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)

OUTLINE OF SYLLABUS

Module	Contents
1	Introduction to programming paradigms and core language design issues
2	Imperative Paradigm: Data abstraction in object orientation
3	Declarative programming paradigm: Functional programming
4	Declarative programming paradigm: Logic programming
5	Alternative paradigm: Concurrency
6	Alternative paradigm: Scripting Languages

OUTLINE OF UNIT-1

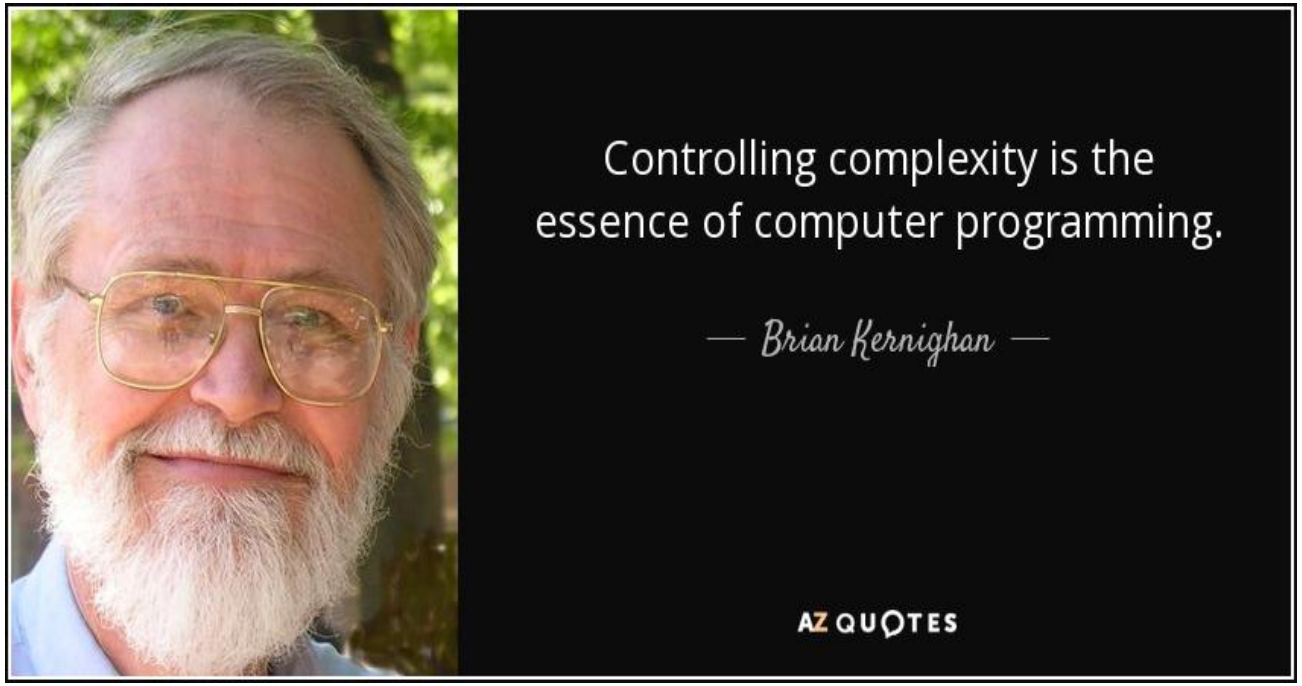
Sub-Unit	Contents
1.1	Introduction to different programming paradigms
1.2	Names, Scopes, Bindings, Scope Rules, Storage Management
1.3	Type Systems, Type checking, Equality testing, and assignment
1.4	Subroutine and control abstraction, Stack layout, calling sequence, parameter passing
1.5	Generic subroutines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

1.1-INTRODUCTION TO DIFFERENT PROGRAMMING PARADIGMS

Any fool can write code that a computer can understand. Good programmers write code that human's can understand

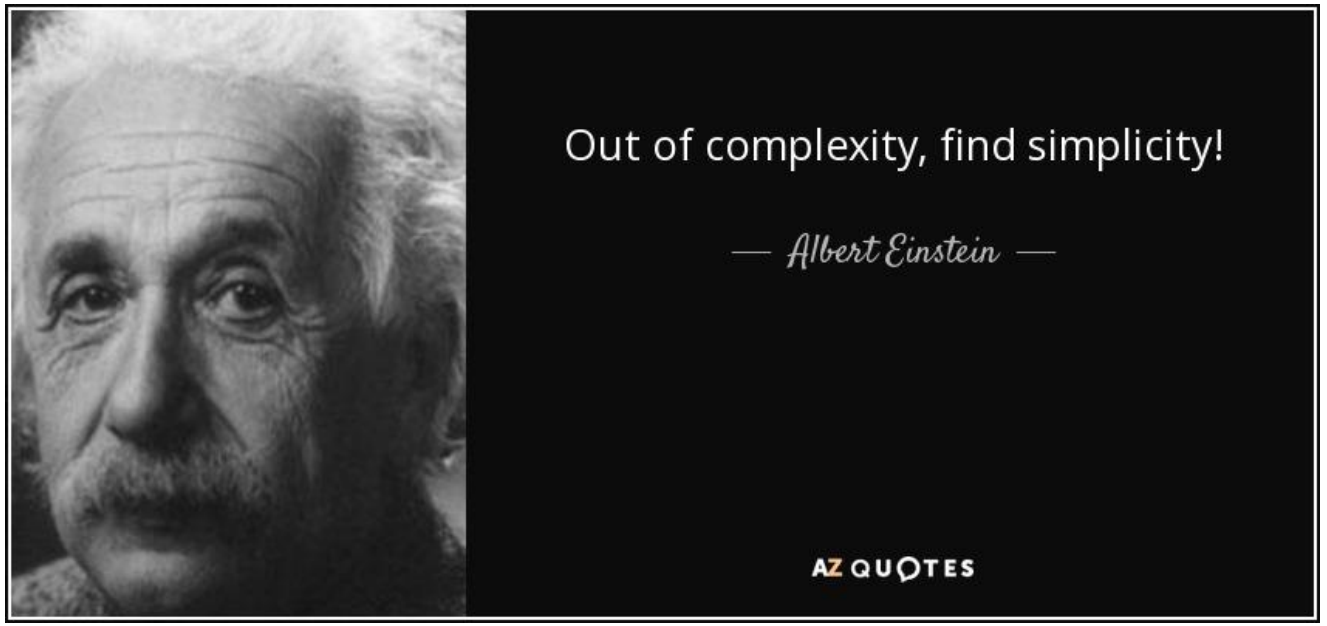
-Martin Fowler

(Martin Fowler is a British software developer, author and international public speaker on software development, specializing in object-oriented analysis and design, UML, patterns)



1. When programming , complexity is always the enemy

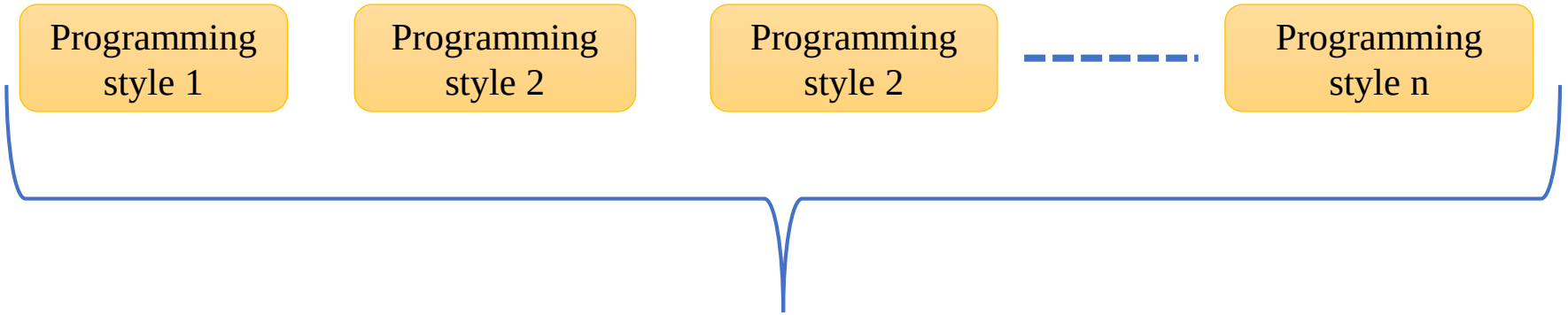
2. Managing complexity is a programmer's main concern



Choosing the right programming paradigms

What is Programming Paradigm?

Definition: A programming paradigm is a style or a "way" of programming. Some languages make it easy to write programs in some paradigms but others do not.



You have different "styles"/ "ways"/ **"Paradigms"** of programming to tackle the issues of handling complexity

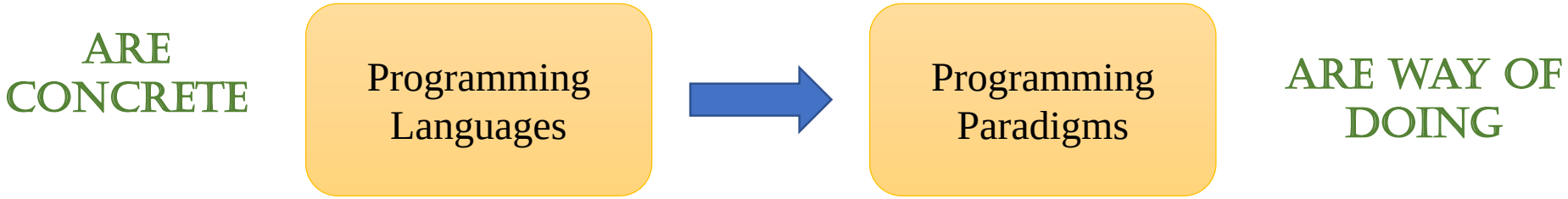
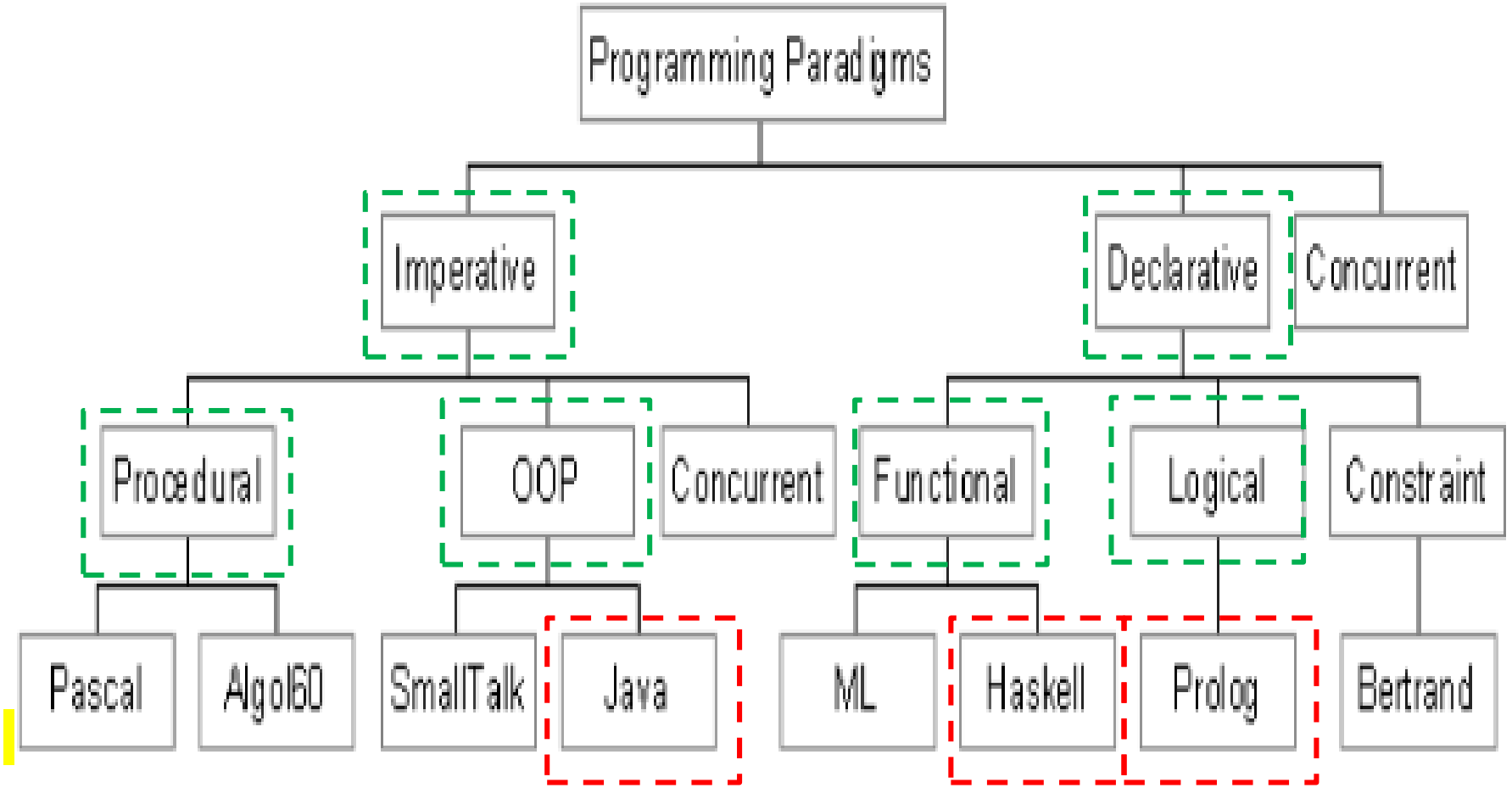


Diagram of Programming Paradigms



Features of Good Programming Language

- Clear, simple
- More expressive
- Orthogonal
- Support of abstraction
- Ease of implementation
- Ease of program verification
- Programming environments
- Portability of programs
- Naturalness for the application
- Low cost of implementation and usage

Why So Many Languages?

