Module I
Part 2

- **Names, Scopes, and Bindings:-**

✓ Names and Scopes, Binding Time,

✓ Scope Rules , Storage Management,

✓ Binding of Referencing Environments.

Names, Scopes, and Bindings:-

- Name: **name** is a string of characters used to identify some entity. Allow us to refer to variables, constants, functions, types, operations etc
- Binding: An association ofa name with an object
- Scope: The part of the program in which the binding is active

Variables

- A variable in an imperative language, or an object-oriented language, is a six-tuple:

  <name, address, value, type, lifetime, scope>

- A **name** is a string of characters used to identify some entity.
- Declaration of type, usage with a value, lifetime, scope of names are a major consideration in programming languages

VARIABLE

- name
- address
- type
- value
- scope
- Binding& lifetime

For example, if we write **int x;**

- what will be the name of the variable and type of x?.
- The place of this declaration in the program decides where and how long x is available (scope, lifetime).
- Its address is determined when its program unit is executing
- Lastly , usage of x in statements decides what is its current value

Name

- Name is a mnemonic character string representing something
- Names are identifiers(alpha numeric tokens)
- Names refer to abstraction-programmer associates name with complicated program fragment/element
  - Control abstraction
    - Allows the programmer to hide sequence of complicated code in a name.
  - Data Abstraction:Allows the programmer to hide data representation behind a set of operations.

- Names in most programming languages have the same form: a letter followed by a string consisting of letters, digits, and (_)
- use of the _ was widely used in the 70s and 80s(not popular)
- C-based languages (C, C++, Java, and C#), replaced the _ by the "camel" notation (ex: **myStack)**
- In C every keywords written in small letters
- In C, C++,Java etc. names are case sensitive

  Ex) rose, Rose, ROSE are distinct names

- the same name can be reused in different contexts and denote different entities.
- Different types, addresses and values may be associated with such occurrences of a name.
- Each occurrence has a different lifetime (when and for how long is an entity created?), and a different scope (where can the name be used?).

```
void a()
{
   int b;
  /* ... */
 }
float b()
{
char a; /* ... */
}
```

- **Variable-** variable is an abstraction of a memory cell(s).
- **Name**

  - Not all variables have names: **Anonymous**, heap-dynamic variables

- **Address**
  - A **variable name may have different addresses at different places**
  - The memory address with which it is associated at different times during execution
  - The address of a variable is sometimes called its ***l*-value** because that is what is required when a variable appears in the **left** side of an assignment statement

- **Type**
  - Determines the **range of values of variables and the set of operations** that are at different times during execution
  - For example, the int type in Java specifies a value range of -2147483648 to 2147483647, and arithmetic operations (+,-,*,/,%)
- **Value**
  - The value of a variable is the **contents of the memory cell** or cells associated with different times during execution
- A variable’s value is sometimes called its r-value because that is what is required when a variable appears in the right side of an assignment statement.

BINDING

Binding , Binding time,& Referencing Envt

- Binding is association of 2 things-an attribute with an entity.
- **Binding time** is the time at which a binding takes place.
- Referencing Environment-complete set of bindings at a given point in a program.

Binding

- ***Binding***
- the operation of associating two things, like a name and the entity it represents.
- Binding is associate an attribute with an entity.
- Examples of attributes are name, type, value.
- ***Binding occurs at various times in the life of a program***
- The compiler performs a process called binding when an object is assigned to an object variable.

Binding time

- Binding time is the the moment when the binding is performed (compilation, execution, etc).
- The early binding (static binding) refers to compile time binding
- late binding (dynamic binding) refers to runtime binding.

The Concept of Binding

- The ***l*-value** of a variable is its **address**.
- The ***r*-value** of a variable is its **value**.
- A **binding** is an association, such as between an attribute and an entity,or between an operation and a symbol.

•A binding is **static** if it first occurs **before** run time and remains unchanged throughout program execution.

•A binding is **dynamic** if it first occurs **during** execution or can change during execution of the program.

