



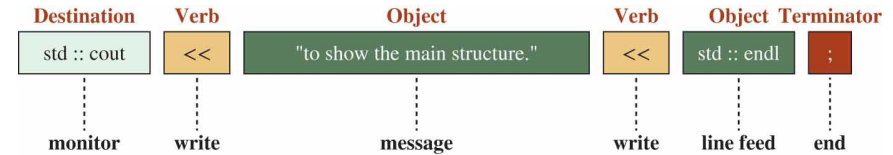
Object-oriented Programming

YoungWoon Cha
CSE Department
Spring 2023

Review

C++ Basics

- ❑ Input/Output Streams
- ❑ Namespace, Std Library
- ❑ Preprocessor, #include <>, ""
- ❑ Visual Studio, Debugging



```
using namespace std;
```

```
.....  
cout << "Hello" << endl;
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {  
    cout << "Hello\n";  
    cout << "This is a First Trial."  
    return 0;  
}
```

Preprocessor

```
namespace kitae {  
    int f();  
    void m();  
}
```

kitae.h

```
namespace mike {  
    int f();  
    int g();  
}
```

mike.h

```
#include "mike.h"  
  
namespace kitae {  
    int f() {  
        return 1;  
    }  
  
    void m() {  
        f();  
        mike::f();  
    }  
}
```

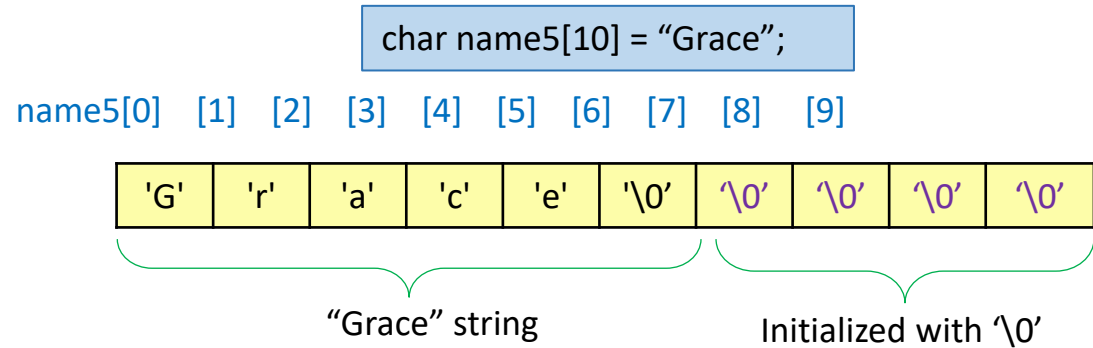
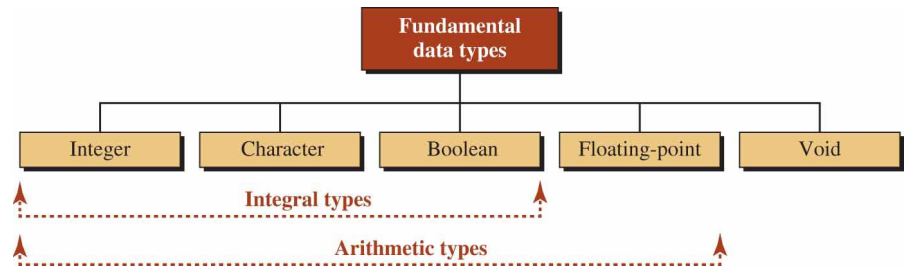
kitae.cpp

```
namespace mike {  
    int f() {  
        return -1;  
    }  
  
    int g() {  
        return 0;  
    }  
}
```

mike.cpp

C++ Basics

- ❑ Primitive Data Types
- ❑ sizeof, special characters
- ❑ C-String
- ❑ String Class

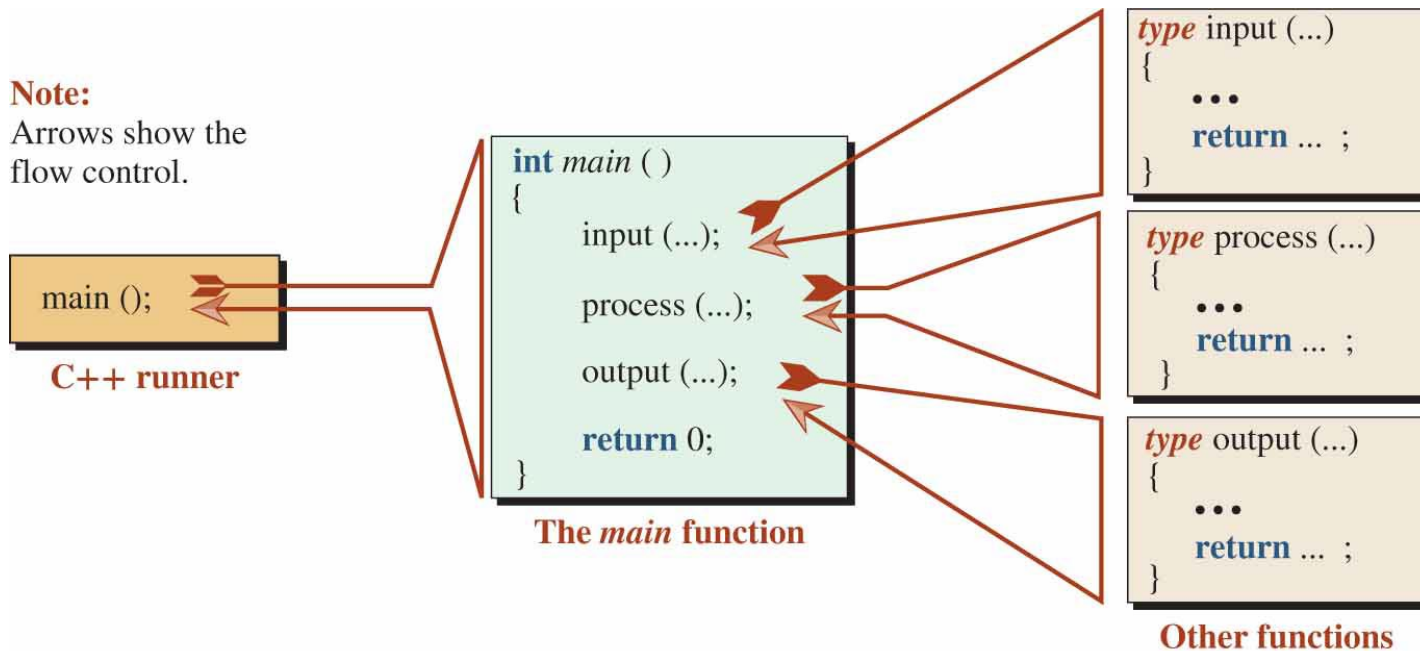


Functions

Functions

A function is an entity designed to do a task; it has a header and a set of statements enclosed in opening and closing braces.

Figure 6.1 *A program made of several functions*



Using Functions

- ❑ What is the benefit of dividing a task into several small tasks using functions while each function involves a little overhead?

