User-defined Functions

Larry Caretto
Computer Science 106

Computing in Engineering and Science

Spring 2005

California State University Northridge

Outline

- Writing and calling a function
  - Header and body
  - Function prototype
  - Passing information to a function
  - Returning values in the function name
- `void` functions with no return value
- Use pass-by-reference to change variables in the calling program
- Variable scope and global variables

Introduction to Functions

- Library functions like `pow`, `atan` and `sqrt` used previously
- Statement to set $x = yz^3$:
  - `x = y * pow( z, 3 );`
  - Note order of arguments important; the call `pow( 3, z )` gives 3^z
  - Use #include <cmath> for this function
- You can write your own functions
  - Why do we write functions?
  - How do we write code for functions?

Why do we write functions?

- First use of functions was for code like mathematical function calculations
  - Specialized calculation done repeatedly
  - Want to write code only one time
  - Want to be able to pass values of parameters to code and get value back
- As programs got more complex, breaking code into functions provided a way to organize complex code

How do we write functions?

- C++ code is a collection of functions
- Each function, including main, has the same level of importance
  - Close code for each function before starting a new function

```
int main()
{       // body of main
}
int myFunction( …… )
{       // body of myFunction
}
```


Operation of Functions

- Any function (the caller) can call another function by using the name of the function being called in an expression
- The statement calling the function sends information from the caller
- Execution control is transferred to the function being called
- The function being called returns control and (usually) results to the caller

Operation of Functions

- The function being called uses the information it receives to do a set of calculations or procedures
- In the usual case, the function being called returns a result to the caller in the location of the function name
- Example: d = pow(b,2) – 4 * a * c;
  - calls the pow function with values of b and 2
  - and the result b^2 is returned in function name, pow, for further use in an expression

Writing and Using Functions

- Organize the program into individual functions that are called by main
  - Simple example: main calls three functions: (1) input function, (2) calculation function and (3) output funciton
- Write code for each function (and main)
  - Write function header to specify information received from calling function
  - Write function body to calculate results and return them to "calling" function
- Write function calls to exchange data

Writing and Using Functions II

- Library functions, such as pow(x, y) to compute x^y, transfer information based on the order of the variables
- This is true for user-defined functions as well
  - Information transferred from a list of variables in the calling function to a list of variables in the function called
  - Correspondence based on order of variables in function header and statement calling the function

Function Basics

- Each function has a header and a body
  - Header specifies
    - Name of function
    - Type of value returned by the function name
    - List of variables in the function whose values are determined by the calling program
  - Body gives code executed by the function
- Function prototypes at start of code provide information to compiler
  - Same as header except a semicolon is added at the end
  - Can omit variable names

Function Basics II

- Example of header and body

```
double myPow ( double number,
  double power )  // header
{
    // Body,in braces contains
    // actual code for function
}
```

- Header defines information that will be received from calling function

Function Header

- Header has following syntax:
  - ***<type>*** ***<name>*** (***<argument list>***)
  - ***<type>*** specifies the type of value returned by the function
  - ***<name>*** is the name you choose for your function; this name is used to call the function from another function
  - ***<argument list>*** specifies type and names of variables in function whose values come from the calling program
  - There is no semicolon in the function header

Function Header II

- Example of function, `myPow`, a user-defined function to replace `pow`
  - Pass the number and the power to the function as type double
  - Return the result as type double
  - *General header syntax from previous chart*
    <type> <name> (***<argument list>***)
  - `double myPow ( double number, double power )`

California State University Northridge 13

Argument List

- Example on previous chart had two parameters in argument list
  - ***<type> <name>*** (***<argument list>***)
  - `double myPow ( double number, double power )`
- Function will use `number` and `power` as type double variables
- Values for these variables set by other function that calls (uses) `myPow`

California State University Northridge 14

Passing arguments

- Based on order of arguments in function header and in calling statement
- Recall the library pow function was called as pow(number, power)
  - pow(3,4) = 3^4 but pow(4,3) = 4^3
  - What is result of following code

```
double number = 3, power = 4;
cout << pow( power, number)
```

Result is 4^3; only the order counts!