Possible Binding Time

- Binding Time is the time at which a binding is created

1. Language design time
2. Language implementation time
3. program writing time
4. compile time
5. link time
6. load time
7. *Runtime*

1. **Language design time** (bind operator symbols to operations. * to mul)
    - program structure, possible types , control flow constructs are chosen
2. **Language implementation time**
    - Coupling of I/O to OS, arithmetic overflow, stack size, type equality ,handling of run time exceptions
    - Ex)A data type such as **int** in C is bound to a **range** of possible values

1. **program writing time**
    - Programmers choose algorithms, data structures and names
2. **compile time**
    - bind a variable to a **particular data type** at compile time

5. link time

- Library of standard subroutines joined together by a linker.

6. load time

- Refers to the point at which the **OS loads the program into memory** so that it can run. virtual address are chosen at link time and physical addresses change at run time.

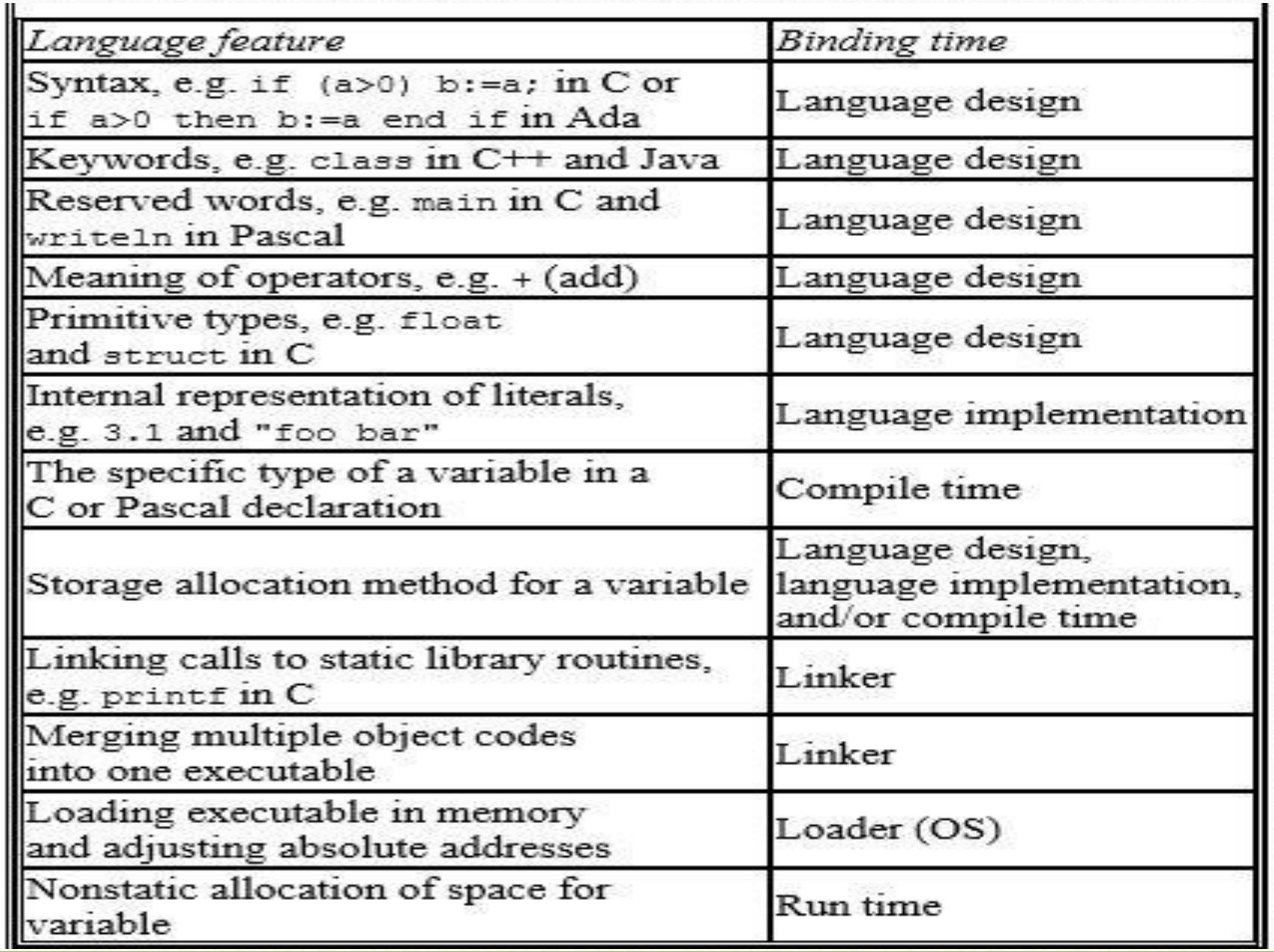
- bind a variable to a **memory cell** (ex. C **static** variables)

7.Runtime

- refers to the entire span from the beginning to the end of execution..virtual functions, values to variables, many more.

