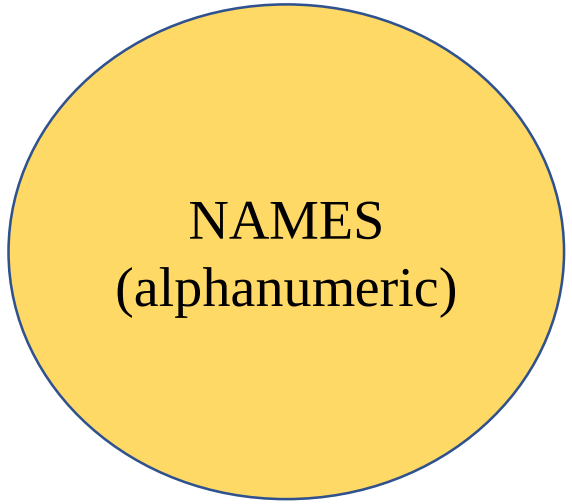
Course: PARADIGMS AND COMPUTER PROGRAMMING FUNDAMENTALS (PCPF)

Course Instructor

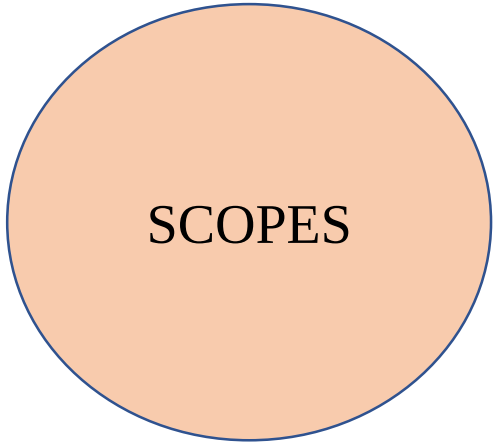
Mrinmoyee Mukherjee B.E (Electronics), M.E (EXTC), PhD (Pursuing)
Assistant Professor
Department of Information Technology
St. Francis Institute of Technology
email: mrinmoyeemukherjee@sfit.ac.in

Academic Year: 2023-24 (Odd Semester)

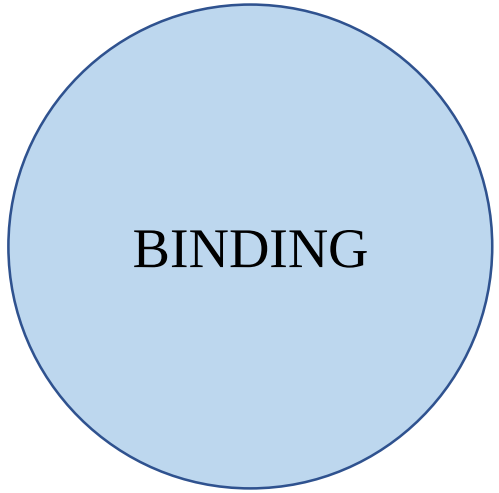
PROGRAMMING LANGUAGES OBEYING DIFFERENT PARADIGM



Names enable programmers to refer to *variables*, *constants*, *operations*, and *types* using identifier names rather than low-level hardware addresses



The scope of a binding is the part of the program (textually) in which the binding is active.



A binding is an association between a name and the thing that is named

Names in PL

- Design issues
- The following are the design issues for names:
  - Maximum **length**?
  - Are names **case sensitive**?
  - Are special words **reserved words** or **keywords**?

Names in PLs

- Length –
  - If too short, they cannot be connotative (capability to convey some meaning)
  - Language examples:
    - Fortran max 6
    - Cobol max 30
    - C# and Java: **no limit**, and all characters are significant
    - C++: **no limit**, but implementers often impose a length limitation because they do not want the **symbol table** in which identifiers are stored during compilation to be too large and also to simplify the **maintenance** of that table.
- Names in most programming languages have the same form:
  - a letter followed by a string consisting of letters, digits, and (_).
  - The use of the _ is less popular.
  - **C-based** languages (C, Objective-C, C++, Java, and C#), replaced the _ by the "camel" notation, as in myStack.

