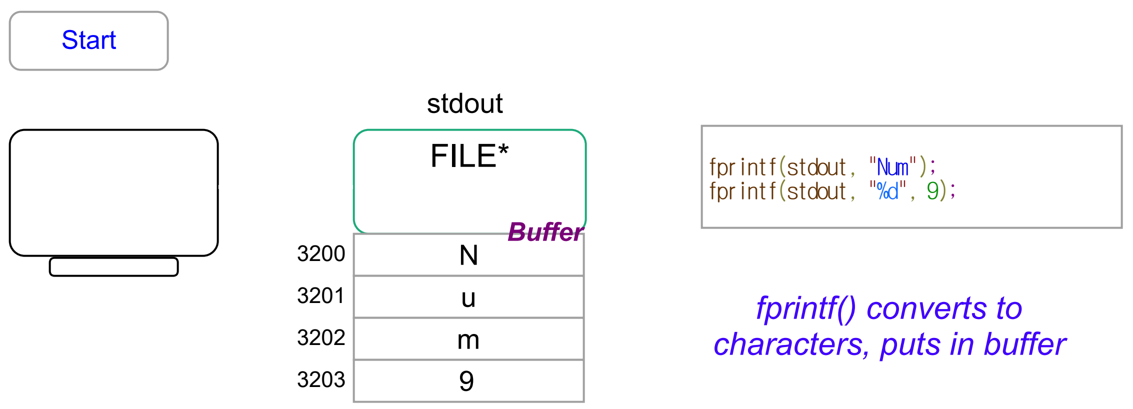
9.1 The stdout file pointer

Programs often need to output data to a screen, file, or elsewhere. A ***FILE****, called a "file pointer," is a pointer to a FILE structure that allows programs to read and write to files. FILE* is available via `#include <stdio.h>`.Pointers

The FILE structure maintains the information needed to access files. The FILE structure typically maintains an output buffer that temporarily stores characters until the system copies those characters to disk or screen.

stdout is a predefined FILE* that is pre-associated with a system's standard output, usually a computer screen. The following animation illustrates.

PARTICIPATION ACTIVITY 9.1.1: Writing to stdout using fprintf().



```
fprintf(stdout, "Num");
fprintf(stdout, "%d", 9);
```

The fprintf() function, or "file print", writes a sequence of characters to a file. The first argument to fprintf() is the FILE* to the file being written. The remaining arguments for fprintf() work the same way as the arguments for printf().

The second argument for the fprintf() function is the ***format string*** that specifies the format of the text that will be printed along with any number of ***format specifiers*** for printing values stored in variables. The arguments following the format string are the expressions to be printed for each of the format specifiers within the format string.

Basic use of printf() and format specifiers was covered in an earlier section, and can be used similarly for fprintf().

PARTICIPATION ACTIVITY 9.1.2: fprintf() and stdout.

1) Write a statement using fprintf() that prints "Enter your age: " to stdout.

Check Show answer

2) Write a statement using fprintf() to print an int variable named numSeats to stdout.

Check Show answer

3) Write a statement using fprintf() to print two float variables named x and y separated by a single comma to stdout.

Check Show answer

4) Will the following two statements both print the same result to the standard output (answer Yes or No)?

```
fprintf(stdout, "nums:");
printf("nums:");
```

Check Show answer

Exploring further:

- More on stdin, stdout, and stderr from msdn.microsoft.com

(*Pointers) Pointers are described in another section. Knowledge of that section is not essential to understanding the current section.

