6.1 User-defined function basics

Functions (general)

A program may perform the same operation repeatedly, causing a large and confusing program due to redundancy. Program redundancy can be reduced by creating a grouping of predefined statements for repeatedly used operations, known as a ***function***. Even without redundancy, functions can prevent a main program from becoming large and confusing.

PARTICIPATION ACTIVITY 6.1.1: Functions can reduce redundancy and keep the main program simple.

Animation content:

`undefined`

Animation captions:

1. Commonly, a program performs the same operation, such as a calculation, in multiple places. Here, the Fahrenheit to Celsius calculation is done in three places.
2. Repeated operations clutter the main program. And such repeated operations are more prone to errors.
3. A better approach defines the Fahrenheit to Celsius calculation once, named F2C here. Then, F2C can be "called" three times, yielding a simpler main program.
4. The impact is even greater when the operation has multiple statements -- here 3 statements, but commonly tens of statements. The main program is much simpler.
5. Even without repeated operations, calling predefined operations keeps the main program simple and intuitive.

PARTICIPATION ACTIVITY 6.1.2: Reasons for functions.

Consider the animation above.

1) In the original main program, the Fahrenheit to Celsius calculation appeared how many times?
   - ○ 1
   - ○ 3
2) Along with yielding a simpler main

program, using the predefined Fahrenheit to Celsius calculation prevented what error in the original program?

- Adding rather than subtracting 32.0
- Multiplying by 9.0 / 5.0 rather than by 5.0 / 9.0

3) In the last example above, the main program was simplified by _____.

- eliminating redundant code for operation XYZ
- predefining operations for XYZ and PQR

Basics of functions

A ***function*** is a named list of statements.

- A ***function definition*** consists of the new function's name and a block of statements. Ex: `void PrintPizzaArea() { /* block of statements */ }`
- A ***function call*** is an invocation of a function's name, causing the function's statements to execute.

The function's name can be any valid identifier. A ***block*** is a list of statements surrounded by braces.

Below, the function call PrintPizzaArea() causes execution to jump to the function's statements. Execution returns to the original location after executing the function's last statement.

PARTICIPATION ACTIVITY 6.1.3: Function example: Printing a pizza area.

Animation captions:

1. The function call jumps execution to the function's statements.
2. After the last statement of the PrintPizzaArea() function, execution returns to the original location.

PARTICIPATION ACTIVITY 6.1.4: Function basics.

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

```
int main() {
   PrintPizzaArea();
   PrintPizzaArea();
   return 0;
}
```

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

Check Show answer

2) How many function definitions of PrintPizzaArea() 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 PrintPizzaArea()?

Check Show answer

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

Check Show answer

Parameters

A programmer can influence a function's behavior via an input.

- A ***parameter*** is a function input specified in a function definition. Ex: A pizza area function might have diameter as an input.

- An ***argument*** is a value provided to a function's parameter during a function call. Ex: A pizza area function might be called as PrintPizzaArea(12.0) or as PrintPizzaArea(16.0).

A parameter is like a variable declaration. Upon a call, the parameter's memory location is allocated, and the parameter is assigned with the argument's value. Upon returning to the original call location, the parameter is deleted from memory.

An argument may be an expression, like 12.0, x, or x * 1.5.

PARTICIPATION ACTIVITY | 6.1.5: Function with parameters example: Printing a pizza area for different diameters.

Animation captions:

1. The function call jumps execution to the function's statements, passing 12.0 to the function's pizzaDiameter parameter.
2. The next function call passes 16.0 to the function's pizzaDiameter parameter, which results in a different pizza area.

PARTICIPATION ACTIVITY | 6.1.6: Parameters.

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

```
void PrintAge(                ) {
```

Check Show answer

2) Write a statement that calls 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

Multiple or no parameters

A function definition may have multiple parameters, separated by commas. Parameters are assigned with argument values by position: First parameter with first argument, second with second, etc.

A function definition with no parameters must still have the parentheses, as in: `void PrintSomething() { ... }`. The call must include parentheses, with no argument, as in: `PrintSomething()`.

Figure 6.1.1: Function with multiple parameters.

```
#include <iostream>
using namespace std;

void PrintPizzaVolume(double pizzaDiameter, double pizzaHeight) {
   double pizzaRadius;
   double pizzaArea;
   double pizzaVolume;
   double piVal = 3.14159265;

   pizzaRadius = pizzaDiameter / 2.0;
   pizzaArea = piVal * pizzaRadius * pizzaRadius;
   pizzaVolume = pizzaArea * pizzaHeight;
   cout << pizzaDiameter << " x " << pizzaHeight << " inch pizza is ";
   cout << pizzaVolume << " inches cubed." << endl;
}

int main() {
   PrintPizzaVolume(12.0, 0.3);
   PrintPizzaVolume(12.0, 0.8);
   PrintPizzaVolume(16.0, 0.8);
   return 0;
}
```



```
12 x 0.3 inch pizza is 33.9292 inches cubed.
12 x 0.8 inch pizza is 90.4779 inches cubed.
16 x 0.8 inch pizza is 160.85 inches cubed.
```

PARTICIPATION ACTIVITY 6.1.7: 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)
```
b is assigned with what value 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 6.1.8: Calls with multiple parameters.

Given:

```
void PrintSum(int num1, int num2) {
   cout << num1 << " + " << num2 << " is " << (num1 + num2);
}
```

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

```
PrintSum(1, 2);
```

Check Show answer

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

Check Show answer

Exploring further:

- Functions tutorial from cplusplus.com

CHALLENGE ACTIVITY 6.1.1: Function parameters.

Start

Type the program's output

```
#include <iostream>
using namespace std;

void printAge(int userAge) {
   cout << "You are " << userAge;
}

int main() {
   int ageToPrint;

   ageToPrint = 21;
   printAge(ageToPrint);

   return 0;
}
```



```
You are 21
```

1 2 3 4

Check Next

CHALLENGE ACTIVITY 6.1.2: Basic function call.

Complete the function definition to output the hours given minutes. Output for sample program:

```
3.5
```

```
#include <iostream>
using namespace std;

void OutputMinutesAsHours(double origMinutes) {

   /* Your solution goes here  */

}

int main() {
   double minutes;

   cin >> minutes;

   OutputMinutesAsHours(minutes); // Will be run with 210.0, 3600.0, and 0.0.
   cout << endl;

   return 0;
```

Run

View your last submission

CHALLENGE ACTIVITY 6.1.3: Function call with parameter: Printing formatted measurement.

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

```
5' 8"
```

Hint: Use \" to print a double quote.

```
#include <iostream>
using namespace std;

/* Your solution goes here  */

int main() {
   int userFeet;
   int userInches;

   cin >> userFeet;
   cin >> userInches;

   PrintFeetInchShort(userFeet, userInches);  // Will be run with (5, 8), then (4, 11)

   return 0;
}
```


Run

View your last submission

6.2 Return

Returning a value from a function

A function may return one value using a ***return statement***. Below, the ComputeSquare() function is defined to have a return type of int; thus, the function's return statement must have an expression that evaluates to an int.

PARTICIPATION ACTIVITY 6.2.1: Function returns computed square.

Animation captions:

1. ComputeSquare is called, passing 7 to the function's numToSquare parameter.
2. The function computes the square of the parameter numToSquare.
3. ComputeSquare(7) evaluates to 49, which is then assigned to numSquared.

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.

PARTICIPATION ACTIVITY 6.2.2: Return.

Given the definition below, indicate which are valid return statements:

```
int CalculateSomeValue(int num1, int num2) { ... }
```


1) `return 9;`
   - Yes
   - No

2) `return num1;`
   - Yes
   - No

3) `return (num1 + num2) + 1 ;`
   - Yes
   - No

4) `return;`
   - Yes
   - No

5) `return num1 num2;`
   - Yes
   - No

6) `return (0);`
   - Yes
   - No

PARTICIPATION ACTIVITY 6.2.3: More on return.

1) The following is a valid function definition:

```
char GetItem() { ... }
```

True

False

2) The following is a valid function definition for a function that returns two items:

```
int, int GetItems() { ... }
```

True

False

3) The following is a valid function definition:

```
void PrintItem() { ... }
```

True

False

Calling functions in expressions

A function call evaluates to the returned value. Thus, a function call often appears within an expression. Ex: `5 + computeSquare(4)` evaluates to 5 + 16, or 21.

A function with a void return type cannot be used within an expression, instead being used in a statement like: `outputData(x, y);`

PARTICIPATION ACTIVITY 6.2.4: Calls in an expression.

Given the definitions below, which are valid statements?

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

1) `y = SquareRoot(49.0);`

True

False

2) `SquareRoot(49.0) = z;`

True

False

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

True

False

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

- True
- False

5) `y = SquareRoot();`

- True
- False

6) `SquareRoot(9.0);`

- True
- False

7) `y = PrintVal(9.0);`

- True
- False

8) `PrintVal(9.0);`

- True
- False

Mathematical functions

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

Figure 6.2.1: Program with a function to convert height in feet/inches to centimeters.


