CS419 Compiler Construction

Lecture 2

Edited by Dr. Ismail Hababeh

Adapted from slides by Dr. Mohammad Daoud

German Jordanian University

Originally from slides by Dr. Robert A. Engelen

Programming Language Basics

- Static and dynamic decision policy
- Environments and states
- Static scope and block structure
- Parameter passing mechanisms

Static and Dynamic Decision

- **Static policy:** the language rules allow the compiler to decide during *compile time*

decide at compile time things like variable types, memory allocation, and whether certain operations are allowed. and if any errors were caught it is considered as compilation time error

- **Dynamic policy:** the language rules allow the compiler to take a decision when the program is executed at *run time.*

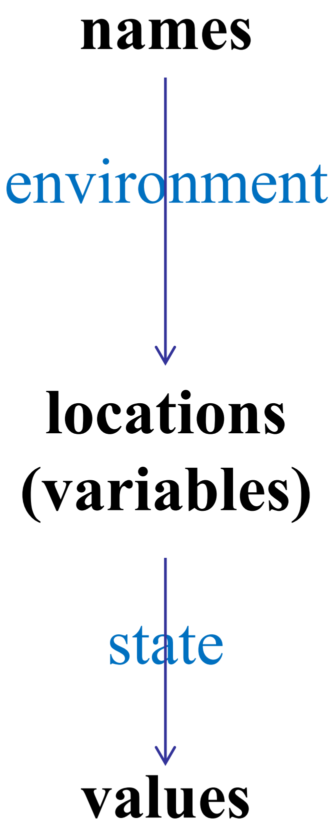
Environments and States

- **Environment:** mapping from names to locations in the memory.

Example: maps x to a memory location that changes x.

- **State:** mapping from location in the memory to their values
- Example: y = x + 1;

return the value stored in memory location.



names

environment

locations (variables)

state

values

Environments and states - Example

```
….
int i;                    // global i
 …
void f(…){
      int i; //local i
      …
      i = 3; //use of local i
      Inside we deal with the local i
      …
}
…
        outside we deal with the global i
      x = i +1;  //use of global i
```

Implicit declaration

Static Scope and Block Structure

Static scope: the scope of a declaration can be determined:

- *Implicitly:* by the location of the declaration inside the program.
- *Explicitly:* using language keywords such as public, private, protected (in C++, Java,…)

Explicit Declaration

- *Public*: accessible from outside the class
- *Protected*: scope is limited to the declaring class and subclasses (inheritance).
- *Private*: scope is limited to the declaring class and friend* classes.

