allow you to specify the maximum number of characters to be processed. This can be used to prevent errors in the event that one of the strings was inadvertently left without a terminating null character.

We describe the string functions in the above table using a dissection of the following example.

```
/* stringLib.c
 * Author - Charlie McDowell
 * Purpose - demo some standard string functions
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_S2 100

int main(void) {
  char *s1 = "String one";
  char *s2 = calloc(MAX_S2, sizeof(char));
  char *s3;

  printf("The length of \"%s\" is %d\n",
         s1, strlen(s1));

  strncpy(s2, "Testing", MAX_S2);
  strncat(s2, " 1 2 3", MAX_S2 - strlen(s2));
  printf("s2 contains:%s\n", s2);

  s3 = strchr(s1, 'g');
  printf("s3 is now %s\n", s3);

  s3 = strstr(s1, "tr");
  printf("s3 is now %s\n", s3);

  printf("comparing %s and %s gives %d\n",
         s1, s2, strncmp(s1, s2, MAX_S2));
  printf("comparing %s and %s gives %d\n",
         s2, s1, strncmp(s2, s1, MAX_S2));
  printf("comparing %s and String one gives %d\n",
         s1, strcmp(s1, "String one"));

  return 0;
}
```

The output for this program is:

```
The length of "String one" is 10
s2 contains:Testing 1 2 3
s3 is now g one
s3 is now tring one
comparing String one and Testing 1 2 3 gives -1
comparing Testing 1 2 3 and String one gives 1
comparing String one and String one gives 0
```

Dissection of stringLib.c

-
  ```
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #define MAX_S2 100
  ```

We need to include the header for the string library, *string.h*. Constants are created in C using the `#define` construct shown. All subsequent uses of the identifier `MAX_S2` will be equivalent to typing the integer `100`.

-
  ```
  char *s1 = "String one";
  char *s2 = calloc(MAX_S2, sizeof(char));
  char *s3;
  ```

We declare three string variables for use in the program. The first is initialized to a specific string literal. The second is pointing to an uninitialized portion of memory large enough to contain 100 characters, counting the null character at the end. The third string variable is uninitialized. Notice that there is no type with the name `string` in C. Instead we use `char *`, a pointer to `char`.

-
  ```
  printf("The length of \"%s\" is %d\n",
         s1, strlen(s1));
  ```

The expression `strlen(s1)` returns the length of `s1`, not including the terminating null character.

-
  ```
  strncpy(s2, "Testing", MAX_S2);
  ```

The expression `strncpy(str1, str2, n)` copies the characters in `str2` into `str1`, copying at most `n` characters. The first parameter must already be pointing to memory large enough to hold the copied characters. In this example, we previously assigned `s2` to point to a freshly allocated array of 100 characters. The function `strcpy()` is similar but it takes only two arguments, the two strings. There is no check to make sure that not too many characters are copied.

-
  ```
  strncat(s2, " 1 2 3", MAX_S2 - strlen(s2));
  printf("s2 contains:%s\n", s2);
  ```

The expression `strncat(str1, str2, n)`, copies at most `n` characters from `str2` onto the end of `str1`, and then adds a null character. In our example we limit the copy by the amount of space remaining in s2. The function `strcat()` is similar but takes only two parameters, the two strings, and no check is made for the number of characters copied. As shown in the output, the result of this concatenation is that `s2` now contains, `"Testing 1 2 3"`.

-
  ```
  s3 = strchr(s1, 'g');
  printf("s3 is now %s\n", s3);
  ```

The expression `strchr(str1, c)`, searches the string `str1` for the first occurrence of the character `c`. The result is a pointer to the matching character that was found. If the character is not found, the return value is NULL. It is important to note that in our example at this point `s3` is pointing into the middle of the string `s1`. If we change any of the characters after the `'g'` in `s1`, we will also be changing the string pointed to by `s3`. Remember, `s1` and and `s3` are not really strings but simply pointers to sequences of characters.

-
  ```
  s3 = strstr(s1, "tr");
  printf("s3 is now %s\n", s3);
  ```

The function `strstr()` is similar to `strchr()` except that instead of searching for a single character in a string, we are searching for a substring within a string. The substring is the second argument. The result is a pointer to the beginning of the matching substring within the original string. If the character is not found, the return value is NULL. As before, after the above statements, the memory for the string pointed to by `s1` and the memory for the string pointed to by `s3` overlap.

```
•   printf("comparing %s and %s gives %d\n",
           s1, s2, strncmp(s1, s2, MAX_S2));
    printf("comparing %s and %s gives %d\n",
           s2, s1, strncmp(s2, s1, MAX_S2));
    printf("comparing %s and String one gives %d\n",
           s1, strcmp(s1, "String one"));
```

Here we show that the function `strncmp()` can be used to compare two strings. The result is -1, 1, or 0, depending upon whether the first argument appears lexicographically before, after, or is equal to the second. The function `strcmp()` is the same as `strncmp()` except that the later has a third parameter that controls the maximum number of characters that will be compared.

Trouble spot: *Failing to include a null character at the end of a string can have very unpredictable results.*

When creating strings with something other than string literals, it is easy to leave off the terminating null character. This is a common source of program errors, often resulting in the program crashing. Sometimes, failing to terminate a string can result in much more insidious errors, that do not crash the program but instead corrupt some internal data, causing the program to behave in mysterious ways. Such errors are often very hard to detect. You should be very careful to always include the null character in any strings you generate. It is also a good idea to use the `strn...` form of the standard string functions whenever possible.

4.3 Reading strings with scanf()

Strings are read using `scanf()` by passing a pointer to the location where the first character of the string should be stored. The following example reads in a string using `scanf()`.

```
/* readingStrings.c
 * Author - Charlie McDowell
 * Purpose - reading strings with scanf()
 */
#include <stdio.h>
#include <strings.h>

int main(void) {
  char strOne[100];
  int status;

  strOne[0] = '\0'; // initialize to the empty string
  printf("Enter a sentence. End with a period.\n");

  status = scanf("%99s", strOne);
  while(status == 1 &&
        strOne[strlen(strOne)-1] != '.')
  {
    printf("%s\n", strOne);
    status = scanf("%99s", strOne);
  }
  printf("%s\n", strOne);

  return 0;
}
```

Dissection of readingStrings.c

- `char strOne[100];`

Here we create enough storage to hold a string of 99 characters plus the terminating null character.

- `int status;`

We will use `status` to check that the `scanf()` successfully read a string.

- `strOne[0] = '\0'; // initialize to the empty string`

Just to be safe, we initialize the string to the empty string by placing the null character in the first character of the string.

