



## COMP 2012H Honors Object-Oriented Programming and Data Structures

### Topic 4: Function

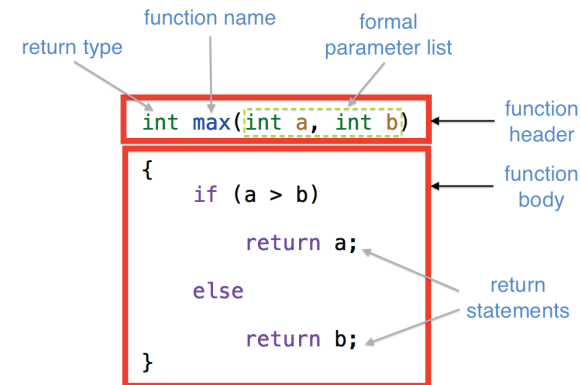
Dr. Desmond Tsoi

Department of Computer Science & Engineering  
The Hong Kong University of Science and Technology  
Hong Kong SAR, China



## Part I

### Function Basics



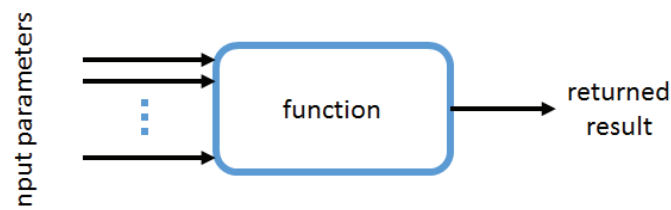
### Basic Function Syntax

#### Syntax: Function Definition

```
<return-type> <function-name> ( <formal-parameter-list> )
{ <function-body> }
```

#### Syntax: Function Call

```
<function-name> ( <actual-parameter-list> )
```



### Function Name

- Any legal C++ identifier can be used for `<function-name>`.
- Just like naming variables and constants, you should use meaningful names for function names.
  - The name should describe what the function does.
- The function name `"main"` is reserved; you must define it, and define it exactly **once**.
  - Recall that each program can only have one `"main( )"` function.
  - When a program is run, the **shell** — command interpreter of the operating system — looks for the `"main( )"` function and starts execution from there.



## Formal Parameter List & Actual Parameter List

- `<formal-parameter-list>` appears in the **function definition**, and is basically a list of **variable declarations** separated by **commas**.

Syntax: `<formal-parameter-list>`

`<type1 variable1>, <type2 variable2>, ..., <typeN variableN>`

- `<actual-parameter-list>` appears in a **function call**, and is a list of **objects** separated by **commas** that are passed to the called function.

Syntax: `<actual-parameter-list>`

`<object1>, <object2>, ..., <objectN>`

- There is a **one-to-one** correspondence between the **actual parameters** (aka **arguments**) and the **formal parameters**.

## Formal Parameter List & Actual Parameter List ..

- During the function call, the following initializations are performed,

```
< type1 variable1 >= object1,  
< type2 variable2 >= object2,  
:  
< typeN variableN >= objectN
```

- Since C++ is a **strongly typed** programming language, the data types of an actual parameter and its corresponding formal parameters must be the same.
- A C++ compiler will perform **type checking** to make sure that their types match with each other.
- Exception: unless an automatic type conversion — **coercion** — can be done, just like normal initialization or assignment of an object to a variable of a different type. (More later.)

## Function Header & Function Body

- In `<return-type> <function-name> ( <formal-parameter-list> )` is also called the **function header**, and the rest is the **function body** enclosed in curly braces.
- The `<function-body>` usually consists of the following parts:
  - ▶ **constant** declarations
  - ▶ **variable** declarations and definitions
  - ▶ other C++ statements
  - ▶ **return** statement
- It is legal to have an empty function body!
- The curly braces must be there, even if there is only **one** single statement inside the function body! (That is different from the **if**-statement or **while**-statement, etc. )

## Return Type

- Usually a function returns something — in C++, we call it an object.
- The returned object may be
  - ▶ a **signal** to tell the caller about the status of the function: does it run successfully? does it fail?
  - ▶ the **result** of some computation. e.g. factorial, sum, etc.
  - ▶ a **new object** created by the function. e.g. a new window.
- `<return-type>` specifies the data type of the **single** returned object.
- `<return-type>` can be any of the C++ built-in data types (e.g., char, int, etc.) or user-defined types, **except the array type**. (Array type will be talked later.)

