UNIT-1: INTRODUCTION TO DIFFERENT

MING

Faculty In-charge

Ms. Aaysha Shaikh

Assistant Professor (IT Dept.)

Room No. 321

email: aayshashaikh@sfit.ac.in

Academic Year: 2021-22

OUTLINE OF UNIT-1

Sub-Unit	Contents
1.1	Introduction to different programming paradigms
□ 1.2	Names, Scopes, Bindings, Scope Rules, Storage Management
1.3	Type Systems, Type checking, Equality testing, and assignment
□ 1.4	Subroutine and control abstraction, Stack layout, calling sequence, parameter passing
1.5	Generic subroutines and modules, Exception handling, co-routines and events

Module 1.2:NAMES, SCOPES, BINDINGS, SCOPE RULES, STORAGE MANAGEMENT

PROGRAMMING LANGUAGES OBEYING DIFFERENT PARADIGM

NAMES
(alphanumeric)

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

SCOPES

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

BINDING

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

NAMES

. Names enable programmers to refer to variables, constants, operations, and types using identifier names (alphanumeric characters) rather than low-level hardware addresses

. Symbols (like ‘+’) can also be names

 A *variable* is a letter or a symbol used to represent a value that can change

 A *constant* is a value that does not change

 Operations are performed on variables and constants to evaluate a value

 Type is the set of all values that a variable can have

Let pi=3.1415926535 (definition)

In pi*pi (use)

. Valid names vary by languages

. Most languages allow you to use upper case and lower case letters

. Some distinguish between upper and lower case and some don’t

. Some languages allow you to use ‘ _’, ‘-’, …it all varies

. Reserved words are generally not allowed to be used as names

Some name shaped words have special meanings.

Eg. if, while, def, return, class…..

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

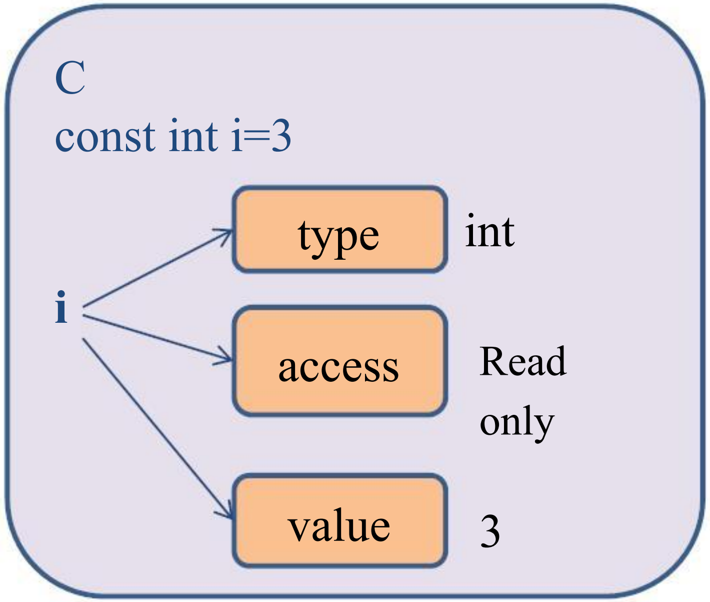
Association between people to form years…life

int num=5;

Association between memory (name) and constant (object)

DEFINITIONS AND BINDINGS

Each definition introduces bindings:



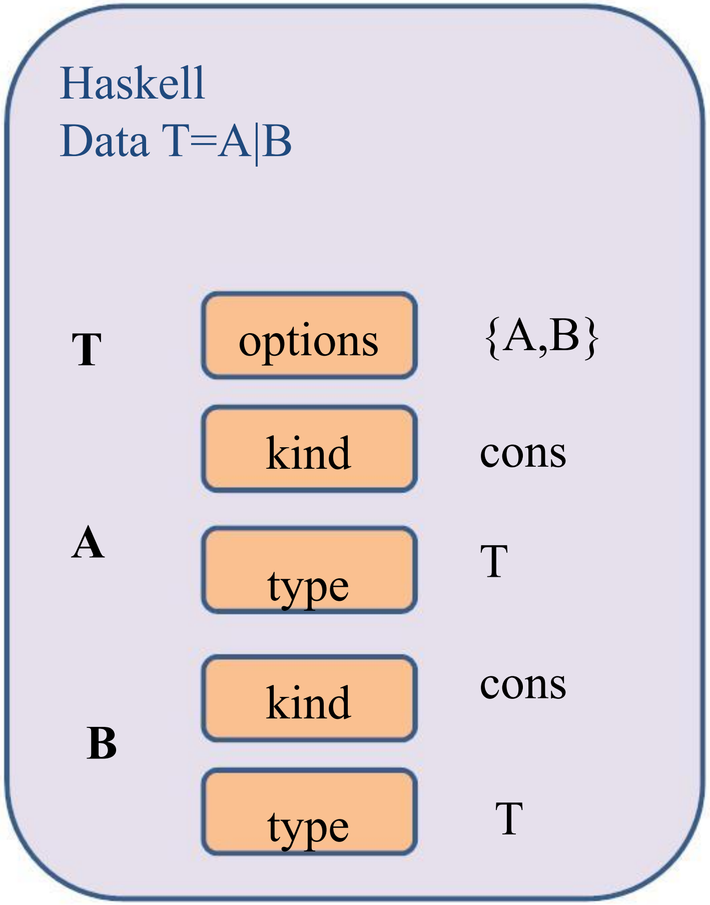
C
const int i=3

i

type int

access Read only

value 3



Haskell
Data T=A|B

T

options {A,B}

kind cons

A

type T

kind cons

B

type T

Binding maps names to attributes

int num=5; → Association between memory (name) and constant (object)

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

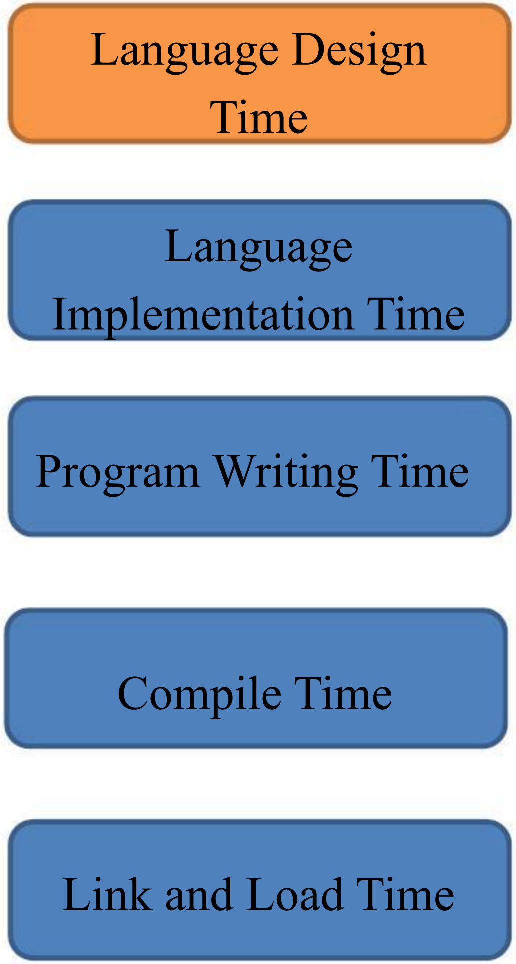
BINDING TIME

- 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.



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)*

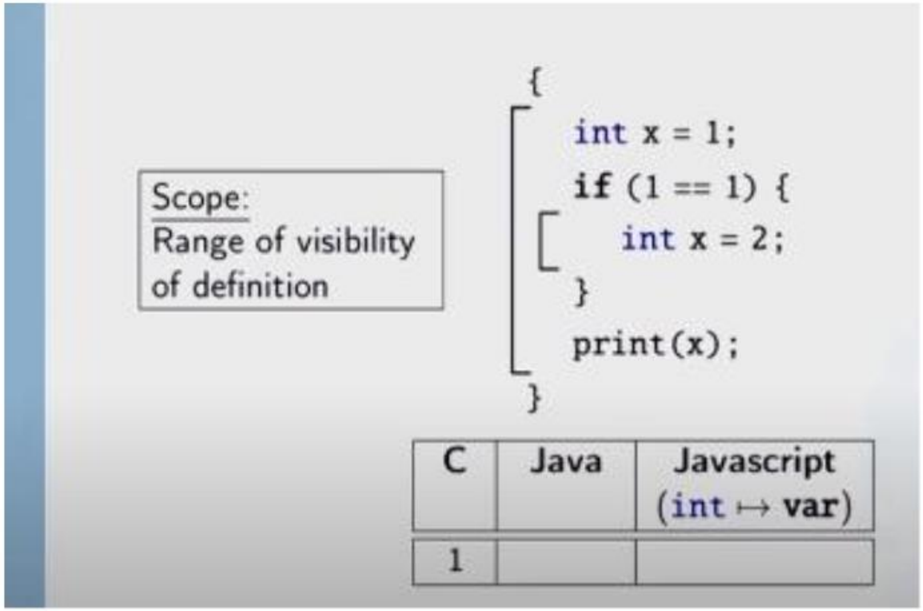
Link and Load Time

. 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- entire span from start to end

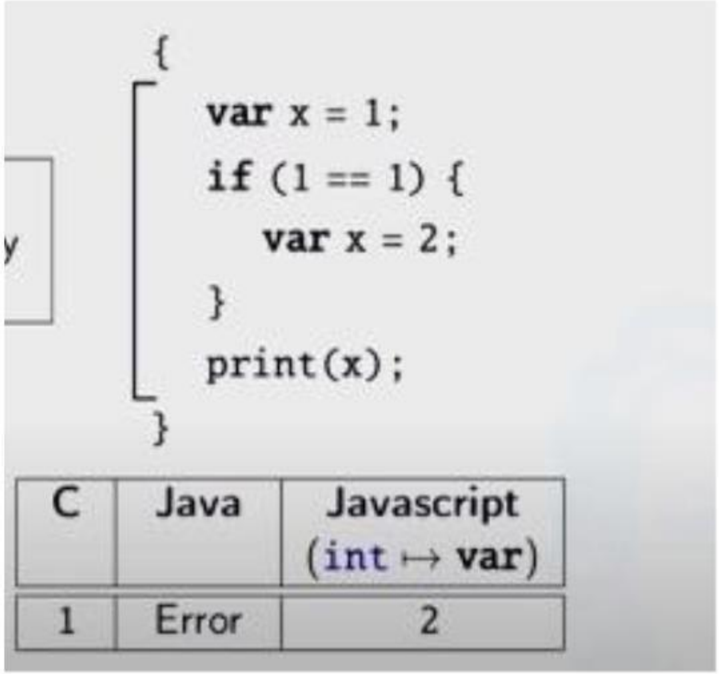
SCOPES



Scope:
Range of visibility
of definition

```
{
  int x = 1;
  if (1 == 1) {
     int x = 2;
  }
  print(x);
}
```

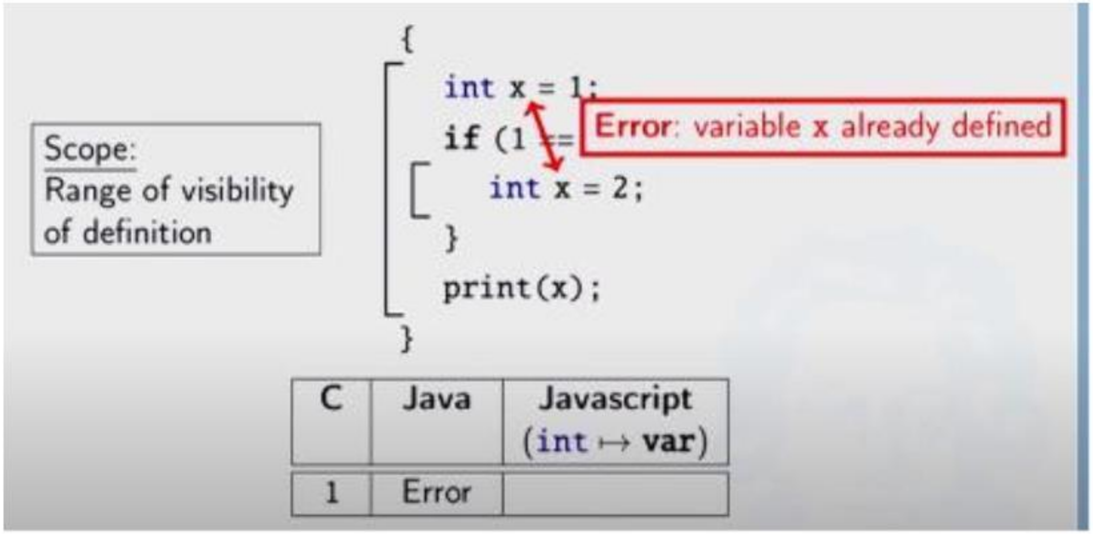
C	Java	Javascript (int ↦ var)
1		



y

```
{
  var x = 1;
  if (1 == 1) {
     var x = 2;
  }
  print(x);
}
```

C	Java	Javascript (int ↦ var)
1	Error	2



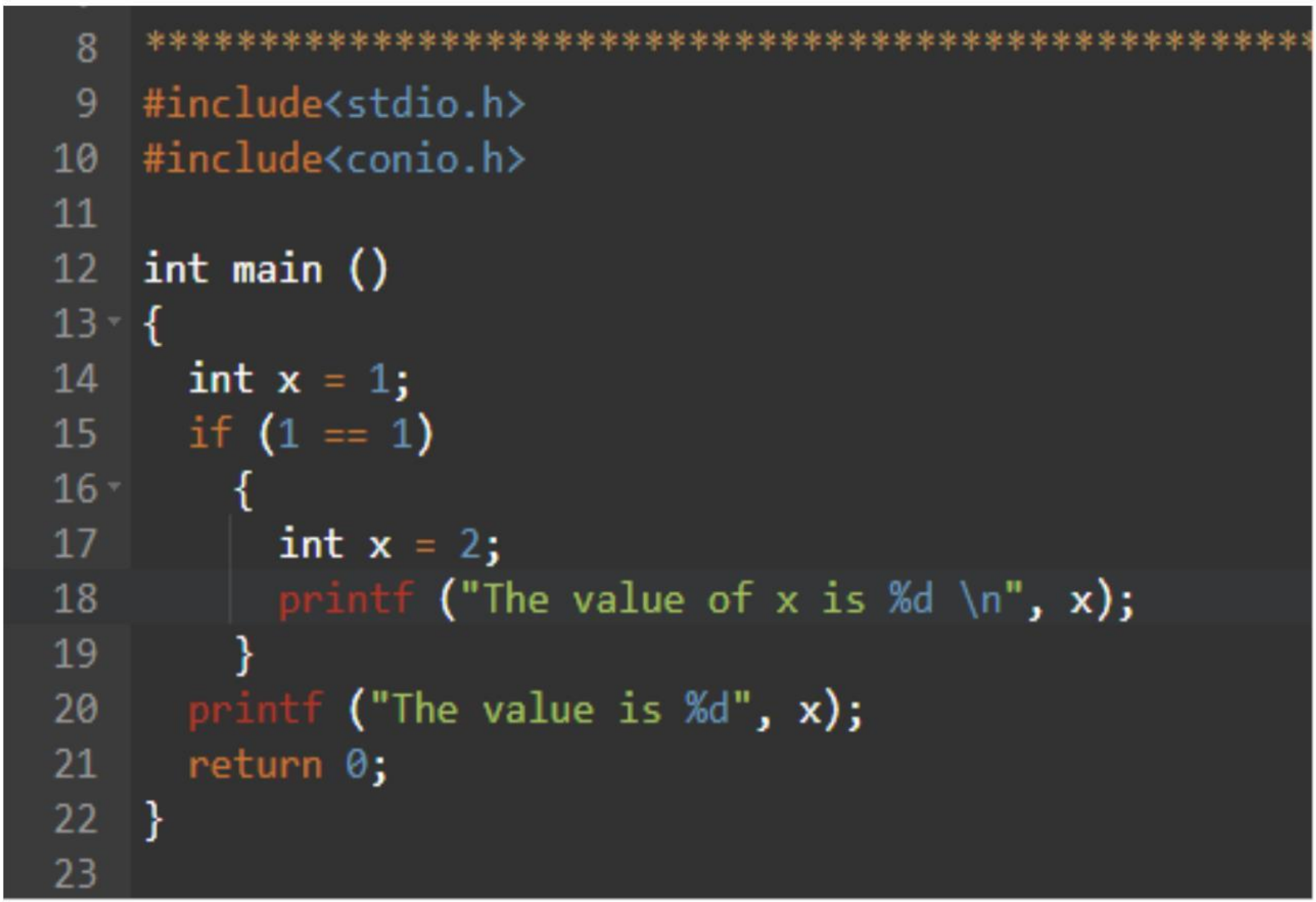
Scope:
Range of visibility
of definition

```
{
  int x = 1;
  if (1 ==
     int x = 2;
  }
  print(x);
}
```

Error: variable x already defined

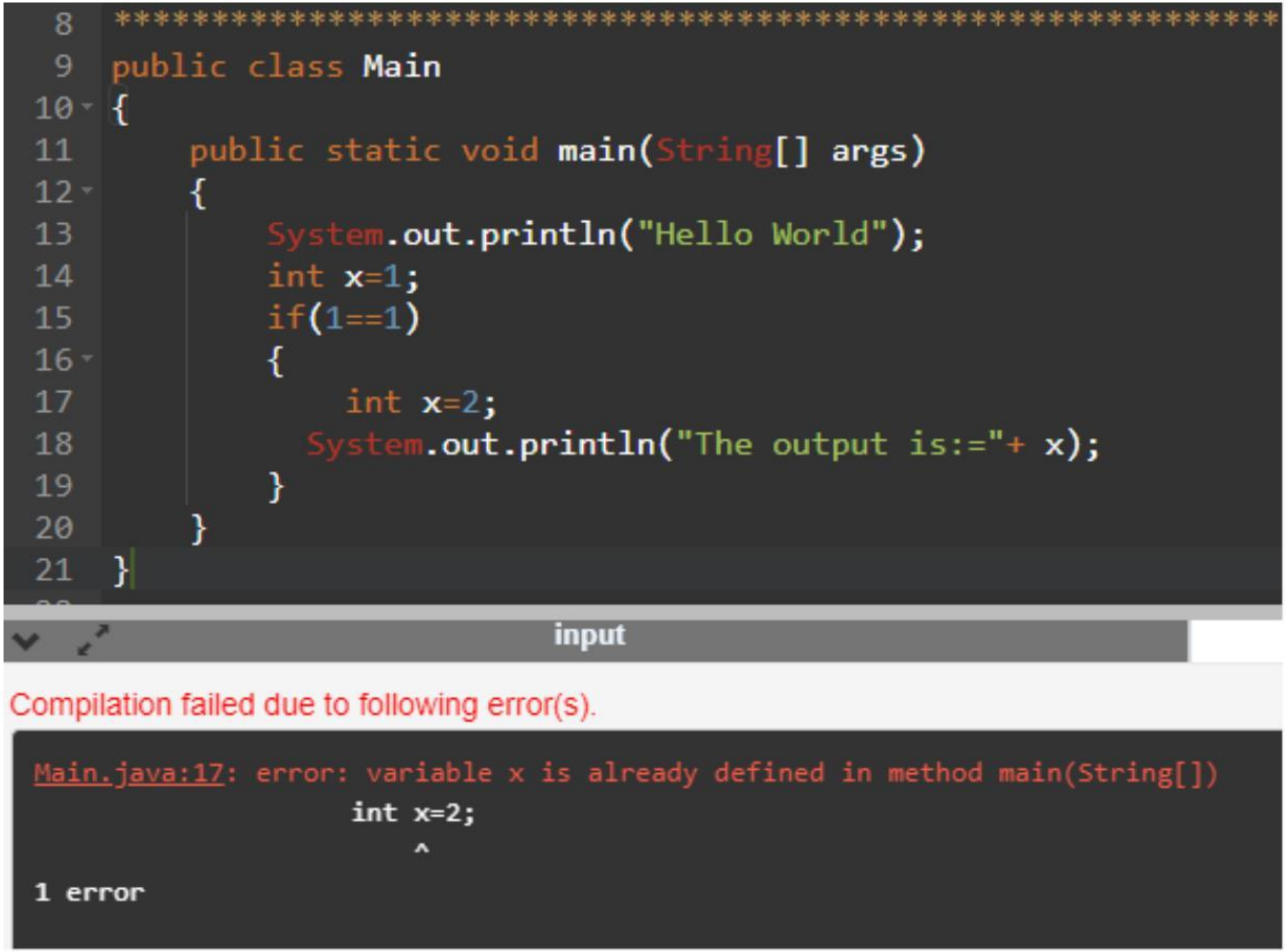
C	Java	Javascript (int ↦ var)
1	Error	

Different languages use different scoping rules



```
***************************************************
#include<stdio.h>
#include<conio.h>

int main ()
{
  int x = 1;
  if (1 == 1)
    {
      int x = 2;
      printf ("The value of x is %d \n", x);
    }
  printf ("The value is %d", x);
  return 0;
}