- `status = scanf("%99s", strOne);`

Although we have ignored it up until now, `scanf()` returns the number of items successfully scanned. The expected result in this example is one, the one string we are scanning. We will use this to control the termination of the loop. This will take effect if the end-of-file is encountered before the period at the end of the sentence. We also used the field width specification to make sure that at most 99 characters are read. Without this, a user could enter a really long string, overrunning the end of the `char` array `strOne`. This would cause unpredictable behavior of the program.

-
  ```
  while(status == 1 &&
        strOne[strlen(strOne)-1] != '.')
  ```

We will continue to read in strings until either the `scanf()` fails to return one, as recorded in `status`, or until a string ending in a period is read. The expression `strlen(strOne)-1` is the index of the last character in the string.

-
  ```
  {
    printf("%s\n", strOne);
    status = scanf("%99s", strOne);
  }
  ```

The body of the loop prints the previously read string on a line by itself and then reads in another string.

- `printf("%s\n", strOne);`

When the loop ends, we print the last string read. Normally this will be the one ending in a period.

The `scanf()` conversion specification `%[...]` can be used to scan an arbitrary set of characters. There are two cases, determined by the first character after the `[`. If the first character is the character ^, then the remaining characters specify which characters will not be scanned. When any one of the "not to be scanned" characters is encountered, the input ends for that argument. If the first character is not the character `^`, then any character not in the set ends the input.

For example, `%[()-0123456789]` might be used to read in a phone number written in the common form such as (888)444-1234, storing the result as a string. The normal string reading conversion specification `%s` reads until a white-space character is read. The `%s` is just shorthand for `%[^\t\n ]`, which specifies to read any characters that are not tab, newline or space (the white space characters).

Summary

- String is not a standard type in C. A string is simply an array of characters.
- A proper string is terminated by a null character, `'\0'`.
- C includes the same string literal syntax as in Java, e.g. `"this is a string"`.
- A `char` array used to hold a string must be long enough to store the terminating null character.
- A string variable can be declared as either a `char*`, or using an array declaration, e.g. `char s[10]`.
- The standard C library includes functions for manipulating strings.
- Strings can be read in from the standard input (typically the keyboard) using `scanf()` with the `%s` conversion character.

Exercises

1. Write a program to read in a string an determine if the string is a palindrome. A palindrome is a string that reads the same backward or forward. A simple example is madam. A Java solution is presented in Section 6.1 of JBD.
2. Modify your solution to the palindrom problem of the previous exercise to ignore space characters. For example, "a man a plan a canal panama" would be a palindrome by this definition.
3. Write a program to play a number guessing game. The player thinks of a number between 1 and 100 and the computer tries to guess. The program prints out how many guesses the computer used. The player must respond to each guess by typing *correct*, *too big*, or *too small*. The pseudocode for one possible solution is shown on page 187 of JBD.

4 Write a program that reads *n* strings into an array where *n* is read in. Then have it print out the strings of the array sorted in alphabetic order. Perform the sort with a method

```
void sort(char* names[]){...}
```

Remember, a string in C is simply a char*, so an array of strings is an array of char*. You can also think of the array of strings as a two-dimensional array of characters, with each row representing one string.

5 A simple encryption scheme is to interchange letters of the alphabet on a one-to-one basis. This scheme can be accomplished with a translation table for the 52 lowercase and uppercase letters. Write a program that uses such a scheme to encode text. Write another program to decode text that has been encoded. This isn't a serious encryption scheme. Do you know why? If you're interested, learn about a more secure encryption system and then program it. (See Section 10.5 in JBD.)

6 Write a program that reads words from the standard input using `scanf("%s",...)` and builds an array of all of the unique words that are found in the input. Print the list of words found.

Chapter 5

Structured data types

C uses the `struct` construct to specify user defined types. A `struct` is similar to a class in Java, but far more limited.

- A `struct` can have only data members, not function members.
- A `struct` cannot be derived from another `struct` using inheritance.
- A `struct` does not have the notions of private, and public. All fields are essentially public.
- Not having function members, a struct cannot have a constructor.

In Java, the name of the class is a new type and can be used in variable declarations. In C, a separate keyword, `typedef`, is used to create a new type.

5.1 Declaring a structured data type

The following example demonstrates the use of `typedef` and `struct` to create a new type.

```
/* structAssignment.c
 * Author - Charlie McDowell
 * Purpose - demo creating a new type
 */
#include <stdio.h>
#include <strings.h>

#define NAME_SIZE 80

typedef struct {
  char last[NAME_SIZE];
  char first[NAME_SIZE];
  double gpa;
  int startYear;
} student_t;

int main(void) {
  student_t studentOne, studentTwo;

  studentOne.gpa = 3.5;
  studentOne.startYear = 2000;
  strncpy(studentOne.first, "Jane", NAME_SIZE);
  strncpy(studentOne.last, "Programmer", NAME_SIZE);

  studentTwo = studentOne; //field by field assignment
  studentTwo.gpa = 3.45;
  strncpy(studentTwo.first, "John", NAME_SIZE);

  printf("%s, %s: gpa:%f, start year:%d\n",
         studentOne.last, studentOne.first,
         studentOne.gpa, studentOne.startYear);

  printf("%s, %s: gpa:%f, start year:%d\n",
         studentTwo.last, studentTwo.first,
         studentTwo.gpa, studentTwo.startYear);
  return 0;
}
```

The output of this program is

```
Programmer, Jane: gpa:3.500000, start year:2000
Programmer, John: gpa:3.450000, start year:2000
```

Dissection of structAssignment.c

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

  #define NAME_SIZE 80
  ```

The `#define` compiler directive is used to create named constants. Constants such as this make the program easier to modify and understand. The program is also less likely to have errors. A minor typographical error in a literal constant, such as 90 instead of 80 will go undetected by the compiler. However, a minor typographical error in a named constant, such as NAME_SSIZE instead of NAME_SIZE, will generally result in a compiler error.

-
  ```
  typedef struct {
    char last[NAME_SIZE];
    char first[NAME_SIZE];
    double gpa;
    int startYear;
  } student_t;
  ```

The above statement declares `student_t` to be a new `struct` type. We use the style convention of appending `_t` to an indentifier to indicate it is a type name. This particular structure type contains four fields. As you can see, fields can be simple types or arrays. Fields can even be other structures.

-
  ```
  int main(void) {
    student_t studentOne, studentTwo;
  ```

In this example, we declare two variables of type `student_t`. The variables `studentOne` and `studentTwo` are not pointers to `student_t` values, they actually contain a `student_t` value, in the same way that an `int` variable contains an `int` value. This is important because we do not need to create a `student_t` value with something like Java's `new`. Furthermore, we cannot change `studentOne` to later refer to a different `student_t` structure, although we can change the values of the fields.

-
  ```
  studentOne.gpa = 3.5;
  studentOne.startYear = 2000;
  ```

These two assignments use the same notation used in Java for assigning values to fields.

