6.1 User-defined function basics

A ***function*** is a named list of statements. Invoking a function's name, known as a ***function call***, causes the function's statements to execute. The following illustrates.

PARTICIPATION ACTIVITY 6.1.1: Function example: Printing a face.

Animation captions:

1. The function call jumps execution to the function's statements.
2. The return jumps execution back to the original call.

A ***function definition*** consists of the new function's name and a block of statements, as appeared above: void PrintFace() { ... }. The name can be any valid identifier. A ***block*** is a list of statements surrounded by braces.

The function call PrintFace() causes execution to jump to the function's statements. The function's ***return*** causes execution to jump back to the original calling location.

Other aspects of the function definition, like the () and the word void, are discussed later.

PARTICIPATION ACTIVITY 6.1.2: Function basics.

Given the PrintFace() function defined above and the following main() function:

```
int main(void) {
   PrintFace();
   PrintFace();
   return 0;
}
```

1) How many function calls to PrintFace() exist in main()?

Check Show answer

2) How many function definitions of PrintFace() exist *within* main()?

Check Show answer

3) How many output statements would execute in total?

Check Show answer

4) How many output statements exist in PrintFace()?

Check Show answer

5) Is main() itself a function definition? Answer yes or no.

Check Show answer

PARTICIPATION ACTIVITY 6.1.3: Adding to the face printing program.

1. Run the following program, observe the face output.
2. Modify main() to print that same face twice.
3. Complete the function definition of PrintFaceB() to print a different face of your choice, and then call that function from main() also.

Load default template... Run

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:12

```

#include <stdio.h>

void PrintFaceB(void) {
   // FIXME: FINISH
   return;
}

void PrintFaceA(void) {
   char faceChar = 'o';

   printf("  %c %c\n", faceChar, faceChar);           // E
   printf("   %c\n", faceChar);                       // N
   printf("  %c%c%c\n", faceChar, faceChar, faceChar); // M

   return;
}

int main(void) {
```

20

Exploring further:

- Functions tutorial from cprogramming.com

CHALLENGE ACTIVITY 6.1.1: Basic function call.

Complete the function definition to print five asterisks ***** when called once (do NOT print a newline). Output for sample program:

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


Run

CHALLENGE ACTIVITY 6.1.2: Basic function call.

Complete the PrintShape() function to print the following shape. End with newline. Example output:

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


Run

6.2 Parameters

Programmers can influence a function's behavior via an input to the function known as a ***parameter***. For example, a face-printing function might have an input that indicates the character to print when printing the face.

PARTICIPATION ACTIVITY 6.2.1: Function example: Printing a face.

Animation captions:

1. The function call jumps execution to the function's statements, passing character 'o' to the function's parameter faceChar.
2. The return jumps execution back to the original call.

The code `void PrintFace(char faceChar)` indicates that the function has a parameter of type char named faceChar.

The function call `PrintFace('o')` passes the value 'o' to that parameter. The value passed to a parameter is known as an ***argument***. An argument is an expression, such as 99, numCars, or numCars + 99.

In contrast to an argument being an expression, a parameter is like a variable declaration. Upon a call, the parameter's memory location is allocated, and the argument's value is assigned to the parameter. Upon a return, the parameter is deleted from memory.

PARTICIPATION ACTIVITY 6.2.2: Parameters.

1) Complete the function beginning to have a parameter named userAge of type int.

```
void PrintAge(          ) {
```

Check Show answer

2) Call a function named PrintAge, passing the value 21 as an argument.

Check Show answer

3) Is the following a valid function definition beginning? Type yes or no.
`void MyFct(int userNum + 5) { ... }`

Check Show answer

4) Assume a function `void PrintNum(int userNum)` simply prints the value of userNum without any space or new line. What will the following output?

```
PrintNum(43);
PrintNum(21);
```

Check Show answer

A function may have multiple parameters, which are separated by commas. Argument values are assigned to parameters by position: First argument to the first parameter, second to the second, etc.

A function definition with no parameters must still have the parentheses, as in: `void PrintSomething() { ... }`. Good practice is to use the keyword void for an empty parameter list, as in: `void PrintSomething(void) { ... }`. The call to such a function there must be parentheses, and they must be empty, as in: PrintSomething().

PARTICIPATION ACTIVITY 6.2.3: Multiple parameters.

1) Which correctly defines two integer parameters x and y for a function definition:
`void CalcVal(...)`?

- (int x; int y)
- (int x, y)
- (int x, int y)

2) Which correctly passes two integer arguments for the function call
`CalcVal(...)`?

- (99, 44 + 5)
- (int 99, 44)
- (int 99, int 44)

3) Given a function definition:
`void CalcVal(int a, int b, int c)`
what value is assigned to b during this function call:
`CalcVal(42, 55, 77);`

- Unknown
- 42
- 55

4) Given a function definition:
`void CalcVal(int a, int b, int c)`
and given int variables i, j, and k, which are valid arguments in the call
`CalcVal(...)`?

-

(i, j)

- (k, i + j, 99)
- (i + j + k)

PARTICIPATION ACTIVITY 6.2.4: Multiple parameters.

Modify PrintFace() to have three parameters: char eyeChar, char noseChar, char mouthChar. Call the function with arguments 'o', '*', and '#', which should draw this face:

```
 o o
  *
 ###
```

Load default template...

Run

```
#include <stdio.h>
void PrintFace(char faceChar) { // FIXME: Create 3 parameters
  printf("  %c %c\n", faceChar, faceChar);           // Eyes
  printf("   %c\n", faceChar);                       // Nose
  printf("  %c%c%c\n", faceChar, faceChar, faceChar); // Mouth
  return;
}
int main(void) {
  PrintFace('o'); // FIXME: Pass 3 arguments
  return 0;
}

```

PARTICIPATION ACTIVITY 6.2.5: Calls with multiple parameters.

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:12

Given:

```
void PrintSum(int num1, int num2) {
   printf("%d + %d is %d", num1, num2, num1 + num2);
   return;
}
```

1) What will be printed for the following function call?

`PrintSum(1, 2);`

Check Show answer

2) Write a function call using PrintSum() to print the sum of x and 400 (providing the arguments in that order). End with ;

Check Show answer

CHALLENGE ACTIVITY 6.2.1: Function parameters.

CHALLENGE ACTIVITY 6.2.2: Function call with parameter: Print tic-tac-toe board.

Complete the PrintTicTacToe function with char parameters horizChar and vertChar that prints a tic-tac-toe board with the characters as follows. End with newline. Ex: PrintTicTacToe('~', '!') prints:

```
x!x!x
~~~~~
x!x!x
~~~~~
x!x!x
```

Run

CHALLENGE ACTIVITY

6.2.3: Function call with parameter: Printing formatted measurement.

Define a function PrintFeetInchShort, with int parameters numFeet and numInches, that prints using ' and " shorthand. Ex: PrintFeetInchShort(5, 8) prints:

```
5' 8"
```

Hint: Use \" to print a double quote.

Run

6.3 Return

A function may return a value using a ***return statement***, as follows.

PARTICIPATION ACTIVITY 6.3.1: Function returns computed square.

Animation captions:

1. Call ComputeSquare and pass in the value 7.
2. Compute the square of numToSquare and return the result.
3. numSquared is assigned the return value of ComputeSquare(7).

The ComputeSquare function is defined to have a return type of int. So the function's return statement must also have an expression that evaluates to an int.

Other return types are allowed, such as char, double, etc. A function can only return one item, not two or more. A return type of ***void*** indicates that a function does not return any value, in which case the return statement should simply be: `return;`

A return statement may appear as any statement in a function, not just as the last statement. Also, multiple return statements may exist in a function.

PARTICIPATION ACTIVITY 6.3.2: Return.

Given:
`int CalculateSomeValue(int num1, int num2) { ... }`
Are the following appropriate return statements?

1) `return 9;`

- Yes
- No

2) `return 9 + 10;`

- Yes
- No

3) `return num1;`

- Yes
- No

4) `return (num1 + num2) + 1 ;`

- Yes
- No

5) `return;`

- Yes
- No

6) `return void;`

- Yes
- No

7) `return num1 num2;`

- Yes
- No

8) `return (0);`

- Yes
- No

9) Given: `void PrintSomething (int num1) { ... }`. Is `return 0;` a valid return statement?

- Yes
- No

A function evaluates to its returned value. Thus, a function call often appears within an expression. For example, 5 + ComputeSquare(4) would become 5 + 16, or 21. A function with a void return type cannot be used as such within an expression.

PARTICIPATION ACTIVITY 6.3.3: Calls in an expression.

Given:

```
double SquareRoot(double x) { ... }
void PrintVal(double x) { ... }
```

which of the following are valid statements?

1) `double y = SquareRoot(49.0);`

- True

False

2) `SquareRoot(49.0) = z;`

True

False

3) `double y = 1.0 + SquareRoot(144.0);`

True

False

4) `double y = SquareRoot(SquareRoot(16.0));`

True

False

5) `double y = SquareRoot;`

True

False

6) `double y = SquareRoot();`

True

False

7) `SquareRoot(9.0);`

True

False

8) `double y = PrintVal(9.0);`

True

False

9) `double y = 1 + PrintVal(9.0);`

True

False

10) `PrintVal(9.0);`

True

False

A function is commonly defined to compute a mathematical function involving several numerical parameters and returning a numerical result. For example, the following program uses a function to convert a person's height in U.S. units (feet and inches) into total centimeters.

Figure 6.3.1: Program with a function to convert height in feet/inches to centimeters.


```
#include <stdio.h>

/* Converts a height in feet/inches to centimeters */
double HeightFtInToCm(int heightFt, int heightIn) {
   const double CM_PER_IN = 2.54;
   const int    IN_PER_FT = 12;
   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: ");
   scanf("%d", &userFt);

   printf("Enter inches: ");
   scanf("%d", &userIn);

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

   return 0;
}
```

```
Enter feet: 5
Enter inches: 8
Centimeters: 172.720000
```

(Sidenotes: Most Americans only know their height in feet/inches, not in total inches or centimeters. Human average height is increasing, attributed to better nutrition (Source: Wikipedia: Human height)).

