11.1 Do-while loops

A ***do-while loop*** is a loop construct that first executes the loop body's statements, then checks the loop condition.

Construct 11.1.1: Do-while loop.

```
do {
   // Loop body
} while (loopExpression);
```

Versus a while loop, a do-while loop is useful when the loop should iterate at least once.

PARTICIPATION ACTIVITY 11.1.1: Do-while loop.

Start

```
#include <stdio.h>

int main(void) {
   char  fill = '*';

   do {
      printf("\n");
      printf("%c %c %c\n", fill, fill, fill);
      printf("%c %c %c\n", fill, fill, fill);
      printf("%c %c %c\n", fill, fill, fill);
      printf("Enter char (q to quit): ");
      scanf(" %c", &fill);
   } while (fill != 'q');

   return 0;
}
```

```
***
***
***
Enter char (q to quit):   x

XXX
XXX
XXX
Enter char (q to quit):   q

(program done)
```


PARTICIPATION ACTIVITY 11.1.2: Do-while loop.

Consider the following loop:

```
int count = 0;
int num = 6;
do {
   num = num - 1;
   count = count + 1;
} while (num > 4);
```

1) What is the value of count after the loop?

- 0
- 1
- 2

2) What initial value of num would prevent count from being incremented?

- 4
- 0
- No such value.

CHALLENGE ACTIVITY 11.1.1: Basic do-while loop with user input.

Complete the do-while loop to output 0 to countLimit using printVal. Assume the user will only input a positive number.

```
1  #include <stdio.h>
2
3  int main(void) {
4     int countLimit = 0;
5     int printVal = 0;
6
7     // Get user input
8     scanf("%d", &countLimit);
9
10    printVal = 0;
11    do {
12       printf("%d ", printVal);
13       printVal = printVal + 1;
14    } while ( /* Your solution goes here  */ );
15    printf("\n");
16
17    return 0;
18 }
```


Run

CHALLENGE ACTIVITY 11.1.2: Do-while loop to prompt user input.

Write a do-while loop that continues to prompt a user to enter a number less than 100, until the entered number is actually less than 100. End each prompt with newline. Ex: For the user input 123, 395, 25, the expected output is:

```
Enter a number (<100):
Enter a number (<100):
Enter a number (<100):
Your number < 100 is: 25
```



```
#include <stdio.h>

int main(void) {
   int userInput = 0;

   /* Your solution goes here  */

   printf("Your number < 100 is: %d\n", userInput);

   return 0;
}
```

Run

11.2 Engineering examples

Arrays can be useful in solving various engineering problems. One problem is computing the voltage drop across a series of resistors. If the total voltage across the resistors is V, then the current through the resistors will be I = V/R, where R is the sum of the resistances. The voltage drop Vx across resistor x is then Vx = I · Rx. The following program uses an array to store a user-entered set of resistance values, computes I, then computes the voltage drop across each resistor and stores each in another array, and finally prints the results.

Figure 11.2.1: Calculate voltage drops across series of resistors.

```
#include <stdio.h>

int main(void) {
   const int NUM_RES = 5;      // Number of resistors
   double resVals[NUM_RES];    // Ohms
   double circVolt = 0;        // Volts
   double vDrop[NUM_RES];      // Volts
   double currentVal = 0;      // Amps
   double sumRes = 0;          // Ohms
   int i = 0;                  // Loop index

   printf("5 resistors are in series.\n");
   printf("Program calculates voltage drop ");
   printf("across each resistor.\n");

   printf("Input voltage applied to circuit: ");
   scanf("%lf", &circVolt);

   printf("Input ohms of %d resistors:\n", NUM_RES);
   for (i = 0; i < NUM_RES; ++i) {
      printf("%d) ", i+1);
      scanf("%lf", &resVals[i]);
   }

   // Calculate current
   for (i = 0; i < NUM_RES; ++i) {
      sumRes = sumRes + resVals[i];
   }
   currentVal = circVolt / sumRes;    // I = V/R

   for (i = 0; i < NUM_RES; ++i) {
      vDrop[i] = currentVal * resVals[i]; // V = IR
   }

   printf("\nVoltage drop per resistor is:\n");
   for (i = 0; i < NUM_RES; ++i) {
      printf("%d) %.1lf V\n", i + 1, vDrop[i]);
   }

   return 0;
}
```

```
5 resistors are in series.
Program calculates voltage drop across each resistor.
Input voltage applied to circuit: 12
Input ohms of 5 resistors:
1) 3.3
2) 1.5
3) 2
4) 4
5) 2.2

Voltage drop per resistor is:
1) 3.0 V
2) 1.4 V
3) 1.8 V
4) 3.7 V
5) 2.0 V
```


PARTICIPATION ACTIVITY 11.2.1: Voltage drop program.

1) What does variable circVolt store?

- ○ Multiple voltages, one for each resistor.
- ○ The resistance of each resistor.
- ○ The total voltage across the series of resistors.

2) What does the first for loop do?

- Gets the voltage of each resistor and stores each in an array.
- Gets the resistance of each resistor and stores each in an array.
- Adds the resistances into a total value.

3) What does the second for loop do?

- Adds the resistances into a single value, so that I = V/R can be computed.
- Computes the voltage across each resistor.

4) What does the third for loop do?

- Update the resistances array with new resistor values.
- Sum the voltages across each resistor into a total voltage.
- Determines the voltage drop across each resistor and stores each voltage in another array.

5) Could the fourth loop's statement have been incorporated into the third loop, thus eliminating the fourth loop?

- No, a resistor's voltage drop isn't known until the entire loop has finished.
- Yes, but keeping the loops separate is better style.

Engineering problems commonly involve matrix representation and manipulation. A matrix can be captured using a two-dimensional array. Then matrix operations can be defined on such arrays. The following illustrates matrix multiplication for 4x2 and 2x3 matrices captured as two-dimensional arrays.