Names in PL

- **Special characters**
  - **PHP**: all variable names must begin with dollar signs **$**
  - **Perl**: all variable names begin with special characters $, @, or %, which specify the variable's type
    - if a name begins with **$ is a scalar**, if a name begins with **@ it is an array**, if it begins with **%, it is a hash structure**
- **Case sensitivity**
  - Disadvantage: readability (names that look alike are different)
  - Names in the C-based languages are case sensitive
  - Worse in C++, Java, and C# because **predefined** names are mixed case (e.g. IndexOutOfBoundsException)
  - In C, however, exclusive use of **lowercase** for **names**.
- C, C++, and Java names are case sensitive ➔ rose, Rose, ROSE are distinct names "What about Readability"

Names in PL

- Special words
- A **keyword** is a word that is special only in certain contexts. Ex: in Fortran
  - Ex: In Fortran, they are only keywords, which means they can be redefined. One could have the statements:

    Real Apple // Real is a data type (keyword) followed with a name

    Real = 3.4 // Real is a variable name

    Integer Real // keyword "Integer" and variable "Real"

    Real Integer // keyword "Real" and variable "Integer"

    - Disadvantage: poor readability, compilers and users must recognize the difference.
- A **reserved** word is a special word that **cannot be used** as a user-defined name.
  - Potential problem with reserved words: If there are **too many, many collisions** occur (e.g., COBOL has 300 reserved words!)
  - As a language design choice, reserved words are better than keywords.

Names in PL

- **Variables**
- A variable is an abstraction of a memory cell.
- Variables can be characterized by following six attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

BINDING

A binding is an association between two things, such as a name and the thing it names.

Association between pages to form chapters…book

Association between people to form years…life

int num=5;

Association between memory (name) and constant (object)

int num=5;

Association between memory (name) and constant (object)

In a program many such bindings will happen between objects and name (memory)

BINDING TIME

- How much time is required for binding?
- What are different types of binding taking place?
- Which is the starting point of binding ??

The set of all bindings at a given point in a program is called as the referencing environment

Assume you are developing a language …..

Binding Time is the point at which a binding is created or, more generally, the point at which any implementation decision is made.

- Language Design Time
- Language Implementation Time
- Program Writing Time
- Compile Time
- Link and Load Time

What are the things that you need to bind while designing the language-

- Control flow constructs (if, if-else, if-else ladder, while for etc)
- Primitive Types (int, float, char, string, double, struct)
- Constructors
- Pointers
- Syntax
- Keywords
- Reserved words
- Meaning of operators (‘+’=add)

Language Design Time

Language Implementation Time

Program Writing Time

Compile Time

Link and Load Time

- Describes the accuracy level primitive types (no of bits for int, float)
- Storage allocation method for variable
- Coupling of I/O to the operating system's notion of files
- Maximum sizes of stack
- Handling of run time errors

(*At the end of this phase, the language has been designed….. Now its time for the programmers to use the language*)

Language Design Time

Language Implementation Time

Program Writing Time

- Programmers choose algorithms, data structures and name

Compile Time

- Mapping of high level constructs to machine code
- Most compilers support separate compilation (*each module is compiled separately)*

Link and Load Time

- Link Time: Compiling different modules of program at different times
- Load Time: Time at which operating system loads program to memory

The last is the run time- the entire span from start to end

Scope

- The **scope** of a variable is the **range of statements over which its declaration is visible**
  - A variable is visible in a statement if it can be **referenced** in that statement
  - **Local** variable is local in a program unit or block if it is declared there.
  - Non-local or **global** variable of a program unit or block are those that are visible within the program unit or block but are not declared there

1.Could a function use a variable declared in the main program?
2.Could a main program use a variable declared in one of its function?

Scope Rules

- The scope rules of a language determine how a particular occurrence of a name is associated with a variable
  1. The **scope** of an entity is the program or function in which it is declared.
  2. A **global entity is visible to all** contained functions, including the function in which that entity is declared.
  3. An entity declared in the scope of another entity is always a different entity even **if their names are identical**.
- Scope rules determine
  - how references to variables declared outside the currently executing subprogram or how blocks are associated with their declarations
- Two types of scope
  - **Static**/lexical scope
  - **Dynamic** scope

Frequently we may have a local entity whose name is identical to the name of a global entity. To resolve this name conflict, we need the following new scope rule:

```
PROGRAM  Scope_3
  IMPLICIT  NONE
  INTEGER :: i, Max = 5

  DO i = 1, Max
    Write(*,*)  Sum(i)
  END DO

CONTAINS

  INTEGER FUNCTION  Sum(n)
    IMPLICIT  NONE
    INTEGER, INTENT(IN) :: n
    INTEGER          :: i, s
    s = 0
    DO i = 1, n
      s = s + i
    END DO
    Sum = s
  END FUNCTION  Sum
```

1. The main program declares a variable **i**, which is global to function **Sum()**.
2. However, **i** is also declared in function **Sum()**.
3. These two **i**s are two different entities.
4. More precisely, when the value of **Sum()**'s **i** is changed, this change will not affect the **i** in the main program and vice versa.