```

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



```
 8  *************************************************************
 9  public class Main
10  {
11      public static void main(String[] args)
12      {
13          System.out.println("Hello World");
14          int x=1;
15          if(1==1)
16          {
17              int x=2;
18            System.out.println("The output is:="+ x);
19          }
20      }
21  }
```

Compilation failed due to following error(s).

```
Main.java:17: error: variable x is already defined in method main(String[])
                 int x=2;
                     ^

1 error
```

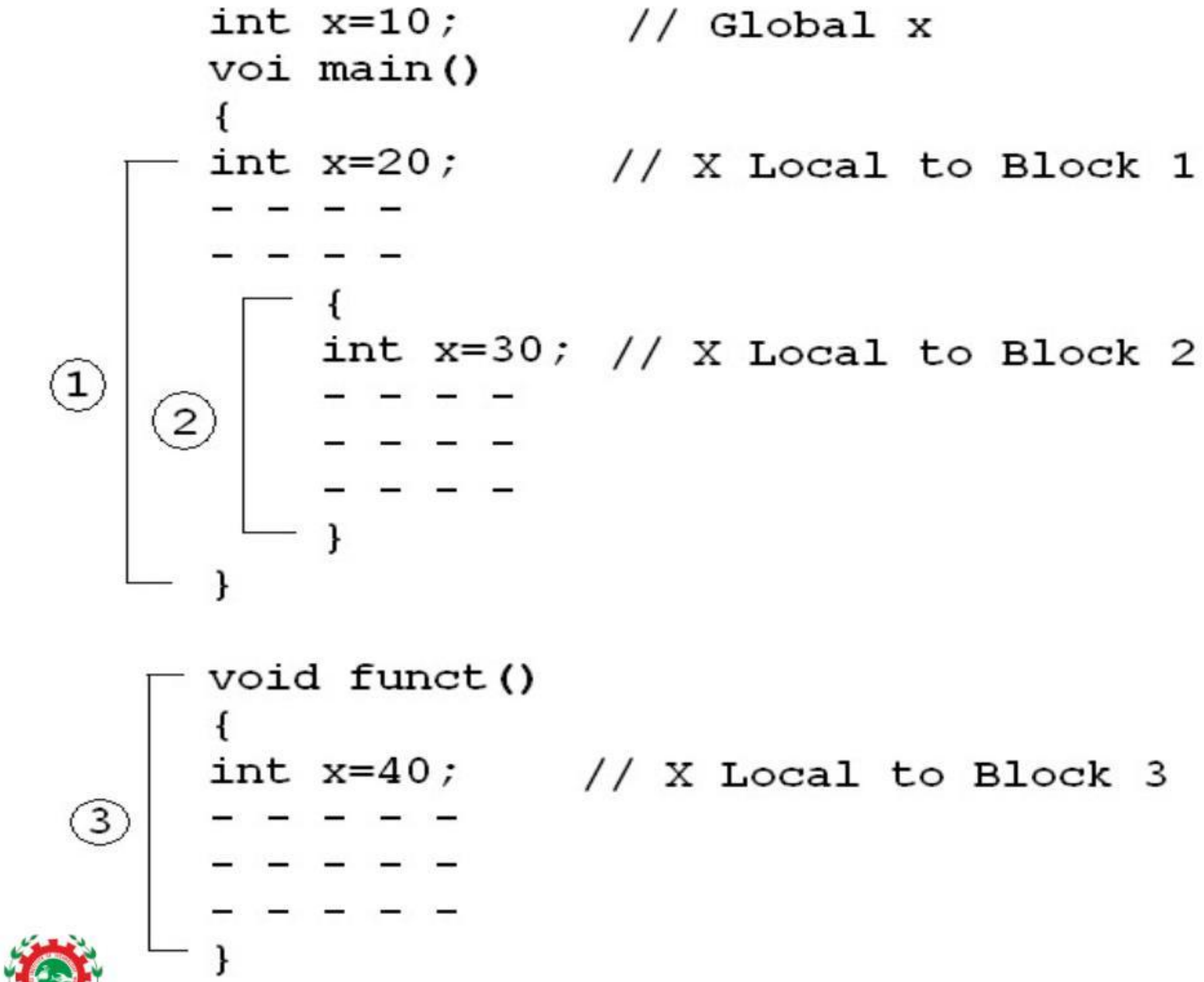
Online tools used: onlinegdb-online compiler for c/C++, Online C compiler -Jdoodle

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.



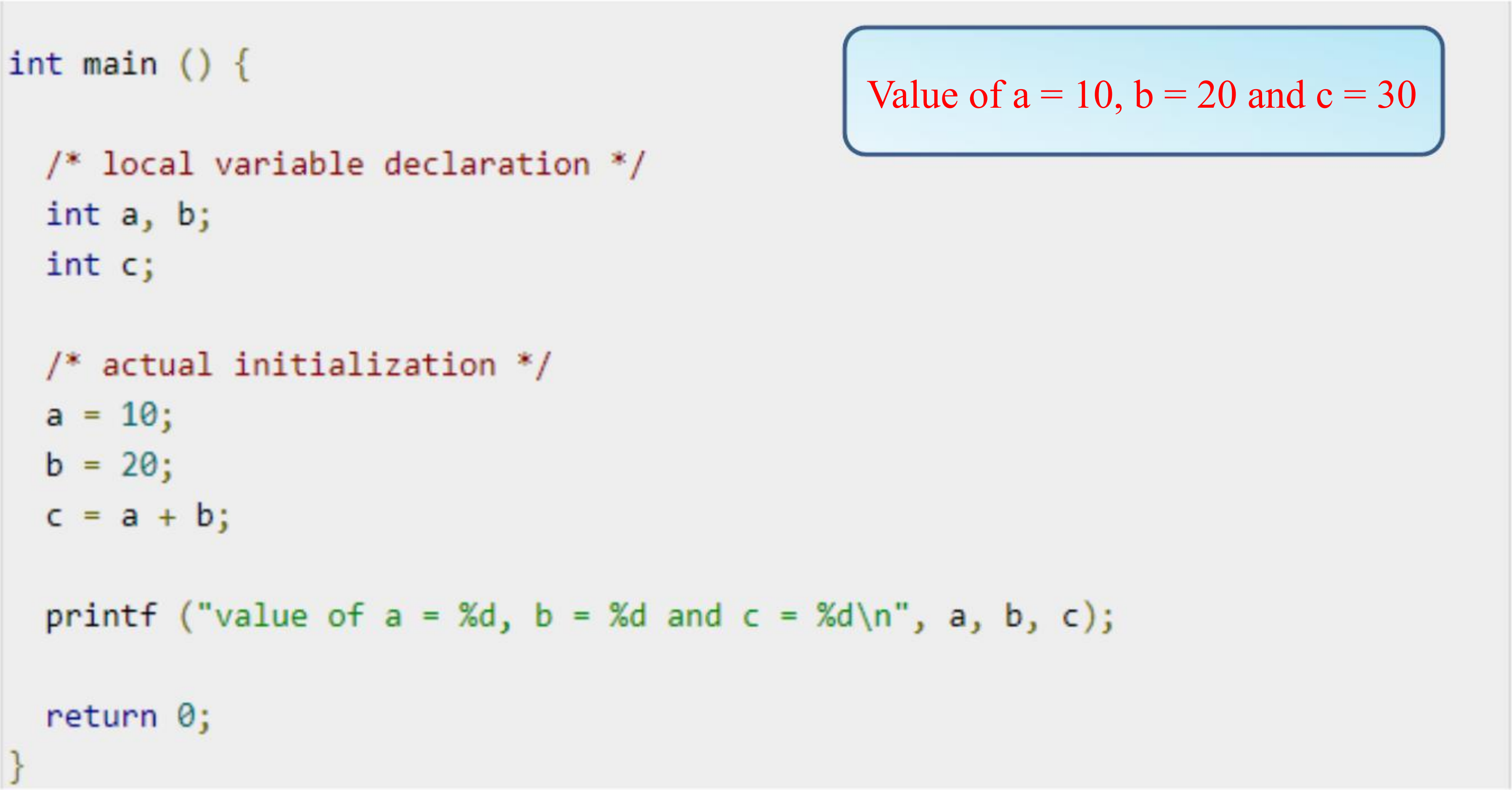
```
int x=10;       // Global x
voi main()
{
int x=20;      // X Local to Block 1
- - - -
- - - -
    {
    int x=30; // X Local to Block 2
    - - - -
    - - - -
    - - - -
    }
}

void funct()
{
int x=40;     // X Local to Block 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 functions outside their own.



```
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

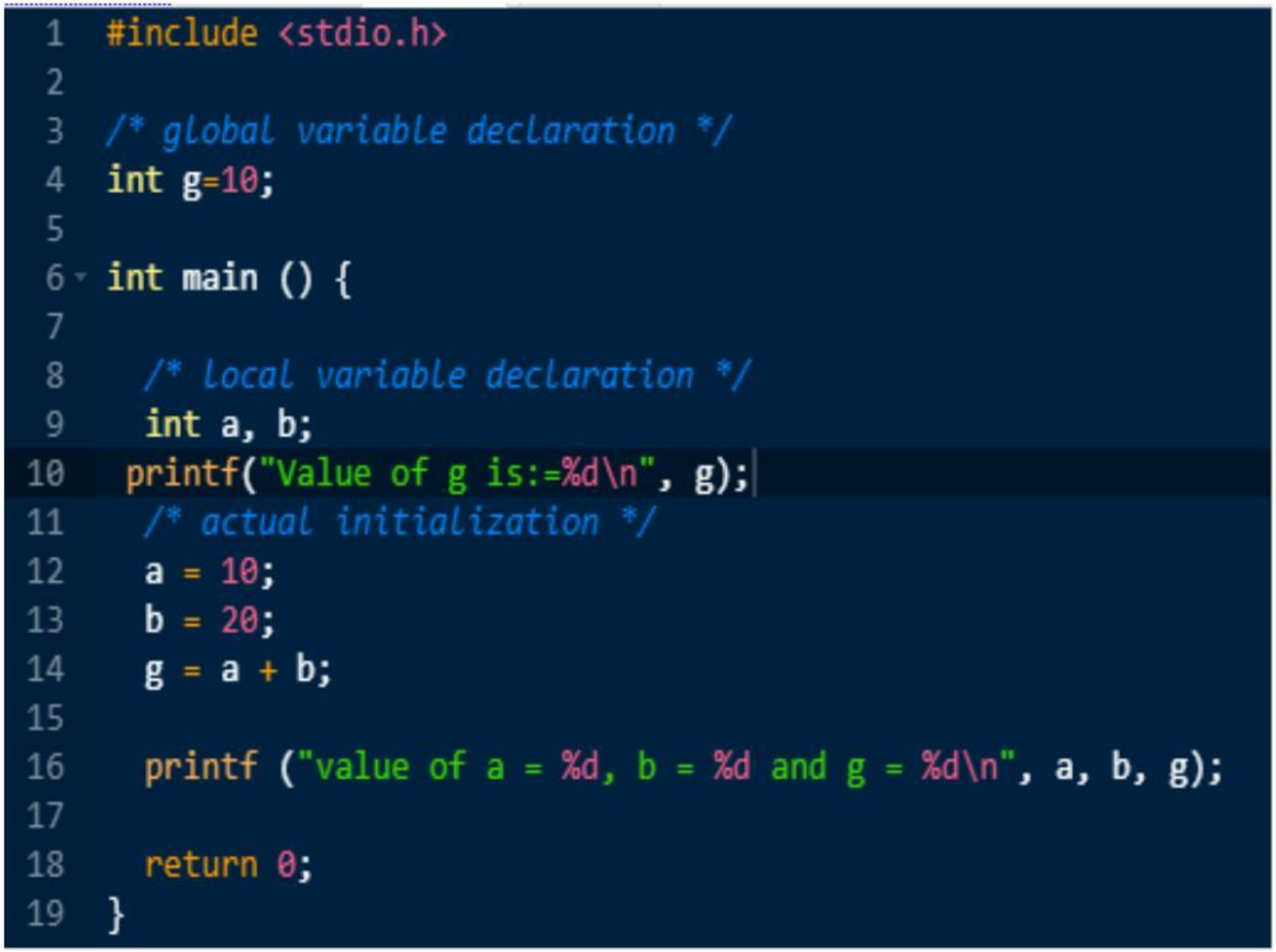
Online tools used: onlinegdb-online compiler for c/C++, Online C compiler -Jdoodle

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.



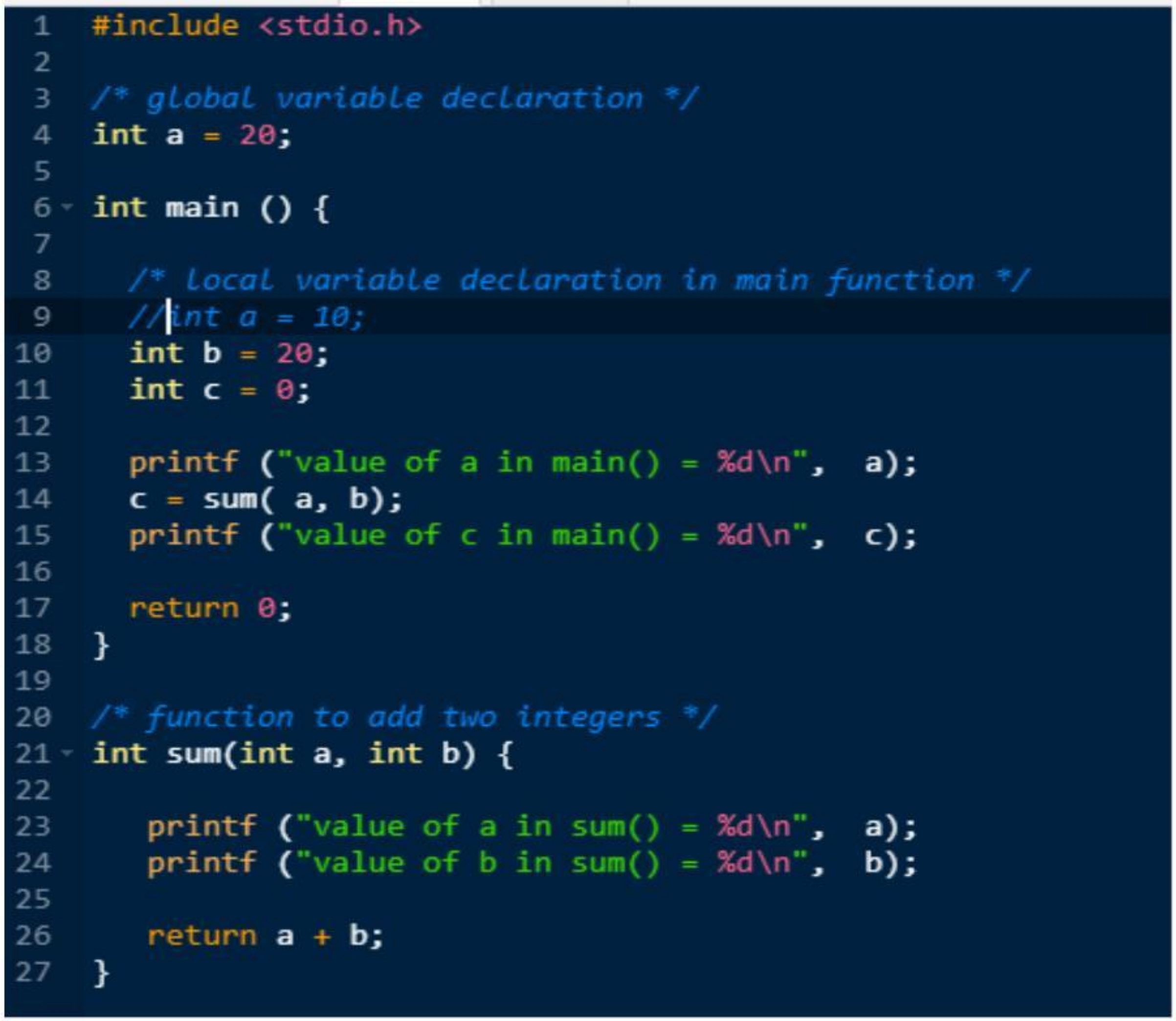
```c
#include <stdio.h>

/* 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
```

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

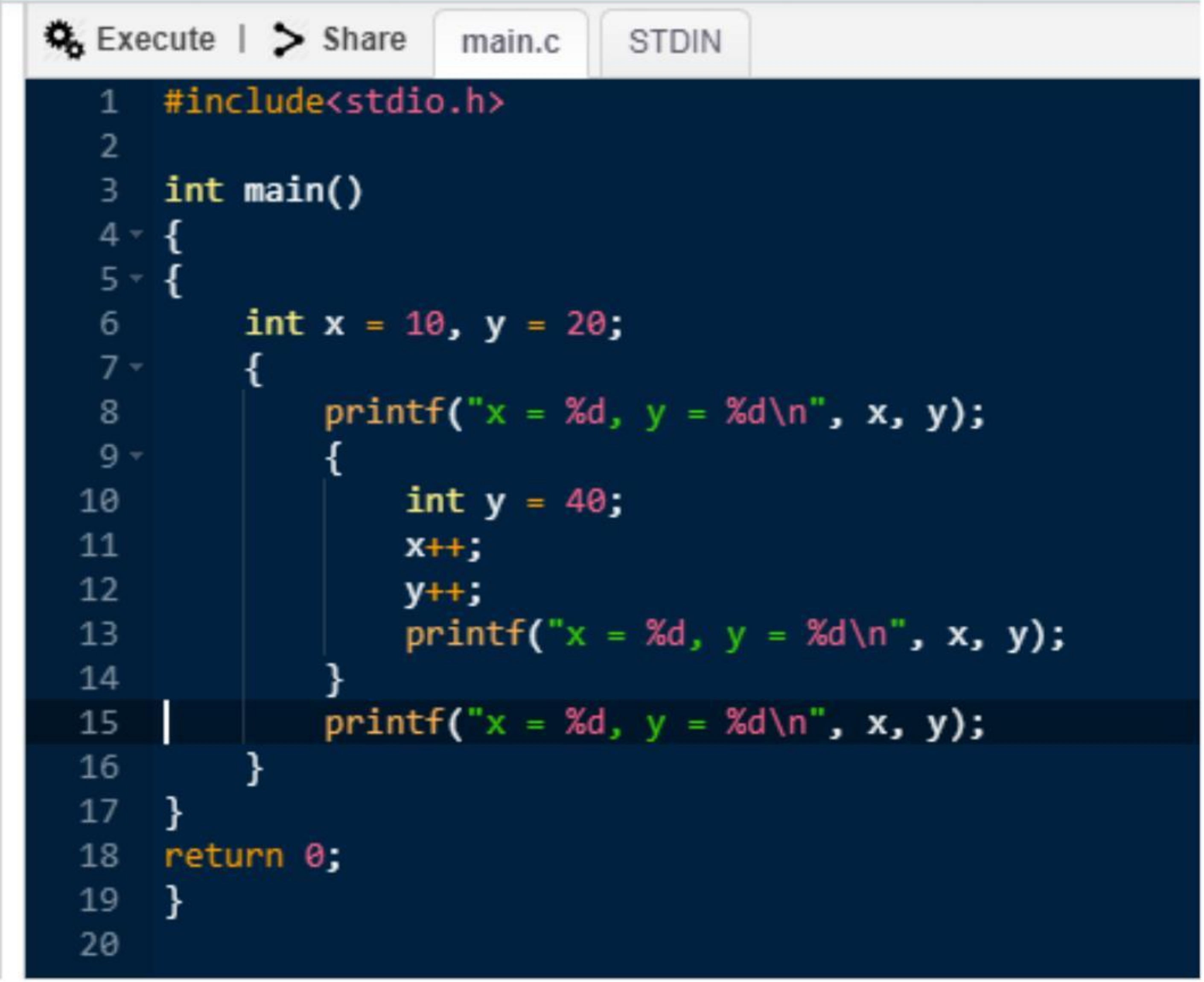
Scope in C: Formal Parameters

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



```
#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