Figure 6.12 *Setting function definition after function call*

Declaration	<code><i>type</i> name (<i>type</i>, <i>type</i>, ...);</code>	Note: The definition goes after the function call, but we need the function declaration before the function call.
Call	<code>name (<i>arg1</i>, <i>arg2</i>, ...)</code>	
Definition	<code><i>type</i> name (<i>type1</i> p1, <i>type2</i> p2, ...) { ... }</code>	

A semicolon is needed at the end of a function declaration.

Benefits Part 2

Easier to Write Simpler Task

Everyone knows that doing simple tasks is easier than doing difficult tasks.

It is easier to concentrate on manufacturing tires for cars than manufacturing the whole car.

Error Checking (Debugging)

One of the big headaches of programming is finding errors (called bugs).

Error checking (or debugging) is much more simple when a program is divided into small functions.

Each function can be debugged and then put together as a one program.

Benefits Part 2

Reusability

We can see the reuse of a small task in many large tasks. If we isolate these small tasks and write a function for each, we can create many large programs by assembling these small functions.

We do not need to rewrite the small task over and over again.

Library of Functions

Some common tasks that involve interaction with the operating system and the computer hardware are pre-written and made available to the user.

We can use these functions without writing the code for them.

Definition, Declaration, and Call Part 1

To work with a function, we need to think about three entities: *function definition*, *function declaration*, and *function call*.

Function Definition

Defining a function means to create the function.

Figure 6.2 *Syntax of function definition*

```
return-type function-name (parameter list)
{
    Body
}
```

Definition, Declaration, and Call Part 2

The following shows the definition of a function that finds and returns the larger of two given integers.

```
int larger(int first, int second)  // header
{
    int temp;
    if (first > second)
    {
        temp = first;
    }
    else
    {
        temp = second;
    }
    return temp;
}
```

Definition, Declaration, and Call Part 3

Function Declaration

The *function declaration* (also called function prototype) is only the header of the function followed by a semicolon.

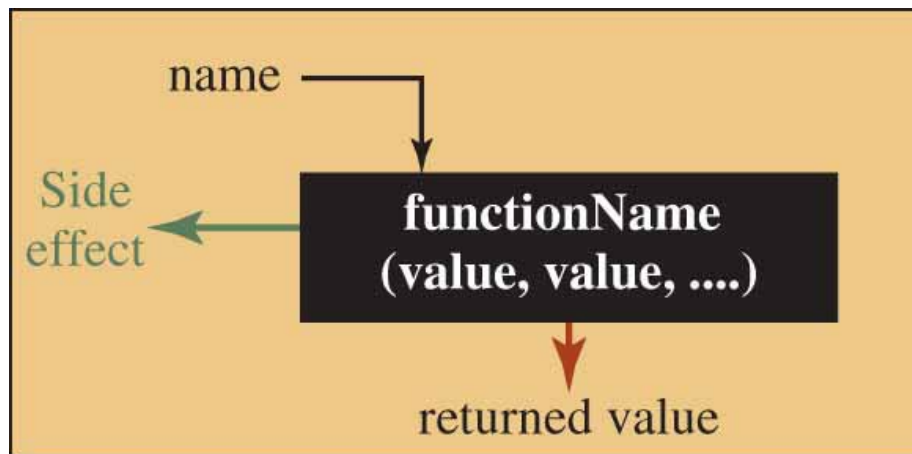
The parameter names are optional; the types are required.

```
int larger(int first, int second); // Using parameter names
int larger(int, int);             // Not using parameter names
```

Definition, Declaration, and Call Part 4

Function Call

A *function call* is a postfix expression that invokes a function to do its job.



Note:

A function call can have a side effect, a return value, or both.

```
int main()
{
    cout << larger(3, 13);
    cout << larger(10, 12);
    cout << larger(2, 12);
    return 0;
}
```

Arguments and Parameters

A *parameter* is a variable declared in the header of the function definition; an *argument* is a value that initializes the parameter when the function is called.

In other words, a parameter is like a variable at the left-hand side of an assignment statement; the corresponding argument is like the value used in the right-hand side of the assignment statement.

```
void fun(int x) // Function definition, x is a parameter
{
    ...
}
```

```
int main()
{
    ...
    fun(5); // Function call; 5 is an argument
    ...
}
```

Library and User-Defined Functions

To be able to use functions in a program, we need to have a *function definition* and a *function call*.

However, there are two cases: library functions and user-defined functions.

Library Functions

There are predefined functions in the C++ library.

We need only the declaration of the functions to be able to call them.

We study some of these functions in the next sections.

User-Defined Functions

There are a lot of functions that we need, but there are no predefined functions for them.

We need to define these functions first and then call them. We learn how to do that later in the chapter.

LIBRARY FUNCTIONS

If there is a predefined function in library, we do not need to worry about the function definition.

We only add the corresponding header file in which the function is defined.

However, we need to call the function, which means that we need to know the declaration of the function.

Mathematical Functions

C++ defines a set of mathematical functions that can be called in our program.

These functions are predefined and collected under the `<cmath>` header.

We need the `<cmath>` header to use the mathematical functions.

Numeric Functions

The numeric functions are used in numeric calculations.

The parameters are normally one or more numeric values and the result is also a numeric value.

Numeric Functions Defined in <cmath>

Table 6.2 *Some numeric functions defined in <cmath>*

Function Declaration	Explanation
type abs (type x);	Returns absolute value of x
type ceil (type x);	Returns largest integral value less than or equal to x.
type floor (type x);	Returns smallest integral value less than or equal to x
type log (type x);	Returns the natural (base e) logarithm of x
type log10 (type x);	Returns the common (base 10) logarithm of x
type exp (type x);	Returns e^x
type pow (type x, type y);	Returns x^y Note that y can also be of type int
type sqrt (type x);	Returns the square root of x ($x^{1/2}$)

Table 6.3 *Trigonometric functions*

Function Declaration	Function Explanation	Argument Units	Returned Range
type cos (type x);	Returns the cosine	radians	[-1, +1]
type sin (type x);	Returns the sine	radians	[-1, +1]
type tan (type x);	Returns the tangent	radians	any
type acos (type x);	Returns the inverse cosine	[-1, +1]	[0, p]
type asin (type x);	Returns the inverse sine	[-1, +1]	[0, p]
type atan (type x);	Returns the inverse tangent	any	[-p/2, +p/2]

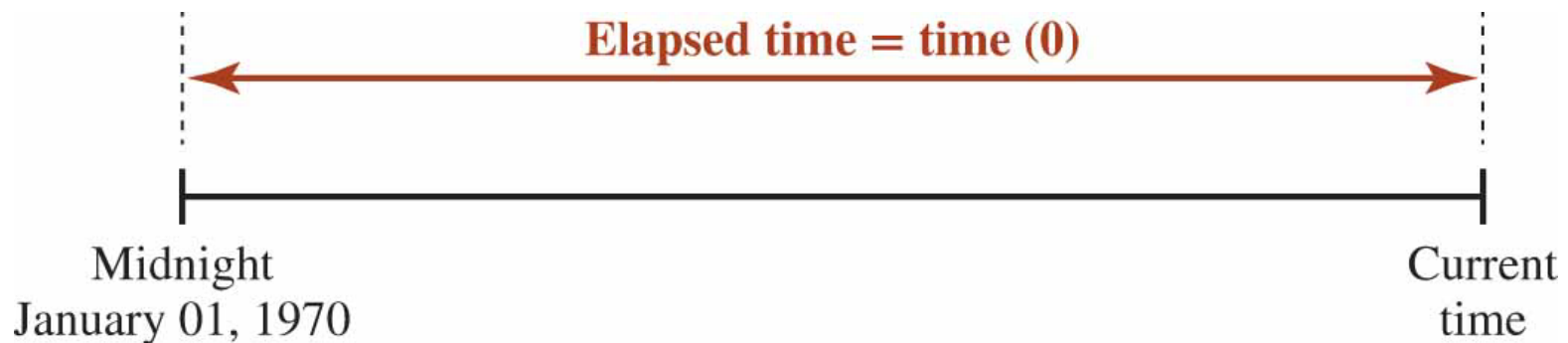
Handling Time

One of the library functions we often encounter in C++ programming is the time function defined in the `<ctime>` header file.