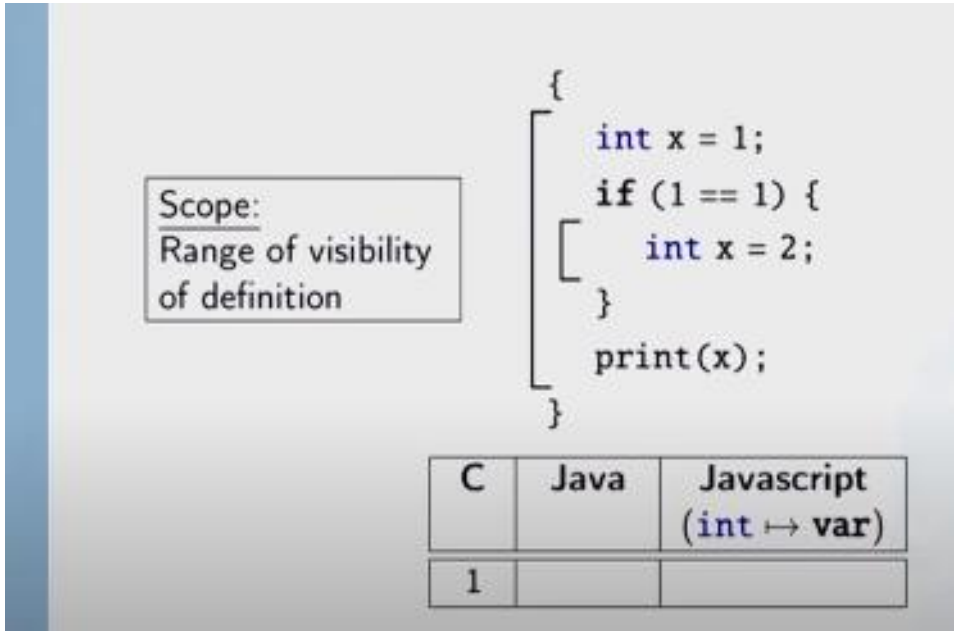
Static Scope

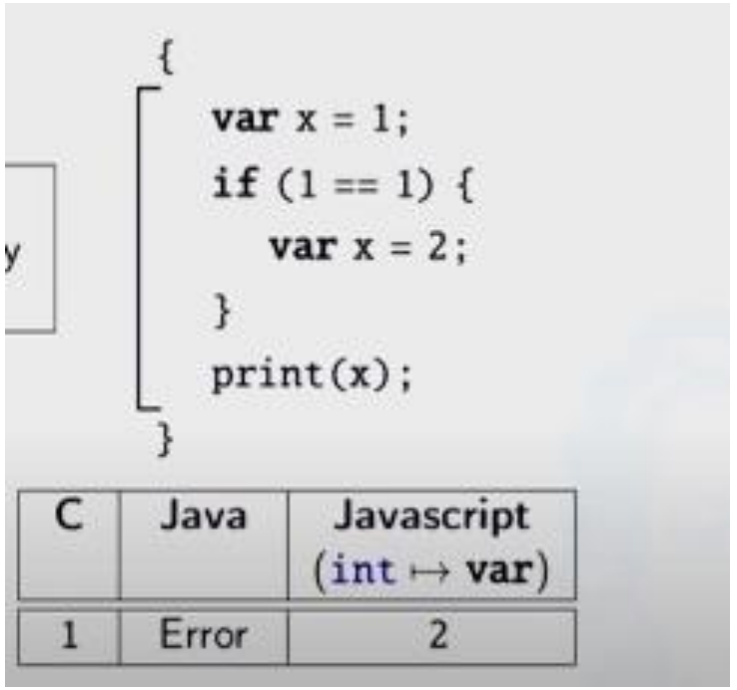
- The scope of a variable can be statically determined, that is, prior to execution. Here a variable always **refers to its top-level environment.**
- Two categories of static-scoped languages
  - Languages **allowing nested subprograms**:
    - Ada, JavaScript, and PHP
  - Languages **which does not allow**
    - subprograms: C-based languages
- To connect a name reference to a variable, you must find the appropriate declaration
- Search process
  1. Search the declaration **locally**
  2. If not found, search the **next-larger enclosing unit**
  3. **Loop over step 2 until a declaration is found** or an undeclared variable error is detected