```
Enter feet: 5
Enter inches: 8
Centimeters: 172.72
```

```
#include <iostream>
using namespace std;

/* 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;
   double cmVal;

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

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

   // Prompt user for feet/inches
   cout << "Enter feet: ";
   cin >> userFt;

   cout << "Enter inches: ";
   cin >> userIn;

   // Output the conversion result
   cout << "Centimeters: ";
   cout << HeightFtInToCm(userFt, userIn) << endl;

   return 0;
}
```


Human average height is increasing, attributed to better nutrition. Source: Our World in Data: Human height.

PARTICIPATION ACTIVITY 6.2.5: Mathematical functions.

Indicate which is a valid use of the HeightFtInToCm() function above. x is type double.

1) x = HeightFtInToCm(5, 0);

- ○ Valid
- ○ Not valid

2) x = 2 * (HeightFtInToCm(5, 0) + 1.0);

- Valid
- Not valid

3) x = (HeightFtInToCm(5, 0) + HeightFtInToCm(6, 1)) / 2.0;

- Valid
- Not valid

4) Suppose `int pow(int y, int z)` returns y to the power of z. Is the following valid?
x = pow(2, pow(3, 2));

- Valid
- Not valid

zyDE 6.2.1: Temperature conversion.

Complete the program by writing and calling a function that converts a temperature fror Celsius into Fahrenheit using the formula: $Fahrenheit = \frac{9}{5} * Celsius + 32$

Load default template...

```
1  #include <iostream>
2  using namespace std;
3
4
5  // FINISH: Define CelsiusToFahrenheit f
6
7
8  int main() {
9     double tempF;
10    double tempC;
11
12    cout << "Enter temperature in Celsiu
13    cin >> tempC;
14
15    tempF = 0.0; // FIXME
16
17    cout << "Fahrenheit: " << tempF;
18
```

100

Run

Calling functions from functions

A function's statements may call other functions. In the example below, the PizzaCalories() function calls the CalcCircleArea() function. (Note that main() itself is the first function called when a program executes, and calls other functions.)

Figure 6.2.2: Functions calling functions.


```
#include <iostream>
using namespace std;

double CalcCircleArea(double circleDiameter) {
   double circleRadius;
   double circleArea;
   double piVal = 3.14159265;

   circleRadius = circleDiameter / 2.0;
   circleArea = piVal * circleRadius * circleRadius;

   return circleArea;
}

double PizzaCalories(double pizzaDiameter) {
   double totalCalories;
   double caloriesPerSquareInch = 16.7;    // Regular crust pepperoni pizza

   totalCalories = CalcCircleArea(pizzaDiameter) * caloriesPerSquareInch;

   return totalCalories;
}


int main() {
   cout << "12 inch pizza has " << PizzaCalories(12.0) << " calories." << endl;
   cout << "14 inch pizza has " << PizzaCalories(14.0) << " calories." << endl;

   return 0;
}
```

```
12 inch pizza has 1888.73 calories.
14 inch pizza has 2570.77 calories.
```

PARTICIPATION ACTIVITY 6.2.6: Functions calling functions.

Complete the PizzaCaloriesPerSlice() function to compute the calories for a single slice of pizza. A PizzaCalories() function returns a pizza's total calories given the pizza diameter passed as an argument. A PizzaSlices() function returns the number of slices in a pizza given the pizza diameter passed as an argument.

```
double PizzaCaloriesPerSlice(double pizzaDiameter) {
   double totalCalories;
   double caloriesPerSlice;

   totalCalories = Placeholder_A;
   caloriesPerSlice = Placeholder_B;

   return caloriesPerSlice;
}
```


1) Type the expression for Placeholder_A to compute the total calories for a pizza with diameter pizzaDiameter.

```
totalCalories =
                              ;
```

Check Show answer

2) Type the expression for Placeholder_B to compute the calories per slice.

```
caloriesPerSlice =
                              ;
```

Check Show answer

Exploring further:

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

CHALLENGE ACTIVITY 6.2.1: Enter the output of the returned value.

Start

Type the program's output

5

```
#include <iostream>
using namespace std;

int ChangeValue(int x) {
   return x + 1;
}

int main() {
   cout << ChangeValue(4) << endl;

   return 0;
}
```


1 2

Check Next

CHALLENGE ACTIVITY 6.2.2: 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.

```
1  #include <iostream>
2  using namespace std;
3
4  double FindMax(double num1, double num2) {
5     double maxVal;
6
7     // Note: if-else statements need not be understood to complete this activity
8     if (num1 > num2) { // if num1 is greater than num2,
9        maxVal = num1;  // then num1 is the maxVal.
10    }
11    else {             // Otherwise,
12       maxVal = num2;  // num2 is the maxVal.
13    }
14
15    return maxVal;
16 }
17
18 int main() {
```


Run

View your last submission

CHALLENGE ACTIVITY 6.2.3: 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).


```
#include <iostream>
using namespace std;

/* Your solution goes here  */

int main() {
   double userLength;
   double userWidth;
   double userHeight;

   cin >> userLength;
   cin >> userWidth;
   cin >> userHeight;

   cout << "Volume: " << PyramidVolume(userLength, userWidth, userHeight) << endl;

   return 0;
}
```

Run

View your last submission

6.3 Reasons for defining functions

Improving program readability

Programs can become hard for humans to read and understand. Decomposing a program into functions can greatly aid program readability, helping yield an initially correct program, and easing future maintenance. Below, the program with functions has a main() that is easier to read and understand. For larger programs, the effect is even greater.

Figure 6.3.1: With program functions: main() is easy to read and understand.

```
#include <iostream>
using namespace std;

double StepsToMiles(int numSteps) {
   const double FEET_PER_STEP = 2.5;              // Typical adult
   const int    FEET_PER_MILE = 5280;

   return numSteps * FEET_PER_STEP * (1.0 / FEET_PER_MILE);
}

double StepsToCalories(int numSteps) {
   const double STEPS_PER_MINUTE = 70.0;          // Typical adult
   const double CALORIES_PER_MINUTE_WALKING = 3.5; // Typical adult
   double minutesTotal;
   double caloriesTotal;

   minutesTotal = numSteps / STEPS_PER_MINUTE;
   caloriesTotal = minutesTotal * CALORIES_PER_MINUTE_WALKING;

   return caloriesTotal;
}

int main() {
   int stepsWalked;

   cout << "Enter number of steps walked: ";
   cin  >> stepsWalked;

   cout << "Miles walked: " << StepsToMiles(stepsWalked)    << endl;
   cout << "Calories: "     << StepsToCalories(stepsWalked) << endl;

   return 0;
}
```

```
Enter number of steps walked: 1600
Miles walked: 0.757576
Calories: 80
```


Figure 6.3.2: Without program functions: main() is harder to read and understand.

```
#include <iostream>
using namespace std;

int main() {
   int    stepsWalked;
   const double FEET_PER_STEP = 2.5;                // Typical adult
   const int    FEET_PER_MILE = 5280;
   const double STEPS_PER_MINUTE = 70.0;            // Typical adult
   const double CALORIES_PER_MINUTE_WALKING = 3.5; // Typical adult
   double minutesTotal;
   double caloriesTotal;
   double milesWalked;

   cout << "Enter number of steps walked: ";
   cin >> stepsWalked;

   milesWalked = stepsWalked * FEET_PER_STEP * (1.0 / FEET_PER_MILE);
   cout << "Miles walked: " << milesWalked << endl;

   minutesTotal = stepsWalked / STEPS_PER_MINUTE;
   caloriesTotal = minutesTotal * CALORIES_PER_MINUTE_WALKING;
   cout << "Calories: " << caloriesTotal << endl;

   return 0;
}
```


PARTICIPATION ACTIVITY 6.3.1: Improved readability.

Consider the above examples.

1) In the example *without* functions, how many statements are in main()?
   - 6
   - 16

2) In the example *with* functions, how many statements are in main()?
   - 6
   - 16

3) Which has fewer *total* lines of code (including blank lines), the program with or without functions?
   - With
   - Without

- Same

4) The program with functions called the functions directly in output statements. Did the program without functions directly put calculations in output statements?
   - No
   - Yes

Modular and incremental program development

Programmers commonly use functions to write programs modularly and incrementally.

- ***Modular development*** is the process of dividing a program into separate modules that can be developed and tested separately and then integrated into a single program.
- ***Incremental development*** is a process in which a programmer writes, compiles, and tests a small amount of code, then writes, compiles, and tests a small amount more (an incremental amount), and so on.
- A ***function stub*** is a function definition whose statements have not yet been written.

A programmer can use function stubs to capture the high-level behavior of main() and the required function (or modules) before diving into details of each function, like planning a route for a road trip before starting to drive. A programmer can then incrementally develop and test each function independently.

PARTICIPATION ACTIVITY 6.3.2: Function stub used in incremental program development.

Animation captions:

1. main() captures the program's high-level behavior and makes calls to three functions stubs. The program can be compiled and tested before writing and testing those functions.
2. The programmer then writes and tests the ConvKilometersToMiles() and ConvLitersToGallons() functions.
3. After testing the ConvKilometersToMiles() and ConvLitersToGallons() functions, the programmer can then complete the program by writing and testing the CalcMpg() function.

PARTICIPATION ACTIVITY 6.3.3: Incremental development.

1) Incremental development may involve more frequent compilation, but

ultimately lead to faster development of a program.

- ○ True
- ○ False

2) The program above does not compile because CalcMpg() is a function stub.

- ○ True
- ○ False

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

- ○ True
- ○ False

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

- ○ True
- ○ False

zyDE 6.3.1: Function stubs.

Complete the program by writing and testing the CalcMpg() function.

Load default templ