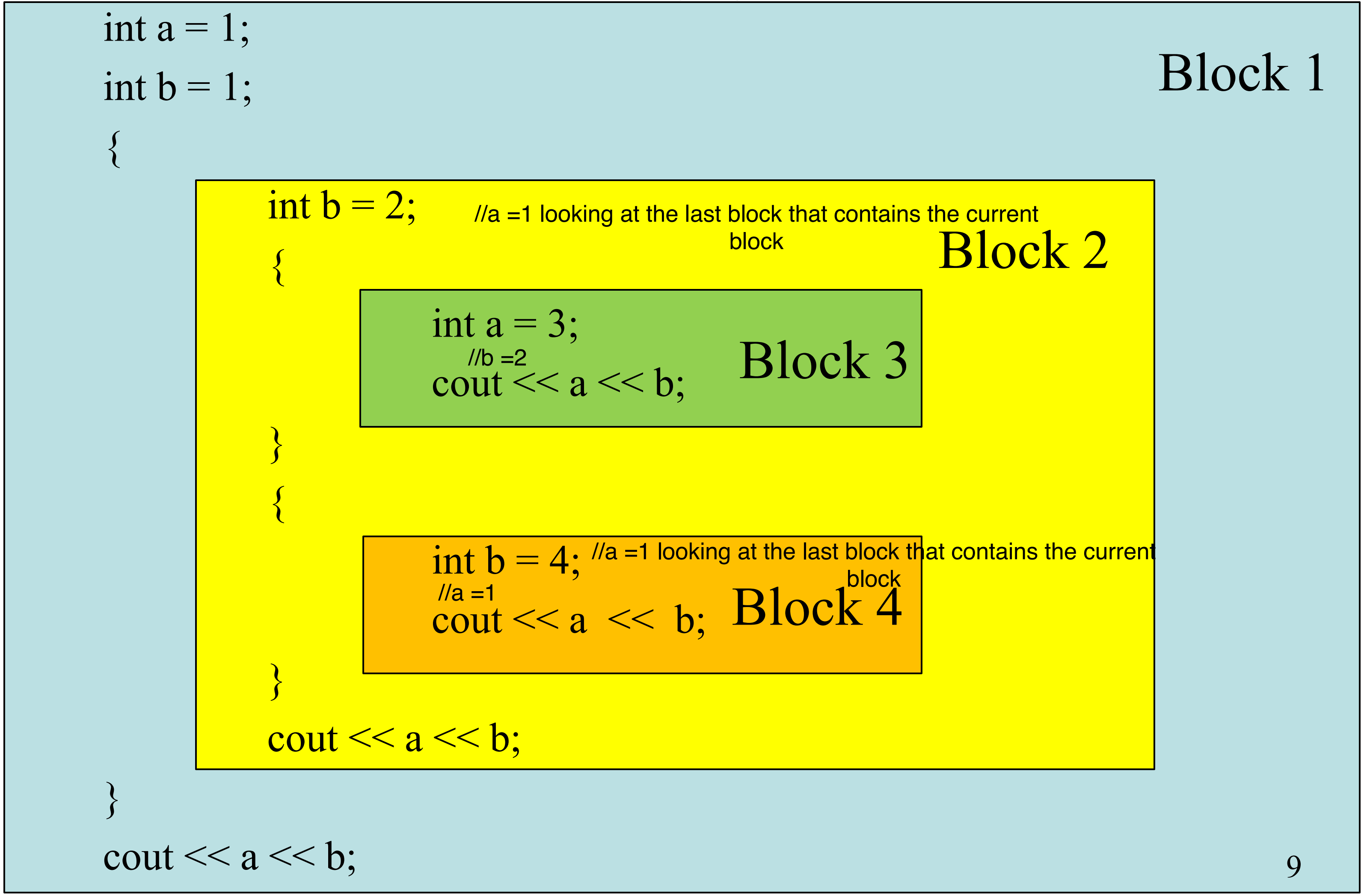
*A **friend class** in C++ can access the "private" and "protected" members of the class in which it is declared as a friend.

Static Scope and Block Structure

- **Block:** a sequence of declarations followed by a sequence of statements, all surrounded by braces { }
- Blocks can be nested inside each other.

Question: Given the following a and b variables’ declarations structure, determine the scope block(s) of each declaration.