This function returns the number of seconds elapsed from the Unix epoch (midnight of January 1, 1970) when the argument passed to the function is 0.

In other words, `time(0)` gives the count of seconds from the epoch until the time that the function is called.

Figure 6.5 *Number of seconds defined by time (0) function*



Finding Current Time Part 1

Program 6.6 Finding current time

```
1  /*****
2   * A program finding the current time using time (0) function *
3   *****/
4  #include <iostream>
5  #include <ctime>
6  using namespace std;
7
8  int main ( )
9  {
10     // Finding elapsed seconds and current second
11     long elapsedSeconds = time (0);
12     int currentSecond = elapsedSeconds % 60;
13     // Finding elapsed minutes and current minute
14     long elapsedMinutes = elapsedSeconds / 60;
15     int currentMinute = elapsedMinutes % 60;
16     // Finding elapsed hours and current hour
17     long elapsedHours = elapsedMinutes / 60;
18     int currentHour = elapsedHours % 24;
19     // Printing current time
20     cout << "Current time: ";
```

Finding Current Time Part 2

Program 6.6 *Finding current time*

```
21     cout << currentHour << " : ";  
22     cout << currentMinute << " : "  
23     cout << currentSecond;  
24     return 0;  
25 }
```

Run:

Current time: 20 : 57 : 59

Run:

Current time: 20 : 58 : 22

Random Number Generation

The C++11 standard defined classes to create random numbers in any distribution as defined in probability theory.

Here we introduce the random number generator inherited from the C language defined in the `<cstdlib>` header file.

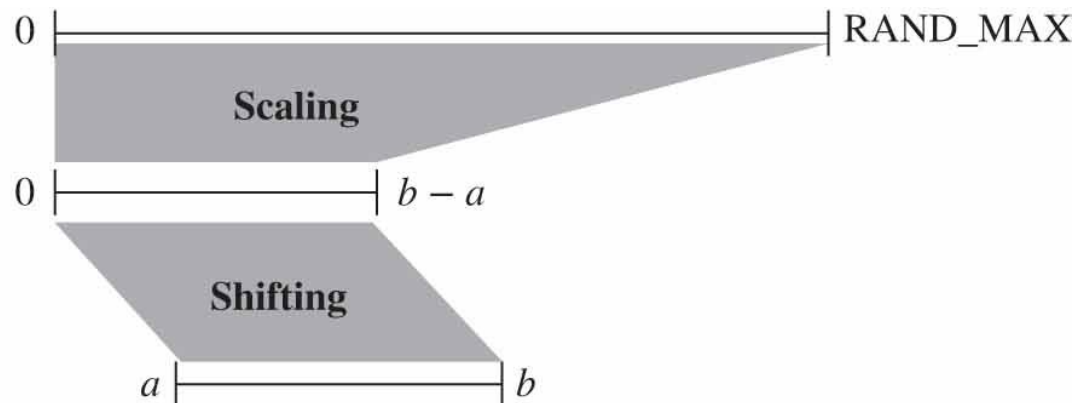
Figure 6.6 *Scaling and shifting a random number*

Scaling

```
temp = rand() % (b - a + 1)
```

Shifting

```
result = temp + a
```



```
v1 = rand() % 100;           // v1 in the range 0 to 99
v2 = rand() % 100 + 1;       // v2 in the range 1 to 100
v3 = rand() % 30 + 1985;     // v3 in the range 1985-2014
```

A Guessing Game Part 1

Program 6.7 A guessing game

```
1  /*****
2  * A program simulating a guessing game using a random number *
3  *****/
4  #include <iostream>
5  #include <cstdlib>
6  #include <ctime>
7  using namespace std;
8
9  int main ( )
10 {
11     // Declaration and initialization
12     int low = 5;
13     int high = 15;
14     int tryLimit = 5;
15     int guess;
16     // Generation of random number
17     srand (time (0));
18     int temp = rand();
19     int num = temp % (high - low + 1) + low;
20     // Guessing loop
```

A Guessing Game Part 2

Program 6.7 A guessing game

```
21  int counter = 1;
22  bool found = false;
23  while (counter <= tryLimit && !found)
24  {
25      do
26      {
27          cout << "Enter your guess between 5 to 15 (inclusive): ";
28          cin >> guess;
29      } while (guess < 5 || guess > 15);
30
31      if (guess == num)
32      {
33          found = true;
34      }
35      else if (guess > num)
36      {
37          cout << "Your guess was too high!" << endl;
38      }
39      else
40      {
```


A Guessing Game Part 3

Program 6.7 A guessing game

```
41         cout << "Your guess was too low!" << endl;
42     }
43     counter++;
44 }
45 // Success response
46 if (found)
47 {
48     cout << "Congratulation: You found it. ";
49     cout << "The number was: " << num;
50 }
51 // Failure response
52 else
53 {
54     cout << "Sorry, you did not find it! ";
55     cout << "The number was: " << num;
56 }
57 return 0;
58 }
```

A Guessing Game Output

Program 6.7 A guessing game

Run:

```
Enter your guess between 5 to 15 (inclusive): 7
Your guess was too low!
Enter your guess between 5 to 15 (inclusive): 8
Congratulation: You found it. The number was: 8
```

Run:

```
Enter your guess between 5 to 15 (inclusive): 15
Your guess was too high!
Enter your guess between 5 to 15 (inclusive): 14
Your guess was too high!
Enter your guess between 5 to 15 (inclusive): 13
Your guess was too high!
Enter your guess between 5 to 15 (inclusive): 12
Your guess was too high!
Enter your guess between 5 to 15 (inclusive): 11
Your guess was too high!
Sorry, you did not find it! The number was: 10
```

User-defined Function

Figure 6.7 *Four categories of functions in C++*

```
void name ( )  
{  
    ...  
    return;  
}
```

Void with no parameters

```
void name (type parameter, ...)  
{  
    ...  
    return;  
}
```

Void with parameters

```
type name ( )  
{  
    ...  
    return value;  
}
```

