Introduction to Object-Oriented Programming

COMP2011: Scope and Separate Compilation

Dr. Cindy Li
Dr. Brian Mak
Dr. Dit-Yan Yeung

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

Part I

Scope of Identifiers

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>                                                    /* 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(void)
{
    // 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.
  - $\Rightarrow$ try to avoid using global variables!
  - $\Rightarrow$ 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 destructed 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 destructed 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++; cout << "x = " << x << endl;                        // x is a local variable with a function scope

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

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

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

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

    increment_pbr(number);                          // global variable, number, used in the function, int main(void)
    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 declared 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(void)
{
    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;
}
```

Part II

Separate Compilation

Motivation Example: Indirect Recursion

```
#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(void)
{
    int x;
    cin >> x;                                        // assume x > 0
    cout << boolalpha << odd(x) << endl;
    cout << boolalpha << even(x) << endl;
    return 0;
}
```

Divided We Win

- The odd-even example consists of 3 functions:
  - bool odd(int);
  - bool even(int);
  - int main(void);
- Now instead of putting them all in one .cpp file, we would like to put each function in a separate .cpp file of its own.
- There are good reasons for doing that:
  - We can then easily reuse a function in another program.
  - In a big project, programmers work in a team. After the program framework is designed in terms of a set of function prototypes, each programmer writes only some functions.
  - If a function needs to be changed, only one file needs to be modified.
- But how to compile the separate files into one single executable program?

Solution #1: Separate Compilation

- In order that each file can be separately compiled on its own, each file must know the existence of every variable, constant, function that it uses.
- All global constants, variables, functions that are *used* in a file "A" but are defined in another file "B" must be declared in file "A" before they are used in the file.
  - global constants: repeat their definitions
  - external variables: add the keyword extern
  - external functions: add their function prototypes. The keyword extern is optional since all C++ functions are global anyway.
- The keyword extern in front of a variable/function means that the variable/function is global and is defined in another file.
- Usually put all external declarations at the top of a file. Why?

Solution #1: Separate Compilation — main( )

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

/* constant definitions */
const int MAX_CALLS = 100;

/* global variable definition */
int num_calls;

/* function declarations */
extern bool odd(int);                    // "extern" is optional for functions

int main(void)
{
    int x;
    while (cin >> x)                                   // assume x > 0
    {
        num_calls = 0;
        cout << boolalpha << odd(x) << endl;
    }

    return 0;
}
```

Solution #1: Separate Compilation — even( )

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

/* constant definitions */
const int MAX_CALLS = 100;

/* global variable declarations */
extern int num_calls;                  // "extern" is a must for global variables

/* external function declarations */
extern bool odd(int);                          // "extern" is optional for functions

bool even(int x)
{
    if (++num_calls > MAX_CALLS)
    {
        cout << "max #calls exceeded\n";
        exit(-1);
    }

    return (x == 0) ? true : odd(x-1);
}
```

Solution #1: Separate Compilation — odd( )

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

/* constants definitions */
const int MAX_CALLS = 100;

/* global variable declarations */
extern int num_calls;                  // "extern" is a must for global variables

/* function declarations */
extern bool even(int);                          // "extern" is optional for functions

bool odd(int x)
{
    if (++num_calls > MAX_CALLS)
    {
        cout << "max #calls exceeded\n";
        exit(-1);
    }

    return (x == 0) ? false : even(x-1);
}
```

Solution #1: Separate Compilation Procedure 1

- Compile all the source .cpp files with the following command:

```
g++ -o odd-even main.cpp even.cpp odd.cpp
```

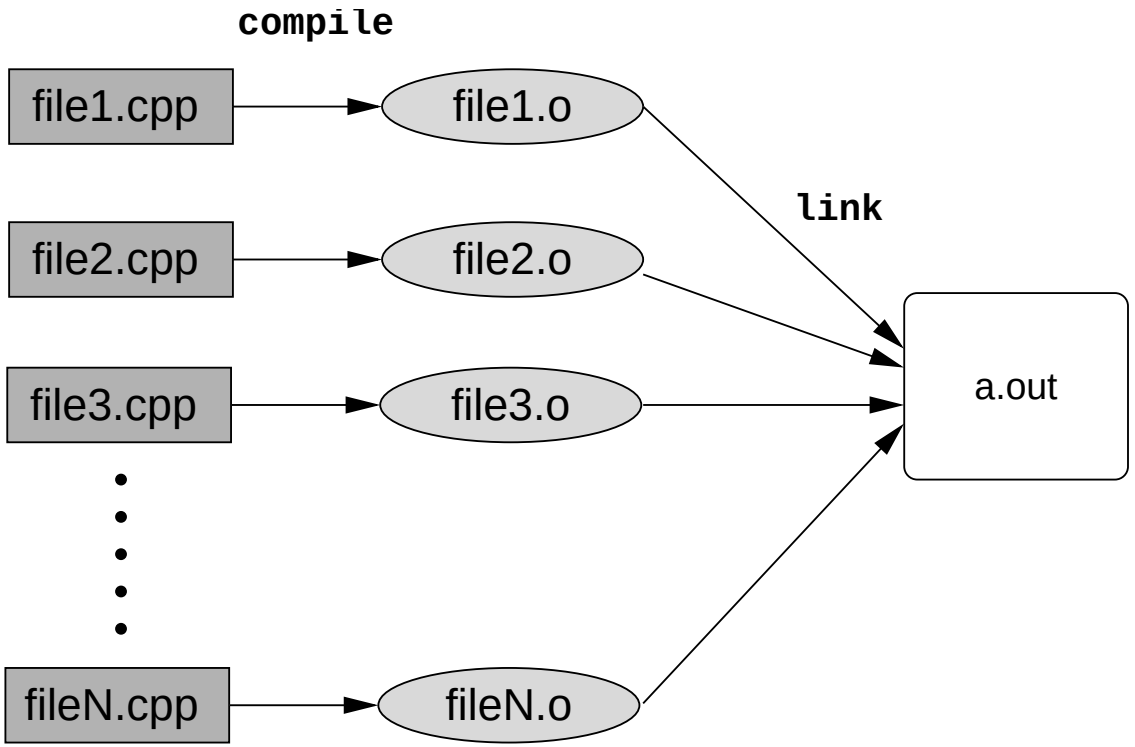
  But this will again compile all files even if you may only change one of the file.