Figure 11.2.2: Matrix multiplication of 4x2 and 2x3 matrices.

```
15 20 24
10 14 17
 5 14 19
 0  4  6
```

```
#include <stdio.h>

int main(void) {
   const int M1_ROWS = 4;       // Matrix 1 rows
   const int M1_COLS = 2;       // Matrix 1 cols
   const int M2_ROWS = M1_COLS; // Matrix 2 rows (must have same value)
   const int M2_COLS = 3;       // Matrix 2 cols
   int i = 0;                   // Loop index
   int j = 0;                   // Loop index
   int k = 0;                   // Loop index
   int dotProd = 0;             // Dot product

   // Populate matrices
   int m1[4][2] = {{3, 4},
                   {2, 3},
                   {1, 5},
                   {0, 2}};

   int m2[2][3] = {{5, 4, 4},
                   {0, 2, 3}};

   int m3[4][3] = {{0, 0, 0},
                   {0, 0, 0},
                   {0, 0, 0},
                   {0, 0, 0}};

   // m1 * m2 = m3
   for (i = 0; i < M1_ROWS; ++i) {
      for (j = 0; j < M2_COLS; ++j) {
         // Compute dot product
         dotProd = 0;
         for (k = 0; k < M2_ROWS; ++k) {
            dotProd = dotProd + (m1[i][k] * m2[k][j]);
         }

         m3[i][j] = dotProd;
      }
   }

   // Print m3 result
   for (i = 0; i < M1_ROWS; ++i) {
      for (j = 0; j < M2_COLS; ++j) {
         printf("%2d ", m3[i][j]);
      }
      printf("Wn");
   }

   return 0;
}
```


PARTICIPATION ACTIVITY 11.2.2: Matrix multiplication program.

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

1) For the first set of for loops, how many dot products are computed? (In other words, how many iterations are due to the outer two for loops?)

Check Show answer

2) For the first set of for loops, the inner-most loop computes a dot product. Each time that inner-most loop is reached, how many times will it iterate?

Check Show answer

11.3 Command-line arguments

Command-line arguments are values entered by a user when running a program from a command line. A *command line* exists in some program execution environments, wherein a user types a program's name and any arguments at a command prompt. To access those arguments, main() can be defined with two special parameters argc and argv, as shown below. The program prints provided command-line arguments. (The "for" loop is not critical to understanding the point, in case you haven't studied for loops yet). The program's executable is named argtest.

Figure 11.3.1: Printing command-line arguments.

```
#include "stdio.h"

int main(int argc, char* argv[]) {
   int i = 0;

   // Prints argc and argv values
   printf("argc: %dWn", argc);
   for (i = 0; i < argc; ++i) {
      printf("argv[%d]: %sWn", i, argv[i]);
   }

   return 0;
}
```

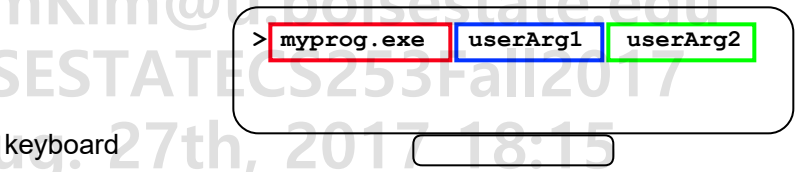
```
> ./argtest
argc: 1
argv[0]: ./argtest

> ./argtest Hello
argc: 2
argv[0]: ./argtest
argv[1]: Hello

> ./argtest Hey ABC 99 -5
argc: 5
argv[0]: ./argtest
argv[1]: Hey
argv[2]: ABC
argv[3]: 99
argv[4]: -5
```


When a program is run, the system passes an int parameter ***argc*** to main(), indicating the number of command-line arguments (argc is short for argument count). The number includes the program name itself, so argc is 2 for the command line: `myprog.exe myfile.txt`.

When a program is run, the system passes a second parameter ***argv*** to main() (argv is short for argument vector), defined as an array of strings: char* argv[]. argv[] consists of one string for each

command-line argument, with argv[0] being the program name.

PARTICIPATION ACTIVITY 11.3.1: Command-line arguments.



User text typed on the command line is passed to the main() function using parameters: int argc, and char argv[]*

argc = 3
argv[0] = "myprog.exe"
argv[1] = "userArg1"
argv[2] = "userArg2"

PARTICIPATION ACTIVITY 11.3.2: Command-line arguments.

1) What is argc for: :
`myprog.exe 13 14 smith`

Check Show answer

2) What is argc for:
`a.out 12:55 PM`

Check Show answer

3) What is the string in argv[2] for:
`a.out Jan Feb Mar`

Check Show answer

The following program, named myprog, expects two command-line arguments.

Figure 11.3.2: Simple use of command-line arguments.

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

/* Usage: program username userage */
int main(int argc, char* argv[]) {
   char nameStr[100] = ""; // User name
   char ageStr[100]  = "";  // User age

   // Get inputs from command line
   strcpy(nameStr, argv[1]);
   strcpy(ageStr, argv[2]);

   // Output result
   printf("Hello %s. ", nameStr);
   printf("%s is a great age.₩n", ageStr);

   return 0;
}
```

```
> myprog.exe Amy 12
Hello Amy. 12 is a great age.
...

> myprog.exe Rajeev 44 HEY
Hello Rajeev. 44 is a great age.
...

> myprog.exe Denming
Segmentation fault
```

For simplicity, the above program used character arrays for the string variables name and ageStr. For those familiar with char* and malloc(), dynamically allocating the appropriately-sized C strings would be preferred.

However, there is no guarantee a user will type two command-line arguments. Extra arguments, like "HEY" above, are ignored. Conversely, too few arguments can cause a problem. In particular, a common error is to access elements in argv without first checking argc to ensure the user entered enough arguments, resulting in an out-of-range array access. In the last run above, the user typed too few arguments, causing an out-of-range array access.

When a program uses command-line arguments, good practice is to check argv for the correct number of arguments. If the number of command-line arguments is incorrect, good practice is to print a usage message. A ***usage message*** lists a program's expected command-line arguments. The program should then return 1, indicating to the system that an error occurred (whereas 0 means no error).

Figure 11.3.3: Checking for proper number of command-line arguments.

```
> myprog.exe Amy 12
Hello Amy. 12 is a great age.

...

> myprog.exe Denming
Usage: myprog.exe name age

...

> myprog.exe Alex 26 pizza
Usage: myprog.exe name age
```

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

/* Usage: program username userage */
int main(int argc, char* argv[]) {
   char nameStr[100] = "";  // User name
   char ageStr[100]  = "";  // User age

   // Check if correct number of arguments provided
   if (argc != 3) {
      printf("Usage: myprog.exe name ageWn");
      return 1; // 1 indicates error
   }
   // Grab inputs from command line
   strcpy(nameStr, argv[1]);
   strcpy(ageStr, argv[2]);

   // Output result
   printf("Hello %s. ", nameStr);
   printf("%s is a great age.Wn", ageStr);

   return 0;
}
```


PARTICIPATION ACTIVITY 11.3.3: Checking the number of command-line arguments.