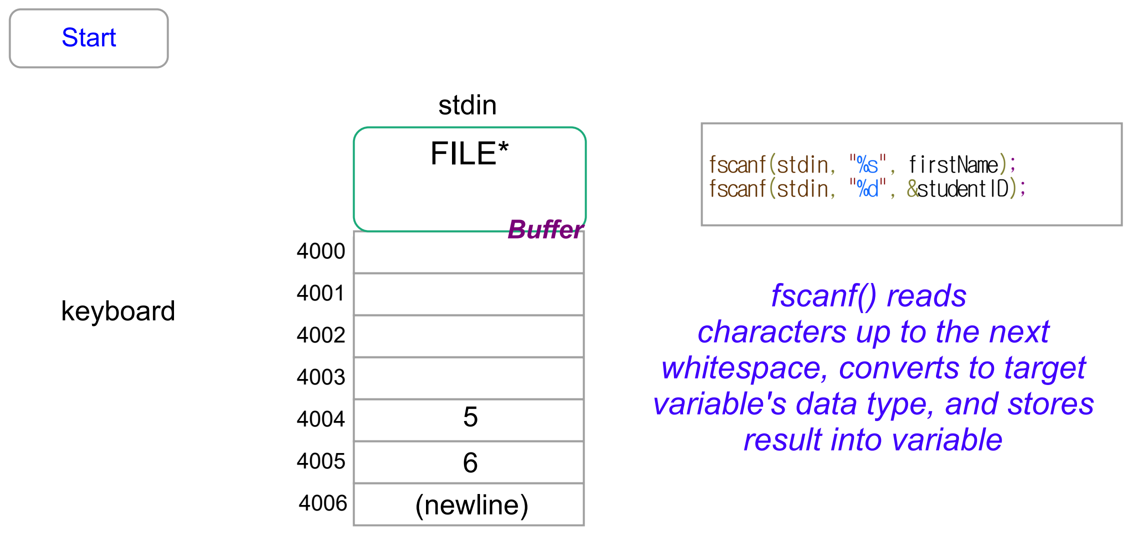
9.2 The stdin file pointer

Programs need a way to receive input data, from a keyboard, touchscreen, or elsewhere. The ***fscanf()*** function is used to read a sequence of characters from a file, storing the converted values into the specified variables; the first "f" stands for "file." The first argument to fscanf() is a FILE* to the file being read. The remaining arguments for fscanf() work the same way as the arguments for scanf().

The second argument for the fscanf() function is the ***format string*** that specifies the type of value to be read using a ***format specifier***. The argument following the format string is the location to store the value that is read.

stdin is a predefined FILE* (a file pointer[FilePointer]) that is pre-associated with a system's standard input, usually a computer keyboard. The system automatically puts the standard input into a data buffer associated with stdin, from which fscanf() can extract data. The following animation illustrates.

PARTICIPATION ACTIVITY 9.2.1: Reading from stdin using fscanf().


Start
stdin
FILE*
Buffer
keyboard
4000
4001
4002
4003
4004
5
4005
6
4006
(newline)
fscanf() reads characters up to the next whitespace, converts to target variable's data type, and stores result into variable

```
fscanf(stdin, "%s", firstName);
fscanf(stdin, "%d", &studentID);
```

Basic use of scanf() and format specifiers were covered in an earlier section, and can similarly be used for fscanf().

PARTICIPATION ACTIVITY 9.2.2: fscanf() and scanf().

1) Write a statement using fscanf() to read a integer value from stdin, storing the value within an int variable named maxEntries.

Check Show answer

2) Write a statement using fscanf() to read a floating-point value from stdin, storing the value within a float variable named tempSetPoint.

Check Show answer

3) Will the following two statement both read a single integer from the standard input (answer Yes or No)?

```
fscanf(stdin, "%d", &x);
scanf("%d", &x);
```

Check Show answer

Exploring further:

- stdin Reference Page from cplusplus.com

(*FilePointer) Pointers are described in another section. Knowledge of that section is not essential to understanding the current section.

9.3 Output formatting

A programmer can adjust the way that output appears, a task known as ***output formatting***. The format specifiers within the format string of printf() and fprintf() can include ***format sub-specifiers***. These sub-specifiers specify how a value stored within a variable will be printed in place of a format specifier.

The formatting sub-specifiers are included between the % and format specifier characters. For example, `printf("%.1f", myFloat);` causes the floating-point variable, myFloat, to be output with only 1 digit after the decimal point; if myFloat was 12.34, the output would be 12.3. Format specifiers and sub-specifiers use the following form:

Construct 9.3.1: Format specifiers and sub-specifiers.

```
%(flags)(width)(.precision)specifier
```

Floating point values

Formatting floating-point output is commonly done using the following sub-specifiers options. For the following assume myFloat has a value of 12.34. Recall that "%f" is used for float values, "%lf" is used for double values, "%e" is used to display float values in scientific notation, and "%le" is used to display double values in scientific notation.

Table 9.3.1: Floating-point formatting.

Sub-specifier	Description	Example

width	Specifies the minimum number of characters to be printed. If the formatted value has more characters than the width, it will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces (or 0's if the '0' flag is specified).	`printf("Value: %7.2f", myFloat);`
		`Value:   12.34`

.precision	Specifies the number of digits to print following the decimal point. If the precision is not specified a default precision of 6 is used.	`printf("%.4f", myFloat);`
		`12.3400`
		
		`printf("%3.4e", myFloat);`
		`1.2340e+01`

flags	-: Left justifies the output given the specified width, padding the output with spaces.
	+: Print a preceding + sign for positive values. Negative numbers are always printed with the - sign.
	0: Pads the output with 0's when the formatted value has fewer characters than the width.
	space: Prints a preceding space for positive value.
		`printf("%+f", myFloat);`
		`+12.340000`
		`printf("%08.2f", myFloat);`
		`00012.34`

Figure 9.3.1: Example output formatting for floating-point numbers.

```
#include <stdio.h>

int main(void) {
   double miles = 0.0;    // User defined distance
   double hrsFly = 0.0;   // Time to fly distance
   double hrsDrive = 0.0; // Time to drive distance

   // Prompt user for distance
   printf("Enter a distance in miles: ");
   scanf("%lf", &miles);

   // Calculate the correspond time to fly/drive distance
   hrsFly = miles / 500.0;
   hrsDrive = miles / 60.0;

   // Output resulting values
   printf("%.2lf miles would take:\n", miles);
   printf("%.2lf hours to fly\n", hrsFly);
   printf("%.2lf hours to drive\n\n", hrsDrive);

   return 0;
}
```

```
Enter a distance in miles: 10.3
10.30 miles would take:
0.02 hours to fly
0.17 hours to drive
```

PARTICIPATION ACTIVITY 9.3.1: Formatting floating point outputs using printf().