- bind a **nonstatic** local variable to a memory cell

Binding Time Examples



Language feature	*Binding time*
Syntax, e.g. `if (a>0) b:=a;` in C or `if a>0 then b:=a end if` in Ada	Language design
Keywords, e.g. `class` in C++ and Java	Language design
Reserved words, e.g. `main` in C and `writeln` in Pascal	Language design
Meaning of operators, e.g. + (add)	Language design
Primitive types, e.g. `float` and `struct` in C	Language design
Internal representation of literals, e.g. `3.1` and `"foo bar"`	Language implementation
The specific type of a variable in a C or Pascal declaration	Compile time
Storage allocation method for a variable	Language design, language implementation, and/or compile time
Linking calls to static library routines, e.g. `printf` in C	Linker
Merging multiple object codes into one executable	Linker
Loading executable in memory and adjusting absolute addresses	Loader (OS)
Nonstatic allocation of space for variable	Run time

Assigning properties to variables

1. variable → name
   - compile time
2. variable → address
   - load time or run time (e.g. C),
   - run time (e.g. Smalltalk)
3. variable → type
   - compile time (e.g. Java),
   - run time (e.g. Scheme)
     - described in declarations, ifbound at compile time

4. variable → value
    - run time,
    - load time (initialization)
        - specified in statements, mainly assignment
5. variable → lifetime
    - compile time
        - described in declarations
6. variable → scope
    - compile time
        - expressed by placement of declarations

Static & Dynamic Binding

- The terms static and dynamic are generally used to refer to things bound before run time and at run time, respectively
- A binding is **static** if it first occurs **before** run time and remains unchanged throughout program execution
- A binding is **dynamic** if it first occurs **during** execution or can change during execution of the program.
- Compiler-based language implementations tend to be more efficient than interpreter-based implementations because they make earlier decisions

Example: Static & Dynamic Binding

- compiler analyzes the syntax and semantics of global variable declarations once, before the program ever runs.
- It decides on a layout for those variables in memory and generates efficient code to access them wherever they appear in the program.
- A pure interpreter,by contrast,must analyze the declarations every time the program begins execution.
- interpreter may reanalyze the local declarations within a subroutine each time that subroutine is called

- When type of the object is determined at compiled time(by the compiler), it is known as static binding.

```
//static Example

class Dog{

 private void eat(){ System.out.println("dog is eating...");
}
public static void main(String args[])
 {
  Dog d1=new
  Dog();  d1.eat();

 }   }
```

#dynamic example

```
class Animal{

 void eat(){System.out.println("animal is eating...");}

}

class Dog extends Animal
{     void eat()      {System.out.println("dog is eating...");}

  public static void main(String args[])
        {     Animal a=new Dog();

        a.eat();

         }    }
```

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type.

o/p—Dog is eating

Dynamic Type Binding (JavaScript and PHP)

- Specified through an assignment statement
- Ex, JavaScript

  list = [2, 4.33, 6, 8];// single-dimensioned array list = 47;//

  scalar variable

- Advantage: **flexibility** (generic program units)
- Disadvantages:

  – **High cost** (dynamic type checking and interpretation)

  - Dynamic type bindings must be implemented using pure interpreter **not** compilers.

  -**Type error detection by the compiler is difficult** because **any** variable can be assigned a value of **any** type.

Effect of Binding Time

- **Early binding times** (before run time)are associated with **greater efficiency**
  - Syntactic and sematic checking can be done at compile time only once and run time overhead can be avoided
- **late binding times**(at run time) are associated with **greater flexibility.**
  - Interpreters allows programs to be extended at run time
  - Method binding in oops must be late to support dynamic binding.