1) If a user types the wrong number of command-line arguments, good practice is to print a usage message.
   - True
   - False

2) If a user types too many arguments but a program doesn't check for that, the program typically crashes.
   - True
   - False

3) If a user types too few arguments but a program doesn't check for that, the program typically crashes.
   - True
   - False

Command-line arguments are C strings. The statement age = atoi(ageStr); converts the ageStr string into an integer, assigning the result into int variable age. So string "12" becomes integer 12. atoi() converts a C string to an integer, and is made available via #include <stdlib.h>.

Putting quotes around an argument allows an argument's string to have any number of spaces.

Figure 11.3.4: Quotes surround the single argument 'Mary Jo'.

```
myprog.exe "Mary Jo" 50
```


PARTICIPATION ACTIVITY 11.3.4: String and integer command-line arguments.

1) What is the string in argv[1] for the following:

```
a.out Amy Smith 19
```

Check Show answer

2) What is the string in argv[1] for the following:

```
a.out "Amy Smith" 19
```

Check Show answer

3) Given the following code snippet, complete the assignment of userNum with argv[1].

```
int main(int argc, char* argv[]) {
   int userNum = 0;
   userNum = /* COMPLETE ASSIGNMENT */;
}
```

userNum = ;

Check Show answer

Exploring further:

- Command-line arguments from cplusplus.com (Applies equally to C)

11.4 The #define directive

The ***#define*** directive, of the form #define MACROIDENTIFIER replacement, instructs the processor to replace any occurrence of MACROIDENTIFIER in the subsequent program code by the replacement text.

Construct 11.4.1: #define directive.

```
#define MACROIDENTIFIER replacement
```

#define is sometimes called a ***macro***. The #define line does *not* end with a semicolon.

Most uses of #define are strongly discouraged by experienced programmers. However, for legacy reasons, #define appears in much existing code, so a programmer still benefits from understanding #define.

One (discouraged) use of #define is for a constant. The directive `#define MAXNUM 99` causes the preprocessor to replace every occurrence of identifier MAXNUM by 99, before continuing with compilation. So `if (x < MAXNUM)` will be replaced by `if (x < 99)`. In contrast, declaring a constant variable `const int MAXNUM = 99` has several advantages over a macro, such as type checking, syntax errors for certain incorrect usages, and more.

Another (discouraged) use of #define is for a type-neutral function. #define may specify arguments, as in `#define FCT1(a, b) ((a + b)/(a * b))`. A program may then have statements like `numInt = FCT1(1,2)` or like `numFloat = FCT1(1.2, 0.7)`, which the preprocessor would replace by `numInt = ((1 + 2) / (1 * 2))` or `numFloat = ((1.2 + 0.7) / (1.2 * 0.7))`, respectively. However, defining functions for each type is better practice.

Some advanced techniques make good use of macros, and are mentioned in other sections, such as a section introducing the assert macro.

PARTICIPATION ACTIVITY 11.4.1: #define.

Given:

```
#define PI_CONST  3.14159

double CircleArea(double radius) {
   return PI_CONST * radius * radius;
}
```

1) The function call CircleArea(1.0) returns 3.14159.

- ○ True
- ○ False

2) Replacing the return expression by `PI_CONST * PI_CONST * radius` yields a compiler error since only one replacement is allowed.

- ○ True
- ○ False

3) Replacing the return expression by `PI_CONSTPI_CONST` would yield 3.141593.14159, which would thus yield a compiler error complaining about the two decimal points.

- ○ True
- ○ False

4) The call CircleArea(PI_CONST) would yield a compiler error complaining that the macro cannot be an argument.

- ○ True
- ○ False

5) Placing a semicolon at the end of the #define line will yield a compiler error at that line.

- ○ True
- ○ False

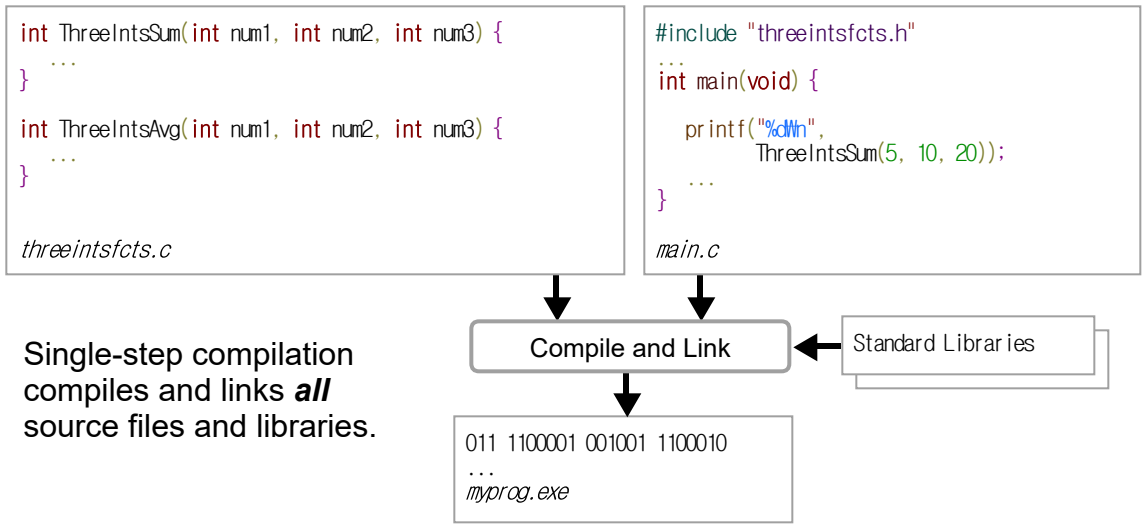
Exploring further:

- Other preprocessor directives include ***#undef***, ***#ifdef***, ***#if***, ***#else***, ***#elif***, ***#pragma***, ***#line***, and ***#error***.
- Preprocessor tutorial on cplusplus.com
- Preprocessor directives on MSDN

11.5 Modular compilation

Using separate files to organize code can also help to make the compilation process more efficient. So far, we used a ***single-step compilation*** approach in which all source files are compiled at the same time to create the executable, e.g. gcc main.c threenumsfcts.c. This approach has a significant drawback. Anytime one of the source files is modified, all files must be recompiled. Even a small change, such as modifying a printf statement to fix a spelling error, would require all files be recompiled. The following animation illustrates the single-step compilation.