-
  ```
  strncpy(studentOne.first, "Jane", NAME_SIZE);
  strncpy(studentOne.last, "Programmer", NAME_SIZE);
  ```

It is not possible to use simple assignment with arrays. For example, the following is not legal syntax in C:

```
   studentOne.first = "Jane";
```

Instead, we use the standard C string copy function, `strncpy()`, to copy the characters from the literal strings for the names, into the character array fields of `studentOne`. Recall that `strncpy()` copies the characters in the second string, into the first string, stopping when either the null character is copied, or the number of characters specified by the third parameter have been copied, whichever comes first. After execution of the two `strncpy()` functions, all of the fields of `studentOne` have been initialized.

-
  ```
  studentTwo = studentOne; // field by field assignment
  ```

This assignment copies each field in the structure `studentOne`, into the corresponding fields of the structure `studentTwo`. This is different from the syntactically similar assignment in Java. In Java this would assign `studentTwo` and `studentOne` to refer to the same object (structure). After execution of this statement in Java, if you modified a field of `studentTwo`, you would also be modifying a field of `studentOne`, because they are the same object. This is very different from C as shown by the next two statements.

-
  ```
  studentTwo.gpa = 3.45;
  strncpy(studentTwo.first, "John", NAME_SIZE);
  ```

These two statements modify two of the fields of `studentTwo`. This has no affect on the fields in the structure `studentOne`.

•
```
printf("%s, %s: gpa:%f, start year:%d\n",
       studentOne.last, studentOne.first,
       studentOne.gpa, studentOne.startYear);

printf("%s, %s: gpa:%f, start year:%d\n",
       studentTwo.last, studentTwo.first,
       studentTwo.gpa, studentTwo.startYear);
```

Here we print out the values of the two structures, to show that indeed, there are two structures, not two references to one structure.

As the above example shows, assignment of structure values in C is treated just like assignment of values of built-in types. This is different from Java where assignment of class type variables has very different behavior from assignment of primitive type variables.

5.2 Passing structures as parameters

As indicated earlier, C, like Java, always uses pass-by-value for parameter passing. The difference between C and Java is that in C, a structure variable actually contains the structure data as opposed to containing a reference to the structure. This was shown in the previous example where we assigned one structure variable to another. The result is that structures are copied when passed as parameters. It is possible to pass a pointer to a structure, but that requires pointer notation (see the examle in Section 5.3). Consider the following example.

```
/* student.c
 * Author - Charlie McDowell
 * Purpose - passing structs to functions
 */
#include <stdio.h>
#include <strings.h>
#define NAME_SIZE 80

typedef struct {
  char last[NAME_SIZE];
  char first[NAME_SIZE];
  double gpa;
  int startYear;
} student_t;

void printRecord(student_t student);
student_t readRecord();

int main(void) {
  student_t student;

  student = readRecord();
  printRecord(student);
  return 0;
}

void printRecord(student_t student) {
  printf("%s, %s: gpa:%f, start year:%d\n",
         student.last, student.first,
         student.gpa, student.startYear);
}
```

```
student_t readRecord() {
  student_t record;

  record.gpa = 3.5;
  record.startYear = 2000;
  strncpy(record.first, "Jane",NAME_SIZE);
  strncpy(record.last, "Programmer",NAME_SIZE);
  return record;
}
```

Dissection of student.c

-
  ```
  void printRecord(student_t student);
  student_t readRecord();
  ```

This example uses the same `student_t` as the previous example. A structure type can be used just like a built-in type. Structure values can be passed as parameters and returned as results of functions. Just as with built-in types, structure values are passed by-value. The above two lines are function prototypes so that `main()` can be defined prior to the full definition of the functions `printRecord()` and `readRecord()`.

-
  ```
  int main(void) {
    student_t student;

    student = readRecord();
    printRecord(student);
    return 0;
  }
  ```

In the example above, we declare a single variable of type `student_t`. The function `readRecord()` will return a `student_t` value that is assigned (field by field) to student. We then pass the value `student`, to `printRecord()` for printing.

-
  ```
  void printRecord(student_t student) {
    printf("%s, %s: gpa:%f, start year:%d\n",
           student.last, student.first,
           student.gpa, student.startYear);
  }
  ```

The `printRecord()` function takes a single argument which is a `student_t` value. The fields of a structure value are accessed much like the fields of a Java object as shown earlier. The expression `student.gpa`, is an `int` value, and `student.startYear` is a `double` value. The expression `student.last[0]` would be the first character of the last name. The parameter student in `printRecord()` will contain a copy of the value of `student` in `main()`. It is not a pointer, or reference to the value as it would be in Java. This is an important difference that is discussed in Section 5.3.

-
  ```
  student_t readRecord() {
    student_t record;

    record.gpa = 3.5;
    record.startYear = 2000;
    strncpy(record.first, "Jane",NAME_SIZE);
    strncpy(record.last, "Programmer",NAME_SIZE);
    return record;
  }
  ```

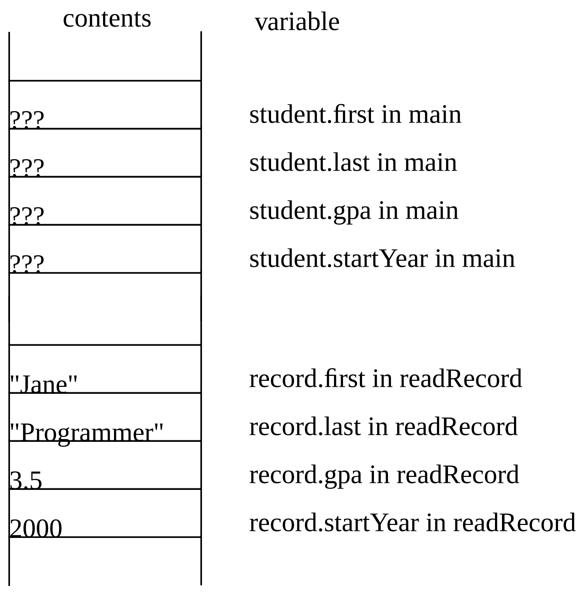
The function `readRecord()` demonstrates that a structure value can be returned as a value from a function. Furthermore, the assignment in `main()`