- Evolution.
- Special purposes /Application Specific
- Personal preference

Language Groups

- Multi-purpose languages
  - Scala, C#, Java, C++, C
  - Haskell, SML, Scheme, LISP
  - Perl, Python, Ruby
- Special-purpose languages
  - UNIX shell
  - SQL
  - LATEX

History of PLs

- 1951–55: Experimental use of expression compilers.
- 1956–60: **FORTRAN**, **COBOL**, **LISP**, **Algol 60**.
- 1961–65: APL notation, Algol 60 (revised), **SNOBOL**, CPL.
- 1966–70: APL, SNOBOL 4, FORTRAN 66, **BASIC**, **SIMULA**, Algol 68, Algol-W, **BCPL**.
- 1971–75: Pascal, PL/1 (Standard), **C**, **Scheme**, **Prolog**.
- 1976–80: **Smalltalk**, Ada, FORTRAN 77, **ML**.
- 1981–85: Smalltalk-80, **Prolog**, Ada 83.
- 1986–90: **C++,** SML, **Haskell**.
- 1991–95: Ada 95, TCL, Perl.
- 1996–2000: **Java**.
- 2000–05: C#, **Python**, Ruby, Scala.

IMPERATIVE PROGRAMMING

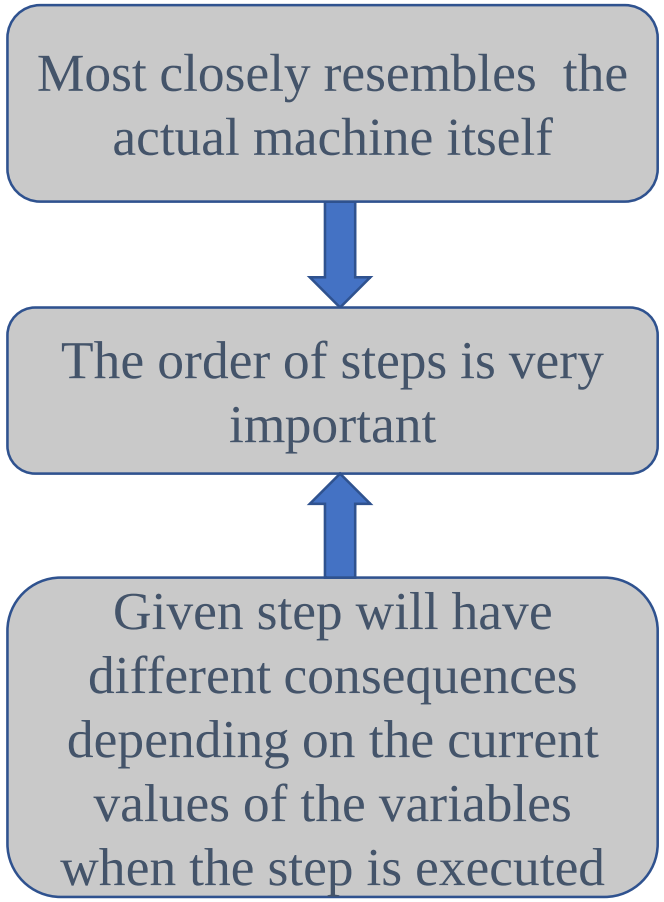
#(first do this then do that…….)

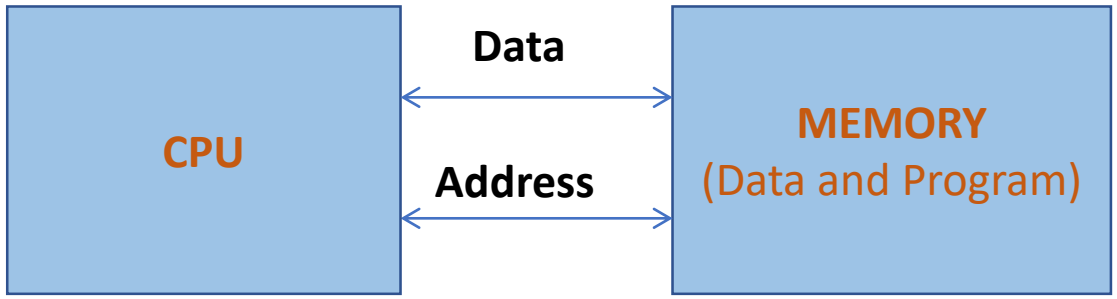
- The imperative paradigm is the oldest and the most popular programming paradigm
- Based on the von Neumann architecture of computers (https://www.javatpoint.com/von-neumann-model)
- Imperative programs define sequences of commands/statements for the computer that change a program state (i.e., set of variables)
  - Commands are stored in memory and executed in the order found
  - Commands retrieve data, perform a computation, and assign the result to a memory location
- The hardware implementation of almost all computers is imperative
- Machine code which is naïve to the computer hardware is written in imperative style

IMPERATIVE PROGRAMMING

Program: Sum of first 5 natural numbers in C

```
#include<stdio.h>
int main( )
{
int sum=0;
sum+=1;
sum+=2;
sum+=3;
sum+=4;
sum+=5;
printf("The sum is: %d/n", sum);
return 0;
}
```





Are there any CENTRAL elements of imperative paradigms

YES!!! YES!!!!

Assignment Statement

- The general syntax:

  <target_var> <assign_operator> <expression>

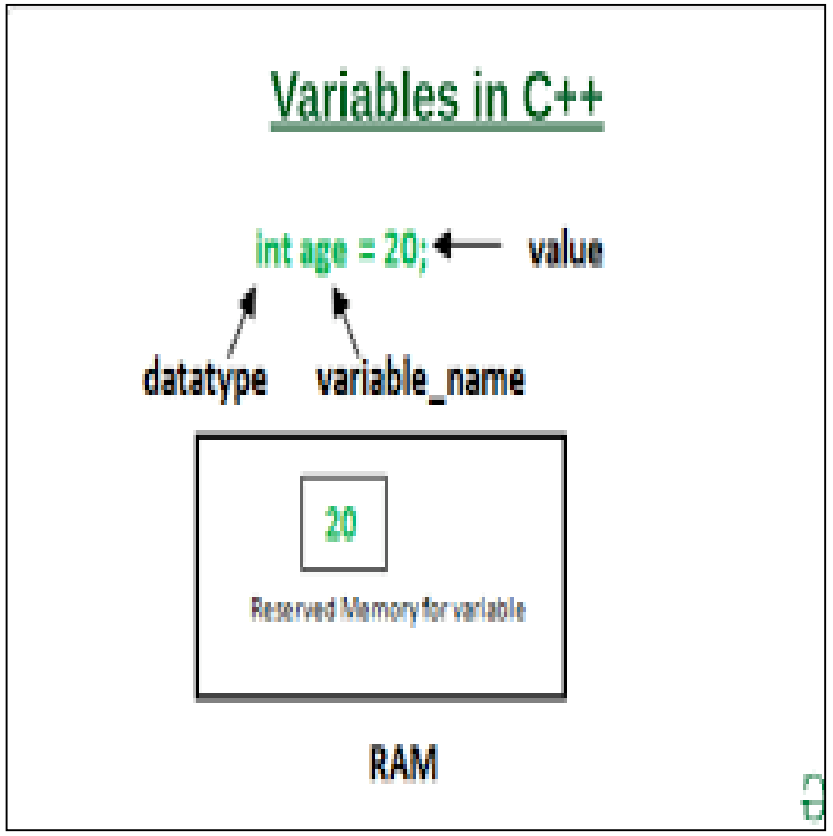
- The assignment operator:

  = FORTRAN, BASIC, PL/I, C, C++, Java.

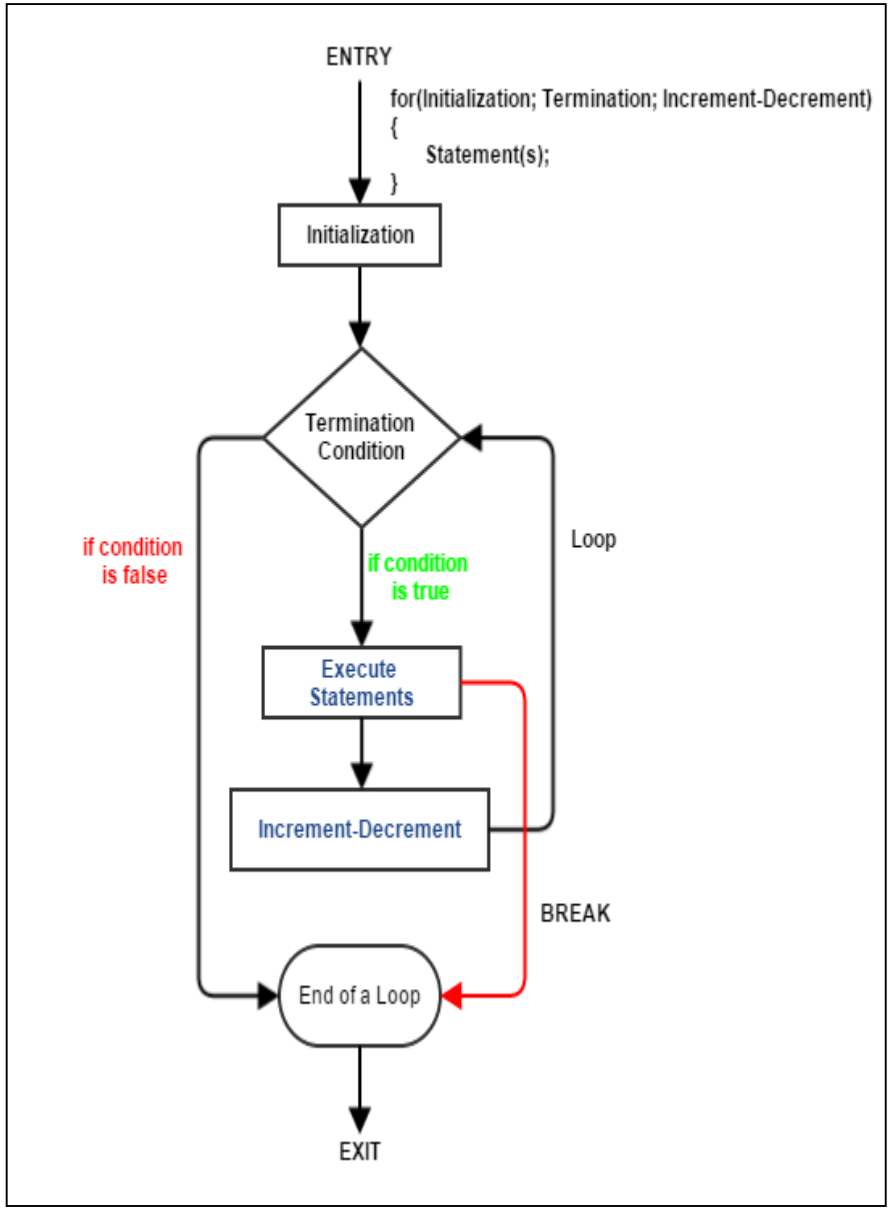
  := ALGOLs, Pascal, Ada.

