Lecture 4

C++ Programming

Arne Kutzner
Hanyang University / Seoul Korea

C++ Namespaces

C++ Namespaces

- What is this strange '**`std::`**' in **`std::cout`** ?
- - Concept of Namespaces -
  Why does it exist?
  We want to use the same identifier in several different contexts
- Occurs in XML as well

Namespaces

- Example for Defintion:

```
namespace myNamespace
{
  int a, b;
}
```

a and b occur in the namespace myNamespace

- Example for use of namespace:

```
myNamespace::a
myNamespace::b
```

“using” a namespace

- Example:

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

int main () {
   cout << 5.0 << “\n”;
}
```

from here on we use the namespace std

`std::` is not necessary now

C++ Functions

Functions Introduction

- Functions are program modules written to
  - Avoid the repetition of identical code parts
  - Solve "a bigger problem" by decomposing it into "smaller problems"
  - In other languages called procedures or subroutines.
- Example:
  A Function **`max`** that delivers the maximum of two values.
- Functions
  1. ***take one or several arguments***,
  2. ***compute some statements*** and
  3. ***return a single value***

Writing a function

- You have decide on what the function will *look* like:
  - Return type
  - Name
  - Formal arguments
    (also called **parameter**)
- You have to write the body (the actual code).

Function Definition in C++

- Syntax

Data type of the returned value

Function name (identifier)

Formal Arguments

```
data_type identifier (arg_1, arg_2,…) {
    local variable declarations

    executable statements
}
```

Formal arguments

- Syntax of a single formal argument:
  data_type identifier
- The Formal arguments behave like *local variables* inside the body of the function.
  - When the function is called they will have the values *passed in*.

Local variables

- Local variables are variables that are known inside a function only
  - Different functions may have local variables with identical names
- They only exist inside the function body.
- Once the function returns, the variables no longer exist!

The **`return`** statement

- Functions return a single value using the return statement.
  Syntax:
  `return expression ;`

Example: **`max`** function

```
int max (int i, int j) {
   int m;

   if (i > j)
      m = i;
   else
      m = j;
   return m;
}
```

Local variable definition

Function Calls

actual arguments

- Syntax:
  function_name(arg1, arg2, …);
  - Actual arguments may be constants, variables, or expressions.
  - Example of function call
    max(a, b)
  - Example of function call plus assignment
    x = max(a, b);

Example Function Call

pass **i**

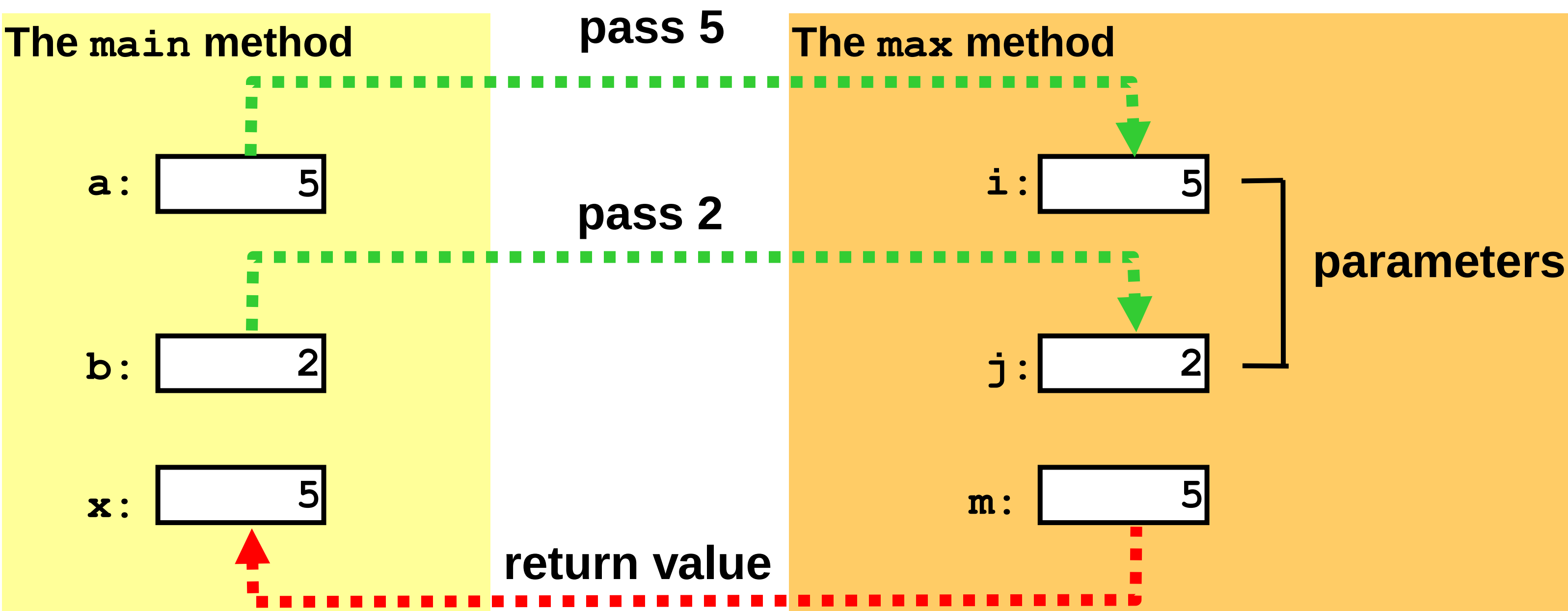
pass **j**