What is the output from the following print statements, assuming

```
float myFloat = 45.1342f;
```

1) `printf("%09.3f", myFloat);`

Check Show answer

2) `printf("%.3e", myFloat);`

Check Show answer

3) `printf("%09.2f", myFloat);`

Check Show answer

Integer values

Formatting of integer values can also be done using sub-specifiers. The behavior of sub-specifiers for integer data behave differently than for floating-point values. For the following assume myInt is an int value of 301.

Table 9.3.2: Integer formatting.

Sub-specifier	Description	Example

width	Specifies the minimum number of characters to be printed. If the formatted value has more characters than the width, it will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces (or 0's if the '0' flag is specified).	`printf("Value: %7d", myInt);`
		`Value:     301`

flags	-: Left justifies the output given the specified width, padding the output with spaces.
	+: Print a preceding + sign for positive values. Negative numbers are always printed with the - sign.
	0: Pads the output with 0's when the formatted value has fewer characters than the width.
	space: Prints a preceding space for positive value.
		`printf("%+d", myInt);`
		`+301`
		
		`printf("%08d", myInt);`
		`00000301`
		
		`printf("%+08d", myInt);`
		`+0000301`

Figure 9.3.2: Output formatting for integers.

```
#include <stdio.h>

int main(void) {
   const unsigned long KM_EARTH_TO_SUN = 149598000;        // Dist from Earth to sun
   const unsigned long long KM_PLUTO_TO_SUN = 5906376272;  // Dist from Pluto to sun

   // Output distances with min number of characters
   printf("Earth is %11lu", KM_EARTH_TO_SUN);
   printf(" kilometers from the sun.\n");
   printf("Pluto is %11llu", KM_PLUTO_TO_SUN);
   printf(" kilometers from the sun.\n");

   return 0;
}
```

```
Earth is   149598000 kilometers from the sun.
Pluto is  5906376272 kilometers from the sun.
```

PARTICIPATION ACTIVITY 9.3.2: Formatting integer outputs using printf().

What is the output from the following print statements, assuming

```
int myInt = -713;
```

1) `printf("%+04d", myInt);`

Check Show answer

2) `printf("%05d", myInt);`

Check Show answer

3) `printf("%+02d", myInt);`

Check Show answer

Strings

Formatting of strings can also be done using sub-specifiers. For the following assume myString is the string "Formatting".

Table 9.3.3: String formatting.

Sub-specifier	Description	Example

width	Specifies the minimum number of characters to be printed. If the string has more characters than the width, it will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces.	`printf("%20s String", myString);`
		`          Formatting String`

.precision	Specifies the maximum number of characters to be printed. If the string has more characters than the precision, it will be truncated.	`printf("%.6s", myString);`
		`Format`

flags	-: Left justifies the output given the	`printf("%-20s String", myString);`

	specified width, padding the output with spaces.	Formatting String

Figure 9.3.3: Example output formatting for Strings.

```
#include <stdio.h>

int main(void) {

   printf("Dog age in human years (dogyears.com)₩n₩n");
   printf("-------------------------₩n");

   // set num char for each column, left justified
   printf("%-10s | %-12s₩n", "Dog age", "Human age");
   printf("-------------------------₩n");

   // set num char for each column, first col left justified
   printf("%-10s | %12s₩n", "2 months", "14 months");
   printf("%-10s | %12s₩n", "6 months", "5 years");
   printf("%-10s | %12s₩n", "8 months", "9 years");
   printf("%-10s | %12s₩n", "1 year", "15 years");
   printf("-------------------------₩n");

   return 0;
}
```

```
Dog age in human years (dogyears.com)

-------------------------
Dog age    | Human age
-------------------------
2 months   |    14 months
6 months   |      5 years
8 months   |      9 years
1 year     |     15 years
-------------------------
```

PARTICIPATION ACTIVITY 9.3.3: Formatting string outputs using printf().

What is the output from the following print statements, assuming

```
char myString[30] = "Testing";
```

Make sure all of your responses are in quotes, e.g. "Test".

