CSC 211: Computer Programming

Scope, Parameter passing, Call stack

Michael Conti

Department of Computer Science and Statistics
University of Rhode Island

Spring 2024

Administrative Notes

- Exam#01 this Thursday
  - Calculator without internet Ok (no phone)
  - 8x11 hand written cheat sheet
  - last day of exam content
- A02 Out, due 03/03
- MC04 Out, due 02/23

Scope of Variables, Passing Parameters

Scope (where is a variable visible?)

- **Local** variables
  - local to a function, cannot be used outside the function
- **Global** variables
  - available to all functions in the same program
  - declared outside any function
  - not recommended, make programs difficult to maintain
- **Global** constants
  - same as global variables, but require the `const` type qualifier

A Global Named Constant (*part 1 of 2*)

```
//Computes the area of a circle and the volume of a sphere.
//Uses the same radius for both calculations.
#include <iostream>
#include <cmath>
using namespace std;

const double PI = 3.14159;

double area(double radius);
//Returns the area of a circle with the specified radius.

double volume(double radius);
//Returns the volume of a sphere with the specified radius.

int main()
{
    double radius_of_both, area_of_circle, volume_of_sphere;

    cout << "Enter a radius to use for both a circle\n"
         << "and a sphere (in inches): ";
    cin >> radius_of_both;

    area_of_circle = area(radius_of_both);
    volume_of_sphere = volume(radius_of_both);

    cout << "Radius = " << radius_of_both << " inches\n"
         << "Area of circle = " << area_of_circle
         << " square inches\n"
         << "Volume of sphere = " << volume_of_sphere
         << " cubic inches\n";

    return 0;
}
```

A Global Named Constant (*part 2 of 2*)

```
double area(double radius)
{
    return (PI * pow(radius, 2));
}

double volume(double radius)
{
    return ((4.0/3.0) * PI * pow(radius, 3));
}
```

Sample Dialogue

```
Enter a radius to use for both a circle
and a sphere (in inches): 2
Radius = 2 inches
Area of circle = 12.5664 square inches
Volume of sphere = 33.5103 cubic inches
```


Passing parameters (pass by value)

- Parameters are actually **local variables** to the function
- The **pass by value** mechanism (default method)
  - parameters are initialized to the values of the arguments in the function call
  - when invoking a function call, **arguments are copied into the parameters** of a function

Block Scope Revisited

```
1   #include <iostream>
2   using namespace std;
3
4   const double GLOBAL_CONST = 1.0;
5
6   int function1 (int param);
7
8   int main()
9   {
10      int x;
11      double d = GLOBAL_CONST;
12
13      for (int i = 0; i < 10; i++)
14      {
15          x = function1(i);
16      }
17      return 0;
18  }
19
20  int function1 (int param)
21  {
22      double y = GLOBAL_CONST;
23      ...
24      return 0;
25  }
```

Local and Global scope are examples of Block scope. A variable can be directly accessed only within its scope.

*Block scope: Variable **i** has scope from lines 13-16*

*Local scope to **main:** Variable **x** has scope from lines 10-18 and variable **d** has scope from lines 11-18*

*Global scope: The constant **GLOBAL_CONST** has scope from lines 4-25 and the function **function1** has scope from lines 6-25*

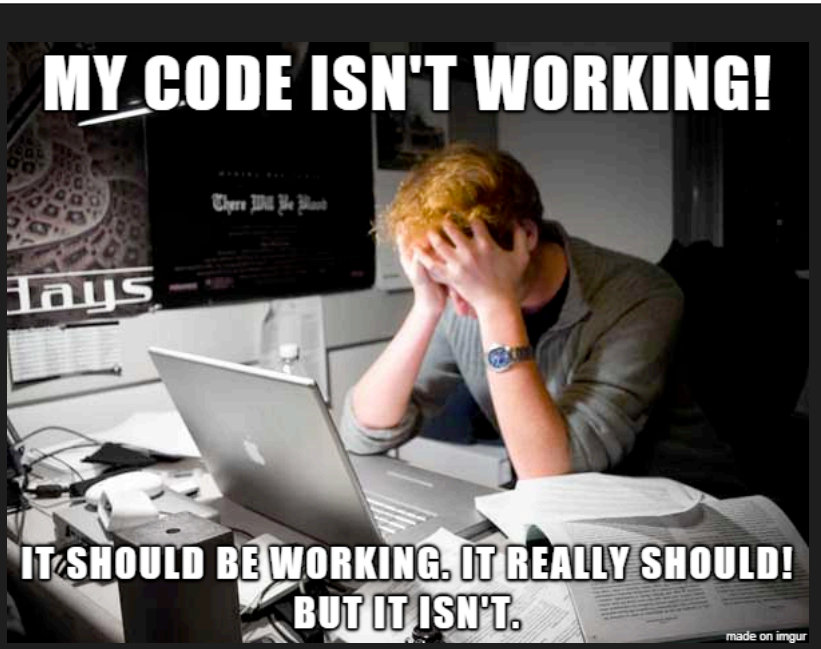
*Local scope to **function1:** Variable **param** has scope from lines 20-25 and variable **y** has scope from lines 22-25*