SCOPE

Scope

- Names are bound to various elements of a program.
- The scope of a name N means all places in the program where N denotes the same object.
- Scope of a variable is the range of statements in which the variable is visible
- A var is **visible** in a statement if it can be referenced in that statement.
  - **Local var** is local in a program unit or block if it is declared there.
  - **Non-local var** of a program unit or block are those that are visible within the program unit or block but are not declared there

❑ Scope rules of a language determine how a particular occurrence of a name is associated with a variable

Types of scoping

- **Static scoping**

✓ It allows us to determine the use of every variable in a program statically, without executing it at compile time.

✓ in **static scope rules** the bindings are defined by the physical (lexical) structure of the program.

- **Dynamic scoping**

✓ With **dynamic scope rules**, bindings depend on the current state of program execution

✓ The idea is to search for a name in a chain of called procedures, starting from the main program. This chain is built according to the visibility rules

Static Scope Rules

- In a language with static(lexical) scoping , the **bindings** between names and objects can be **determined at compile time** by examining **the text of the program** , without consideration of the **flow of control at run time**
- The simplest static scope rule is probably that of early versions of Basic
- Scope rules are somewhat more complex in (pre-Fortran 90) Fortran

Static Scope Rules

Static scope refers to the scope of the container

– To connect a name reference to a variable, the compiler must find the declaration

- ❑ Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- ❑ Enclosing static scopes (to a specific scope) are called its static ancestors; the nearest static ancestor is called a static parent

- Ex: **Suppose a reference is made to a var x in subprogram Sub1**.
  - correct declaration is found by searching the declarations in subprogram Sub1.
  - If no declaration is found for the **var x** there, the search continues in the declarations of the subprogram that declared subprogram Sub1, which is called its **static parent**.
  - If a declaration of x is not found there, the search continues to the next larger enclosing unit (the unit that declared Sub1's parent), and so forth, until a declaration for x is found or the largest unit's declarations have been searched without success . An undeclared var error has been detected.
  - The static parent of subprogram Sub1, and its **static parent**, and so forth up to and including the main program, are called the static **ancestors** of Sub1.

Static Scope

- Under static scoping, the reference to the var X in Sub1 is to the X declared in the procedure Big.
- This is true b/c the search for X begins in the procedure in which the reference occurs, Sub1, but no declaration for X is found there.
- The search thus continues in the **static parent of Sub1 is Big,** where the declaration of X is found.

```
Procedure Big is
   X : Integer;
   Procedure Sub1 is
      Begin          -- of Sub1
      ...X...
      end;           -- of Sub1
   Procedure Sub2 is
      X Integer;
      Begin          -- of Sub2
      ...X...
      end;           -- of Sub2
   Begin             -- of Big
   ...
   end;              -- of Big
```

Static Scope rules

- The count of sub is **hidden** from the code inside the while loop.
- A declaration for a var effectively hides any declaration of a var with the same name in a larger enclosing scope.
- C++ and Ada allow access to these "hidden" variables

  - In Ada: Main.X

  -In C++: class_name::name.
- The reference to count in the while loop is to that loop’s local count.

Ex: Skeletal C#

```
void sub()
{
   int count;
   ...
   while (...)
   {
       int count;
       count ++;
       ...
     }
     ...
}
```

Dynamic scoping Rule

Dynamic scope refers to the caller of the function

- In a language with dynamic scoping, the **bindings** between names and objects **depend on the flow of control at run time**, and in particular on **the order in which subroutines are called.**
- Based on **calling sequences** of program units, not their textual layout and thus the scope is determined at **run time**.
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point

Dynamic Scope

- **Big calls Sub1**

The dynamic parent of Sub1 is Big. The reference is to the X in **Big**.

- **Big calls Sub2 and Sub2 calls Sub1**

The search proceeds from the local procedure, Sub1, to its caller,**Sub2**, where a declaration of X is found.

-if static scoping was used, in either calling sequence the reference to X in Sub1 would be to **Big’s X**.