```
student = readRecord();
```

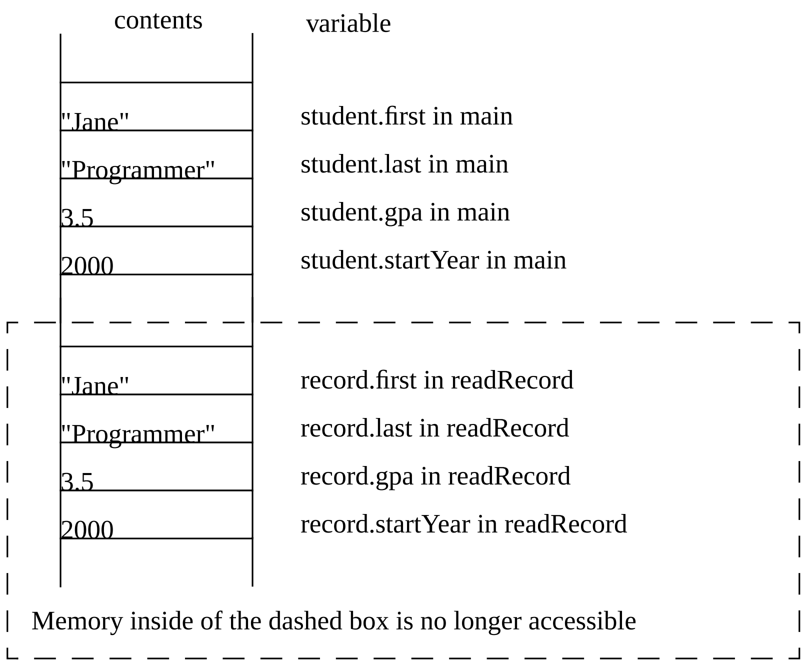
shows that structure values can be assigned just like built-in values such as `int`.

Although similar in syntax to returning an object value from a method in Java, operationally C is quite different. In the previous C example, the result of the return from `readRecord()` and the subsequent assignment in `main()` resulted in the values in the fields of the structure value `record` being copied into the fields of structure value `student` in `main()`. We are not returning a reference or pointer to the structure value `record` in `readRecord()`. The actual memory locations used to store the fields of `record` in `readRecord()` are different from the memory locations used to store the fields of `student` in `main()`. The following figure shows the layout of memory just before the return from `readRecord()` and again, just after the assignment to `student` in `main()`.

Before the return



After the return and the assignment to student


contents
variable
"Jane"
student.first in main
"Programmer"
student.last in main
3.5
student.gpa in main
2000
student.startYear in main
"Jane"
record.first in readRecord
"Programmer"
record.last in readRecord
3.5
record.gpa in readRecord
2000
record.startYear in readRecord
Memory inside of the dashed box is no longer accessible

In Java, when you pass an object variable, you are actually passing a reference or pointer, so changes to the fields of the object will be seen back at the caller. In C, changing the fields in a structure parameter of a function such as `printRecord()` will not have any affect on the original value. Consider the following example, which attempts to modify a structure value passed as a parameter, with the intent that the changes will be visible back at the point of the call. The program is erroneous, because the function `readRecord()` as defined below, has no effect in `main()`.

```
/* structByValue.c
 * Author - Charlie McDowell
 * Purpose - show that structs are copied
 */
#include <stdio.h>
#include <strings.h>

#define NAME_SIZE 80

typedef struct {
  char last[NAME_SIZE];
  char first[NAME_SIZE];
  double gpa;
  int startYear;
} student_t;

void printRecord(student_t student);
void readRecord(student_t student);

int main(void) {
  student_t student;

  // initialize all fields to null string or zero
  student.last[0] = '\0';
  student.first[0] = '\0';
  student.gpa = 0;
  student.startYear = 0;

  readRecord(student);
  printRecord(student);
  return 0;
}

void printRecord(student_t student) {
  printf("%s, %s: gpa:%f, start year:%d\n",
         student.last, student.first,
         student.gpa, student.startYear);
}
```

```
void readRecord(student_t record) {
  record.gpa = 3.5;
  record.startYear = 2000;
  strncpy(record.first, "Jane",NAME_SIZE);
  strncpy(record.last, "Programmer",NAME_SIZE);
}
```

The output of this program is

```
, : gpa:0.000000, start year:0
```

Dissection of structByValue

- `void readRecord(student_t student);`

This program is largely the same as *student.c*, discussed earlier. The main change is that the function `readRecord()` takes a `student_t` parameter instead of returning a `student_t` value as a result.

-
  ```
  int main(void) {
    student_t student;

    // initialize all fields to null string or zero
    student.last[0] = '\0';
    student.first[0] = '\0';
    student.gpa = 0;
    student.startYear = 0;
  ```

Local variables, such as `student` above, are not initialized to any particular value by default. To make this program have predictable output, we initialize all of the fields of the structure variable `student`.

- `readRecord(student);`

In the previous example we had `readRecord()` return a value that was assigned to `student`. In this example we pass `student` as a parameter. If this were Java, the method `readRecord()` could modify the fields of the object referred to by `student`, so that upon the return from `readRecord()`, the fields would have been changed. Using the notation shown here, C copies the entire `student` structure value, into the formal parameter in the function `readRecord()`. The result will be that any changes made inside of `readRecord()` will have no effect.

-
  ```
  void readRecord(student_t record) {
    record.gpa = 3.5;
    record.startYear = 2000;
    strncpy(record.first, "Jane",NAME_SIZE);
    strncpy(record.last, "Programmer",NAME_SIZE);
  }
  ```

The above function is erroneous. It assigns values to all of the fields of the formal parameter `record`, but this does not change any of the fields in the variable `student` in `main()`. It is important to note that even the array fields, `first` and `last`, are copied. This is different from the handling of an array passed as a parameter by itself. When an array is passed directly as a parameter, the array is not copied, only the address of the first element of the array is passed. When an array is part of a structure, the entire array is copied. The following figure shows the layout of memory just before the return from `readRecord()`.

contents	variable
""	student.first in main
""	student.last in main
0	student.gpa in main
0	student.startYear in main
"Jane"	record.first in readRecord
"Programmer"	record.last in readRecord
3.5	record.gpa in readRecord
2000	record.startYear in readRecord

5.3 Pointers to structures

In our examples of structures above, we were dealing with structure variables which contained structure values. Although similar in syntax to reference variables in Java, these structure variables were not pointers or references. As with the built-in types, it is possible to declare and use pointers to structures. The behavior of pointers to structures is very similar to the behavior of references in Java. Unfortunately the Java syntax for references is most similar to the C syntax for structures that are *not* pointers. The pointer syntax introduced earlier for use with built-in types, can be used with structure values as shown in the following example.