```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;
}
```

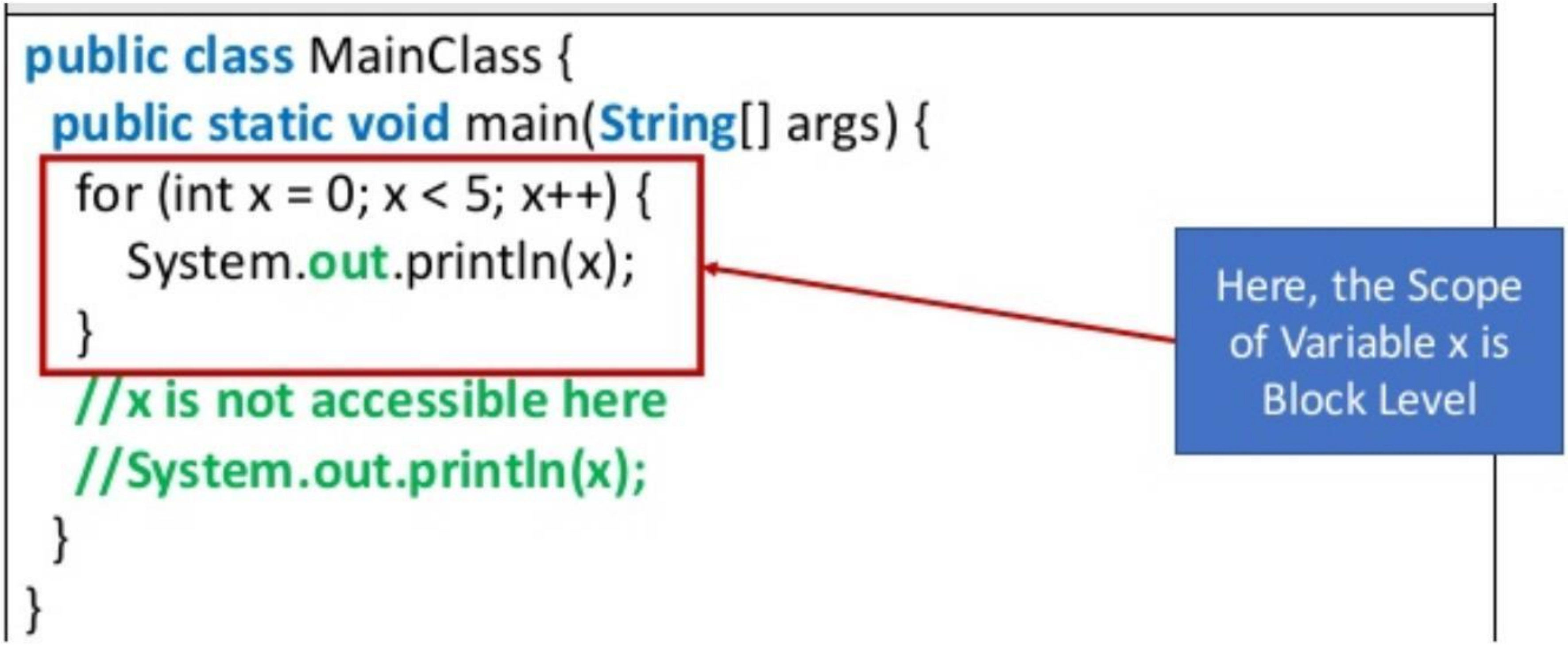
```
x = 10, y = 20
x = 11, y = 41
x = 11, y = 20
```

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

Scope in JAVA : Block Level

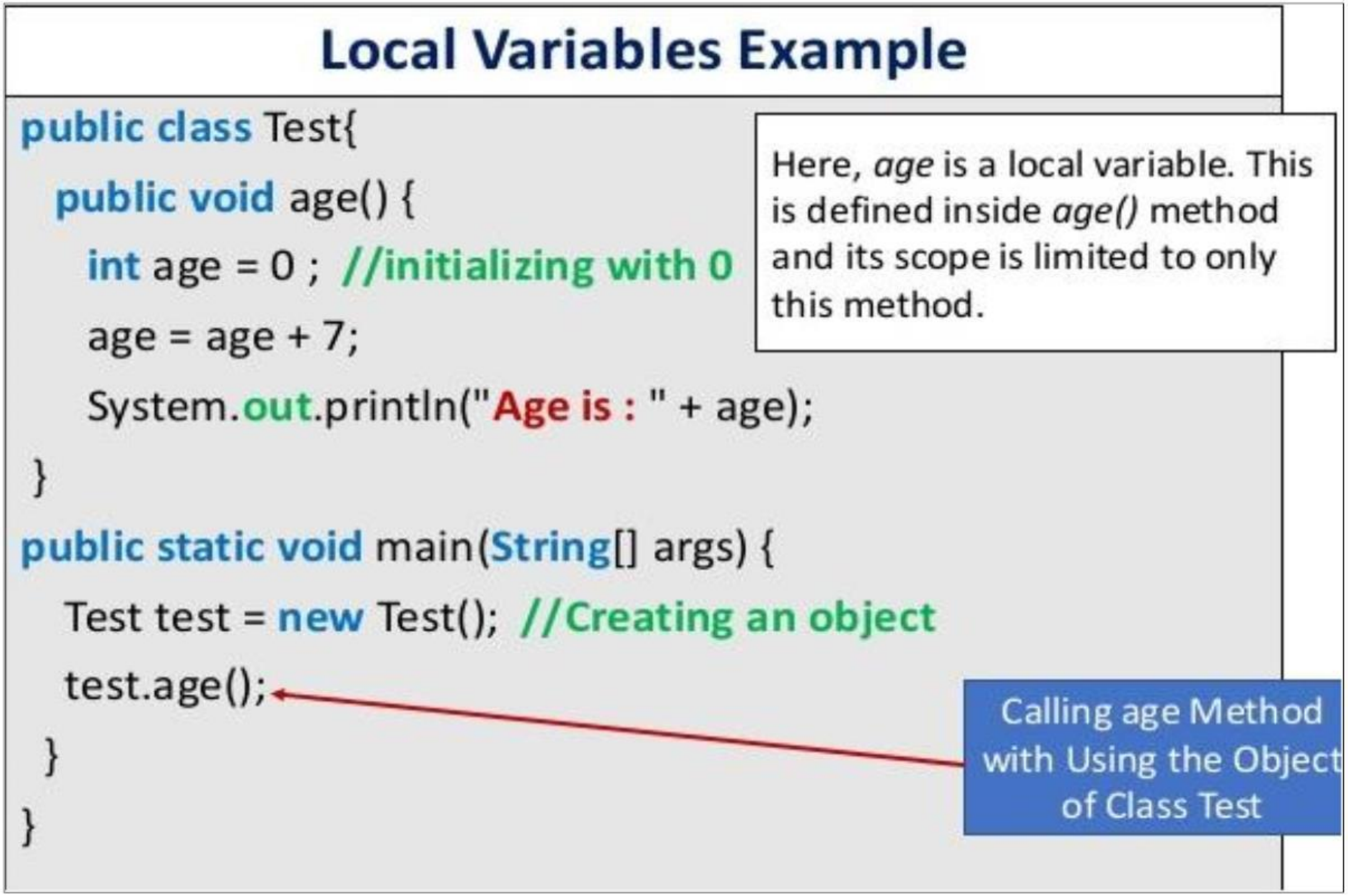
. The variables that are defined in a block are only accessible from within the block.

. 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

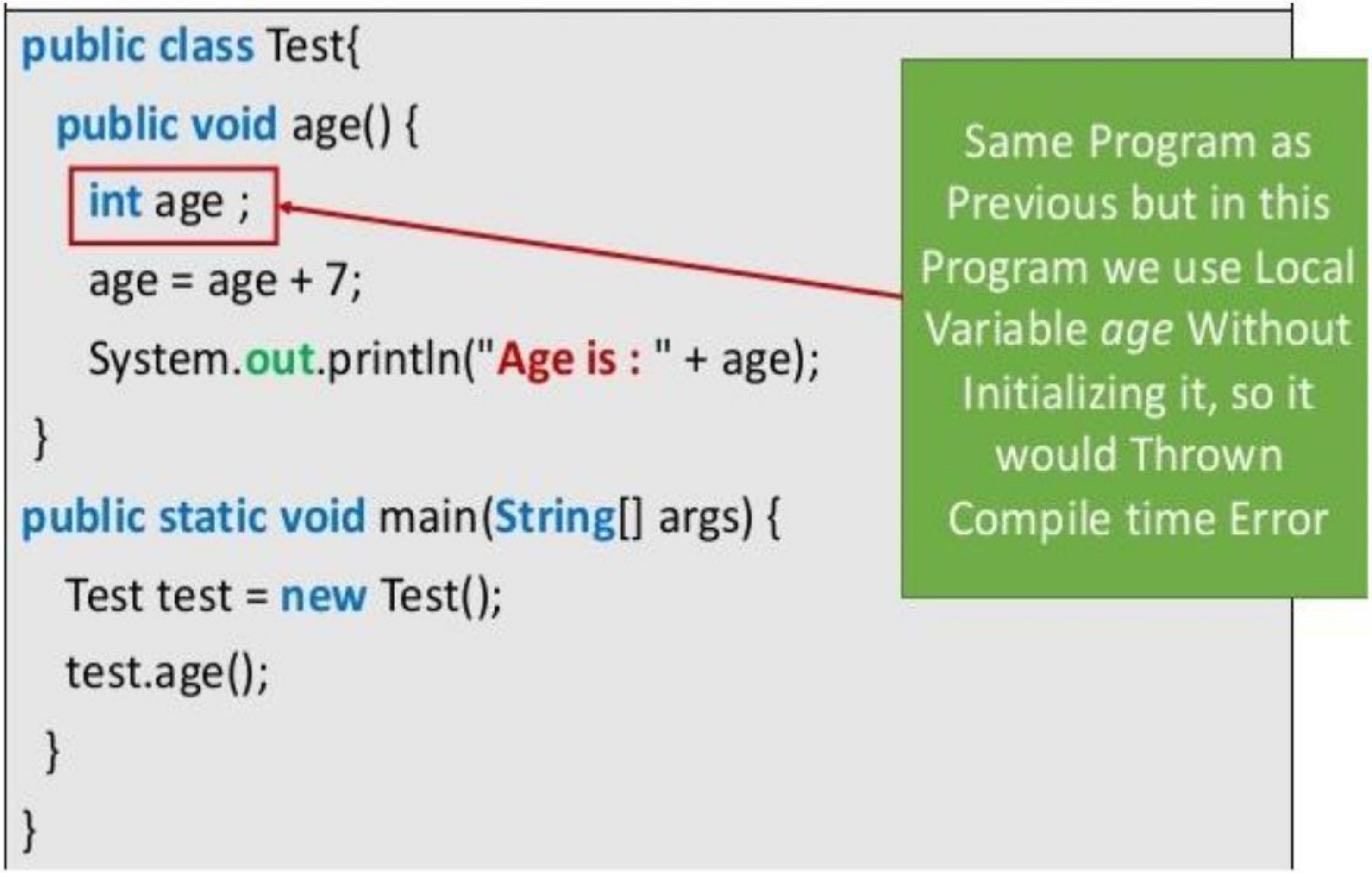


Local Variables Example

```
public class Test{
  public void age() {
    int age = 0 ;  //initializing with 0
    age = age + 7;
    System.out.println("Age is : " + age);
 }
public static void main(String[] args) {
   Test test = new Test();  //Creating an object
   test.age();
  }
}
```

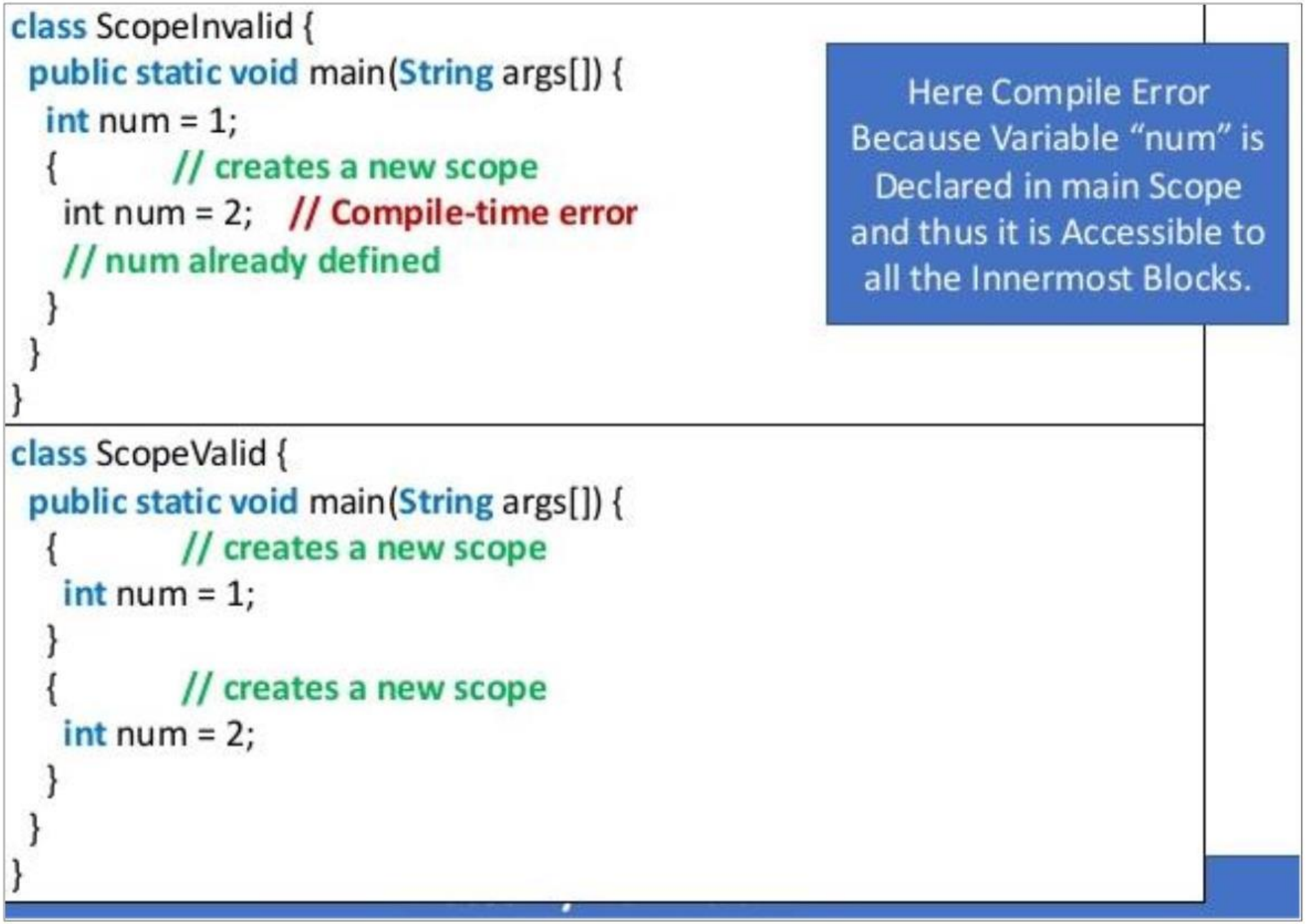
Here, *age* is a local variable. This is defined inside *age()* method and its scope is limited to only this method.

Calling age Method with Using the Object of Class Test



```
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();
 }
}
```

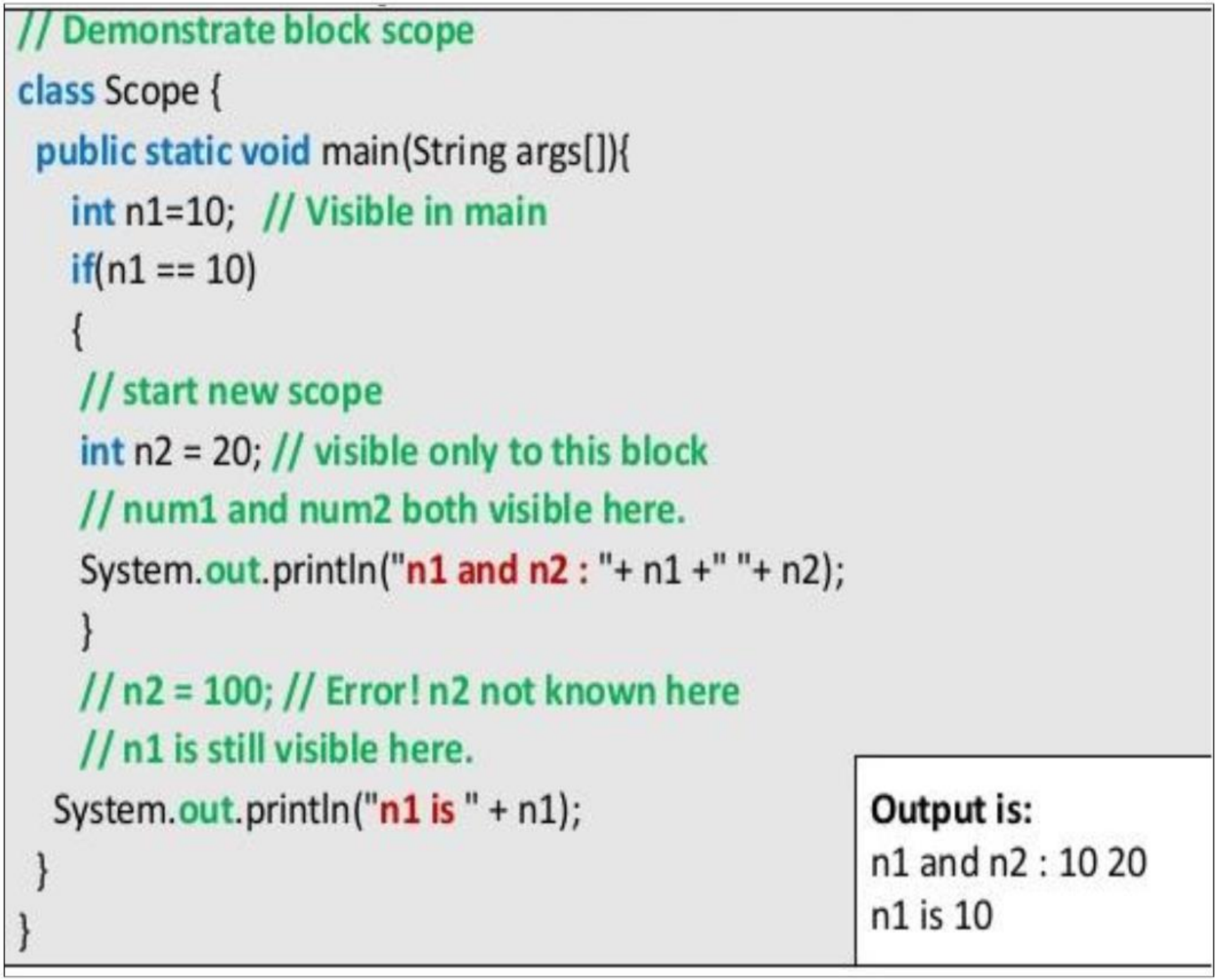
Same Program as Previous but in this Program we use Local Variable *age* Without Initializing it, so it would Thrown Compile time Error



```
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.