1) `printf("%4s", myString);`

Check Show answer

2) `printf("%8s", myString);`

Check Show answer

3) `printf("%.4s", myString);`

Show answer

Check Show answer

4) `printf("%.10s", myString);`

Check Show answer

Flushing output

Printing characters from the buffer to the output device (e.g., screen) requires a time-consuming reservation of processor resources; once those resources are reserved, moving characters is fast, whether there is 1 character or 50 characters to print. As such, the system may wait until the buffer is full, or at least has a certain number of characters before moving them to the output device. Or, with fewer characters in the buffer, the system may wait until the resources are not busy. However, sometimes a programmer does not want the system to wait. For example, in a very processor-intensive program, such waiting could cause delayed and/or jittery output. The programmer can use the function ***fflush()***. The fflush() function will immediately flush the contents of the buffer for the specified FILE*. For example, fflush(stdout) will write the contents of the buffer for stdout to the computer screen.

Exploring further:

- More formatting options exist. See printf Reference Page from cplusplus.com.

CHALLENGE ACTIVITY 9.3.1: Output formatting.

Start

Type the program's output.

```
#include <stdio.h>

int main(void) {
   int myInt = 508;

   printf("%2d\n", myInt);
   printf("%5d\n", myInt);

   return 0;
}
```

```
508
  508
```

1 2 3 4 5 6

Check Next

CHALLENGE ACTIVITY 9.3.2: Output formatting.

Write a single statement that prints outsideTemperature with a + or - sign. End with newline. Sample output:

```
+103.500000
```

```
#include <stdio.h>

int main(void) {
   double outsideTemperature = 103.5;

   /* Your solution goes here  */

   return 0;
}
```

Run

CHALLENGE ACTIVITY 9.3.3: Output formatting: Printing a maximum number of digits in the fraction.

Write a single statement that prints outsideTemperature with 2 digits in the fraction (after the decimal point). End with a newline. Sample output:

```
103.46
```

```
#include <stdio.h>

int main(void) {
   double outsideTemperature = 103.45632;

   /* Your solution goes here  */

   return 0;
}
```

Run

9.4 Input parsing

This section describes features of the similar functions ***scanf***, ***fscanf***, and the soon-to-be-introduced sscanf, that support input parsing. The section illustrates using scanf, but the features apply to all three functions.

A programmer can control the way that input is read when using scanf(), a task known as ***input parsing***. The format specifiers within the format string of scanf() can include ***format sub-specifiers***. These sub-specifiers specify how the input will be read for that format specified. One of the most useful specifiers is the width specifier that can be used with the following form:

Construct 9.4.1: Format specifiers and sub-specifiers.

```
%(width)specifier
```

The width specifies the maximum number of character to read for the current format specifier. For example, the format string "%2d" will read in up to 2 characters -- in this case decimal digits -- converting the characters to the corresponding decimal value and storing that value into an integer variable.

A single scanf() statement can be used to read into multiple variables. The format string can include whitespace characters separating the format specifiers. These whitespace characters will cause the scanf() function to read all whitespace characters from the input until a non-whitespace character is reached. For example, the format string "%d %d" will read two decimal integers from the input separated by whitespace. That whitespace may be a single space, a newline, a space followed by a newline, or any combination thereof.

The following program uses a single scanf() statement to read two values for feet and inches, printing to equivalent distance in centimeters.

Figure 9.4.1: Reading multiple values using a single scanf().

```
#include <stdio.h>

const double CM_PER_IN = 2.54;
const int    IN_PER_FT = 12;

/* Converts a height in feet/inches to centimeters */
double HeightFtInToCm(int heightFt, int heightIn) {
   int totIn = 0;
   double cmVal = 0.0;

   totIn = (heightFt * IN_PER_FT) + heightIn; // Total inches
   cmVal = totIn * CM_PER_IN;                 // Conv inch to cm
   return cmVal;
}

int main(void) {
   int userFt = 0; // User defined feet
   int userIn = 0; // User defined inches

   // Prompt user for feet/inches
   printf("Enter feet and inches separated by a space: ");
   scanf("%d %d", &userFt, &userIn);

   // Output converted feet/inches to cm result
   printf("Centimeters: %lf₩n",
          HeightFtInToCm(userFt,userIn));

   return 0;
}
```

```
Enter feet and inches separated by a space: 13 5
Centimeters: 408.940000

...

Enter feet and inches separated by a space: 3 5
Centimeters: 104.140000
```

PARTICIPATION ACTIVITY 9.4.1: Parsing input using scanf().

Answer the following questions assuming the user input is:

```
1053 17.5 42
```

1) What is the value of the variable val3 after the following scanf():
`scanf("%d %f %d", &val1, &val2, &val3);`

Check Show answer

2) What is the value of the variable val3 after the scanf():
`scanf("%2d %f %d", &val1, &val2, &val3);`

Check Show answer

User input often may include additional characters that are common to the format of the data being entered. For example, when receiving a time from a user, the programmer may prefer to allow users to use a common time format, such as "12:35 AM". In this example, the ':' is only used to format the data, separating the hour from the minute value.