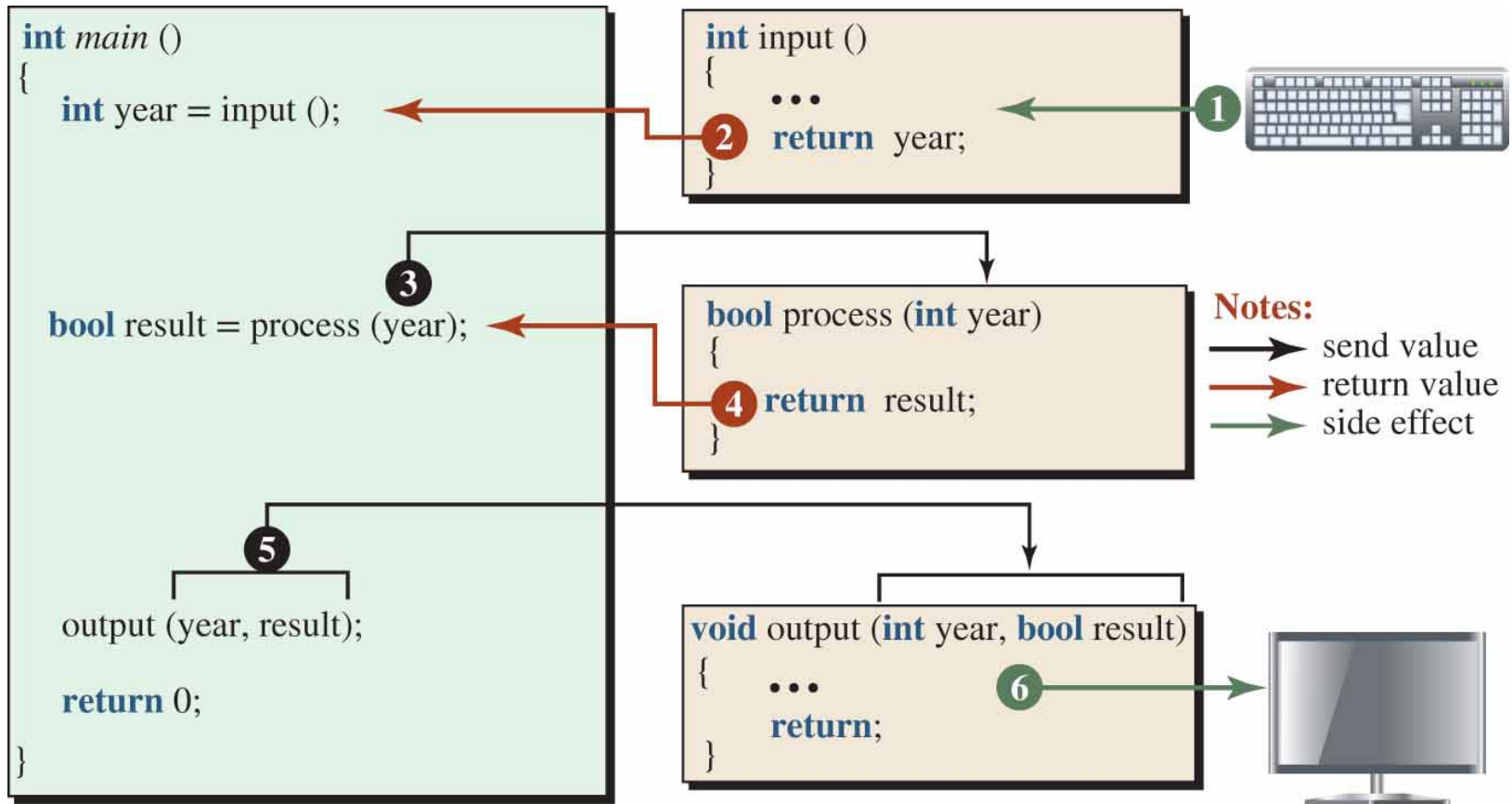
Value-returning with no parameters

```
type name (type parameter, ...)  
{  
    ...  
    return value;  
}
```

Value-returning with parameters

Communication Between main and Other Functions

Figure 6.13 *Communication between main and other functions*



Passing Data Part 1

If the parameter list of the called function is not empty, data is passed from each argument to the corresponding parameter.

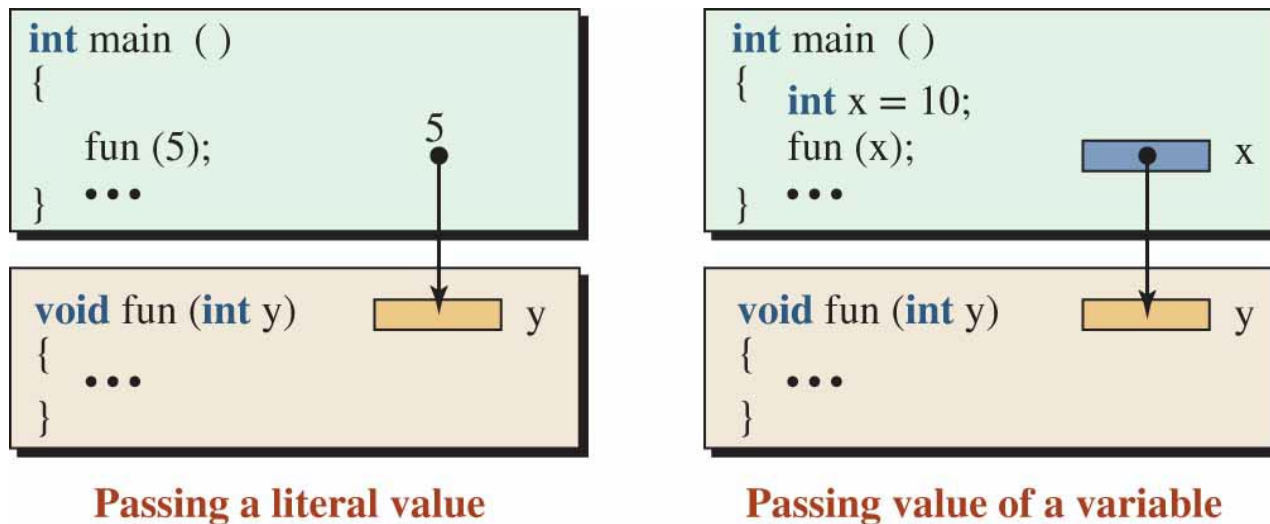
We can use three mechanisms for passing data from an argument to a parameter: *pass-by-value*, *pass-by-reference*, and *pass-by-pointer*.

Passing Data Part 2

Pass by Value

In the *pass-by-value* mechanism, the argument sends a copy of the data item to the corresponding parameter.

Figure 6.15 *The concept of pass-by-value*



Passing Data Part 3

We use *pass-by-value* when we do not want a called function changing the value of the arguments passed to it.