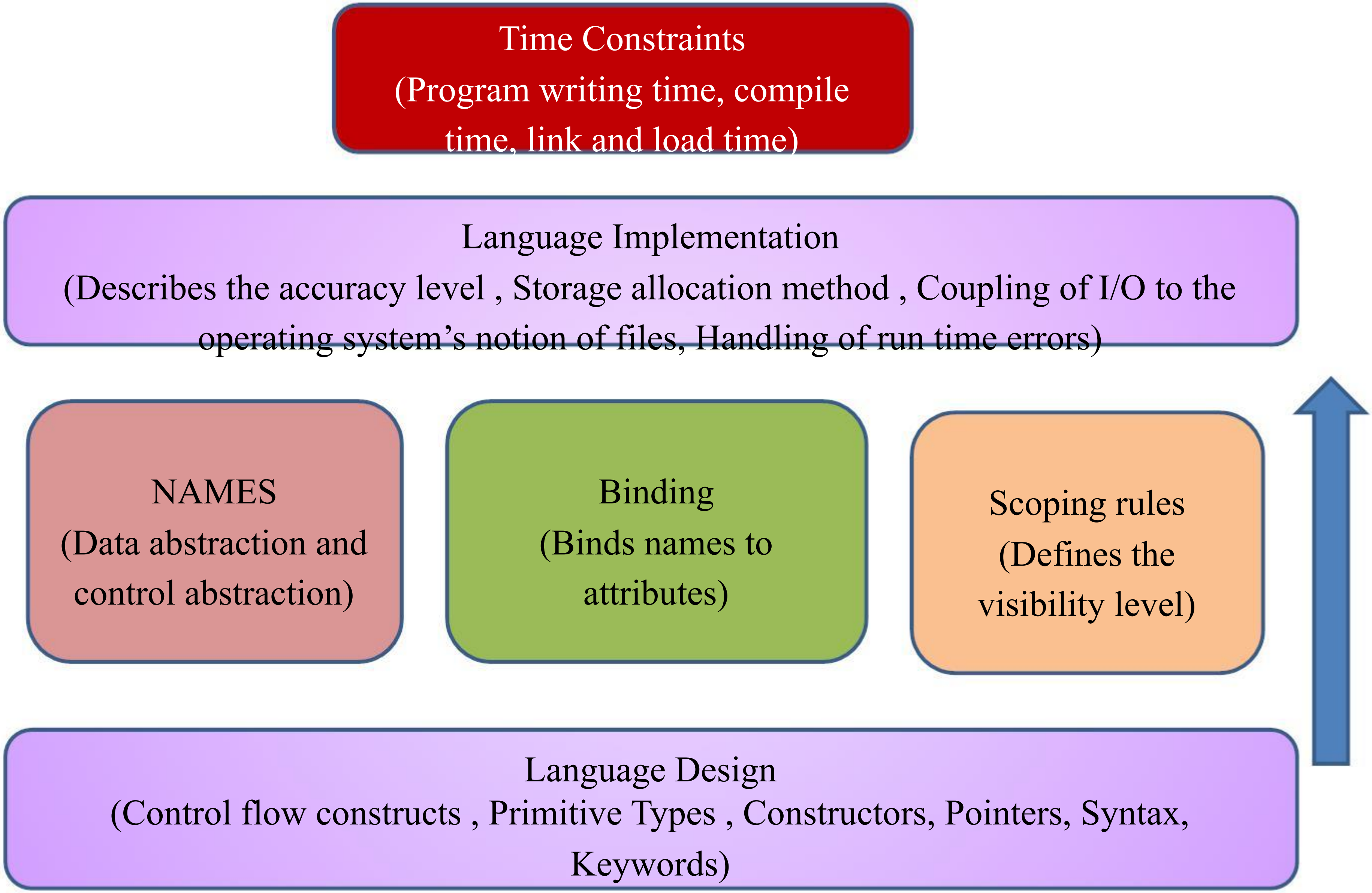
```
class ScopeValid {
 public static void main(String args[]) {
  {          // creates a new scope
   int num = 1;
  }
  {          // creates a new scope
   int num = 2;
  }
 }
}
```



```
// 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
```



Time Constraints
(Program writing time, compile time, link and load time)

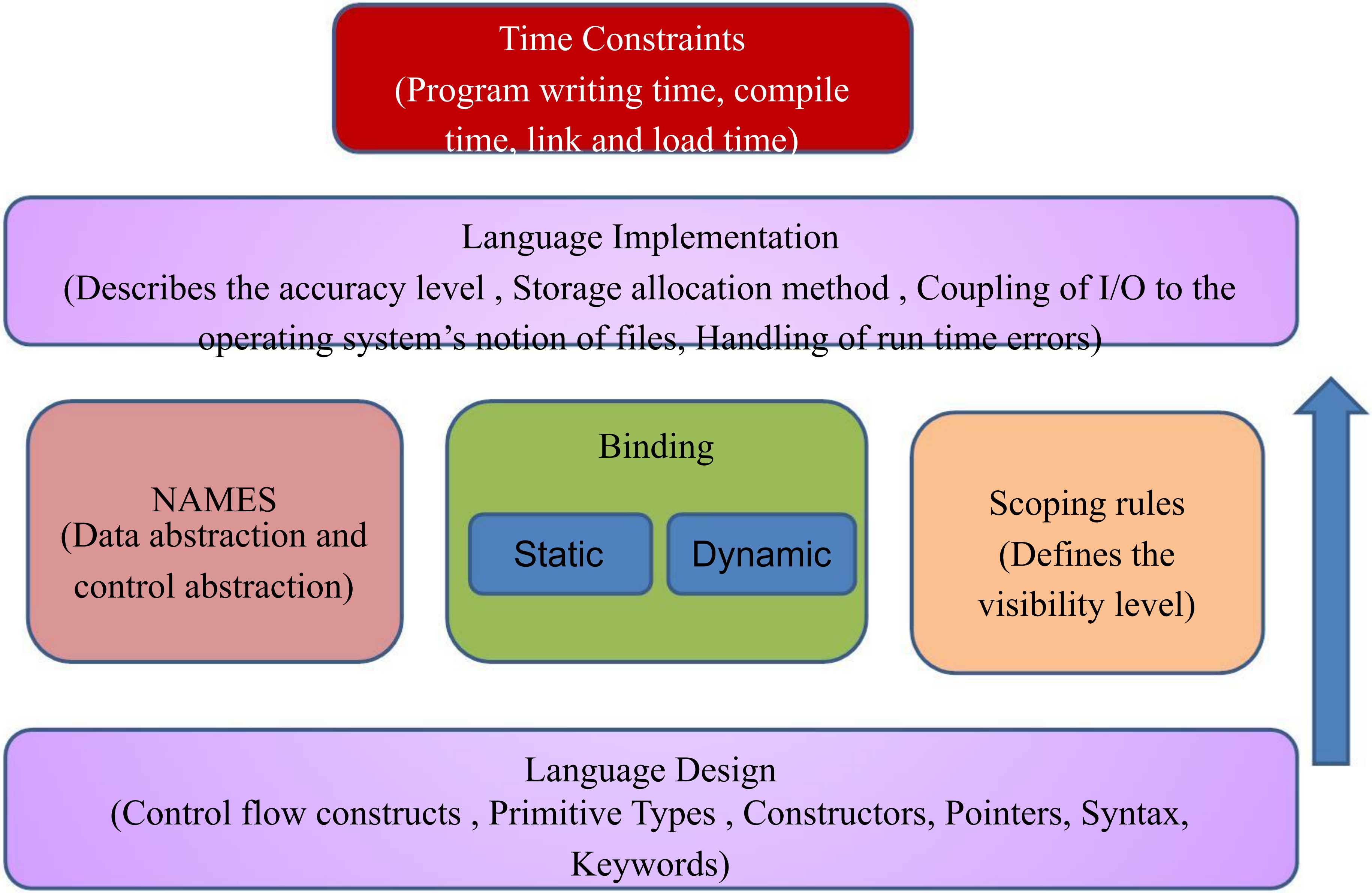
Language Implementation
(Describes the accuracy level , Storage allocation method , Coupling of I/O to the operating system's notion of files, Handling of run time errors)

NAMES
(Data abstraction and control abstraction)

Binding
(Binds names to attributes)

Scoping rules
(Defines the visibility level)

Language Design
(Control flow constructs , Primitive Types , Constructors, Pointers, Syntax, Keywords)



Time Constraints
(Program writing time, compile time, link and load time)

Language Implementation
(Describes the accuracy level , Storage allocation method , Coupling of I/O to the operating system's notion of files, Handling of run time errors)

NAMES
(Data abstraction and control abstraction)

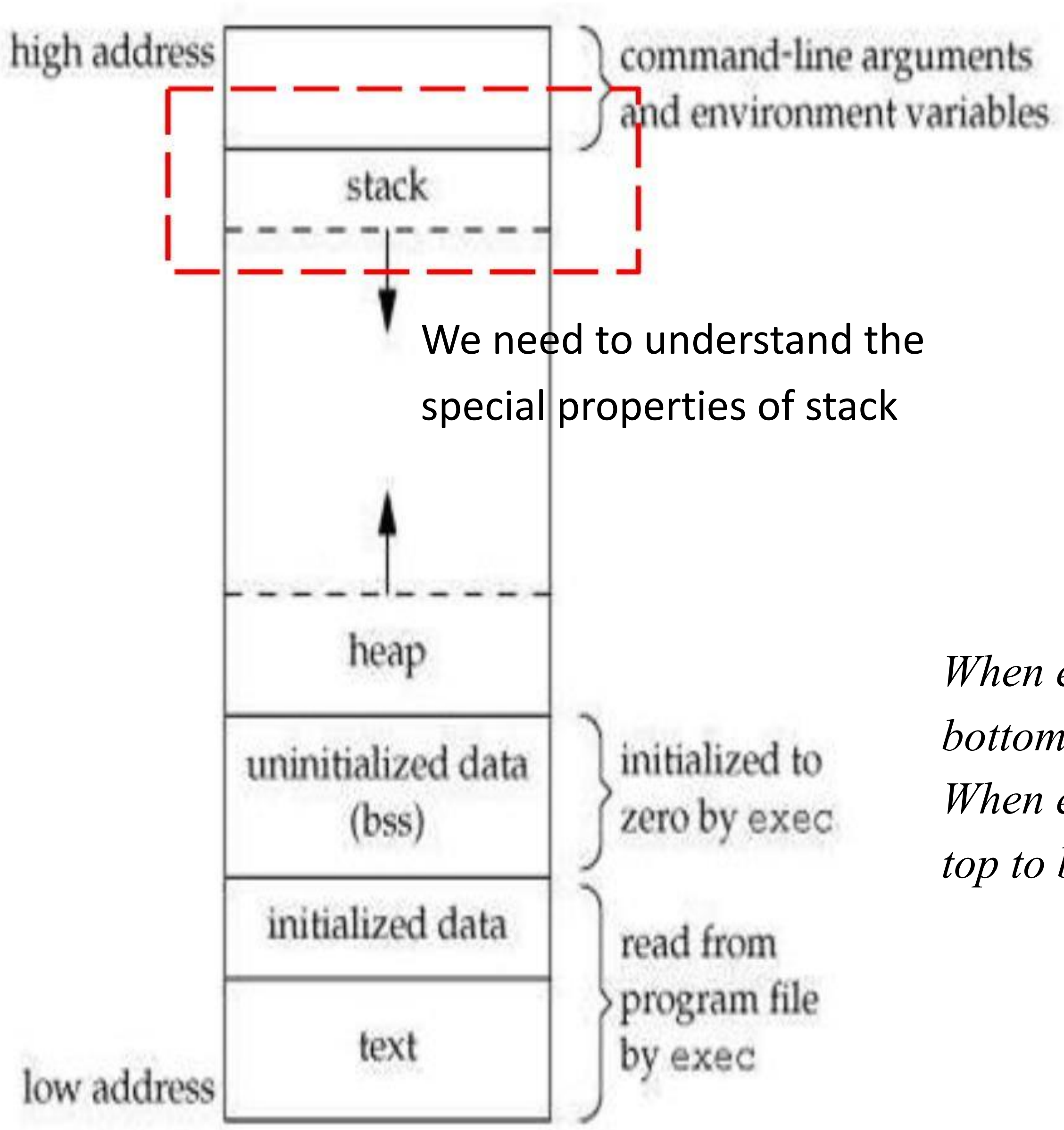
Binding

Static

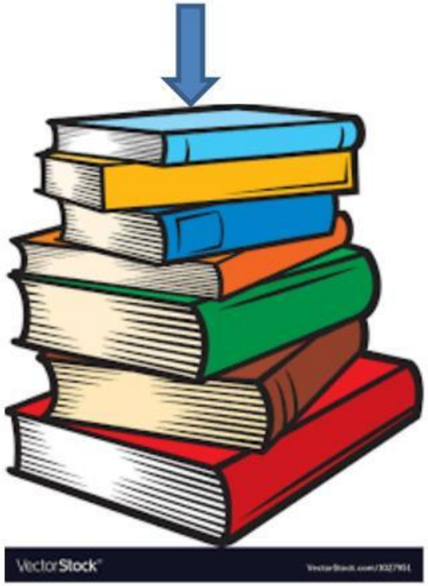
Dynamic

Scoping rules
(Defines the visibility level)

Language Design
(Control flow constructs , Primitive Types , Constructors, Pointers, Syntax, Keywords)



We need to understand the special properties of stack



When ever we are placing…its bottom to top

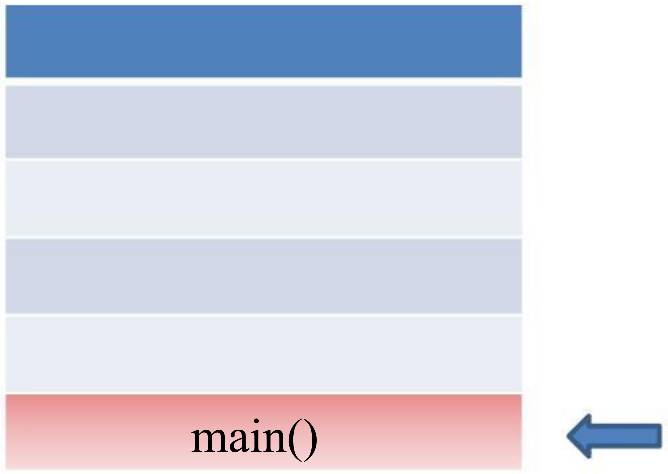
When ever we are removing ….its top to bottom

. 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

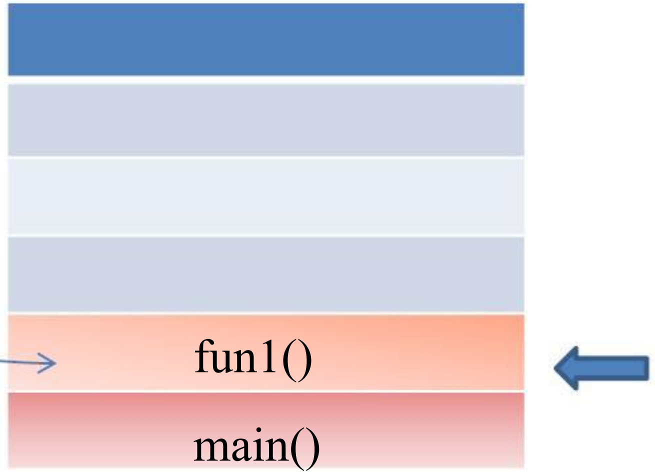
. Data is retrieved in Last in First Out fashion

. 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

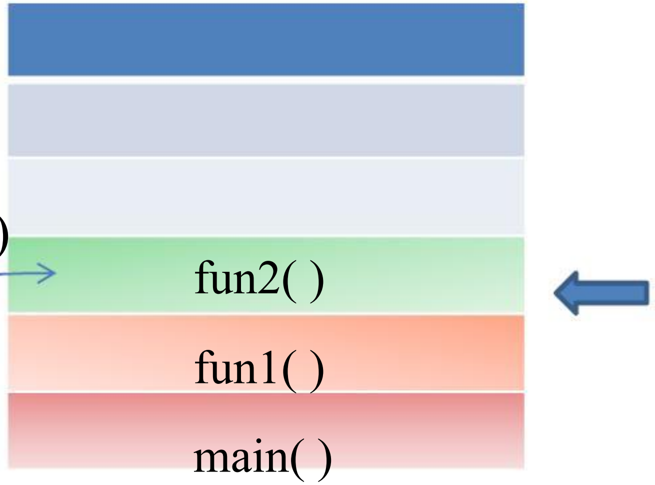
. Data is retrieved in Last in First Out fashion

. 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

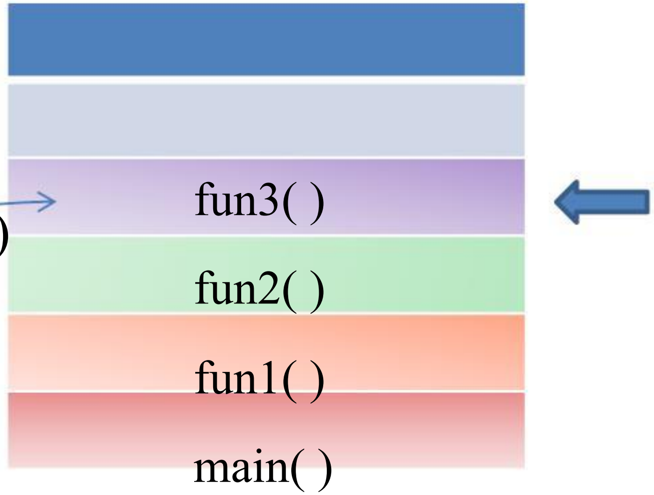
. Data is retrieved in Last in First Out fashion

. 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

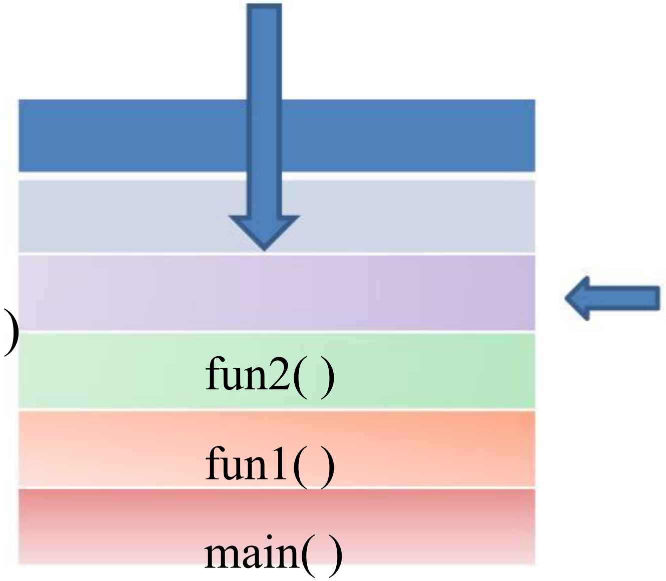
. Two operations: push and pop

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

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

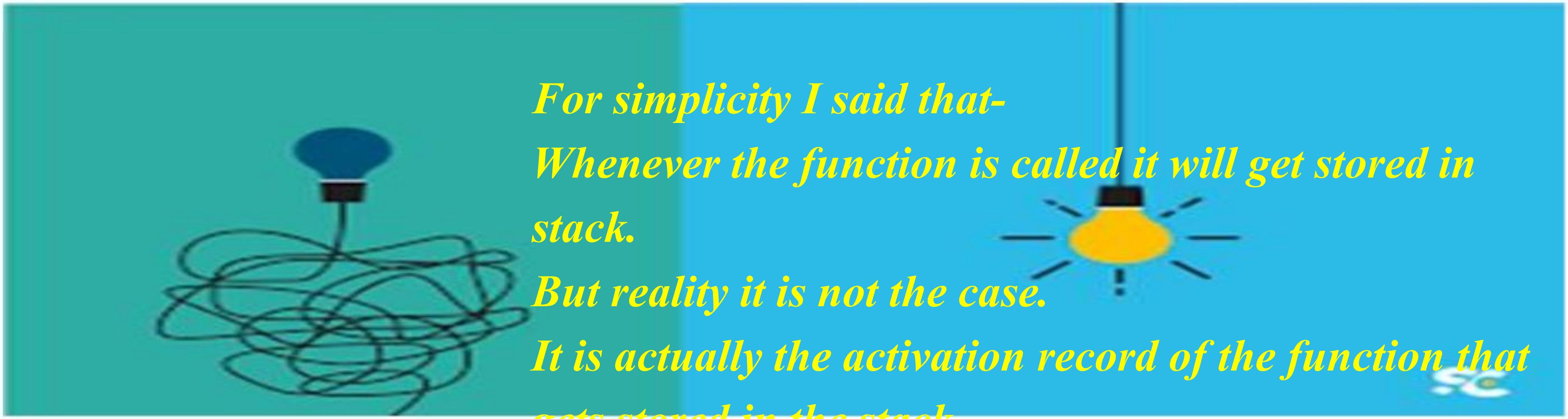
```
fun1( ) { fun2( ); }
fun2( ) { fun3( ); }
fun3( ) { return; }
```

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



Stack (or Call Stack)

Capable of storing functions



For simplicity I said that-

Whenever the function is called it will get stored in stack.