- Simple assignment:
  - `a = b;`
  - `a = b = c;`
    - Suppose **a**, **b**, and **c** are integers.
    - In C, the integer value of **c** is assigned to **b**, which is in turn assigned to **a** (**multiple targets**).

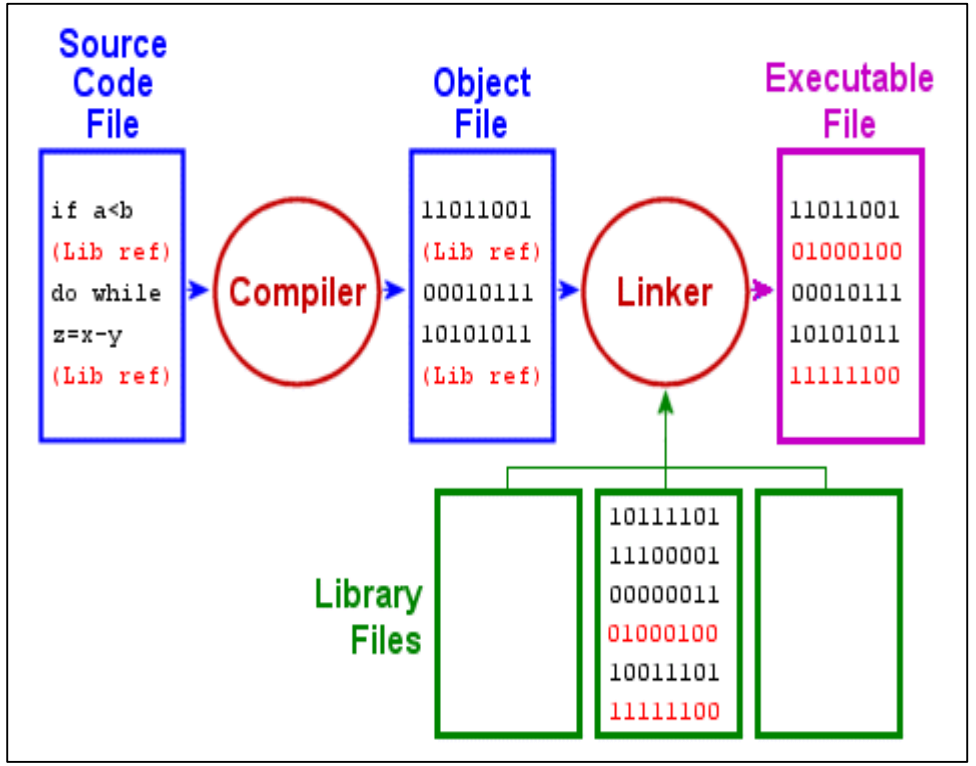
Variables



Flow control



Step by step execution



Examples of imperative programming language (languages that follow imperative programming paradigm)

C: Developed by Dennis Ritchie and ken Thompson

FORTAN: Developed by John Backus from IBM

Basic: Developed by John Kemeny and Thomas E Kurtz

PROCEDURAL PROGRAMMING
(for repetitive job)

- Procedural programming is a **refinement** of the imperative paradigm adding subroutines (or procedures)
- Procedures can be used the same way that built-in commands are used (allows re-usability)

?? What is a procedure

- Set of subroutines
- May or may not return a value

In a program for drawing shapes, the program could ask the user what shape to draw. The instructions for drawing a square could be captured in a **procedure**.

The algorithm for this action could be a set of tasks, such as these:

Repeat the next two steps four times:

Draw a line of length n.

Turn right by 90 degrees.

If this were a computer program, this set of instructions could be given the name '*square*' and this sequence would be executed by **running** (calling) that **procedure**.

Example of computing the factorial of a number:

IMPERATIVE

```
unsigned int n = 5;
unsigned int result = 1;
while(n > 1) {
 result *= n;
n--;
}
```

Introduce procedure, have return type

PROCEDURAL

Forming a procedure

```
int factorial(unsigned int n)
{
unsigned int result = 1;
 while(n > 1) {
result *= n;
 n--;
 }
return result;
 }
```

Returning a value

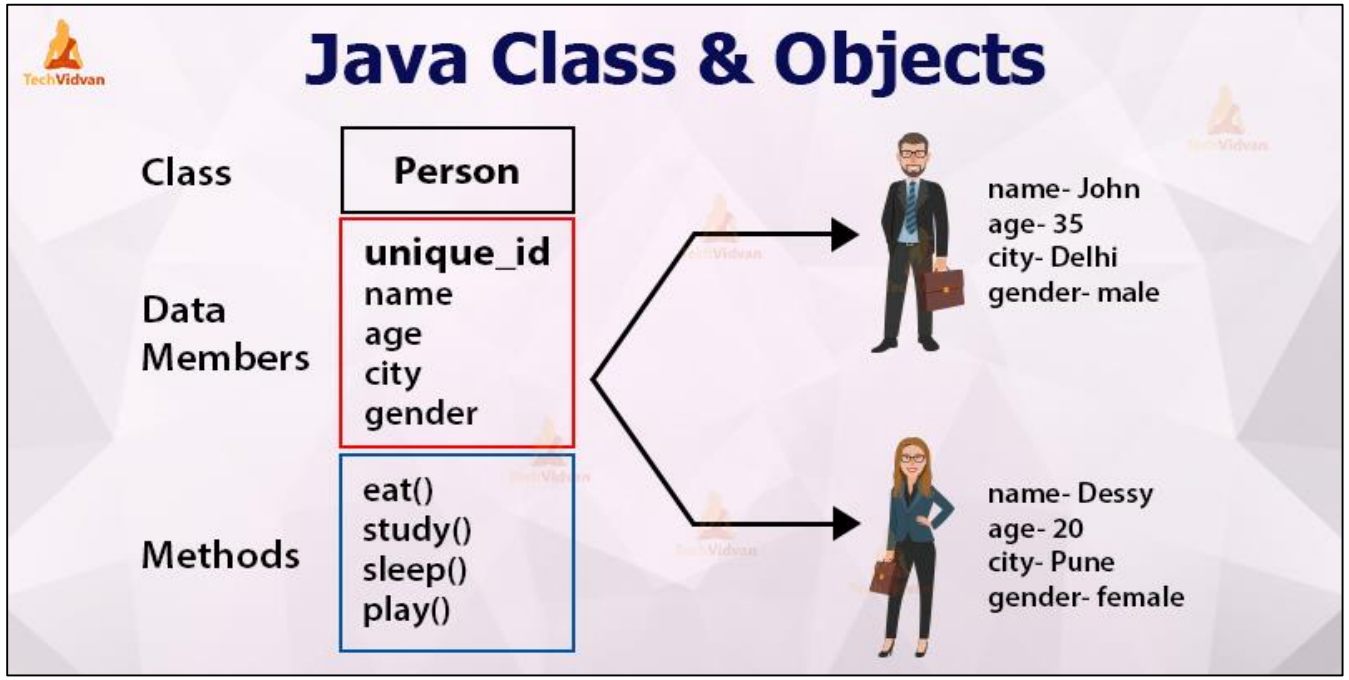
Procedure will be called from main/home

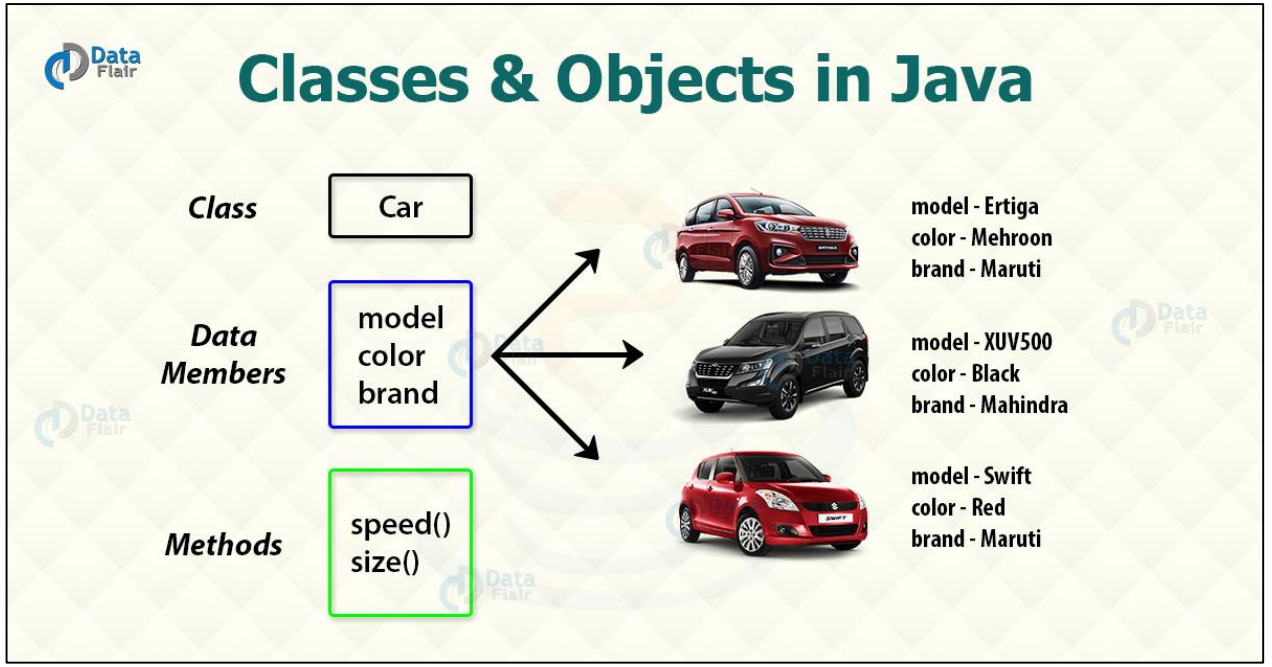
Examples of procedural programming language (languages that follow procedural programming paradigm)

C: Developed by Dennis Ritchie and ken Thompson
C++: Developed by Bjarne Stroustrup at Bell Labs
JAVA: Developed by James Gosling at Sun Microsystems
ColdFusion: Adobe, Joseph J. Allaire, Macromedia

OBJECT ORIENTED PROGRAMMING PARADIGM

- An approach to the solution of problems in which all computations are performed in the context of objects
- The program is written as a collection of classes and objects.
- The smallest and the basic entity is object
- Emphasis is on data rather than procedure
- Methods that operate on the data of an object is tied together in the data structure
- Data is hidden and cannot be accessed by external function
- Objects may communicate with each other through methods
- Follows bottom up approach in program design
- **Ruby, Java, C++, Python, Simula**


TechVidyan
Java Class & Objects
Class
Person
Data Members
unique_id
name
age
city
gender
Methods
eat()
study()
sleep()
play()
name- John
age- 35
city- Delhi
gender- male
name- Dessy
age- 20
city- Pune
gender- female


Data Flair
Classes & Objects in Java
Class
Car
Data Members
model
color
brand
Methods
speed()
size()
model - Ertiga
color - Mehroon
brand - Maruti
model - XUV500
color - Black
brand - Mahindra
model - Swift
color - Red
brand - Maruti

DECLARATIVE POGRAMMING PARADIGM

- The style expresses the logic of a computation without talking about its control flow
- It defines what needs to be accomplished by the program without defining how it needs to be implemented

IMPERATIVE

Mam……pls tell WHAT to do and HOW to do

DECLARATIVE

Mam……Only tell WHAT to do and **FORGET** how to do

IMPERATIVE: Provides instructions for assembly

DECLARATIVE: Provides a picture of finished piece as a template

The more sophisticated the application, the greater the danger that the code becomes so convoluted that it can only be read by the developer who originally wrote it

Examples of declarative programming paradigm

Prolog : Developed by by Alain Colmerauer at the University of Aix-Marseille, France

Haskell : First proposed by Philip Wadler and Stephen Blott

Miranda : Developed by David Turner Research Software Ltd.