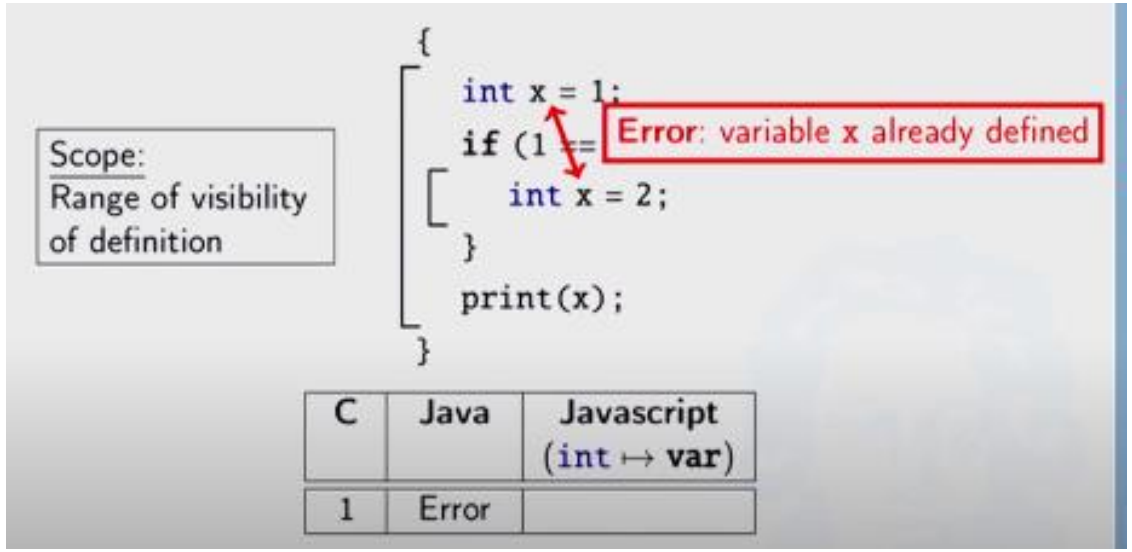
Dynamic Scope

- Dynamic scoping is based on the calling sequence of subprograms, not on their spatial relationship to each other
- Dynamic scope can be determined only at runtime
- Always used in interpreted languages, which usually does not have type checking at compile time
- In dynamic scoping, **a global identifier refers to the identifier associated with the most recent environment**

SCOPES







Different languages use different scoping rules

Scope in C

There are three places where variables can be declared in C programming language −

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

```
int x=10;          // Global x
voi main()
{
int x=20;         // X Local to Block 1
- - - -
- - - -
  {
  int x=30; // X Local to Block 2
  - - - -
  - - - -
  - - - -
  }
}
```

(1) (2)

```
void funct()
{
int x=40;       // X Local to Block 3
- - - - -
- - - - -
- - - - -
}
```

(3)

Scope in C: Local Variables

- Variables that are declared inside a function or block are called local variables.
- They can be used only by statements that are inside that function or block of code.
- Local variables are not known to function outside their own.

```c
int main () {

  /* local variable declaration */
  int a, b;
  int c;

  /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;

  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

  return 0;
}
```

Value of a = 10, b = 20 and c = 30

Online tools used: onlinegdb-online compiler for c/C++, Online C compiler -Jdoodle

Scope in C: Global Variables

- Global variables are defined outside a function, usually on top of the program.
- Global variables hold their values throughout the lifetime of your program
- They can be accessed inside any of the functions defined for the program.

```c
#include <stdio.h>

/* global variable declaration */
int g=10;

int main () {

  /* local variable declaration */
   int a, b;
  printf("Value of g is:=%d\n", g);
   /* actual initialization */
   a = 10;
   b = 20;
   g = a + b;

   printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

   return 0;
}
```

```
g is:=10
value of a = 10,
 b = 20 and
g = 30
```

Scope in C: Formal Parameters

- Formal parameters, are treated as local variables within a function and they take precedence over global variables.

```c
#include <stdio.h>

/* global variable declaration */
int a = 20;

int main () {

  /* local variable declaration in main function */
  //int a = 10;
  int b = 20;
  int c = 0;

  printf ("value of a in main() = %d\n",  a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n",  c);

  return 0;
}

/* function to add two integers */
int sum(int a, int b) {

   printf ("value of a in sum() = %d\n",  a);
   printf ("value of b in sum() = %d\n",  b);

   return a + b;
}
```

a=20, c=40

Execute | Share | main.c | STDIN

```c
#include<stdio.h>

int main()
{
{
    int x = 10, y = 20;
    {
        printf("x = %d, y = %d\n", x, y);
        {
            int y = 40;
            x++;
            y++;
            printf("x = %d, y = %d\n", x, y);
        }
        printf("x = %d, y = %d\n", x, y);
    }
}
return 0;
}

```

x = 10, y = 20
x = 11, y = 41
x = 11, y = 20

Scope in JAVA: Block Level

- The variables that are defined in a block are only accessible from within the block.
- The scope of the variable is the block in which it is defined

```
public class MainClass {
  public static void main(String[] args) {
    for (int x = 0; x < 5; x++) {
        System.out.println(x);
    }
    //x is not accessible here
    //System.out.println(x);
  }
}
```

Here, the Scope of Variable x is Block Level

```
public class Test{
  public void age() {
    int age ;
    age = age + 7;
    System.out.println("Age is : " + age);
}
public static void main(String[] args) {
   Test test = new Test();
   test.age();
 }
}
```