PARTICIPATION ACTIVITY 11.5.1: Single-step compilation process.



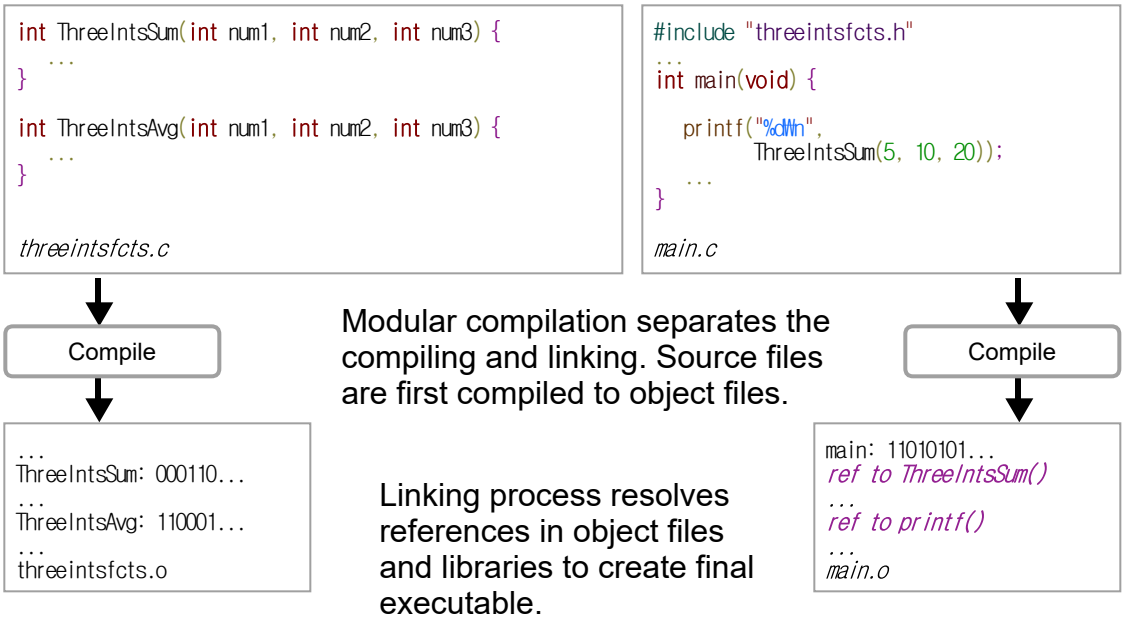
As programs become larger and the number of source files increases, the time required to recompile and link all source files can become very long - often requiring minutes to hours. Instead of compiling an executable using a single step, a ***modular compilation*** approach can be used that separates the compiling and linking steps within the compilation process. In this approach, each source file is independently compiled into an object file. An ***object file*** contains machine instructions for the compiled code along with placeholders, often referred to as references, for calls to functions or accesses to variables or classes defined in other source files or libraries.

For a program involving two files main.c and threenumsfcts.c, the files can be compiled separately into two object files named main.o and threenumsfcts.o respectively. The resulting object files will include

several placeholders for functions that are defined in other files. For example, the main.o object file may contain placeholders for calls to any functions in threenumsfcts.o.

After each source file has been compiled, the ***linker*** will create the final executable by ***linking*** together the object files and libraries. For each placeholder found within an object file, the linker searches the other object files and libraries to find the referenced function or variable. When linking the main.o and threenumsfcts.o files, the placeholder for a call to a function in main.o will be replaced with a jump to the first instruction of that function within the threenumsfcts.o object file. This creates a link between the code within the main.o and threenumsfcts.o object files. Once all placeholders have been linked together, the final executable can be created. The following animation illustrates.

PARTICIPATION ACTIVITY 11.5.2: Modular compilation process.



Using a modular compilation approach has the benefit of reducing the time required to recompile and link the program executable. Instead of recompiling all source files, only the source files that have been modified need to be recompiled to create an updated object file. The linker can then use these newly recompiled object files along with the previously compiled object files for any unmodified source files to create the updated executable.

PARTICIPATION ACTIVITY 11.5.3: Modular compilation.

1) A single-step compilation approach is faster than a modular compilation

approach.

- True
- False

2) An object file contains machine instructions for the compiled code, along with references to functions or access to variables or classes in other files or libraries.

- True
- False

3) A linker is responsible for creating object files.

- True
- False

11.6 Makefiles

While modular compilation offers many advantages, manually compiling and linking a program with numerous source and header files can be challenging. For example, a change within a single header file will require recompiling all source files that include that header file. Keeping track of these dependencies to determine which files need to be recompiled is not trivial and can require considerable effort, if done manually. For large programs, programmers often utilize ***project management tools*** to automate the compilation and linking process. ***make*** is one project management tool that is commonly used on Unix and Linux computer systems.

The make utility uses a ***makefile*** to recompile and link a program whenever changes are made to the source or header files. The makefile consists of a set of rules and commands. ***Make rules*** are used to specify dependencies between a target file (e.g., object files and executable) and a set of prerequisite files that are needed to generate the target file (e.g., source and header files). A make rule can include one or more commands -- referred to as ***make recipe*** -- that will be executed in order to generate the target file. The following shows the general form for a make rule. The make rule's target and prerequisites are defined on a single line. The commands for the make rule should be specified one per line starting with a single tab character.

Construct 11.6.1: Makefile rules and commands.

```
target : prerequisite1 prerequisite2 ... prerequisiteN
    command1
    command2
    ...
    commandN
```

A common error is to use spaces instead of a single tab character for specifying the commands. Another common error is including a tab character on an empty line, which will be reported by make as an error. As distinguishing between tabs and spaces in some text editors can be difficult, these errors are often challenging to find.

The following provides an example makefile for a program consisting of three files main.c, threeintsfcts.c, and threeintsfcts.h. The executable named myprog.exe is dependent on the object files main.o and threeintsfcts.o. The main.o object file is dependent on the main.c source file and the threeintsfcts.h header file. The threeintsfcts.o object file is dependent on the threeintsfcts.c and threeintsfcts.f files.