Advantages:-

- Short, efficient code

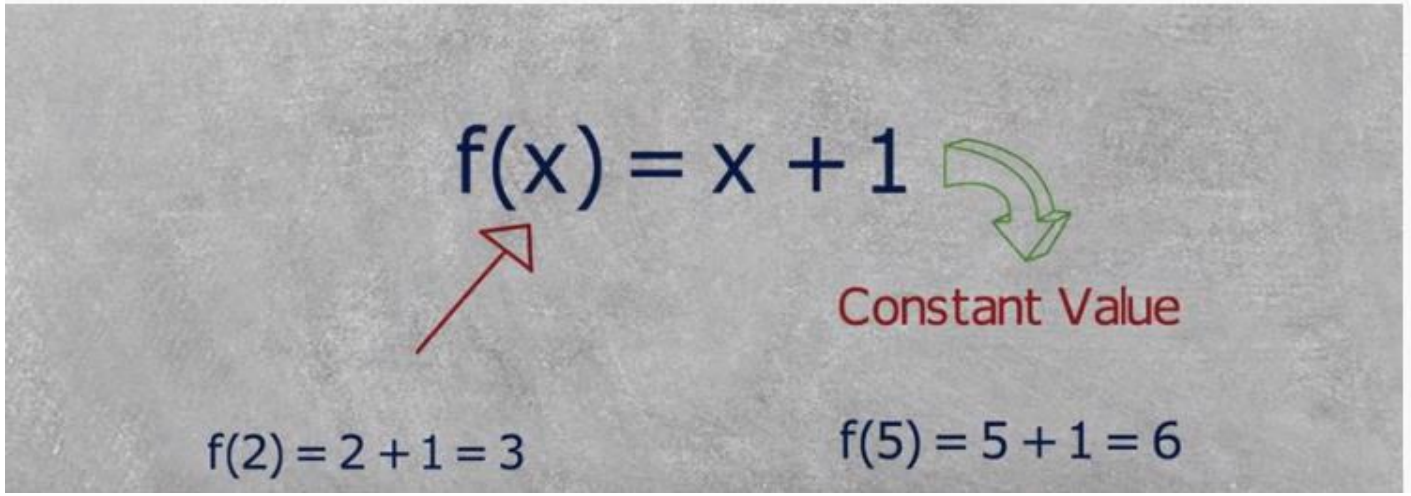
- Easy optimization as implementation is controlled by an algorithm

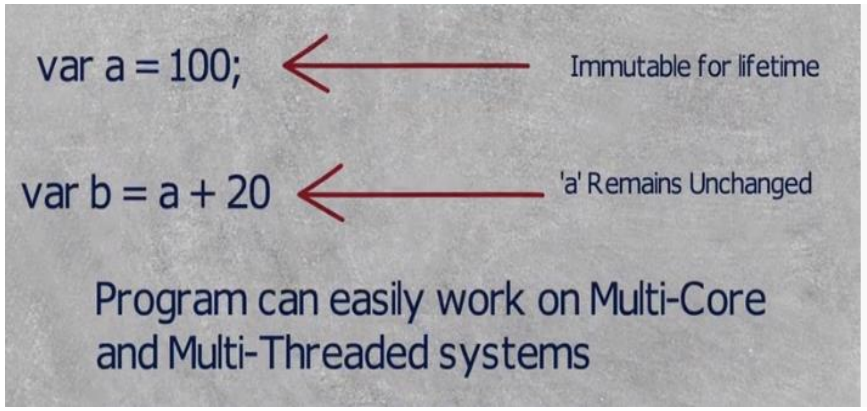
```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs

main :: IO ()
main = print (sum [1..10000])
```

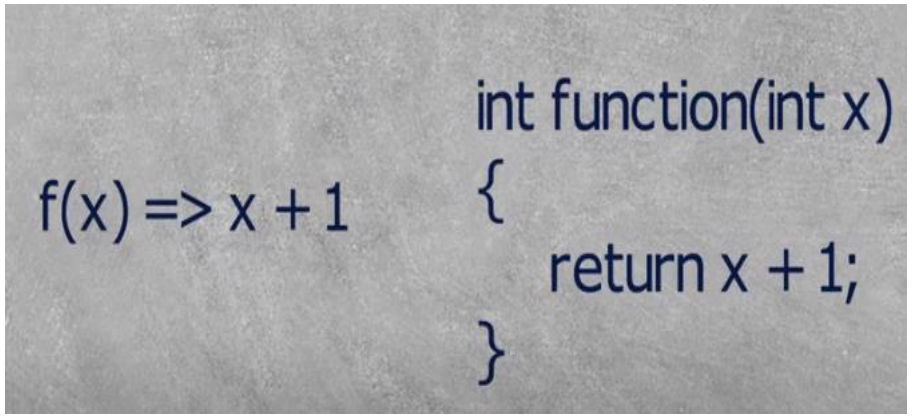
FUNCTIONAL PROGRAMMING

In functional programming we write the function exactly as mathematical function

f(x)=x+1







Immutable means non modifiable, mutable means modifiable

LOGIC PROGRAMMING PARADIGM

- Logic programming refers loosely to
  - The use of facts and rules to represent information
  - The use of deduction to answer queries

Lets understand with example
Overlap(X,Y):-member (M,X), member (M,Y)
In the words, lists X and Y overlap if there is some M that is a member of both X and Y.
Kowalski illustrates the division of labor in logic programming by writing the informal equation

Here logic refers to the facts and rules specifying what the algorithm does, and control refers to how the algorithm can be implemented by applying the rules in a particular order.

We (The programmers) supply the logic part and the programming language supplies the control

- Programs are written in language of some LOGIC
- Execution of a logic program is a theorem proving process; that is computation is done by logic inferences
- Prolog (PROgramming in LOGic) is a representative programming language

Here logic refers to the facts and rules specifying what the algorithm does

- A **logic** is a language. It has syntax and semantics. More than a language, it has inference rules .
- **Syntax:** The rules about how to form formulas, this is usually the easy part of a logic
- **Semantics:** About meaning carried by the formulas, mainly in terms of a logical consequences
- **Inference rules:** describes the correct way to derive conclusions

REVISION

1. **Imperative:** what to do and how to do

2. **Procedural:** Refinement of imperative, introduces the concept of procedures

3. **Object-Oriented Programming:** Solution of problems in which all computations are performed in the context of objects

4. **Declarative:** Only tell what to do, forget how to do

5. **Logic:** Use facts and rules to represent information and deduce answer

Low Level and High Level Language

Low-level language	High-level language
It is a **machine-friendly** language, i.e., the computer understands the machine language, which is represented in 0 or 1.	It is a **user-friendly** language as this language is written in English words, and easily understood by humans.
The low-level language is **slow**, takes more time to execute.	It executes at a **faster** pace.
It requires the **assembler** to convert the assembly code into machine code.	It requires the **compiler** to convert the high-level language instructions into machine code.
The machine code cannot run on all machines, so it is **not a portable** language.	The high-level code can run all the platforms, so it is a **portable** language.
It is **memory efficient and better performance**	It is **less memory efficient**
Debugging and **maintenance** are **not easier** in a low-level language.	Debugging and **maintenance** are **easier** in a high-level language.

References

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

Web References

1. NPTEL Online Video resources- Lecture-01
http://www.nptelvideos.in/2012/11/principles-of-programming-languages.html

2. Stanford University Online lectures- Lecture-01 and Lecture-02
https://www.youtube.com/watch?v=Ps8jOj7diA0&list=PL9D558D49CA734A02

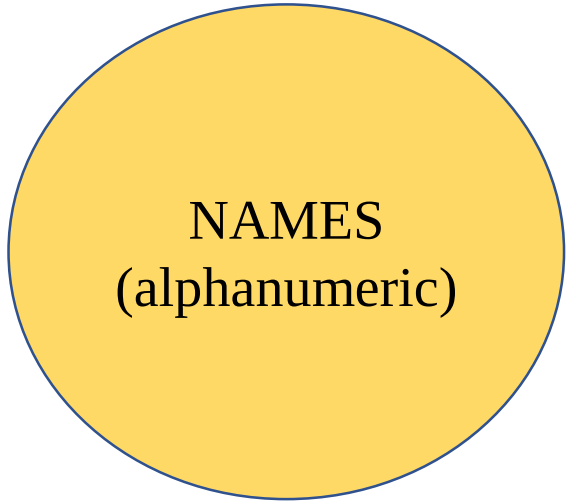
Course: PARADIGMS AND COMPUTER PROGRAMMING FUNDAMENTALS (PCPF)

Course Instructor

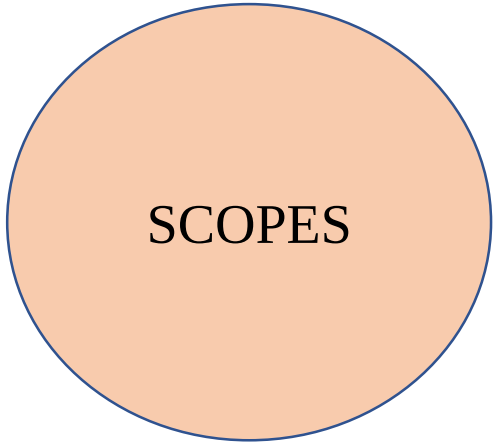
Mrinmoyee Mukherjee B.E (Electronics), M.E (EXTC), PhD (Pursuing)
Assistant Professor
Department of Information Technology
St. Francis Institute of Technology
email: mrinmoyeemukherjee@sfit.ac.in

Academic Year: 2023-24 (Odd Semester)

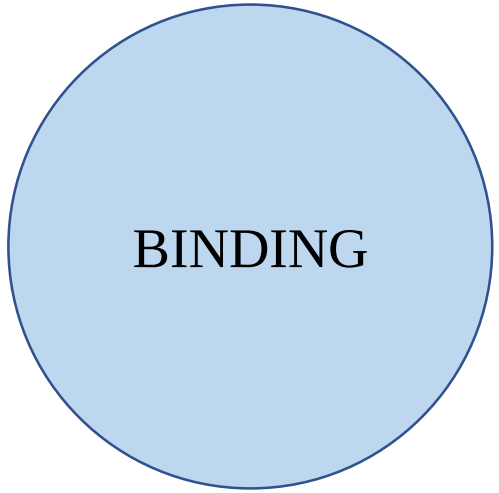
PROGRAMMING LANGUAGES OBEYING DIFFERENT PARADIGM



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.

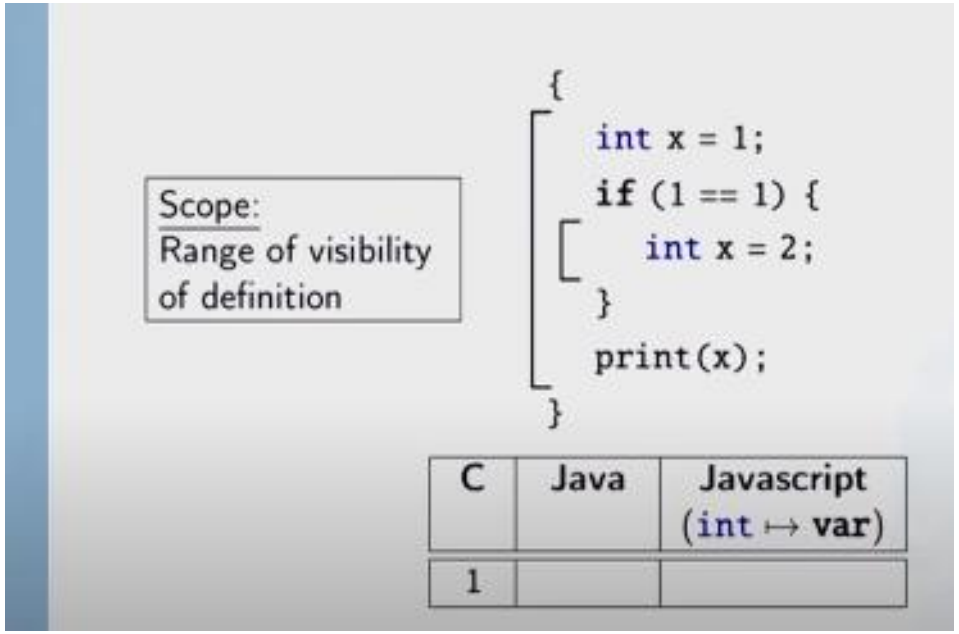
Static Scope

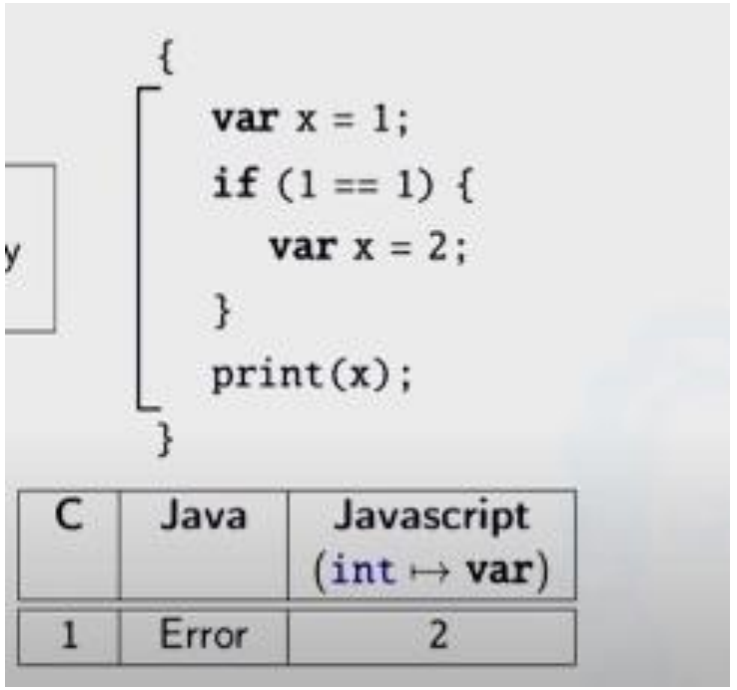
- The scope of a variable can be statically determined, that is, prior to execution. Here a variable always **refers to its top-level environment.**
- Two categories of static-scoped languages
  - Languages **allowing nested subprograms**:
    - Ada, JavaScript, and PHP
  - Languages **which does not 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