California State University Northridge 15

The `return` Statement

- We have used this statement in main as `return EXIT_SUCCESS;`
- The general syntax of this statement is `return` ***<value>***;
- ***<value>*** may be a constant, a variable or an expression
- This is value returned to calling program in function name
- `return` always transfers control

California State University Northridge 16

Organization of Function Code

```
double myPow( double number,
              double power )
{
    double result = exp( power *
                    log( number ) );
    return result;
}
```

- Place following prototype at top of code

```
double myPow( double number,
              double power );
```

California State University Northridge 17

Alternative Function Code

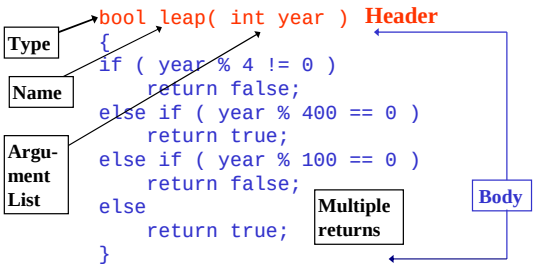
```
double myPow( double number,
              double power )
{
    return = exp( power *
                    log( number ) );
}
```

- Can use following prototype without variable names at top of code

```
double myPow( double, double );
```

California State University Northridge 18

Another function Example



Use of bool leap(int year)

```
bool leap ( int year );// prototype
int main()  // examples of use
{  cout << "Enter a year: ";
   int y; cin >> y;
   bool cond = leap( y );
   if ( leap( y ) ) {…}
   if ( leap( y ) && month == 2 ) {…}
   return EXIT_SUCCESS
}
// leap and other functions go here
```


Exercise

- Write a function that takes two type int arguments and returns their difference
- Use this function to compute 3 – 5
- Write the prototype

```
int diff( int a, int b)
{    return a – b }
```

//use: `cout << diff( 3, 5 );`

// prototype: `int diff( int a, int b);`

Exercise

- Write a function that takes two type double arguments and returns their quotient
- Use this function to compute 5/3
- Write the prototype

```
double div( double a, double b)
{    return a / b }
```

//use: `cout << div( 5, 3 );`

// prototype: `double div(double a, double b);`

`void` functions with no `return`

- The type void used for functions that do not return a value
- Example: error message function

```
void printError( int code )
{
  if ( code == 1)
    cout << "Type one error\n";
  else if ( code == 2 )
    cout << "Error two is …
  else if // additional code
}  // no return needed here
```


The `return` Statement

- The `return` statement returns control and a value to the calling program
  - Functions, other than void functions use the syntax `return` ***<value>*** to return a value to the calling function in the function name
  - Void functions may have a simple `return` statement without a value to return control to the calling program at some point before the end of the function
- Functions may have more than one return statement
  - `return` transfers control immediately

Empty Argument List

- If a function does not need any values from the calling program an empty set of parentheses is required
- Example is function with several output statements to describe purpose of code

```
void describeCode()
{
    cout << "This code …..
    cout << "Still more output
  // No return needed for type void
}
```


Kinetic Energy Function

- Write a function that takes two type double parameters, mass and velocity and computes kinetic energy = $mV^2/2$

```
double KE( double m, double V )
{
    return m * V * V / 2;
}
```

- Possible calls to this function

```
totalE = KE( 4, 3 ) + PE;
```

- What is result of this call

```
result = 50 + KE( 5, 2 );
```

$50 + 5 * 2^2 / 2 = 60$

Kinetic Energy Function II

```
double KE( double m, double V )
{
    return m * V * V / 2;
}
```

- What is output from these calls?

```
double mass = 5, velocity = 2;
cout << KE( velocity, mass );
double e = PE + KE( 2*pow( velocity,
                    2), velocity);
double total = KE( mass * velocity,
                                mass );
```

10

e = KE($2*2^2$,2) = KE(8,2)=$8*2^2/2$ = 16

total = KE(5*2,5) = KE(10,5)=$10*5^2/2$ = 125

Data Validation Function

- The function `getValidInt( int xMin, int xMax,string name )` does the following tasks
  - Prompts the user for an input variable (named in the string passed in the third parameter) within a range defined by the first and second parameters
  - Gets the input from the user
  - Tells the user if there is an error and gets new input from the user in this case
  - Returns valid input to the calling function

