



Object Oriented Programming by C++

Functions (2/2)

Advanced: Local & Global Variable, Call by Value & Reference

2017. 8.

Sungwon Lee / Professor

Email: drsungwon@khu.ac.kr

Web: <http://mobilelab.khu.ac.kr/>

Modified & taught by Jinwoo Choi
in 2023 Spring

Email: jinwoochoi@khu.ac.kr

Webpage: <https://sites.google.com/site/jchoivision/>

Contents

- Local Variable 지역 변수
- Global Variable 광역 변수
- Pass by Value
- Recursion
- Reference Variable (= Aliasing)
- Pass by Reference

Variable Scope

Local Variables

```
#include <iostream>
using namespace std;
int makeDouble( int param )
{
    int x;          int x = 0;
    cout << "03: " << x << '\n';
    x = param;
    cout << "04: " << x << '\n';
    return param * 2;
}
int main()
{
    int x = 10;
    cout << "01: " << x << '\n';
    x = makeDouble( x );  cout << "02: " << makeDouble(x) << '\n';
    cout << "02: " << x << '\n';
    return 0;
}
```

[Output]

01: 10
03: -1031346592
04: 10
02: 20
Program ended with exit code: 0

WHY?

02: 03: -1031346592
04: 10
20

Variable Scope

Local Variables Life-Cycle

```
#include <iostream>
using namespace std;
int makeDouble( int param )
{
    int x; → Located at Memory A
    cout << "03: " << x << '\n';
    x = param;
    cout << "04: " << x << '\n';
    return param * 2;
}
int main()
{
    int x = 10; → Located at Memory B
    cout << "01: " << x << '\n';
    x = makeDouble( x );
    cout << "02: " << x << '\n';
    return 0;
}
```

Variable Scope

Local Variables Life-Cycle

```
#include <iostream>
using namespace std;
int makeDouble( int param )
{
    int x;
    cout << "03: " << x << '\n';
    x = param;
    cout << "04: " << x << '\n';
    return param * 2;
}
int main()
{
    int x = 10;
    cout << "01: " << x << '\n';
    x = makeDouble( x );
    cout << "02: " << x << '\n';
    return 0;
}
```

Creation: function invoked

Termination: function returning

Creation: main() executed

Termination: main() ended

Variable Scope

Local Variables

```
01: #include <iostream>
02: using namespace std;
03: int makeDouble( int param )
04: {
05:     int x;
06:     cout << "03: " << x << '\n';
07:     x = param;
08:     cout << "04: " << x << '\n';
09:     return param * 2;
10: }
11: int main()
12: {
13:     int x = 10;
14:     cout << "01: " << x << '\n';
15:     x = makeDouble( x );
16:     cout << "02: " << x << '\n';
17:     return 0;
18: }
```

[Execution Sequence]

```
13: x@main created & set to 10
14: print x@main(= 10)
15: copy x@main's 10 to makeDouble
03: copy 10 to param@makeDouble
05: x@makeDouble created & no value
06: print x@makeDouble(= Garbage)
07: copy param@makeDouble(= 10) to
    x@makeDouble
08: print x@makeDouble(= 10)
09: return param@makeDouble(= 10)*2
10: x@makeDouble & param@makeDouble
    terminated
15: copy param@makeDouble(= 10)*2
    to x@main
16: print x@makeDouble(= 20)
18: x@main terminated
```

Variable Scope

Global Variables

```
01: #include <iostream>
02: using namespace std;
03: int x = 10;
04: int makeDouble( int param )
05: {
06:     int x;
07:     cout << "03: " << x << '\n';
08:     x = param;
09:     cout << "04: " << x << '\n';
10:    return param * 2;
11: }
12: int main()
13: {
14:     cout << "01: " << x << '\n';
15:     x = makeDouble( x );
16:     cout << "02: " << x << '\n';
17:     return 0;
18: }
```

[Execution Sequence]

```
03: x@global created & set to 10
14: print x@global(= 10)
15: copy x@global's 10 to
     makeDouble
04: copy 10 to param@makeDouble
06: x@makeDouble created & no value
07: print x@makeDouble(= Garbage)
08: copy param@makeDouble(= 10) to
     x@makeDouble
09: print x@makeDouble(= 10)
10: return param@makeDouble(= 10)*2
11: x@makeDouble & param@makeDouble
     terminated
15: copy param@makeDouble(= 10)*2
     to x@global
16: print x@makeDouble(= 20)
18: x@global terminated
```

