CSC2212

C++ Programming

Functions

Ch 6: Gaddis, Starting Out with C++ Early Objects 8th ed

Lab five

Modular Programming

- ***Modular programming***: breaking a program up into smaller, manageable functions or modules. Supports the divide-and-conquer approach to solving a problem.

- ***Function***: a collection of statements to perform a specific task

Modular Programming

- **Motivation for modular programming**
  - Simplifies the process of writing programs
  - Improves maintainability of programs

Defining and Calling Functions

- **Function definition**: a statements that make up a function

- **Function call**: a statement that causes a function to execute

Function Definition

- Definition includes:

  name: name of the function. Function names follow same rules as variable names

  parameter list: variables that hold the values passed to the function

  body: statements that perform the function's task

  return type: data type of the value the function returns to the part of the program that called it

Function Definition *(example 1)*

Return type

Name

Parameter list (This one is empty)

Body

```
int main ()
{
   cout << "Hello World\n";
   return 0;
}
```

Function Definition *(example 2)*

```
double cube( double a){
  double b = a * a* a;
  return b;
  }
```

Function Header

- **Function header** consists of
  - the function *return type*
  - the function *name*
  - the function *parameter list*

- Example:

```
int main()
```

- *Note: no* `;` *at the end of the header*

Function Return Type

- If a function returns a value, the type of the value must be indicated

```
int main()
```

- If a function does not return a value, its return type is **`void`**

```
void printHeading()
{
   cout << "\tMonthly Sales\n";
}
```

Calling a Function

- To call a function, use the function name followed by `()` and `;`

  Example: `printHeading();`

- When a function is called, the program executes the body of the function.

- After the function terminates, execution resumes in the calling module at the point of call

Calling a Function *(cont.)*

- `main` is automatically called when the program starts

- `main` can call any number of functions

- Functions can call other functions

Function Prototypes

The compiler must know the following about a function before it is called

- name
- return type
- number of parameters
- data type of each parameter

Function Prototypes *(cont.)*

Ways to notify the compiler about a function before a call to the function:

1. Place function definition before calling function's definition
2. Use a **function prototype** (similar to the heading) of the function
   - Heading: `void printHeading()`
   - Prototype: `void printHeading();`

– *Function prototype is also called a* ***function declaration***

Function Prototype Notes

- Place prototypes near top of program.
- Program must include either prototype or full function definition before any call to the function, otherwise a compiler error occurs.
- When using prototypes, function definitions can be placed in any order in the source file. Traditionally, **`main`** is placed first.

Sending Data into a Function

- Can pass values into a function at time of call

```
c = sqrt(a*a + b*b);
```

- Values passed to function are **arguments**

- Variables in function that hold values passed as arguments are **parameters**

Parameters, Prototypes, & Function Headings

- For each function argument,
  - the **prototype** must include the data type of each parameter in its **()**

```
Example: void evenOrOdd(int);  //prototype
```

  - the **heading** must include a declaration, with variable type and name, for each parameter in its **()**

```
Example: void evenOrOdd(int num);  //heading
```

The function call for the above function would look like this:

```
evenOrOdd(val);  //call
```

Note: no data type on argument in call

Calling Functions with Multiple Arguments

When calling a function with multiple arguments

- the number of arguments in the call must match the function prototype and definition

- the first argument will be copied into the first parameter, the second argument into the second parameter, etc.

Calling Functions with Multiple Arguments Illustration

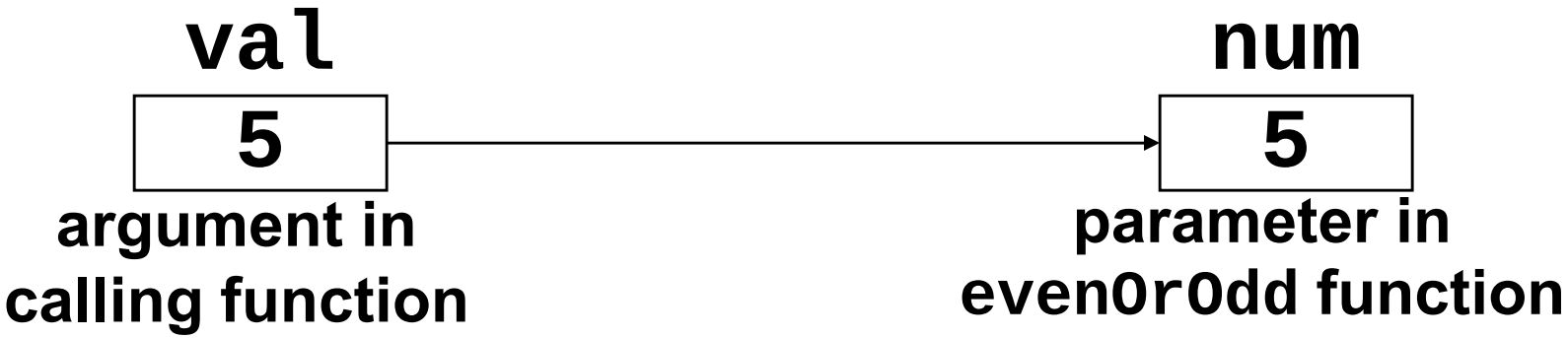
```
displayData(height, weight);  // call


void displayData(int h, int w)// heading
{
   cout << "Height = " << h << endl;
   cout << "Weight = " << w << endl;
}
```