```
/* struct.c
 * Author - Charlie McDowell
 * Purpose - demo creating a new type
 */
#include <stdio.h>
#include <strings.h>

#define NAME_SIZE 80

typedef struct {
  char last[NAME_SIZE];
  char first[NAME_SIZE];
  double gpa;
  int startYear;
} student_t;

void printRecord(student_t student);
void readRecord(student_t *record_p);

int main(void) {
  student_t student;

  readRecord(&student);
  printRecord(student);
  return 0;
}
```

```
void printRecord(student_t student) {
  printf("%s, %s: gpa:%f, start year:%d\n",
         student.last, student.first, student.gpa,
         student.startYear);
}

void readRecord(student_t *record_p) {
  (*record_p).gpa = 3.5;
  (*record_p).startYear = 2000;
  strncpy((*record_p).first, "Jane",NAME_SIZE);
  strncpy((*record_p).last, "Programmer",NAME_SIZE);
}
```

This example is very similar to the erroneous example from the previous section. In this case the program is correct. In particular, the function `readRecord()`, because it is passed a pointer to the structure value in `main()`, can modify the fields of the structure variable `student` in `main()`. Notice that the call to `readRecord()` passes the *address of* `student`, not the *value* student. Then in the function `readRecord()`, the parameter is declared to be a pointer to a `student_t`, not a simple `student_t` value. Finally, we use the indirection operator to get to the actual `student_t` value and then access the fields. The parenthesis in statements such as

```
(*record_p).gpa = 3.5;
```

are required. Both * and . are operators in this statement. They have precedence, and the precedence for . is higher than the precedence for *. Therefore, without the parenthesis

```
*record_p.gpa = 3.5;
```

would first try to access the field `gpa` of the pointer `record_p`. But `record_p` is not a structure value that has fields, it is a pointer. Thus the expression `record_p.gpa` is a syntax error. With the parenthesis, the expression `(*record_p)` is a `student_t` value that does have a field `gpa` so that `(*record_p).gpa` is a legal expression.

Because pointers to structures are frequently used in C, probably more often that actual structure variables, C provides a special syntax for accessing a field of a structure using a pointer to the structure value. The following is an equivalent way of writing `readRecord()` using the preferred syntax for accessing fields, given a pointer to a structure value.

```
void readRecord(student_t *record_p) {
  record_p->gpa = 3.5;
  record_p->startYear = 2000;
  strncpy(record_p->first, "Jane",NAME_SIZE);
  strncpy(record_p->last, "Programmer",NAME_SIZE);
}
```

The operator `->` is called the structure pointer operator. As you can see, the expression `record_p->gpa` is equivalent to the expression `(*record_p).gpa`. A trick for remembering when to use . and when to use `->` for accessing fields is that you use the `->` when you have a pointer, and `->` looks somewhat like a pointer or arrow.

5.4 Dynamic allocation of structure values

As with arrays, structure values can be dynamically allocated. The following example dynamically allocates a `student_t` value and assigns `studentOne_p` to point to that value. The example also shows that pointer assignment in C is analogous to assignment with reference variables in Java. Compare this program and its output with the program *structAssignment.c* from Section 5.1. The two `student_t` variables in the earlier example have been changed to pointers. Most uses of the variables have been adjusted to accommodate the fact that they are now pointers. However, the assignment `studentTwo_p` = `studentOne_p` below was left as a simple assignment. The result is that both `studentTwo_p` and `studentOne_p` will be pointing to the same structure value.

```
/* structPointer2.c
 * Author - Charlie McDowell
 * Purpose - structure pointer assignment
 */
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

#define NAME_SIZE 80
```

```
typedef struct {
  char last[NAME_SIZE];
  char first[NAME_SIZE];
  double gpa;
  int startYear;
} student_t;

int main(void) {
  student_t *studentOne_p, *studentTwo_p;

  studentOne_p =
      (student_t *)malloc(sizeof(student_t));
  studentOne_p->gpa = 3.5;
  studentOne_p->startYear = 2000;
  strncpy(studentOne_p->first, "Jane", NAME_SIZE);
  strncpy(studentOne_p->last, "Programmer",NAME_SIZE);

  studentTwo_p = studentOne_p; // pointer assignment

  studentTwo_p->gpa = 3.45;
  strncpy(studentTwo_p->first, "John", NAME_SIZE);

  printf("%s, %s: gpa:%f, start year:%d\n",
         studentOne_p->last, studentOne_p->first,
         studentOne_p->gpa, studentOne_p->startYear);

  printf("%s, %s: gpa:%f, start year:%d\n",
         studentTwo_p->last, studentTwo_p->first,
         studentTwo_p->gpa, studentTwo_p->startYear);

  return 0;
}
```

The output of this program is

```
Programmer, John: gpa:3.450000, start year:2000
Programmer, John: gpa:3.450000, start year:2000
```

Dissection of structPointer2.c

- `student_t *studentOne_p, *studentTwo_p;`

This program creates two pointers to student_t values. No student_t values have yet been created. Notice that the * is associated with the variable name, not with the type. Consider the declaration

```
student_t* studentOne_p, studentTwo_p;
```

The above declaration is syntactically legal but it declares studentOne_p to be a pointer to a student_t and it declares studentTwo_p to be a student_t, not a pointer.

-
  ```
  studentOne_p =
        (student_t *)malloc(sizeof(student_t));
  ```

This statement creates on student_t value, the only one created in this program.

-
  ```
  studentOne_p->gpa = 3.5;
  studentOne_p->startYear = 2000;
  strncpy(studentOne_p->first, "Jane", NAME_SIZE);
  strncpy(studentOne_p->last, "Programmer",NAME_SIZE);
  ```

Here we fill in the fields of the student_t value pointed to by studentOne_p.

- `studentTwo_p = studentOne_p; // pointer assignment`

This assignment is like a Java assignment of two reference variables. Specifically, after execution of this statement, studentTwo_p will be pointing to the exact same memory location (student_t value) that studentOne_p is pointing to.

-
  ```
  studentTwo_p->gpa = 3.45;
  strncpy(studentTwo_p->first, "John", NAME_SIZE);
  ```

Using the variable studentTwo_p, we change two of the fields in the one student_t value created in this program. This is the same student_t value that studentOne_p is pointing to, as shown by the print statements that follow.

```
•   printf("%s, %s: gpa:%f, start year:%d\n",
             studentOne_p->last, studentOne_p->first,
             studentOne_p->gpa, studentOne_p->startYear);

    printf("%s, %s: gpa:%f, start year:%d\n",
             studentTwo_p->last, studentTwo_p->first,
             studentTwo_p->gpa, studentTwo_p->startYear);
```

Here we print out the values pointed to by both student_t pointers. As shown above, the output of both statements is the same.

There is only one `student_t` value in the above program. Try replacing the assignment

```
   studentTwo_p = studentOne_p;
```

with the following:

```
   studentTwo_p = (student_t *)malloc(sizeof(student_t));
   *studentTwo_p = *studentOne_p;
```

Now you will get the same output as that produced by the earlier example in Section 5.1. Try leaving out the first of the two statements above? What happens?

We used the standard library function `calloc()` to allocate arrays in Section 3.4. We could allocate a `student_t` structure using the call `calloc(1, sizeof(student_t))`. However, it is more common to use the standard function `malloc()` which takes only a single parameter, the size of the structure being allocated. These allocation functions return a generic pointer with the type `void*`. To avoid error messages, these generic pointers must be cast into the appropriate type. This is just like casting in Java except that there are no runtime checks to make sure that the cast is valid. In the example above we are casting the result in to a pointer to a `student_t`.

Summary

- A `struct` in C is like a class with only data members, no methods.

- Unlike Java, a `struct` value can be stored directly in a variable. C can also declare pointers to `struct` values. These pointers are like Java references.

- To access a field of a `struct` value, if the variable contains the `struct` value, then you you use the field selection operator, dot, e.g. `studentOne.gpa`.

- To access a field of a pointer to a `struct` value, use the indirect field selection operator, `->`, e.g. `record_p->gpa`.

- Although a pointer to a `struct` value *behaves* most like a reference to a Java object, *syntactically,* field access in C is most similar to Java when using actual `struct` values, not pointers.

- Passing a `struct` value in C is not like passing a Java object. To get the same behavior as in Java, in C we must pass a pointer to a `struct` value.

Exercises

1 Create a structured type to represent an amount of change. An amount of change is determined by a specified number of dollars, quarters, dimes, nickels and pennies. Write a function makeChange() that takes two double parameters representing an amount of money paid and the amount of money owed, and returns a value of the newly defined change type to represent the change that should be returned. Include this function in a program to test it by prompting the user for the amounts owed and paid, and then printing the change. A Java solution to this problem is presented in Section 6.8 of JBD.

2 Create a `struct` to represent complex numbers. Complex numbers have two parts: a real part and an imaginary part. So a complex number value could be (1, 1.5i), where 1 is the real part and 1.5i is the imaginary part. Create functions to add, subtract, and multiply two complex numbers. The rules for these operations are

$$(a, bi) + (c, di) = (a + c, (b + d)i);$$
$$(a, bi) - (c, di) = (a - c, (b - d)i);$$
$$(a, bi) \times (c, di) = (ac - bd, (ad + bc)i)$$

3 Create a struct to represent a person with fields for first and last name, address, social security number, and telephone number. Write a program that reads in the data for a list of people and stores the data in an array of your person struct values. Then sort the array on the last name of the person. Finally, printed the sorted list of entries. If you define the new type with typedef as in

```
typedef struct {...} person_t;
```

Then you can create an array of 100 person records with

```
person_t data[100];
```

4 Modify your solution to the previous problem to represent the address info as a `struct` value also, with fields for street address, city, state, and zipcode. Fields in `struct`s can themselves be `struct`s.

Chapter 6

File I/O and multiple source files

There are many libraries of functions that can be called from C programs. Some of them are standard and found on essentially all platforms that support C, while others are specific to a particular platform. For example, there is standard C library support for file I/O, but no standard C library for supporting the development of graphical user interfaces.We begin this chapter with a brief discussion of the standard C support for file I/O.

We conclude this chapter and this primer with an extended example that also introduces the use of header files and multiple source files.

6.1 File I/O and Command Line Arguments

In this section we briefly introduce some standard C functions for performing input and output using files. The example also introduces the use of command line arguments, which is very similar to how they are handled in Java. The following program opens a text file, and copies the entire contents into another text file. The names of the two files will be specified on the command line when the program is executed.

```
/* fileCopy.c
 * Author - Charlie McDowell
 * Purpose - Demonstratie text file I/O by copying one
 *    file to another. The program also uses command
 *    line arguments.
 */
#include <stdio.h>

void copy(FILE* in, FILE* out);

int main(int argc, char* argv[]) {
  FILE* in_file;
  FILE* out_file;

  if(argc != 3) {
    printf("Usage: fileCopy inputFile outputFile\n");
    exit(1);
  }

  in_file = fopen(argv[1],"r");
  if(in_file == NULL) {
    fprintf(stderr,
            "ERROR: couldn't open %s for reading\n",
            argv[1]);
    exit(1);
  }

  out_file = fopen(argv[2],"w");
  if(out_file == NULL) {
    fprintf(stderr,
            "ERROR: couldn't open %s for writing\n",
            argv[2]);
    exit(1);
  }
```

```
    copy(in_file, out_file);

    return 0;
  }

  #define BUF_SIZE 200

  void copy(FILE* in, FILE* out) {
    char buffer[BUF_SIZE];

    while(fgets(buffer, BUF_SIZE, in) != NULL) {
      fprintf(out,"%s",buffer);
    }
  }
```

If the above program was compiled into an executable with the name fileCopy, then the following command could be used to copy the contents of the file inputFile.txt into the file outputFile.txt as shown below.

```
os-prompt>gcc fileCopy.c -o fileCopy
os-prompt>fileCopy inputFile.txt outputFile.txt
os-prompt>
```

Dissection of fileCopy.c

- `#include <stdio.h>`

The function prototypes for the I/O functions used in this program are in the standard header file, stdio.h.

- `void copy(FILE* in, FILE* out);`

To make the program more readable, we have separated the actual copying into a separate function. The function is defined after main(), thus the need for a function prototype. The type FILE, is a standard type from the header file stdio.h. This is what we will use to refer to a file once it has been opened for reading or writing.

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

In all previous examples, `main()` was not passed any arguments. As shown in this example, the `main()` function of a C program can be passed any array of strings. Recall that `char*` is the closest thing to the type string in C, so `argv` above is an array of strings. Because C arrays do not include a length, `main()` is also passed the length of the array, in the argument labeled `argc`. The names `argc` and `argv` are simple identifiers, and can in principle be any legal identifier, however, these are the names normally used by convention. The names come from **arg**ument **c**ount and **arg**ument **v**ector.

-
  ```
  FILE* in_file;
  FILE* out_file;
  ```

Here we declare two `FILE*` variables. The standard file manipulation routines deal with this type, as shown below.

-
  ```
  if(argc != 3) {
    printf("Usage: fileCopy inputFile outputFile\n");
    exit(1);
  }
  ```

Unlike Java, the command line arguments passed to a C program include the name of the program as the first entry in the array. Therefore this program is expecting the array to be length 3, the program name, the input file and the output file. The function `exit()` is a standard function that aborts the program.