Variable Scope

Global Variables

```
01: #include <iostream>
02: using namespace std;
03: int x = 10;
04: int makeDouble( int param )
05: {
06:     int x;
07:
08:     cout << "03: " << x << '\n';
09:
10:    x = param;
11:
12:    cout << "04: " << x << '\n';
13:
14:    return param * 2;
15:
16: }
17:
18: int main()
19: {
20:     cout << "01: " << x << '\n';
21:
22:     x = makeDouble( x );
23:
24:     cout << "02: " << x << '\n';
25:
26:     return 0;
27: }
```

Creation: program executed

Termination: program terminated

Variable Scope

Global Variables

```
01: #include <iostream>
02: using namespace std;
03: int x = 10;
04: int makeDouble( int param )
05: {
06:     int x;
07:     cout << "03: " << x << '\n';
08:     x = param;
09:     cout << "04: " << x << '\n';
10:    return param * 2;
11: }

12: int main()
13: {
14:     cout << "01: " << x << '\n';
15:     x = makeDouble( x );
16:     cout << "02: " << x << '\n';
17:     return 0;
18: }
```

[Output]

```
01: 10
03: -1031346592 (Garbage)
04: 10
02: 20
Program ended with exit code: 0
```

Parameter Transfer to Functions

Pass by Value

```
01: #include <iostream>
02: using namespace std;
03: int makeDouble( int param )
04: {
05:     int x;
06:     cout << "03: " << x << '\n';
07:     x = param;
08:     cout << "04: " << x << '\n';
09:     return param * 2;
10: }
11: int main()
12: {
13:     int x = 10;
14:     cout << "01: " << x << '\n';
15:     x = makeDouble( x );
16:     cout << "02: " << x << '\n';
17:     return 0;
18: }
```

COPY

[Execution Sequence]

```
13: x@main created & set to 10
14: print x@main(= 10)
15: copy x@main's 10 to makeDouble
03: copy 10 to param@makeDouble
05: x@makeDouble created & no value
06: print x@makeDouble(= Garbage)
07: copy param@makeDouble(= 10) to x@makeDouble
08: print x@makeDouble(= 10)
09: return param@makeDouble(= 10)*2
10: x@makeDouble & param@makeDouble
terminated
15: copy param@makeDouble(= 10)*2
to x@main
16: print x@makeDouble(= 20)
18: x@main terminated
```

Parameter Transfer to Functions

Pass by Value Example

Listing 9.9: passbyvalue.cpp

```
#include <iostream>

/*
 * increment(x)
 *     Illustrates pass by value protocol.
 */
void increment(int x) {
    std::cout << "Beginning execution of increment, x = "
        << x << '\n';
    x++; // Increment x
    std::cout << "Ending execution of increment, x = "
        << x << '\n';
}

int main() {
    int x = 5;
    std::cout << "Before increment, x = " << x << '\n';
    increment(x);
    std::cout << "After increment, x = " << x << '\n';
}
```

Parameter Transfer to Functions

Pass by Value Example Results

Listing 9.9 (passbyvalue.cpp) produces

```
Before increment, x = 5
Beginning execution of increment, x = 5
Ending execution of increment, x = 6
After increment, x = 5
```

```
/*
void increment(int x) {
    std::cout << "Beginning execution of increment, x = "
        << x << '\n';
    x++; // Increment x
    std::cout << "Ending execution of increment, x = "
        << x << '\n';
}

int main() {
    int x = 5;
    std::cout << "Before increment, x = " << x << '\n';
    increment(x);
    std::cout << "After increment, x = " << x << '\n';
}
```

Recursion

Function calls itself

- Why? $f(n+2) = f(n+1) + f(n)$
0 1 1 2 3 5 8 13

The *factorial* function is widely used in combinatorial analysis (counting theory in mathematics), probability theory, and statistics. The factorial of n is often expressed as $n!$. Factorial is defined for nonnegative integers as

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots 2 \cdot 1$$

and $0!$ is defined to be 1. Thus $6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720$. Mathematicians precisely define factorial in this way:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise.} \end{cases} \quad f(n) = n * f(n-1)$$

This definition is *recursive* since the `!` function is being defined, but `!` is also used in the definition. A C++ function can be defined recursively as well. Listing 10.6 (factorialtest.cpp) includes a factorial function that exactly models the mathematical definition.