**Question:** Since only a single object is returned by a function, how can you return multiple objects back to the caller?

## return Statement

### Syntax: return Statement

`return < expression > ;`

- The `return` statement generally returns “2” things to the caller:
  - ▶ **program control**: it stops running the called function, and the function caller takes back the control and continue its execution.
  - ▶ **an object**: the object (or value) represented by the `< expression >` is returned to the caller.
- The value of `< expression >` in the `return` statement should have the same type as the `<return-type>`. Or, if it can be converted to the `<return-type>` by **coercion**, otherwise it will be a compilation error.
- If a function has a return value, the function body must have at least one `return` statement.

## Example: max

```
#include <iostream>      /* File: max.cpp */
using namespace std;     /* To find the greater value between x and y */

int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
} // Question: can you write with only 1 return statement?

int main()
{
    int x, y;
    cout << "Enter 2 numbers: ";
    cin >> x >> y;

    cout << "The bigger number is " << max(x, y) << endl;
    return 0;
}
```

## void: a New Type

- “**void**” means *nothing, emptiness*.
- A function that returns nothing back to the caller has a **return type** of `void`.
- A function that does not take any arguments from the caller may
  - ▶ leave the `<formal-parameter-list>` empty.

```
int fcn_example() { ... }
```

- ▶ put the `<formal-parameter-list>` as `void`.

```
void print_hkust(void) { cout << "hkust" << endl; }
```

## Remarks: Why Function?

- When you have several segments of codes doing similar things, then they are good candidates for a function.
- A function allows “**write-once-call-many**”: you only write it once they can be called **several times** in the same program with the same or different arguments.
- Functions make programs **easier** to **understand**.
- Functions make programs **easier** to **modify**.
- Functions allow **reusable code**. (e.g. `log`, `sqrt`, `sin`, etc.)
- Functions separate the **concept** (what is done) from the **implementation** (how it is done).
- The last two remarks lead to the creation of **binary libraries** which are a set of compiled functions. These libraries can be **shared**, yet the users do **not** know their implementation. (You’ll learn how to do this later.)

# Parameter Passing Methods



## How Actual Parameters are Passed to Formal Parameters

ACTUAL PARAMETER		FORMAL PARAMETER
$object_1$	$\longrightarrow$	$variable_1$
$object_2$	$\longrightarrow$	$variable_2$
$\vdots$	$\vdots$	$\vdots$
$object_N$	$\longrightarrow$	$variable_N$

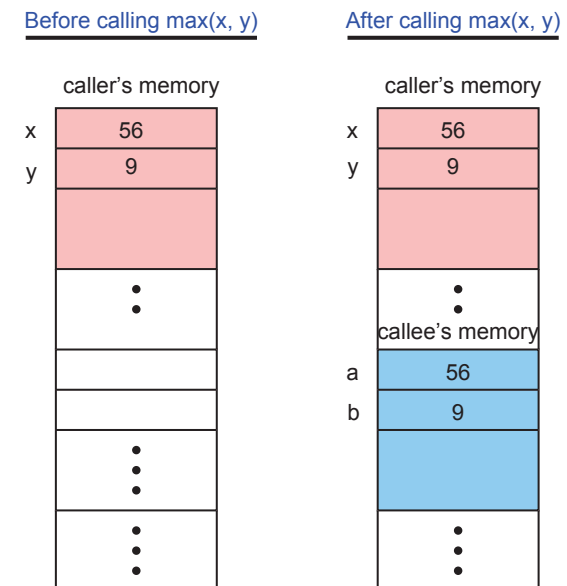
- C++ supports 2 ways to pass arguments to a function:
  1. **pass-by-value** (PBV), or call-by-value
  2. **pass-by-reference** (PBR), or call-by-reference
- Notice that if you call a function with an **expression**, the expression is first **evaluated**, and the **result** is then passed to the function.  
e.g. `max(3 + 5, 2 + 9)` → `max(8, 11)` before calling the max function.