Data Validation Function II

- The function `getValidInt` described on the previous chart is used in exercise seven and project one
- Examples of its use

```
int month = getValidInt( 1, 12,
  "month");
int mayDay = getValidInt( 1, 31, "day
  of the month" );
int year = getValidInt( 1901, 2000,
  "year in the 20th century" );
```



```
int getValidInt( int xMin, int xMax,
                 string name )
{
     // Function used to input integer
     // data within a stated range
     // Example function call to input a
     // value for a variable named
     // hour with range between 0 and 23:
     //   int hour =
     //     getValidInt( 0, 23, "hour" );

    int x;         // Input data value
    bool badData;  // Bad data flag
```

// continued on next chart

```
   do    // Loop until user data in range
   {
     cout << "Enter a value for " << name
          << " between " << xMin << " and "
          << xMax << ": ";
     cin >> x;
     badData = x < xMin || x > xMax;
     if ( badData )
        // error message code on next chart
   }
   while ( badData );

}  // end of function
```

```
   if ( badData )
                // print error message
   {
       cout << "\n\nIncorrect data; you "
            << "entered " << name
            << " = " << x << "\n"
            << name << "must be between "
            << xMin << " and " << xMax
            << " Reenter the data now.\n";
   }
```

getValidInt Screen Results

- Call to getValidInt

```
int mayDay = getValidInt( 1, 31, "day
  of the month" );
```

- Screen prompt showing parameters and user input

```
Enter a value for day of the month
  between 1 and 31: 0
Incorrect data; you entered day of the
  month = 0.  day of the month must be
  between 1 and 31. Reenter the data
  now.
Enter a value for day of the month
  between 1 and 31: 1
```


Program Structure Example

- Next chart: example for getValidInt
- Function prototype before program that uses function
- In this example main calls function
- Complete code for main is written start of code for function
- A call to the function transfers control to function with values in the function header variables from the caller
- Function returns value to main


```
   int getValidInt( int, int, string);
            // prototype above
   int main()
   {
        int month = getValidInt( 1, 12,
                      "month" );
        ……..  // other code
        return EXIT_SUCCESS
   }
   int getValidInt( int xMin, int xMax,
                    string name )
   {    int x;
        …….. // other code
        cin >> x;
        …….. // other code
        return x;
   }
```

Passing Information to Functions

- Parameters in function header: formal parameters or dummy parameters (also called formal or dummy arguments)
- Values sent to function by calling program: actual parameters or actual arguments
- Pass by value is default process: when a function is called a copy of the value of the argument is passed to the function

More on Information to Functions

- In pass-by-value, the values of the actual arguments in the calling program are not changed
- The alternative to pass by value is pass by reference
  - The memory address of the actual parameter is passed to the function
  - Changes to the dummy parameter in the function change the actual parameter in the calling program

Pass-by-Value Example

```
//calling program
double x = 10, y = 2;
cout << "fake = " << fake( x, y );
cout << ", x = " << ", y = " << y;
        // what is printed?
//function
double fake( double x, double y )
{
    x +=10; y *= x;  return 3 * y;
}
```


Pass-by-value Operation

- The code on the previous chart does not change the x and y values in the calling program
- Only values of x and y from the calling program are passed to the function
- Functions cannot changed values of variables that are passed by value
- How do we use pass by reference to change the values of parameters passed into a function?

Pass-by-reference

- To use pass by reference place an ampersand (&) between the type and the parameter name in the function header: `int f1( int& x, int& y )`
  - Not a preferred programming style
  - Used only when we have to change more than one parameter (*e.g.*, input routine, vector components, *etc.*)
  - Exercise seven uses an input function which must have pass by reference

Pass-by-reference II

- Default is pass-by-value where changes to parameters do not affect variables in the calling program

```
double fake1 ( int x, double y )
{  x++;  y += x;  return x * y; }
```

- Ampersand (&) gives pass by reference that changes program variables

```
double fake2 ( int& x, double& y )
{  x++;  y += x;  return x * y; }
```