But reality it is not the case.

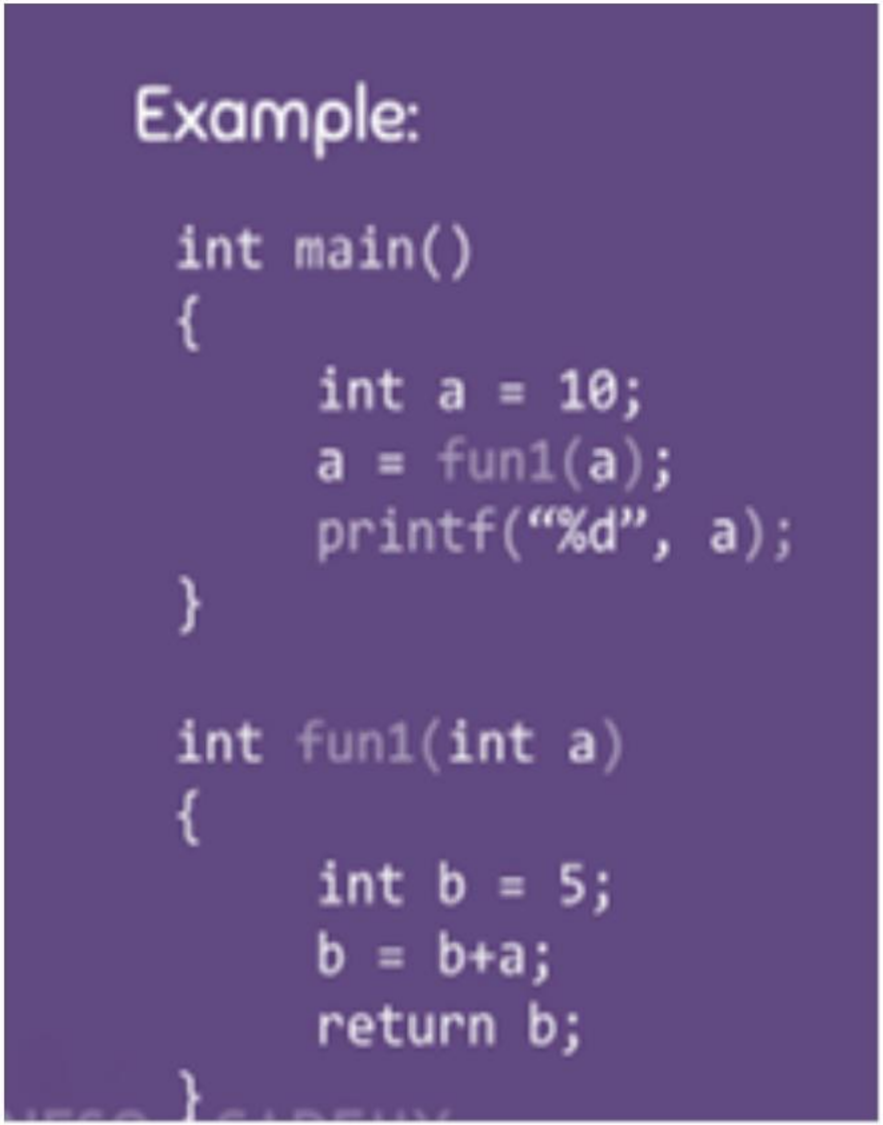
It is actually the activation record of the function that gets stored in the stack

Activation record is a portion of the stack which is generally composed of –

1. Locals of the callee

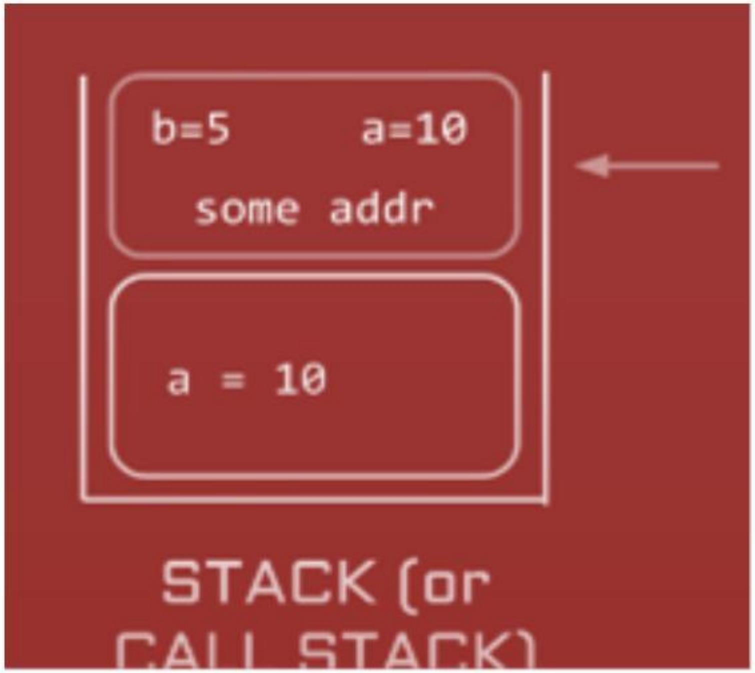
2. Return address of the caller

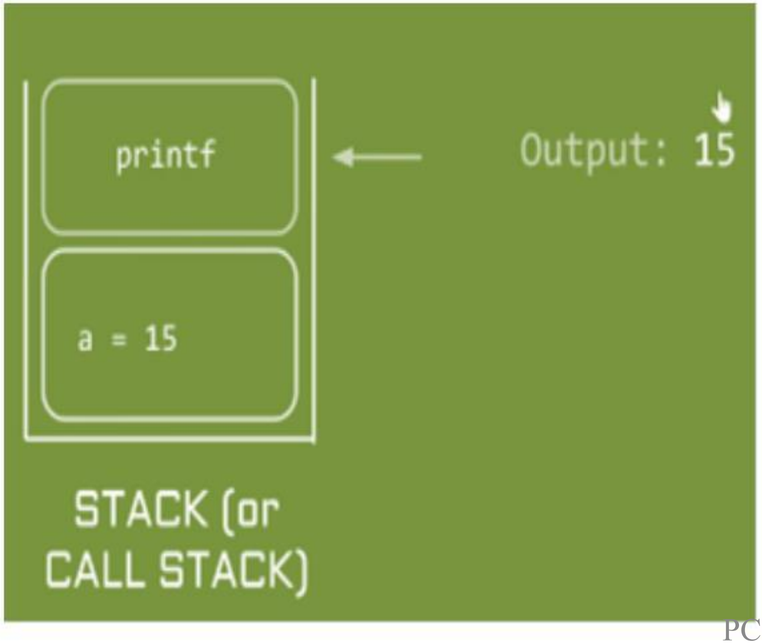
3. Parameters of the callee



Example:

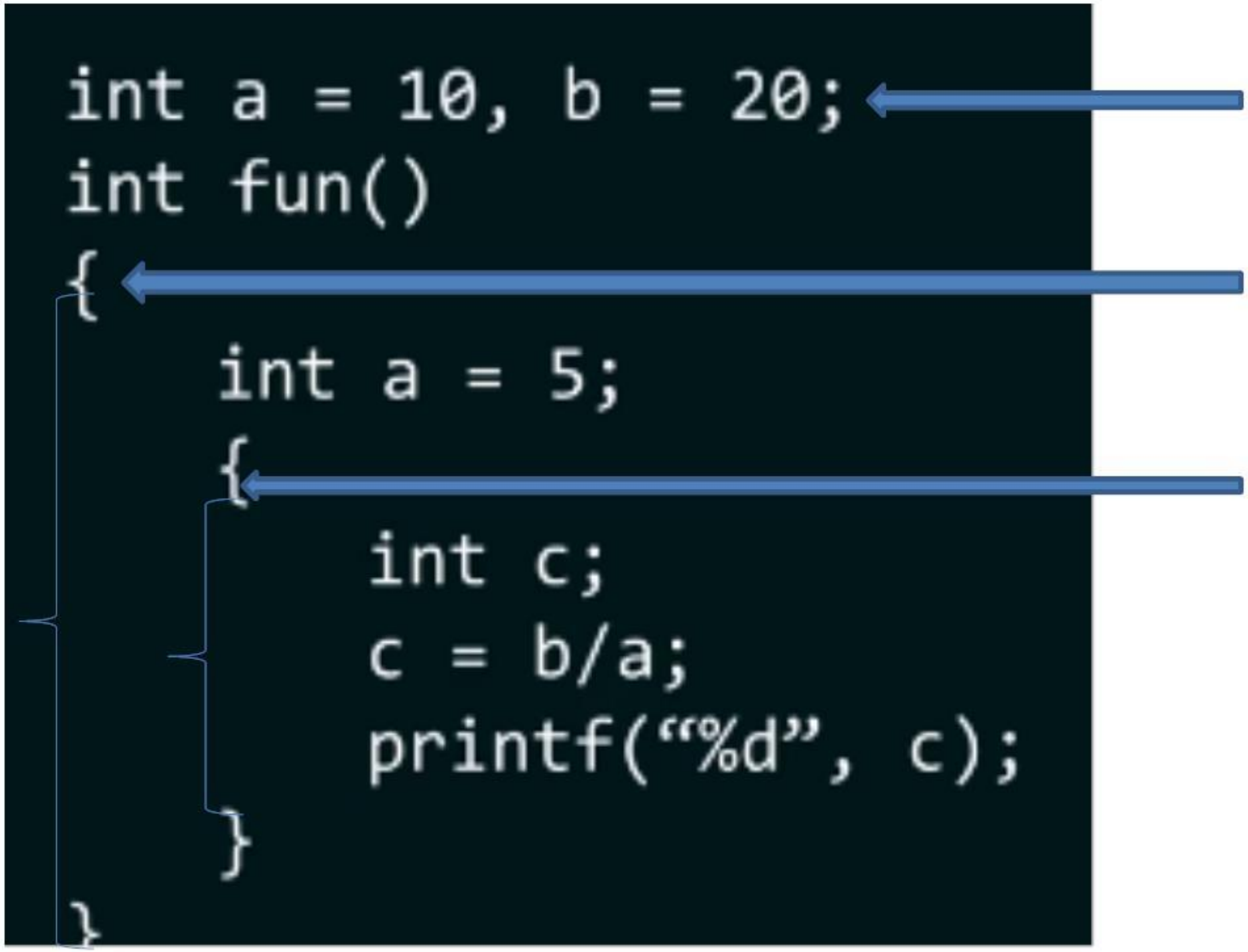
```
int main()
{
        int a = 10;
        a = fun1(a);
        printf("%d", a);
}

int fun1(int a)
{
        int b = 5;
        b = b+a;
        return b;
}
```





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



```
int a = 10, b = 20;
int fun()
{
    int a = 5;
    {
        int c;
        c = b/a;
        printf(“%d”, c);
    }
}
```

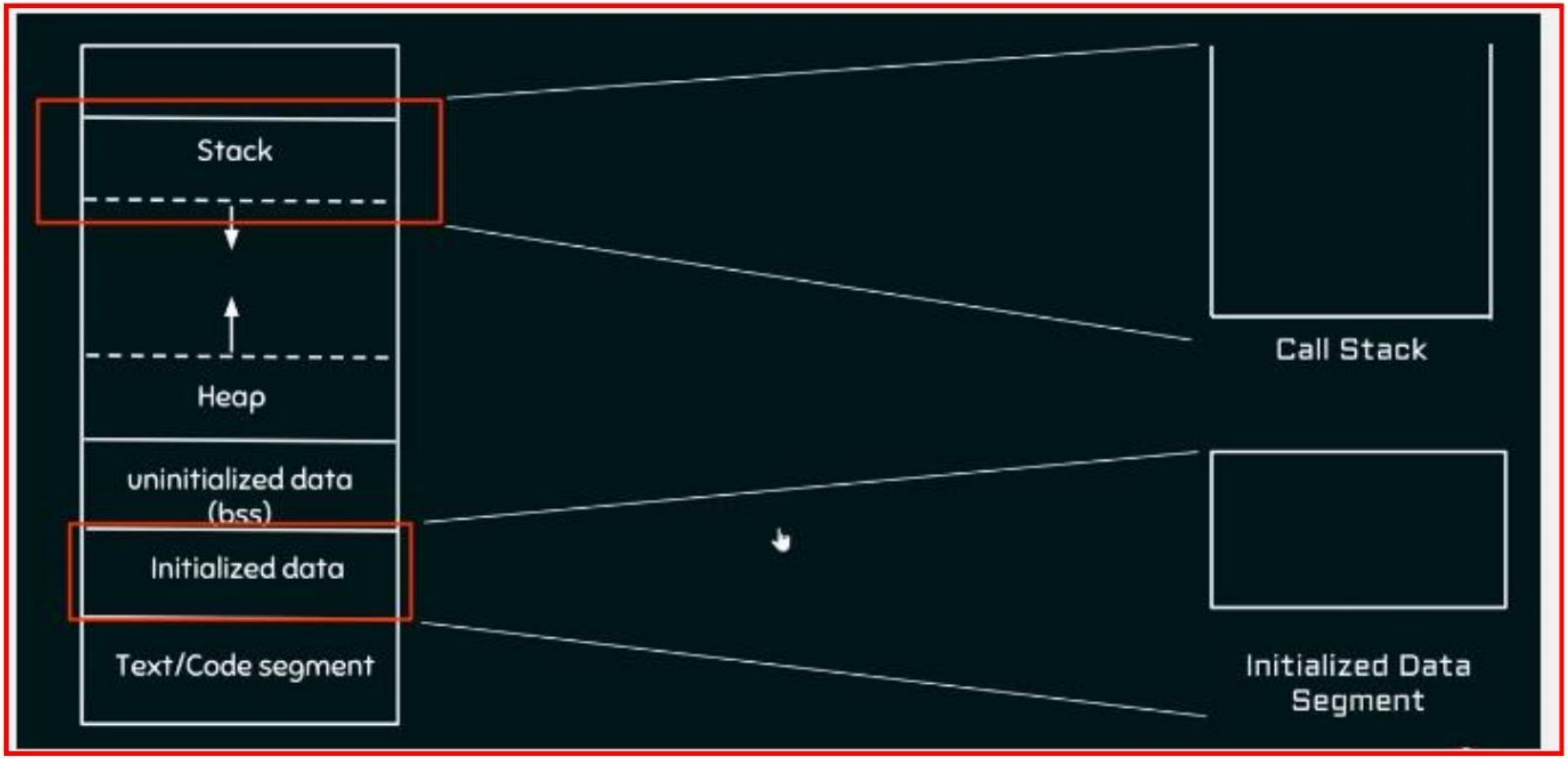
Third search for a and b

Second search for a and b

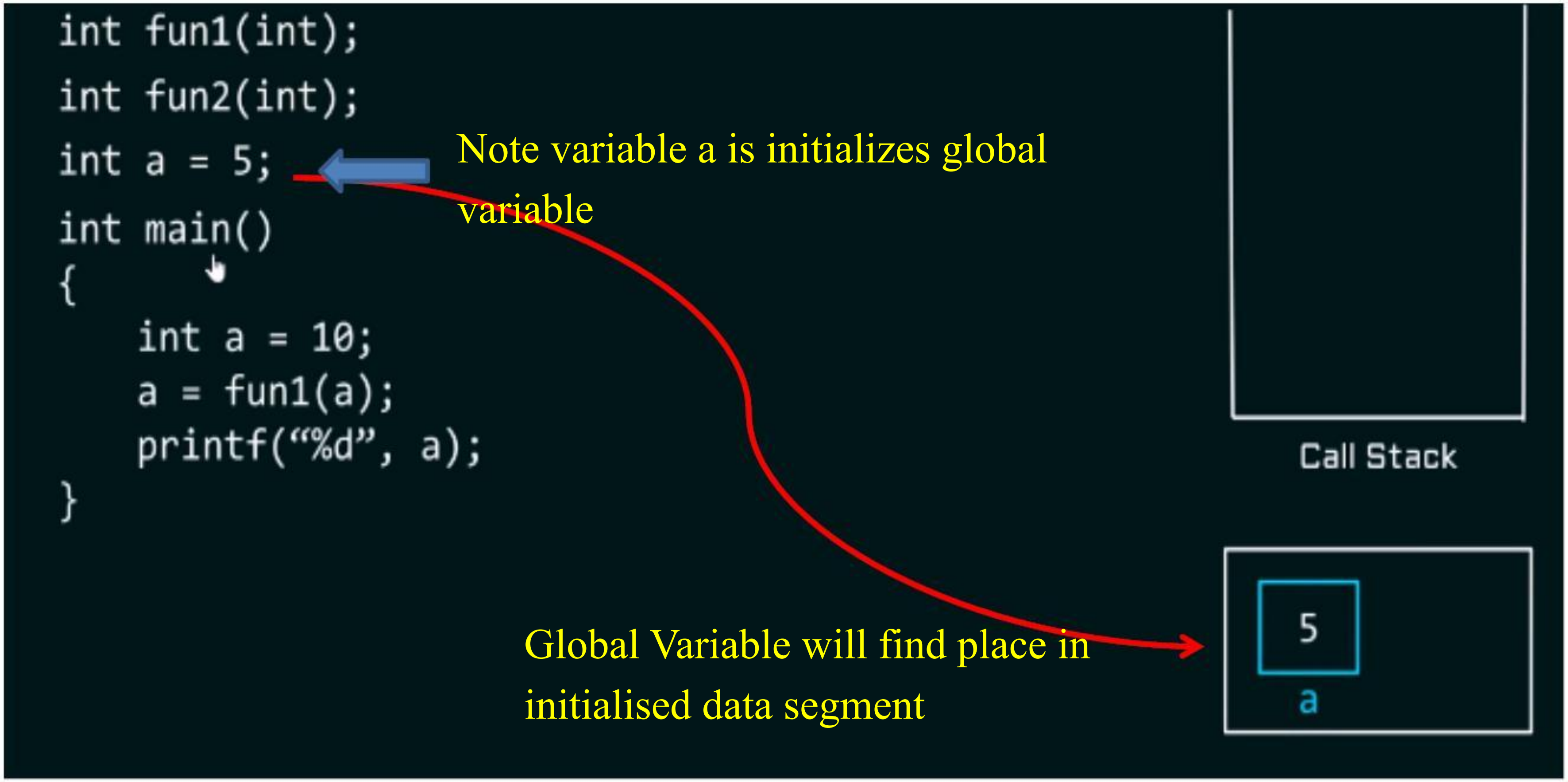
First search for a and b

Output:

c=4



#CprogrammingByNeso(Static and Dynamic Scoping –Part-1)

