

Ch 6. Functions

Part 3

CS 1428
Fall 2011

Jill Seaman

Lecture 22

1

Passing Arguments by Reference

- Pass by reference: when an argument is passed to a function, the function has direct access to the original argument.
- Pass by reference in C++ is implemented using a reference parameter, which has an ampersand (&) in front of it:

```
void changeMe (int &myValue);
```

- A reference parameter acts as an alias to its argument.
- Changes to the parameter in the function **DO** affect the value of the argument

2

Example: Pass by Reference

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

void changeMe(int &);

int main() {
    int number = 12;
    cout << "number is " << number << endl;
    changeMe(number);
    cout << "Back in main, number is " << number << endl;
    return 0;
}

void changeMe(int &myValue) {
    myValue = 200;
    cout << "myValue is " << myValue << endl;
}
```

Output:
number is 12
myValue is 200
Back in main, number is **200**

myValue is an alias for number

3

Using Pass by Reference for input

```
double square(double) {
    return number * number;
}

void getRadius(double &rad) {
    cout << "Enter the radius of the circle: ";
    cin >> rad;
}

int main() {
    const double PI = 3.14159;
    double radius;
    double area;
    cout << fixed << setprecision(2);
    getRadius(radius);
    area = PI * square(radius);
    cout << "The area is " << area << endl;
    return 0;
}
```

During the function execution,
rad is an alias to radius in the
main program.

4

Pass by Reference

- Changes to a reference parameter are actually made to its argument
- The & must be in the function header AND the function prototype.
- The argument passed to a reference parameter must be a variable – it cannot be an expression or constant
- Use when appropriate – don't use when
 - argument should not be changed by function
 - function needs to return only 1 value

5

More About Variable Definitions and Scope

- The scope of a variable is the part of the program where the variable may be used.
- For a variable defined inside of a function, its scope is the function, from the point of definition to the end of the function.
- For a variable defined inside of a block, its scope is the innermost block in which it is defined, from the point of definition to the end of that block.

6

Variables in functions and blocks

```
int main()
{
    double income; //scope of income is red + blue
    cout << "What is your annual income? ";
    cin >> income;

    if (income >= 35000) {
        int years; //scope of years is blue;
        cout << "How many years at current job? ";
        cin >> years;
        if (years > 5)
            cout << "You qualify.\n";
        else
            cout << "You do not qualify.\n";
    }
    else
        cout << "You do not qualify.\n";
    cout << "Thanks for applying.\n";
    return 0;
}
```

Cannot access years
down here

7

Variables with the same name

- In an inner block, a variable can have the same name as a variable in the outer block.
- When in the inner block, the outer definition is not available (it is hidden).
- Not good style: difficult to trace code and find bugs

Variables with the same name

```
int main()
{
    int number;
    cout << "Enter a number greater than 0: ";
    cin >> number;
    if (number > 0) {
        int number; // another variable named number
        cout << "Now enter another number ";
        cin >> number;
        cout << "The second number you entered was ";
        cout << number << endl;
    }
    cout << "Your first number was " << number << endl;

    return 0;
}
```

Output:
Enter a number greater than 0: **88**
Now enter another number **2**
The second number you entered was 2
Your first number was 88

9

Local and Global Variables

- Variables defined inside a function are local to that function.
 - They are hidden from the statements in other functions, which cannot access them.
- Because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.
 - This is not bad style. These are easy to keep straight.

10

Local variables are hidden from other functions

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

void anotherFunction();

int main() {
    int num = 1;
    cout << "In main, num is " << num << endl;
    anotherFunction();
    cout << "Back in main, num is " << num << endl;
    return 0;
}

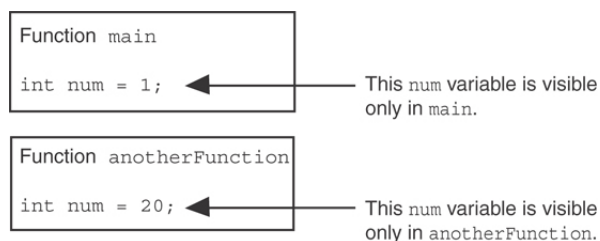
void anotherFunction() {
    int num = 20;
    cout << "In anotherFunction, num is " << num << endl;
}
```

Output:
In main, num is 1
In anotherFunction, num is 20
Back in main, num is 1

11

Local variables are hidden from other functions

- When the program is executing main, the num variable defined in main is visible.
- When anotherFunction is called, only variables defined inside it are visible, so the num variable in main is hidden.



12

Local Variable Lifetime

- Parameters have the same scope as local variables in the function.
- When the function begins, its parameters and local variables (as their definitions are encountered) are created in memory, and when the function ends, the parameters and local variables are destroyed.
- This means that any value stored in a local variable is lost between calls to the function in which the variable is declared.

13

Global Variables

- A global variable is any variable defined outside all the functions in a program.
- The scope of a global variable is the portion of the program from the variable definition to the end.
- This means that a global variable can be accessed by all functions that are defined after the global variable is defined

14

Global Variables: example

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

void anotherFunction();
int num = 2;

int main() {
    cout << "In main, num is " << num << endl;
    anotherFunction();
    cout << "Back in main, num is " << num << endl;
    return 0;
}

void anotherFunction() {
    cout << "In anotherFunction, num is " << num << endl;
    num = 50;
    cout << "But now it is changed to " << num << endl;
}
```

Output:

In main, num is 2
In anotherFunction, num is 2
But now it is changed to 50
Back in main, num is 50

Global Variables

- You should avoid using global variables because:
- They make programs difficult to debug.
 - If the wrong value is stored in a global var, you have to find every place in the whole program where the value is changed
- Functions that access globals are not self-contained
 - cannot easily reuse the function in another program.
 - cannot understand the function without understanding how the global is used everywhere

Global Constants: example

- It is ok to use global constants because their values do not change.

```
double getArea(double);  
double getPerimeter(double);  
  
const double PI = 3.14159;  
  
int main() {  
    double radius;  
    cout << fixed << setprecision(2);  
    cout << "Enter the radius of the circle: ";  
    cin >> radius;  
  
    cout << "The area is " << getArea(radius) << endl;  
    cout << "The perimeter is " << getPerimeter(radius) << endl;  
    return 0;  
}
```

17

Global Constants: example

```
double getArea(double number) {  
    return PI * number * number;  
}  
  
double getPerimeter(double number) {  
    return PI * 2 * number;  
}
```

Output:
Enter the radius of the circle: 2.2
The area is 15.21
The perimeter is 13.82

18

Scope Rules Summary

- Variable scope: to end of the block it's defined in.
- Variables cannot have same name in same exact scope.
 - Variable defined in inner block can hide a variable with the same name from outer block.
- Variables defined in one function cannot be seen from another.
- Parameter scope: the body of the function
 - cannot have function variable same name as parameter
- Variable lifetime: variables are destroyed at the end of their scope
- Global variable/constant scope: to end of entire program
 - variables defined inside a function are called Local