

Programming with C++

COMP2011: Scope

Cecia Chan

Cindy Li

Brian Mak

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



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** 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 **destroyed** when the function returns.

- **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 **destructured** when the block is finished.

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

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!
```


Identifiers of the Same Name ..

- 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;
}
```