## Pass-by-Value

- In **pass-by-value**, the **value** of an actual parameter is **copied** into the formal parameters of the function.
- If the actual parameter is a **literal constant** (e.g. calling `max(2, 3)` ), obviously it won't change.
- If the actual parameter is a **variable** (e.g. calling `max(x, y)` ), only its **value** is **copied** to the function, otherwise it has nothing to do with the operation of the function. In particular, its value **cannot be modified** by the function.
- All the function examples presented so far use **pass-by-value** to pass the arguments.

**Question:** What happens if the argument is a big object (e.g. of several MB)?

## Pass-by-Value Illustration



## Reference Variable

### Syntax: Reference Variable Definition

```
<type>& <variable1> = <variable2>;  
<type> &<variable1> = <variable2>;  
...
```

- A **reference variable** is an **alias** of another variable.
- A **reference variable** must always be **bound** to an object. Therefore, it must be **initialized** when they are **defined**.
- Once a **reference variable** is defined and bound with a variable, you **cannot** “re-bind” it to another object.

In the example,

- Variables *a*, *x*, *w* all refer to the **same** integer **object**; similarly, variables *b*, *y*, *z* also all refer to the **same** integer **object**.
- Variables *a*, *x*, *w* share the **same memory space**, so that you may modify the value in that memory space through **any** of them! (Same for *b*, *y*, *z*.)
- In the line `z = a;`, the **reference variable** *z* is not re-bound to *a*, but the value of *a* is assigned to *z*.

## Example: Reference Variables

```
#include <iostream>      /* File: ref-declaration.cpp */  
using namespace std;  
  
int main()  
{  
    int a = 1, b = 2;  
    int& x = a;           // now x = a = 1  
    int &y = b;           // now y = b = 2  
    int &w = a, &z = y;   // now w = a = x = 1, z = b = y = 2  
  
    a++; cout << a << '\t' << x << '\t' << w << endl;  
    x += 5; cout << a << '\t' << x << '\t' << w << endl;  
    a = w - x; cout << a << '\t' << x << '\t' << w << endl;  
  
    y *= 10; cout << b << '\t' << y << '\t' << z << endl;  
    b--; cout << b << '\t' << y << '\t' << z << endl;  
    z = 999; cout << b << '\t' << y << '\t' << z << endl;  
  
    z = a;                // that is not re-binding z to a  
    cout << b << '\t' << y << '\t' << z << endl;  
    return 0;  
}
```

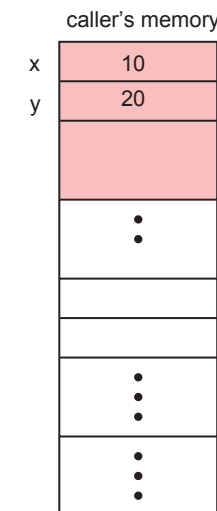
## Pass-by-Reference Example: swap

```
/* File: pbr-swap.cpp */  
#include <iostream>  
using namespace std;  
  
void swap(int& a, int& b)  
{  
    int temp = a;  
    a = b;  
    b = temp;  
}  
  
int main()  
{  
    int x = 10, y = 20;  
    swap(x, y);  
    cout << "(x , y) = " << '(' << x  
        << " , " << y << ')' << endl;  
    return 0;  
}
```