Same Program as Previous but in this Program we use Local Variable *age* Without Initializing it, so it would Thrown Compile time Error

```
class ScopeInvalid {
  public static void main(String args[]) {
    int num = 1;
    {          // creates a new scope
      int num = 2;    // Compile-time error
      // num already defined
    }
  }
}
```

Here Compile Error Because Variable "num" is Declared in main Scope and thus it is Accessible to all the Innermost Blocks.

```
class ScopeValid {
  public static void main(String args[]) {
    {          // creates a new scope
      int num = 1;
    }
    {          // creates a new scope
      int num = 2;
    }
  }
}
```

```
// Demonstrate block scope
class Scope {
  public static void main(String args[]){
     int n1=10;   // Visible in main
     if(n1 == 10)
     {
      // start new scope
      int n2 = 20; // visible only to this block
      // num1 and num2 both visible here.
      System.out.println("n1 and n2 : "+ n1 +" "+ n2);
      }
      // n2 = 100; // Error! n2 not known here
      // n1 is still visible here.
    System.out.println("n1 is " + n1);
  }
}
```

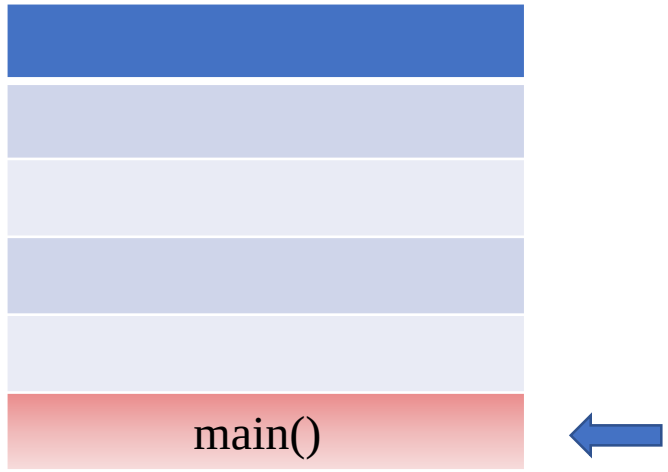
Output is:

```
n1 and n2 : 10 20
n1 is 10
```

- Stack is a container or a memory segment which holds some data
- Data is retrieved in Last in First Out fashion
- Two operations: push and pop

```
main( )
{

}
```



Stack (or Call Stack)

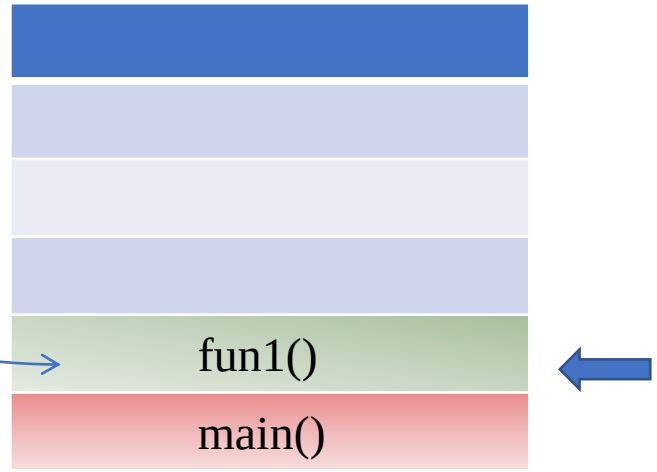
Capable of storing functions

- Stack is a container or a memory segment which holds some data
- Data is retrieved in Last in First Out fashion
- Two operations: push and pop

```
main( )
{
  fun1( );
}
```

Control will get transferred from main to fun1()

```
fun1( ) {                    }
```



Stack (or Call Stack)

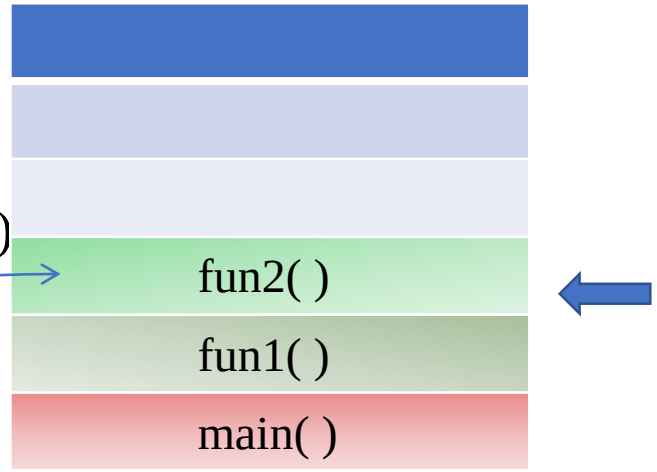
Capable of storing functions

- Stack is a container or a memory segment which holds some data
- Data is retrieved in Last in First Out fashion
- Two operations: push and pop

```
main( )
{
  fun1( );
}
```