Passing Data by Value

- **Pass by value**: when an argument is passed to a function, a copy of its value is placed in the parameter
- The function cannot access the original argument
- Changes to the parameter in the function do not affect the value of the argument in the calling function

Passing Data by Value *(Example)*

- Example: `int val = 5;`
  `evenOrOdd(val);`



- **`evenOrOdd`** function can change variable **`num`**, but it will not have effect on variable **`val`.**

Passing Data by Reference

- Mechanism that allows a function to work with the original argument from the function call, not a copy of the argument
- Allows the function to modify values stored in the calling environment
- Provides a way for the function to 'return' more than 1 value

Reference Variables

- A reference variable is an alias for another variable
- It is defined with an ampersand (**&**) in the prototype and in the header

```
void getDimensions(int&, int&);
```

- Changes to a reference variable are made to the variable it refers to
- Use reference variables to implement passing parameters by reference

Pass by Reference Example

```
void squareIt(int &);        //prototype

void squareIt(int &num)  //definition
{
    num *= num;
}

int localVar = 5;
squareIt(localVar);          // Call

      // localVar now contains 25
```

Reference Variable Notes

- Each reference parameter must contain **&**
- Argument passed to reference parameter must be a variable. It cannot be an expression or a constant.
- Use only when appropriate, such as when the function must input or change the value of the argument passed to it
- Files (*i.e.*, file stream objects) should be passed by reference

The `return` Statement

- Used to end execution of a function
- Can be placed anywhere in a function
  - *Statements that follow the* **`return`** *statement will not be executed*
- Can be used to prevent abnormal termination of program
- Without a **`return`** statement, the function ends at its last **`}`**

Returning a Value from a Function

- **`return`** statement can be used to return a value from the function to the module that made the function call
- Prototype and definition must indicate data type of return value (not **`void`**)
- Calling function can use the return value, *e.g.*,
  - assign it to a variable
  - send it to **`cout`**
  - use it in expression

Returning a Value
the return Statement

- Format: **`return`** ***`expression;`***

- ***`expression`*** may be a variable, a literal value, or an expression.

- ***`expression`*** should be of the same data type as the declared return type of the function (will be converted if not)

Boolean `return` Example

- Function can return **`true`** or **`false`**

```
bool isValid(int);          // prototype

bool isValid(int val)       // heading
{  int min = 0, max = 100;
   if (val >= min && val <= max)
      return true;
   else
      return false; }

if(isValid(score))       // function call
   …
```

Local Variable Lifetime

- A local variable only exists while its defining function is executing
- Local variables are destroyed when the function terminates

Local and Global Variables

- **global variable**: a variable defined outside all functions; it is accessible to all functions within its scope
- Easy way to share large amounts of data between functions
- Scope of a global variable is from its point of definition to the program end

Static Local Variables

- **Local variables**
  - Only exist while the function is executing
  - Are redefined each time function is called
  - Lose their contents when function terminates
- **`static` local variables**
  - Are defined with key word **`static`**

```
static int counter;
```

  - Are defined and initialized only the first time the function is executed
  - Retain their contents between function calls

Default Arguments

- Values passed automatically if arguments are missing from the function call
- Must be a constant declared in prototype or header (whichever occurs first)

```
void evenOrOdd(int a = 0);
```

- Calling the function:
  - `evenOrOdd();`
  - `evenOrOdd(3);`

Default Arguments *(cont.)*

- Multi-parameter functions may have default arguments for some or all parameters

```
int getSum(int, int=0, int=0);
```

- If not all parameters to a function have default values, the ones without defaults must be declared first in the parameter list

```
int getSum(int, int=0, int=0);// OK
int getSum(int, int=0, int);  // wrong!
```

Default Arguments *(cont.)*

- When an argument is omitted from a function call, all arguments after it must also be omitted

```
sum = getSum(num1, num2);     // OK

sum = getSum(num1, , num3);   // wrong!
```

Exercise 1

Write a simple program that can use function to find the biggest number among two numbers.

Overloading Functions

- Overloaded functions are two or more functions that have the same name, but different parameter lists
- Can be used to create functions that perform the same task, but take different parameter types or different number of parameters
- Compiler will determine which version of the function to call by the argument and parameter list

Overloaded Functions *Example*

If a program has these overloaded functions,

```
void getDimensions(int);              // 1
void getDimensions(int, int);         // 2
void getDimensions(int, float);       // 3
void getDimensions(double, double);// 4
```

then the compiler will use them as follows:

```
int length, width;
double base, height;
getDimensions(length);                // 1
getDimensions(length, width);         // 2
getDimensions(length, height);        // 3
getDimensions(height, base);          // 4
```

The `exit()` Function

- Terminates execution of a program
- Can be called from any function
- Can pass a value to operating system to indicate status of program execution
- Usually used for abnormal termination of program
- Requires **`cstdlib`** header file
- Use with care

Exercise 2

Modify Exercise 1, with an overloaded function such that it will be able to take two or three numbers.

Exercise 3

Modify the program such that it will be able to take up to ten(10) numbers, and return the largest among the them. (Hint: use array)