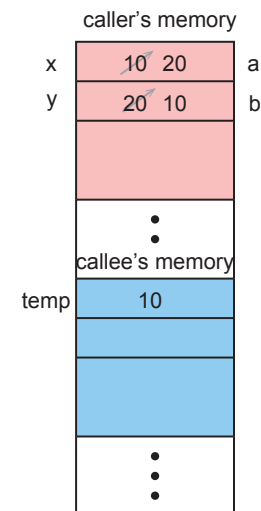
```
// execution of swap is  
// equivalent to running  
// the following codes  
int& a = x;  
int& b = y;  
int temp = a;  
a = b;  
b = temp;  
  
// OR, equivalently  
int temp = x;  
x = y;  
y = temp;
```

## Pass-by-Reference Illustration

Before calling swap(x, y)



After calling swap(x, y)



## Pass-by-Reference

- **Pass-by-reference** does not copy the value of **actual parameters** to the **formal parameters** of the function.
- When an **actual parameter** is **passed by reference**, its corresponding **formal parameter** becomes its **reference variable** (alias).
- In the swap example, on entering the swap function, the following codes are run: `int& a = x; int& b = y;` That is, the **formal parameters** *a* and *b* are declared as **reference variables** and are initialized or bound to their corresponding **actual parameters** *x* and *y* respectively.
- You must add the symbol “&” after the type name of the **formal parameter** if you want **pass-by-reference**.
- When an **actual parameter** is **passed by reference** to its **formal parameter**, since they **share** the **same memory**, any modification made to the **formal parameter** also changes the value of the corresponding **actual parameter**.

## Remarks ...

- All the **local variables** defined inside a function, including the formal parameters, are **destroyed** on **return** of the function call.
  - ▶ These **local variables** are created **every time** the function is called.
  - ▶ These **local variables** created on the current call are **different** from those created in the previous calls.
  - ▶ However, if a formal parameter is a **reference variable**, only itself is destroyed when the function returns, the variable (actual parameter) bound to it still **exists** afterwards.
- **Pass-by-reference** is **more efficient** when a large object has to be passed to a function as **no copying** takes place. However, there is a **risk** that you may accidentally modify the object.

**Question:** Is there a way to pass a large object to a function such that the function cannot modify its value?

Example: Some PBV, Some PBR

```

#include <iostream>      /* File: sum-and-difference.cpp */
using namespace std;

// To find the sum and difference of 2 given numbers
void sum_and_difference(int x, int y, int& sum, int& difference)
{
    sum = x + y;
    difference = x - y;
}

int main()
{
    int x, y, sum, difference;
    cout << "Enter 2 numbers: ";
    cin >> x >> y;

    sum_and_difference(x, y, sum, difference);
    cout << "The sum of " << x << " and " << y << " is " << sum << endl;
    cout << "The difference between " << x << " and " << y << " is "
        << difference << endl;
    return 0;
}

```

## Part III

## Function Declaration and Function Definition





## Some Function Terminology

function prototype

`int max(int, int);`

↑ return type    ↑ name    signature

## Function Prototype

A **function prototype** consists of

1. **function name**
2. **return data type**
3. the **number** of formal parameters
4. the **data type** of the formal parameters

### Example: Function Prototypes

```
// int factorial(int n) { ... }  
int factorial(int);  
  
// float distance(float x1, float y1, float x2, float y2) { ... }  
float distance(float, float, float, float);  
  
/* void print_tree(int tree_height, char tree_symbol,  
    char trunk_symbol, char pot_symbol) { ... } */  
void print_tree(int, char, char, char);
```

## Function Prototype ..

- The **identifier names** of the formal parameters are not part of the signature as the names are **immaterial**.

### Example: Variable Names are Immaterial in a Function Prototype

```
/* All the following 3 function definitions are equivalent */  
  
int max(int x, int y) { return (x > y) ? x : y; }  
  
int max(int a, int b) { return (a > b) ? a : b; }  
  
int max(int f, int g) { return (f > g) ? f : g; }
```

- A **function prototype** describes the **interface** of the function: what parameters it takes in and what value it returns.
- Technically, a **function prototype** is also called the **application programming interface (API)**.