Recursion

Example

Listing 10.6: factorialtest.cpp

```
#include <iostream>

/*
 * factorial(n)
 *     Computes n!
 *     Returns the factorial of n.
 */
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}

int main() {
    // Try out the factorial function
    std::cout << " 0! = " << factorial(0) << '\n';
    std::cout << " 1! = " << factorial(1) << '\n';
    std::cout << " 6! = " << factorial(6) << '\n';
    std::cout << "10! = " << factorial(10) << '\n';
}
```

Recursion

Example Results

Listing 10.6: factorialtest.cpp

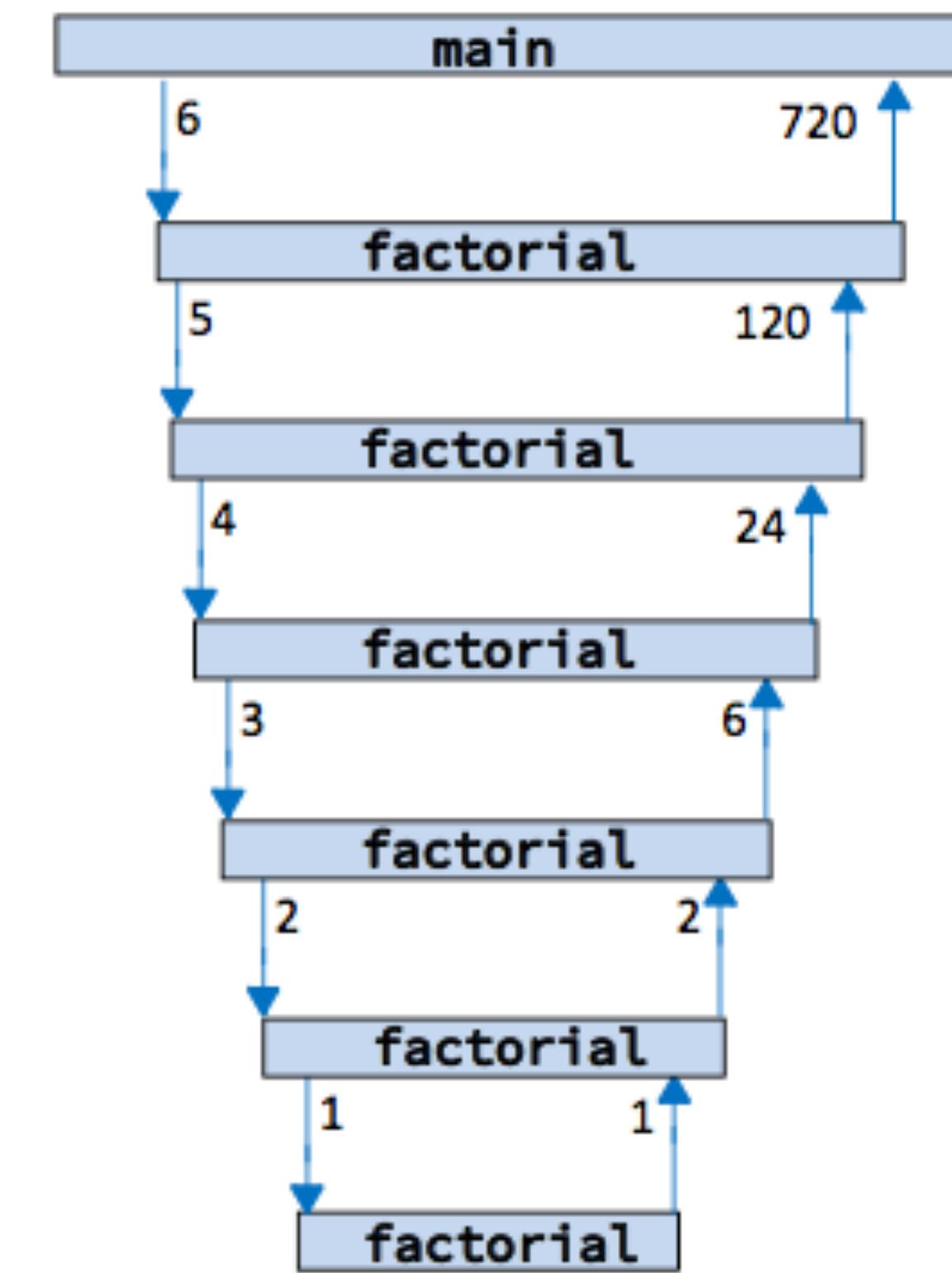
```
#include <iostream>

/*
 * factorial(n)
 *     Computes n!
 *     Returns the factorial of n.
 */
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}

int main() {
    // Try out the factorial function
    std::cout << " 0! = " << factorial(0) << '\n';
    std::cout << " 1! = " << factorial(1) << '\n';
    std::cout << " 6! = " << factorial(6) << '\n';
    std::cout << "10! = " << factorial(10) << '\n';
}
```