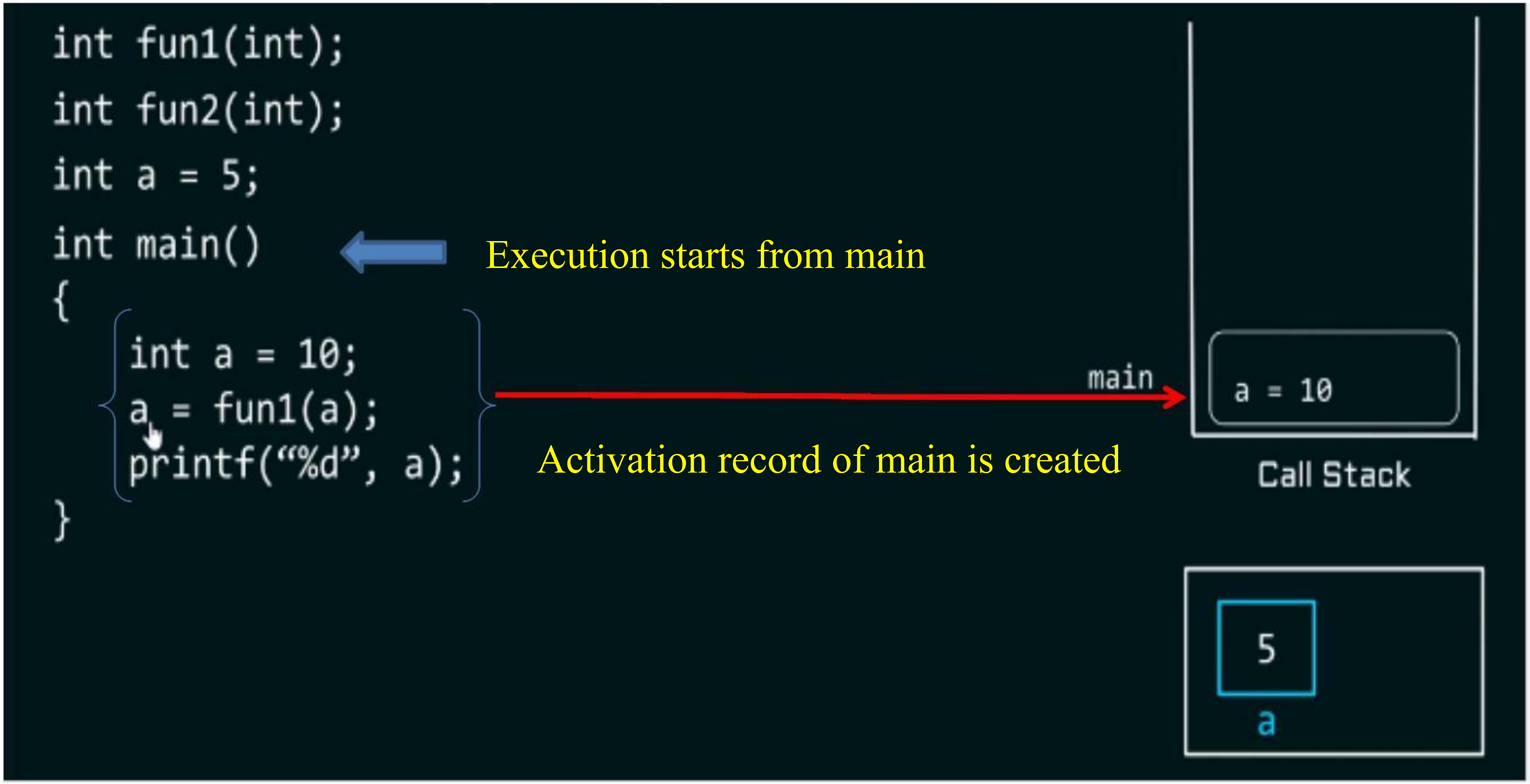


```
int fun1(int);
int fun2(int);
int a = 5;
int main()
{
    int a = 10;
    a = fun1(a);
    printf("%d", a);
}
```

Note variable a is initializes global variable

Global Variable will find place in initialised data segment

#CprogrammingByNeso(Static and Dynamic Scoping –Part-1)

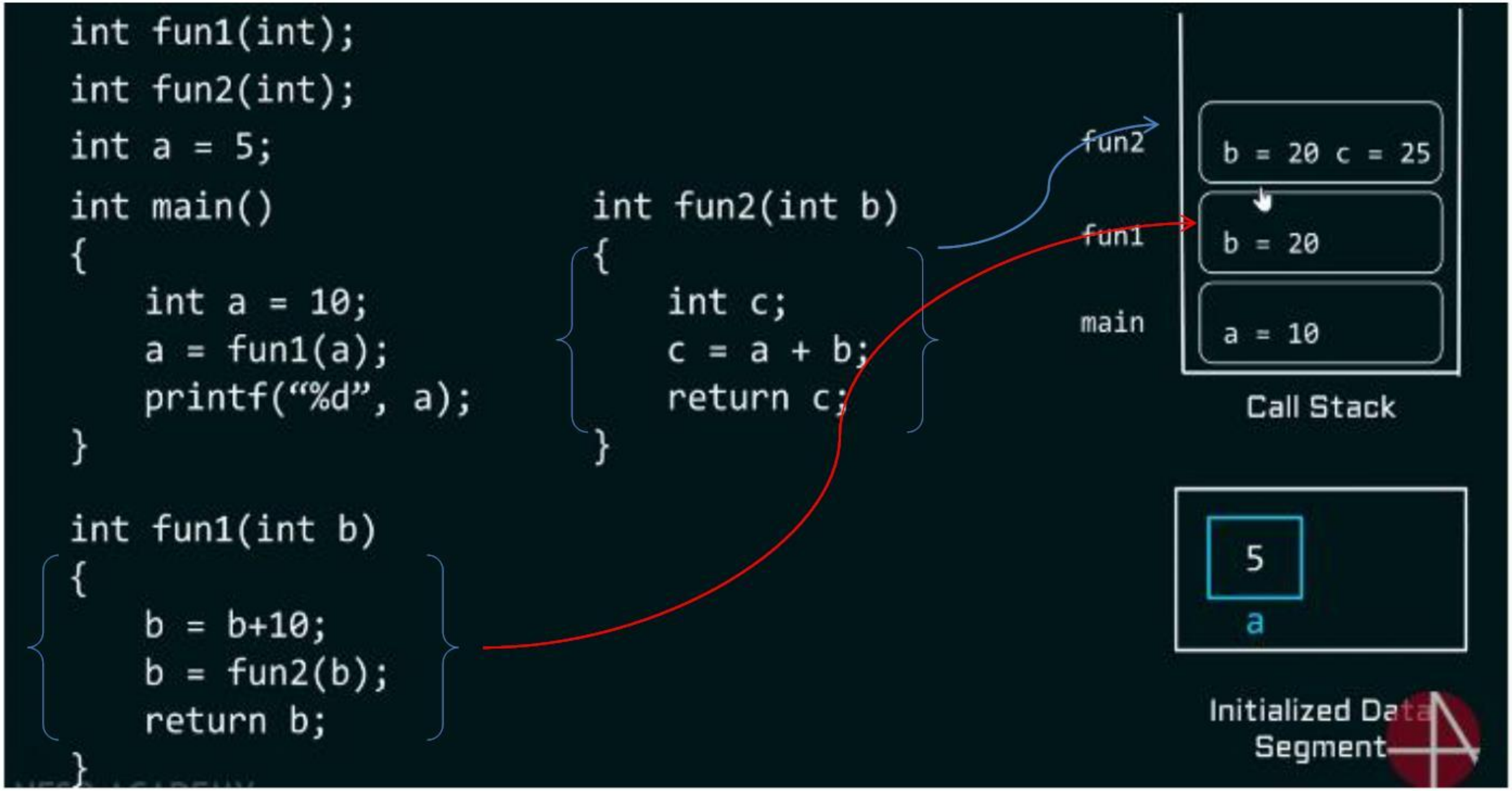


```
int fun1(int);
int fun2(int);
int a = 5;
int main()
{
    int a = 10;
    a = fun1(a);
    printf(“%d”, a);
}
```

Execution starts from main

Activation record of main is created

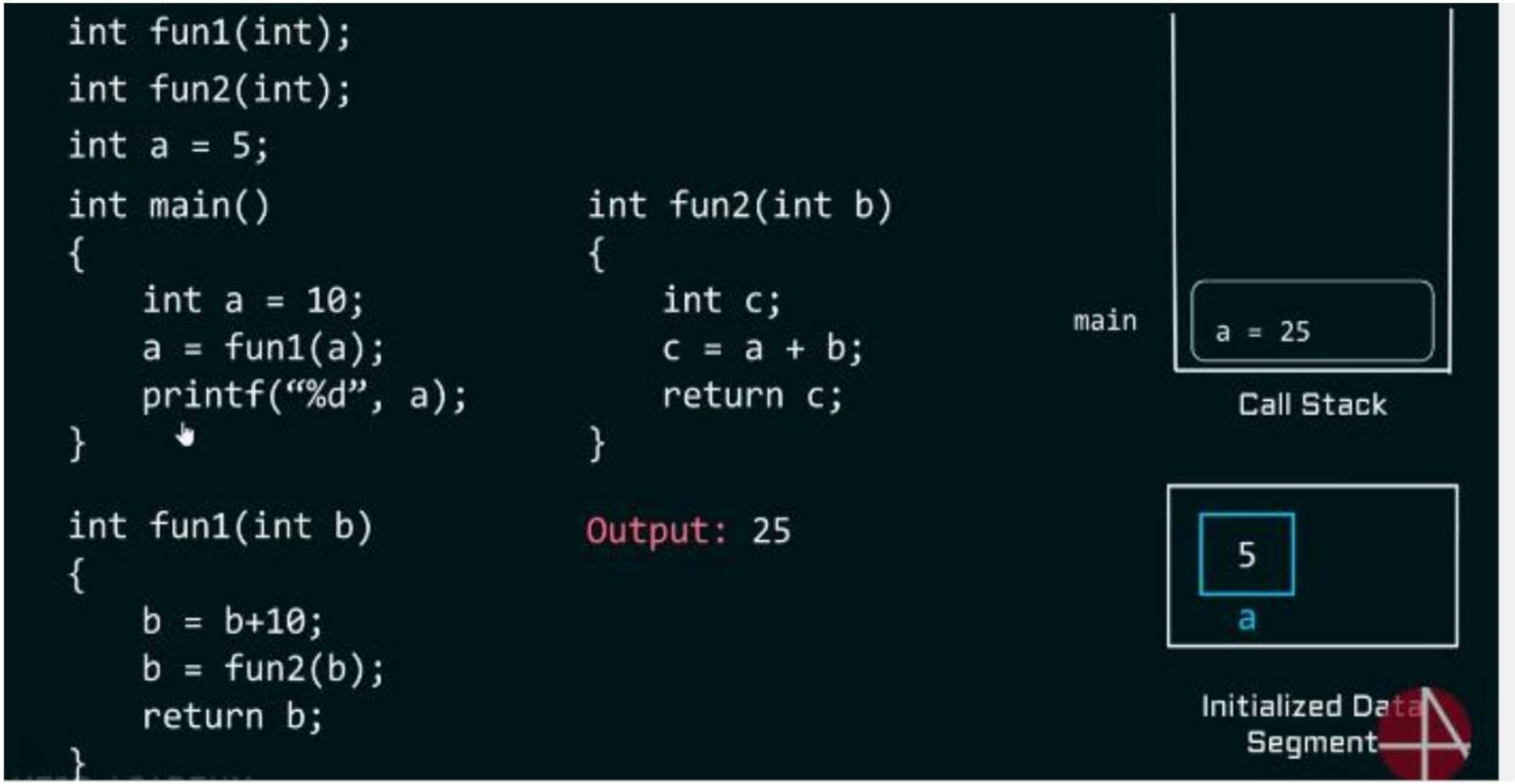
#CprogrammingByNeso(Static and Dynamic Scoping –Part-1)



```
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;
}
```

#C-programmingByNeso (Static and Dynamic Scoping –Part-1)



```
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;
}
```

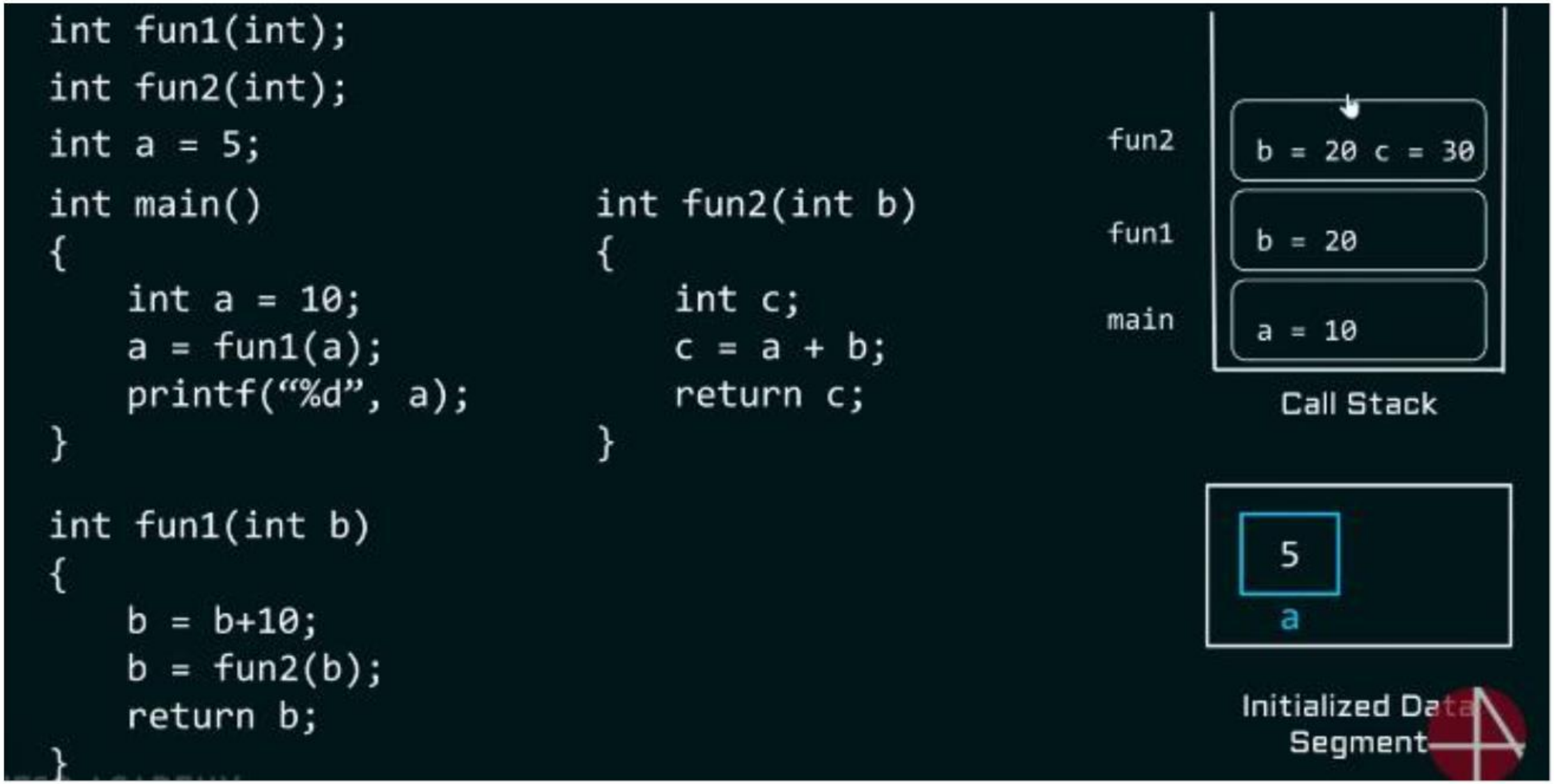
```
int fun2(int b)
{
    int c;
    c = a + b;
    return c;
}
```

```
Output: 25
```

#CprogrammingByNeso(Static and Dynamic Scoping –Part-1)

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;
}
```

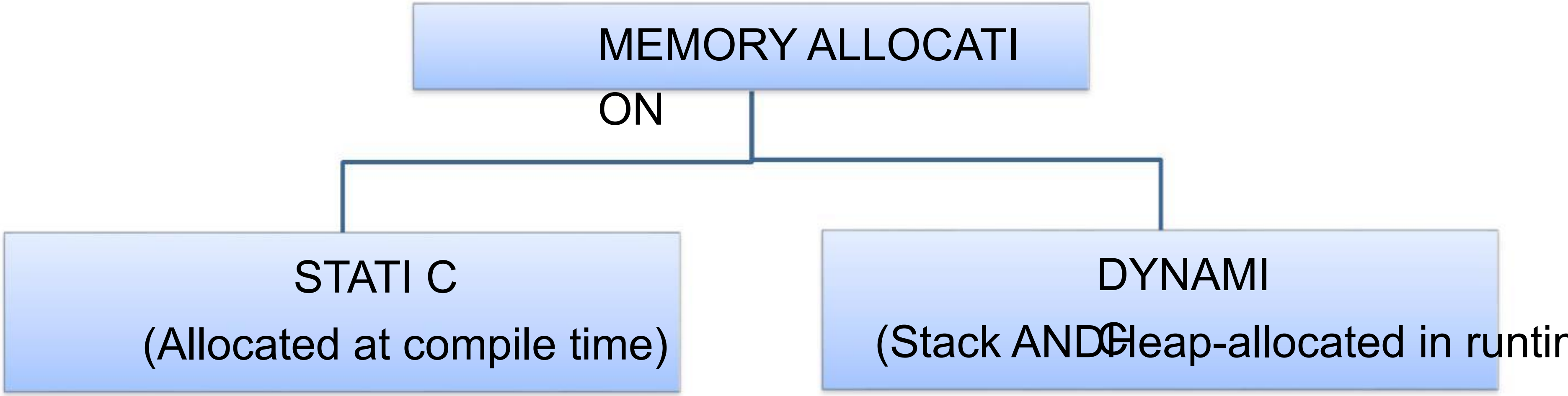
#CprogrammingByNeso(Static and Dynamic Scoping –Part-2)

LIFETIME AND STORAGE MANAGEMENT

. Binding Lifetime: The time between creation and destruction of name to object binding

. Object Lifetime: Time between creation and destruction of an object

. Dangling reference: Binding to object that no longer live



MEMORY ALLOCATI ON

STATI C

(Allocated at compile time)

DYNAMI C

(Stack AND Heap-allocated in runti

Static Allocation: Given absolute address retained throughout the program execution

. Memory allocated during compile time

. The memory is fixed and cannot be increased or decreased

