

Naming (II)

Scope vs Lifetime (cont.)

- **Local scope**: accessible within function or statement block from point of declaration until the end of the block
- **Class scope**: accessible by class members
- **File scope**: accessible within the current file (a global variable in a C file has file scope)
- **Global scope**: accessible everywhere unless “hidden”
- REF: <https://www.csee.umbc.edu/~chang/cs202.f15/Lectures/modules/m05-scope/slides.php?print>
- **Lifetime** of an identifier is the **time interval** during which the **identifier has been allocated a block of memory**.
 - Once the lifetime of a variable ends, the value the variable held is lost. Even if the identifier is recreated, it may be bound to a different memory location.
- Today, most PLs **link lifetime and scope**. Variables do not generally retain their value once they go out of scope because their identifiers lose the binding to a specific memory address.
- Ways to modify the lifetime/scope of variables
 - **static**: a static variable's lifetime is the **entire duration of the program execution**.
 - **extern**: **propagate global variables to other compilable units and make the global variable has global scope**. Global variables can be hidden from other compilable units by making them as static variables. (Used in C header file)
- Some C code below to demonstrate the scope

```
/**
 * staticInC.c
 * Ying Li
 */

#include <stdio.h>

void foo () {
    int i = 10;
    static int si = 10;
    const int ci = 10;

    i += 10;
    si += 10;

    printf("i = %3d, si = %3d, ci = %3d \n", i, si, ci);
}

int main () {
    for (int i = 0; i < 10; i++)
        foo();

    return 0;
}
```

Q1: will the value of i and si change in each iteration? [si change, i not]

Q2: can the program change ci's value? [No]

The lifetime of static variable is the entire duration of the program execution.

The lifetime of const is function block. A new const int is declared every time the foo function is invoked.

```
/**
 * forLoopScopeInC.c
 * Ying Li
 */

#include <stdio.h>

int gbl = 10; // program lifetime and file scope

int main () {
    int i = 1; // function scope

    printf("function i %d\n", i);

    if (1) { // 0 is false, non-zero is true
        int i = 2;
        printf("block i %d\n", i);

        // for loop in C has an implicit scope for loop variable declaration
        for(int i = 0; i < 5; i++, printf("for loop variable i %d\n", i)) {
            int i = 10;
            printf("for loop i %d\n", i);
        }

    }

    printf("function i %d\n", i);

    return 0;
}
```

Q1. What is the scope of the variable gbl and int i = 1? [gbl file scope, int i = 1 local/function scope]

Q2. Can the program be compiled successfully? [Yes]

Q3. What is the order of the two printf statements in the first for loop? Will the “for loop i” or the “for loop variable i” be printed first in each iteration? [“for loop i” is printed first]

For loop in C has an implicit scope for the loop variable declaration. So, C allows nested for loops share the name loop variable name.

Header Files of C

- Just like Java and Python have their own libraries, head files are a way for C to realize its libraries.
- A header file contains C **declarations and macro definitions** to be **shared between several source files**.
- Header files server **two purposes**
 - **System header files** declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations you need to invoke system calls and library. `#include <stdlib.h>`
 - **Your own header files** contain declarations for interfaces between the source files of your program. Each time you have a **group of related declarations and macro** definitions all or most of which are **needed in several different source files**, it is a good idea to create a header file for them. `#include "cstk.h"`
- Header files enable **more efficient** code
 - Including a header file produces the same results as copying the header file into each source file that needs it. Such copying would be time-consuming and error-prone.
 - With a header file, the **related declaration appear in only one place**. Easy to update.
 - The directive works by directing the C preprocessor to scan the specified file as an input before continuing with the rest of the current file.
- Once-Only Header
 - If a header file happens to be included twice, the compiler will process its contents twice. This is very likely to cause an error. Even if it does not, it will certainly waste time.
 - The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this

```
/* File foo.*/  
#ifndef FILE_F00  
#define FILE_F00  
  
the entire file  
  
#endif /* end of file foo*/
```

- This construct is commonly known as a **wrapper #ifndef**. When the header is included again, the conditional will be false. The preprocessor will skip over the entire contents of the file, and compiler will not see it twice.
- `FILE_F00` is called the *controlling macro* or *guard macro*.
- REF: <https://gcc.gnu.org/onlinedocs/cpp/Header-Files.html>

```
//
//  playExternCallee.c
//
//

#include "playExtern.h"

int gblA = 3; // global lifetime, file scope
int *gblPtr = &gblA;

void printExternVal () {
    printf("Callee gblA Address: %p \n", gblPtr);
    printf("Callee gblA %d\n", gblA);
}


//
//  playExtern.h
//
//

#ifndef playExtern_h
#define playExtern_h

#include <stdio.h>

extern int gblA; // global scope
//int gblA; // not an appropriate way
void printExternVal ();

#endif /* playExtern_h */


//
//  playExternCaller.c
//
//

#include "playExtern.h"

//int gblA; // not an appropriate way

int main () {
    printExternVal();
    gblA = 10;
    printf("Caller gblA Address %p\n", &gblA);
    printf("Caller gblA %d\n", gblA);

    return 0;
}
```