If we were at a certain block and a variable was not declared inside that block then we look at the block containing our current block



```
main(){
    int a = 1;
    int b = 1;
    {
        int b = 2;     //a =1 looking at the last block that contains the current block
        {
            int a = 3;
            //b =2
            cout << a << b;
        }
        {
            int b = 4; //a =1 looking at the last block that contains the current block
            //a =1
            cout << a  <<  b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

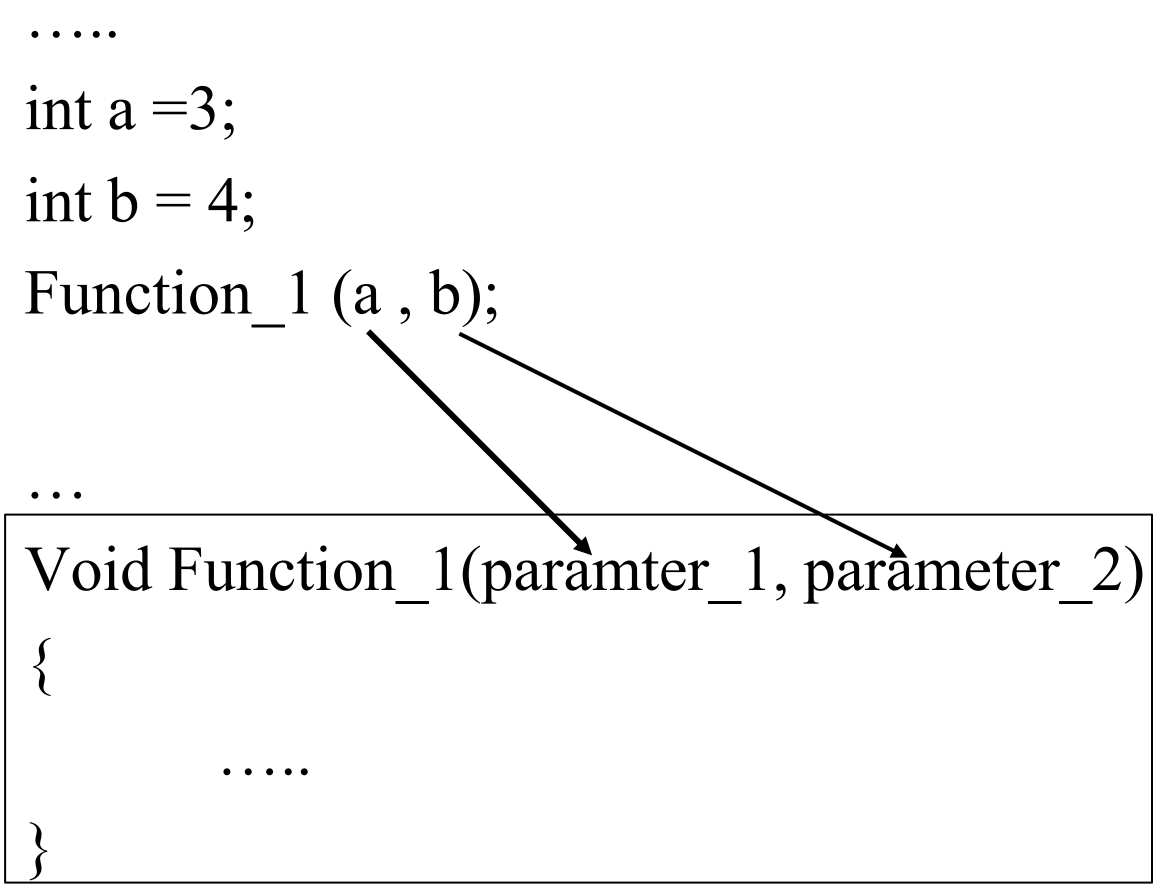
Question Solution

Declaration	Scope
int a = 1;	Block 1 – Block 3
int b = 1;	Block 1 – Block 2
int b = 2;	Block 2 – Block 4
int a = 3;	Block 3
int b = 4;	Block 4

Parameter Passing Mechanisms

- Call-by-value
  - The **value** of the *actual* parameter in the calling function is copied to the *formal* parameter in the called procedure.
  - All computations involving the formal parameters done by the called procedure is local to that procedure.

Call-by-value - Example



```
…..
int a =3;
int b = 4;
Function_1 (a , b);

…
Void Function_1(paramter_1, parameter_2)
{
            …..
}
```

Parameter Passing Mechanisms

- Call-by-reference
  - The *address of the actual parameter* in the calling function is passed to the called procedure as the *value of the formal parameter.*
  - Changes to the formal parameter appear in the called procedure as changes to the actual parameter in the calling function
  - Used to reduce the memory requirements of passing large arrays and objects by passing their addresses only.

Parameter Passing Mechanisms

- Call-by-pointer
  - Call by pointer do the same thing as call-by-reference. The only difference between them is the fact that a *pointer can be null*, or maybe *pointing to invalid places* in memory, while references are never be null.

Question: What will be the value of x after executing the following program?

```
// by value
void by_value(int a){
        a+=10;
}
// by pointer
void by_pointer(int *a){
        (*a)+=10;
}
 // by reference
void by_ref(int &a){
        a+=10;
}
```

```
int main(){
        int x=40;
        by_value(x);
        //x = ?
        by_pointer(&x);
        //x = ?
        by_ref(x);
        //x = ?
        return 0;
}
```

Question Solution

```
void by_value(int a){
        a+=10;
}

void by_pointer(int *a){
        (*a)+=10;
}

void by_ref(int &a){     //Can not be null
        a+=10;
}
```

Solution:

```
int main(){
        int x=40;
        by_value(x);
        //x=40
        by_pointer(&x);
        //x=50
        by_ref(x);
        //x=60
        return 0;
}
```