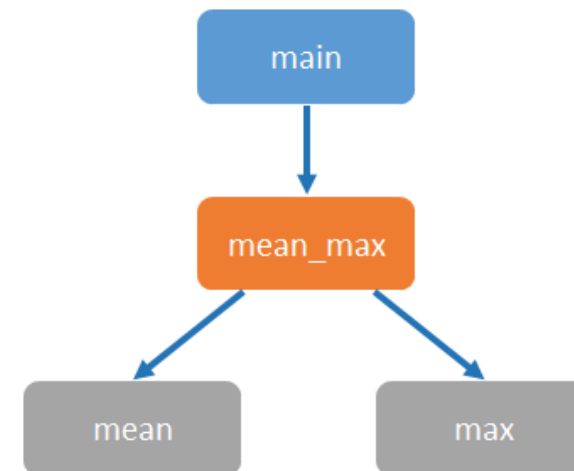
## Function Declaration vs. Definition

- A function is **declared** by writing down its interface — its **function prototype**.
- A function is **defined** by writing down its **function header** *plus* its **function body**.
- A **function definition** will ask the compiler to generate **machine codes** according to the C++ codes in its function body.
- A **function declaration** just informs the compiler about the function's **interface** *without* generating any machine codes.
- A function may be **declared many times**, but a function can be **defined only once**.
- Of course, when a function is **defined**, it is also **declared**.
- But, simply **declaring** a function does **not** **define** the function.

## Function Declaration vs. Definition ..

- In C++, all functions must be **declared** before they can be used, so that the compiler can
  - ▶ make sure the exact **number of arguments** are passed.
  - ▶ do **type checking** on the arguments passed to the function.
- That is, if function A wants to call function B, function B must be
  - ▶ **declared/defined before**, or
  - ▶ **declared inside** function A **before** calling function B.
- However, a function need not be defined before it can be used, although it must be defined **eventually somewhere** in the whole program in order that the program can be compiled to an executable.

## Example: A Program with 3 Levels of Functions



## Example: Declare Functions by Defining the Functions

```
#include <iostream>      /* File: fcn-prototype1.cpp */
using namespace std;

int max(int x, int y) { return (x > y) ? x : y; }
int mean(int x, int y) { return (x + y)/2; }

void mean_max(int x, int y, int& mean_num, int& max_num)
{
    mean_num = mean(x, y);
    max_num = max(x, y);
}

int main()
{
    int average, bigger;

    mean_max(6, 4, average, bigger);
    cout << "mean = " << average << endl << "max = " << bigger << endl;
    return 0;
}
```

## Example: Declare Functions Globally

```
#include <iostream>      /* File: fcn-prototype2.cpp */
using namespace std;

void mean_max(int, int, int&, int&); // main only needs to know mean_max

int main()
{
    int average, bigger;
    mean_max(6, 4, average, bigger);
    cout << "mean = " << average << endl << "max = " << bigger << endl;
    return 0;
}

int max(int, int);      // mean_max needs to know max and mean
int mean(int, int);

void mean_max(int x, int y, int& mean_num, int& max_num)
{
    mean_num = mean(x, y);
    max_num = max(x, y);
}

int max(int x, int y) { return (x > y) ? x : y; }
int mean(int x, int y) { return (x + y)/2; }
```



## Example: Declare Functions Locally

```
#include <iostream>      /* File: fcn-prototype3.cpp */
using namespace std;

int main()
{
    void mean_max(int, int, int&, int&);
    int average, bigger;

    mean_max(6, 4, average, bigger);
    cout << "mean = " << average << endl << "max = " << bigger << endl;
    return 0;
}

void mean_max(int x, int y, int& mean_num, int& max_num)
{
    int max(int, int);
    int mean(int, int);

    mean_num = mean(x, y);
    max_num = max(x, y);
}

int max(int x, int y) { return (x > y) ? x : y; }
int mean(int x, int y) { return (x + y)/2; }
```

