

C++ Programming Essentials

Unit 2: Structured Programming

CHAPTER 7: FUNCTIONS

DR. ERIC CHOU

IEEE SENIOR MEMBER

LECTURE 1

Modularization Using Function



Chapter 7 Topics

- Writing a Program Using Functional Decomposition
- Writing a Void Function for a Task
- Using Function Arguments and Parameters
- Differences between Value Parameters and Reference Parameters
- Using Local Variables in a Function
- Function Preconditions and Postconditions



Functions

- every C++ program must have a function called main
 - Starts by looking for a “main”
- program execution always begins with function main
- any other functions are subprograms and must be called



Function Calls

- One function calls another by using the name of the called function next to () enclosing an argument list.

ex. `strlen(FirstName)`

- A function call temporarily transfers control from the calling function to the called function.

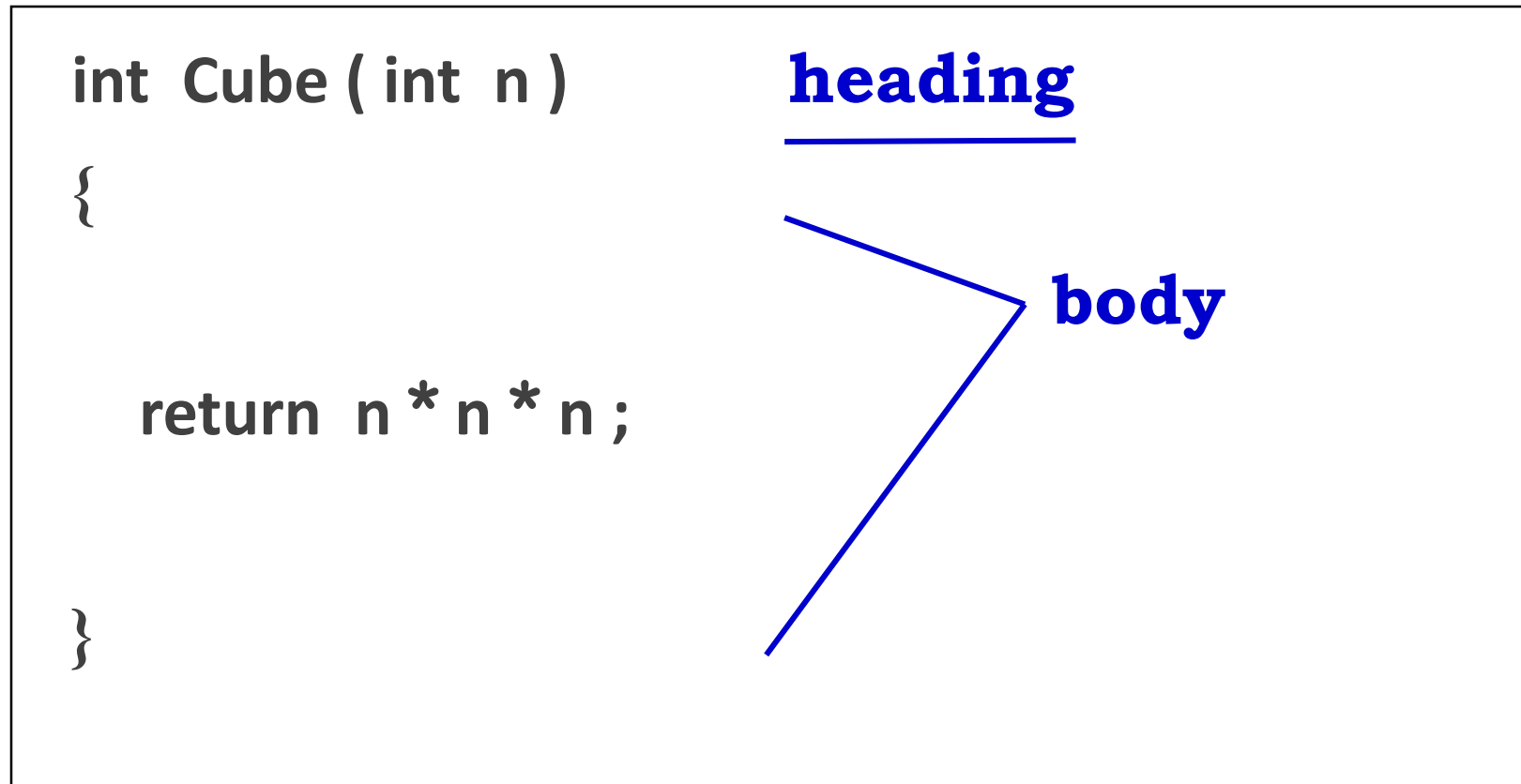


Function Call Syntax

```
FunctionName ( Argument List )
```

- The argument list is a way for functions to communicate with each other by passing information.
- The argument list can contain 0, 1, or more arguments, separated by commas, depending on the function.

Two Parts of Function Definition





What is in a heading?

type of value returned

name of
function

parameter list

```
int Cube ( int n )
```


LECTURE 2

Function's Prototype



What is in a prototype?

- A prototype looks like the heading
- parameter list must contain the type of each parameter.
- Like a variable each function must be declared before it is used.

```
int Cube( int );    // prototype
```



When a function is called,

- temporary memory is set up (for its value parameters and any local variables, and also for the function's name if the return type is not void).
- Then the flow of control passes to the first statement in the function's body. The called function's body statements are executed until one of these occurs:
 - return statement (with or without a return value),
 - or,
 - closing brace of function body.
- Then control goes back to where the function was called.

```
#include <iostream>

int Cube ( int ) ;           // prototype

using namespace std;

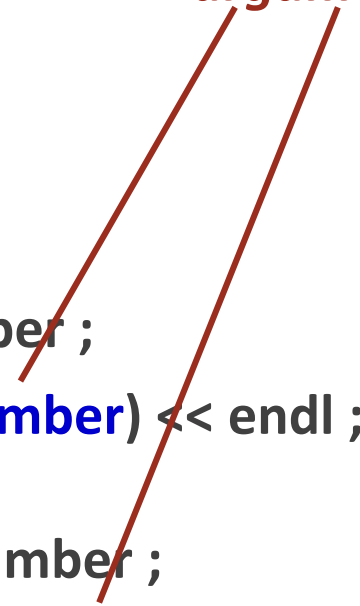
void main ( ){

    int    yourNumber ;
    int    myNumber ;
    yourNumber = 14 ;
    myNumber  = 9 ;
    cout << "My Number = " << myNumber ;
    cout << "its cube is " << Cube (myNumber) << endl ;

    cout << "Your Number = " << yourNumber ;
    cout << "its cube is " << Cube (yourNumber) << endl ;

}
```

arguments





Demo Program:

cube.cpp

Go Dev C++!!!

```
1  #include <iostream>
2
3  int Cube(int);           // prototype
4  using namespace std;
5
6  int Cube (int x){
7      return x*x*x;
8  }
9
10 int main(void){
11     int yourNumber;       // arguments
12     int myNumber;
13     yourNumber = 14;
14     myNumber   = 9 ;
15     cout << "My Number = " << myNumber ;
16     cout << "its cube is " << Cube (myNumber) << endl ;
17
18     cout << "Your Number = " << yourNumber ;
19     cout << "its cube is " << Cube (yourNumber) << endl ;
20     return 0;
21 }
```

C:\Eric_Chou\Cpp Course\C++ Programming Essentials\CppDev\ch7\Cube\Cube.exe

```
My Number = 9its cube is 729
Your Number = 14its cube is 2744
```



To Compile Successfully,

- before a function is called in your program, the compiler must previously process either the function's prototype, or the function's definition (heading and body)



A C++ function can return

- in its identifier at most 1 value of the type which was specified (called the return type) in its heading and prototype
- but, a void-function cannot return any value in its identifier

LECTURE 3

void Function



Write a `void` function

called `DisplayMessage ()` which you can call from `main ()` to describe the pollution index value it receives as a parameter.