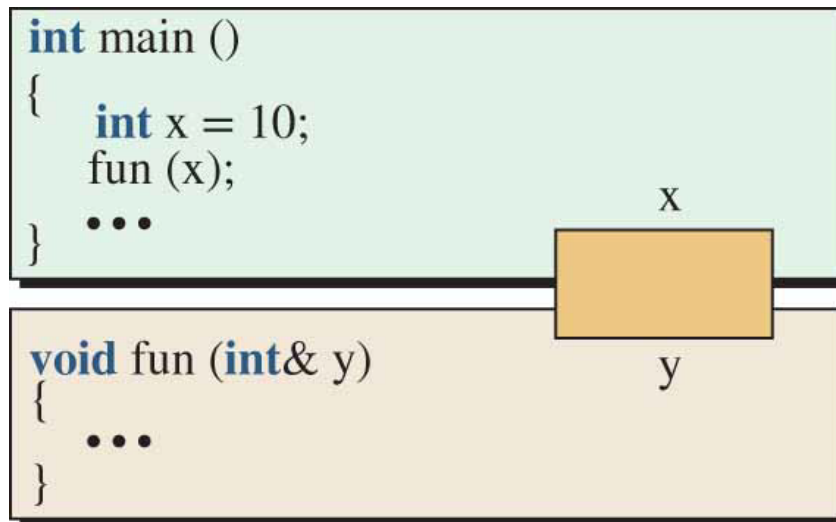
In other words, the called function can only read the value of the arguments; it cannot modify them.

In computer parlance, this is referred to as read-only access.

The pass-by-value method initializes a parameter with the copy of the corresponding argument.

Pass by Reference

Figure 6.16 *The concept of pass-by-reference*



Note:

No memory location is allocated in the called function. The called function uses the location allocated for the argument as its parameter.

Testing pass-by-reference

Program 6.14 *Testing pass-by-reference*

```
1  /*****
2   * This program shows how a change in the parameter can change the *
3   * corresponding argument when data is passed by reference.      *
4   *****/
5  #include <iostream>
6  using namespace std;
7
8  // Function declaration;
9  void fun (int& y); // The ampersand tell us that y is an alliance
10
11 int main ()
12 {
13     // Declaration and initialization of an argument
14     int x = 10;
15     // Calling function fun and passing x as an argument
16     fun (x);
17     // Printing the value of x to see change
18     cout << "Value of x in main: " << x << endl;
19     return 0;
20 }
```

Testing pass-by-reference

Program 6.14 Testing pass-by-reference

```
21  /*****
22  * Fun is a function that receives the value of y by reference,      *
23  * which means the parameter y is an alias for the argument x in    *
24  * the function call. This function increments its parameters which  *
25  * results in incrementing the argument in main.                    *
26  *****/
27  void fun (int& y)
28  {
29      y++;
30      cout << "Value of y in fun: " << y << endl;
31      return;
32  }
```

Run

```
Value of y in fun: 11
Value of x in main: 11 // Change of y in fun, has changed x in main
```

Advantages and Disadvantages

- a. Pass-by-value*** is very simple and protects arguments from being changed by the called function.

However, it has one disadvantage: the argument object needs to be copied and passed from the calling function to the called function.

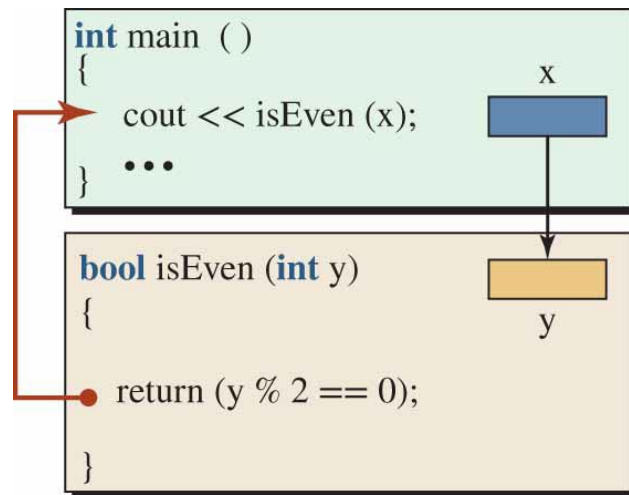
- b. Pass-by-reference*** changes the arguments in the calling function when the parameters are changed by the called function. If this is the purpose of the function, such as the case in swapping, the pass-by-reference is the best choice.
- c. Pass-by-pointer*** has the same advantage and disadvantage as pass-by-reference. It is normally avoided in C++, but it can be used if the nature of the data to be passed involves pointers.

Returning Data

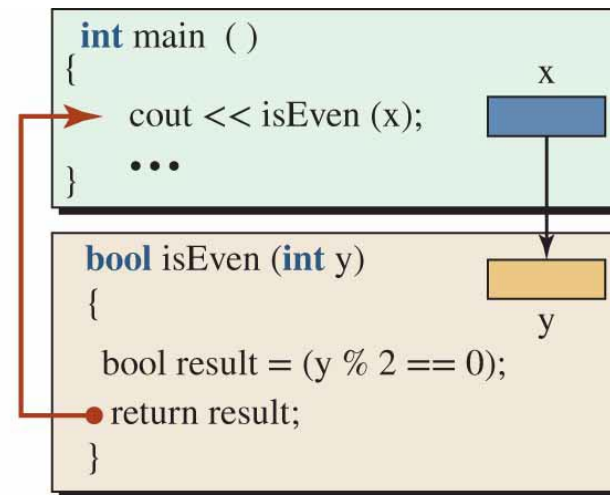
Data can be returned in three ways: *return-by-value*, *return-by-reference*, and *return-by-pointer*.

Return by Value

The called function creates an expression in its body and returns its value to the calling function. In this case, the function-call expression needs to be called in a situation that needs a value.



Returning a literal value



Returning value of a variable

Return by Reference

However, in object-oriented programming we encounter large objects to be returned. For efficiency we should return them by reference.

A problem occurs however when the called function creates the object because after it terminates the object does not exist anymore; it cannot be returned by reference.

Return by Pointer ✕

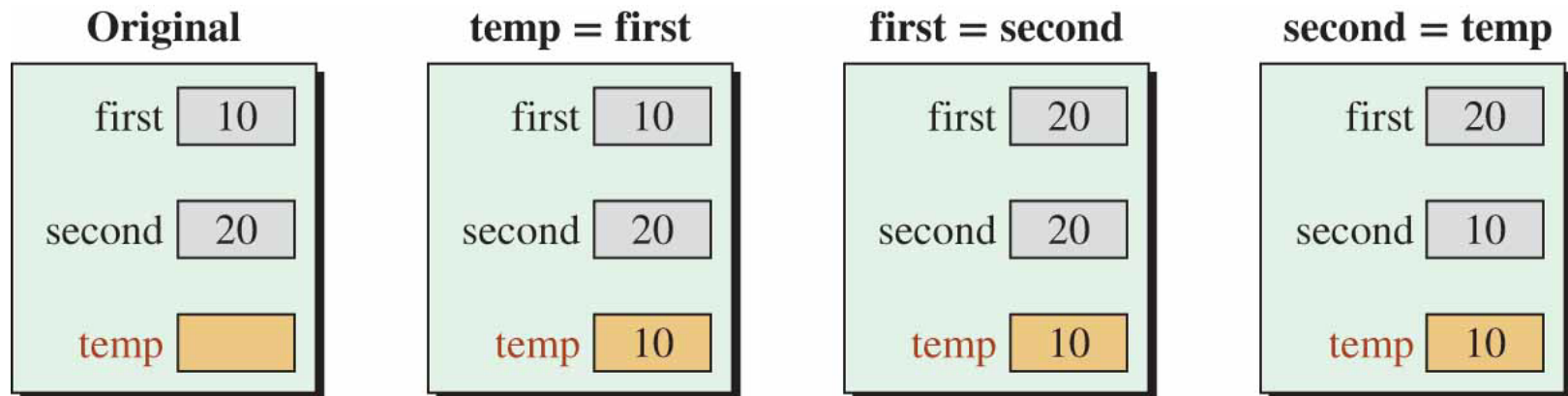
In the same way we can have return-by-pointer with the same effect as return-by-reference, but this practice is seldom used.