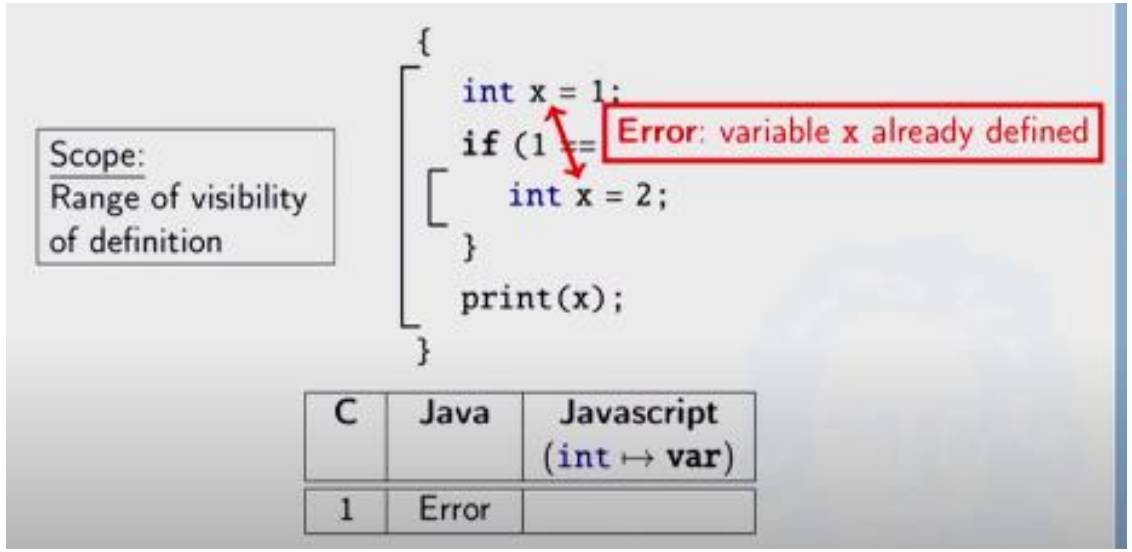
Dynamic Scope

- 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;
    }
  }
}
```

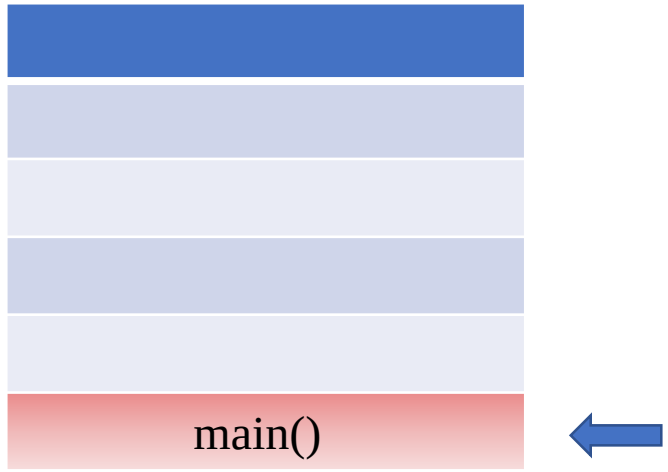
```
// Demonstrate block scope
class Scope {
  public static void main(String args[]){
     int n1=10;   // Visible in main
     if(n1 == 10)
     {
      // start new scope
      int n2 = 20; // visible only to this block
      // num1 and num2 both visible here.
      System.out.println("n1 and n2 : "+ 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)

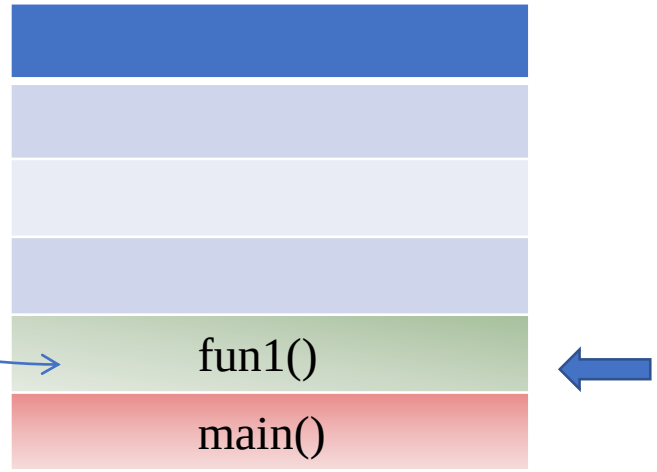
Capable of storing functions

- 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)

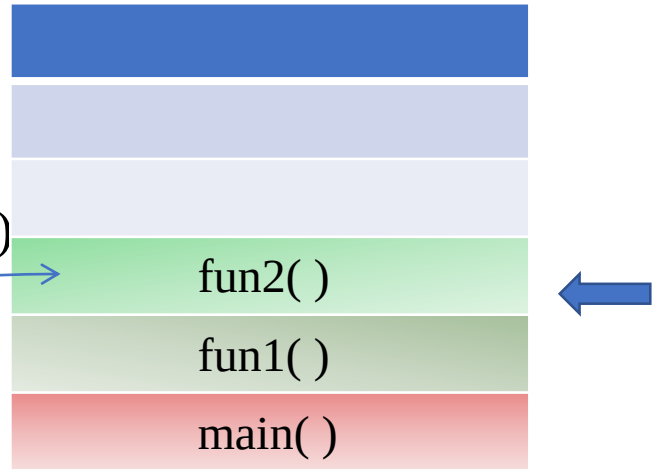
Capable of storing functions

- 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)