The format string for scanf() can be configured to read the ':' character from the input but not store within a variable. scanf() will attempt to read any non-whitespace characters from the input. scanf() will only read the non-whitespace character if that character matches the provided user input.

Ex: the format string "%2d:%2d %2s" can be used to read in a time value:

- The first format specifier "%2d" will read up to two decimal digits for the hour.
- scanf() will then attempt to read a ':' character. If ':' is found in the user input, then ':' will be read and discarded.
- The subsequent two format specifiers will read in the minutes and AM/PM setting.

Figure 9.4.2: An example of using non-whitespace characters in a format string to parse formatted input.

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

int main(void) {
   int currHour = 0;      // User defined hour
   int currMinute = 0;    // User defined minutes
   char optAmPm[3] = ""; // User defined am/pm

   // Prompt user for input
   printf("Enter the time using the format: HH:MM AM/PM: ");
   scanf("%2d:%2d %2s", &currHour, &currMinute, optAmPm);

   // Output time in 12 hrs
   printf("In 12 hours it will be: ");
   if (strcmp(optAmPm, "AM") == 0) {
      printf("%02d:%02d PM\n", currHour, currMinute);
   }
   else {
      printf("%02d:%02d AM\n", currHour, currMinute);
   }

   return 0;
}
```

```
Enter the time using the format: HH:MM AM/PM: 12:35 PM
In 12 hours it will be: 12:35 AM

...

Enter the time using the format: HH:MM AM/PM: 4:12AM
In 12 hours it will be: 04:12 PM
```

Importantly, as soon as scanf() is not able to match the format string, it will stop reading from the input. For example, if the user does not enter the ':' character, scanf() will immediately stop reading from the input. In such a situation the currMinutes and optAmPm variables will not be updated.

PARTICIPATION ACTIVITY 9.4.2: scanf() parsing.

Try running the program with the following user inputs

- 12:35 PM
- 12 35 PM
- "12 35 PM", "Time", "1235"

Load default template...

```
12:35 PM
```

Run

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

int main(void) {
   int currHour = 0;      // User defined hour
   int currMinute = 0;    // User defined minutes
   char optAmPm[3] = ""; // User defined am/pm

   // Prompt user for input
   printf("Enter the time using the format: HH:MM AM/PM: ")
   scanf("%2d:%2d %2s", &currHour, &currMinute, optAmPm);

   // Output time in 12 hrs
   printf("In 12 hours it will be: ");
   if (strcmp(optAmPm, "AM") == 0) {
      printf("%02d:%02d PM\n", currHour, currMinute);
   }
   else {
```

PARTICIPATION ACTIVITY 9.4.3: Parsing non-whitespace characters using scanf().

Assume all variables are initialized to zero. Answer the following questions assuming the user input is:

```
19, 20, 21
```

1) What is the value of the variable val2 after the scanf()?
`scanf("%d %f %d", &val1, &val2, &val3);`

Check Show answer

2) What is the value of the variable val3 after the scanf()?
`scanf("%d, %f, %d", &val1, &val2, &val3);`

Check Show answer

To check for such errors, the scanf() function returns an integer value for the number of items read using scanf() and stored within the specified variables. This return value can be checked to see if the user input matches the specified format. For example, if the user enters a valid time for the format string, scanf() will return 3. The following program extends the earlier example, printing an error message if the user input did not match the specified format string for all three format specifiers.

Figure 9.4.3: Using the return value from scanf() to check for parsing errors.

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

int main(void) {
   int currHour = 0;     // User defined hour
   int currMinute = 0;   // User defined minutes
   char optAmPm[3] = ""; // User defined am/pm

   // Prompt user for input
   printf("Enter the time using the format: HH:MM AM/PM: ");

   // Check number of items read
   if (scanf("%2d:%2d %2s", &currHour, &currMinute, optAmPm) != 3 ) {
      printf("\nInvalid time format\n");
   }
   else {
      printf("In 12 hours it will be: ");
      if (strcmp(optAmPm, "AM") == 0) {
         printf("%02d:%02d PM\n", currHour, currMinute);
      }
      else {
         printf("%02d:%02d AM\n", currHour, currMinute);
      }
   }
   return 0;
}
```

```
Enter the time using the format: HH:MM AM/PM: 12:35 PM
In 12 hours it will be: 12:35 AM

...

Enter the time using the format: HH:MM AM/PM: 412AM

Invalid time format
```

Sometimes a programmer wishes to read input data from a string rather than from the keyboard (standard input). The ***sscanf()*** function is used to read a sequence of characters from a C string, parsing the data stored within that string and storing the converted value within variables. The first argument to sscanf() is the string being read. The remaining arguments for sscanf() work the same way as the arguments for scanf(). Specifically, the second argument for the sscanf() function is the ***format string*** that specifies the type of value to be read using a ***format specifier***. The argument following the format string is the location to store the values that are read.

Unlike the scanf() function that continues reading from the user input where the previous scanf() stopped, sscanf() always starts at the beginning of the specified string. In addition, the contents of the string being read are not modified by sscanf(). The following program illustrates.

Figure 9.4.4: Using sscanf() to parse a string.