In-class Exercise

- ❑ Implement swap1 function using pass-by-reference
- ❑ Implement swap2 function using pass-by-pointer

In the *pass-by-pointer* mechanism, the argument sends a memory address to the corresponding parameter. The parameter stores the address and can access the value stored in the argument through that address.

Figure 6.17 *Using a temporary variable to swap two values*



Default Parameters

A program may call a function several times. If the function is designed with pass-by-value parameters, it may happen that the value of a parameter is the same most of the time.

In this case, we can use a parameter default value for one or more of the parameter values (including all parameters).

If only some parameters have default parameters, they must be the right-most ones.

The following declaration statement shows the declaration for a function named *calcEarnings* with a default of 40 hours.

```
double calcEarnings(double rate, double hours = 40.0) ;
```

Function Overloading

Can we have two functions with the same name?

The answer is positive if their parameter lists are different (in type, in number, or in order).

In C++, the practice is called overloading.

The criteria the compiler uses to allow two functions with the same name in a program is referred to as the function *signature*.

The *signature* of a function is the combination of the name and the types in the parameter list.

If two function definitions have different signatures, the compiler can distinguish between them.

Note that the return type of a function is not included in the signature.

The return value of a function is not included in its signature.

Function Overloading

The following shows how to define different functions to find the maximum between two integers and two floating-point values.

```
int max(int a, int b)
{
    ...
}
```

```
double max(double a, double b)
{
    ...
}
```

The following two functions are not recognized as two overloaded functions because their signatures are the same.

```
int get()
{
    ...
}
```

```
double get()
{
    ...
}
```

The compiler does not compile a program that contains two function definitions with the same signature.

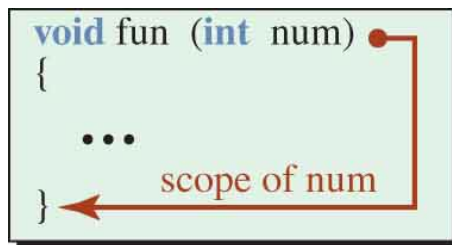
Scope

Scope is a region with a declaration.

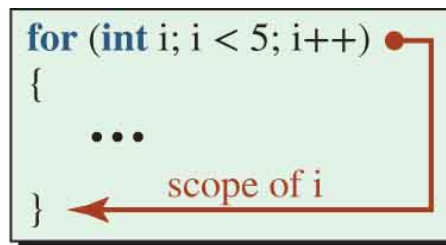
Local Scope

An entity that has local scope is visible from the point it is declared until the end of the block (defined by the closing brace).

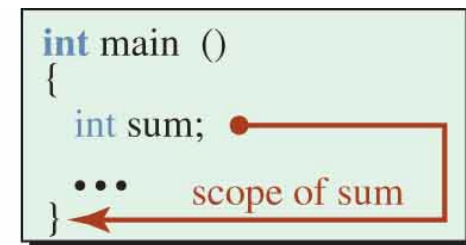
Figure 6.21 *Three cases of a local scope*



Scope of a parameter



Scope of a counter



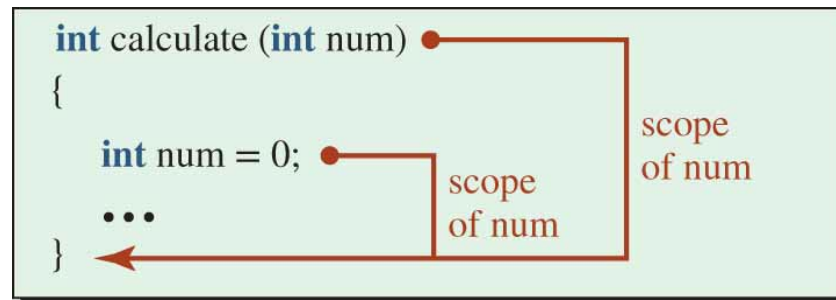
Scope of a local variable

Overlapped Scopes

We need to be aware that we cannot have two entities with the same name in a block.

This creates a compilation error.

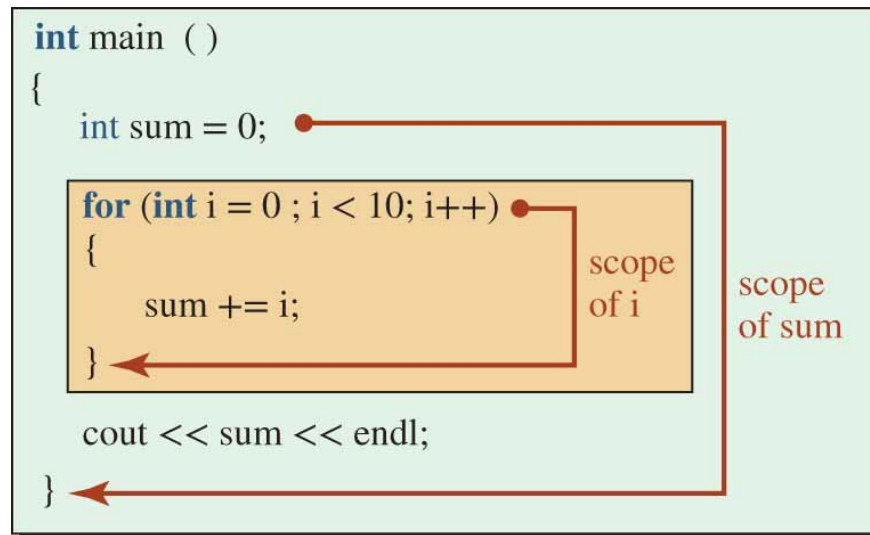
Figure 6.22 *Overlapping of scopes (error in this case)*