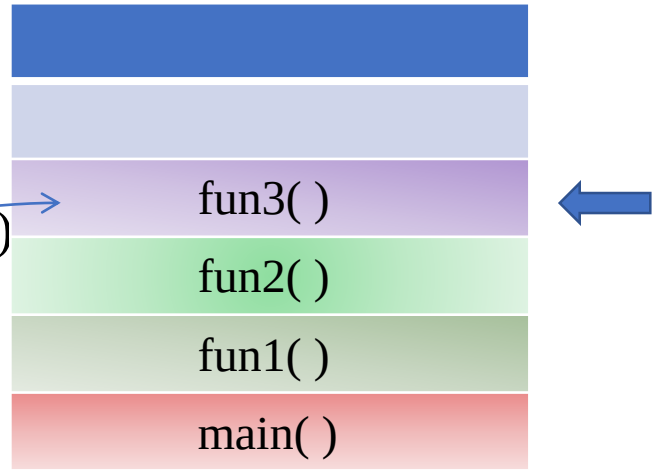
Capable of storing functions

- 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

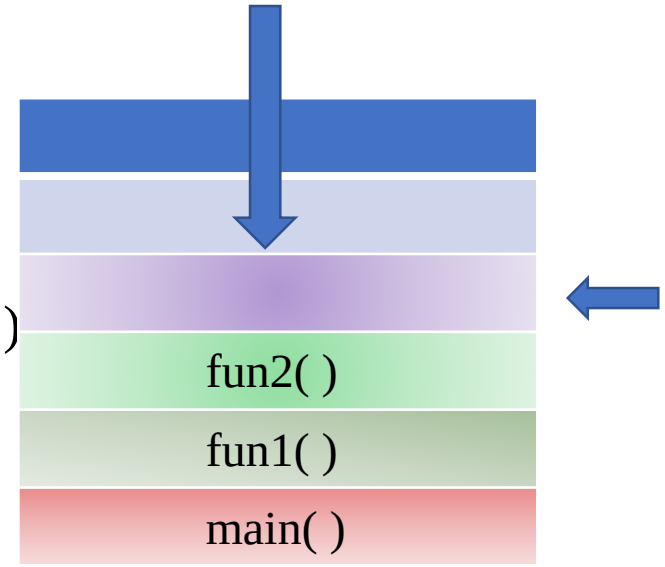
- Stack is a container or a memory segment which holds some data
- Data is retrieved in Last in First Out fashion
- Two operations: push and pop

```
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

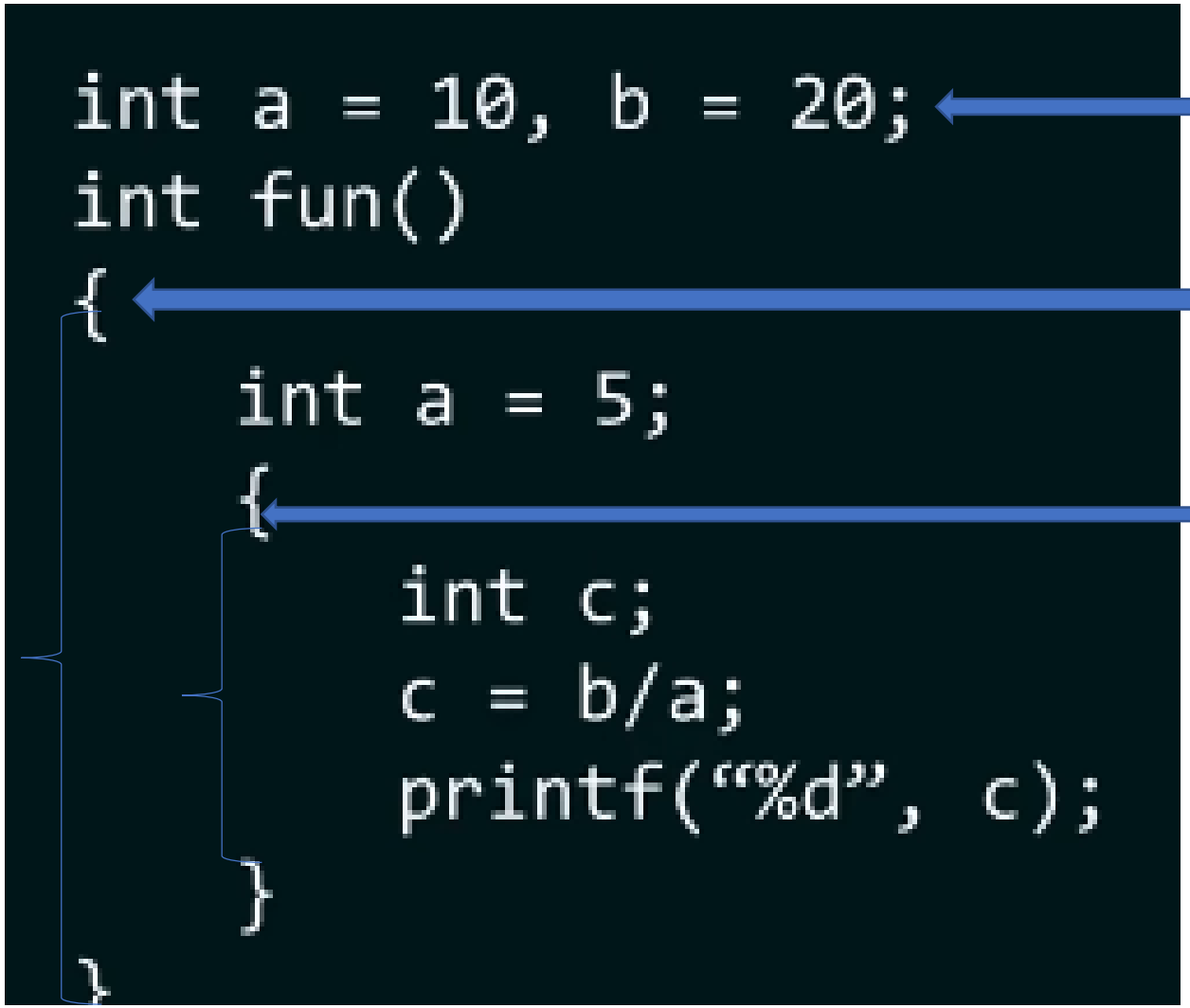


Stack (or Call Stack)

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

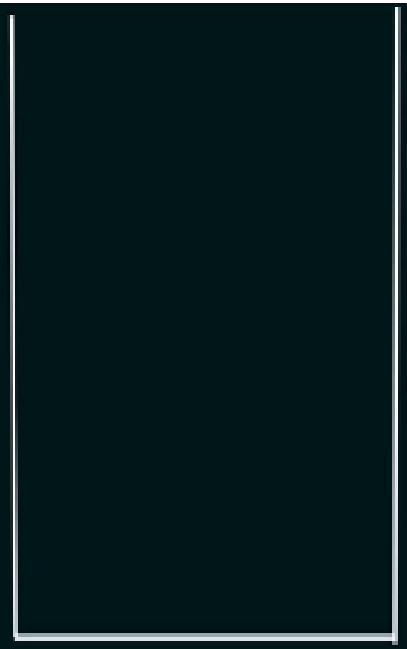
Second search for a and b

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



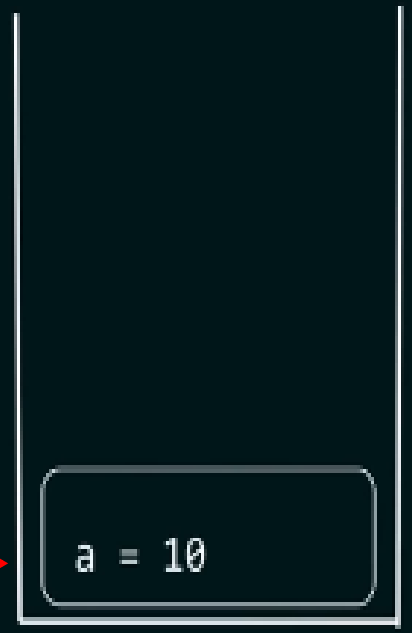
Global Variable will find place in initialised data segment

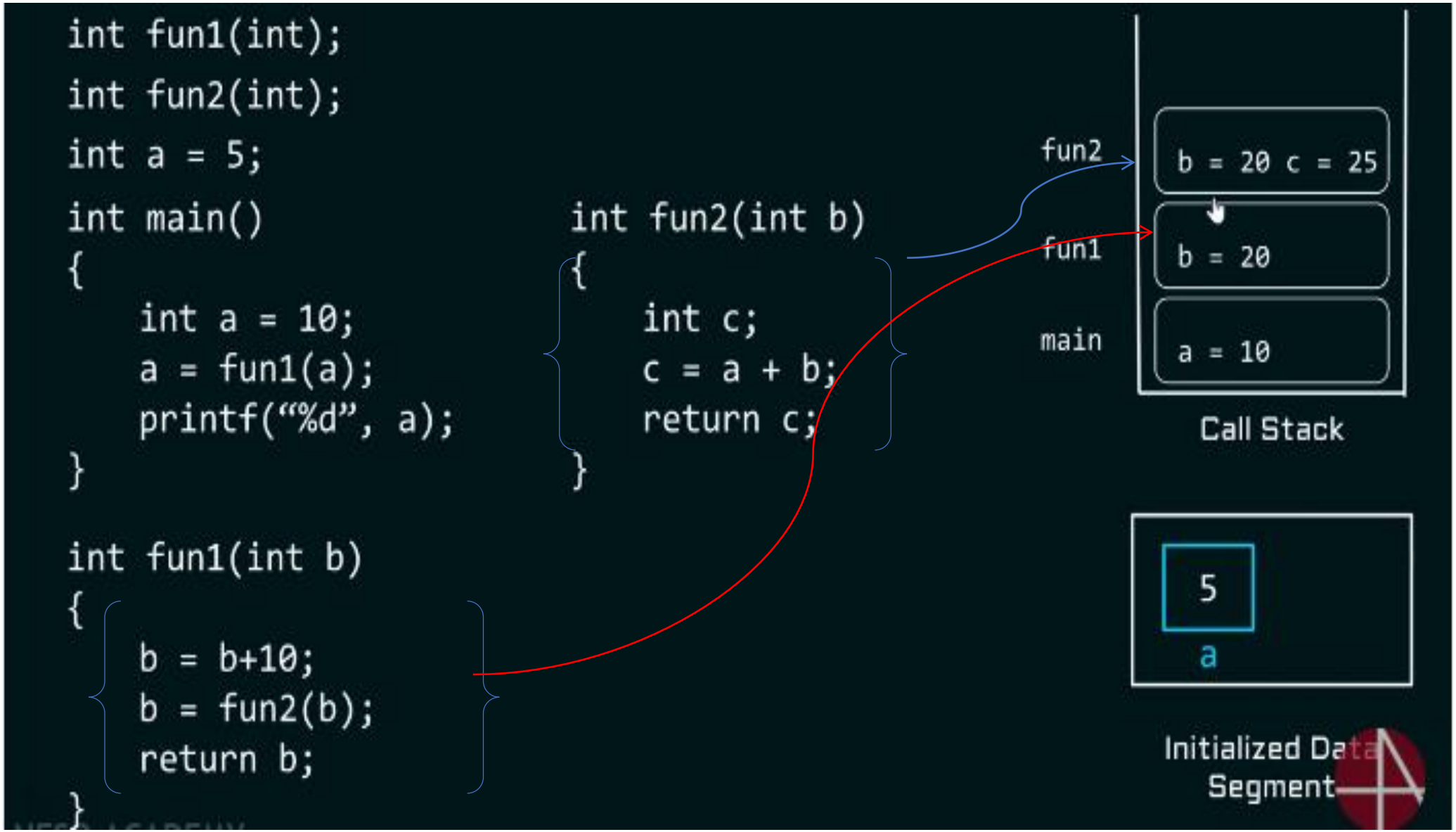
```
int fun1(int);
int fun2(int);
int a = 5;
int main()
{
    int a = 10;
    a = fun1(a);
    printf("%d", a);
}
```

Execution starts from main

main

Activation record of main is created



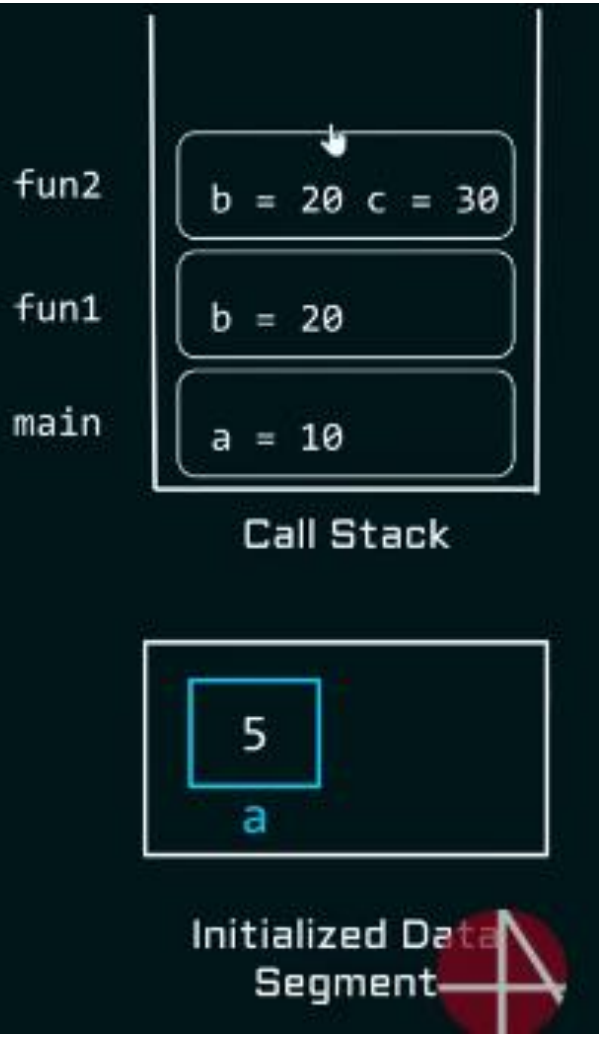


DYNAMIC SCOPING

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;
}
```



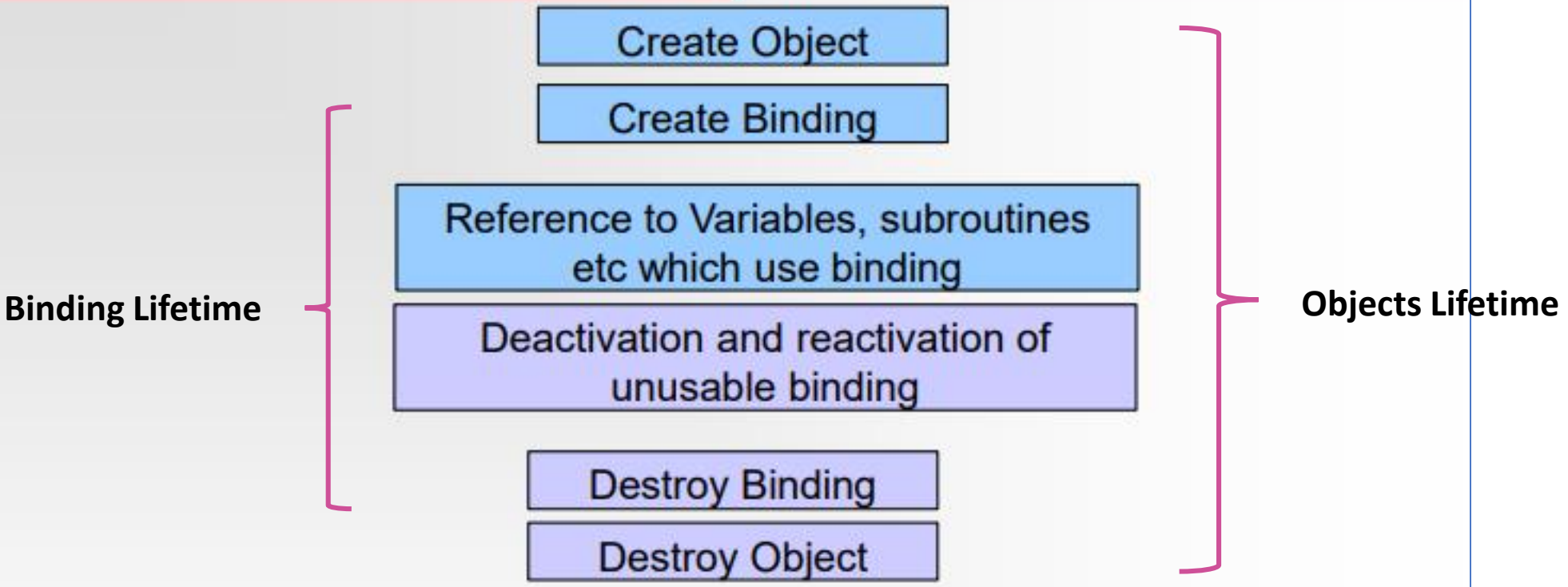
Course: PARADIGMS AND COMPUTER PROGRAMMING FUNDAMENTALS (PCPF)

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)

Object Lifetime



- **Binding's lifetime:**
  - The period of time between the creation and the destruction of a name-to-object binding
- **Object's lifetime:**
  - The time between the creation and destruction of an object

Stoage Allocation Mechanisms

- Object or variable lifetimes generally correspond to one of **three principal storage allocation mechanisms**, used to manage the object's space:

1. Static objects are given an absolute address that is retained throughout the program's execution.
2. Stack objects are allocated and deallocated in last-in, first-out order (LIFO), usually in conjunction with subroutine calls and returns.
3. Heap objects may be allocated and deallocated at arbitrary times. They require a more general (and expensive) storage management algorithm

Static Variables

Ex1: Java static variable

```
class Student{
   int rollno;
   String name;
   String college="ITS";
}
```

```
class Student{
   int rollno;
   String name;
   static String college ="ITS";
                //static variable
}
```

Ex2: Initialization

```
#include <stdio.h>
int main()
{
   static int x;
   int y;
   printf("%d %d", x, y);
}
```

Output: 0 and garbage

- Static variables are bound to memory cells (**not stack**) before execution begins and remains bound to the same memory cell throughout execution of program.
  - All FORTRAN variables, C static variables in functions
- Static variables are initialized as 0 if not initialized explicitly similar to global variables.
- Advantages:
  - Efficiency: All **addressing** of static variables can be **direct**. **No run-time overhead** is incurred for allocation and deallocation of static variables.
  - History-sensitive: variables retain their values between separate executions of the subprogram.
- Disadvantage:
  - Storage cannot be shared among variables.
  - Ex: if two large arrays are used by two subprograms, which are never active at the same time, they cannot share the same storage for their arrays.