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:12

PARTICIPATION ACTIVITY 6.3.4: Temperature conversion.

Complete the program by writing and calling a function that converts a temperature from Celsius into Fahrenheit.

Load default template...

```
100
```

```

#include <stdio.h>

```

```
// FINISH  Define CelsiusToFahrenheit function here


int main(void) {
   double tempF = 0.0;
   double tempC = 0.0;

   printf("Enter temperature in Celsius:\n");
   scanf("%lf", &tempC);

   // FINISH

   printf("Fahrenheit: %lf\n", tempF);

   return 0;
}

```

Run

CHALLENGE ACTIVITY 6.3.1: Enter the output of the returned value.

A function's statements may include function calls, known as ***hierarchical function calls*** or ***nested function calls*** . Note that main() itself is a function, being the first function called when a program begins executing, and note that main() calls other functions in the earlier examples.

Exploring further:

- Function definition from msdn.microsoft.com
- Function call from msdn.microsoft.com

CHALLENGE ACTIVITY 6.3.2: Function call in expression.

Assign to maxSum the max of (numA, numB) PLUS the max of (numY, numZ). Use just one statement. Hint: Call FindMax() twice in an expression.

Run

CHALLENGE ACTIVITY 6.3.3: Function definition: Volume of a pyramid.

Define a function PyramidVolume with double parameters baseLength, baseWidth, and pyramidHeight, that returns as a double the volume of a pyramid with a rectangular base.
Relevant geometry equations:
Volume = base area x height x 1/3
Base area = base length x base width.
(Watch out for integer division).

Run

6.4 Reasons for defining functions

Several reasons exist for defining new functions in a program.

1: Improve program readability

A program's main() function can be easier to understand if it calls high-level functions, rather than being cluttered with computation details. The following program converts steps walked into distance walked and into calories burned, using two user-defined functions. Note how main() is easy to understand.

Figure 6.4.1: User-defined functions make main() easy to understand.

```
#include <stdio.h>

// Function converts steps to feet walked
int StepsToFeet(int baseSteps) {
   const int FEET_PER_STEP = 3;  // Unit conversion
   int feetTot = 0;              // Corresponding feet to steps

   feetTot = baseSteps * FEET_PER_STEP;

   return feetTot;
}

// Function converts steps to calories burned
double StepsToCalories(int baseSteps) {
   const double STEPS_PER_MINUTE = 70.0;              // Unit conversion
   const double CALORIES_PER_MINUTE_WALKING = 3.5;   // Unit conversion
   double minutesTot = 0.0;                           // Corresponding min to steps
   double caloriesTot = 0.0;                          // Corresponding calories to min

   minutesTot = baseSteps / STEPS_PER_MINUTE;
   caloriesTot = minutesTot * CALORIES_PER_MINUTE_WALKING;

   return caloriesTot;
}

int main(void) {
   int stepsInput = 0;         // User defined steps
   int feetTot = 0;            // Corresponding feet to steps
   double caloriesTot = 0.0; // Corresponding calories to steps

   // Prompt user for input
   printf("Enter number of steps walked: ");
   scanf("%d", &stepsInput);

   // Call functions to convert steps to feet/calories
   feetTot = StepsToFeet(stepsInput);
   printf("Feet: %d\n", feetTot);

   caloriesTot = StepsToCalories(stepsInput);
   printf("Calories: %lf\n", caloriesTot);

   return 0;
}
```

```
Enter number of steps walked: 1000
Feet: 3000
Calories: 50.000000
```


PARTICIPATION ACTIVITY 6.4.1: Improved readability.

1) A common reason for using functions is to create code that is easier to understand.

- True
- False

2: Modular program development

A function has precisely-defined input and output. As such, a programmer can focus on developing a particular function (or ***module***) of the program independently of other functions.

Programs are typically written using incremental development, meaning a small amount of code is written, compiled, and tested, then a small amount more (an incremental amount) is written, compiled, and tested, and so on. To assist with that process, programmers commonly introduce ***function stubs***, which are function definitions whose statements haven't been written yet. The benefit of a function stub is that the high-level behavior of main() can be captured before diving into details of each function, akin to planning the route of a roadtrip before starting to drive. The following illustrates.

Note that switch statements need not be understood to appreciate function stub usage.

Figure 6.4.2: Function stub used in incremental program development.

```
#include <stdio.h>

// Program calculates price of lumber. Hardwoods are sold
// by the board foot (measure of volume, 12"x12"x1").

// Function determines board foot based on lumber dimensions
double CalcBoardFoot(double boardHeight, double boardLength,
                     double boardThickness) {

   // board foot = (h * l * t)/144
   printf("FIXME: finish board foot calc\n");

   return 0;
}

// Function calculates price based on lumber type and quantity
double CalcLumberPrice(int lumberType, double boardFoot) {
   const double CHERRY_COST_BF = 6.75;  // Price of cherry per board foot
   const double MAPLE_COST_BF = 10.75;  // Price of maple per board foot
   const double WALNUT_COST_BF = 13.00; // Price of walnut per board foot
   double lumberCost = 0.0;             // Total lumber cost

   // Determine cost of lumber based on type
   switch (lumberType) {
      case 0:
         lumberCost = CHERRY_COST_BF;
         break;
      case 1:
         lumberCost = MAPLE_COST_BF;
         break;
      case 2:
         lumberCost = WALNUT_COST_BF;
         break;
      default:
         lumberCost = -1.0;
         break;
   }

   lumberCost = lumberCost * boardFoot;
   return lumberCost;
}

int main(void) {
   double heightDim = 0.0;  // Board height
   double lengthDim = 0.0;  // Board length
   double thickDim = 0.0;   // Board thickness
   int boardType = 0;       // Type of lumber
   double boardFoot = 0.0;  // Volume of lumber

   // Prompt user for input
   printf("Enter lumber height (in):");
   scanf("%lf", &heightDim);

   printf("Enter lumber length (in):");
   scanf("%lf", &lengthDim);

   printf("Enter lumber width (in):");
   scanf("%lf", &thickDim);

   printf("Enter lumber type (0: Cherry, 1: Maple, 2: Walnut):");
   scanf("%d", &boardType);

   // Call functions to calculate lumber cost
   boardFoot = CalcBoardFoot(heightDim, lengthDim,thickDim);
   printf("Cost of Lumber = $%lf\n", CalcLumberPrice(boardType, boardFoot));
```

The program can be compiled and executed, and the user can enter numbers, but then the above FIXME messages will be printed. Alternatively, the FIXME message could be in a comment. The programmer can later complete CalcBoardFoot().

PARTICIPATION ACTIVITY 6.4.2: Incremental development.

1) Incremental development may involve more frequent compilation, but ultimately lead to faster development of a program.

- True
- False

2) A key benefit of function stubs is faster running programs.

- True
- False

3) Modular development means to divide a program into separate modules that can be developed and tested separately and then integrated into a single program.

- True
- False

PARTICIPATION ACTIVITY 6.4.3: Function stubs.

Run the lumber cost calculator with the test values from the above example. Finish the incomplete function and test again.


```

#include <stdio.h>

// Program calculates price of lumber. Hardwoods are sold
// by the board foot (measure of volume, 12"x12"x1").

// Function determines board foot based on lumber dimensions
double CalcBoardFoot(double boardHeight, double boardLength,
                     double boardThickness) {

   // board foot = (h * l * t)/144
   printf("FIXME: finish board foot calc\n");

```

```
   return 0;
}

// Function calculates price based on lumber type and quantity
double CalcLumberPrice(int lumberType, double boardFoot) {
   const double CHERRY_COST_BF = 6.75;  // Price of cherry per board foot
   const double MAPLE_COST_BF = 10.75;  // Price of maple per board foot
```

```
30.6 10 2 0
```

Run

3: Avoid writing redundant code

A function can be defined once, then called from multiple places in a program, thus avoiding redundant code. Examples of such functions are math functions like pow() and abs() that prevent a programmer from having to write several lines of code each time he/she wants to compute a power or an absolute value.

Figure 6.4.3: Function call from multiple locations in main.

```
Enter first value: 2
Enter second value: 7
Total: 9

...

Enter first value: -1
Enter second value: 3
Total: 4

...

Enter first value: -2
Enter second value: -6
Total: 8
```

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

// Program calculates X = | Y | + | Z |

// Function returns the absolute value
int AbsValueConv(int origValue) {
   int absValue = 0;   // Resulting abs val

   if(origValue < 0){  // origVal is neg
      absValue = -1 * origValue;
   }
   else{               // origVal is pos
      absValue = origValue;
   }

   return absValue;
}

int main(void) {
   int userValue1 = 0;  // First user value
   int userValue2  = 0; // Second user value
   int sumValue = 0;    // Resulting value

   // Prompt user for inputs
   printf("Enter first value: ");
   scanf("%d", &userValue1);

   printf("Enter second value: ");
```


The skill of decomposing a program's behavior into a good set of functions is a fundamental part of programming that helps characterize a good programmer. Each function should have easily-recognizable behavior, and the behavior of main() (and any function that calls other functions) should be easily understandable via the sequence of function calls. As an analogy, the main behavior of "Starting a car" can be described as a sequence of function calls like "Buckle seat belt," "Adjust mirrors," "Place key in ignition," and "Turn key." Note that each function itself consists of more detailed operations, as in "Buckle seat belt" actually consisting of "Hold belt clip," "Pull belt clip across lap," and "Insert belt clip into belt buckle until hearing a click." "Buckle seat belt" is a good function definition because its meaning is clear to most people, whereas a coarser function definition like "GetReady" for both the seat belt and mirrors may not be as clear, while finer-grained functions like "Hold belt clip" are distracting from the purpose of the "Starting a car" function.

As general guidance (especially for programs written by beginner programmers), a function's statements should be viewable on a single computer screen or window, meaning a function usually shouldn't have more than about 30 lines of code. This is not a strict rule, but just guidance.

PARTICIPATION ACTIVITY 6.4.4: Reasons for defining functions.

1) A key reason for creating functions is to help main() run faster.

- True
- False

2) Avoiding redundancy means to avoid calling a function from multiple places in a program.

- True
- False

3) If a function's internal statements are revised, all function calls will have to be modified too.

- True
- False

4) A benefit of functions is to increase redundant code.

- True
- False

CHALLENGE ACTIVITY 6.4.1: Function stubs: Statistics.

Define stubs for the functions called by the below main(). Each stub should print "FIXME: Finish FunctionName()" followed by a newline, and should return -1. Example output:

```
FIXME: Finish GetUserNum()
FIXME: Finish GetUserNum()
FIXME: Finish ComputeAvg()
Avg: -1
```

Run

6.5 Functions with branches/loops

A function's block of statements may include branches, loops, and other statements. The following example uses a function to compute the amount that an online auction/sales website charges a customer who sells an item online.

Figure 6.5.1: Function example: Determining fees given an item selling price for an auction website.