Lets try a `swap` function …

```
void swap (int x, int y) {
    int temp;

    temp = x;
    x = y;
    y = temp;

    return;
}
```

What is the output?

```
#include <iostream>

void swap (int x, int y);

int main () {
    int x = 100;
    int y = 200;

    std::cout << "Value of x :" << x << '\n';
    std::cout << "Value of y :" << y << '\n';

    swap(x, y);

    std::cout << "Value of x :" << x << '\n';
    std::cout << "Value of y :" << y << '\n';

    return 0;
}
```

An **Integrated Development Environment** (IDE) usually provides a built-in **debugger**

References

- A **reference** is an alias for another variable
  - just another name for the same memory location

```
int main() {
    int val1 = 1, val2 = 5;
    int &ref = val1;

    val1 += 1;
    ref += 1;
    ref = val2;
    ref *= 2;

    return 0;
}
```

&

References

```
int main() {
    int val1 = 1, val2 = 5;
    int &ref = val1;

    val1 += 1;
     //checkpointA
    ref += 1;
     //checkpointB
    ref = val2;
     //checkpointC
    ref *= 2;
   //checkpointD

    return 0;
}
```


Pass by reference

- You can pass arguments to functions **by reference**
- Modifying the reference parameter modifies the actual argument!

```
void swap (int& x, int& y) {
    int temp;

    temp = x;
    x = y;
    y = temp;

    return;
}
```


What is the output?

```
#include <iostream>

void swap (int &x, int &y);

int main () {
    int x = 100;
    int y = 200;

    std::cout << "Value of x :" << x << '\n';
    std::cout << "Value of y :" << y << '\n';

    swap(x, y);

    std::cout << "Value of x :" << x << '\n';
    std::cout << "Value of y :" << y << '\n';

    return 0;
}
```


What is the output

```
#include <iostream>

void mystery(int& b, int c, int& a) {
    a ++;
    b --;
    c += a;
}

int main() {
    int a = 5;
    int b = 10;
    int c = 15;

    mystery(c, a, b);
    std::cout << a << ' ' << b << ' ' << c << '\n';

    return 0;
}
```

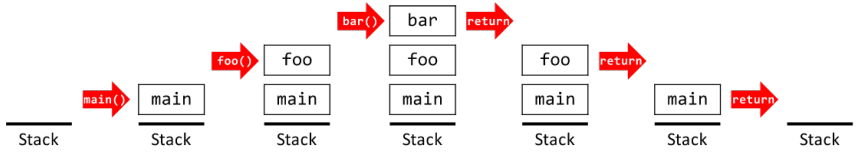
The call stack

Function calls and the call stack

- Variables are stored at different locations in memory
- In practice, it is well more structured …
  - **stack-based memory management** is used by many language implementations
- Program execution needs a **call stack** to deal with functions
  - a **stack frame** stores data for a function call, essentially local variables

Stack frames

```
void bar() {
}

void foo() {
    bar();
}

int main() {
    foo();
}
```



Stack frames (detailed view)

```
#include <iostream>

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

int plus_two(int x) {
    return plus_one(x + 1);
}

int main() {
    int result = 0;
    result = plus_one(0);
    result = plus_two(result);
    std::cout << result;
}
```

Trace the stack

```
int bar(int &b) {
  int c = 0;
  while (b > 0){
    c += 2;
     b -= 2;

  }
  return c;
}

int foo(int a) {
  int temp = 7;
  a = a + bar(temp);
  return a;
}

int main() {
  int a = 5;
  int b = 5;
  int c = foo(a + b);

  return 0;
}
```

Line 7

Report the status of the call stack if we pause the execution of the program exactly at line number 7. Assume the stack grows from top to bottom.

Frame	Variable Name	Current Value

https://tinyurl.com/yyxr6key

Additional remarks on functions

Preconditions and Postconditions

from: Problem Solving with C++, 10th Edition, Walter Savitch

Testing and Debugging

- Each function must be tested as a separate an independent unit
- Once properly tested, the function then can be used in the program

Functions must be tested in environments where every other function has already been fully tested and debugged