Example of Static Variable

```
#include<stdio.h>
int fun()
{
 static int count = 0;
 count++;
 return count;
}
int main()
{
 printf("%d ", fun());
 printf("%d ", fun());
 return 0;
}
```

Output: 1 2

```
#include<stdio.h>
int fun()
{
 int count = 0;
 count++;
 return count;
}
int main()
{
 printf("%d ", fun());
 printf("%d ", fun());
 return 0;
}
```

Output: 1 1

Stack-dynamic Variables

Recursion

```
int f(int x) {
        int a = 0;
        int y=5;
        ...
        f(y+a);
        ...
    }
```

- A stack-dynamic variable is one that is bound to an address on the stack, and created dynamically for that purpose
- It may also be unbound during run-time, and its memory cell deallocated by being popped off the stack.
- Stack-dynamic variables are allocated from the run-time stack.
- At any time during the run of the program, the stack contains the memory cells for all the subroutines currently executing, including all the invocations of recursive subroutines currently executing.
- Advantages:
  - Allows recursion: each active copy of the recursive subprogram has its own version of the local variables.
  - In the absence of recursion, it **conserves storage** b/c all subprograms share the same memory space for their locals

Example Subroutine Calls

Function-1

```
int max(int num1, int num2) {
   int result;          /* local variable */
   if (num1 > num2)
      result = num1;
   else
      result = num2;
   displayOutput(result);
   return result;
}
```

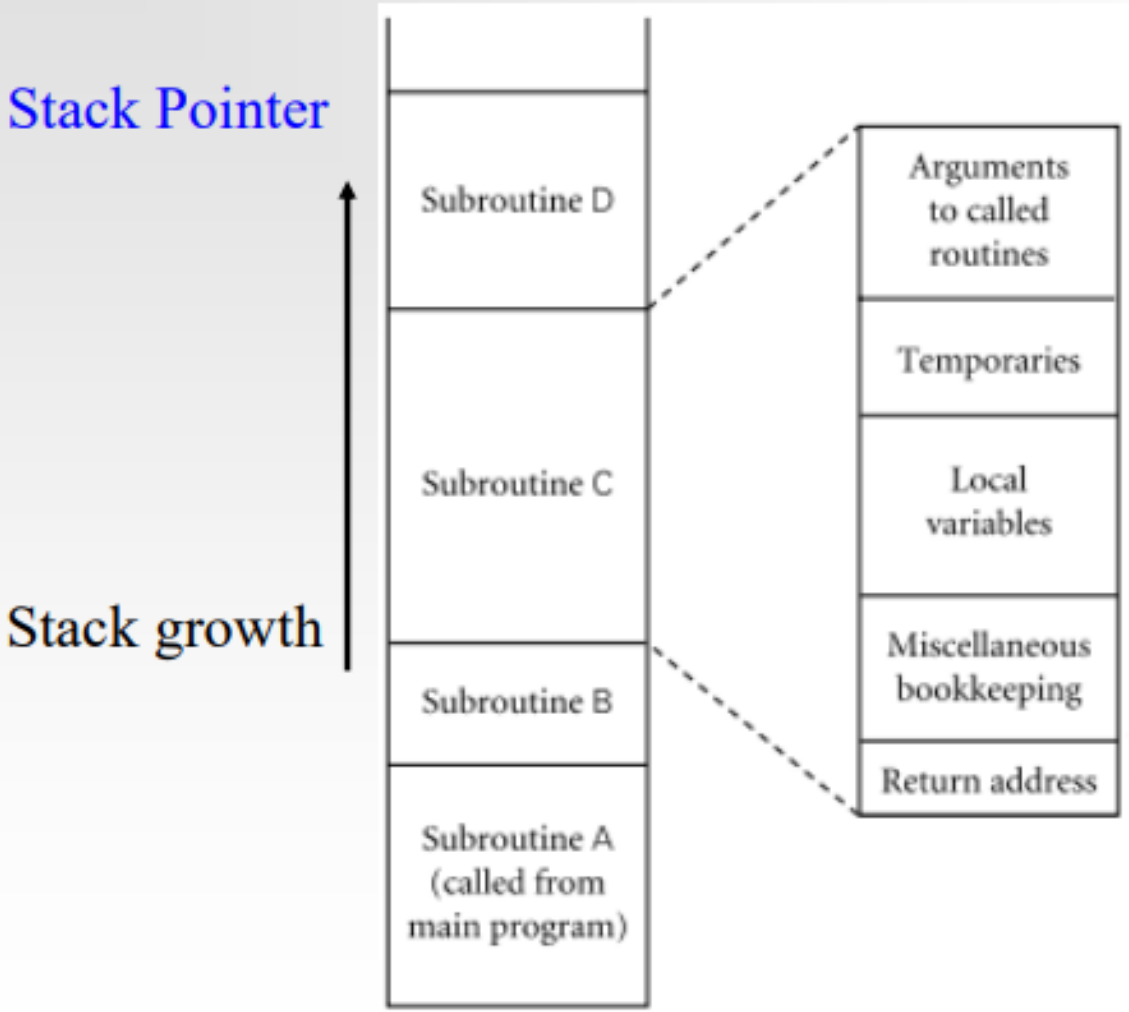
Function-2

```
displayOutput(int x)\{
            -----
}
```

```
int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;

   /* calling a function to get max value */
   ret = max(a, b);

   printf( "Max value is : %d\n", ret );

   return 0;
}
```

Stack-dynamic Variables in Subroutine

Subroutine A calls subroutine B and B calls subroutine C
and C calls subroutine D



Subroutines

```
int f(int x) {
        int a = 0;
        int y=5;
        ...
        f(y+a);
        ...
}
```

Recursion

```
int f(int x) {
        int a = 0;
        int y=5;
        ...
        f(y+a);
        ...
}
```

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

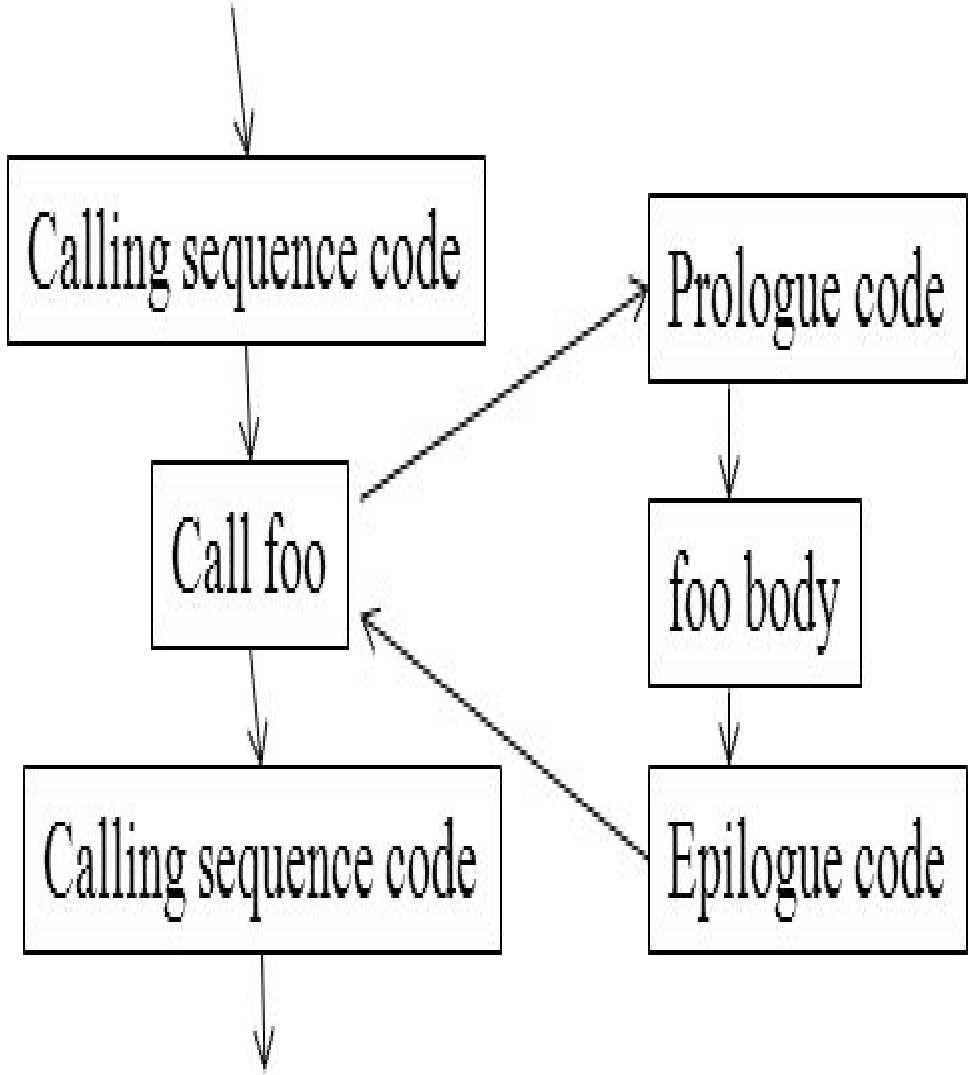
Calling Sequence: Code executed by the caller immediately before and after the call

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



Explicit Heap-dynamic Variables

- The heap is a unstructured pool of nameless memory cells.
- variables are bound to memory cells that are allocated and deallocated by explicit run time instructions specified by programmer during execution.
- lifetime = from explicit allocation to explicit deallocation
- These variables, which are allocated to and deallocated from the heap, can only be referenced through pointers or reference variables.
- The heap is a collection of storage cells whose organization is highly disorganized because of the unpredictability of its use.
- e.g. Dynamic objects in C++ (via new and delete)

```
int *intnode;                  // create a pointer
intnode = new int;             // allocates the heap-dynamic variable
delete intnode;    // deallocates heap-dynamic variable to which intnode points
```

  - An explicit heap-dynamic variable of int type is created by the new operator.
  - This operator can be referenced through the pointer, intnode.
  - The var is deallocated by the delete operator.

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.