```
Enter item selling price (e.g., 65.00): 9.95
eBay fee: $1.793500

...

Enter item selling price (e.g., 65.00): 40
eBay fee: $5.700000

...

Enter item selling price (e.g., 65.00): 100
eBay fee: $9.500000

...

Enter item selling price (e.g., 65.00): 500.15
eBay fee: $29.507500

...

Enter item selling price (e.g., 65.00): 2000
eBay fee: $74.500000
```

```c
#include <stdio.h>

/* Returns fee charged by ebay.com given the selling
 price of fixed-price books, movies, music, or video-games.
 Fee is $0.50 to list plus 13% of selling price up to $50.00,
 5% of amount from $50.01 to $1000.00, and
 2% for amount $1000.01 or more.
 Source: http://pages.ebay.com/help/sell/fees.html, 2012.

 Note: double variables are not normally used for dollars/cents
 due to the internal representation's precision, but are used
 here for simplicity.
 */

// Function determines eBay price given item selling price
double EbayFee(double sellPrice) {
   const double BASE_LIST_FEE      = 0.50; // Listing Fee
   const double PERC_50_OR_LESS    = 0.13; // % $50 or less
   const double PERC_50_TO_1000    = 0.05; // % $50.01..$1000.00
   const double PERC_1000_OR_MORE = 0.02; // % $1000.01 or more
   double feeTot = 0.0;                    // Resulting eBay fee

   feeTot = BASE_LIST_FEE;

   // Determine additional fee based on selling price
   if (sellPrice <= 50.00) { // $50.00 or lower
      feeTot = feeTot + (sellPrice * PERC_50_OR_LESS);
   }
   else if (sellPrice <= 1000.00) { // $50.01..$1000.00
      feeTot = feeTot + (50 * PERC_50_OR_LESS )
      + ((sellPrice - 50) * PERC_50_TO_1000);
   }
   else { // $1000.01 and higher
      feeTot = feeTot + (50 * PERC_50_OR_LESS)
      + ((1000 - 50) * PERC_50_TO_1000)
      + ((sellPrice - 1000) * PERC_1000_OR_MORE);
   }

   return feeTot;
}

int main(void) {
   double sellingPrice = 0.0;  // User defined selling price

   // Prompt user for selling price, call eBay fee function
   printf("Enter item selling price (e.g., 65.00): ");
   scanf("%lf", &sellingPrice);
   printf("eBay fee: $%lf\n", EbayFee(sellingPrice));
```

PARTICIPATION ACTIVITY 6.5.1: Analyzing the eBay fee calculator.

1) For any call to EbayFee() function, how many assignment statements for the variable feeTot will execute? Do not count variable initialization as an assignment.

Check Show answer

2) What does EbayFee() function return if its argument is 0.0 (show your answer in the form #.##)?

Check Show answer

3) What does EbayFee() function return if its argument is 100.00 (show your answer in the form #.##)?

Check Show answer

The following is another example with user-defined functions. The functions keep main()'s behavior readable and understandable.

Figure 6.5.2: User-defined functions make main() easy to understand.

```
Enter value for first input
Enter a positive number (>0):
13

Enter value for second input
Enter a positive number (>0):
7

Least common multiple of 13 and 7 is 91
```

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

// Function prompts user to enter postiive non-zero number
int GetPositiveNumber() {
   int userNum = 0;

   while (userNum <= 0) {
      printf("Enter a positive number (>0): ₩n");
      scanf("%d", &userNum);

      if (userNum <= 0) {
         printf("Invalid number.₩n");
      }
   }

   return userNum;
}

// Function returns greatest common divisor of two inputs
int FindGCD(int aVal, int bVal) {
   int numA = aVal;
   int numB = bVal;

   while (numA != numB) { // Euclid's algorithm
      if (numB > numA) {
         numB = numB - numA;
      }
      else {
         numA = numA - numB;
      }
   }

   return numA;
}

// Function returns least common multiple of two inputs
int FindLCM(int aVal, int bVal) {
   int lcmVal = 0;

   lcmVal = abs(aVal * bVal) / FindGCD(aVal, bVal);

   return lcmVal;
}

int main(void) {
   int usrNumA = 0;
   int usrNumB = 0;
   int lcmResult = 0;

   printf("Enter value for first input₩n");
   usrNumA = GetPositiveNumber();

   printf("₩nEnter value for second input₩n");
   usrNumB = GetPositiveNumber();

   lcmResult = FindLCM(usrNumA, usrNumB);

   printf("₩nLeast common multiple of %d and %d is %d₩n",
         usrNumA, usrNumB, lcmResult);
```


Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:12

PARTICIPATION ACTIVITY 6.5.2: Analyzing the least common multiple program.

1) Other than main(), which user-defined

function calls another user-defined function? Just write the function name.

Check Show answer

2) How many user-defined function calls exist in the program code?

Check Show answer

CHALLENGE ACTIVITY 6.5.1: Function with branch: Popcorn.

Complete function PrintPopcornTime(), with int parameter bagOunces, and void return type. If bagOunces is less than 3, print "Too small". If greater than 10, print "Too large". Otherwise, compute and print 6 * bagOunces followed by "seconds". End with a newline. Example output for ounces = 7:

```
42 seconds
```


Run

CHALLENGE ACTIVITY 6.5.2: Function with loop: Shampoo.

Write a function PrintShampooInstructions(), with int parameter numCycles, and void return type. If numCycles is less than 1, print "Too few.". If more than 4, print "Too many.". Else, print "N: Lather and rinse." numCycles times, where N is the cycle number, followed by "Done.". End with a newline. Example output for numCycles = 2:

```
1: Lather and rinse.
2: Lather and rinse.
Done.
```


Hint: Declare and use a loop variable.

Run

6.6 Unit testing (functions)

Testing is the process of checking whether a program behaves correctly. Testing a large program can be hard because bugs may appear anywhere in the program, and multiple bugs may interact. Good practice is to test small parts of the program individually, before testing the entire program, which can more readily support finding and fixing bugs. ***Unit testing*** is the process of individually testing a small part or unit of a program, typically a function. A unit test is typically conducted by creating a ***testbench***, a.k.a. test harness, which is a separate program whose sole purpose is to check that a function returns correct output values for a variety of input values. Each unique set of input values is known as a ***test vector***.

Consider a function HrMinToMin() that converts time specified in hours and minutes to total minutes. The figure below shows a test harness that tests that function. The harness supplies various input vectors like (0,0), (0,1), (0,99), (1,0), etc.

Figure 6.6.1: Test harness for the function HrMinToMin().

```
#include <stdio.h>

double HrMinToMin(int origHours, int origMinutes) {
   int totMinutes = 0;  // Resulting minutes

   totMinutes = (origHours * 60) + origMinutes;

   return origMinutes;
}

int main(void) {

   printf("Testing started₩n");

   printf("0:0, expecting 0, got: %lf₩n",    HrMinToMin(0,0) );
   printf("0:1, expecting 1, got: %lf₩n",    HrMinToMin(0,1) );
   printf("0:99, expecting 99, got: %lf₩n",  HrMinToMin(0,99));
   printf("1:0, expecting 60, got: %lf₩n",   HrMinToMin(1,0) );
   printf("5:0, expecting 300, got: %lf₩n",  HrMinToMin(5,0) );
   printf("2:30, expecting 150, got: %lf₩n", HrMinToMin(2,30));
   // Many more test vectors would be typical...

   printf("Testing completed₩n");

   return 0;
}
```

```
Testing started
0:0, expecting 0, got: 0.000000
0:1, expecting 1, got: 1.000000
0:99, expecting 99, got: 99.000000
1:0, expecting 60, got: 0.000000
5:0, expecting 300, got: 0.000000
2:30, expecting 150, got: 30.000000
Testing completed
```

Manually examining the program's printed output reveals that the function works for the first several vectors, but fails on the next several vectors, highlighted with colored background. Examining the output, one may note that the output minutes is the same as the input minutes; examining the code indeed leads to noticing that parameter origMinutes is being returned rather than variable totMinutes. Returning totMinutes and rerunning the test harness yields correct results.

Each bug a programmer encounters can improve a programmer by teaching him/her to program differently, just like getting hit a few times by an opening door teaches a person not to stand near a closed door.

PARTICIPATION ACTIVITY 6.6.1: Unit testing.

1) A test harness involves temporarily modifying an existing program to test a particular function within that program.

- True
- False

2) Unit testing means to modify function inputs in small steps known as units.