Pass-by-Value Example

```
//calling program segment
double u = 5, v = 2;
cout << "fake = " << fake( u, v );
cout << "\nu =" << u << ", v =" << v;
   // what is printed?
//function
double fake( double x, double y )
{
    x +=10; y *= x;  return 3 * y;
}
```

fake = 90
u = 5, v = 2

x = 5 + 10 = 15 y = 2 * 15 = 30 fake(u, v) = 3 * 30 = 90

Pass-by-Reference Example

```
//calling program segment
double u = 3, v = 4;
cout << "fake = " fake( u, v );
cout << "\nu =" << u << ", v =" << v;
     // what is printed?
//function
double fake(double& x, double& y )
{
    x +=10; y *= x;  return 3 * y;
}
```

fake = 156
u = 13, v = 52

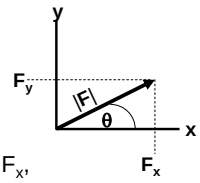
x = 3 + 10 = 13 y = 13 * 4 = 52 fake(u, v) = 3 * 52 = 156

Pass-by-Reference Example II

```
double u = 3, v = 4;
cout << "fake = " fake( u, v );
cout << "\nu =" << u << ", v =" << v;
double fake( double& x, double& y )
{ x +=10; y *= x;  return 3 * y; }
// at start fake has x = 3, y = 4
// fake code sets x = x + 10 = 13
// and y = y * x = 4 * 13 = 52
// fake returns 3 * 52 = 156 and
// changes u to 13 and v to 52
```


Example: Converting Vectors

- Convert different representations of two-dimensional vectors
  - Polar: magnitude, |**F**| and direction, θ
  - Rectangular: components, F_x, and F_y, along the x and y axes
- Conversion equations
  - $|\mathbf{F}| = (F_x^2 + F_y^2)^{1/2}$, $\theta = \tan^{-1}(F_y / F_x)$
  - $F_x = |\mathbf{F}|\cos\theta$, $F_y = |\mathbf{F}|\sin\theta$,



Converting Vectors II

- Write two functions to convert between the two different representations
  - Polar to rectangular function has magnitude and direction as inputs and returns x-component and y-component
  - Rectangular to polar function has x-component and y-component as inputs and returns magnitude and direction
  - Use atan2 function for $\theta = \tan^{-1}(F_y / F_x)$ to get full 2π result ($-\pi/2 < \text{atan result} < \pi/2$)

Converting Vectors III

- Each function has two input values and computes two results
- Use pass by reference to get results back to calling program
- Inputs to function are pass by value
- Function type can be void since function name need not return a value
  - Functions using pass by reference to return values sometimes return an error code in the function name

Converting Vectors IV

```
void polarToRectangular (
    double magnitude, double angle,
    double& xComponent,
    double& yComponent )
{
   xComponent = magnitude
              * cos( angle );
   yComponent = magnitude
              * sin( angle );
}
```

Converting Vectors V

```
void rectangularToPolar (
    double xComponent,
    double yComponent,
    double& magnitude, double& angle )
{
   magnitude = sqrt(
               pow( xComponent, 2 ) +
               pow( yComponent, 2 ) );
   angle = atan2 ( yComponent,
                   xComponent );
}
```

California State University Northridge 49

Use of Conversion Functions

```
const double PI = 4 * atan(1);
double x = 3, y = 4;
double A, theta;
rectangularToPolar(x, y, A, theta );
cout << x << "    " << y << "    "
     << A << "    " << theta << endl;
double size = 10, direction = PI / 8;
double xComp, yComp;
polarToRectangular( size, direction,
                    xComp, yComp );
cout << size << "  " << direction <<
      "  " << xComp << "  " << yComp;
```

California State University Northridge 50

Pass-by-Reference Exercise

- Write an input function that prompts the user to enter two type double variables, x and y, and returns these values to the calling program

```
void input( double& x, double& y)
{
    cout << "Enter x and y: ";
    cin >> x >> y;
}
```

California State University Northridge 51

Pass-by-Reference Exercise

```
void input( double& x, double& y)
{
    cout << "Enter x and y: ";
    cin >> x >> y
}
```

- Write the prototype for this function and a call to the function to get x and y

```
void input( double& x, double& y );
double x, y;
input ( x, y );
```