**factorial(6) function call sequence
(called from main)**

→ Program Execution Timeline →



Recursion

Example Results

Listing 10.6: factorialtest.cpp

```
#include <iostream>

/*
 * factorial(n)
 *     Computes n!
 *     Returns the factorial
 */
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}

int main() {
    // Try out the factorial function
    std::cout << " 0! = " << factorial(0) << '\n';
    std::cout << " 1! = " << factorial(1) << '\n';
    std::cout << " 6! = " << factorial(6) << '\n';
    std::cout << "10! = " << factorial(10) << '\n';
}
```

```
factorial(6) = 6 * factorial(5)
              = 6 * 5 * factorial(4)
              = 6 * 5 * 4 * factorial(3)
              = 6 * 5 * 4 * 3 * factorial(2)
              = 6 * 5 * 4 * 3 * 2 * factorial(1)
              = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
              = 6 * 5 * 4 * 3 * 2 * 1 * 1
              = 6 * 5 * 4 * 3 * 2 * 1
              = 6 * 5 * 4 * 3 * 2
              = 6 * 5 * 4 * 6
              = 6 * 5 * 24
              = 6 * 120
              = 720
```

Reference Variable

Aliasing

- When the & symbol is used as part of the type name during a variable declaration, as in

int x;

int& r = x;

int& k=x;

- We say *r* is a ***reference variable***.
- This declaration creates a variable *r* that refers to the ***same memory*** location as the variable *x*.
- We say that ***r aliases x***.

Reference Variable

Reference Variable Example

Listing 10.16: referencevar.cpp

```
#include <iostream>

int main() {
    int x = 5;
    int y = x;
    int& r = x;
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';
    std::cout << "Assign 7 to x\n";
    x = 7;
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';
    std::cout << "Assign 8 to y\n";
    y = 8;
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';
    std::cout << "Assign 2 to r\n";
    r = 2;
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';
}
```

Reference Variable

Reference Variable Example Result

Listing 10.16: referencevar.cpp

```
#include <iostream>

int main() {
    int x = 5;
    int y = x;
    int& r = x;
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';
    std::cout << "Assign 7 to x\n";
    x = 7;
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';
    std::cout << "Assign 8 to y\n";
    y = 8;
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';
    std::cout << "Assign 2 to r\n";
    r = 2;
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';
}
```

The output Listing 10.16 (referencevar.cpp):

```
x = 5
y = 5
r = 5
Assign 7 to x
x = 7
y = 5
r = 7
Assign 8 to y
x = 7
y = 8
r = 7
Assign 2 to r
x = 2
y = 8
r = 2
```

Parameter Transfer to Functions

Wrong Example using Pass by Value

Listing 10.17: faultyswap.cpp

```
#include <iostream>

/*
 * swap(a, b)
 *   Attempts to interchange the values of
 *   its parameters a and b. That it does, but
 *   unfortunately it only affects the local
 *   copies.
 */
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

/*
 * main
 *   Attempts to interchange the values of
 *   two variables using a faulty swap function.
 */
int main() {
    int var1 = 5, var2 = 19;
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
    swap(var1, var2);
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
}
```

Parameter Transfer to Functions

Wrong Example Results

Listing 10.17: **faultyswap.cpp**

The output of Listing 10.17 (**faultyswap.cpp**) is

```
var1 = 5, var2 = 19
var1 = 5, var2 = 19
```

```
/*
 *      copies.
 */
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

/*
 *  main
 *  Attempts to interchange the values of
 *  two variables using a faulty swap function.
 */
int main() {
    int var1 = 5, var2 = 19;
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
    swap(var1, var2);
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
}
```

Parameter Transfer to Functions

Pass by Reference Example

Listing 10.19: swapwithreferences.cpp

```
#include <iostream>

/*
 *  swap(a, b)
 *      Interchanges the values of memory
 *      referenced by its parameters a and b.
 *      It effectively interchanges the values
 *      of variables in the caller's context.
 */
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

/*
 *  main
 *      Interchanges the values of two variables
 *      using the swap function.
 */
int main() {
    int var1 = 5, var2 = 19;
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
    swap(var1, var2);
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
}
    int& a=var1;
    int& b=var2;
```



Parameter Transfer to Functions

Pass by Reference Example Result

Listing 10.19: swapwithreferences.cpp

```
#include <iostream>

/*
 *  swap(a, b)
 *      Interchanges the values of memory
 *      referenced by its parameters a and b.
 *      It effectively interchanges the values
 *      of variables in the caller's context.
 */
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

/*
 *  main
 *      Interchanges the values of two variables
 *      using the swap function.
 */
int main() {
    int var1 = 5, var2 = 19;
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
    swap(var1, var2);
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
}
```

```
var1 = 5, var2 = 19
var1 = 19, var2 = 5
```



Object Oriented Programming by C++

Sungwon Lee / Professor

Email: drsungwon@khu.ac.kr

Web: <http://mobilelab.khu.ac.kr/>