```
#include <iostream>
using namespace std;
// Program converts a trip's kilometers and liters into miles, gallons, and mpg

double ConvKilometersToMiles(double numKm) {
   double milesPerKm = 0.621371;
   return numKm * milesPerKm;
}

double ConvLitersToGallons(double numLiters) {
   double gallonsPerLiter = 0.264172;
   return numLiters * gallonsPerLiter;
}

double CalcMpg(double distMiles, double gasGallons) {
   cout << "FIXME: Calculate MPG" << endl;
   return 0.0;
```

410.2 33.5

Run

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 abs() that relieve a programmer from having to write several lines of code each time an absolute value needs to be computed.

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.

A general guideline (especially for beginner programmers) is that a function's definition usually shouldn't have more than about 30 lines of code, although this guideline is not a strict rule.

PARTICIPATION ACTIVITY 6.3.4: Redundant code can be replaced by multiple calls to one function.

Animation content:

`undefined`

Animation captions:

1. Circle area is calculated twice, leading to redundant code.
2. The redundant code can be replaced by defining a CalcCircleArea function.
3. Then main is simplified by calling the CalcCircleArea() function from multiple places in the program.

PARTICIPATION ACTIVITY 6.3.5: 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 6.3.1: Functions: Factoring out a unit-conversion calculation.

Write a function so that the main() code below can be replaced by the simpler code that calls function MphAndMinutesToMiles(). Original main():

```
int main() {
   double milesPerHour;
   double minutesTraveled;
   double hoursTraveled;
   double milesTraveled;

   cin >> milesPerHour;
   cin >> minutesTraveled;

   hoursTraveled = minutesTraveled / 60.0;
   milesTraveled = hoursTraveled * milesPerHour;

   cout << "Miles: " << milesTraveled << endl;

   return 0;
}
```



```
1 #include <iostream>
2 using namespace std;
3
4 /* Your solution goes here  */
5
```

```
int main() {
   double milesPerHour;
   double minutesTraveled;

   cin >> milesPerHour;
   cin >> minutesTraveled;

   cout << "Miles: " << MphAndMinutesToMiles(milesPerHour, minutesTraveled) << endl;

   return 0;
}
```


Run

View your last submission

CHALLENGE ACTIVITY 6.3.2: 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
```

```
#include <iostream>
using namespace std;

/* Your solution goes here  */

int main() {
   int userNum1;
   int userNum2;
   int avgResult;

   userNum1 = GetUserNum();
   userNum2 = GetUserNum();

   avgResult = ComputeAvg(userNum1, userNum2);

   cout << "Avg: " << avgResult << endl;

   return 0;
```


Run

View your last submission

6.4 Functions with branches/loops

Example: Auction website fee calculator

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

```
Enter item selling price (Ex:
65.00): 9.95
eBay fee: $1.7935

...

Enter item selling price (Ex:
65.00): 40
eBay fee: $5.7

...

Enter item selling price (Ex:
65.00): 100
eBay fee: $9.5

...

Enter item selling price (Ex:
65.00): 500.15
eBay fee: $29.5075

...

Enter item selling price (Ex:
65.00): 2000
eBay fee: $74.5
```

```
#include <iostream>
using namespace std;

/* 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 a % of the selling
price:
   13% for $50.00 or less
    plus 5% for $50.01 to $1000.00
    plus 2% for $1000.01 or more
   Source:
http://pages.ebay.com/help/sell/fees.html, 2012.

   Note: double variables often are not used for
dollars/cents,
   but here the dollar fraction may extend past two
decimal places.
*/

// 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 feeTotal;                       //
Resulting eBay fee

   feeTotal = BASE_LIST_FEE;

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

   return feeTotal;
}

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

   cout << "Enter item selling price (Ex: 65.00): ";
   cin >> sellingPrice;

   cout << "eBay fee: $" << EbayFee(sellingPrice) <<
endl;

   return 0;
}
```


©zyBooks 03/25/21 10:43 488201
xiang zhao
BAYLORCSI14301440Spring2021

PARTICIPATION ACTIVITY 6.4.1: Analyzing the eBay fee calculator.

1) For any call to EbayFee() function, how many assignment statements for the variable `feeTotal` will execute?

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

Example: Least-common multiple calculator

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

Figure 6.4.2: User-defined functions make main() easy to understand.

```
#include <iostream>
#include <cmath>
using namespace std;

// Function prompts user to enter positive non-zero
number
```

```
int GetPositiveNumber() {
   int userNum;

   userNum = 0;

   while (userNum <= 0) {
      cout << "Enter a positive number (>0): " <<
endl;
      cin >> userNum;

      if (userNum <= 0) {
         cout << "Invalid number." << endl;
      }
   }

   return userNum;
}


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

   numA = aVal;
   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;

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

   return lcmVal;
}

int main() {
   int usrNumA;
   int usrNumB;
   int lcmResult;

   cout << "Enter value for first input" << endl;
   usrNumA = GetPositiveNumber();

   cout << endl << "Enter value for second input" <<
endl;
   usrNumB = GetPositiveNumber();

   lcmResult = FindLCM(usrNumA, usrNumB);

   cout << endl << "Least common multiple of " <<
usrNumA
        << " and " << usrNumB << " is " << lcmResult
<< endl;

   return 0;
```

```
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
```


©zyBooks 03/25/21 10:43 488201
xiang zhao
BAYLORCSI14301440Spring2021

```
}
```

PARTICIPATION ACTIVITY 6.4.2: 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 6.4.1: Output of functions with branches/loops.

Start

Type the program's output

```
#include <iostream>
#include <string>
using namespace std;

void PrintMessage(string message) {
   if (message.length() > 7) {
      cout << "too long" << endl;
   }
   else {
      cout << message << endl;
   }
}

int main() {
   PrintMessage("Hi!");
   PrintMessage("Are you here?");

   return 0;
}
```

```
Hi!
too long
```


1 2

Check Next

CHALLENGE ACTIVITY 6.4.2: Function with branch: Popcorn.

Complete function PrintPopcornTime(), with int parameter bagOunces, and void return type. If bagOunces is less than 2, 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
```

```
#include <iostream>
using namespace std;

void PrintPopcornTime(int bagOunces) {

   /* Your solution goes here  */

}

int main() {
   int userOunces;

   cin >> userOunces;
   PrintPopcornTime(userOunces);

   return 0;
}
```

Run

View your last submission

CHALLENGE ACTIVITY 6.4.3: 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 with input 2:

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


Hint: Declare and use a loop variable.

```
#include <iostream>
using namespace std;

/* Your solution goes here  */

int main() {
   int userCycles;

   cin >> userCycles;
   PrintShampooInstructions(userCycles);

   return 0;
}
```

Run

View your last submission

6.5 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 6.5.1: Test harness for the function HrMinToMin().


```
#include <iostream>
using namespace std;

// Function converts hrs/min to min
int HrMinToMin(int origHours, int origMinutes) {
   int totMinutes; // Resulting minutes

   totMinutes = (origHours * 60) + origMinutes;

   return origMinutes;
}

int main() {

   cout << "Testing started" << endl;

   cout << "0:0, expecting 0, got: "    << HrMinToMin(0, 0)
<< endl;
   cout << "0:1, expecting 1, got: "    << HrMinToMin(0, 1)
<< endl;
   cout << "0:99, expecting 99, got: "  << HrMinToMin(0, 99)
<< endl;
   cout << "1:0, expecting 60, got: "   << HrMinToMin(1, 0)
<< endl;
   cout << "5:0, expecting 300, got: "  << HrMinToMin(5, 0)
<< endl;
   cout << "2:30, expecting 150, got: " << HrMinToMin(2, 30)
<< endl;
   // Many more test vectors would be typical...

   cout << "Testing completed" << endl;

   return 0;
}
```

```
Testing started
0:0, expecting 0, got: 0
0:1, expecting 1, got: 1
0:99, expecting 99, got:
99
1:0, expecting 60, got:
0
5:0, expecting 300, got:
0
2:30, expecting 150, got:
30
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 6.5.1: 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. 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 cassert library, part of the standard library, as shown below.