-
  ```
  in_file = fopen(argv[1],"r");
  if(in_file == NULL) {
    fprintf(stderr,
            "ERROR: couldn't open %s for reading\n",
            argv[1]);
    exit(1);
  }

  out_file = fopen(argv[2],"w");
  if(out_file == NULL) {
    fprintf(stderr,
            "ERROR: couldn't open %s for writing\n",
            argv[2]);
    exit(1);
  }
  ```

The function `fopen()` is a standard C function for opening a file. The first argument is the name of the file. The second argument is a string indicating how the file should be opened. There are a number of variations. In this example we use only `"r"` for reading the input file, and `"w"` for writing the output file. The function `fopen()` will return a pointer to the `FILE` structure used to manipulate the file, if the file is opened successfully. If the open fails, the function returns `NULL`. To make our program reasonably user friendly, we test the return value and print an appropriate message if the open fails. The function fprintf() is the same as printf() except that it takes an additional parameter that is a pointer of an open `FILE` structure. This allows us to print to any file. The variable `stderr`, is a predefined value that refers to the standard error output stream. Although not shown, there is a function `fscanf()` that is the same as `scanf()` except that, like `fprintf()`, it takes a file parameter.

- `copy(in_file, out_file);`

Here we simply pass the pointers to the two successfully opened files to the method copy(), which does the actual copying.

- `#define BUF_SIZE 200`

We use a constant to document the size of the buffer we will use for reading text from the input file.

•
```
void copy(FILE* in, FILE* out) {
  char buffer[BUF_SIZE];

  while(fgets(buffer, BUF_SIZE, in) != NULL) {
    fprintf(out,"%s",buffer);
  }
}
```

The function `fgets()` is a standard C library routine that reads strings from a text file. Each call to `fgets()` transfers characters from the input file into the char array indicated by the first parameter (`buffer` in our example). The function continues to copy characters until either `BUF_SIZE-1` characters (the second parameter minus 1) have been transfered, or a newline has been transfered, or the end-of-file is encountered. In every case, the string is terminated with a null character. Notice that by stopping at `BUF_SIZE-1`, there will always be room for the null character to terminate the string. If the call transfers any characters, the `fgets()` returns its first parameter, otherwise it returns `NULL`. As you can see, the end-of-file is signalled by a return value of `NULL`.

The standard C libraries include other functions for performing file I/O. Refer to the documentation with your system or a standard book on C for more details.

6.2 Header Files

C, like Java, allows programs to be composed from multiple source files. As we saw in Section 2.1, it is necessary to provide a function prototype for a function, if we call the function before defining the function. This same mechanism is used when calling a function that is defined in a different file. It is for this reason that we have included files like *stdio.h* using the `#include` directive. These files, called header files, include function prototypes, type definitions (using `typedef`) and constants (using `#define`). By placing this information in a header file, we can avoid having to retype the information in all of the files that may need it. All we need to type is the single line:

```
#include <someHeader.h>
```

Just as in Java, a C program must contain a function `main()`. And just as in Java, you can combine several files together to make a program. Because all functions in C share the same name space, that is, they are not grouped into classes, there can only be one file in your program that contains a function `main()`. Recall that you can have several Java classes that each contain a `main()` method. When you start the Java program, you indicate which class you want to use as the main class. In C, the compiler will combine all of the source files into a single output file which must contain only one `main()` function.

Both header files (files ending in *.h*) and normal C source files (files ending in *.c*) are considered part of the C source code. As mentioned above, the header files will generally contain type definitions, constant definitions, and function prototypes. The other C source files will contain the function definitions. The following program is composed of three source files, two regular source files and one header file. One source file will contain the `main()` function, and the other will contain methods to implement a simple stack (see JBD Section 12.2).

```
/* fileMain.c
 * Author - Charlie McDowell
 * Purpose - Demonstratie separate compilation and
 *    header files using a simple stack implementation.
 */
#include <stdio.h>
#include "stack.h"

int main(void) {
  stack_t* myStack = newStack();

  push(myStack, 123);
  push(myStack, 99);
  push(myStack, 4444);
  while(!empty(myStack)) {
    int value;
    value = pop(myStack);
    printf("popped: %d\n", value);
  }
}
```

```
/* stack.h
 * Author - Charlie McDowell
 * Purpose - A header file for a stack containing
 *    type declarations and function prototypes.
 */
#include <stdio.h>

typedef struct stackElem {
  int value;
  struct stackElem* next_p;
} stackElem_t;

typedef struct {
  stackElem_t* top_p;
} stack_t;

int pop(stack_t* stack_p);
void push(stack_t* stack_p, int value);
int top(stack_t* stack_p);
stack_t* newStack();
int empty(stack_t* stack_p);
```

```
/* stack.c
 * Author - Charlie McDowell
 * Purpose - a simple integer stack implementation.
 *    This implementation uses a linked list.
 */
#include "stack.h"

stack_t* newStack() {
  stack_t* result = (stack_t *)malloc(sizeof(stack_t));
  result->top_p = NULL;
  return result;
}

int pop(stack_t* stack_p) {
  if(stack_p == NULL || stack_p->top_p == NULL) {
    fprintf(stderr,
            "ERROR: tried to pop an empty stack\n");
    exit(1); // abort the program
  }
  else {
    int result;
    stackElem_t* result_elem;
    result_elem = stack_p->top_p;
    result = result_elem->value;
    stack_p->top_p = result_elem->next_p;
    free(result_elem);
    return result;
  }
}

void push(stack_t* stack_p, int value) {
  stackElem_t* temp =
      (stackElem_t *)malloc(sizeof(stackElem_t));
  temp->value = value;
  temp->next_p = stack_p->top_p;
  stack_p->top_p = temp;
}

int top(stack_t* stack_p) {
  if(stack_p == NULL || stack_p->top_p == NULL) {
    fprintf(stderr,
      "ERROR: tried to examine top of empty stack\n");
    exit(1); // abort the program
  }
  else
    return stack_p->top_p->value;
}
```

```
int empty(stack_t* stack_p) {
  return (stack_p->top_p == NULL);
}
```

The easiest way to compile a C program that consists of multiple files is to compile all of the files with a single command. The program above can be compiled and executed as shown below.

```
os-prompt>gcc stackMain.c stack.c -o stackTest
os-prompt>stackTest
popped: 4444
popped: 99
popped: 123
os-prompt>
```

For large C programs, consisting of many source files, there are mechanisms that do not require every source file to be recompiled each time. The details of these mechanisms vary from system to system.

Dissection of stackMain.c

-
  ```
  #include <stdio.h>
  #include "stack.h"
  ```

When including header files from the standard C library, the name of the file is enclosed in <angle brackets>. When including header files that you create yourself, and that reside in the same directory as your other source files, the file name is enclosed in quotes. The choice of enclosing symbols affects where the C compiler expects to find the header file. The function prototype for `printf()` is in *stdio.h*, and the prototypes and typedefs for our stack implementation are in *stack.h*, dissected later.

-
  ```
  int main(void) {
    stack_t* myStack = newStack();
  ```

The type `stack_t` is defined in the header file *stack.h*. The method `newStack()` is implemented in the file *stack.c* and a prototype for the function is in *stack.h*. At this point it is sufficient to know that `newStack()` returns a pointer to a `stack_t` value.

-
  ```
  push(myStack, 123);
  push(myStack, 99);
  push(myStack, 4444);
  ```

To demonstrate our use of our stack, we push three integer values onto the stack. The function push() is defined in stack.c, dissected later. Recall that in Java the first call would probably have been written as

```
myStack.push(123);
```

In C, functions aren't part of the type and there is no implicit first parameter (see JBD Section 6.3). Therefore we explicitly pass the stack as the first parameter to the function `push()`.