Error: Overlapping scopes in a single block

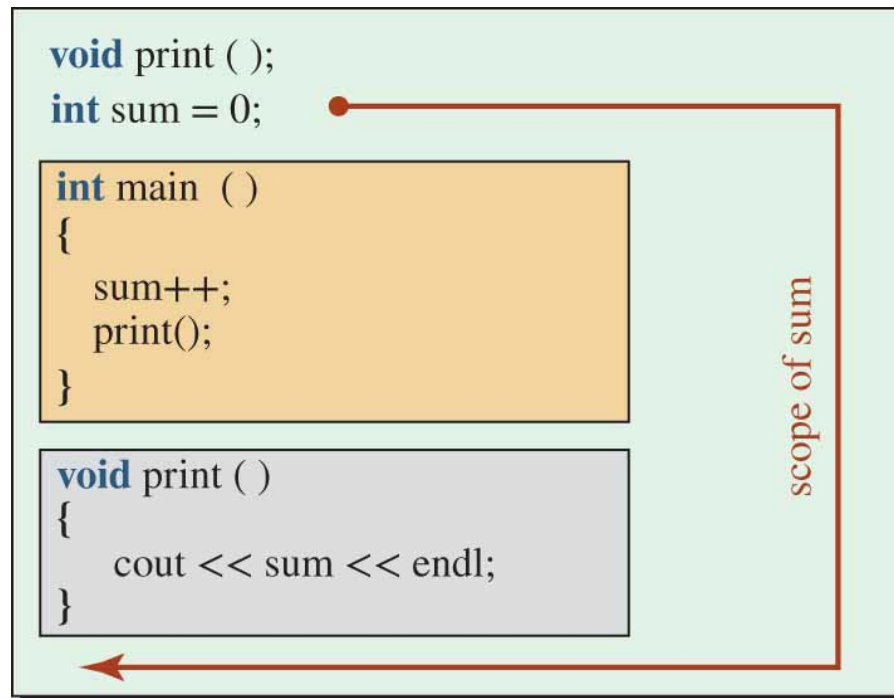
Scope in the Nested Block

Figure 6.23 *Scope in the nested block*




A nested block

Figure 6.24 *Scope of a global entity*



Shadowing in local scope

```
1  /*****
2  * A variable declared inside a block shadows another variable *
3  * with the same name declared outside the block                *
4  *****/
5  #include <iostream>
6  using namespace std;
7
8  int main ( )
9  {
10     int sum = 5;
11     cout << sum << endl;
12     {
13         int sum = 3;
14         cout << sum << endl; // The sum in the inner block is visible
15     }
16     cout << sum << endl; // The sum in the outer block is visible
17     return 0;
18 }
```



Run

5
3
5

Shadowing in global scope

```
1  /*****
2   * A program to test shadowing of global scope *
3   *****/
4  #include <iostream>
5  using namespace std;
6
7  int num = 5; // Global variable
8
9  int main ( )
10 {
11     cout << num << endl; // global num
12     int num = 25; // Local variable
13     cout << num; // local num shadows global num
14     return 0;
15 } // End of main
```

Run

5

25

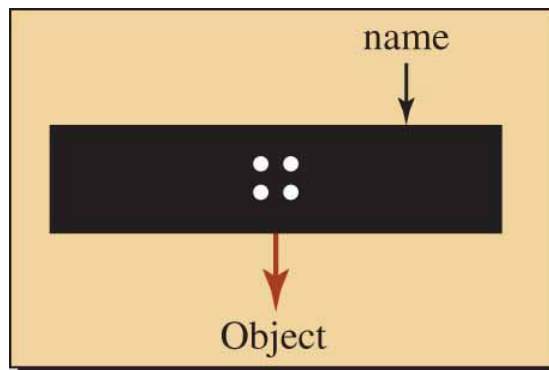
Scope Resolution Operator

Sometimes we may need to override the shadowing and access a global entity inside a local block.

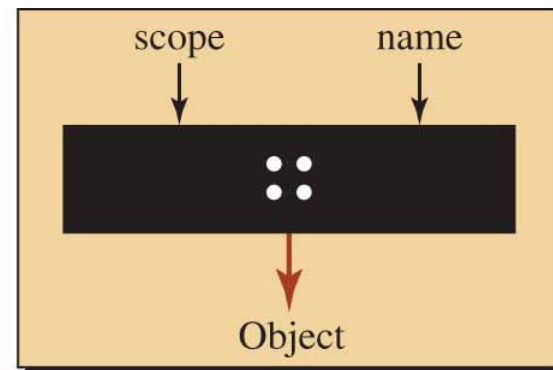
C++ provides an operator (`::`) that can explicitly or implicitly define the scope of the entity.

The first version takes one right operand; the second version takes two operands.

Figure 6.25 *Two versions of scope resolution operator*



Version with one operand



Version with two operands

Use of Scope Resolution Operator

Program 6.22 Using the scope resolution operator

```
1  /*****
2   * A program to test the use of scope operator          *
3   *****/
4  #include <iostream>
5  using namespace std;
6
7  int num = 5; // A global variable
8
9  int main ( )
10 {
11  int num = 25; // A local variable
12  cout << " Value of Global num: " << ::num << endl;
13  cout << " Value of Local num: " << num << endl;
14  return 0;
15 } // End of main
```

Run

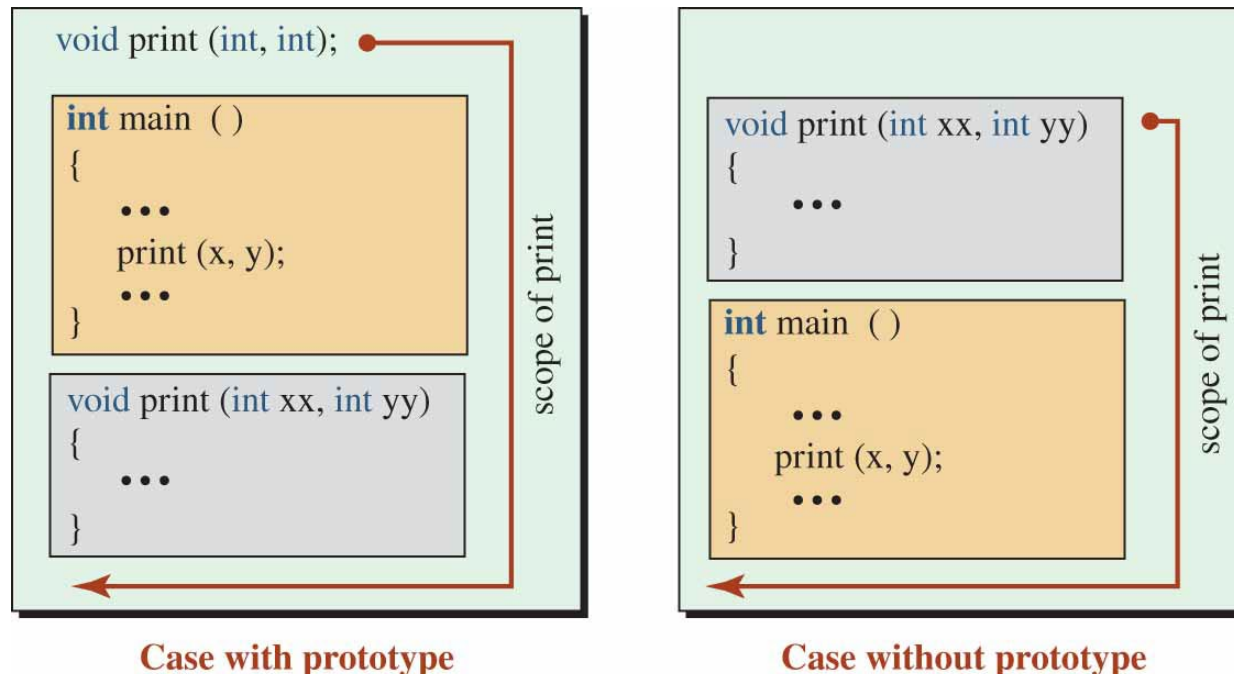
Value of Global num: 5
Value of Local num: 25

Scope of Function Name

A function is an entity with a name; it has a scope.