Your city describes a pollution Index

less than 35 as “Pleasant”,

35 through 60 as “Unpleasant”,

and above 60 as “Health Hazard.”

The Program (displaymessage2.cpp)


```
#include <iostream>

void DisplayMessage (int);           // prototype

using namespace std;

int main ( )
{
    int pollutionIndex;

    cout << "Enter air pollution index";
    cin >> pollutionIndex;
    DisplayMessage(pollutionIndex); // call
    return 0;
}
```



argument

parameter

```
void DisplayMessage( int index )  
  
{  
    if ( index < 35 )  
        cout << "Pleasant";  
    else if ( index <= 60 )  
        cout << "Unpleasant";  
    else  
        cout << "Health Hazard";  
}
```



return ;

- Return with no value after it
- is valid only in the body block of void functions
- causes control to leave the function and immediately return to the calling block leaving any subsequent statements in the function body unexecuted

LECTURE 4

Header File



Header files contain declarations of

- named constants like
 - `const int INT_MAX = 32767;`
- function prototypes like
 - `float sqrt(float);`
- classes like
 - `string, ostream, istream`
- objects like
 - `cin, cout`

Program with Several Functions

function prototypes

main function

Square function

Cube function

Value-returning Functions

```
#include <iostream>

int Square ( int );           // prototypes
int Cube ( int );

using namespace std;

int main ( )
{
    cout << "The square of 27 is "
          << Square (27) << endl;    // function call

    cout << "The cube of 27 is "
          << Cube (27) << endl;      // function call

    return 0;
}
```


Rest of Program

```
int Square ( int n )      // header and body
{
    return n * n;
}
```

```
int Cube ( int n )       // header and body
{
    return n * n * n;
}
```



Demo Program:

cube2.cpp

Go Dev C++!!!

```
1  #include <iostream>
2  #include "cube2.h"
3  using namespace std;
4  int main(void){
5      cout << "The square of 27 is "
6          << Square(27) << endl;    // function call
7      cout << "The cube of 27 is "
8          << Cube(27) << endl;    // function call
9      return 0;
10 }
11 int Square(int n){    // header and body
12     return n * n;
13 }
14 int Cube(int n){    // header and body
15     return n * n * n;
16 }
17
```

```
1  #ifndef CUBE2_H
2  #define CUBE2_H
3  int Square (int);
4  int Cube (int);
5  #endif
```

A void function call stands alone


```
#include <iostream>

void DisplayMessage ( int ) ;           // prototype

using namespace std;

int main ( )
{
    DisplayMessage(15);                 // function call
    cout << "Good Bye" << endl;
    return 0;
}
```

argument



A `void` function does NOT return a value

```
void DisplayMessage ( int n )  
{  
    cout << "I have liked math for "  
        << n << " years" << endl ;  
  
    return ;  
}
```


parameter



Demo Program:

displaymessage3.cpp

Go Dev C++!!!

 C:\Eric_Chou\Cpp Course\C++ Programming Essentials\CppDev\ch7\DisplayMessage3\DisplayMessage3.exe

```
I have liked math for 15 years  
Good Bye
```

LECTURE 5

Argument List



Parameter List

is the means used for a function to share information with the block containing the call



Classified by Location definitions:

Arguments	Parameters
Always appear in a function call within the calling block.	Always appear in the function heading , or function prototype .



Some C++ Texts

- use the term “actual parameters” for arguments
- those books then refer to parameters as “formal parameters”

LECTURE 6

Call by Value and Call by Reference

4000

25

age

Argument in Calling Block

Value Parameter

The value (25) of the argument is passed to the function when it is called.

In this case, the argument can be a variable identifier, constant, or expression.

Reference Parameter

The memory address (4000) of the argument is passed to the function when it is called.

In this case, the argument must be a variable identifier.



Parameters

- By default, parameters (of simple types like int, char, float, double) are always value parameters, unless you do something to change that.

To get a reference parameter you need to place **&** after the type in the function heading and prototype.