- True
- False

Manually examining a program's printed output is cumbersome and error prone. A better test harness would only print a message for incorrect output. PrintIf The language provides a compact way to print an error message when an expression evaluates to false. assert() is a macro (similar to a function) that prints an error message and exits the program if assert()'s input expression is false. The error message includes the current line number and the expression (a nifty trick enabled by using a macro rather than an actual function; details are beyond our scope). Using assert requires first including the assert library, part of the standard library, as shown below.

Figure 6.6.2: Test harness with assert for the function HrMinToMin().

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

double HrMinToMin(int origHours, int origMinutes) {
   int totMinutes = 0;  // Resulting minutes

   totMinutes = (origHours * 60) + origMinutes;

   return origMinutes;
}

int main(void) {

   printf("Testing started\n");

   assert(HrMinToMin(0, 0) == 0);
   assert(HrMinToMin(0, 1) == 1);
   assert(HrMinToMin(0, 99) == 99);
   assert(HrMinToMin(1, 0) == 60);
   assert(HrMinToMin(5, 0) == 300);
   assert(HrMinToMin(2, 30) == 150);
   // Many more test vectors would be typical...

   printf("Testing completed\n");

   return 0;
}
```



```
Testing started
Assertion failed: (HrMinToMin(1, 0) == 60), function main, file main.c, line 19.
```

assert() enables compact readable test harnesses, and also eases the task of examining the program's output for correctness; a program without detected errors would simply output "Testing started" followed by "Testing completed".

A programmer should choose test vectors that thoroughly exercise a function. Ideally the programmer would test all possible input values for a function, but such testing is simply not practical due to the large number of possibilities -- a function with one integer input has over 4 billion possible input values, for example. Good test vectors include a number of normal cases that represent a rich variety of typical input values. For a function with two integer inputs as above, variety might include mixing small and large numbers, having the first number large and the second small (and vice-versa), including some 0 values, etc. Good test vectors also include ***border cases*** that represent fringe scenarios. For example, border cases for the above function might include inputs 0 and 0, inputs 0 and a huge number like 9999999 (and vice-versa), two huge numbers, a negative number, two negative numbers, etc. The programmer tries to think of any extreme (or "weird") inputs that might cause the function to fail. For a simple function with a few integer inputs, a typical test harness might have dozens of test vectors. For brevity, the above examples had far fewer test vectors than typical.

PARTICIPATION ACTIVITY 6.6.2: Assertions and test cases.

1) Using assert() is a preferred way to test a function.
   - True
   - False

2) For function, border cases might include 0, a very large negative number, and a very large positive number.
   - True
   - False

3) For a function with three integer inputs, about 3-5 test vectors is likely sufficient for testing purposes.
   - True
   - False

4) A good programmer takes the time to test all possible input values for a function.
   - True
   - False

Exploring further:

- assert reference page from cplusplus.com

CHALLENGE ACTIVITY 6.6.1: Unit testing.

Add two more statements to main() to test inputs 3 and -1. Use print statements similar to the existing one (don't use assert).

Run

(*PrintIf) If you have studied branches, you may recognize that each print statement in main() could be replaced by an if statement like:

```
if ( HrMinToMin(0, 0) != 0 ) {
   printf("0:0, expecting 0, got: %lf", HrMinToMin(0, 0));
}
```

But the assert is more compact.

6.7 How functions work

Each function call creates a new set of local variables, forming part of what is known as a ***stack frame***. A return causes those local variables to be discarded.

PARTICIPATION

ACTIVITY 6.7.1: Function calls and returns.

Some knowledge of how a function call and return works at the assembly level can not only satisfy curiosity, but can also lead to fewer mistakes when parameter and return items become more complex. The following animation illustrates by showing, for a function named FindMax(), some sample high-level code, compiler-generated assembly instructions in memory, and data in memory during runtime. This animation presents advanced material intended to provide insight and appreciation for how a function call and return works.

The compiler generates instructions to copy arguments to parameter local variables, and to store a return address. A jump instruction jumps from main to the function's instructions. The function executes and stores results in a designated return value location. When the function completes, an instruction jumps back to the caller's location using the previously-stored return address. Then, an instruction copies the function's return value to the appropriate variable.

Press Compile to see how the compiler generates the machine instructions. Press Run to see how those instructions execute the function call.

PARTICIPATION ACTIVITY 6.7.2: How function call/return works.

PARTICIPATION ACTIVITY 6.7.3: How functions work.

1) After a function returns, its local variables keep their values, which serve as their initial values the next time the function is called.

- True
- False

2) A return address indicates the value returned by the function.

- True
- False

6.8 Functions: Common errors

A common error is to copy-and-paste code among functions but then not complete all necessary modifications to the pasted code. For example, a programmer might have developed and tested a function to convert a temperature value in Celsius to Fahrenheit, and then copied and modified the original function into a new function to convert Fahrenheit to Celsius as shown:

Figure 6.8.1: Copy-paste common error: Pasted code not properly modified. Find error on the right.

```
double Cel2Fah(double celVal) {
   double convTmp = 0.0;
   double fahVal = 0.0;

   convTmp = (9.0 / 5.0) * celVal;
   fahVal = convTmp + 32;

   return fahVal;
}
```

```
double Fah2Cel(double fahVal) {
   double convTmp = 0.0;
   double celVal = 0.0;

   convTmp = fahVal - 32;
   celVal = convTmp * (5.0 / 9.0);

   return fahVal;
}
```


The programmer forgot to change the return statement to return celVal rather than fahVal. Copying-and-pasting code is a common and useful time-saver, and can reduce errors by starting with known-correct code. Our advice is that when you copy-paste code, be extremely vigilant in making all necessary modifications. Just as the awareness that dark alleys or wet roads may be dangerous can cause you to vigilantly observe your surroundings or drive carefully, the awareness that copying-and-pasting is a common source of errors, may cause you to more vigilantly ensure you modify a pasted function correctly.

PARTICIPATION ACTIVITY 6.8.1: Copy-pasted sum-of-squares code.

Original parameters were num1, num2, num3. Original code was:

```
int sum = 0;

sum = (num1 * num1) + (num2 * num2) + (num3 * num3);

return sum;
```

New parameters are num1, num2, num3, num4. Find the error in the copy-pasted new code below.

1) int sum = 0;

sum = (num1 * num1) + (num2 * num2) + (num3 * num3) + (num3 * num4)

return sum;

Another common error is to return the wrong variable, such as typing `return convTmp;` instead of fahVal or celVal. The function will work and sometimes even return the correct value.

Failing to return a value for a function is another common error. If execution reaches the end of a function's statements, the function automatically returns. For a function with a void return type, such an automatic return poses no problem, although some programmers recommend including a return statement for clarity. But for a function defined to return a value, the returned value is undefined; the value could be anything. For example, the user-defined function below lacks a return statement:

Figure 6.8.2: Missing return statement common error: Program may sometimes work, leading to hard-to-find bug.

```
#include <stdio.h>

int StepsToFeet(int baseSteps) {
   const int FEET_PER_STEP = 3;  // Unit conversion
   int feetTot = 0;              // Corresponding feet to steps

   feetTot = baseSteps * FEET_PER_STEP;
}

int main(void) {
   int stepsInput = 0;       // User defined steps
   int feetTot = 0;          // Corresponding feet to steps

   // Prompt user for input
   printf("Enter number of steps walked: ");
   scanf("%d", &stepsInput);

   // Call functions to convert steps to feet/calories
   feetTot = StepsToFeet(stepsInput);
   printf("Feet: %d₩n", feetTot);

   return 0;
}
```

```
Enter number of steps walked: 1000
Feet: 3000
```

Sometimes a function with a missing return statement (or just `return;`) still returns the correct value. The reason is that the compiler uses a memory location to return a value to the calling expression. That location may have also been used by the compiler to store a local variable of that function. If that local variable happens to be the item that was supposed to be returned, the value in that location is the correct return value. But a later seemingly unrelated change to a function, like defining a new variable, may cause the compiler to use different memory locations, and the function suddenly no longer returns the correct value, leading to a bewildered programmer.

PARTICIPATION ACTIVITY 6.8.2: Common function errors.

Find the error in the function's code.

1)
```
int ComputeSumOfSquares(int num1, int num2) {
   int sum = 0;

   sum = (num1 * num1) + (num2 * num2);

   return;
}
```

2)
```
int ComputeEquation1(int num, int val, int k) {
   int sum = 0;

   sum = (num * val) + (k * val);

   return num;
}
```

PARTICIPATION ACTIVITY 6.8.3: Common function errors.

1) Forgetting to return a value from a function is a common error.
- True
- False

2) Copying-and-pasting code can lead to common errors if all necessary changes are not made to the pasted code.
- True
- False

3) Returning the incorrect variable from a function is a common error.

- True
- False

4) Is this function correct for squaring an integer?

```
int sqr(int a) {
   int t;
   t = a * a;
}
```

- Yes
- No

5) Is this function correct for squaring an integer?