Figure 6.5.2: Test harness with assert for the function HrMinToMin().

```
#include <iostream>
#include <cassert>
using namespace std;

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

   totMinutes = (origHours * 60) + origMinutes;

   return origMinutes;
}

int main() {

   cout << "Testing started" << endl;

   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...

   cout << "Testing completed" << endl;

   return 0;
}
```

```
Testing started
Assertion failed: (HrMinToMin(1, 0) == 60), function main, file main.cpp, line 20.
```

Using branches for unit tests

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


```
if ( HrMinToMin(0, 0) != 0 ) {
   cout << "0:0, expecting 0, got: " << HrMinToMin(0, 0) << endl;
}
```

But the assert is more compact.

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 6.5.2: Assertions and test cases.

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

- ○ True
- ○ False

2) For a 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 6.5.1: 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).

```
#include <iostream>
using namespace std;

// Function returns origNum cubed
int CubeNum(int origNum) {
   return origNum * origNum * origNum;
}

int main() {

   cout << "Testing started" << endl;

   cout << "2, expecting 8, got: " << CubeNum(2) << endl;

   /* Your solution goes here  */

   cout << "Testing completed" << endl;

```


Run

View your last submission

6.6 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 6.6.1: Function calls and returns.

Animation captions:

1. Each function call creates a new set of local variables.
2. Each return causes those local variables to be discarded.

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.

PARTICIPATION ACTIVITY 6.6.2: How function call/return works.

Animation content:


```
undefined
```

Animation captions:

1. The compiler converts high-level code into assembly instructions in memory.
2. Before executing the function, arguments are copied to parameter local variables and a return address is stored.

3. The function executes and stores the result in a designated return value location.
4. 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.

PARTICIPATION ACTIVITY 6.6.3: 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

6.7 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 6.7.1: Copy-paste common error: Pasted code not properly modified. Find error on the right.

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

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

   return fahVal;
}
```

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

   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 6.7.1: Copy-pasted sum-of-squares code.

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

```
int sum;

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;

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

```
#include <iostream>
using namespace std;

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

   feetTot = baseSteps * FEET_PER_STEP;
}

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

   // Prompt user for input
   cout << "Enter number of steps walked: ";
   cin >> stepsInput;

   // Call functions to convert steps to feet
   feetTot = StepsToFeet(stepsInput);
   cout << "Feet: " << feetTot << endl;

   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 6.7.2: Common function errors.

Find the error in the function's code.


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

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

     return;
```

```
}
```

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

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

   return num;
}
```


PARTICIPATION ACTIVITY 6.7.3: 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 6.7.1: 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.


```
#include <iostream>
using namespace std;

double CelsiusToKelvin(double valueCelsius) {
   double valueKelvin;

   valueKelvin = valueCelsius + 273.15;

   return valueKelvin;
}

/* Your solution goes here  */

int main() {
   double valueC;
   double valueK;

   valueC = 10.0;
```

Run

View your last submission

6.8 Pass by reference

Pass by reference

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

Animation captions:

1. The user is prompted to specify the total time in minutes. Function ConvHrMin is then called with arguments totTime, usrHr, and usrMin.
2. ConvHrMin's parameters are passed by value, so the arguments' values are copied into local variables.
3. Upon return, ConvHrMin's local variables are discarded. hrVal and minVal are local copies that do not impact usrHr and usrMin.

C++ supports another kind of parameter that enables updating of an argument variable. A ***pass by reference*** parameter does *not* create a local copy of the argument, but rather the parameter refers directly to the argument variable's memory location. Appending & to a parameter's data type makes the parameter pass by reference type.

PARTICIPATION ACTIVITY 6.8.2: A pass by reference parameter allows a function to update an argument variable.

Animation captions:

1. The user is prompted to specify the total time in minutes. Function ConvHrMin is then called with arguments totTime, usrHr, and usrMin.
2. The & indicates that the hrVal and minVal parameters are passed by reference.
3. Parameters passed by reference refer to that variable's memory location, so updates to hrVal and minVal update usrHr and usrMin.
4. Upon return from ConvHrMin, usrHr and usrMin retain the updated values.

Pass by reference 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 reference 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& numl` or converting from polar to Cartesian coordinates, whose function might be `void PolarToCartesian(int radialPol, int anglePol, int& xCar, int& yCar)`.

zyDE 6.8.1: Calculating monetary change.

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

Load default templ


```
#include <iostream>
using namespace std;

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

   cout << "FIXME: Finish writing ComputeChange" << endl;

   numQuarters = totCents / 25;
}

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

   cout << "Enter total cents: " << endl;
```

83

Run

PARTICIPATION ACTIVITY 6.8.3: Function definition returns and arguments.

Choose the most appropriate function definition.

1) Convert inches into centimeters.

- `void InchToCM(double inches, double centimeters) ...`
- `double InchToCM(double inches) ...`
- More than one function should

be written.

2) Get a user's full name by prompting "Enter full name" and then automatically separating into first and last names.

- `void GetUserFullName(string& firstName, string& lastName) ...`
- `string GetUserFullName() ...`
- `string, string GetUserFullName() ...`
- More than one function should be written.

3) 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 6.8.4: Function definitions with pass by value and pass by reference.

Complete the function definition, creating pass by value or pass by reference parameters as appropriate.

1) Convert gallons to liters. Parameter is userGallons, type is double.

```
double GallonsToLiters(
                       ) {
```

Check Show answer

2) Convert userMeters into userFeet and userInches (three parameters, in that order), types are doubles.

```
void MetersToFeetInches(
                        ) {
```


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 6.8.1: 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 6.8.5: 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 an additional local variable.
   - True
   - False

Reference variables

A programmer can also declare a reference variable. A ***reference*** is a variable type that refers to another variable. Ex: `int& maxValRef` declares a reference to a variable of type int. The programmer must initialize each reference with an existing variable, which can be done by initializing the reference variable when the reference is declared. Ex: `int& maxValRef = usrInput3;`

In the example below, usrValRef is a reference that refers to usrValInt. The user-entered number is assigned to the variable usrValInt. Because usrValRef refers to usrValInt, printing usrValInt or usrValRef will print the number.

Figure 6.8.2: Reference variable example.

```
#include <iostream>
using namespace std;

int main() {
   int usrValInt;
   int& usrValRef = usrValInt;  // Refers to usrValInt

   cout << "Enter an integer: ";
   cin  >> usrValInt;

   cout << "We wrote your integer to usrValInt." << endl;
   cout << "usrValInt is: " << usrValInt << "." << endl;
   cout << "usrValRef refers to usrValInt, and is: " << usrValRef << "." << endl;

   usrValInt = 99;
   cout << endl << "We assigned usrValInt with 99." << endl;
   cout << "usrValInt is now: " << usrValInt << "." << endl;
   cout << "usrValRef is now: " << usrValRef << "." << endl;
   cout << "Note that usrValRef refers to usrValInt, so changed too." << endl;
   return 0;
}
```



```
Enter an integer: 42
We wrote your integer to usrValInt.
usrValInt is: 42.
usrValRef refers to usrValInt, and is: 42.

We assigned usrValInt with 99.
usrValInt is now: 99.
usrValRef is now: 99.
Note that usrValRef refers to usrValInt, so changed too.
```

PARTICIPATION ACTIVITY 6.8.6: Reference variables.

1) What does the following output?

```
int numAStudents = 12;
int numBStudents = 5;
int& studentsRef = numAStudents;

cout << studentsRef;
```

Check Show answer

2) What does the following output?

```
int examGrade = 95;
int& gradeRef = examGrade;

examGrade = examGrade + 1;
cout << gradeRef;
```


Check Show answer

3) What does the following output?

```
double treeHeightFt = 7.1;
double& heightRef = treeHeightFt;

heightRef = 12.2;
cout << treeHeightFt;
```

Check Show answer

4) Declare a reference named myScore and initialize the reference to the previously declared variable int teamScore.

Check Show answer

Exploring further:

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

CHALLENGE ACTIVITY 6.8.1: Function pass by reference: Transforming coordinates.

Define a function CoordTransform() that transforms the function's 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.

```
1  #include <iostream>
2  using namespace std;
3
4  /* Your solution goes here  */
5
6  int main() {
7     int xValNew;
8     int yValNew;
9     int xValUser;
10    int yValUser;
11
12    cin >> xValUser;
13    cin >> yValUser;
14
```

```
15    CoordTransform(xValUser, yValUser, xValNew, yValNew);
16    cout << "(" << xValUser << ", " << yValUser << ") becomes (" << xValNew << ", " << y
17
18    return 0;
```

Run

View your last submission

6.9 Functions with string/vector parameters

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

Figure 6.9.1: Modifying a string parameter, which should be pass by reference.