```
First name: Amy
Last name: Smith
Age: 19
```

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

int main(void) {
   char myString[100] = "Amy Smith 19";  // Input string
   char firstName[50] = "";              // Last name
   char lastName[50] = "";               // First name
   int userAge = 0;                      // Age

   // Parse input, break up into first/last name and age
   sscanf(myString, "%49s %49s %d", firstName, lastName, &userAge);

   // Output parsed values
   printf("First name: %s\n", firstName);
   printf("Last name: %s\n", lastName);
   printf("Age: %d\n", userAge);

   return 0;
}
```

A common use of scanf() is to process user input line-by-line. The following program reads in the line as a string, and then extracts individual data items from that string.

Figure 9.4.5: Using a sscanf() to parse a line of input text.

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

int main(void) {
   const int USER_TEXT_LIMIT = 1000;   // Limit input size
   char userText[USER_TEXT_LIMIT];     // Holds input
   char firstName[50] = "";            // Last name
   char lastName[50] = "";             // First name
   int userAge = 0;                    // Age
   int valuesRead = 0;                 // Holds number of inputs read
   bool inputDone = false;             // Flag to indicate next iteration

   // Prompt user for input
   printf("Enter ₩"firstname lastname age₩" on each line₩n");
   printf("(₩"Exit₩" as firstname exits).₩n₩n");

   // Grab data as long as "Exit" is not entered
   while (!inputDone) {

      // Grab entire line, store in userText
      fgets(userText, USER_TEXT_LIMIT, stdin);

      // Parse the line and check for correct number of entries.
      valuesRead = sscanf(userText, "%49s %49s %d", firstName, lastName, &userAge);
      if (valuesRead >= 1 && strcmp(firstName, "Exit") == 0) {
         printf("Exiting.₩n");
         inputDone = true;
      }
      else if (valuesRead == 3) {
         printf("   First name: %s₩n", firstName);
         printf("   Last  name: %s₩n", lastName);
         printf("   Age: %d₩n", userAge);
         printf("₩n");
      }
      else {
         printf("Invalid entry. Please try again.₩n₩n");
      }
   }

   return 0;
}
```

```
Enter "firstname lastname age" on each line
("Exit" as firstname exits).

Amy Smith 19
   First name: Amy
   Last  name: Smith
   Age: 19

Mike Smith 24
   First name: Mike
   Last  name: Smith
   Age: 24

No Age
Invalid entry. Please try again.

Exit
Exiting.
```

The program uses fgets() to read an input line into a string. Recall that C string are implemented using character arrays. As the size of the character array -- or string -- must be known before calling fgets(), if the user enters a line of text that is longer than the length of that string, care must be taken to ensure the user input is not written to an out of bounds index.

The second argument to the fgets() function is an integer value specifying the maximum number of characters to write to the specified string. Using this input correctly ensures fgets() will not write to out of range values for the specified string. For example, if inputBuffer is declared as `char inputBuffer[100]`, the statement `fgets(inputBuffer, 100, stdin);` will ensure that no more than 100 characters are written to the string inputBuffer. Additionally, fgets() will ensure that the null character will be written to the end of the string read.

Similarly, when parsing a string -- or user input -- to read a string, the width sub-specifier of the "%s" format specifier should be used. Recall that the width sub-specifier specifies the maximum number of characters to read. If myString is defined char myString[50], the format specifier "%49s" can be used to ensure no more than 49 characters are read from the input, leaving one space for the null character at the end of the string.

A good practice is to always use the width sub-specifier when reading strings using scanf(), fscanf(), or sscanf().

PARTICIPATION ACTIVITY 9.4.4: More input parsing.

Answer the following questions assuming the user input is:

```
1053 17.5 42 Smith
```

1) What is the value of the variable str2 after the scanf() (include quotes in your answer)?
`scanf("%s %d %s", str1, &val1, str2);`

Check Show answer

2) What is the return value from the following scanf():
`scanf("%f %d %d %d", &val1, &val2, &val3, &val4);`

Check Show answer

3) What is the value of the variable str3 after the fgets() (include quotes in your

answer)?
fgets(str3, USER_TEXT_LIMIT,
stdin);

Check Show answer

Exploring further:

- getc() from cplusplus.com
- getchar() from cplusplus.com

CHALLENGE ACTIVITY 9.4.1: Input parsing.

Start

Type the program's output.

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

int main(void) {
   char objectInfo[100] = "Book 12 19";
   char object[50] = "";
   int quantity = 0;
   int price = 0;

   sscanf(objectInfo, "%s %d %d", object, &quantity, &price);

   printf("%s x%d\n", object, quantity);
   printf("Price: %d", price);

   return 0;
}
```

```
Book x12
Price: 19
```

1 2

Check Next

CHALLENGE ACTIVITY 9.4.2: Input parsing: Reading an entire line.