Control will get transferred from fun1( ) to fun2( )

```
fun1( ) {    fun2( );      }
```



Stack (or Call Stack)

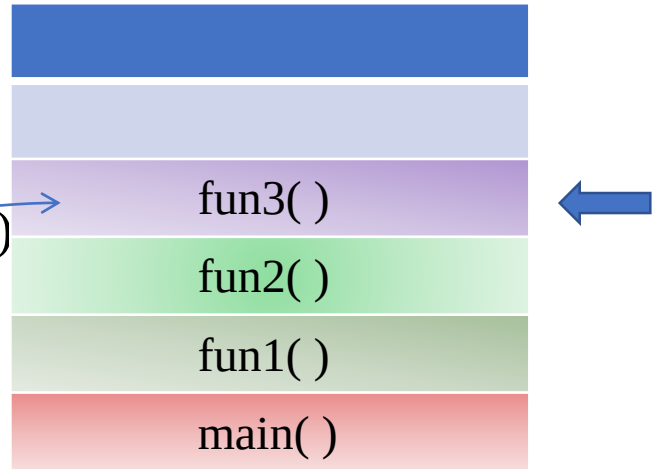
Capable of storing functions

- Stack is a container or a memory segment which holds some data
- Data is retrieved in Last in First Out fashion
- Two operations: push and pop

```
main( )
{
  fun1( );
}
```

Control will get transferred from fun1( ) to fun2( )

```
fun1( ) {   fun2( );   }
fun2( ) {   fun3( );   }
```



Stack (or Call Stack)

Capable of storing functions

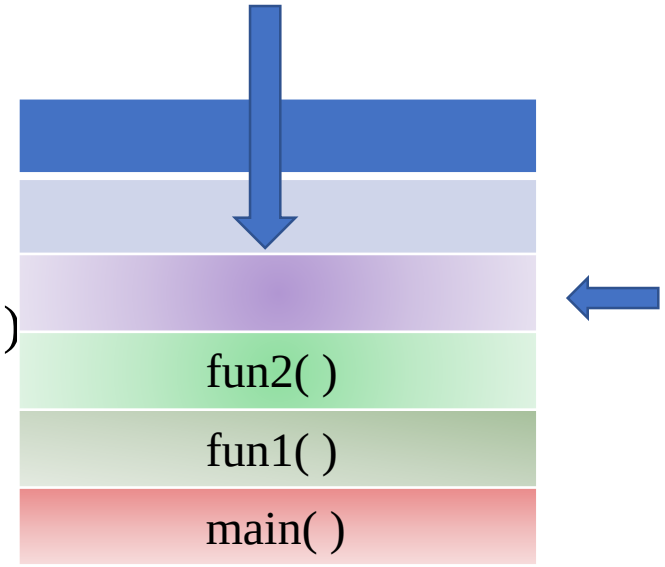
- Stack is a container or a memory segment which holds some data
- Data is retrieved in Last in First Out fashion
- Two operations: push and pop

```
main( )
{
  fun1( );
}
```

Control will get transferred from fun1( ) to fun2( )

```
fun1( ) {   fun2( );    }
fun2( ) {   fun3( );    }
fun3( ) {   return;     }
```