```
int sqr(int a) {
   int t;
   t = a * a;
   return a;
}
```

- Yes
- No

CHALLENGE ACTIVITY 6.8.1: Function errors: Copying one function to create another.

Using the CelsiusToKelvin function as a guide, create a new function, changing the name to KelvinToCelsius, and modifying the function accordingly.

Run

6.9 Pass by pointer

Pass by pointer

New programmers sometimes assign a value to a parameter, believing the assignment updates the corresponding argument variable. An example situation is when a function should return two values, whereas a function's *return* construct can only return one value. Assigning a normal parameter fails to update the argument's variable, because normal parameters are ***pass by value***, meaning the argument's value is copied into a local variable for the parameter.

PARTICIPATION ACTIVITY 6.9.1: Assigning a normal pass by value parameter has no impact on the corresponding argument.

In contrast, defining a parameter as pointer enables updating of an argument variable. The calling function passes a pointer to a variable by prepending &, and each access in the function dereferences that pointer by prepending *, thus referring to the argument variable's memory location.

PARTICIPATION ACTIVITY 6.9.2: A pass by pointer parameter allows a function to update an argument variable.

Animation captions:

1. totTime is passed by value, so local copy. usrHr and usrMin are passed by pointer so the memory addresses are passed

2. hrVal and minVal point to usrHr and usrMin, so *hrVal and *minVal update usrHr and usrMin.
3. Return and print.

Defining a parameter as a pointer to enable updating the argument variable is commonly known as ***pass by reference***. However, to avoid confusion with pass by reference in C++, which is achieved quite differently, this material uses the term ***pass by pointer***. Pointers are introduced elsewhere; here, the programmer need only remember to pass the argument variable using &, and to define and access the parameter using *.

Pass by pointer parameters should be used sparingly. For the case of two return values, commonly a programmer should instead create two functions. For example, defining two separate functions `int StepsToFeet(int baseSteps)` and `int StepsToCalories(int totCalories)` is better than a single function `void StepsToFeetAndCalories(int baseSteps, int* baseFeet, int* totCalories)`. The separate functions support modular development, and enables use of the functions in an expression as in `if (StepsToFeet(mySteps) < 100)`.

Using multiple pass by pointer parameters makes sense when the output values are intertwined, such as computing monetary change, whose function might be `void ComputeChange(int totCents, int* numQuarters, int* numDimes, int* numNickels, int* r` , or converting from polar to Cartesian coordinates, whose function might be `void PolarToCartesian(int radialPol, int anglePol, int* xCar, int* yCar)`.

PARTICIPATION ACTIVITY 6.9.3: Calculating monetary change.

Complete the monetary change program. Use the fewest coins (i.e., using maximum larger coins first).

```

#include <stdio.h>

// FIXME: Add parameters for dimes, nickels, and pennies.
void ComputeChange(int totCents, int* numQuarters ) {

   printf("FIXME: Finish writing ComputeChange\n");

   *numQuarters = totCents / 25;

   return;
}

int main(void) {
   int userCents   = 0;
   int numQuarters = 0;
   // FIXME add variables for dimes, nickels, pennies


   printf("Enter total cents: \n");
   scanf("%d", &userCents);
```

83

Run

PARTICIPATION ACTIVITY 6.9.4: Function definition returns and arguments.

Choose the most appropriate function definition.

1) Convert inches into centimeters.

- `void InchToCM(int inches, int centimeters) ...`
- `int InchToCM(int inches) ...`
- More than one function should be written.

2) Compute the area and diameter of a circle given the radius.

- `void GetCircleAreaDiam(double radius, double* area, double* diameter) ...`
- `double GetCircleAreaDiam (double radius, double* area) ...`
- `double, double GetCircleAreaDiam(double radius) ...`
- More than one function should be written.

PARTICIPATION ACTIVITY 6.9.5: Function with pass by pointer.

Given the following ConvDigits() function,

```
void ConvDigits(int inNum, int* tensVal, int* onesVal) {
   *onesVal = inNum % 10;
   *tensVal = inNum / 10;
}
```

1) What is the value of the variable numTens, after the following function call?

```
int numTens;
int numOnes;

ConvDigits(45, &numTens, &numOnes);
```


Check Show answer

2) What is the value of the variable numOnes, after the following function call?

```
int numTens;
int numOnes;

ConvDigits(93, &numTens, &numOnes);
```

Check Show answer

3) Provide the code needed to obtain a pointer to a variable named `tensDigit`.

Check Show answer

4) Write a function call using ConvDigits() to store the tens place and ones place digits of the number 32 within the variables tensDigit and onesDigit. End with ;

Check Show answer

Avoid assigning pass by value parameters

Although a pass by value parameter creates a local copy, good practice is to avoid assigning such a parameter. The following code is correct but bad practice.

Figure 6.9.1: Programs should not assign pass by value parameters.

```
int IntMax(int numVal1, int numVal2) {
   if (numVal1 > numVal2) {
      numVal2 = numVal1; // numVal2 holds max
   }

   return numVal2;
}
```


Assigning a parameter can reduce code slightly, but is widely considered a lazy programming style. Assigning a parameter can mislead a reader into believing the argument variable is supposed to be updated. Assigning a parameter also increases likelihood of a bug caused by a statement reading the parameter later in the code but assuming the parameter's value is the original passed value.

PARTICIPATION ACTIVITY 6.9.6: Assigning a pass by value parameter.

1) Assigning a pass by value parameter in a function is discouraged due to potentially confusing a program reader into believing the argument is being updated.

   - True
   - False

2) Assigning a pass by value parameter in a function is discouraged due to potentially leading to a bug where a later line of code reads the parameter assuming the parameter still contains the original value.

   - True
   - False

3) Assigning a pass by value parameter can avoid having to declare a local variable.

   - True
   - False

Exploring further:

- Passing arguments by value and by reference from msdn.microsoft.com

CHALLENGE ACTIVITY 6.9.1: Function pass by pointer: Transforming coordinates.

Define a function CoordTransform() that transforms its first two input parameters xVal and yVal into two output parameters xValNew and yValNew. The function returns void. The transformation is new = (old + 1) * 2. Ex: If xVal = 3 and yVal = 4, then xValNew is 8 and yValNew is 10.

Run

6.10 Functions with array parameters

Functions commonly have array parameters. The following program uses a function to calculate the average value of the elements within an array of test scores.

Figure 6.10.1: Array parameters in an average test score calculation program.

```
#include <stdio.h>

double CalculateAverage(const double scoreVals[], int numVals) {
   int i = 0;
   double scoreSum = 0.0;

   for (i = 0; i < numVals; ++i) {
      scoreSum = scoreSum + scoreVals[i];
   }

   return scoreSum / numVals;
}

int main(void) {
   const int NUM_SCORES = 4;      // Array size
   double testScores[NUM_SCORES]; // User test scores
   int i = 0;
   double averageScore = 0.0;

   // Prompt user to enter test scores
   printf("Enter %d test scores:₩n", NUM_SCORES);
   for (i = 0; i < NUM_SCORES; ++i) {
      printf("Test score: ");
      scanf("%lf", &(testScores[i]));
   }
   printf("₩n");

   // Call function to calculate average
   averageScore = CalculateAverage(testScores, NUM_SCORES);
   printf("Average adjusted test score: ");
   printf("%lf₩n", averageScore);

   return 0;
}
```

```
Enter 4 test scores:
Test score: 90.5
Test score: 92.0
Test score: 87.5
Test score: 97.0

Average adjusted test score: 91.750000
```


The parameter definition (yellow highlighted) uses [] to indicate an array parameter. The function call's argument (orange highlighted) does not use []. The compiler automatically passes array arguments by passing a pointer to the array, rather than making a copy like for other parameter types, in part to avoid the inefficiency of copying large arrays.

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:12

Functions with array parameters, other than string parameters (discussed elsewhere), need a second parameter indicating the number of elements in the array. Ex: numVals indicates the number of elements within the scoresVals array.

PARTICIPATION ACTIVITY 6.10.1: Functions with array parameters.

1) Passing an array creates a copy of that array within the function.

- True
- False

2) An array is automatically passed by pointer.

- True
- False

3) In the GetAvgScore function, totalScores must be an explicit pointer parameter for an array.

```
void GetAvgScore(int* totalScores, int*
numScores) {
   // ...
}
```

- True
- False

PARTICIPATION ACTIVITY 6.10.2: Functions with arrays.

1) Complete the FindMin function definition to have an array parameter of type double named arrayVals and second parameter of type int numVals indicating the number of array elements.

```
double FindMin(
[          ]
) {
   // ...
}
```

Check Show answer

2) Complete the PrintVals function to print each array element on a separate line.

```
void PrintVals(int arrayVals[],
int numElements) {
   int i = 0;

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

Check Show answer

3) sellingPrices is an array of type double with 10 elements. Complete the statement to find the lowest price by calling the GetLowestPrice function defined below.

```
GetLowestPrice(double itemPrices[], int
numItems);

lowestPrice = GetLowestPrice(
                    );
```

Check Show answer

A programmer can explicitly define an array parameter as a pointer, but good practice is to use [] to define array parameters to clearly indicate the parameter is an array and not a pointer to a single variable. Ex: `void PrintVals(int arrVals[], int numVals)` is equivalent to `void PrintVals(int* arrVals, int numVals)`.

Because an array is passed to a function by passing a pointer to the array, a function can modify the elements of an array argument.

PARTICIPATION ACTIVITY 6.10.3: Functions can modify elements of an array argument.

Animation captions:

1. Arrays are automatically passed by pointer. So the address of testScores is passed to the scoreVals parameter.
2. A function can modify the elements of an array argument. Because scoreVals is a pointer to testScores, the function modifies testScores' elements.
3. Function adds 2.0 to all elements of the array argument.

The keyword ***const*** can be prepended to a function's array parameter to prevent the function from modifying the parameter. In the following program, the AdjustScores function modifies the array so defines a normal array parameter (highlighted yellow). The PrintScores and CalculateAverage functions do *not* modify the array so define a const array parameter (highlighted orange).