```
#include <iostream>
#include <string>
using namespace std;

// Function replaces spaces with hyphens
void StrSpaceToHyphen(string& modStr) {
   unsigned int i;   // Loop index

   for (i = 0; i < modStr.size(); ++i) {
      if (modStr.at(i) == ' ') {
         modStr.at(i) = '-';
      }
   }
}

int main() {
   string userStr;  // Input string from user

   // Prompt user for input
   cout << "Enter string with spaces: " << endl;
   getline(cin, userStr);

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

   // Output modified string
   cout << "String with hyphens: ";
   cout << userStr << endl;

   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 string serves as function input and output. The string parameter must be pass by reference, achieved using & (yellow highlighted), so that the function modifies the original string argument (userStr) and not a copy.

zyDE 6.9.1: Modifying a string parameter: Spaces to hyphens.

1. Run the program, noting correct output.
2. Remove the & and run again, noting the string is not modified, because the string i by value and thus the function modifies a copy. When done replace the &
3. Modify the function to also replace each '!' by a '?'.

Load default template...

```
#include <iostream>
#include <string>
using namespace std;

// Function replaces spaces with hyphen
void StrSpaceToHyphen(string& modStr) {
   unsigned int i;  // Loop index

   for (i = 0; i < modStr.size(); ++i)
      if (modStr.at(i) == ' ') {
         modStr.at(i) = '-';
      }
   }
}

int main() {
   string userStr;  // Input string fro

```

Hello there everyone!!!

Run

Sometimes a programmer defines a vector or string parameter as pass by reference even though the function does not modify the parameter, to prevent the performance and memory overhead of copying the argument that would otherwise occur.

The keyword ***const*** can be prepended to a function's vector or string parameter to prevent the function from modifying the parameter. Programmers commonly make a large vector or string input parameter pass by reference, to gain efficiency, while also making the parameter const, to prevent assignment.

The following illustrates. The first function modifies the vector so defines a normal pass by reference (highlighted yellow). The second function does *not* modify the vector but for efficiency uses constant pass by reference (highlighted orange).

Figure 6.9.2: Normal and constant pass by reference vector parameters in a vector reversal program.

```
#include <iostream>
#include <vector>
using namespace std;

void ReverseVals(vector<int>& vctrVals) {
   unsigned int i;  // Loop index
   int tmpVal;      // Temp variable for swapping

   for (i = 0; i < (vctrVals.size() / 2); ++i) {
      tmpVal = vctrVals.at(i); // These statements
swap
      vctrVals.at(i) = vctrVals.at(vctrVals.size() - 1
- i);
      vctrVals.at(vctrVals.size() - 1 - i) = tmpVal;
   }
}

void PrintVals(const vector<int>& vctrVals) {
   unsigned int i;  // Loop index

   // Print updated vector
   cout << endl << "New values: ";
   for (i = 0; i < vctrVals.size(); ++i) {
      cout << " " << vctrVals.at(i);
   }
   cout << endl;
}

int main() {
   const int NUM_VALUES = 8;             // Vector size
   vector<int> userValues(NUM_VALUES);  // User values
   int i;                               // Loop index

   // Prompt user to populate vector
   cout << "Enter " << NUM_VALUES << " values..." <<
endl;
   for (i = 0; i < NUM_VALUES; ++i) {
      cout << "Value: ";
      cin >> userValues.at(i);
   }

   // Call function to reverse vector values
   ReverseVals(userValues);

   // Print reversed values
   PrintVals(userValues);

   return 0;
}
```

```
Enter 8 values...
Value: 10
Value: 20
Value: 30
Value: 40
Value: 50
Value: 60
Value: 70
Value: 80

New values:  80 70 60 50 40 30
20 10
```


©zyBooks 03/25/21 10:43 488201
xiang zhao
BAYLORCSI14301440Spring2021

A reader might wonder why all input parameters are not defined as constant pass by reference parameters: Why make local copies at all? The reason is efficiency. For parameters involving just a few memory locations, making a local copy enables the compiler to generate more efficient code, in part because the compiler can place those copies inside a tiny-but-fast memory inside the processor called a register file—further details are beyond our scope.

In summary:

- Define a function's output or input/output parameters as pass by reference.
  - But create output parameters sparingly, striving to use return values instead.
- Define input parameters as pass by value.
  - Except for large items (perhaps 10 or more elements); use constant pass by reference for those.

PARTICIPATION ACTIVITY 6.9.1: Constants and pass by reference.

How should a function's vector parameter `ages` be defined for the following situations?

1) ages will always be small (fewer than 10 elements) and the function will not modify the vector.
   - Constant and pass by reference.
   - Constant but not pass by reference.
   - Pass by reference but not constant.
   - Neither constant nor pass by reference.

2) ages will always be small, and the function will modify the vector.
   - Constant and pass by reference.
   - Constant but not pass by reference.
   - Pass by reference but not constant.
   - Neither constant nor pass by reference.

3) ages may be very large, and the function will modify the vector.
   - Constant and pass by reference.
   - Constant but not pass by reference.
   - Pass by reference but not constant.
   -

Neither constant nor pass by reference.

4) ages may be very large, and the function will not modify the vector.

- Constant and pass by reference.
- Constant but not pass by reference.
- Pass by reference but not constant.
- Neither constant nor pass by reference.

PARTICIPATION ACTIVITY 6.9.2: Vector parameters.

Define a function's vector parameter **ages** for the following situations. Assume ages is a vector of integers. Example: ages will always be small (fewer than 10 elements) and the function will not modify the vector: `const vector<int> ages`.

1) ages will always be small, and the function will modify the vector.

```
void MyFct (
                    ) {
```

Check Show answer

2) ages may be very large, and the function will modify the vector

```
void MyFct (
                    ) {
```

Check Show answer

3) ages may be very large, and the function will not modify the vector.

```
void MyFct (
                    ) {
```

Check Show answer

CHALLENGE

ACTIVITY 6.9.1: Use an existing function.

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

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



```
#include <iostream>
#include <string>
using namespace std;

void GetUserInfo(int& userAge, string& userName) {
   cout << "Enter your age: " << endl;
   cin >> userAge;
   cout << "Enter your name: " << endl;
   cin >> userName;
}

int main() {
   int userAge;
   string userName;

   /* Your solution goes here  */

   cout << userName << " is " << userAge << " years old." << endl;
```

Run

View your last submission

CHALLENGE ACTIVITY 6.9.2: 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!"
```



```
#include <iostream>
#include <string>
using namespace std;

void MakeSentenceExcited(string& sentenceText) {
```

```
   /* Your solution goes here  */

}

int main() {
   string testStr;

   getline(cin, testStr);
   MakeSentenceExcited(testStr);
   cout << testStr;

   return 0;
```


Run

View your last submission

CHALLENGE ACTIVITY 6.9.3: Modify a vector parameter.

Write a function SwapVectorEnds() that swaps the first and last elements of its vector parameter. Ex: sortVector = {10, 20, 30, 40} becomes {40, 20, 30, 10}.

```
#include <iostream>
#include <vector>
using namespace std;

/* Your solution goes here  */

int main() {
   vector<int> sortVector(4);
   unsigned int i;
   int userNum;

   for (i = 0; i < sortVector.size(); ++i) {
      cin >> userNum;
      sortVector.at(i) = userNum;
   }

   SwapVectorEnds(sortVector);

```


Run

View your last submission

6.10 Functions with C string parameters

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

Figure 6.10.1: Modifying a C string parameter.


```
#include <iostream>
#include <cstring>
using namespace std;

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

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

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

   // Prompt user for input
   cout << "Enter string with spaces: " << endl;
   cin.getline(userStr, INPUT_STR_SIZE);

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

   cout << "String with hyphens: " << userStr <<
endl;

   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 C 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 C string parameter does not require a second parameter to specify the string size.

zyDE 6.10.1: Modifying a C 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 <iostream>
#include <cstring>
using namespace std;

// Function replaces spaces with hyphen
void StrSpaceToHyphen(char modString[])
   int i;      // Loop index

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

int main() {
```

Hello there everyone!!!

Run

PARTICIPATION ACTIVITY 6.10.1: Functions with string parameters.

1) A parameter declared as `char movieTitle[]` is a string.
   - 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 C string parameters, such as in the C string library functions.

Figure 6.10.2: Modifying a C string using a pointer parameter.


```
#include <iostream>
#include <cstring>
using namespace std;

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

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

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

   // Prompt user for input
   cout << "Enter string with spaces: " << endl;
   cin.getline(userStr, INPUT_STR_SIZE);

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

   cout << "String with hyphens: " << userStr <<
endl;

   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 6.10.2: Functions with C string parameters.

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

- True
- False

2) A C string is automatically passed by pointer.