Figure 11.6.1: Makefile example for three integer functions program.

```
myprog.exe : main.o threeintsfcts.o
    gcc main.o threeintsfcts.o -o myprog.exe

main.o : main.c threeintsfcts.c threeintsfcts.h
    gcc -Wall -c main.c

threeintsfcts.o : threeintsfcts.c threeintsfcts.h
    gcc -Wall -c threeintsfcts.c

clean :
    rm *.o myprog.exe
```

Running make for first time:	`> make`
	`gcc -Wall -c main.c`
	`gcc -Wall -c threeintsfcts.c`
	`gcc main.o threeintsfcts.o -o myprog.exe`

Running make after modifying main.c:	`> make`
	`gcc -Wall -c main.c`
	`gcc main.o threeintsfcts.o -o myprog.exe`

When the make command is executed, if any of the prerequisites for the rule have been modified since the target was last created, the commands for the rule will be executed to create the target file. For example, in the above makefile, if main.c is modified make will first execute the command gcc -Wall -c main.c to create the main.o object file. The -c flag is used here to inform the compiler (e.g. gcc) that the source file should only be compiled (and not linked) to create an object file. As main.o has now been modified, make will then execute the command `gcc main.o threeintsfcts.o -o myprog.exe`. The -o

flag is used here to inform the linker (e.g. gcc) to link the object files into the final executable using the name specified after the -o. In this case, the executable is named myprog.exe.

Make rules can also be used to define common operations used when managing larger programs. One common make rule is the `clean : rule` that is used to execute a command for deleting all generated files such as object files and the program executable. In the above example, this is accomplished by running the command `rm *.o myprog.exe`.

By default make assumes the makefile is named `makefile` or `Makefile`. The -f flag can be used to run make using a different filename. For example, `make -f MyMakefile` will run make using the file named `MyMakefile`.

PARTICIPATION ACTIVITY 11.6.1: Makefiles.

Answer the following using the Makefile provided above.

1) List the targets affected if main.c is changed. (Note: List the targets separated by spaces and in the order that their rules would execute.)

Check Show answer

2) What command will execute after threeintsfcts.o is changed? (Hint: The answer starts with gcc.)

Check Show answer

3) How many commands will execute when threeintsfcts.h is changed?

Check Show answer

Exploring further:

- Makefile tutorial from cprogramming.com

- Manual for make from gnu.org

11.7 Binary file I/O

Programs can also access files using a ***binary file mode***. Using the binary file mode, a program can directly copy data in the program's memory to and from the file. This mode is referred to as binary mode because reading and writing to the file will copy the contents to and from memory bit by bit.

Binary files provide several advantages.

- Reading from and writing to binary files is very efficient because these operations directly copy data without first converting the data into a human readable format. For example, consider writing a 32-bit integer value, such as -13645766, to an output file. Using a character file, this 32-bit integer would first need to be converted to the sequence of characters "-13645766" representing that integer value. When writing to a binary file, this conversion is not needed.
- Binary files can be more space efficient than character files. Again, consider writing the same integer value -13645766 to an output file. Using a binary file, the value requires 32-bits -- or 4 bytes -- as the integer value is stored in memory using 4 bytes. Using a character file would require 9 bytes -- one byte for each character within "-13645766".

Binary files also have disadvantages. The main disadvantage of binary files is that they cannot be easily read or edited by humans. A good practice is to only use binary files when those files will never be directly read or edited by a user.

PARTICIPATION ACTIVITY 11.7.1: Binary files basics.

1) Binary files are human readable.
   - True
   - False

2) Binary files are both more efficient with memory space, as well as, reading and writing operations.
   - True
   - False

To open a file using a binary mode, the file mode string in the call to fopen() should include the character b. To open a file for reading using the binary mode, the file mode string "rb" would be used. Similarly, to open a file for writing using the binary mode, the file mode string "wb" would be used.

A program can write to a binary file using the fwrite() function using the following form:

Construct 11.7.1: fwrite() function.

```
fwrite(variablePointer, sizeof(type), numElements, outputFile)
```


The first argument of fwrite() is a pointer to the variable, or memory location, that will be copied and written to the binary file. The second and third arguments specify the size in bytes of each element and the number of elements, respectively, being written to the file. The sizeof() operation can be used to determine the number of bytes required for the data type of the variable or array being written. The final argument to fwrite() is a FILE* for the output file being written. fwrite() will return the total number of bytes written to the output file.

The following demonstrates the use of fwrite() to write several user-entered integer numbers to a binary file.

Figure 11.7.1: Sample code for writing an array of integers one-by-one to a binary file.

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

int main(void) {
   const int NUM_VALUES = 8; // Num user numbers
   FILE* outFile = NULL;     // File pointer
   int userNums[NUM_VALUES]; // User numbers
   int i = 0;                // Loop index

   // Get numbers from user
   printf("\nEnter %d numbers...\n", NUM_VALUES);
   for (i = 0; i < NUM_VALUES; ++i) {
      printf("%d: ", i);
      scanf("%d", &(userNums[i]));
   }

   // Try to open the file
   outFile = fopen("myfile.bin", "wb");
   if( outFile == NULL ) {
      printf("Could not open file myfile.bin.\n");
      return -1; // -1 indicates error
   }

   // Write user values to output file
   printf("Writing numbers to myfile.bin.\n");
   for (i = 0; i < NUM_VALUES; ++i) {
      fwrite(&(userNums[i]), sizeof(int), 1, outFile);
   }

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

   return 0;
}
```

```
Enter 8 numbers...
0: 567
1: 342
2: 7
3: 1000000
4: 34965
5: 42
6: 1700
7: -25
Writing numbers to myfile.bin.
Closing file myfile.bin.
```

Try 11.7.1: Viewing a binary file in a text editor.

Run the above program and try opening the resulting myfile.bin file using a text editor. Notice that the contents of the file are not easily readable.

As the elements of an array are stored in sequential memory locations, the fwrite() command can also be used to directly copy the entire contents of an array to the output file by passing the number of elements within the array as third argument to fwrite(). The following programs illustrates.

Figure 11.7.2: Sample code for writing an array of integers to a binary file using a single fwrite() call.

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

int main(void) {
   const int NUM_VALUES = 8; // Num user numbers
   FILE* outFile = NULL;     // File pointer
   int userNums[NUM_VALUES]; // User numbers
   int i = 0;                // Loop index


   // Get numbers from user
   printf("\nEnter %d numbers...\n", NUM_VALUES);
   for (i = 0; i < NUM_VALUES; ++i) {
      printf("%d: ", i);
      scanf("%d", &(userNums[i]));
   }

   // Try to open the file
   outFile = fopen("myfile.bin", "wb");
   if( outFile == NULL ) {
      printf("Could not open file myfile.bin.\n");
      return -1; // -1 indicates error
   }

   // Write entire int array on N elements to output file
   fwrite(userNums, sizeof(int), NUM_VALUES, outFile);

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

   return 0;
}
```

```
Enter 8 numbers...
0: 567
1: 342
2: 7
3: 1000000
4: 34965
5: 42
6: 1700
7: -25
Writing numbers to myfile.bin.
Closing file myfile.bin.
```


A program can read from a binary file using the fread() function using the following form:

Construct 11.7.2: fread() function.

```
fread(variablePointer, sizeof(type), numElements, inputFile)
```

The first argument of fread() is a pointer to the variable in which the contents read from the input file will be copied. The second and third arguments specify the size in bytes of each element and the number of elements, respectively, being read from the file. The final argument to fread() is a FILE* for the input file being read. fread() will return the total number of bytes successfully read from the file.