This will pop out fun3 () and will return to previous function

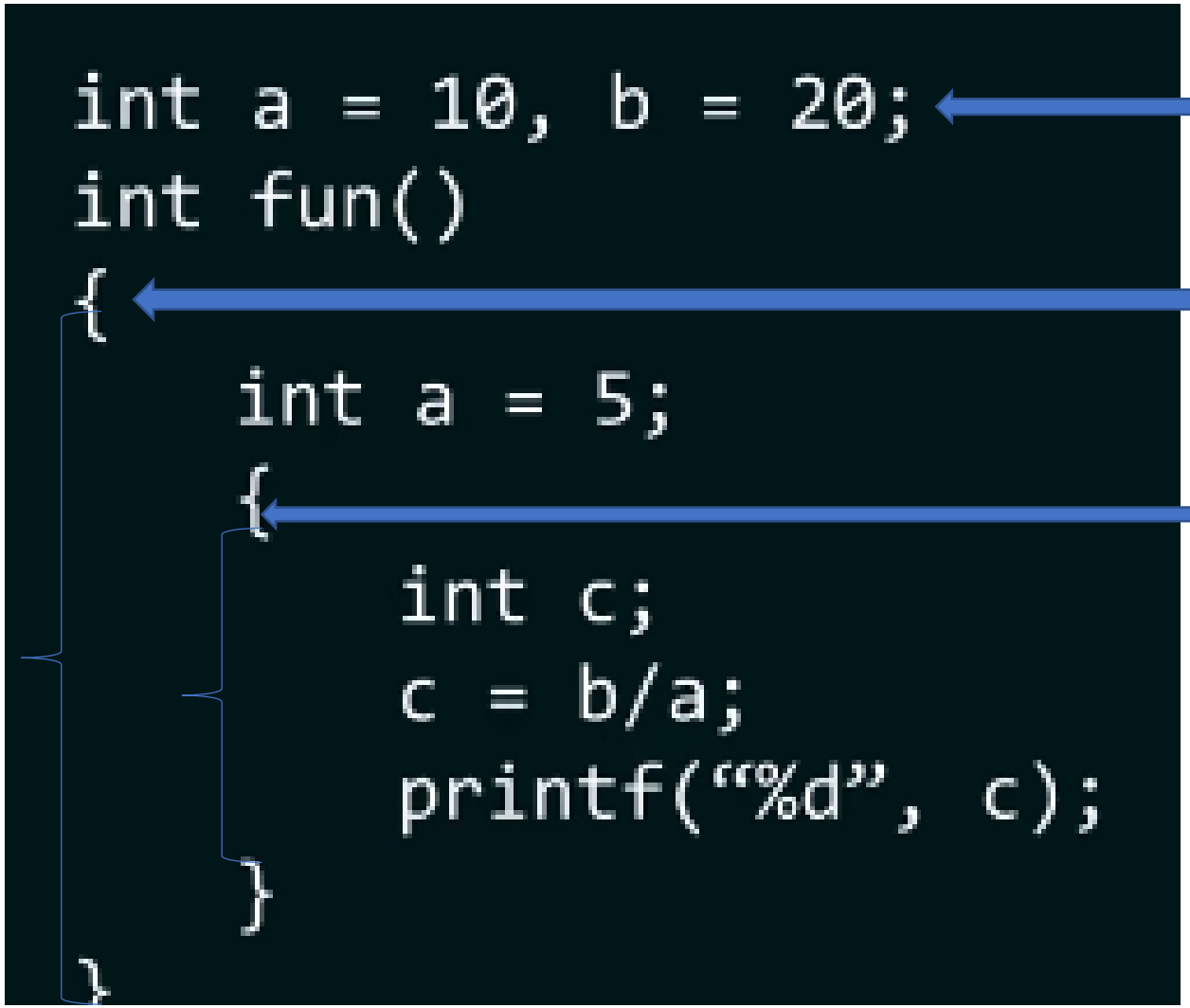


Stack (or Call Stack)

Capable of storing functions

STATIC SCOPING

In static scoping (or lexical scoping) , definition of a variable is resolved by searching its containing block or function. If that fails, then searching the outer containing block and so on. Scoping allows us to reuse the variable name



Third search for a and b

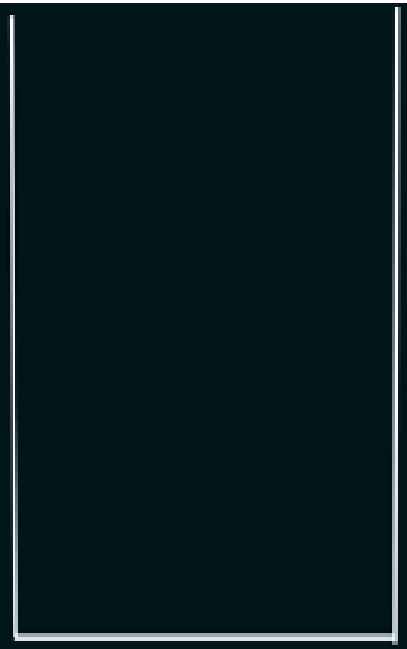
Second search for a and b

First search for a and b

Output:
c=4

```
int fun1(int);
int fun2(int);
int a = 5;
int main()
{
    int a = 10;
    a = fun1(a);
    printf("%d", a);
}
```

Note variable a is initializes global variable



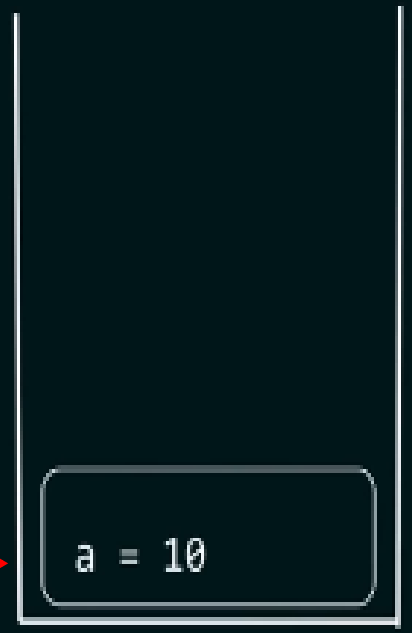
Global Variable will find place in initialised data segment