- True

False

CHALLENGE ACTIVITY 6.10.1: Modify a C 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!"
```

```
#include <iostream>
#include <cstring>
using namespace std;

void MakeSentenceExcited(char* sentenceText) {

   /* Your solution goes here  */

}

int main() {
   const int TEST_STR_SIZE = 50;
   char testStr[TEST_STR_SIZE];

   cin.getline(testStr, TEST_STR_SIZE);
   MakeSentenceExcited(testStr);
   cout << testStr << endl;

```

Run

View your last submission

6.11 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 6.11.1: Local variable scope.

```
#include <iostream>
using namespace std;

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;
   double cmVal;

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

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

   // Prompt user for feet/inches
   cout << "Enter feet: ";
   cin >> userFt;

   cout << "Enter inches: ";
   cin >> userIn;

   // Output the conversion result
   cout << "Centimeters: ";
   cout << HeightFtInToCm(userFt, userIn) << endl;

   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 6.11.1: 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 programmer declares a function's local variable to have the same name as a function parameter, the name will refer to the local variable.
   - True
   - False

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

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


```
#include <iostream>
#include <cmath> // To use "pow" function
using namespace std;

/* 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() {
   double pastDol;         // Starting dollar amount
   double currDol;         // Ending dollar amount (converted value)
   int pastYr;             // Starting year
   int currYr;             // Ending year (converted to year)

   // Prompt user for previous year/dollar and current year
   cout << "Enter current year: ";
   cin >> currYr;
   cout << "Enter past year: ";
   cin >> pastYr;
   cout << "Enter past dollars (Ex: 1000): ";
   cin >> pastDol;

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

   cout << "$" << pastDol << " in " << pastYr;
   cout << " is about $" << currDol << " in ";
   cout << currYr << endl;

   return 0;
}

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

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

   return currDol;
}
```

©zyBooks 03/25/21 10:43 488201
xiang zhao
BAYLORCSI14301440Spring2021

```
Enter current year: 2015
Enter past year: 1970
Enter past dollars (Ex: 1000): 10000
$10000 in 1970 is about $58411.8 in 2015
(Note: 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 in 2015
(Note: 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 $922.435 in 2015
(Note: 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 $151.871 in 2015
(Note: 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 6.11.2: 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

6.12 Default parameter values

Sometimes a function's last parameter (or last few) should be optional. A function call could then omit the last argument, and instead the program would use a default value for that parameter. A function can have a ***default parameter value*** for the last parameter(s), meaning a call can optionally omit a corresponding argument.

Figure 6.12.1: Parameter with a default value.

```
#include <iostream>
using namespace std;

// Function prints date in two styles (0: American (default), 1: European)
void DatePrint(int currDay, int currMonth, int currYear, int printStyle = 0) {

   if (printStyle == 0) {      // American
      cout << currMonth << "/" << currDay << "/" << currYear;
   }
   else if (printStyle == 1) { // European
      cout << currDay << "/" << currMonth << "/" << currYear;
   }
   else {
      cout << "(invalid style)";
   }
}

int main() {

   // Print dates given various style settings
   DatePrint(30, 7, 2012, 0);
   cout << endl;

   DatePrint(30, 7, 2012, 1);
   cout << endl;

   DatePrint(30, 7, 2012); // Uses default value for printStyle
   cout << endl;

   return 0;
}
```

```
7/30/2012
30/7/2012
7/30/2012
```

The fourth (and last) parameter has a default value: `int printStyle = 0`. If a function call does not provide a fourth argument, then the style parameter is 0.

The same can be done for other parameters, as in:
`void DatePrint(int currDay = 1, int currMonth = 1, int currYear = 2000, in`
Because arguments are matched with parameters based on their ordering in the function call, only the last arguments can be omitted. The following are valid calls to this DatePrint() function having default values for all parameters:

Figure 6.12.2: Valid function calls with default parameter values.

```
DatePrint(30, 7, 2012, 0); // No defaults
DatePrint(30, 7, 2012);    // Defaults:                              style=0
DatePrint(30, 7);          // Defaults:                  year=2000, style=0
DatePrint(30);             // Defaults:        month=1, year=2000, style=0 (strange, but
valid)
DatePrint();               // Defaults: day=1, month=1, year=2000, style=0
```

If a parameter does not have a default value, then failing to provide an argument generates a compiler error. Ex: Given: void DatePrint(int currDay, int currMonth, int currYear, int printStyle = 0). Then the call DatePrint(30, 7) generates the following error message from g++.

Figure 6.12.3: Compiler error if parameters corresponding to omitted arguments don't have default values.

```
fct_defparm.cpp: In function int main():
fct_defparm.cpp:5: error: too few arguments to function void DatePrint(int, int, int, int)
fct_defparm.cpp:22: error: at this point in file
```

PARTICIPATION ACTIVITY 6.12.1: Function parameter defaults.

Given:

```
void CalcStat(int num1, int num2, int num3 = 0, char usrMethod = 'a') { ... }
```


1) A compiler error will occur because only an int parameter can have a default value.
   - ○ True
   - ○ False

2) The call CalcStat(44, 47, 42, 'b') uses usrMethod = 'a' because the parameter default value of 'a' overrides the argument 'b'.

- True
- False

3) The call CalcStat(44, 47, 42) uses usrMethod = 'a'.

- True
- False

4) The call CalcStat(44, 47, 'b') uses num3 = 0.

- True
- False

5) The following is a valid start of a function definition: `void myFct(int num1 = 0, int num2 = 0, char usrMethod) {`

- True
- False

Exploring further:

- Default arguments from msdn.microsoft.com

CHALLENGE ACTIVITY 6.12.1: Functions with default parameters.

Start

Type the program's output

```
#include <iostream>
using namespace std;

void show(int a, int b, int c = 10) {
   cout << a << ", " << b << ", " << c << endl;
}

int main() {
   show(4, 3, 2);
   show(1, 8);

   return 0;
}
```

```
4, 3, 2
1, 8, 10
```


1 2

Check Next

CHALLENGE ACTIVITY 6.12.2: Return number of pennies in total.

Write a function NumberOfPennies() that returns the total number of pennies given a number of dollars and (optionally) a number of pennies. Ex: 5 dollars and 6 pennies returns 506.

```
1  #include <iostream>
2  using namespace std;
3
4  /* Your solution goes here  */
5
6  int main() {
7     cout << NumberOfPennies(5, 6) << endl; // Should print 506
8     cout << NumberOfPennies(4) << endl;    // Should print 400
9     return 0;
10 }
```


Run

View your last submission

6.13 Function name overloading

Sometimes a program has two functions with the same name but differing in the number or types of parameters, known as ***function name overloading*** or just ***function overloading***. The following two functions print a date given the day, month, and year. The first function has parameters of type int, int, and int, while the second has parameters of type int, string, and int.

Figure 6.13.1: Overloaded function name.

```
#include <iostream>
#include <string>
using namespace std;

void DatePrint(int currDay, int currMonth, int currYear) {
   cout << currMonth << "/" << currDay << "/" << currYear;
}

void DatePrint(int currDay, string currMonth, int currYear) {
   cout << currMonth << " " << currDay << ", " << currYear;
}

int main() {

   DatePrint(30, 7, 2012);
   cout << endl;

   DatePrint(30, "July", 2012);
   cout << endl;

   return 0;
}
```

```
7/30/2012
July 30, 2012
```

The compiler determines which function to call based on the argument types. DatePrint(30, 7, 2012) has argument types int, int, int, so calls the first function. DatePrint(30, "July", 2012) has argument types int, string, int, so calls the second function.

More than two same-named functions is allowed as long as each has distinct parameter types. Thus, in the above program:

- DatePrint(int month, int day, int year, int style) can be added because the types int, int, int, int differ from int, int, int, and from int, string, int.
- DatePrint(int month, int day, int year) yields a compiler error, because two functions have types int, int, int (the parameter names are irrelevant).

A function's return type does not influence overloading. Thus, having two same-named function definitions with the same parameter types but different return types still yield a compiler error.

The use of overloading and of default parameter values may be combined as long as no ambiguity is introduced. Adding the function

`void DatePrint(int month, int day, int year, int style = 0)` above would generate a compiler error because the compiler cannot determine if the function call `DatePrint(7, 30, 2012)` should go to the "int, int, int" function or to that new "int, int, int, int" function with a default value for the last parameter.

PARTICIPATION ACTIVITY 6.13.1: Function name overloading.

Given the following function definitions, select the number that each function call would print. If the function call would not compile, choose Error.

```
void DatePrint(int day, int month, int year) {
   cout << "1" << endl;
}

void DatePrint(int day, string month, int year) {
   cout << "2" << endl;
}

void DatePrint(int month, int day) {
   cout << "3" << endl;
}
```

1) DatePrint(30, 7, 2012);

- 1
- 2
- 3
- Error

2) DatePrint(30, "July", 2012);

- 1
- 2
- 3
- Error

3) DatePrint(7, 2012);

- 1
- 2
- 3
- Error

4) DatePrint(30, 7);

- 1
- 2
- 3

- Error

5) DatePrint("July", 2012);

- 1
- 2
- 3
- Error

Exploring further:

- Overloaded functions from cplusplus.com.

CHALLENGE ACTIVITY 6.13.1: Overload salutation printing.

Complete the second PrintSalutation function to print the following given personName "Holly" and customSalutation "Welcome":

```
Welcome, Holly
```

End with a newline.