```
 int main ()
{
int arr[5]=[1,2,3,4,5]
}
```

. If the values are less than the size specified, there will be wastage of memory

. If the values are more, then the program may crash or misbehave

Dynamic Allocation :The process of allocating memory at run time (at the time o execution)

. **Stack:** Allocated and de-allocated in LIFO order. Applicable for subroutine (call and return)

. **Heap**: Allocated and de-allocated at arbitrary times

STATIC ALLOCATION

. Objects whose values should not change during program execution

. Statically allocated objects

. Global variables

. Instructions for machine language translation

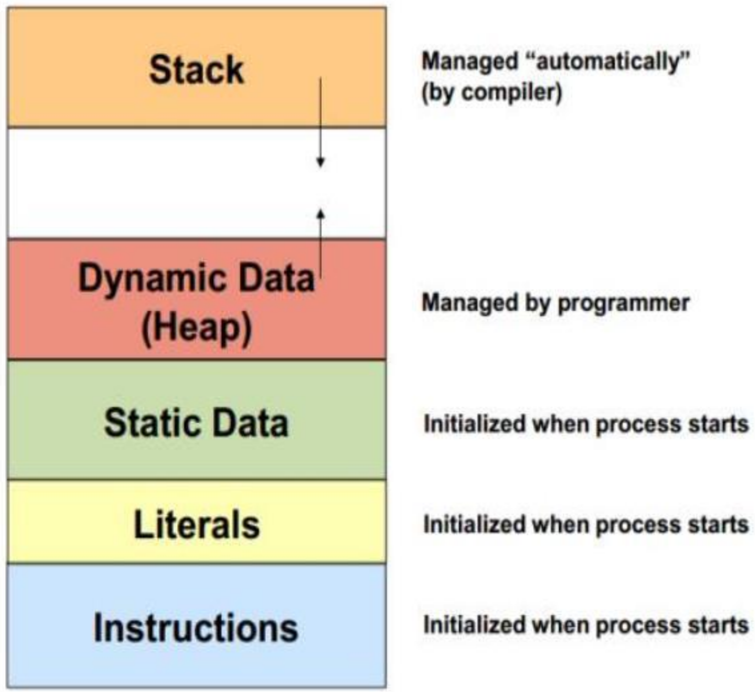
. Local variables that retain values b/w invocatio

. Numeric and string valued constant literals

. A=b/14.7
. print("hello world")

. Tables used at run-time routines

. (Debugging, dynamic type checking, garb


Stack
Managed "automatically" (by compiler)
Dynamic Data (Heap)
Managed by programmer
Static Data
Initialized when process starts
Literals
Initialized when process starts
Instructions
Initialized when process starts

STACK ALLOCATION

. For recursion static allocation is not possible.

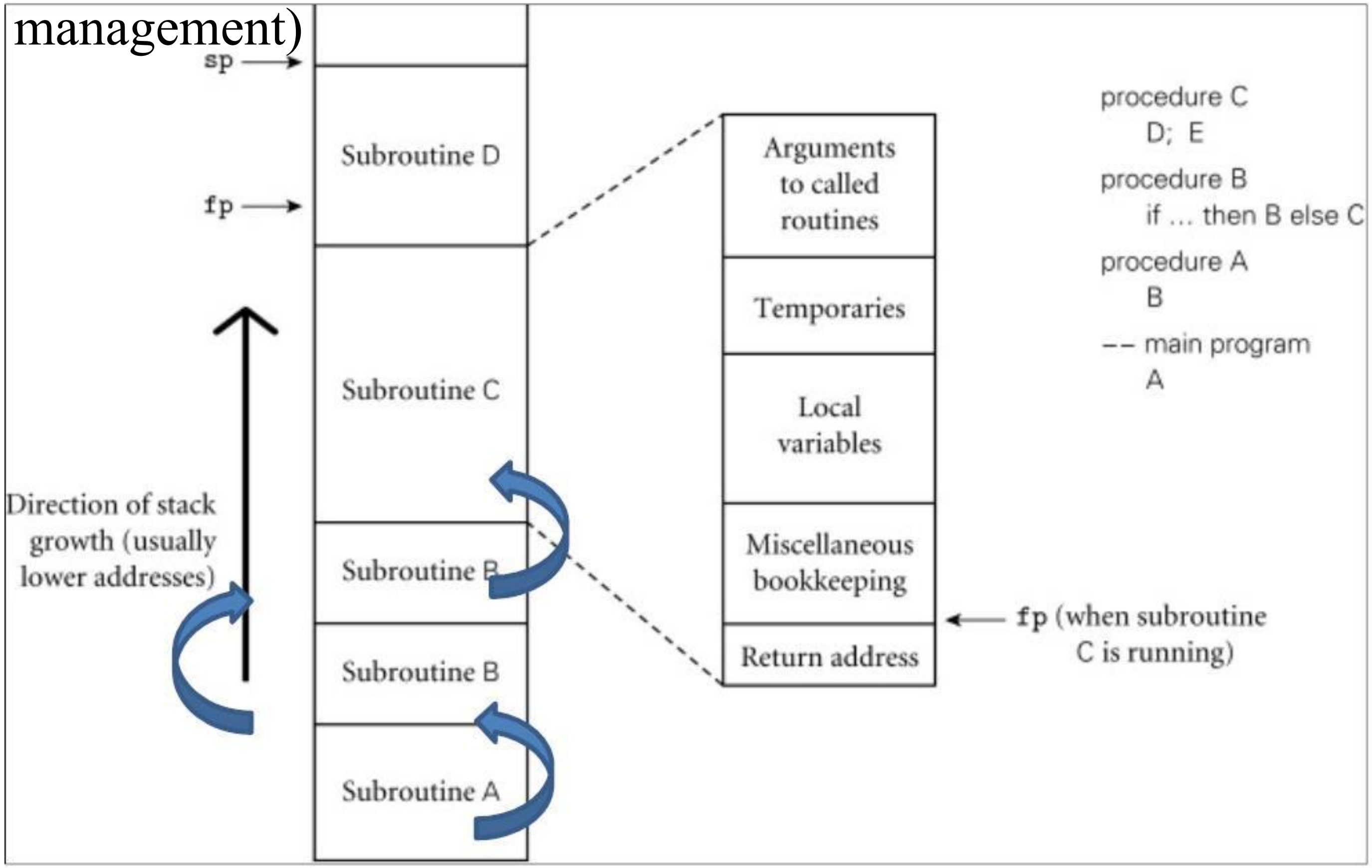
  . Number of instances of a variable is unbounded

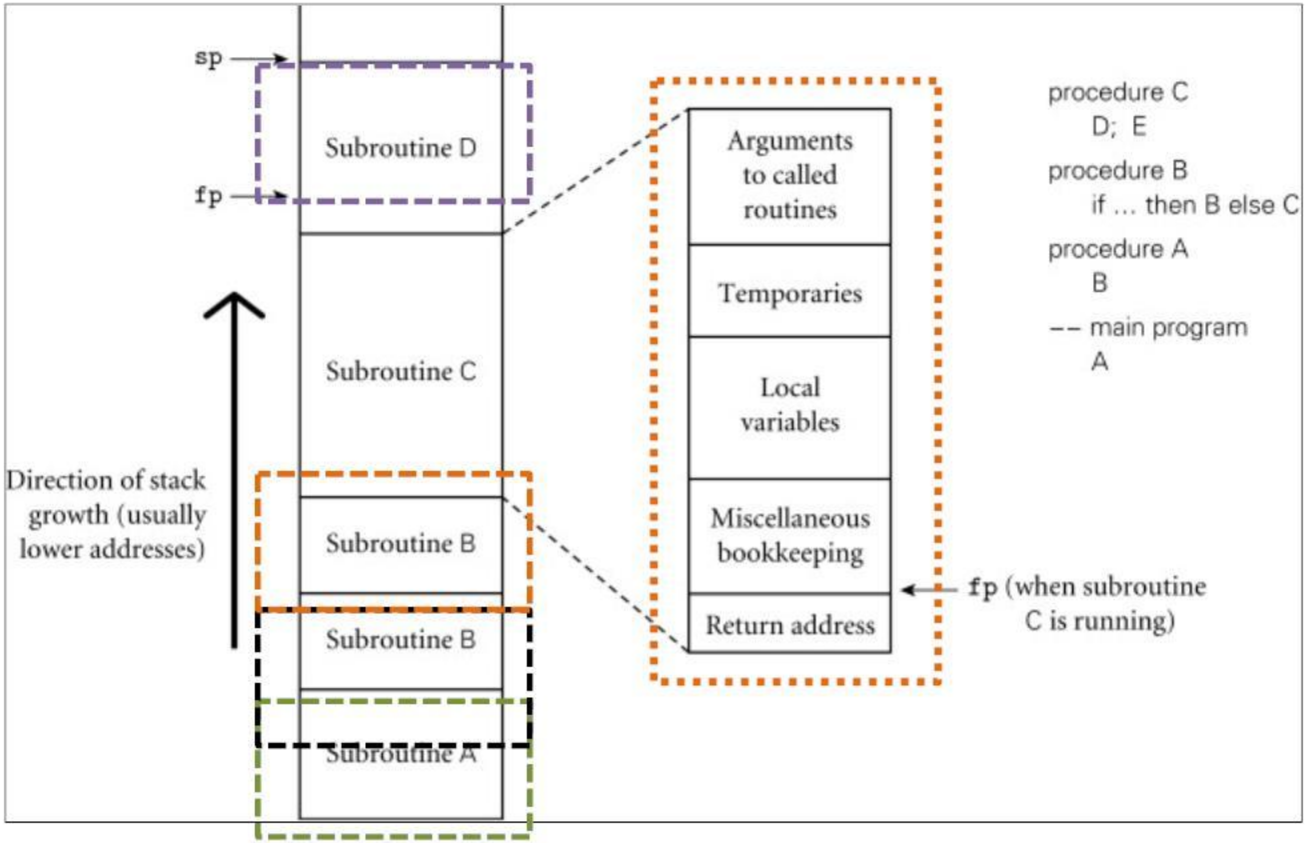
. Stack allocation is the solution

. Stack allocation allows us to

  . allocate space for recursive routines

. Reuse (space management)




sp
fp
Subroutine D
Subroutine C
Subroutine B
Subroutine B
Subroutine A
Direction of stack growth (usually lower addresses)
Arguments to called routines
Temporaries
Local variables
Miscellaneous bookkeeping
Return address
fp (when subroutine C is running)

```
procedure C
    D;  E
procedure B
    if ... then B else C
procedure A
    B
-- main program
    A
```

The stack frame will have the following contents-

1. Arguments
2. Local variables
3. Temporary variables
4. Return addresses
5. Miscellaneous bookkeeping

Who is responsible to take care of this??

The maintenance of the stack is done by subroutine calling sequence, prologue and epilogue

Calling Sequence: Code executed by the caller immediately before and after the call

Prologue: Code executed at the beginning

. Allocates a frame by subtracting frame size from sp

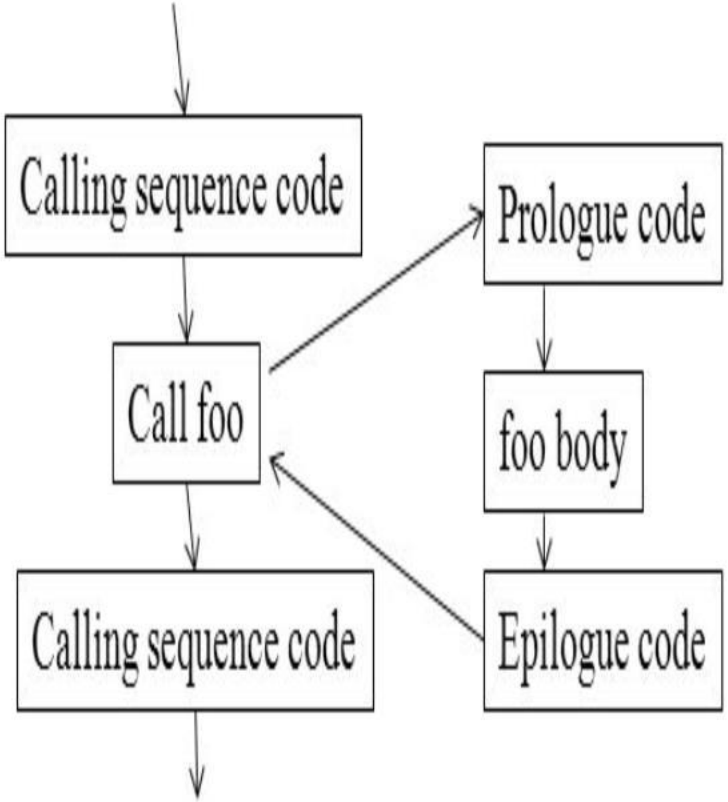
. Saves callee-saves registers used anywhere inside callee

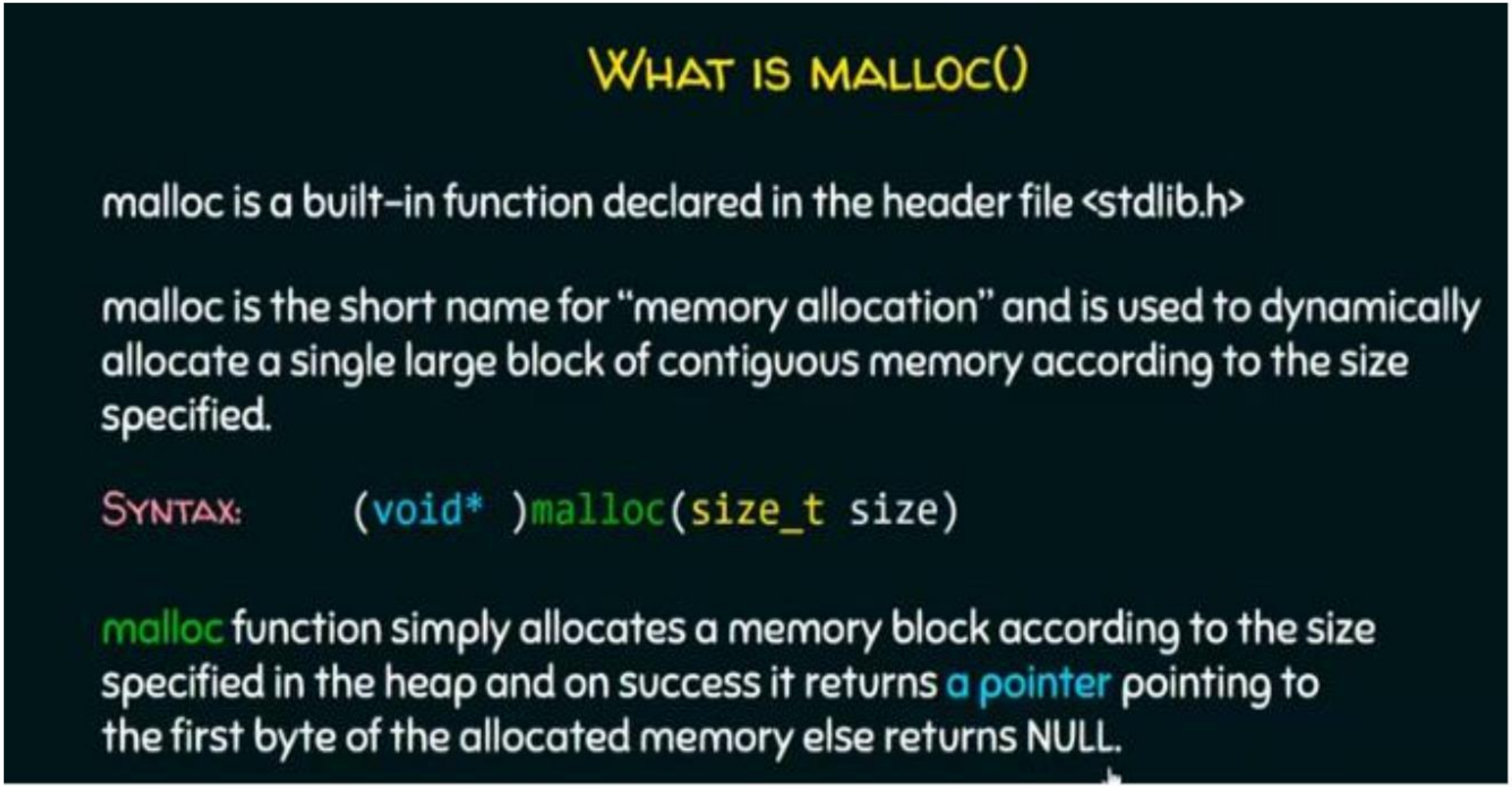
Epilogue: Code executed at the end

. Puts return values into registers

. Restores saved registers using sp as base

. Adds sp to de-allocate frame





WHAT IS MALLOC()

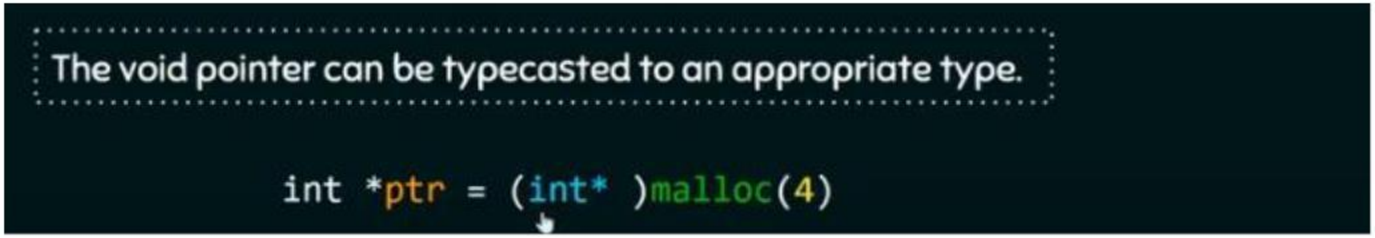
malloc is a built-in function declared in the header file <stdlib.h>

malloc is the short name for "memory allocation" and is used to dynamically allocate a single large block of contiguous memory according to the size specified.

SYNTAX:

```
(void* )malloc(size_t size)
```

malloc function simply allocates a memory block according to the size specified in the heap and on success it returns a pointer pointing to the first byte of the allocated memory else returns NULL.



The void pointer can be typecasted to an appropriate type.

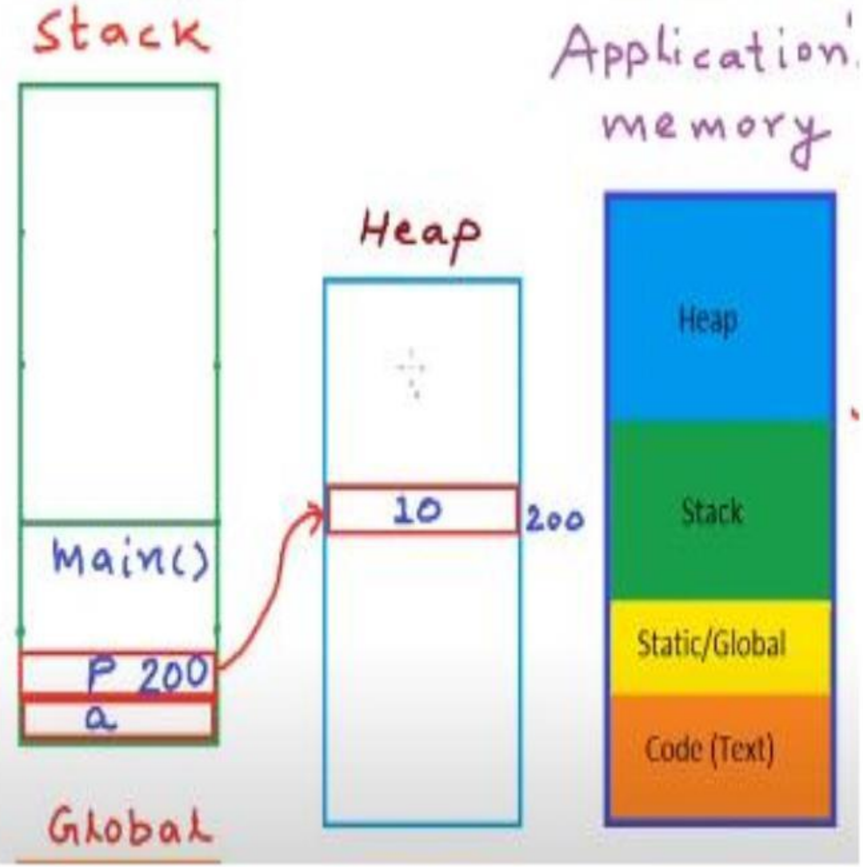
```
int *ptr = (int* )malloc(4)
```

HEAP

. Heap is a region of memory (storage) in which sub-blocks can be allocated and de-allocated at arbitrary time.

. Required for dynamic allocation of memory (at run time)