```
void main() {
  int a, b, x;
  a = 5;
  b = 2;

  x = max (a, b);
  printf("%d", x);
}
```

```
int max (int i, int j) {
  int m;

  if (i > j)
      m = i;
  else
      m = j;
  return m;
}
```

Example Function Call, cont.

- The values of **`a`** and **`b`** **<u>are copied</u>** to **`i`** and `j` .Graphically:



- Because the arguments are copied we talk of a **call by value** semantic

Function without Returned Value

- Syntax

```
void fname (arg1, arg2, …) {
    local variable declarations
    executable statements
}
```

- The keyword **void** indicates that the function does not return any value

Library functions

- C++ includes a bunch of libraries, which contain predefined functions
  – You don’t have to know how they internally do their job.
  – But, you have to know what they do and what they return.
- Let us have a short look into the Math library math.h …

Telling the compiler about the function `sqrt`

- We will work with the square root function in the Math-library
- We have to tell the compiler that want to use **`math.h`** (Math-library):
  `#include <math.h>`
- The name of the square root function is **`sqrt`**

`double sqrt( double )`

- When *calling* **`sqrt`**, we have to give it a **`double`**.
- The **`sqrt`** function returns a **`double`**.
- We have to give it a **`double`**.

```
double y = 25.0;
x = sqrt(y);
x = sqrt(100.0);
```

Table of square roots

```
int i;
for (i=1;i<10;i++)
  cout << sqrt(i) << "\n";
```

- But I thought we had to give **`sqrt()`** a double?
- C++ does automatic *type conversion* for you.

More functions defined in `math.h`

Function	Purpose
`double ceil(double x)`	smallest integer greater than x
`double exp(double x)`	e^x
`double fabs(double x)`	absolute value of x
`double floor(double x)`	largest integer less than x
`double log(double x)`	natural logarithm of x
`double log10(double x)`	base 10 logarithm of x
`double sqrt(double x)`	square root of x

More functions defined in `math.h`

Function	Purpose
`double sin(double x)`	sine of x
`double cos(double x)`	cosine of x
`double tan(double x)`	tangent of x
`double pow(double x, double y)`	x^y

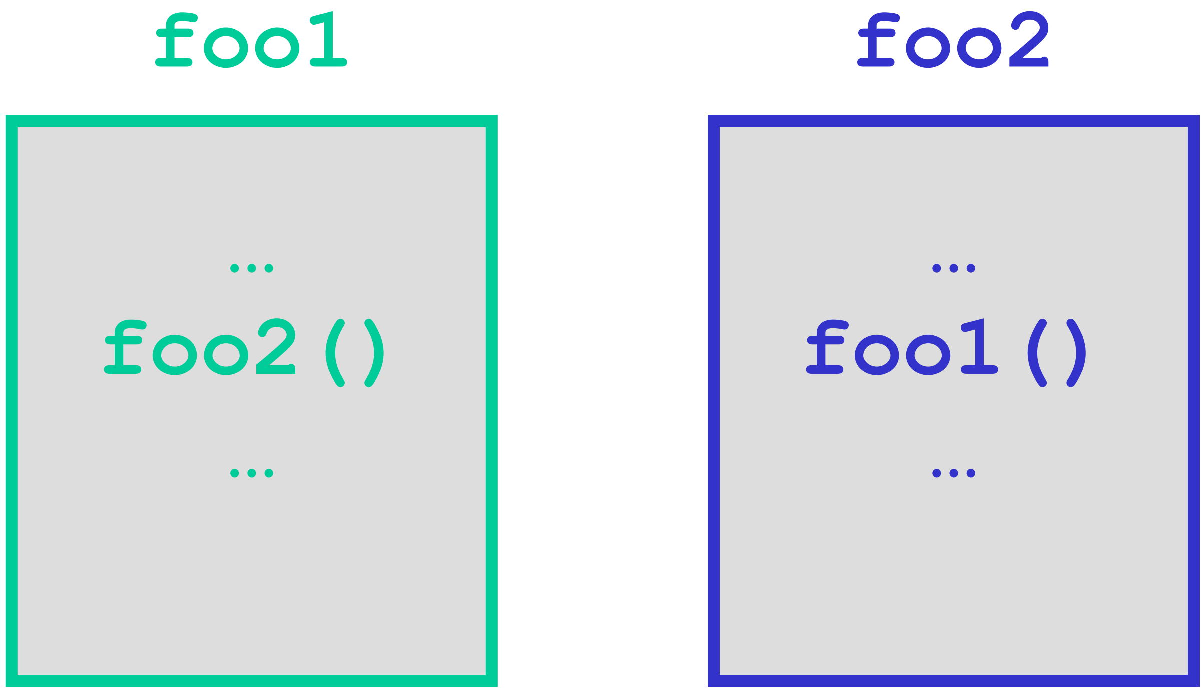
Scope of Functions, Mutual Calls, Recursion

The Scope of Functions

- Scope of a function / variable:
  The part of the program where a variable can be referenced.
- Function names have *global* scope:
  "Everything" that follows a function definition in the same file can use the function.
  - Sometimes this is not convenient:
    - We want to call the function om the top of the file and define it at the bottom of the file.

The Scope of Functions

- The global scope of functions becomes a troublemaker in the context of mutually calling functions:



Example for mutually calling functions:

```
char *chicken( int generation ) {
  if (generation == 0)
    return("Chicken!");
  else
    return(egg(generation-1));
}
```

```
char *egg( int generation ) {
  if (generation == 0)
    return("Egg!");
  else
    return(chicken(generation-1));
}
```

Function Prototypes

- A **Function prototype** can be used to *tell* the compiler what a function looks like
  - So that it can be called even though the compiler has not yet seen the function definition.
- A function prototype specifies the **function name**, **return type** and **parameter types**.
  - But, it never comprises any function body!

Prototypes - Example code

```
int counter( void );
```

Prototype for function **`counter`**

```
int main(void) {
  cout << counter() << endl;
  cout << counter() << endl;
  cout << counter() << endl;
}
int count = 0;

int counter( void ) {
  count++;
  return(count);
}
```

Prototypes for chicken-egg-Example

```
char *egg( int );
char *chicken( int );
```

Prototypes

```
int main(void) {
   int startnum;

   cout << "Enter starting generation of
   your chicken" << endl;
   cin >> startnum;
   cout << "Your chicken started as a " <<
            chicken(startnum) << endl;
   return(0);
}
```

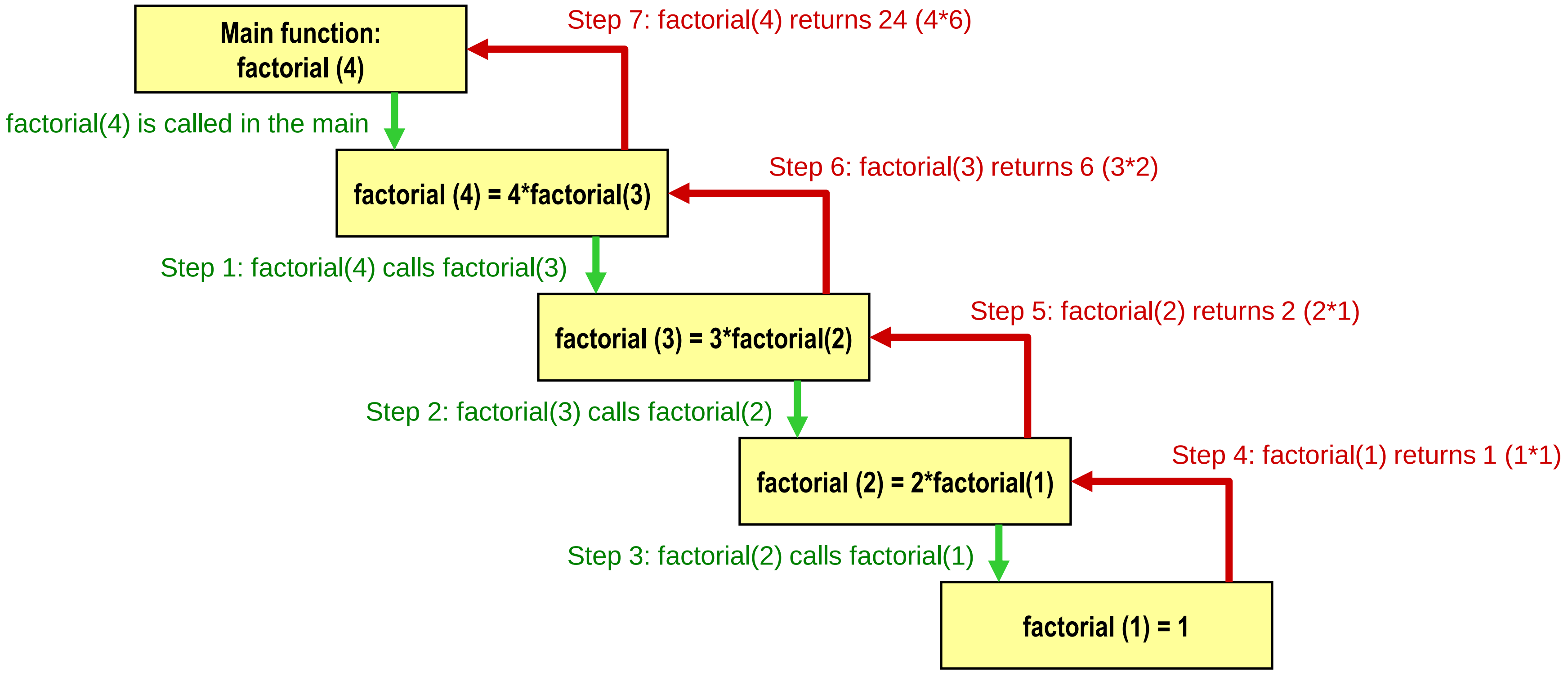
Recursion

- Functions can call themselves!
  This is called (direct) recursion.
  - The chicken-egg examples contains indirect recursion
- Recursion can be useful – There are problems with a quite simple recursive solution but without a simple iterative solution.
- Example: Tower of Hanoi problem

Recursive Example - Computing Factorials