Figure 6.10.2: Normal and const array parameters in test score adjustment and averaging program.

```c
#include <stdio.h>

void AdjustScores(double scoreVals[], int numVals,
                  double scoreAdj) {
   int i = 0;

   for (i = 0; i < numVals; ++i) {
      scoreVals[i] = scoreVals[i] + scoreAdj;
   }

   return;
}

void PrintScores(const double scoreVals[], int numVals) {
   int i = 0;

   for (i = 0; i < numVals; ++i) {
      printf(" %lf", scoreVals[i]);
   }
   printf("\n");

   return;
}

double CalculateAverage(const double scoreVals[], int numVals) {
   int i = 0;
   double scoreSum = 0.0;

   for (i = 0; i < numVals; ++i) {
      scoreSum = scoreSum + scoreVals[i];
   }

   return scoreSum / numVals;
}

int main(void) {
   const int NUM_SCORES = 4;     // Array size
   double testScores[NUM_SCORES];  // User test scores
   int i = 0;
   double averageScore = 0.0;

   // Prompt user to enter test scores
   printf("Enter %d test scores:\n", NUM_SCORES);
   for (i = 0; i < NUM_SCORES; ++i) {
      printf("Test score: ");
      scanf("%lf", &(testScores[i]));
   }
   printf("\n");

   // Print original scores
   printf("Original test scores: ");
   PrintScores(testScores, NUM_SCORES);

   AdjustScores(testScores, NUM_SCORES, 2.0);

   printf("Adjusted test scores: ");
   PrintScores(testScores, NUM_SCORES);

   // Call function to calculate average
   averageScore = CalculateAverage(testScores, NUM_SCORES);
   printf("Average adjusted test score: ");
   printf("%lf\n", averageScore);

   return 0;
}
```


Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:12

```
Enter 4 test scores:
Test score: 90.0
Test score: 95.5
Test score: 97.0
Test score: 92.5

Original test scores:  90.000000 95.500000 97.000000 92.500000
Adjusted test scores:  92.000000 97.500000 99.000000 94.500000
Average adjusted test score: 95.750000
```


PARTICIPATION ACTIVITY 6.10.4: Const array function parameters.

Can the following functions modify the array parameter?

1) `void PrintArray(double userNums[], int numElements)`

- Yes
- No

2) `int GetHighScore(int userScores[], const int numHighScores)`

- Yes
- No

3) `int FindIndex(const int searchVals[], int numVal, int key)`

- Yes
- No

CHALLENGE ACTIVITY 6.10.1: Modify an array parameter.

Write a function SwapArrayEnds() that swaps the first and last elements of the function's array parameter. Ex: sortArray = {10, 20, 30, 40} becomes {40, 20, 30, 10}. The array's size may differ from 4.

Run

6.11 Functions with C string parameters

Functions commonly modify strings. The following function modifies a string by replacing spaces with hyphens.

Figure 6.11.1: Modifying a string parameter.

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

// Function replaces spaces with hyphens
void StrSpaceToHyphen(char modString[]) {
   int i = 0;  // Loop index

   for (i = 0; i < strlen(modString); ++i) {
      if (modString[i] == ' ') {
         modString[i] = '-';
      }
   }

   return;
}

int main(void) {
   const int INPUT_STR_SIZE = 50;  // Input string size
   char userStr[INPUT_STR_SIZE];   // Input string from user

   // Prompt user for input
   printf("Enter string with spaces: \n");
   fgets(userStr, INPUT_STR_SIZE, stdin);

   // Call function to modify user defined string
   StrSpaceToHyphen(userStr);

   printf("String with hyphens: %s\n", userStr);

   return 0;
}
```

```
Enter string with spaces:
Hello there everyone.
String with hyphens: Hello-there-everyone.

...

Enter string with spaces:
Good bye  now   !!!
String with hyphens: Good-bye--now---!!!
```

The parameter definition (yellow highlighted) uses [] to indicate an array parameter. The function call's argument (orange highlighted) does not use []. The compiler *automatically passes the string as a pointer*. Hence, the above function modifies the original string argument (userStr) and not a copy.

The strlen() function can be used to determine the length of the string argument passed to the function. So, unlike functions with array parameters of other types, a function with a string parameter does not require a second parameter to specify the string size.

PARTICIPATION ACTIVITY 6.11.1: Modifying a string parameter: Spaces to hyphens.

1. Run the program, noting correct output.
2. Modify the function to also replace each '!' by a '?'.

Load default template...

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

// Function replaces spaces with hyphens
void StrSpaceToHyphen(char modString[]) {
   int i = 0;  // Loop index

   for (i = 0; i < strlen(modString); ++i) {
      if (modString[i] == ' ') {
         modString[i] = '-';
      }
   }

   return;
}

int main(void) {
   const int INPUT_STR_SIZE = 50;  // Input string size
   char userStr[INPUT_STR_SIZE];   // Input string from use
```

Hello there everyone!!!

Run

Recall that a scanf() string argument did not require a prepended & as in `scanf("%s", userName);`, in contrast to integer or other argument types. The & is not needed because the compiler automatically passes an array as a pointer. The same is true for the userStr argument in fgets() above.

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:12

PARTICIPATION ACTIVITY 6.11.2: Functions with string parameters.

1) A string parameter defined as a char array must use [] after the parameter name.

- True
-

False

2) For a function with a string parameter, the function must include a second parameter for the string size.

- True
- False

3) To pass a string to a function, the argument must include [], as in `GetMovieRating(favMovie[])`.

- True
- False

A programmer can explicitly define an array parameter as a pointer. The following uses `char* modString` instead of the earlier `char modString[]`. Such pointer parameters are common for string parameters, such as in the string library functions.

Figure 6.11.2: Modifying a string using a pointer parameter.

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

// Function replaces spaces with hyphens
void StrSpaceToHyphen(char* modString) {
   int i = 0;  // Loop index

   for (i = 0; i < strlen(modString); ++i) {
      if (modString[i] == ' ') {
         modString[i] = '-';
      }
   }

   return;
}

int main(void) {
   const int INPUT_STR_SIZE = 50;  // Input string size
   char userStr[INPUT_STR_SIZE];   // Input string from user

   // Prompt user for input
   printf("Enter string with spaces: \n");
   fgets(userStr, INPUT_STR_SIZE, stdin);

   // Call function to modify user defined string
   StrSpaceToHyphen(userStr);

   printf("String with hyphens: %s\n", userStr);

   return 0;
}
```

```
Enter string with spaces:
Hello there everyone!
String with hyphens: Hello-there-everyone!

...

Enter string with spaces:
Good bye  now   !!!
String with hyphens: Good-bye--now---!!!
```

PARTICIPATION ACTIVITY 6.11.3: Functions with string parameters.

1) Passing a string to a function creates a copy of that string within the function.

- True
- False

2) A string is automatically passed by pointer.

- True
- False

CHALLENGE ACTIVITY 6.11.1: Use an existing function.

Use the function GetUserInfo() to get a user's information. If user enters 20 and Holly, sample program output is:

```
Holly is 20 years old.
```


Run

CHALLENGE

ACTIVITY 6.11.2: Modify a string parameter.

Complete the function to replace any period by an exclamation point. Ex: "Hello. I'm Miley. Nice to meet you." becomes:

```
"Hello! I'm Miley! Nice to meet you!"
```


Run

6.12 Functions with array parameters: Common er

For arrays other than C strings, a common error is defining a function with an array parameter without a second parameter indicating the number of array elements. Without a parameter indicating the number of array elements, the function will only work for one specific array size. A programmer may call the function with a larger or smaller array, which may lead to incorrect results.

PARTICIPATION ACTIVITY 6.12.1: Common error: Functions with arrays without a parameter indicating the number of array elements.

Animation captions:

1. Passing the 7 element array to the GetAvgScore will incorrectly calculate the average of the firs 5 elements, not all 7 elements. GetAvgScore does not use a second parameter specifying the number of array elements.
2. For an array with less than 5 elements, the function will access memory outside the array bounds.

PARTICIPATION ACTIVITY 6.12.2: Function with array missing parameter indicating number of array elements.

1) The GetMinScore() function has a common error of not including a parameter to indicate the number of array elements. Given the arrays preGames, seasonGames, and playoffGames, which function call does not return the minimum value?

```
int GetMinScore(int scoreVals[]) {
   int i = 0;
   int minScore = 0.0;

   minScore = scoreVals[0];
   for (i = 0; i < 5; ++i) {
      if (scoreVals[i] < minScore) {
         minScore = scoreVals[i];
      }
   }

   return minScore;
}

int preGames[]     = { 80, 45, 86, 85, 64,
99, 98 };
int seasonGames[]  = { 92, 90, 84, 82, 98 };
int playoffGames[] = { 85, 83, 75, 90, 70,
68, 69 };
```

- `GetMinScore(preGames);`
- `GetMinScore(seasonGames);`
- `GetMinScore(playoffGames);`

2) Which function definition correctly supports arrays of any size?