Optional

California State University Northridge 52

Use of Pass by Reference

- Calling programs use same approach for pass by reference and pass by value
- Variable or expression is placed as one of the arguments to the function
- Do not use a constant in a function call unless it is passed by value
- Data types (and ampersands for pass by reference) are not used in function call

California State University Northridge 53

Scope of a Variable

- Scope of a variable is the part of program that can use the variable
- We see that we can have the same variable name in different functions
- These names, although the same, occupy two different memory locations in the computer and are not related
- Even within a single function we can limit the part of a function in which a variable is in scope (exists)

California State University Northridge 54

Background

- All variables must be declared (given a type) before they are used
- Variables can be declared given a value when declared or later in the code
- Usually assign a value before first use
- Scope refers only to declaring a variable, not to assigning it a value
  - This is just a reminder that we have to initialize variables as well as declare them

Basic Rule for Scope

- A variable defined in a set of braces only exists within those braces
- It can be used anywhere in the program below its initial declaration
  - This includes sets of braces that are opened below the initial declaration
- After close of brace where variable is declared, the variables "goes out of scope" it cannot be used

Example of Scope

```
double x;
if ( c == 4 )
{
    x = 12;
    double y = 2;  // limited scope
}
cout << x << "  " << y;
// statement above will give syntax
// error; y is not defined here
```


Another example of Scope

```
double y = 0, c = 4;
if ( c == 4 )
{
   double y = 2;  // different var-
        // iable with limited scope
}
cout << "y = " << y;
// statement above will print y = 0
// from initial declaration of y
```


Where to Declare Variables

- Current programming practice declares variables as close to first time of use as possible
- May have to be declared earlier in the code to give appropriate scope
  - First use of variable may be inside a loop
  - We must declare it prior to the loop if we want to use if following the loop

Another Example

- Code below will not work because yesNo goes out of scope after closing brace

```
do
{       // program code
   cout << "Another run(Y/N)? ";
   char yesNo; // bad location
   cin >> yesNo;
}
while( yesNo == 'Y' || yesNo == 'y' );
```

Another Example Corrected

- Code below works because yesNo is declared before brace opening the loop

```
char yesNo; // correct location
do
{        // program code
   cout << "Another run(Y/N)? ";
   cin >> yesNo;
}
while( yesNo == 'Y' || yesNo == 'y' );
```


Global Variables

- Global variables have scope of more than one function
  - Declared outside function boundaries
  - Have scope of all functions from declaration to end of file
  - Usually declared at top of program to be present in all functions
  - Considered bad programming practice
  - Use only when variable must be accessed by several functions or there are problems in passing the variable

Trace Global Variables

- What is program output?

```
int status = 0;   // global
int main() {
   cout << status << "   ";
   f1();   f2();
   cout << " " << status // more
}
void f1() { status = 1; }
void f2() {cout << status << endl;}
```

- Program output is 0 1 1

Project Two Global Variables

- In project two the main function calls a function which calls a third function
- We want to get data from main to the third function
- We do not want to rewrite the second function, but it does not allow us to pass the necessary information
- Use global variables to get the information from main to third function

Summary

- Use functions to organize code
- Elements of a function
  - Header with type, name, and argument list
  - Body with code that function executes
  - Statement to return information through function name in calling program must be included in function body
  - Prototype at start of program which is header with a semicolon
- Function name calls function and returns value

Summary Continued

- Pass information to function through argument list in function header
  - Correspondence by position of arguments in header and position of arguments in calling function
  - Default of pass by value will not change arguments in calling function
  - Pass by reference (requires ampersand(&) in function header and prototype) changes arguments in calling function

Summary Concluded

- Scope of variables is part of program where a variable can be used
- Variables can only be used within braces where there are declared and only following the declaration
- Global variables, declared outside any function, can be used by any function following the declaration

Review Function Introduction

- A C++ program is a collection of functions
  - Each function is written as a unit
  - Complete code for one function before starting to write a new one
  - Execution starts in main function
- Upon calls to a function, information and control is transferred to the function
- Value returned in function name

Information Transfer

- Function header has argument list
- Variables in that list (called dummy parameters or dummy arguments) are determined by call to function
- Call to function has actual arguments or actual parameters in same order that dummy arguments appear
  - Order is all that matters in transferring information to a function