```
int factorial( int x ) {
  if (x <= 1)
    return(1);
  else
    return(x * factorial(x-1));
}
```

Computing Factorial



A recursive Chicken-Egg …

```
char *chicken_or_egg( int gen ) {
   if (gen == 0)
      return("Chicken!");
   else if (gen == 1)
      return("Egg!");
  else
      return(chicken_or_egg(gen-1));
}
```

Designing Recursive Functions

- Define “Base Case”:
  - The situation in which the function does **not** call itself.
- Define “recursive step”:
  - Compute the return value the help of the function itself.

Recursion Base Case

- The base case corresponds to a case in which you know the answer (the function returns the value immediately), or can easily compute the answer.
- If you don’t have a base case you can’t use recursion! (and you probably don’t understand the problem).

Recursive Step

- Use the recursive call to solve a **sub-problem.**
  - The parameters must be different (or the recursive call will get us no closer to the solution).
  - You generally need to do something besides just making the recursive call.

Variables, Scopes and Storage Classes

Block Variables

- You can also declare variables that exist only within the *body* of a compound statement *(a block*)*:*

```
{
int foo;
        ...
        ...
}
```

Global variables

- You can declare variables outside of any function definition – these variables are *global variables*.
- Any function can access/change global variables.
- Example: flag that indicates whether debugging information should be printed.

Scope of variables

- Remember: The ***scope*** of a variable is the portion of a program where the variable has meaning (where it exists).
- A variables scope starts with its definition, it is never known before its definition (declaration)!
  - A global variable has global scope. *(until end of file)*
  - A local variable's scope is restricted to the function that declares the variable. *(until end of function)*
  - A block variable's scope is restricted to the block in which the variable is defined. *(until end of block)*

Example: Block Scope

```
int main(void) {
  int y;

  {
    int a = y;
    cout << a << endl;
  }
  cout << a << endl;
}
```

Error – a doesn't exist outside the block!

Scopes: Example

```
void foo(void) {
  for (int j=0;j<10;j++) {
     int k = j*10;
     cout << j << “,” << k << endl;
     {
     int m = j+k;
     cout << m << “,” << j << endl;
     }
  }
}
```

Storage Class

- Each variable has a *storage class*.
  - Determines the period during which the variable exists *in memory*.
  - Some variables are created only once (memory is set aside to hold the variable value)
    - Global variables are created only once.
  - Some variables are re-created many times
    - Local variables are re-created each time a function is called.

Storage Classes

- **`static`** – created only once, even if it is a local variable.
- **`extern`** – global variable defined elsewhere.
- **`auto`** – deprecated
- **`register`** – deprecated

Specifying Storage Class

```
static char remember_me;

extern double a_global;
```

Storage Class cont.

- Declaring a local variable as **`static`** means it will *remember* it's last value (it's not destroyed and recreated each time it's scope is entered).
  - Local variables are **`auto`** by default. (created each time the block in which they exist is *entered*.)

`static` example

```
int countcalls(void) {
  static int count = 0;
  count++;
  return(count);
}
...
cout << countcalls() << endl;
cout << countcalls() << endl;
cout << countcalls() << endl;
```