```
int GetMaxScoreA(int scoreVals[]) {
   int i = 0;
   int maxScore = 0.0;

   maxScore = scoreVals[0];
   for (i = 0; i < sizeof(scoreVals); ++i) {
      if (scoreVals[i] > maxScore) {
         maxScore = scoreVals[i];
      }
   }

   return maxScore;
}
```



```
int GetMaxScoreB(int scoreVals[], int
numScores) {
   int i = 0;
   int maxScore = 0.0;

   maxScore = scoreVals[0];
   for (i = 0; i < numScores; ++i) {
      if (scoreVals[i] < maxScore) {
         maxScore = scoreVals[i];
      }
   }

   return maxScore;
}
```

- GetMaxScoreA()
- GetMaxScoreB()

Another common error is modifying array elements in a function that should not modify the array. Arrays are automatically passed to functions using a pointer to the array. So, a function that modifies an array parameter will modify the array argument passed to the function, and not a local copy.

The FindMaxAbsValueIncorrect() function incorrectly modifies the inputVals array to find the array element with the largest absolute value.

Figure 6.12.1: FindMaxAbsValueIncorrect() incorrectly modifies inputVals array to find element with the largest absolute value.

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

float FindMaxAbsValueIncorrect(float inputVals[], int numVals) {
   int i = 0;
   float maxAbsVal = 0.0;

   // Incorrectly updates inputVals to calculate absolute value
   // of array elements
   for (i = 0; i < numVals; ++i) {
      inputVals[i] = fabs(inputVals[i]);
   }

   maxAbsVal = inputVals[0];
   for (i = 0; i < numVals; ++i) {
      if (inputVals[i] > maxAbsVal) {
         maxAbsVal = inputVals[i];
      }
   }

   return maxAbsVal;
}

int main(void) {
   const int NUM_VALUES = 5;
   // Array of changes in temperatures
   float tempChanges[5] = {10.0, 0.5, -5.1, -11.2, 3.0};
   float maxAbsChange = 0.0;
   int i = 0;

   // Print array before function call
   printf("tempChanges array before function call: ");
   for(i = 0; i < NUM_VALUES; ++i) {
      printf("%f ", tempChanges[i]);
   }
   printf("\n");

   // Find the largest temperature change, and print result.
   maxAbsChange = FindMaxAbsValueIncorrect(tempChanges, NUM_VALUES);
   printf("Max absolute temperature change: %f\n", maxAbsChange);

   // Print array before function call
   printf("tempChanges array after function call:  ");
   for(i = 0; i < NUM_VALUES; ++i) {
      printf("%f ", tempChanges[i]);
   }
   printf("\n");

   return 0;
}
```

```
tempChanges array before function call: 10.000000 0.500000 -5.100000 -11.200000 3.000000
Max absolute temperature change: 11.200000
tempChanges array after function call:  10.000000 0.500000 5.100000 11.200000 3.000000
```


Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:12

The FindMaxAbsValue() function performs the same computation but avoids modifying the inputVals array. The FindMaxAbsValue() also defines the inputVals as const, which prevents the function from modifying the parameter. For arrays that should not be modified in a function, good practice is to define the array parameter as const.

Figure 6.12.2: FindMaxAbsValue() computes the largest absolute value without modifying the array.

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

float FindMaxAbsValue(const float inputVals[], int numVals) {
   int i = 0;
   float maxAbsVal = 0.0;
   float inputAbsVal = 0.0;

   maxAbsVal = fabs(inputVals[0]);
   for (i = 0; i < numVals; ++i) {
      inputAbsVal = fabs(inputVals[i]);
      if (inputAbsVal > maxAbsVal) {
         maxAbsVal = inputAbsVal;
      }
   }

   return maxAbsVal;
}

int main(void) {
   const int NUM_VALUES = 5;
   // Array of changes in temperatures
   float tempChanges[5] = {10.0, 0.5, -5.1, -11.2, 3.0};
   float maxAbsChange = 0.0;
   int i = 0;

   // Print array before function call
   printf("tempChanges array before function call: ");
   for(i = 0; i < NUM_VALUES; ++i) {
      printf("%f ", tempChanges[i]);
   }
   printf("\n");

   // Find the largest temperature change, and print result.
   maxAbsChange = FindMaxAbsValue(tempChanges, NUM_VALUES);
   printf("Max absolute temperature change: %f\n", maxAbsChange);

   // Print array before function call
   printf("tempChanges array after function call:  ");
   for(i = 0; i < NUM_VALUES; ++i) {
      printf("%f ", tempChanges[i]);
   }
   printf("\n");

   return 0;
}
```

```
tempChanges array before function call: 10.000000 0.500000 -5.100000 -11.200000 3.000000
Max absolute temperature change: 11.200000
tempChanges array after function call:  10.000000 0.500000 -5.100000 -11.200000 3.000000
```

PARTICIPATION ACTIVITY 6.12.3: Functions array parameters.

Based on the function names, should the following functions allow the array parameters to be modified?

1) `void PrintReverse(int inputVals[], int numVals)`
   - Yes
   - No

2) `void SortArrayAscending(int inputVals[], int numVals)`
   - Yes
   - No

3) `int FindIndexSmallest(int inputVals[], int numVals)`
   - Yes
   - No

6.13 Scope of variable/function definitions

The name of a defined variable or function item is only visible to part of a program, known as the item's ***scope***. A variable declared in a function has scope limited to inside that function. In fact, because a compiler scans a program line-by-line from top-to-bottom, the scope starts *after* the declaration until the function's end. The following highlights the scope of local variable cmVal.

Figure 6.13.1: Local variable scope.

```c
#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: ");
   scanf("%d", &userFt);

   printf("Enter inches: ");
   scanf("%d", &userIn);

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

   return 0;
}
```


Note that variable cmVal is invisible to the function main(). A statement in main() like `newLen = cmVal;` would yield a compiler error, e.g., the "error: cmVal was not declared in this scope". Likewise, variables userFt and userIn are invisible to the functionHeightFtInToCm(). Thus, a programmer is free to define items with names userFt or userIn in function HeightFtInToCm.

A variable declared outside any function is called a ***global variable***, in contrast to a *local variable* declared inside a function. A global variable's scope extends after the declaration to the file's end, and reaches into functions. For example, HeightFtInToCm() above accesses global variables CM_PER_IN and IN_PER_FT.

Global variables should be used sparingly. If a function's local variable (including a parameter) has the same name as a global variable, then in that function the name refers to the local item and the global is inaccessible. Such naming can confuse a reader. Furthermore, if a function updates a global variable, the function has effects that go beyond its parameters and return value, known as ***side effects***, which make program maintenance hard. Global variables are typically limited to const variables like the number of centimeters per inch above. Beginning programmers sometimes use globals to avoid having to use parameters, which is bad practice. Good practice is to minimize the use of non-const global variables.

PARTICIPATION ACTIVITY 6.13.1: Variable/function scope.

1) A local variable is declared inside a function, while a global is declared outside any function.
   - True
   - False

2) A local variable's scope extends from a function's opening brace to the function's closing brace.
   - True
   - False

3) If a function's local variable has the same name as a function parameter, the name will refer to the local variable.
   - True
   - False

4) If a function's local variable has the same name as a global variable, the name will refer to the local variable.
   - True
   - False

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:12

5) A function that changes the value of a global variable is sometimes said to have "side effects".
   - True
   - False

A function also has scope, which extends from its definition to the end of the file. Commonly, a programmer wishes to have the main() definition appear near the top of a file, with other functions definitions appearing further below, so that the main function is the first thing a reader sees. However, given function scope, main() would not be able to call any of those other functions. A solution involves function declarations. A ***function declaration*** specifies the function's return type, name, and parameters, ending with a semicolon where the opening brace would have gone. A function declaration is also known as a ***function prototype*** The function declaration gives the compiler enough information to recognize valid calls to the function. So by placing function declarations at the top of a file, the main function can then appear next, with actual function definitions appearing later in the file.

Figure 6.13.2: A function declaration allows a function definition to appear later in a file.

```
#include <stdio.h>
#include <math.h> // To use "pow" function

/* Program to convert given-year U.S. dollars to
 current dollars, using simplistic method of 4% annual inflation.
 Source: http://inflationdata.com (See: Historical) */

// (Function DECLARATION)
double ToCurrDollars (double pastDol, int pastYr, int currYr);

int main(void) {
   double pastDol = 0.0;  // Starting dollar amount
   double currDol = 0.0;  // Ending dollar amount (converted value)
   int pastYr = 0;        // Starting year
   int currYr = 0;        // Ending year (converted to year)

   // Prompt user for previous year/dollar and current year
   printf("Enter current year: ");
   scanf("%d", &currYr);
   printf("Enter past year: ");
   scanf("%d", &pastYr);
   printf("Enter past dollars (Ex: 1000): ");
   scanf("%lf", &pastDol);

   // Function call to convert past to current dollars
   currDol = ToCurrDollars(pastDol, pastYr, currYr);

   printf("$%lf in %d is about $%lf in %d\n",
          pastDol, pastYr, currDol, currYr);

   return 0;
}

// (Function DEFINITION)
// Functin returns equivalent value of pastDol in pastYr to currYr
double ToCurrDollars (double pastDol, int pastYr, int currYr) {
   double currDol = 0.0;  // Equivalent dollar amount given inflation

   currDol = pastDol * pow(1.04, currYr - pastYr );

   return currDol;
}
```

```
Enter current year: 2015
Enter past year: 1970
Enter past dollars (Ex: 1000): 10000
$10000 in 1970 is about $58411.756815 in 2015
(average annual U.S. income in 1970)

...

Enter current year: 2015
Enter past year: 1970
Enter past dollars (Ex: 1000): 23000
$23000 in 1970 is about $134347.040674 in 2015
(average U.S. house price in 1970)

...

Enter current year: 2015
Enter past year: 1933
Enter past dollars (Ex: 1000): 37
$37 in 1933 is about $922.434519 in 2015
(cost of Golden Gate Bridge, in millions)

...