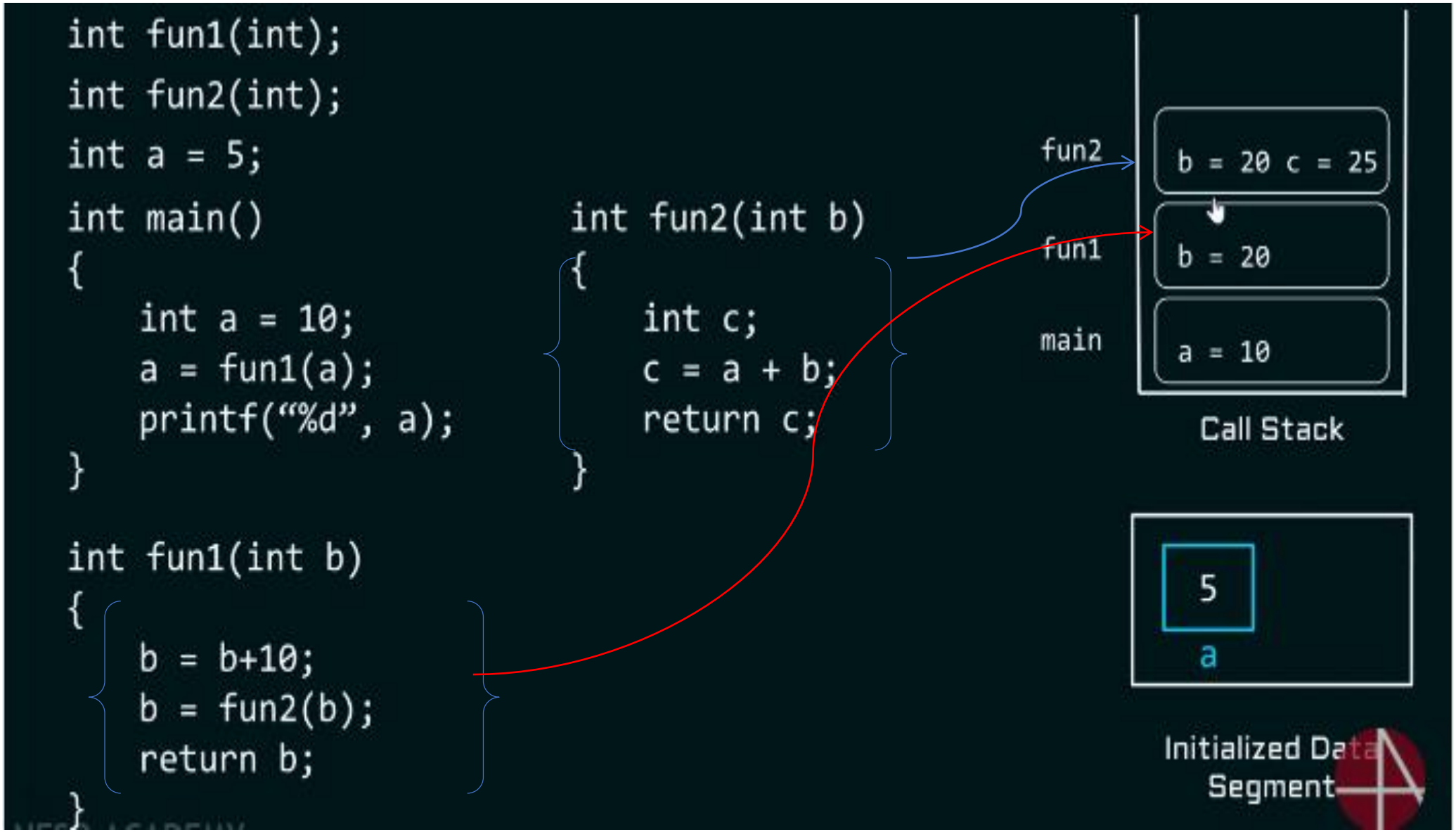
```
int fun1(int);
int fun2(int);
int a = 5;
int main()
{
    int a = 10;
    a = fun1(a);
    printf("%d", a);
}
```

Execution starts from main

main

Activation record of main is created





DYNAMIC SCOPING

In dynamic scoping the definition of variable is resolved by **searching its containing block** and if **not found, then searching its calling function** and **if still not found then the function which called that calling function** will be searched and so on…

```
int fun1(int);
int fun2(int);
int a = 5;
int main()
{
    int a = 10;
    a = fun1(a);
    printf("%d", a);
}

int fun1(int b)
{
    b = b+10;
    b = fun2(b);
    return b;
}
```

```
int fun2(int b)
{
    int c;
    c = a + b;
    return c;
}
```