The following programs uses fread() to read and print integer values from a binary input file until the end of the file is reached.

Figure 11.7.3: Sample code for reading and printing integers values read from a binary file using fread().

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

int main(void) {
   FILE* inFile = NULL;   // File pointer
   int fileNum = 0;       // Data value from file
   long numBytesRead = 0; // Num bytes read from file

   // Try to open the file
   printf("Opening file myfile.bin.Wn");

   inFile = fopen("myfile.bin", "rb");
   if( inFile == NULL ) {
      printf("Could not open file myfile.bin.Wn");
      return -1; // -1 indicates error
   }

   // Print read numbers to output
   printf("Reading and printing numbers.Wn");
   numBytesRead = fread(&fileNum, sizeof(int), 1, inFile);

   while (numBytesRead != 0) {
      printf("num: %dWn", fileNum);
      numBytesRead = fread(&fileNum, sizeof(int), 1, inFile);
   }

   // Done with file, so close it
   printf("Closing file myfile.bin.Wn");
   fclose(inFile);

   return 0;
}
```

```
Opening file myfile.bin.
Reading and printing numbers.
num: 567
num: 342
num: 7
num: 1000000
num: 34965
num: 42
num: 1700
num: -25
Closing file myfile.bin.
```


PARTICIPATION ACTIVITY 11.7.2: Opening file using open().

1) Write a statement to open the "file1.bin" for reading using binary mode and a FILE* variable named inputFile.

Check Show answer

2) Assuming an int is 32 bits and nums is an array of int, how many bytes will be written to the output file for the following call to fwrite(): `fwrite(nums, sizeof(int), 7, outputFile);`

Check Show answer

3) Write a single fread() statement to read three float values into an array named myVals using a FILE* variable named inputFile.

Check Show answer

Exploring further:

- stdio.h Reference Page from cplusplus.com
- fread() Reference Page from cplusplus.com
- fwrite() Reference Page from cplusplus.com
- C File IO Tutorial from cprogramming.com

11.8 Engineering examples using functions

This section contains examples of functions for various engineering calculations.

Gas equation

An equation used in physics and chemistry that relates pressure, volume, and temperature of a gas is PV = nRT. P is the pressure, V the volume, T the temperature, n the number of moles, and R a constant. The function below outputs the temperature of a gas given the other values.

Figure 11.8.1: PV = nRT. Compute the temperature of a gas.

```
#include <stdio.h>

const double GAS_CONSTANT = 8.3144621;  //  J / (mol * K)

/* Converts a pressure, volume, and number of moles
 of a gas into a temperature. */
double PVnToTemp(double gasPressure, double gasVolume, double numMoles) {
   return (gasPressure * gasVolume) / (numMoles * GAS_CONSTANT);
}

int main(void) {
   double gasPress = 0.0;  // User defined pressure
   double gasVol   = 0.0;  // User defined volume
   double gasMoles = 0.0;  // User defined moles

   // Prompt user for input parameters
   printf("WnEnter pressure (in Pascals): ");
   scanf("%lf", &gasPress);

   printf("Enter volume (in cubic meters): ");
   scanf("%lf", &gasVol);

   printf("Enter number of moles: ");
   scanf("%lf", &gasMoles);

   // Call function to calculate temperature
   printf("Temperature = %.2lf KWn",
          PVnToTemp(gasPress, gasVol, gasMoles));

   return 0;
}
```

```
Enter pressure (in Pascals): 2500
Enter volume (in cubic meters): 35.5
Enter number of moles: 18
Temperature = 593.01 K
```


PARTICIPATION ACTIVITY 11.8.1: PV = nRT calculation.

Questions refer to PVnToTemp() above.

1) PVnToTemp() uses a rewritten form of PV = nRT to solve for T, namely T = PV/nR.

   - ○ True
   - ○ False

2) PVnToTemp() uses a constant variable

for the gas constant R.

- True
- False

3) TempVolMolesToPressure() would likely return (temp * vlm) / (mols * GAS_CONSTANT).

- True
- False

Projectile location

Common physics equations determine the x and y coordinates of a projectile object at any time, given the object's initial velocity and angle at time 0 with initial position x = 0 and y = 0. The equation for x is v * t * cos(a). The equation for y is v * t * sin(a) - 0.5 * g * t * t. The following provides a single function to compute an object's position; because position consists of two values (x and y), the function uses pass by pointer parameters to return values for x and y. The program's main function asks the user for the object's initial velocity, angle, and height (y position), and then prints the object's position for every second until the object's y position is no longer greater than 0 (meaning the object fell back to earth).

Figure 11.8.2: Trajectory of object on Earth.