Enter current year: 2015
Enter past year: 1969
Enter past dollars (Ex: 1000): 25
$25 in 1969 is about $151.870568 in 2015
(cost of Apollo space program, in billions)
```


A common error is for the function definition to not match the function declaration, such as a parameter defined as double in the declaration but as int in the definition, or with a slightly different identifier. The compiler detects such errors.

PARTICIPATION ACTIVITY 6.13.2: Function declaration and definition.

1) A function declaration lists the contents of a function, while a function definition just specifies the function's interface.

- True
- False

2) A function declaration enables calls to the function before the function definition.

- True
- False

Exploring further:

- More on Scope from msdn.microsoft.com

6.14 Preprocessor and include

The ***preprocessor*** is a tool that scans the file from top to bottom looking for any lines that begin with #, known as a ***hash symbol***. Each such line is not a program statement, but rather directs the preprocessor to modify the file in some way before compilation continues, each such line being known as a ***preprocessor directive***. The directive ends at the end of the line, no semicolon is used at the end of the line.

Perhaps the most commonly-used preprocessor directive is ***#include***, known as an ***include directive***. #include directs the compiler to replace that line by the contents of the given filename.

Construct 6.14.1: Include directives.

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

The following animation illustrates.

PARTICIPATION ACTIVITY 6.14.1: Preprocessor's handling of an include directive.

Good practice is to use a .h suffix for any file that will be included in another file. The h is short for header, to indicate that the file is intended to be included at the top (or header) of other files. Although any file can be included in any other file, convention is to only include .h files.

The characters surrounding the filename determine where the preprocessor looks for the file.

- `#include "myfile.h"` -- A filename in quotes causes the preprocessor to look for the file in the same folder/directory as the including file.
- `#include <stdfile.h>` -- A filename in angle brackets causes the preprocessor to look in the system's standard library folder/directory. Programmers typically use angle brackets only for standard library files, using quotes for all other include files. Note that nearly every previous example has included at least one standard library file, using angle brackets.

PARTICIPATION ACTIVITY 6.14.2: Include directives.

1) The preprocessor processes any line beginning with what symbol?
   - #
   - <filename>
   - "filename"

2) After a source file is processed by the preprocessor, is it correct to say that all hash symbols will be removed from the code remaining to be compiled?
   - yes
   - no

3) Do header files have to end in .h?
   - yes
   - no

4) Where does the preprocessor look for myfile.h in the line:
   `#include "myfile.h"`
   - Current folder
   - System folder
   - Unknown

5) What one symbol is incorrect in the following:
   `#include <stdlib.h>;`
   - #
   - <>
   - ;

Exploring further:

- Preprocessor tutorial on cplusplus.com
- Preprocessor directives on MSDN

6.15 Separate files

Separating part of a program's code into a separate file can yield several benefits. One benefit is preventing a main file from becoming unmanageably large. Another benefit is that the separated part could be useful in other programs.

Suppose a program has several related functions that operate on triples of numbers, such as computing the maximum of three numbers or computing the average of three numbers. Those related functions' definitions can be placed in their own file as shown below in the file threeintsfcts.c.

Figure 6.15.1: Putting related functions in their own file.

main.c

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

// Normally lots of other code here

int main(void) {

   printf("%d\n", ThreeIntsSum(5, 10, 20));
   printf("%d\n", ThreeIntsAvg(5, 10, 20));

   return 0;
}

// Normally lots of other code here
```

threeintsfcts.c

```
int ThreeIntsSum(int num1, int num2, int num3) {
   return (num1 + num2 + num3);
}

int ThreeIntsAvg(int num1, int num2, int num3) {
   int sum = 0;
   sum = num1 + num2 + num3;
   return (sum / 3);
}
```

threeintsfcts.h

```
int ThreeIntsSum(int num1, int num2, int num3);
int ThreeIntsAvg(int num1, int num2, int num3);
```

```
> a.out
35
11
>
```


One could then compile the main.c and threeintsfcts.c files together as shown below.

Figure 6.15.2: Compiling multiple files together.

Without `#include "threeintsfcts.h"` in main.c	With `#include "threeintsfcts.h"` in main.c

<pre>> gcc -Wall main.c threeintsfcts.c
main.c: In function 'main':
main.c:8: warning: implicit declaration of function 'ThreeIntsSum'
main.c:9: warning: implicit declaration of function 'ThreeIntsAvg'</pre>
	<pre>> gcc -Wall main.c threeintsfcts.c
	></pre>

Just compiling those two files (without the `#include "threeintsfcts.h"` line in the main file) would yield an error, as shown above on the left. The problem is that the compiler does not see the function definitions while processing the main file because those definitions are in another file, which is similar to what occurs when defining functions after main(). The solution for both situations is to provide function declarations before main() so the compiler knows enough about the functions to compile calls to those functions. Instead of typing the declarations directly above main(), a programmer can provide the function declarations in a header file, such as the threeintsfcts.h file provided in the figure above. The programmer then includes the contents of that file into a source file via the line: `#include "threeintsfcts.h"`.

The reader may note that the .h file could have contained function definitions rather than just function declarations, eliminating the need for two files (one for declarations, one for definitions). However, the two file approach has two key advantages. One advantage is that with the two file approach, the .h file serves as a brief summary of all functions available. A second advantage is that the main file's copy does not become exceedingly large during compilation, which can lead to slow compilation.

One last consideration that must be dealt with is that a header file could get included multiple times, causing the compiler to generate errors indicating an item defined in that header file is defined multiple times (the above header files only declared functions and didn't define them, but other header files may define functions, types, constants, and other items). Multiple inclusion commonly can occur when one header file includes another header file, e.g., the main file includes file1.h and file2.h, and file1.h also includes file2.h -- thus, file2.h would get included twice into the main file.

The solution is to add some additional preprocessor directives, known as header file guards, to the .h file as follows.

Construct 6.15.1: Header file guards.

```
#ifndef FILENAME_H
#define FILENAME_H

// Header file contents

#endif
```

Header file guards are preprocessor directives cause the compiler to only include the contents of the header file once. `#define FILENAME_H` defines the symbol FILENAME_H to the preprocessor. The `#ifndef FILENAME_H` and `#endif` form a pair that instructs the preprocessor to process the code between the pair only if FILENAME_H is not defined ("ifndef" is short for "if not defined"). Thus, if the preprocessor includes encounter the header more than once, the code in the file during the second and any subsequent encounters will be skipped because FILENAME_H was already defined.

Good practice is to guard every header file. The following shows the threeintsfcts.h file with the guarding code added.

Figure 6.15.3: All header files should be guarded.

```
#ifndef THREEINTSFCT_H
#define THREEINTSFCT_H

int ThreeIntsSum(int num1, int num2, int num3);
int ThreeIntsAvg(int num1, int num2, int num3);

#endif
```

PARTICIPATION ACTIVITY 6.15.1: The earth.

1) Header files must end with .h.
   - True
   - False

2) Header files should contain function definitions for functions declared in another file.
   - True
   - False

3) Guarding a header file prevents multiple inclusion of that file by the preprocessor.
   - True
   - False

4) Is the following the correct two-line sequence to guard a file named

myfile.h?

```
#ifdef MYFILE_H
#define MYFILE_H
```

- True
- False

Exploring further:

- Preprocessor tutorial on cplusplus.com
- Preprocessor directives on MSDN

6.16 C example: Salary calculation with functions

PARTICIPATION ACTIVITY 6.16.1: Calculate salary: Using functions.

Separating calculations into functions simplifies modifying and expanding programs.

The following program calculates the tax rate and tax to pay, using functions. One function returns a tax rate based on an annual salary.

1. Run the program below with annual salaries of 40000, 60000, and 0.
2. Change the program to use a functions to input the annual salary.
3. Run the program again with the same annual salaries as above. Are results the same?

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:12

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

double GetCorrespondingTableValue(int search, int baseTable[], double valueTable[], int tableSize) {
   double value = 0.0;
   int i = 0;
   bool keepLooking = true;

   while ((i < tableSize) && keepLooking) {
      if (search <= baseTable[i]) {
         value = valueTable[i];
         keepLooking = false;
      }
      else {
         ++i;
      }
   }
```

```
   return value;
}
```

40000 60000 0

Run

A solution to the above problem follows. The program was altered slightly to allow a zero annual salary and to end when a user enters a negative number for an annual salary.

PARTICIPATION ACTIVITY 6.16.2: Calculate salary: Using functions (solution).

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

// Function to prompt for and input an integer
int PromptForInteger(const char userPrompt[]) {
   int inputValue = 0;

   printf("%s: \n", userPrompt);
   scanf("%d", &inputValue);

   return inputValue;
}

// **************************************************************************

// Function to get a value from one table based on a range in the other table
double GetCorrespondingTableValue(int search, int baseTable[], double valueTable[], int tableSize)
   double value = 0.0;
   int i = 0;
   bool keepLooking = true;

```

60000 40000 1000000
-1

Run

6.17 C example: Domain name validation with func

PARTICIPATION ACTIVITY 6.17.1: Validate domain names with functions.

Functions facilitate breaking down a large problem into a collection of smaller ones.

A ***top-level domain*** (TLD) name is the last part of an Internet domain name like .com in example.com. A ***core generic top-level domain*** (core gTLD) is a TLD that is either .com, .net, .org, or .info. A ***restricted top-level domain*** is a TLD that is either .biz, .name, or .pro. A ***second-level domain*** is a single name that precedes a TLD as in apple in apple.com

The following program repeatedly prompts for a domain name and indicates whether that domain name is valid and has a core gTLD. For this program, a valid domain name has a second-level domain followed by a TLD, and the second-level domain has these three characteristics:

1. Is 1-63 characters in length.
2. Contains only uppercase and lowercase letters or a dash.
3. Does not begin or end with a dash.

For this program, a valid domain name must contain only one period, such as apple.com, but not support.apple.com. The program ends when the user presses just the Enter key in response to a prompt.

1. Run the program. Note that a restricted gTLD is not recognized as such.
2. Change the program by writing an input function and adding the validation for a restricted gTLD. Run the program again.

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:12

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

// Global variables used for array lengths
const int MAX_NUMS = 4;
const int MAX_SIZE = 6;

// ***********************************************************************

// Returns the position of a single period in a string
int GetPeriodPosition(char stringToSearch[]) {
   int stringLength   = 0;
   int periodCounter  = 0;
   int periodPosition = -1;
```

```
   int i = 0;

   stringLength = strlen(stringToSearch);

```

apple.com
APPLE.com
apple.comm

Run

PARTICIPATION ACTIVITY 6.17.2: Validate domain names with functions.

A solution to the above problem follows.

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

// Global variables used for array lengths
const int MAX_NUMS = 4;
const int MAX_SIZE = 6;

// **********************************************************************

// Returns the position of a single period in a string
int GetPeriodPosition(char stringToSearch[]) {
   int stringLength   = 0;
   int periodCounter  = 0;
   int periodPosition = -1;
   int i = 0;

   stringLength = strlen(stringToSearch);

```


apple.com
APPLE.com
apple.comm

Run