Write a single statement that reads an entire line from stdin. Assign streetAddress with the user input. Ex: If a user enters "1313 Mockingbird Lane", program outputs:

```
You entered: 1313 Mockingbird Lane
```

```
#include <stdio.h>

int main(void) {
   const int ADDRESS_SIZE_LIMIT = 50;
   char streetAddress[ADDRESS_SIZE_LIMIT];

   printf("Enter street address: ");

   /* Your solution goes here  */

   printf("You entered: %s", streetAddress);

   return 0;
}
```

Run

CHALLENGE ACTIVITY 9.4.3: Input parsing: Reading multiple items.

Complete scanf() to read two comma-separated integers from stdin. Assign userInt1 and userInt2 with the user input. Ex: "Enter two integers separated by a comma: 3, 5", program outputs:

```
3 + 5 = 8
```

```
#include <stdio.h>

int main(void) {
   int userInt1 = 0;
   int userInt2 = 0;

   printf("Enter two integers separated by a comma: ");
   scanf(/* Your solution goes here  */);
   printf("%d + %d = %d\n", userInt1, userInt2, userInt1 + userInt2);

```

```
    return 0;
}
```

Run

9.5 File input and output

Sometimes a program should get input from a file rather than from a user typing on a keyboard. To achieve this, a programmer can open another input file, rather than the predefined input file stdin that comes from the standard input (keyboard). That new input file can then be used with fscanf() just like using scanf() with the stdin file, as the following program illustrates. Assume a text file exists named myfile.txt with the contents shown (created for example using Notepad on a Windows computer or using TextEdit on a Mac computer).

Figure 9.5.1: Input from a file.

myfile.txt with two integers:

```
5
10
```

```
Opening file myfile.txt.
Reading two integers.
Closing file myfile.txt.
num1 = 5
num2 = 10
num1+num2 = 15
```

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

int main(void) {
   FILE* inFile = NULL; // File pointer
   int fileNum1 = 0;    // Data value from file
   int fileNum2 = 0;    // Data value from file

   // Try to open file
   printf("Opening file myfile.txt.\n");

   inFile = fopen("myfile.txt", "r");
   if (inFile == NULL) {
      printf("Could not open file myfile.txt.\n");
      return -1; // -1 indicates error
   }

   // Can now use fscanf(inFile, ...) like scanf()
   // myfile.txt should contain two integers, else problems
   printf("Reading two integers.\n");
   fscanf(inFile, "%d %d", &fileNum1, &fileNum2);

   // Done with file, so close it
   printf("Closing file myfile.txt.\n");
   fclose(inFile);

   // Output values read from file
   printf("num1 = %d\n", fileNum1);
   printf("num2 = %d\n", fileNum2);
   printf("num1+num2 = %d\n", (fileNum1 + fileNum2));

   return 0;
}
```

Six lines are needed for input from a file, highlighted above.

- The `#include <stdio.h>` enables use of FILE* variables and supporting functions.
- A new FILE* variable has been declared: `FILE* inputFile;`.FilePointer
- The line `inputFile = fopen("myfile.txt", "r");` then opens the file for reading and associates the file with the FILE*. The first argument to ***fopen()*** is a string with the name of the file to open. The second argument of fopen() is a string indicating the file mode, which specifies if the file should be open for reading or writing. The string "r" indicates the file should be open for reading, referred to as ***read mode***. Upon success, fopen() will return a pointer to the FILE structure for the file that was opened. If fopen() could not open the file, it will return NULL.
- Because of the high likelihood that the open fails, usually because the file does not exist or is in use by another program, the program checks whether the open was successful using `if (inputFile == NULL)`.
- The successfully opened input file is read from using fscanf(), e.g., using `fscanf(inFile, "%d %d", &num1, &num2);` to read two integers into num1 and num2.
- Finally, when done using the file, the program closes the file using `fclose(inputFile);`.

A common error is to specify the file mode as a character (e.g. 'r') rather than a string (e.g."r"). Another common error is a mismatch between the variable data type and the file data, e.g., if the data type is int but the file data is "Hello".

Try 9.5.1: Good and bad file data.

File input, with good and bad data: Create myfile.txt with contents 5 and 10, and run the above program. Then, change "10" to "Hello" and run again, observing the incorrect output.

The following provides another example wherein the program reads items into a dynamically allocated array. For this program, myfile.txt's first entry must be the number of numbers to read, followed by those numbers, e.g., 5 10 20 40 80 1.

Figure 9.5.2: Program that reads data from myfile.txt into an array.

myfile.txt file contents:

```
5
10
20
40
80
1
```

```
Numbers: 10 20 40 80 1
```

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

int main(void) {
   FILE* inFile = NULL; // File pointer
   int* userNums;       // User numbers; memory allocated later
   int arrSize = 0;     // User-specified number of numbers
   int i = 0;           // Loop index


   // Try to open the file
   inFile = fopen("myfile.txt", "r");

   if (inFile == NULL) {
      printf("Could not open file myfile.txt.\n");
      return -1; // -1 indicates error
   }

   // Can now use fscanf(inFile, ...) like scanf()
   // myfile.txt should contain two integers, else problems
   fscanf(inFile, "%d", &arrSize);

   // Allocate enough memory for nums
   userNums = (int*)malloc(sizeof(int)*arrSize);
   if (userNums == NULL) {
      fclose(inFile); // Done with file, so close it
      return -1;
   }

   // Get user specified numbers. If too few, may encounter problems
   i = 1;
   while (i <= arrSize) {
      fscanf(inFile, "%d", &(userNums[i-1]));
      i = i + 1;
   }

   // Done with file, so close it
   fclose(inFile);

   // Print numbers
   printf("Numbers: ");

   i = 0;
   while (i < arrSize) {
      printf("%d ", userNums[i]);
      ++i;
   }

   printf("\n");

   return 0;
}
```

A program can read varying amounts of data in a file by using a loop that reads until the end of the file has been reached, as follows.

The ***feof()*** function returns 1 if the previous read operation reached the end of the file. Errors may be encountered while attempting to read from a file, including end-of-file, corrupt data, etc. So, a program should check that each read was successful before using the variable to which the data read was assigned. fscanf() returns the number of items read from the file and assigned to a variable, which can be checked to determine if the read operation was successful. Ex: `if( fscanf(inFile, "%d", &fileNum) == 1 ) {...}` checks that fscanf() read and assigned a value to fileNum.

Figure 9.5.3: Reading a varying amount of data from a file.

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

int main(void) {
   FILE* inFile = NULL; // File pointer
   int fileNum = 0;     // Data value from file
   int numRead = 0;

   // Open file
   printf("Opening file myfile.txt.\n");
   inFile = fopen("myfile.txt", "r");

   if (inFile == NULL) {
      printf("Could not open file myfile.txt.\n");
      return -1; // -1 indicates error
   }

   // Print read numbers to output
   printf("Reading and printing numbers.\n");

   while (!feof(inFile)) {
      numRead = fscanf(inFile, "%d", &fileNum);
      if ( numRead == 1 ) {
         printf("num: %d\n", fileNum);
      }
   }

   printf("Closing file myfile.txt.\n");

   // Done with file, so close it
   fclose(inFile);

   return 0;
}
```

myfile.txt with variable number of integers:

```
111
222
333
444
555
```

```
Opening file myfile.txt.
Reading and printing numbers.
num: 111
num: 222
num: 333
num: 444
num: 555
Closing file myfile.txt.
```

Similarly, a program may write output to a file rather than to standard output, as shown below. To open an output file, the string "w" is used as the file mode within the call to fopen(), referred to as ***write mode***. Using the write mode, if a file with specified name already exists, that file will be replaced with the newly created file.

Figure 9.5.4: Sample code for writing to a file.

Contents of myoutfile.txt after running the program:

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

int main(void) {
   FILE* outFile = NULL; // File pointer

   // Open file
   outFile = fopen("myoutfile.txt", "w");

   if (outFile == NULL) {
      printf("Could not open file
myoutfile.txt.₩n");
      return -1; // -1 indicates error
   }

   // Write to file
   fprintf(outFile, "Hello₩n");
   fprintf(outFile, "1 2 3₩n");

   // Done with file, so close it
   fclose(outFile);

   return 0;
}
```

```
Hello
1 2 3
```

fopen() supports several additional file modes. See http://www.cplusplus.com/reference/cstdio/fopen/.

PARTICIPATION ACTIVITY 9.5.1: Opening file using open().

Answer the following assuming the file "file1.txt" exists and can be accessed by the user and "file2.txt" does not exist.

1) Write a statement to open the "file1.txt" for input, assigning the return from fopen() to a FILE* variable named inputFile.

Check Show answer

2) What is the value of the FILE* inputFile after the following call to fopen():
`inputFile = fopen("file2.txt", "r");`

Check Show answer

3) Write a statement to open the "file2.txt" for output, assigning the return from fopen() to a FILE* variable named outputFile.

Check Show answer

4) Write a statement that can read in data from an already established input file `inputFile` until the end of file has been reached.

```
while (                ) {
   // Read/manipulate file data
}
```

Check Show answer

Exploring further:

- stdlib.h reference page from cplusplus.com

(*FilePointer) Pointers are described in another section. Knowledge of that section is not essential to understanding the current section.