The scope of the function name is from the point that it is declared until the end of the program.

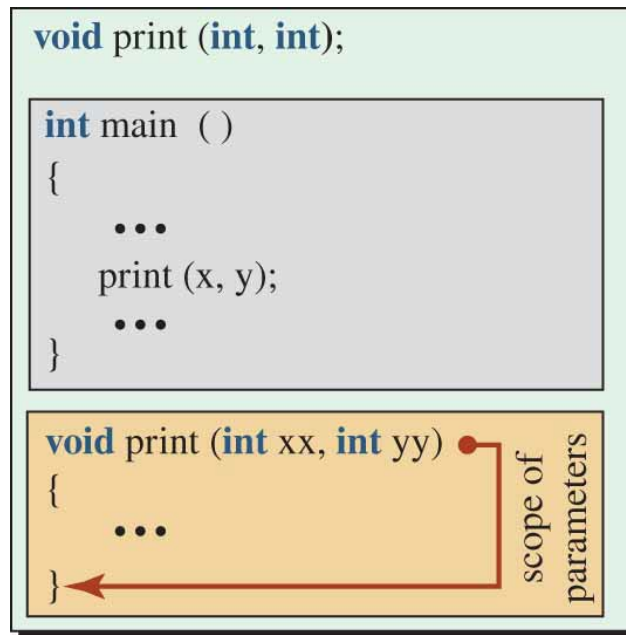
Figure 6.26 *Scope of function name*



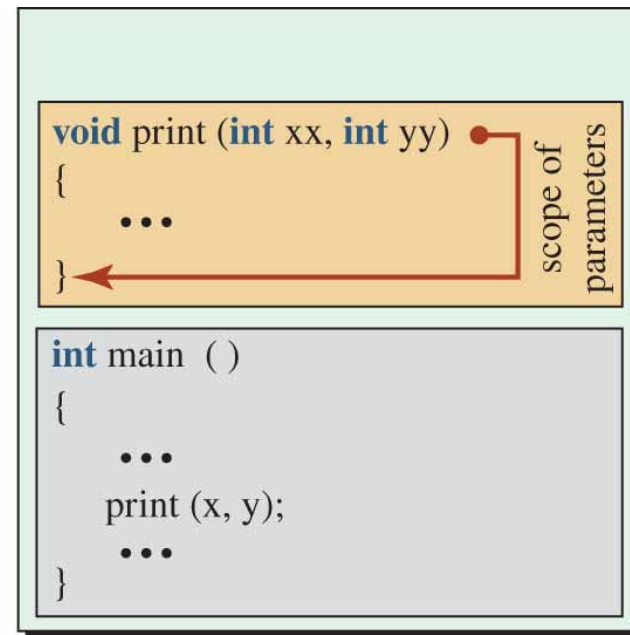
Scope of Function Parameters

The scope of function parameters starts from the point they are declared up to the point when the function block is terminated.

Figure 6.27 *Scope of function parameter*



Case with prototype



Case without prototype

Automatic Local Variable

An automatic local variable is born when the function is called and dies when the function terminates.

By default all local variables in a function have automatic lifetime, but we can explicitly use the modifier *auto* in front of the variable declaration to emphasize that they are reborn each time the function is called.

Test of Automatic Local Variable Part 1

Program 6.23 Testing automatic local variables

```
1  /*****
2   * A program to test the use of an automatic local variable      *
3   *****/
4  #include <iostream>
5  using namespace std;
6
7  void fun ( );
8
9  int main ( )
10 {
11     fun ( );
12     fun ( );
13     fun ( );
14     return 0;
15 }
16 /*****
17  * The function has two automatic local variables, num and count *
18  * In each function call, these variables are initialized.      *
19  *****/
20 void fun ( )
```

Test of Automatic Local Variable Part 2

Program 6.23 *Testing automatic local variables*

```
21 {  
22     int num = 3; // implicit auto variable  
23     auto int count = 0; // explicit auto variable  
24     num++;  
25     count++;  
26     cout << "num = " << num << " and " << "count = " << count << endl;  
27 }
```

Run

```
num = 4 and count = 1  
num = 4 and count = 1  
num = 4 and count = 1
```

Static Local Variable

A local variable can be made *static* by using the *static* modifier.

The lifetime of a static variable is the lifetime of the program in which it is defined.

A static variable is initialized only once, but the system keeps track of its contents as long as the program is running.

This means it is initialized in the first call to the function, but it can be changed during subsequent function calls.

We can change the variable *count* in the previous program to a static variable and see how the system keeps the value of *count* from call to call

Test of Static Variables Part 1

Program 6.24 Testing static variables

```
1  /*****
2   * A program to test the use of static variables      *
3   *****/
4  #include <iostream>
5  using namespace std;
6
7  void fun ( );
8
9  int main ( )
10 {
11     fun ( );
12     fun ( );
13     fun ( );
14     return 0;
15 }
16 /*****
17 * The function has one static variable named count. It is      *
18 * initialized in the first call, but it holds its value for    *
19 * the next time. This means value is incremented in each call *
20 *****/
```


Test of Static Variables Part 2

Program 6.24 Testing static variables

```
21 void fun ( )  
22 {  
23     static int count = 0; // explicit static variable  
24     count++;  
25     cout << "count = " << count << endl;  
26 }
```

Run

```
count = 1  
count = 2  
count = 3
```

Initialization

It is worthwhile to compare the initialization of a global variable, an automatic local variable, and a static local variable.

If an automatic local variable is not explicitly initialized, it holds the garbage left over from the previous use.

The global and static local variables on the other hand are initialized to default values (0 for integral types, 0.0 for floating-point types, and false for Boolean types) if they are not initialized explicitly.

In other words, global and static local variables behave the same with respect to initialization.

If not initialized explicitly, global and static local variables are initialized to default values, but local variables contain garbage left over from previous use.

Initialization of Different Variable Types

```
1  /*****
2  * A program to test the initialization of variables      *
3  *****/
4  #include <iostream>
5  using namespace std;
6
7  int global; // A global variable
8
9  int main ( )
10 {
11     static int sLocal; // A static local variable
12     auto int aLocal; // An automatic local variable
13     // Printing values
14     cout << "Global = " << global << endl;
15     cout << "Static Local = " << sLocal << endl;
16     cout << "Automatic Local = " << aLocal << endl;
17     return 0;
18 }
```

Run

Global = 0

Static Local = 0

Automatic Local = 4202830 // Garbage value

In-class Exercise

□ Given an array with all distinct elements, find the largest two distinct elements in an array.

- Define a function in different file other than main.cpp
- *Call the function in main()*
- *Print the largest two distinct elements in main()*

- Test set:

```
Input: arr[] = {10, 4, 3, 50, 23, 90}
```

```
Output: 90, 50
```

```
Input: arr[] = {99, 77, 11, 15, 88, 1}
```

```
Input: arr[] = {10,9636, 2401, 777, 2080, 1, 50}
```

Thank you

E-mail: youngcha@konkuk.ac.kr