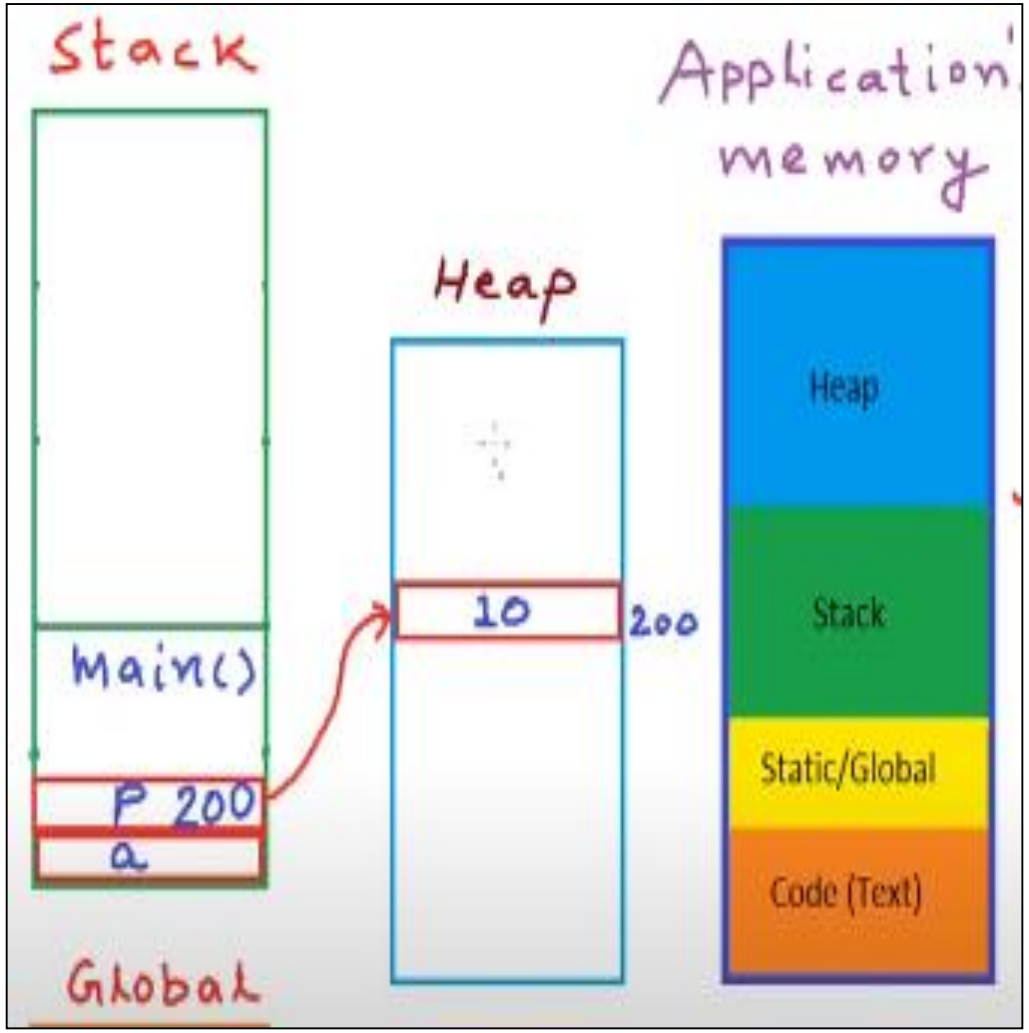
```
int *ptr = (int* )malloc(4)
```

HEAP

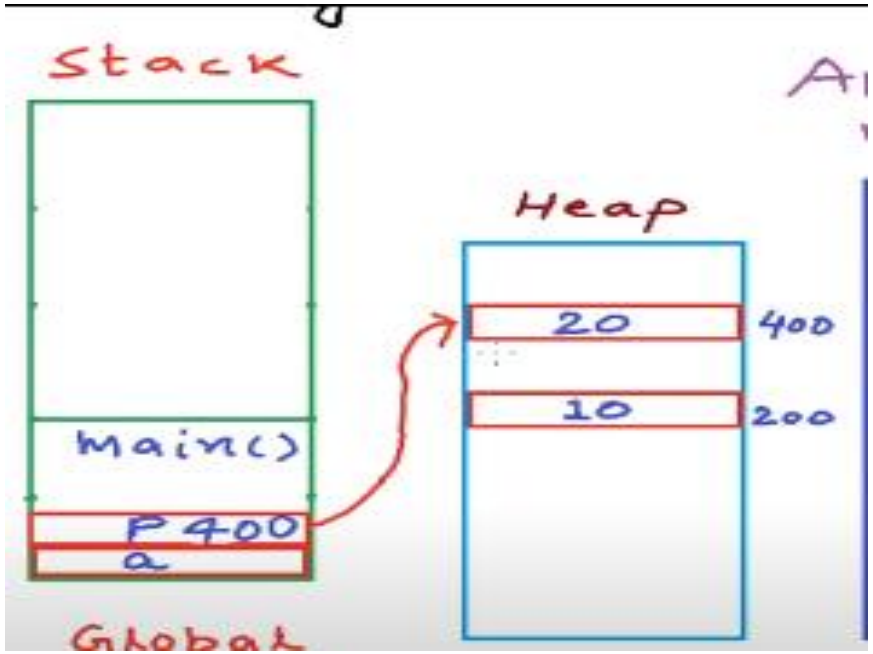
- Heap is a region of memory (storage) in which sub-blocks can be allocated and de-allocated at arbitrary time.
- Required for dynamic allocation of memory (at run time)

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

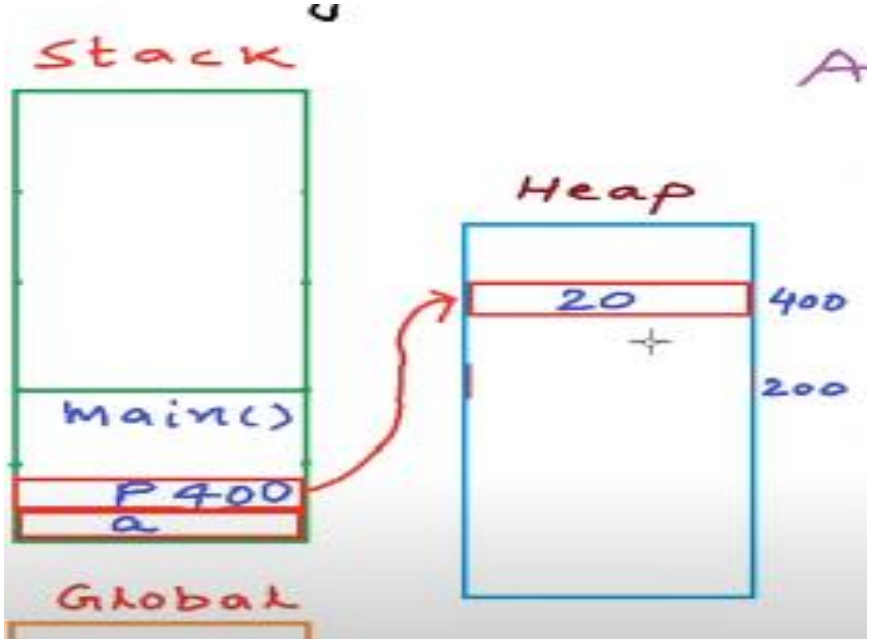
malloc()
realloc()
free()



```
#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;
```



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

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)

Type in Programming

- Most high-level programming languages include a notion of type (data type) for expressions and/or objects. (Ch-7 from Scott)
- In general, a type is a collection of computational entities that share some common property.
- A type gives semantics to a piece of data.
- A type is a set of value. For the type integer, it will be a range of decimals. For the type string, it will be a range of possible strings.

1. Types **provide implicit context for many operations**, so the programmer does not have to specify that context explicitly.
   - Ex. In Pascal, for instance, the expression **z = a + b** will use integer addition if a and b are of integer type; or it will use floating-point addition if a and b are of real type.
2. Types **limit the set of operations** that may be performed in a semantically valid program.
   - Ex. They prevent the programmer from adding a character and a record, or
   - Ex. passing a file as a parameter to a subroutine that expects an integer.

Data Types	Sizes in byte	Sizes in bits	Range formula 2^n-1	Ranges
int	4 bytes	32bits	2^{32}-1	-2,147,483,648 to 2,147,483,647
unsigned int	4 bytes	32 bits	2^{32}-1	0 to 4294967295
float	4 bytes	32 bits	2^{32}-1(5 points)	3.4×10^{-38} to $3.4 \times 10^{+38}$
double	8 bytes	64 bits	2^{64}-1(15 points)	1.7×10^{-308} to $1.7 \times 10^{+308}$
long double	10 bytes	80 bits	2^{80}-1(19 points)	1.7×10^{-4932} to $1.7 \times 10^{+4932}$
char	1 byte	8 bits	2^8-1	0 to 255

Type in Programming

- There are three main uses of types in programming languages:
  - **naming** and **organizing** concepts,
  - making sure that bit sequences in computer memory are **interpreted consistently,**
  - **providing information to the compiler** about data manipulated by the program.
- Using types to organise a program **makes it easier** for someone to **read, understand, and maintain the program**.
- Types can serve an important purpose in **documenting the design and intent** of the program.
- Type information in programs can be **used for** many kinds of **optimisations**.

Example of Type Systems

- Ex of Type system

```
<?php
printf((3 + "Hello World!") . "\n");
// => PHP Warning: A non-numeric value encountered in /home/myusername/phpgoodies/test.php on line 3
// => 3
printf("The execution continue!");
// => The execution continue!
?>
```

- We are trying to add integer to string. When you violate the rules of a type system, the outcome can range between these two extremes:
  - The interpreter or compiler will silently try to fix the problem and continue.
  - The interpreter or compiler will throw an error and stop.

Type Checking

- If types push us to respect some rules, a programming language need an algorithm to check if we respect them. This is called type checking.
- In most statically typed languages, every definition of an object (constant, variable, subroutine, etc.) must specify the object's type
- A type error occurs when a computational entity (object) is used in a manner that is inconsistent with the concept it represents.
- Type checking is the process of ensuring that a program obeys the language's type compatibility rules.
- Type checking may occur at compile time (a static check) or at run-time.
- Type checking is used to prevent type errors, ensuring that the operations in a program are applied properly.
- A violation of the rules is known as a type clash.

Type Checking

- Strongly-typed Language:
  - If a language specification requires its typing rules strongly (i.e., more or less allowing only those automatic type conversions that do not lose information), one can refer to the process as strongly typed otherwise it is weakly typed.
  - It does not allow variables to be used in a way inconsistent with their type
  - If the compiler can guarantee that the accepted programs will execute without type errors.
  - Strong typing offers more safety, but cannot guarantee complete type safety.
- Statically-typed Language: A language is said to be statically typed if it is strongly typed and type checking can be performed at compile time.

Example: Type Safe

- Consider example

```
var x := 5;
var y := "37";
var z := x + y;
```

In this example z will be 42.

- Type safe language will convert y to integer and add to x
- But if var y := ”Hello world"; program will crash
- Hence program of a language is both type-safe and memory-safe

Example: Type Safe

- Consider example in C

```
int x = 5;
char y[] = "37";
char* z = x + y;
printf("%c\n", *z);
```

- In this example z will point to a memory address five characters beyond y.
- This is memory that the program is not expected to access.
- In C terms this is simply undefined behaviour and the program may do anything; with a simple compiler it might actually print whatever byte is stored after the string "37".
- As this example shows, C is not memory-safe.
- As arbitrary data was assumed to be a character, it is also not a type-safe language.

Type Safety

- A programming language is type safe **if no program is allowed to violate its type distinctions.**
- A type system enables compiler to detect meaningless/ invalid code.
- E.g., we can identify an expression 3 / "Hello, World" as invalid.

Safety	Example language	Explanation
Not safe,	C, C++	Type casts pointer arithmetic
Almost safe	Pascal	Explicit deallocation;
Safe	LISP, SML, Smalltalk, Java	Type checking

Static and Dynamic Type Checking

- Run-time (Dynamic) type checking:
  - Dynamic type checking is the process of verifying the type safety of a program at runtime. Its slow process.
  - Implementations of dynamically type-checked languages generally associate each runtime object with a type tag (i.e., a reference to a type) containing its type information.
  - Examples: LISP, Smalltalk.
- Compile-time (Static) type checking:
  - The compiler checks the program text for potential type errors before run time. Its fast process.
  - Example: SML.
- Most programming languages use some combination of compile-time and run-time type checking because many useful features or properties are difficult or impossible to verify statically.

Type Compatibility

- A type checker for a statically-typed language must verify that the type of any expression is consistent with the type expected by the context in which that expression appears.
- For example, in an assignment statement of the form x := e, the inferred type of the expression e must be consistent with the declared type of the variable x.
- This notion of consistency, called compatibility, is specific to each programming language.

Type Equality

- The question of type equality arises during type checking.
- What does it mean for two types to be equal?
  - Structural equality.
  - Name equality

1. Structural equality.

- Two type expressions are structurally equal if and only if they are equivalent under the following three rules.
  - A type name is structurally equal to itself.
  - Two types are structurally equal if they are formed by applying the same type constructor to structurally equal types.
  - After a type declaration, say type n = T, the type name n is structurally equal to T.

Type Equality Example

```
typedef struct {
        int data[100];
        int count;
        } Stack;

  typedef struct {
        int data[100];
        int count;
        } Set;

  Stack x, y;
  Set r, s;
```

- x and y are name equivalent having same datatype, x=y is allowed
- r and s are name equivalent having same datatype, , r=s is allowed
- In structural equivalence two types are equal if, and only if, they have the same "structure", which can be interpreted in different ways.
- Here x and r have same structure
- C does not allow structural equivalence

THANK YOU