```
Launch angle (deg): 45
Launch velocity (m/s): 100
Initial height (m): 3
Time   1   x =   0   y =   3
Time   2   x =  71   y =  66
Time   3   x = 141   y = 122
Time   4   x = 212   y = 168
Time   5   x = 283   y = 204
Time   6   x = 354   y = 231
Time   7   x = 424   y = 248
Time   8   x = 495   y = 255
Time   9   x = 566   y = 252
Time  10   x = 636   y = 239
Time  11   x = 707   y = 217
Time  12   x = 778   y = 185
Time  13   x = 849   y = 143
Time  14   x = 919   y =  91
Time  15   x = 990   y =  30
```

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

// Note: 1-letter variable names are typically avoided,
// but used below where standard in physics.

const double PI_CONST = 3.14159265;

// Given time, angle, velocity, and gravity
// Update x and y values
void ObjectTrajectory(double t, double a, double v, double g,
                      double* x, double* y) {
   *x = v * t * cos(a);
   *y = v * t * sin(a) - 0.5 * g * t * t;
   return;
}

// Convert degree value to radians
double DegToRad(double inDeg) {
   return ((inDeg * PI_CONST) / 180.0);
}

int main(void) {
   const double GRAVITY  = 9.8; // Earth gravity (m/s^2)
   double launchAngle = 0.0;    // Angle of launch (rad)
   double launchVelocity = 0.0; // Velocity (m/s)
   double elapsedTime = 1.0;    // Time (s)

   double yLoc = -1.0;          // Object's height above ground (m)
   double xLoc = 0.0;           // Object's horiz. dist. from start (m)

   printf("Launch angle (deg): ");
   scanf("%lf", &launchAngle);
   launchAngle = DegToRad(launchAngle); // To radians

   printf("Launch velocity (m/s): ");
   scanf("%lf", &launchVelocity);

   printf("Initial height (m): ");
   scanf("%lf", &yLoc);

   while ( yLoc > 0.0 ) { // While above ground
      printf("Time %3.0f    x = %3.0lf    y = %3.0lfWn",
            elapsedTime, xLoc, yLoc);
      ObjectTrajectory(elapsedTime, launchAngle, launchVelocity,
                       GRAVITY, &xLoc, &yLoc);
      elapsedTime = elapsedTime + 1.0;
   }

   return 0;
}
```


Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

PARTICIPATION ACTIVITY 11.8.2: Projectile location.

Questions refer to ObjectTrajectory() above.

1) ObjectTrajectory() cannot return two values (for x and y), so instead takes x and y as modifiable parameters and changes their values.
   - True
   - False

2) ObjectTrajectory() could replace double types by int types without causing much change in computed values.
   - True
   - False

3) Each iteration of the loop will see yLoc increase.
   - True
   - False

4) Assuming the launch angle is less than 90 degrees, each iteration of the loop will see xLoc increase.
   - True
   - False

CHALLENGE ACTIVITY 11.8.1: Function to compute gas volume.

Define a function ComputeGasVolume that returns the volume of a gas given parameters pressure, temperature, and moles. Use the gas equation PV = nRT, where P is pressure in Pascals, V is volume in cubic meters, n is number of moles, R is the gas constant 8.3144621 (J / (mol*K)), and T is temperature in Kelvin.

```
1 #include <stdio.h>
2
3 const double GAS_CONST = 8.3144621;
4
5 /* Your solution goes here  */
6
```

```
int main(void) {
   double gasPressure = 0.0;
   double gasMoles = 0.0;
   double gasTemperature = 0.0;
   double gasVolume = 0.0;

   gasPressure = 100;
   gasMoles = 1 ;
   gasTemperature = 273;

   gasVolume = ComputeGasVolume(gasPressure, gasTemperature, gasMoles);
   printf("Gas volume: %lf m^3\n", gasVolume);

   return 0;
```

Run

11.9 Command-line arguments and files

The location of an input file or output file may not be known before writing a program. Instead, a program can use command-line arguments to allow the user to specify the location of an input file as shown in the following program. Assume two text files exist named "myfile1.txt" and "myfile2.txt" with the contents shown. The sample output shows the results when executing the program for each input file and for an input file that does not exist.

Figure 11.9.1: Using command-line arguments to specify the name of an input file.

myfile1.txt:

```
5
10
```

myfile2.txt:

```
-34
7
```

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

int main(int argc, char* argv[]) {
  FILE* inFile = NULL; // File pointer
  int fileNum1 = 0;    // Data value from file
  int fileNum2 = 0;    // Data value from file

  // Check number of arguments
  if( argc != 2 ) {
    printf("Usage: myprog.exe inputFileNameWn");W
    return 1; // 1 indicates error
  }

  // Try to open the file
  printf("Opening file %s.Wn", argv[1]);

  inFile = fopen(argv[1], "r");
  if( inFile == NULL ) {
    printf("Could not open file %s.Wn", argv[1]);
    return -1; // -1 indicates error
  }

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

  // Done with file, so close it
  printf("Closing file %s.WnWn", argv[1]);
  fclose(inFile);

  // Output values read from file
  printf("num1: %dWn", fileNum1);
  printf("num2: %dWn", fileNum2);
  printf("num1 + num2: %dWn", (fileNum1 + fileNum2));

  return 0;
}
```

```
> myprog.exe myfile1.txt
Opening file myfile.txt.
Reading two integers.
Closing file myfile.txt.

num1: 5
num2: 10
num1 + num2: 15

...

> myprog.exe myfile2.txt
Opening file myfile2.txt.
Reading two integers.
Closing file myfile2.txt.

num1: -34
num2: 7
num1 + num2: -27

...

> myprog.exe myfile3.txt
Opening file myfile3.txt.
Could not open file
myfile3.txt.
```


PARTICIPATION ACTIVITY 11.9.1: Filename command-line arguments.

1) A program "myprog" has two command-line arguments, one for an input file and a second for an output file. Type a command to run the program with input file "infile.txt" and output file "out".

Check Show answer

2) For a program run as `progname data.txt`, what is argv[1]? Don't use

quotes in your answer.

Check Show answer

11.10 Additional practice: Output art

The following is a sample programming lab activity; not all classes using a zyBook require students to fully complete this activity. No auto-checking is performed. Users planning to fully complete this program may consider first developing their code in a separate programming environment.

The following program prints a simple triangle.

PARTICIPATION ACTIVITY 11.10.1: Create ASCII art.

Load default template...

Pre-enter any input for program, the

Run

```
1
2 #include <stdio.h>
3
4 int main(void) {
5
6    printf("  *  \n");
7    printf(" *** \n");
8    printf("*****\n");
9
10   return 0;
11 }
12
```


Create different versions of the program:

1. Print a tree by adding a base under a 4-level triangle:

```
   *
  ***
```

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

2. Print the following "cat":

```
 ^   ^
 o o
 =   =
  ---
```


3. Allow a user to enter a number, and then print the original triangle using that number instead of asterisks, as in:

```
  9
 999
99999
```

Pictures made from keyboard characters are known as ***ASCII art***. ASCII art can be quite intricate, and fun to make and view. Wikipedia: ASCII art provides examples. Doing a web search for "ASCII art (someitem)" can find ASCII art versions of an item. For example, searching for "ASCII art cat" turns up thousands of examples of cats, most much more clever than the cat above.

11.11 Additional practice: Grade calculation

The following is a sample programming lab activity; not all classes using a zyBook require students to fully complete this activity. No auto-checking is performed. Users planning to fully complete this program may consider first developing their code in a separate programming environment.

PARTICIPATION ACTIVITY 11.11.1: Grade calculator.

The following incomplete program should compute a student's total course percentage based on scores on three items of different weights (%s):

- 20% Homeworks (out of 80 points)
- 30% Midterm exam (out of 40 points)
- 50% Final exam (out of 70 points)

Suggested (incremental) steps to finish the program:

1. First run it.
2. Next, complete the midterm exam calculation and run the program again. Use the constant variables where appropriate.
3. Then, complete the final exam calculation and run the program. Use the constant variables where appropriate.
4. Modify the program to include a quiz score out of 20 points. New weights: 10% homework, 15% quizzes, 30% midterm, 45% final. Run the program again.
5. To avoid having one large expression, introduce variables homeworkPart, quizPart, midtermPart, and finalPart. Compute each part first; each will be a number between 0 and 1. Then combine the parts using the weights into the course value. Run the program again.