Ex.) `void Cube(int &x);`



When to Use Reference Parameters

- If you want your function to
 - Assign a value to a variable from where it was called
 - Change the value of a variable permanently

Using a Reference Parameter

- is like giving someone the key to your home
- the key can be used by the other person to change the contents of your home!





Main Program Memory

4000

25

age

- If you pass only a copy of 25 to a function, it is called “**pass-by-value**” and the function will not be able to change the contents of age. It is still 25 when you return.



Main Program Memory

4000



age

- BUT, if you pass 4000, the address of age to a function, it is called “**pass-by-reference**” and the function will be able to change the contents of age. It could be 23 or 90 when you return.



Pass-by-reference is also called . . .

- pass-by-address, or
- pass-by-location



Example of Pass-by-Reference

- We want to find 2 real roots for a quadratic equation with coefficients a, b, c . Write a prototype for a void function named `GetRoots()` with 5 parameters. The first 3 parameters are type `float`. The last 2 are reference parameters of type `float`.
- Need to pass 2 things back...

LECTURE 7

Comparison of Function Calling Mechanisms



Function's prototype

```
void GetRoots ( float , float , float , float& , float& );
```

- Now write the function definition using this information.
- This function uses 3 incoming values a, b, c from the calling block. It calculates 2 outgoing values root1 and root2 for the calling block. They are the 2 real roots of the quadratic equation with coefficients a, b, c.

Function Definition

```
void  GetRoots( float a,  float b,  float c,  
                float& root1, float& root2)  
{  
    float temp;                // local variable  
  
    temp = b * b - 4.0 * a * c;  
  
    root1 = (-b + sqrt(temp) ) / ( 2.0 * a );  
  
    root2 = (-b - sqrt(temp) ) / ( 2.0 * a );  
  
    return;  
}
```

```

#include <iostream>
#include <fstream>
#include <cmath>
#include <iomanip>
#include "quadratic.h"
void GetRoots(float, float, float, float&, float&);
using namespace std;
int main(void){
    ifstream myInfile;
    ofstream myOutfile;
    float a, b, c, first, second;
    myInfile.open("q.dat"); // open files
    myOutfile.open("qout.dat");
    while (myInfile >> a >> b >> c){
        GetRoots(a, b, c, first, second); //call
        myOutfile << setw(30) << a << b << c << first << second << endl;
    } // close files
    myInfile.close();
    myOutfile.close();
    return 0;
}

```



Demo Program:

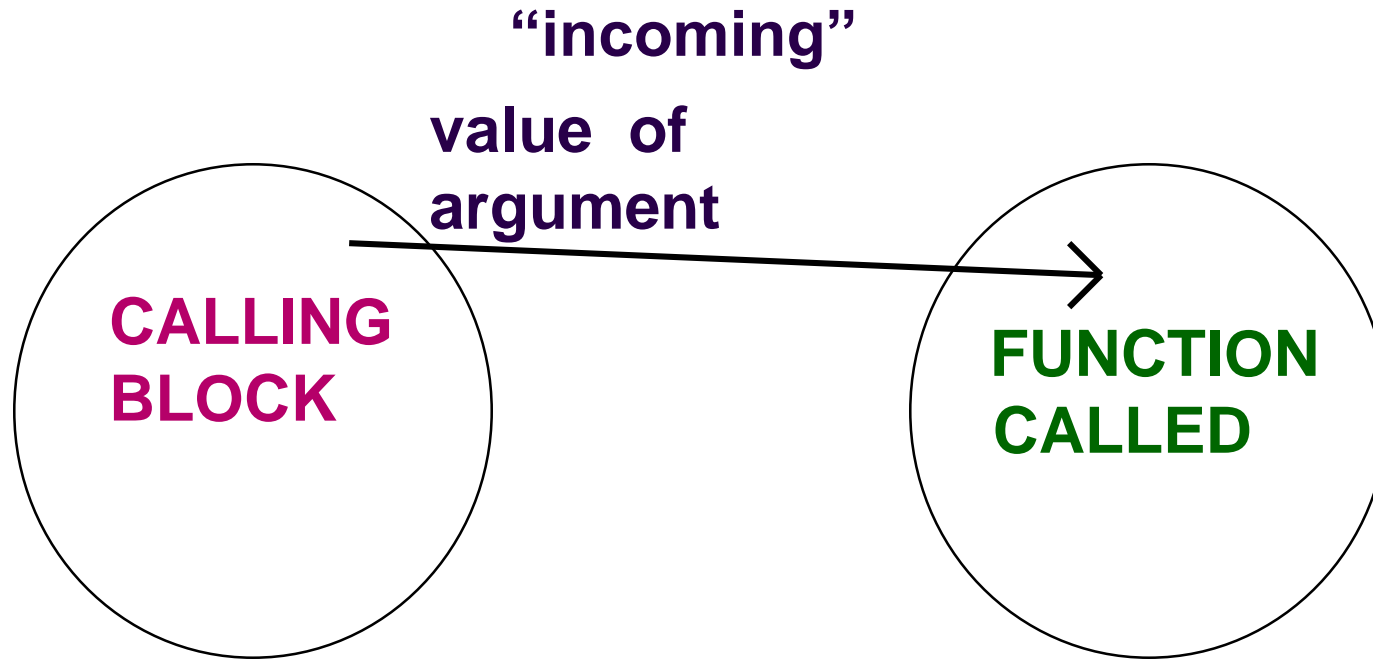
quadratic.cpp

Go Dev C++!!!

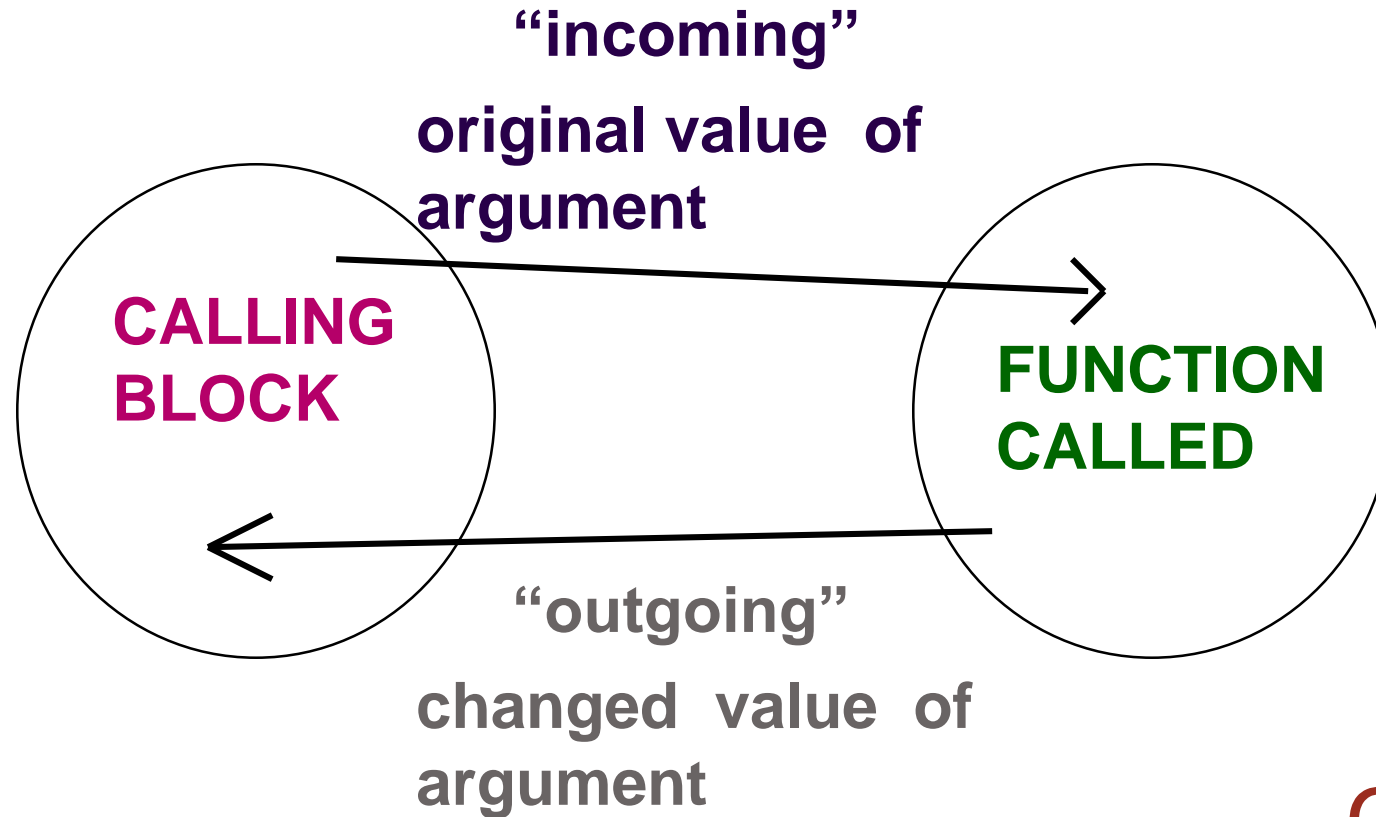
1	1	2	1	-1	-1
2	2	3	4	nan	nan
3	1	-4	4	2	2

```
7 using namespace std;
8 int main(void){
9     ifstream myInfile;
10    ofstream myOutfile;
11    float a, b, c, first, second;
12    myInfile.open("q.dat");           // open files
13    myOutfile.open("qout.dat");
14    while (myInfile >> a >> b >> c){
15        GetRoots(a, b, c, first, second); //call
16        myOutfile << setw(10) << a
17                << setw(10) << b
18                << setw(10) << c
19                << setw(10) << first
20                << setw(10) << second << endl;
21    }                                   // close files
22    myInfile.close();
23    myOutfile.close();
24    return 0;
25 }
26 void GetRoots( float a, float b, float c, float& root1, float& root2){
27     float temp;                       // local variable
28     temp = b * b - 4.0 * a * c;
29     root1 = (-b + sqrt(temp) ) / ( 2.0 * a );
30     root2 = (-b - sqrt(temp) ) / ( 2.0 * a );
31     return;
32 }
```