- Better to compile them separately:

```
g++ -c main.cpp
g++ -c even.cpp
g++ -c odd.cpp
g++ -o odd-even main.o even.o odd.o
```

- The command `g++ -c a.cpp` will produce an object file "a.o" for the source file "a.cpp".
- Then the final line `g++ -o odd-even main.o even.o odd.o` invokes the linker to link or merge the separate object files into one single executable program "odd-even".

Solution #1: Separate Compilation Procedure 2



- Now, if you later modify only "main.cpp", then you just need to re-compile "main.cpp" and re-link all object .o files.

```
g++ -c main.cpp
g++ -o odd-even main.o even.o odd.o
```

- In general, just re-compile those source files that are modified, and re-link all object files of the project.

Part III

Definition vs. Declaration and Header Files

Variable and Function Definition

A definition introduces the name and type of an identifier such as a variable or a function.

- A variable definition requires the compiler to reserve an amount of memory for the variable as required by its type.
- A variable may also be initialized in its definition. For instance, `int x = 5;` .
- A function definition generates machine codes for the function as specified by its (function) body.
- In both cases, definition causes memory to be allocated to store the variable or function.
- A variable and function identifier must be defined exactly once in the whole program even if the program is written in separate files.

Variable and Function Declaration

The declaration of a variable or function announces that the variable or function exists and is defined somewhere — in the same file, or in a separate file.

- A variable's declaration consists of the its name and type preceded by the keyword extern. No initialization is allowed.
- A function's declaration consists of the its prototype, and may by optionally preceded by the keyword extern.
- A declaration does not generate codes for a function, and does not reserve memory for a variable.

Variable and Function Declaration ..

- There can be many declarations for a variable or function in the whole program.
- An identifier must be defined or declared before it can be used.
- During separate compilation, the compiler generates necessary information so that when the linker combines the separate object files, it can tell that the variable/function declared in a file is the same as the global variable/function defined in another file, and they should share the same memory or codes.

Header Files

- In Solution#1, you see that many global variable or function declarations are repeated in "odd.cpp" and "even.cpp". That is undesirable because:
  - We are lazy, and we do not want to repeat writing the same declarations in multiple files.
  - Should a declaration require updating, one has to go through all files that have the declaration and make the change.
  - More importantly, maintaining duplicate information in multiple files is error-prone.
- The solution is to use .h header files which contains
  - definitions of global variables and constants
  - declarations of global variables and functions
- Header files are inserted to a file by the preprocessor directive #include.

```
#include <iostream>          // standard library header files
#include "my_include.h"      // user-defined header files
```

Solution #2: Separate Compilation — Header Files

```
/* File: my_include.h */
/* include system library info files or user-defined header files */
#include <iostream>
#include <stdlib.h>
using namespace std;

/* constant definitions */
const int MAX_CALLS = 100;

/* external function declarations */
extern bool odd(int);          // "extern" is optional for functions
extern bool even(int);
```

```
/* File: global.h */
/* global variable definitions */
int num_calls;
```

```
/* File: extern.h */
/* external global variable declarations */
extern int num_calls;
```

Solution #2: Separate Compilation — main( )

```
#include "my_include.h"                    /* File: main.cpp */
#include "global.h"

int main(void)
{
    int x;
    while (cin >> x)                       // assume x > 0
    {
        num_calls = 0;
        cout << boolalpha << odd(x) << endl;
    }

    return 0;
}
```

Solution #2: Separate Compilation — even( )

```
#include "my_include.h"                    /* File: even.cpp */
#include "extern.h"

bool even(int x)
{
    if (++num_calls > MAX_CALLS)
    {
        cout << "max #calls exceeded\n";
        exit(-1);
    }

    return (x == 0) ? true : odd(x-1);
}
```

Solution #2: Separate Compilation — odd( )

```
#include "my_include.h"                    /* File: odd.cpp */
#include "extern.h"

bool odd(int x)
{
    if (++num_calls > MAX_CALLS)
    {
        cout << "max #calls exceeded\n";
        exit(-1);
    }

    return (x == 0) ? false : even(x-1);
}
```

Header Files of the Standard C++ Libraries

- iostream: input/output functions
- iomanip: input/output manipulation functions
- cctype: character functions
  e.g. int isdigit(char); int isspace(char); int isupper(char);
- cstring C string functions:
  e.g. int strlen(const char []);
  int strcmp(const char [], const char []);
- cmath: math functions
  e.g. double sqrt(double); double cos(double);
- cstdlib: commonly used functions
  e.g. int system(const char []); int atoi(const char []);
  void exit(int); int rand( ); void srand(unsigned int);