```
#include <iostream>
#include <string>
using namespace std;

void PrintSalutation(string personName) {
   cout << "Hello, " << personName << endl;
}

// Define void  PrintSalutation(string personName, string customSalutation)...

/* Your solution goes here  */

int main() {
   PrintSalutation("Holly", "Welcome");
   PrintSalutation("Sanjiv");

   return 0;
}
```


Run

View your last submission

CHALLENGE ACTIVITY 6.13.2: Convert a height into inches.

Write a second ConvertToInches() with two double parameters, numFeet and numInches, that returns the total number of inches. Ex: ConvertToInches(4.0, 6.0) returns 54.0 (from 4.0 * 12 + 6.0).


```
#include <iostream>
using namespace std;

double ConvertToInches(double numFeet) {
   return numFeet * 12.0;
}

/* Your solution goes here  */

int main() {
   double totInches;

   totInches = ConvertToInches(4.0, 6.0);
   cout << "4.0, 6.0 yields " << totInches << endl;

   totInches = ConvertToInches(5.8);
   cout << "5.8 yields " << totInches << endl;
   return 0;
```

Run

View your last submission

6.14 Parameter error checking

Verifying parameter values

Commonly, a function expects parameter values to be within some range. A good practice is to check that a parameter's value is within an expected range. If not in the range, the function might take one or more of various actions, like outputting an error message, assigning a valid value, returning a value indicating failure, exiting the program, etc.

Figure 6.14.1: Function with parameter error checking.

```
#include <iostream>
using namespace std;

void DatePrint(int currDay, int currMonth, int currYear) {

   // Parameter error checking
   if ((currDay < 1) || (currDay > 31)) {
      cout << "Invalid day (" << currDay << "). Using 1." << endl;
      currDay = 1;
   }

   if ((currMonth < 1) || (currMonth > 12)) {
      cout << "Invalid month (" << currMonth << "). Using 1." << endl;
      currMonth = 1;
   }

   // Begin function's normal behavior
   cout << currMonth << "/" << currDay << "/" << currYear;
}


int main() {

   DatePrint(30, 7, 2012);
   cout << endl << endl;

   DatePrint(40, 7, 2012);
   cout << endl << endl;

   DatePrint(30, 13, 2012);
   cout << endl << endl;

   return 0;
}
```

```
7/30/2012

Invalid day (40). Using 1.
7/1/2012

Invalid month (13). Using 1.
1/30/2012
```


PARTICIPATION ACTIVITY 6.14.1: Checking parameter values.

Consider the example above.

1) The month must be in the range from 1 to _____ .

Check Show answer

2)

If the month parameter is not in the valid range, the code outputs an error message, and assigns currMonth with _____ .

Check Show answer

3) In addition to currMonth, which other parameter is checked for being in a valid range?

Check Show answer

6.15 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 6.15.1: Include directives.

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


The following animation illustrates.

PARTICIPATION ACTIVITY 6.15.1: Preprocessor's handling of an include directive.

Animation captions:

1. The preprocessor replaces include by contents of myfile.h during compilation.

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>` -- 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.

- Header files that are part of the standard C++ library do not have a .h extension.
- Items that were originally part of the C standard library have a "c" prepended, as in cmath.

PARTICIPATION ACTIVITY 6.15.2: 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

6.16 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.cpp`.

Figure 6.16.1: Putting related functions in their own file.

main.cpp

threeintsfcts.cpp

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

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

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

```
#include <iostream>
#include "threeintsfcts.h"
using namespace std;

// Normally lots of other code
here

int main() {

   cout << ThreeIntsSum(5, 10,
20) << endl;
   cout << ThreeIntsAvg(5, 10,
20) << endl;

   return 0;
}

// Normally lots of other code
here
```

threeintsfcts.h

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


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

Figure 6.16.2: Compiling multiple files together.

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

`> g++ -Wall main.cpp threeintsfcts.cpp`
`main.cpp: In function  int main():`
`main.cpp:8: error: ThreeIntsSum was not declared in this scope`
`main.cpp:9: error: ThreeIntsAvg was not declared in this scope`
	`> g++ -Wall main.cpp threeintsfcts.cpp`
	`>`

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 6.16.1: Header file guards.

```
#ifndef FILENAME_H
#define FILENAME_H

// Header file contents

#endif
```

Header file guards are preprocessor directives, which 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 6.16.3: All header files should be guarded.


```
#ifndef THREEINTSFCTS_H
#define THREEINTSFCTS_H

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

#endif
```

PARTICIPATION ACTIVITY 6.16.1: Header files.

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

6.17 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 6.17.1: Array parameters in an average test score calculation program.

```
#include <iostream>
#include <vector>
using namespace std;

double CalculateAverage(double scoreVals[], int numVals) {
   int index;
   double scoreSum = 0.0;

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

   return scoreSum / numVals;
}

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

   // Prompt user to enter test scores
   cout << "Enter " << NUM_SCORES << " test scores:" << endl;
   for (index = 0; index < NUM_SCORES; ++index) {
      cout << "Test score: ";
      cin >> testScores[index];
   }
   cout << endl;

   // Call function to calculate average
   averageScore = CalculateAverage(testScores, NUM_SCORES);
   cout << "Average test score: "
        << averageScore << endl;

   return 0;
}
```

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

Average test score: 91.75
```


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.

Functions with array parameters, other than C 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 6.17.1: 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

PARTICIPATION ACTIVITY 6.17.2: 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 ;

   for(i = 0; i <
                          ; ++i) {
      cout << arrayVals[i] <<
endl;
   }
}
```


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

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 6.17.3: 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 6.17.2: Normal and const array parameters in test score adjustment and averaging program.

```
Enter 4 test scores:
Test score: 90.0
Test score: 95.5
Test score: 97.0
Test score: 92.5
Original test scores:  90 95.5
97 92.5
Adjusted test scores:  92 97.5
99 94.5
Average adjusted test score:
95.75
```

```
#include <iostream>
using namespace std;

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

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

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

   for (i = 0; i < numVals; ++i) {
      cout << " " << scoreVals[i];
   }
   cout << endl;
}

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

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

   return scoreSum / numVals;
}

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

   // Prompt user to enter test scores
   cout << "Enter " << NUM_SCORES << " test scores:"
<< endl;
   for (i = 0; i < NUM_SCORES; ++i) {
      cout << "Test score: ";
      cin >> testScores[i];
   }
   cout << endl;

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

   AdjustScores(testScores, NUM_SCORES, 2.0);

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

   // Call function to calculate average
   averageScore = CalculateAverage(testScores,
NUM_SCORES);
   cout << "Average adjusted test score: "
        << averageScore << endl;

   return 0;
}
```


©zyBooks 03/25/21 10:43 488201
xiang zhao
BAYLORCSI14301440Spring2021

PARTICIPATION ACTIVITY 6.17.4: 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 6.17.1: Functions with array parameters.

Start

Type the program's output

Input

```
9
2
6
0
```

Output

```
#include <iostream>
using namespace std;

void PrintScoreVals(const int scoreVals[], int numVals) {
   int i;

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

int main() {
   const int NUM_SCORES = 4;
   int quizScores[NUM_SCORES];
   int i;

   for (i = 0; i < NUM_SCORES; ++i) {
      cin >> quizScores[i];
   }

   PrintScoreVals(quizScores, NUM_SCORES);

   return 0;
}
```

```
9
2
6
0
```


1 2 3

Check Next

CHALLENGE ACTIVITY 6.17.2: 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}.