Pass-by-value “one way”

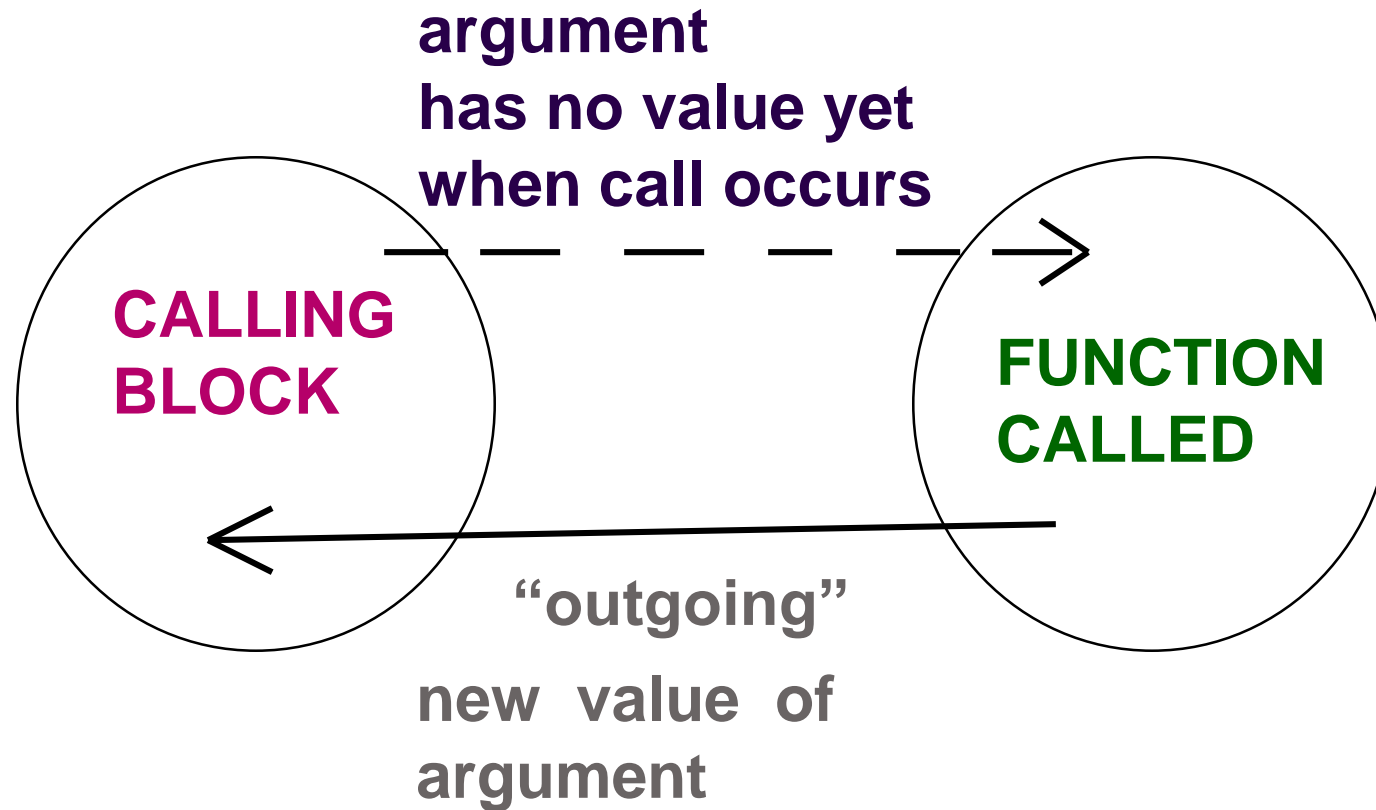


Pass-by-reference “two way”



OR,

Pass-by-reference





Data Flow Determines Passing-Mechanism

Parameter Data Flow	Passing-Mechanism
Incoming <i>/* in */</i>	Pass-by-value
Outgoing <i>/* out */</i>	Pass-by-reference
Incoming/outgoing <i>/* inout */</i>	Pass-by-reference



Questions

- Why is a function used for a task?
- To cut down on the amount of detail in your main program (encapsulation).
- Can one function call another function?
- Yes
- Can a function even call itself?
- Yes, that is called recursion. It is very useful and requires special care in writing.



More Questions

Does it make any difference what names you use for parameters?

NO. Just use them in function body.

Do parameter names and argument names have to be the same?

NO.

What is the advantage of that? It seems confusing.



Functions are written to specifications

- the specifications state the return type, the parameter types, whether any parameters are “outgoing,” and what task the function is to perform with its parameters
- the advantage is that teamwork can occur without knowing what the argument identifiers (names) will be



Choosing a Parameter Passing Mechanism

- Pass-by-Reference
 - use **only if** the design of the called function requires that it be able to modify the value of the parameter
- Pass-by-Constant-Reference
 - use if the called function has no need to modify the value of the parameter, but the parameter is very large (e.g., an array as discussed in Chapter 8)
 - use as a **safety net** to guarantee that the called function cannot be written in a way that would modify the value passed in[†]
- Pass-by-Value
 - use in all cases where none of the reasons given above apply
 - pass-by-value is safer than pass-by-reference

Note that if a parameter is passed by value, the called function may make changes to that value as the formal parameter is used within the function body. Passing by constant reference guarantees that even that sort of internal modification cannot occur.