## Example: Forward Function Declaration

```
#include <iostream>      /* File: odd-even.cpp */
using namespace std;

bool even(int);

bool odd(int x) { return (x == 0) ? false : even(x-1); }

bool even(int x) { return (x == 0) ? true : odd(x-1); }

int main()
{
    int x;
    cin >> x;           // Assume x > 0

    cout << boolalpha << odd(x) << endl;
    cout << boolalpha << even(x) << endl;

    return 0;
}
```

## Part IV

### Function Overloading



## Signature of a Function

- Recall that in C++, all functions are **global**. That means, in general, all functions can “see” each other.
- Just as we use one’s signature to identify the person, we identify a function by its **name** and **signature**.
- A function’s **signature** is the list of **formal parameters** without their identifier names.
- **No** two C++ functions can have the **same name** and **same signature** but **different return type**.
- **BUT** two C++ functions can have the **same name** but **different signature**  $\Rightarrow$  **function overloading**.

### Example: No 2 Function Prototypes Differ Only in Return Type

```
// The following 2 function definitions of
// pick_one cannot appear in the same program

int pick_one(int x, float y) { return x; }
float pick_one(int x, float y) { return y; }
```

## Function Overloading

C++ allows **several functions** to have the **same name** but **different types** of input parameters.

### Example: Overloaded Functions

```
int max(int x, int y) { return (x > y) ? x : y; }
int max(int x, int y, int z) { return max(max(x,y), z); }
double max(double a, double b) { return (a > b) ? a : b; }

void swap(int& a, int& b) { int temp = a; a = b; b = temp; }
void swap(float& a, float& b) { float temp = a; a = b; b = temp; }
void swap(double& a, double& b) { double temp = a; a = b; b = temp; }

int absolute(int a) { return (a < 0) ? -a : a; }
int absolute(int& a) { return (a = (a < 0) ? -a : a); }
```

**Question:** How can you call the following version of `absolute( )`?

```
int absolute(int&);
```

## Example: Invalid Function Overloading

```
/* Identifier names of formal parameters are immaterial */
```

```
int max(int x, int y) { return (x > y) ? x : y; }
int max(int a, int b) { return (a > b) ? a : b; }
```

```
/* Return type is not part of the signature */
```

```
void swap(int& a, int& b) { int temp = a; a = b; b = temp; }
int swap(int& a, int& b) { int temp = a; a = b; b = temp; return a; }
```



## Overloaded Function Resolution

- When an **overloaded function** is called, C++ will determine exactly which function among those with the same name should be called — **function resolution**.
- Function resolution** is done by analyzing the type of actual parameters being passed during a function call.
- If there are **exact matches** between the types of formal parameters in an **overloaded function** and those of actual parameters, that function is picked.
- If there is no exact match, C++ uses a set of **pre-defined rules** for **function resolution**.
- The basic idea is to try to match their data types by:
  - first, **widening conversion** (coercion)
  - then, **narrowing conversion** (coercion)

## Example: Function Resolution

```
int test(int a, double b);
int test(double a, int b);
```

- If you make the following function call: `test(3, 4.6)`, the compiler will pick the **first version**.
- If you make the following function call: `test('a', 4.6)`, the compiler will again pick the **first version** by converting 'a' to an int.
- If you make the following function call: `test(3.2, 4.6)`, it can either
  - match to the first version by narrowing conversion of the first parameter to int.
  - match to the second version by narrowing conversion of the second parameter to int.
  - since neither one is more preferable than the other one  
⇒ **compilation error!**