-
  ```
  while(!empty(myStack)) {
    int value;
    value = pop(myStack);
    printf("popped: %d\n", value);
  }
  ```

The functions `empty()` and `pop()` are defined in *stack.c*, and have prototypes in *stack.h*.

Dissection of stack.h

-
  ```
  typedef struct stackElem {
    int value;
    struct stackElem* next_p;
  } stackElem_t;
  ```

Here we declare a type that will be used to store each of the values pushed onto the stack, in a simple linked list. Each element will point to the next element in the stack. This example presents a complication that arises when declaring recursive data structures such as this in C. In Java, the name of a class is immediately available to the implementation of the class as a type. In C, the `typedef` notation shown here for creating a new type, does not make the new type available until the `typedef` has been completely processed. The solution is as shown here. By including an identifier such as `stackElem` after the keyword `struct`, we can immediately use `struct stack-`

`Elem` as the type. In fact, the `typedef` as used here is a way of defining the name `stackElem_t` to be a shorthand for `struct stackElem`. The shorthand version can't be used within the `struct` definition. The identifier after the `struct` is optional if we only plan to use the shorthand version.

-
  ```
  typedef struct {
    stackElem_t* top_p;
  } stack_t;
  ```

This is the main stack type. It is simply a pointer to the top element in the stack. An empty stack will be represented by having `top_p` be `NULL`.

-
  ```
  int pop(stack_t* stack_p);
  void push(stack_t* stack_p, int value);
  int top(stack_t* stack_p);
  stack_t* newStack();
  int empty(stack_t* stack_p);
  ```

These are the function prototypes for the functions that implement the stack. The functions are defined in *stack.c*. By placing the prototypes in this header file, functions in other files, such as *stackMain.c*, can call the functions defined in *stack.c* by simply including *stack.h* using the `#include` directive in the other source files.

Dissection of stack.c

-
  ```
  #include "stack.h"
  ```

The file *stack.c* provides the implementation for the function prototypes in *stack.h*. In addition, *stack.c* use the types defined in *stack.h* to implement the stack operations.

-
  ```
  stack_t* newStack() {
    stack_t* result =
        (stack_t *)malloc(sizeof(stack_t));
    result->top_p = NULL;
    return result;
  }
  ```

For this stack design, we decided that users of the stack should always manipulate pointers to stack values. They obtain a pointer to a properly initialized stack value by calling this method. This method simulates the function of a Java constructor.

-
  ```
  int pop(stack_t* stack_p) {
    if(stack_p == NULL || stack_p->top_p == NULL) {
      fprintf(stderr,
              "ERROR: tried to pop an empty stack\n");
      exit(1); // abort the program
    }
    else {
      int result;
      stackElem_t* result_elem;
      result_elem = stack_p->top_p;
      result = result_elem->value;
      stack_p->top_p = result_elem->next_p;
      free(result_elem);
      return result;
    }
  }
  ```

C does not have direct support for exception handling. In a Java program we might throw an exception if an attempt was made to pop an empty stack or a stack that had not been properly initialized. In this example we simply abort the program printing an error message. If the stack is not empty we remove the top element, advance `top_p` to the next element in the stack and free the memory occupied by the popped `stackElem_t` value. Without the `free()`, the memory for the popped `stackElem_t` values would constitute a memory leak. A program using this stack implementation would slowly consume more and more memory until it finally ran out of memory. There are weaknesses with this implementation. There is nothing in the language that prevents other parts of a program from obtaining pointers to `stackElem_t` values from the stack and using them. However, if a program did this, then unpredictable behavior would result when we free up the memory as we do here. There are ways to implement a stack that provide increased data hiding, but those techniques are beyond the scope of this brief primer on C.

-
  ```
  void push(stack_t* stack_p, int value) {
    stackElem_t* temp =
        (stackElem_t *)malloc(sizeof(stackElem_t));
    temp->value = value;
    temp->next_p = stack_p->top_p;
    stack_p->top_p = temp;
  }
  ```

This function assumes the `stack_t` value has been properly initialized (i.e. the `top_p` field has a meaningful value). Because C `struct`s do not have constructors, there is no guarantee that the value has been initialized. If the stack value was created using the method `newStack()` then the `top_p` field will be valid. The method allocates memory for a new `stackElem_t` value and then links that value onto the top of the stack.

-
  ```
  int top(stack_t* stack_p) {
    if(stack_p == NULL || stack_p->top_p == NULL) {
      fprintf(stderr,
       "ERROR: tried to examine top of empty stack\n");
      exit(1); // abort the program
    }
    else
      return stack_p->top_p->value;
  }
  ```

This method returns the same value as pop(), but it doesn't modify the stack.

-
  ```
  int empty(stack_t* stack_p) {
    return (stack_p->top_p == NULL);
  }
  ```

This method allows the user of a stack to check and see if there are any elements remaining in the stack. Such a method is essential given this design where `pop()` aborts the program if you try to pop an element from an empty stack.

Summary

- C provides access to command line arguments by passing an array of strings and an argument count as parameters to `main()`.

- A file can be opened for reading or writing using the standard C function `fopen()`.
- The function `fprintf()` is like `printf()` but it can be used to print to a specific file.
- The function `fscanf()` is like `scanf()` but it can be used to read from a specific file.
- The functions of a C program can be grouped into multiple source files.
- Header files (ending in *.h* by convention) are used to hold function prototypes, constant declarations, and type declarations. These header files can then be included in any source file needing those declarations.
- Unlike a Java class name, the type name defined by a `typedef` cannot be used as a type within the `typedef`. For `struct` types, the new type can be referred to using the notation `struct name`, even within the type definition. For example, in the declaration

  ```
  typedef struct someType {...} someType_t;
  ```

  the name `someType_t` cannot be used to define a field in this `struct` type, however, the type `struct someType` can be used. The identifier `someType` is not required if this `struct` type is not recursive.

Exercises

1 Write a C program that prints out any command line arguments, one per line. A Java solution to this problem is presented in Section 5.11.1 of JBD.

2 Write a C program that encrypts a text file using a one-time pad. The names of the clear text file and the resulting encrypted text file should be taken from the command line. A Java solution to this problem is presented in Section 10.5 of JBD.

3 Write a program to decode the results of encoding a file using the program from the previous exercise. If you use srand() to seed the random sequence in the encoding program, you will need to use the same seed in the decoding program.

4 Modify *stack.c* and *stack.h* from Section 6.2 to implement a stack using an array instead of a linked list.

5 Implement a linked list with the same capabilities as the linked list implementation in Section 12.3 of JBD. You should create a header file to contain any declarations that are needed by a program that uses your linked list. Provide a sample program that tests your linked list implementation.