LECTURE 8

Function and Driver (Program Tester)



Write prototype and function definition for

- a void function called **GetRating()** with one reference parameter of type char
- the function repeatedly prompts the user to enter a character at the keyboard until one of these has been entered: E, G, A, P to represent Excellent, Good, Average, Poor

void GetRating(char&); *// prototype*

```
void GetRating (char& letter)
{
    cout << "Enter employee rating." << endl;
    cout << "Use E, G, A, or P : " ;
    cin >> letter;
    while ( (letter != 'E') && (letter != 'G') &&
            (letter != 'A') && (letter != 'P') )
    {
        cout << "Rating invalid. Enter again: ";
        cin  >> letter;
    }
}
```



What is a driver?

- It is a short main program whose only purpose is to call a function you wrote, so you can determine whether it meets specifications and works as expected.
- write a driver for function **GetRating()**

```
#include <iostream>

void  GetRating( char& );           // prototype

using namespace std;

int main( )
{
    char  rating;
    rating = 'X';
    GetRating(rating);             // call

    cout << "That was rating = "
          << rating << endl;

    return 0;
}
```



An Assertion

- is a truth-valued statement--one that is either true or false (not necessarily in C++ code)

EXAMPLES

- `studentCount > 0`
- `sum is assigned && count > 0`
- `response == 'y' or 'n'`
- `0.0 <= deptSales <= 25000.0`
- `beta == beta @ entry * 2`



Demo Program:

rating.cpp

Go Dev C++!!!

```
1  #include <iostream>
2  #include "rating.h"
3  using namespace std;
4  int main(void)
5  {
6      char rating;
7      rating = 'X';
8      GetRating(rating);    // call
9      cout << "That was rating = "
10         << rating << endl;
11     return 0;
12 }
13 void GetRating (char& letter)
14 {
15     cout << "Enter employee rating." << endl;
16     cout << "Use E, G, A, or P : " ;
17     cin >> letter;
18     while ((letter != 'E') && (letter != 'G') && (letter != 'A') && (letter != 'P')){
19         cout << "Rating invalid. Enter again: ";
20         cin >> letter;
21     }
22 }
```

```
1  #ifndef RATING_H
2  #define RATING
3  void GetRating(char&);    // prototype
4  #endif
```

LECTURE 9

Pre-condition and Post-condition of Functions



Preconditions and Postconditions

- the **precondition** is an assertion describing everything that the function requires to be true at the moment the function is invoked
- the **postcondition** describes the state at the moment the function finishes executing
- the *caller* is responsible for ensuring the precondition, and the *function code* must ensure the postcondition

FOR EXAMPLE . . .

Function with Postconditions

```
void GetRating ( /* out */ char& letter)
// Precondition: None
// Postcondition: User has been prompted to enter a character
//                && letter == one of these input values: E,G,A, or P
{   cout << "Enter employee rating." << endl;
    cout << "Use E, G, A, or P : " ;
        cin >> letter;
        while ( (letter != 'E') && (letter != 'G') &&
            (letter != 'A') && (letter != 'P') )
        {
            cout << "Rating invalid. Enter again: ";
            cin  >> letter;
        }
}
```

Function with Preconditions and Postconditions

```
void GetRoots( /* in */ float a, /* in */ float b, /* in */ float c,  
              /* out */ float& root1, /* out */ float& root2 )  
  
    // Precondition:  a, b, and c are assigned  
    //                && a != 0  && b*b - 4*a*c != 0  
  
    // Postcondition: root1 and root2 are assigned  
    // && root1 and root2 are roots of quadratic with coefficients a, b, c  
  
    {  
        float temp;  
  
        temp = b * b - 4.0 * a * c;  
  
        root1 = (-b + sqrt(temp) ) / ( 2.0 * a );  
        root2 = (-b - sqrt(temp) ) / ( 2.0 * a );  
  
        return;  
    }
```

Function with Preconditions and Postconditions

```
void Swap( /* inout */ int& firstInt,  
           /* inout */ int& secondInt )  
  
// Precondition: firstInt and secondInt are assigned  
// Postcondition: firstInt == secondInt@entry  
//                && secondInt == firstInt@entry  
{  
    int temporaryInt ;  
    temporaryInt = firstInt ;  
  
    firstInt = secondInt ;  
  
    secondInt = temporaryInt ;  
  
}
```