## Default Function Argument

- Sometimes, we would like a function to have certain **default** behaviour, but still allow the user to **change** it.
- C++ allows the user to call a function with **fewer arguments** if all he wants is its **default behaviour**, and with **more arguments** if he wants some **particular behaviour** of the function.
- A function may have more than 1 **default argument**.
- But all **default arguments** must be specified at the **end** of the **formal parameter list**.

```
/* The following 2 prototypes are equivalent */  
void func(int x, float& y, char gender = 'M', bool alive = true);  
void func(int, float&, char = 'M', bool = true);
```

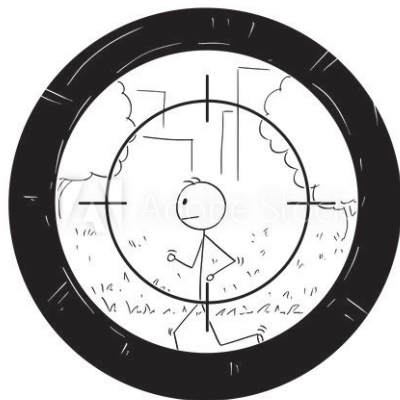
- The default argument(s) may be specified in a **function declaration** or **function definition**, but **not** both.
  - ▶ usually we put it on the **function declaration**. **Why?**
- A function with **default arguments** looks like several **overloaded functions**, but it is not.

## Example: Increment with Default Argument

```
#include <iostream>      /* File: increment-default-arg.cpp */  
using namespace std;  
  
int increment(int x, int step = 1)  
{  
    return (x + step);  
}  
  
int main()  
{  
    cout << increment(10) << endl;  
    cout << increment(10, 5) << endl;  
  
    return 0;  
}
```

## Part V

### Scope of Identifiers



#193567711

## Scope

### What is the Scope of an Identifier?

**Scope** is the **region of codes**  
in which an **identifier declaration** is **active**.

- **Scope** for an identifier is determined by the **location** of its **declaration**.
- In general, an identifier is **active** from the location of its declaration to the end of its **scope**.
- In C++, there is a big difference between identifiers declared **outside** or **inside** a **function**.
- Programmers commonly talk about the following 2 kinds of scope, though they are *not* official in C++'s standard:
  - ▶ **global scope**: when an identifier is declared **outside** any function.
  - ▶ **local scope**: when an identifier is declared **inside** a function.
- Technically, there are at least 3 kinds of scope: **file scope**, **function scope**, and **block scope**.

## Example: File/Function/Block Scope

```
#include <iostream>      /* File: scope.cpp */
using namespace std;

void my_print(const int b[], int size) // b and size are local variables with a FUNCTION SCOPE
{
    for (int j = 0; j < size; j++) // j is a local variable with a BLOCK SCOPE
    {
        int k = 10;           // k is a local variable with a BLOCK SCOPE
        cout << "array[" << j << "] = " << b[j] << '\t' << k*b[j] << endl;
    }
    cout << endl;
}

int a[] = {1,2,3,4,5}; // a is a global variable with a FILE SCOPE

void bad_swap(int& x, int& y) // x, y are local variables with a FUNCTION SCOPE
{
    int temp = x;           // temp is a local variable with a FUNCTION SCOPE
    x = y;
    y = temp;

    a[3] = 100;
}

int main()
{
    // num_array_elements is a local variable with a FUNCTION SCOPE
    int num_array_elements = sizeof(a)/sizeof(int);

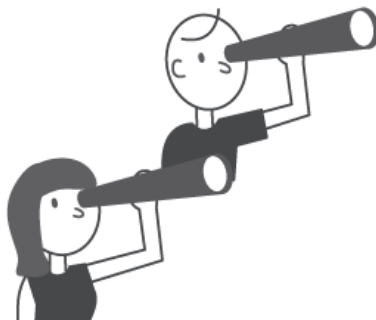
    bad_swap(a[1], a[2]); my_print(a, num_array_elements);
    bad_swap(a[3], a[4]); my_print(a, num_array_elements);
    return 0;
}
```

## File Scope

- **File scope** is the technical term for **global scope**.
- Variables with file scope are **global variables** and can be accessed by **any** functions in the **same** file or **other** files with proper **external declarations**. (More about this later.)
- Unlike local variables, **global variables** are initialized to **0** when they are defined without an **explicit initializer**.
- All function identifiers have **file scope**; thus, *all functions* are **global** in C++.
- Undisciplined use of global variables may lead to **confusion** and makes a program **hard to debug**.
  - ⇒ **try to avoid using global variables!**
  - ⇒ **use only local variables**, and pass them between functions.