```
1
2  #include <stdio.h>
3
4  int main(void) {
5     const double HOMEWORK_MAX = 80.0;
6     const double MIDTERM_MAX  = 40.0;
7     const double FINAL_MAX    = 70.0;
8     const double HOMEWORK_WEIGHT = 0.20; // 20%
9     const double MIDTERM_WEIGHT  = 0.30;
10    const double FINAL_WEIGHT    = 0.50;
11
12    double homeworkScore    = 0.0;
13    double midtermScore     = 0.0;
14    double finalScore       = 0.0;
15    double coursePercentage = 0.0;
16
17    printf("Enter homework score:\n");
18    scanf ("%lf", &homeworkScore);
19
20    printf("Enter midterm exam score:\n");
21    scanf ("%lf", &midtermScore);
```

78 36 62

Run

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15

11.12 Additional practice: Health data

The following is a sample programming lab activity; not all classes using a zyBook require students to fully complete this activity. No auto-checking is performed. Users planning to fully complete this program may consider first developing their code in a separate programming environment.

The following calculates a user's age in days based on the user's age in years.

PARTICIPATION ACTIVITY 11.12.1: Calculating user health data.

```
1
2  #include <stdio.h>
3
4  int main(void) {
5     int userAgeYears = 0;
6     int userAgeDays  = 0;
7
8     printf("Enter your age in years: \n");
9     scanf("%d", &userAgeYears);
10
11    userAgeDays = userAgeYears * 365;
12
13    printf("You are %d days old.\n", userAgeDays);
14
15    return 0;
16 }
17
```

```
19
```

Run

Create different versions of the program that:

1. Calculates the user's age in minutes and seconds.
2. Estimates the approximate number of times the user's heart has beat in his/her lifetime using an average heart rate of 72 beats per minutes.
3. Estimates the number of times the person has sneezed in his/her lifetime (research on the Internet to obtain a daily estimate).

4. Estimates the number of calories that the person has expended in his/her lifetime (research on the Internet to obtain a daily estimate). Also calculate the number of sandwiches (or other common food item) that equals that number of calories.
5. Be creative: Pick other health-related statistic. Try searching the Internet to determine how to calculate that data, and create a program to perform that calculation. The program can ask the user to enter any information needed to perform the calculation.

11.13 Additional practice: Tweet decoder

The following is a sample programming lab activity; not all classes using a zyBook require students to fully complete this activity. No auto-checking is performed. Users planning to fully complete this program may consider first developing their code in a separate programming environment.

The following program decodes a few common abbreviations in online communication as communications in Twitter ("tweets") or email, and provides the corresponding English phrase.

PARTICIPATION ACTIVITY 11.13.1: Tweet decoder.

Load default template...

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

int main(void){
   char origTweet[10] = "";

   printf("Enter abbreviation from tweet: \n");
   scanf("%s", origTweet);

   if (strcmp(origTweet, "LOL") == 0) {
      printf("LOL = laughing out loud\n");
   }
   else if (strcmp(origTweet, "BFN") == 0) {
      printf("BFN = bye for now\n");
   }
   else if (strcmp(origTweet, "FTW") == 0) {
      printf("FTW = for the win\n");
   }
   else if (strcmp(origTweet, "IRL") == 0) {
      printf("IRL = in real life\n");
```

LOL

Run

Create different versions of the program that:

1. Expands the number of abbreviations that can be decoded. Add support for abbreviations you commonly use or search the Internet to find a list of common abbreviations.
2. For abbreviations that do not match the supported abbreviations, check for common misspellings. Provide a suggestion for correct abbreviation along with the decoded meaning. For

example, if the user enters "LLO", your program can output "Did you mean LOL? LOL = laughing out loud".

11.14 Additional practice: Dice statistics

The following is a sample programming lab activity; not all classes using a zyBook require students to fully complete this activity. No auto-checking is performed. Users planning to fully complete this program may consider first developing their code in a separate programming environment.

Analyzing dice rolls is a common example in understanding probability and statistics. The following calculates the number of times the sum of two dice (randomly rolled) equals six or seven.

PARTICIPATION ACTIVITY 11.14.1: Dice rolls: Counting number of rolls that equals six or seven.

Load default template...

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <time.h>
5
6  int main(void){
7     int i = 0;          // Loop counter iterates numRolls tim
8     int numRolls = 0;   // User defined number of rolls
9     int numSixes = 0;   // Tracks number of 6s found
10    int numSevens = 0; // Tracks number of 7s found
11    int die1 = 0;       // Dice values
12    int die2 = 0;       // Dice values
13    int rollTotal = 0; // Sum of dice values
14
15    printf("Enter number of rolls: \n");
16    scanf("%d", &numRolls);
17    printf("%d",numRolls);
18    srand(time(0));
19
20    if (numRolls > 1) {
21
```

10

Run

Create different versions of the program that:

1. Calculates the number of times the sum of the randomly rolled dice equals each possible value from 2 to 12.
2. Repeatedly asks the user for the number of times to roll the dice, quitting only when the user-entered number is less than 1. Hint: Use a while loop that will execute as long as numRolls is greater than 1. Be sure to initialize numRolls correctly.
3. Prints a histogram in which the total number of times the dice rolls equals each possible value is displayed by printing a character like * that number of times, as shown below.

Figure 11.14.1: Histogram showing total number of dice rolls for each possible value.

```
Dice roll histogram:

2:  ******
3:  ****
4:  ***
5:  ********
6:  ********************
7:  ************
8:  ************
9:  **************
10: ***********
11: *****
12: ****
```


Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:15