```
1  #include <iostream>
2  using namespace std;
3
4  /* Your solution goes here  */
5
6  int main() {
7     const int SORT_ARR_SIZE = 4;
8     int sortArray[SORT_ARR_SIZE];
9     int i;
10    int userNum;
11
12    for (i = 0; i < SORT_ARR_SIZE; ++i) {
13       cin >> userNum;
14       sortArray[i] = userNum;
15    }
16
17    SwapArrayEnds(sortArray, SORT_ARR_SIZE);
18
```


Run

Run

View your last submission

6.18 Functions with array parameters: Common errors

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 | 6.18.1: 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 first 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 | 6.18.2: 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;
   int minScore;

   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;
   int maxScore;

   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;
   int maxScore;

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

©zyBooks 03/25/21 10:43 488201
xiang zhao
BAYLORCSI14301440Spring2021

```
#include <iostream>
#include <cmath>

using namespace std;

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

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

   // Print array before function call
   cout << "tempChanges array before function call: ";
   for(i = 0; i < NUM_VALUES; ++i) {
      cout << tempChanges[i] << " ";
   }
   cout << endl;

   // Find the largest temperature change, and print result.
   maxAbsChange = FindMaxAbsValueIncorrect(tempChanges, NUM_VALUES);
   cout << "Max absolute temperature change: " << maxAbsChange << endl;

   // Print array after function call
   cout << "tempChanges array after function call:  ";
   for(i = 0; i < NUM_VALUES; ++i) {
      cout << tempChanges[i] << " ";
   }
   cout << endl;

   return 0;
}
```



```
tempChanges array before function call: 10 0.5 -5.1 -11.2 3
Max absolute temperature change: 11.2
tempChanges array after function call:  10 0.5 5.1 11.2 3
```


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

```
#include <iostream>
#include <cmath>

using namespace std;

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

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

   return maxAbsVal;
}


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

   // Print array before function call
   cout << "tempChanges array before function call: ";
   for(i = 0; i < NUM_VALUES; ++i) {
      cout << tempChanges[i] << " ";
   }
   cout << endl;

   // Find the largest temperature change, and print result.
   maxAbsChange = FindMaxAbsValue(tempChanges, NUM_VALUES);
   cout << "Max absolute temperature change: " << maxAbsChange << endl;

   // Print array after function call
   cout << "tempChanges array after function call:  ";
   for(i = 0; i < NUM_VALUES; ++i) {
      cout << tempChanges[i] << " ";
   }
   cout << endl;

   return 0;
}
```


©zyBooks 03/25/21 10:43 488201
xiang zhao
BAYLORCSI14301440Spring2021

```
tempChanges array before function call: 10 0.5 -5.1 -11.2 3
Max absolute temperature change: 11.2
tempChanges array after function call:  10 0.5 -5.1 -11.2 3
```

PARTICIPATION ACTIVITY 6.18.3: 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

6.19 C++ example: Salary calculation with functions

zyDE 6.19.1: 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 funct 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 function to input the annual salary.
3. Run the program again with the same annual salaries as above. Are results the sa

Load default templ

```
#include <iostream>
#include <vector>
using namespace std;

double GetCorrespondingTableValue(int search, vector<int> baseTable, vector<dou
   int baseTableLength;
   double value;
   int i;
   bool keepLooking;

   baseTableLength = baseTable.size();
   i = 0;
   keepLooking = true;

   while ((i < baseTableLength) && keepLooking) {
      if (search <= baseTable.at(i)) {
         value = valueTable.at(i);
         keepLooking = false;
```


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.

zyDE 6.19.2: Calculate salary: Using function (solution).

Load default templ

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

// Function to prompt for and input an integer
int PromptForInteger(const string userPrompt) {
   int inputValue;

   cout << userPrompt << ": " << endl;
   cin >> inputValue;

   return inputValue;
}

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

// Function to get a value from one table based on a range in the other table
```

```
60000 40000 1000000
-1
```

Run

6.20 C++ example: Domain name validation with functions

zyDE 6.20.1: 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, .org, or .info. A ***restricted top-level domain*** is a TLD that is either .biz, .name, or .pro. A ***s 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 th 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.con 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.

Load default templ

```
#include <iostream>
#include <cctype>
#include <vector>
#include <string>
using namespace std;

const int MAX_NUMS = 4;  // Global variable used for vector sizes

// ***************************************************************************
// GetPeriodPosition - Pass a string and return the position of the period
// ***************************************************************************
int GetPeriodPosition(string stringToSearch) {
   int periodCounter;
   int periodPosition;
   unsigned int i;

   periodCounter = 0;
   periodPosition = -1;
```



```
apple.com
APPLE.com
apple.comm
```

Run

zyDE 6.20.2: Validate domain names with functions.

A solution to the above problem follows.

Load default templ

```
#include <iostream>
#include <cctype>
#include <vector>
#include <string>
using namespace std;

const int MAX_NUMS = 4;  // Global variable used for vector sizes

// ***************************************************************************
// GetPeriodPosition - Pass a string and return the position of the period
// ***************************************************************************
int GetPeriodPosition(string stringToSearch) {
   int periodCounter;
   int periodPosition;
   unsigned int i;

   periodCounter = 0;
   periodPosition = -1;
```

```
apple.com
APPLE.com
apple.comm
```

Run

6.21 LAB: Flip a coin

Write a program that simulates flipping a coin to make decisions. The input is how many decisions are needed, and the output is either heads or tails. Assume the input is a value greater than 0.

Ex: If the input is:

```
3
```

the output is:


```
heads
tails
heads
```

For reproducibility needed for auto-grading, seed the program with a value of 2. In a real program, you would seed with the current time. In that case, every program's output would be different, which is what is desired but can't be auto-graded.

Note: A common student mistake is to call srand() before each call to rand(). But seeding should only be done once, at the start of the program, after which rand() can be called any number of times.

Your program must define and call the following function that randomly picks 0 or 1 and returns "heads" or "tails". Assume the value 0 represents "heads" and 1 represents "tails".

`string HeadsOrTails()`

LAB ACTIVITY | 6.21.1: LAB: Flip a coin | 10 / 10

main.cpp — Load default template...

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;

string HeadsOrTails() {
   string ret;

   if (rand() % 2 == 0) {
      ret = "heads";
   }
   else {
      ret = "tails";
   }

   return ret;

}
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first

box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

3

Run program

Program output displayed here

Signature of your work What is this?

```
3/25.. R--|10 ..3/25
```

6.22 LAB: Exact change - functions

Write a program with total change amount as an integer input that outputs the change using the fewest coins, one coin type per line. The coin types are dollars, quarters, dimes, nickels, and pennies. Use singular and plural coin names as appropriate, like 1 penny vs. 2 pennies.

Ex: If the input is:

```
0
```

or less, the output is:

```
no change
```

Ex: If the input is:


```
45
```

the output is:

```
1 quarter
2 dimes
```

Your program must define and call the following function. Positions 0-4 of coinVals should contain the number of dollars, quarters, dimes, nickels, and pennies, respectively.

```
void ExactChange(int userTotal, vector<int>& coinVals)
```

LAB ACTIVITY | 6.22.1: LAB: Exact change - functions | 10 / 10

main.cpp

Load default template...

```
#include <iostream>
#include <vector>
using namespace std;

void ExactChange(int userTotal, vector<int>& coinVals) {
   coinVals[0] = userTotal / 100;
   userTotal = userTotal % 100;

   coinVals[1] = userTotal / 25;
   userTotal = userTotal % 25;

   coinVals[2] = userTotal / 10;
   userTotal = userTotal % 10;

   coinVals[3] = userTotal / 5;
   userTotal = userTotal % 5;

   coinVals[4] = userTotal;
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

33

Run program

Input (from above) → **main.cpp** (Your program) → Outp

Program output displayed here

Signature of your work What is this?

```
3/25.. R--|0-|0|0|0|7|9|10 ..3/25
```

6.23 LAB: Output values below an amount - functions

Write a program that first gets a list of integers from input. The input begins with an integer indicating the number of integers that follow. Then, get the last value from the input, and output all integers less than or equal to that value.

Ex: If the input is:

```
5 50 60 140 200 75 100
```

the output is:

```
50 60 75
```

The 5 indicates that there are five integers in the list, namely 50, 60, 140, 200, and 75. The 100 indicates that the program should output all integers less than or equal to 100, so the program outputs 50, 60, and 75. For coding simplicity, follow every output value by a space, including the last one.

Such functionality is common on sites like Amazon, where a user can filter results.

Write your code to define and use two functions:
`vector<int> GetUserValues(vector<int>& userValues, int numValues)`
`void OutputIntsLessThanOrEqualToThreshold(vector<int> userValues, int upperThreshold)`

Utilizing functions helps to make main() very clean and intuitive.

LAB ACTIVITY | 6.23.1: LAB: Output values below an amount - functions | 10 / 10

main.cpp — Load default template...

```
#include <iostream>
#include <vector>
using namespace std;

vector<int> GetUserValues(vector<int>& userValues, int numValues) {
   userValues.resize(numValues);
   for (int i = 0; i < numValues; i++) {
      cin >> userValues.at(i);
   }

   return userValues;
}
```

```
15 void OutputIntsLessThanOrEqualToThreshold(vector<int> userValues, int upperThreshold) {
16    for (unsigned int i = 0; i < userValues.size(); i ++) {
17       if (userValues.at(i) <= upperThreshold)
18          cout << userValues.at(i)  << " ";
```

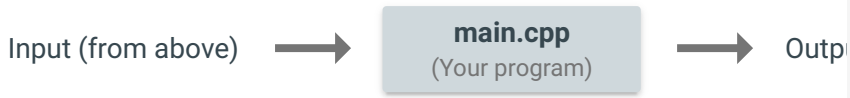
Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

5 50 60 140 200 75 100

Run program



Program output displayed here

Signature of your work What is this?

```
3/25.. R---|4|4|10|10 ..3/25
```