## Function Scope

- **Function scope** is one kind of **local scope**.
- All variables/constants declared in the **formal parameter list**, or inside the **function body** have **function scope**.
- They are also called **local variables/constants** because they can only be accessed **within** the function — and not by any other functions.
- They are **short-lived**. They come and go: they are **created** when the function is called, and are **deconstructed** when the function returns.



## Block Scope

- **Block scope** is also a kind of **local scope**.
- A **block** of codes is created when you enclose codes within a pair of braces **{ }**. For example,
  - ▶ codes inside the body of **for**, **while**, **do-while**, **if**, **else**, **switch**, etc.
- Variables/constants with **block scope** are also **local** because they can only be used **within** the block.
- Similarly to the function scope, variables or constants having **block scope** are **short-lived**: they are **created** when the block is entered, and are **deconstructed** when the block is finished.

(There are also namespace scope and class scope but we won't talk about them now.)

## Example: Problems with a Global Variable

```
#include <iostream>    /* File: global-var-confusion.cpp */
using namespace std;

int number; // Definition of the global variable, number, with FILE scope. It is initialized to 0.

void increment_pbv(int x)
{
    x++;           // x is a local variable with a FUNCTION scope
    cout << "x = " << x << endl;

    number++; // global variable, number, used in the function, void increment_pbv(int)
}

void increment_pbr(int& y)
{
    y++;           // y is a local reference variable with a FUNCTION scope
    cout << "y = " << y << endl;

    number++; // global variable, number, used in the function, void increment_pbr(int&)
}

int main()
{
    increment_pbv(number); // global variable, number, used in the function, int main()
    cout << "number = " << number << endl;

    increment_pbr(number); // global variable, number, used in the function, int main()
    cout << "number = " << number << endl;
    return 0;
}
```

## Identifiers of the Same Name

The notion of **scope** has the following implications:

- An identifier can only be **declared once** in the **same scope**.
- Only the **name** matters: you cannot declare 2 variables/constants of the **same** name in the **same** scope even if they have **different** types.

```
int x = 1;
char x = 'b'; // error!
```

- However, the **same identifier name** may be “re-used” for variables or constants in **different scopes**.
- The different scopes may **not overlap** with each other, or, one scope may be **inside** another scope.

### Compiler Scope Rule

When an identifier is declared more than once but under different **scopes**, the compiler associates an **occurrence** of the identifier with its declaration in the **innermost enclosing scope**.

## Example: Scope Resolution

```
int main()
{
    int j;           // Apply to S1,S5,S6
    int k;           // Apply to S1,S2,S3,S4,S6
    S1;

    for (...)
    {
        int j;       // Apply to S2,S4
        S2;
        while (...)
        {
            int j;    // Apply to S3
            S3;
        }
        S4;
    }

    while (...)
    {
        int k;       // Apply to S5
        S5;
    }
    S6;
}
```

## Quiz: Which j applies to S7?

```
int main()
{
    int j;           // Apply to S1,S5,S6
    int k;           // Apply to S1,S2,S3,S4,S6
    S1;

    for (...)
    {
        int j;       // Apply to S2,S4
        S2;
        while (...)
        {
            S7;       // <--- Which j?
            int j;    // Apply to S3
            S3;
        }
        S4;
    }
    while (...)
    {
        int k;       // Apply to S5
        S5;
    }
    S6;
}
```

That's all!  
Any questions?