```
#include<stdio.h>
#include<stdio.h>
int main()
{
int a; //goes on stack
int *p;
p=(int*)malloc(sizeof(int));
*p=10;
```



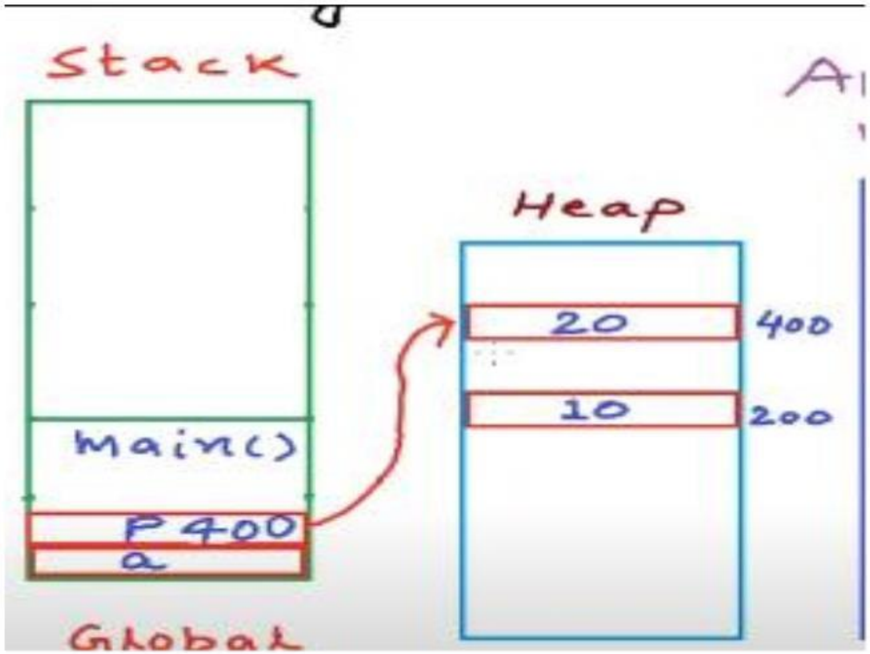
malloc()

calloc()

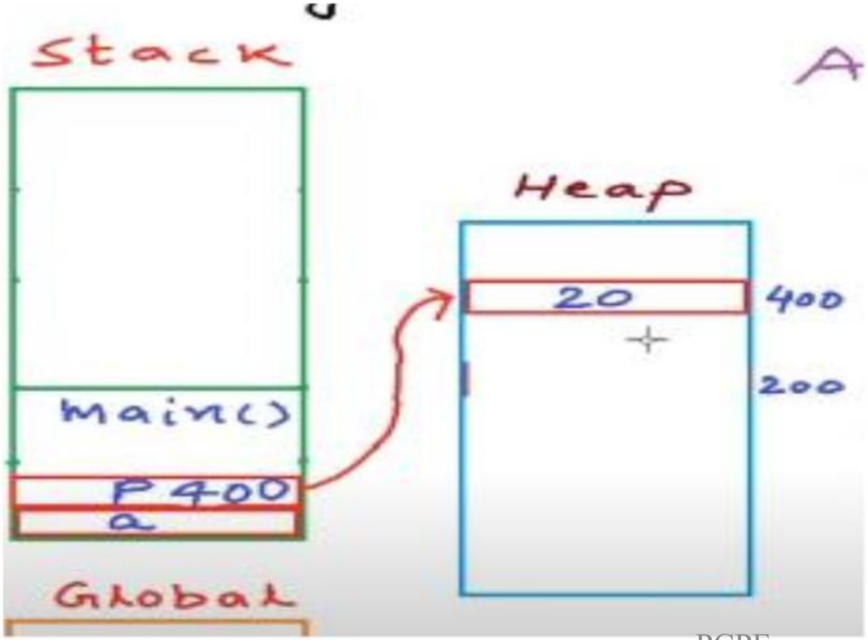
realloc()

free()

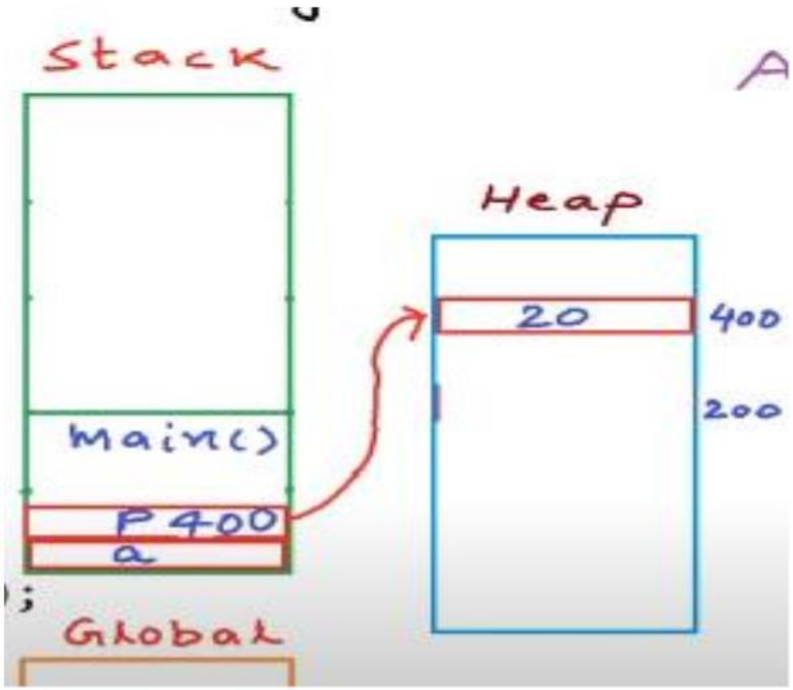
```
#include<stdio.h>
#include<stdio.h>
int main()
{
int a; //goes on stack
int *p;
p=(int*)malloc(sizeof(int));
*p=10;
p=(int*)malloc(sizeof(int));
*p=20;
```



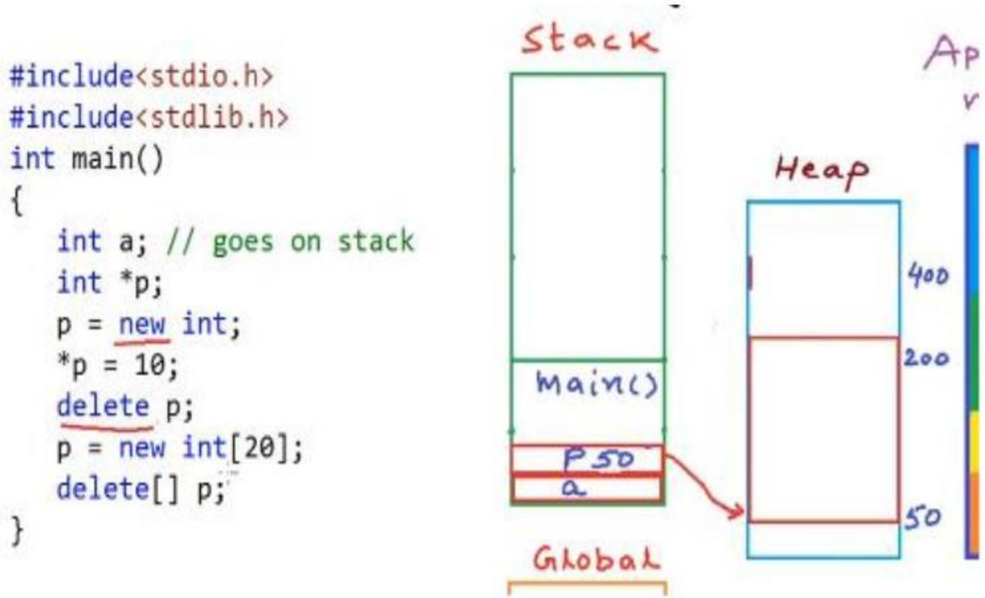
```
#include<stdio.h>
#include<stdio.h>
int main()
{
int a; //goes on stack
int *p;
p=(int*)malloc(sizeof(int));
*p=10;
free(p)
p=(int*)malloc(sizeof(int));
*p=20;
```



```
#include<stdio.h>
#include<stdio.h>
int main()
{
int a; //goes on stack
int *p;
p=(int*)malloc(sizeof(int));
*p=10;
free(p);
p=(int*) malloc (20*sizeof(int));
*p=20;
```



```
#include<stdio.h>
#include<stdio.h>
int main()
{
int a; //goes on stack
int *p;
*p=10;
delete p;
p=new int [20];
delete[] p;
```



```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int a; // goes on stack
    int *p;
    p = new int;
    *p = 10;
    delete p;
    p = new int[20];
    delete[] p;
}
```

References

1. Michael L Scott, " Programming Language Pragmatics", Third edition, Elsevier publication (Chapter-3)
2. Ravi Sethi, " Programming Languages-concepts and constructs", Pearson Education (Chapter-3,4,5)

Web Resources

1. NPTEL Online Video resources- Lecture-02, Lecture-03, Lecture-10

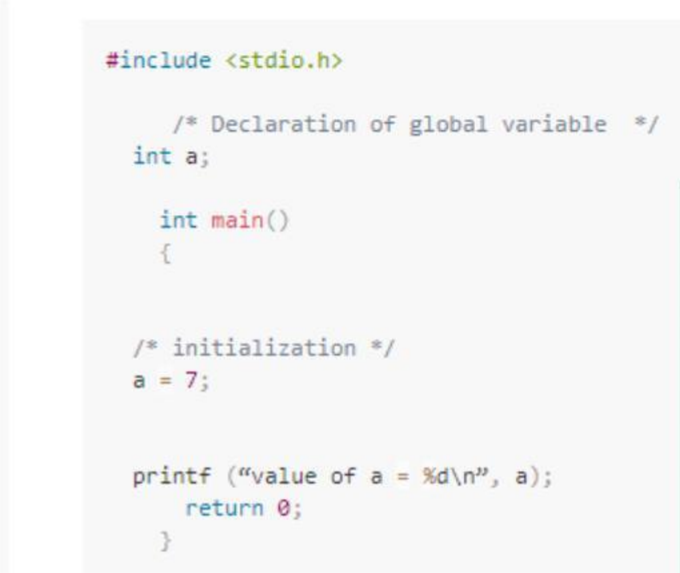
http://www.nptelvideos.in/2012/11/principles-of-programming-languages.html

2. Stanford University Online lectures- Lecture-02 and Lecture-03

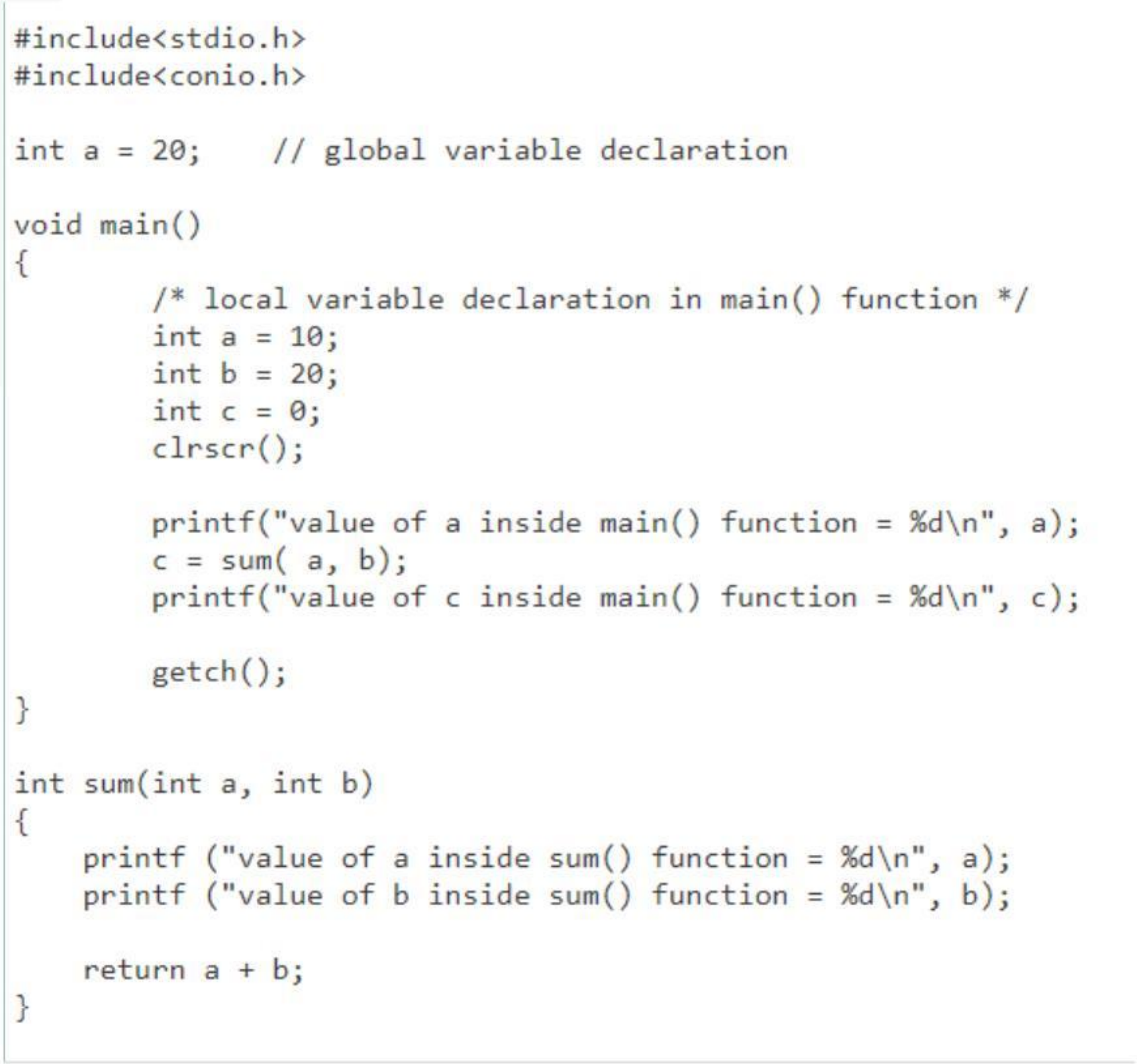
https://www.youtube.com/watch?v=Ps8jOj7diA0&list=PL9D558D49CA734A02

3. Neso Academy- Static and Dynamic Scoping (Part I and II)

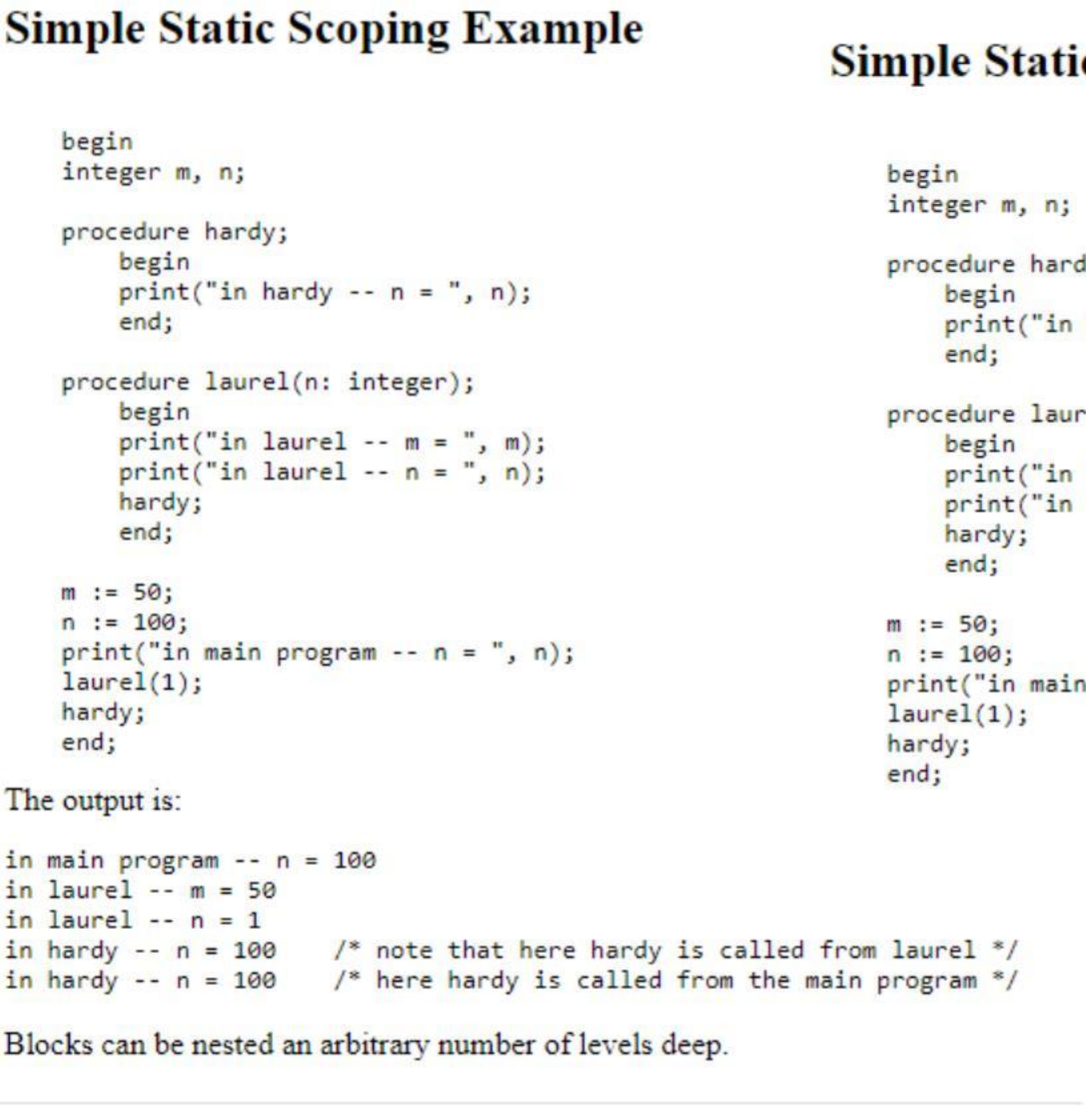
https://www.youtube.com/watch?v=L53nqHCSSFY&t=52s



```
#include <stdio.h>

     /* Declaration of global variable  */
  int a;

    int main()
    {


  /* initialization */
  a = 7;


  printf (“value of a = %d\n”, a);
      return 0;
    }
```



```
#include<stdio.h>
#include<conio.h>

int a = 20;    // global variable declaration

void main()
{
        /* local variable declaration in main() function */
        int a = 10;
        int b = 20;
        int c = 0;
        clrscr();

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

        getch();
}

int sum(int a, int b)
{
    printf ("value of a inside sum() function = %d\n", a);
    printf ("value of b inside sum() function = %d\n", b);

    return a + b;
}
```



Simple Static Scoping Example

```
begin
integer m, n;

procedure hardy;
    begin
    print("in hardy -- n = ", n);
    end;

procedure laurel(n: integer);
    begin
    print("in laurel -- m = ", m);
    print("in laurel -- n = ", n);
    hardy;
    end;

m := 50;
n := 100;
print("in main program -- n = ", n);
laurel(1);
hardy;
end;
```

The output is:

```
in main program -- n = 100
in laurel -- m = 50
in laurel -- n = 1
in hardy -- n = 100    /* note that here hardy is called from laurel */
in hardy -- n = 100    /* here hardy is called from the main program */
```

Blocks can be nested an arbitrary number of levels deep.

Simple Static Scoping Example

```
begin
integer m, n;

procedure hardy;
    begin
    print("in hardy -- n = ", n);
    end;

procedure laurel(n: integer);
    begin
    print("in laurel -- m = ", m);
    print("in laurel -- n = ", n);
    hardy;
    end;

m := 50;
n := 100;
print("in main program -- n = ", n);
laurel(1);
hardy;
end;
```