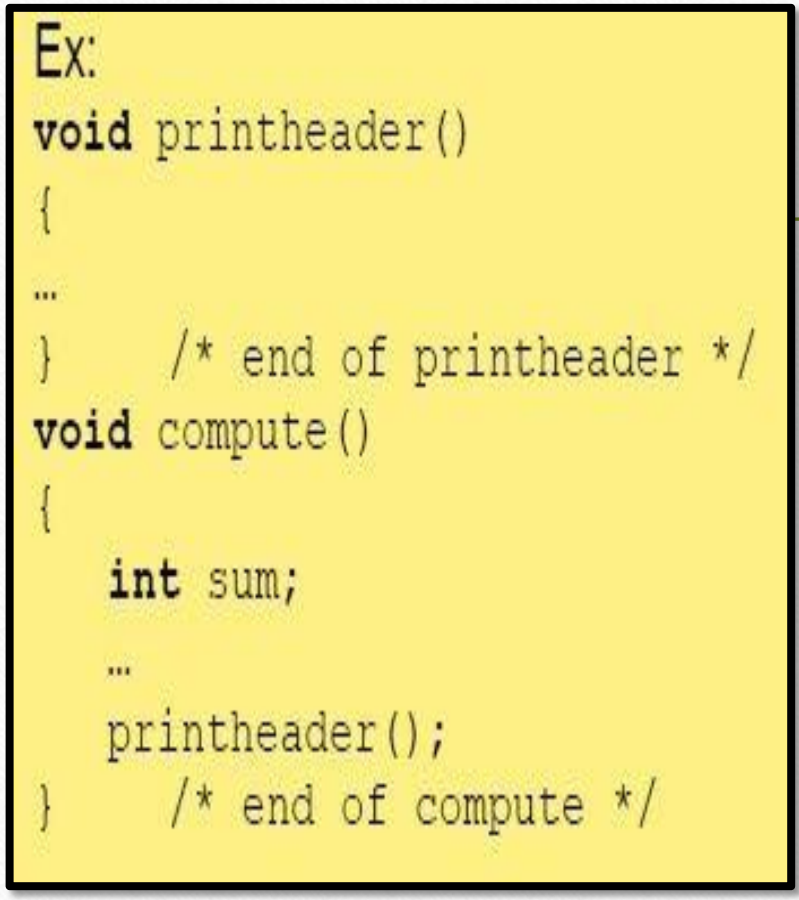
```
Procedure Big is
   X : Integer;
   Procedure Sub1 is
      Begin         -- of Sub1
      ...X...
      end;          -- of Sub1
   Procedure Sub2 is
      X Integer;
      Begin         -- of Sub2
      ...X...
      end;          -- of Sub2
   Begin            -- of Big
   ...
   end;             -- of Big
```

Life time

Life time

- *Object lifetime* - the period between the object creation and destruction.
  - Example: time between creation and destruction of a dynamically allocated variable in C++ using new and delete
- *Binding lifetime* - the period between the creation and destruction of the binding.
  - Ex) A functions formal argument is bound to actual argument
  - Ex)A reference variable is assigned with the address of an object.

Scope & Lifetime



Ex:
```
void printheader()
{
…
}     /* end of printheader */
void compute()
{
   int sum;
   …
   printheader();
}     /* end of compute */
```

- The **scope** of sum is contained within compute.
- The **lifetime** of sum extends over the time during which printheader executes.
- Whatever storage location sum is bound to before the call to printheader,that binding will continue during and after the execution of printheader.

STORAGE MANAGEMENT

Storage Management

- **how the memory of   the computer is organized for a running program?**
- **Allocation** - getting a cell from some pool of available cells.
- **Deallocation** - putting a cell back into the pool.
- The **lifetime** of a variable is the time during which it is bound to a particular memory cell.

So the lifetime of a var begins when it is bound to a specific cell and ends when it is unbound from that cell.

- When a program is loaded into memory, it is organized into three areas of memory, called *segments*:

  -*text segment* (code segment)
  -*stack segment*, and
  -*heap segment*.

Segments in Memory

–**static**: global variable storage, permanent for the entire run of the program.

– **stack**: local variable storage (automatic, continuous memory).

–**heap**: dynamic storage (large pool of memory, not allocated in contiguous order).

Three segments in memory

code segment

-It is the place where the compiled code of the program resides.

-The remaining two areas of system memory is where storage may be allocated by the compiler for data storage

Stack segment

-stack is a special region of your computer's memory that stores temporary variables created by each function.

-stack is a *Last In First Out* (LIFO) storage device where new storage is allocated and deallocated at only one "end",called the Top of the stack

stable

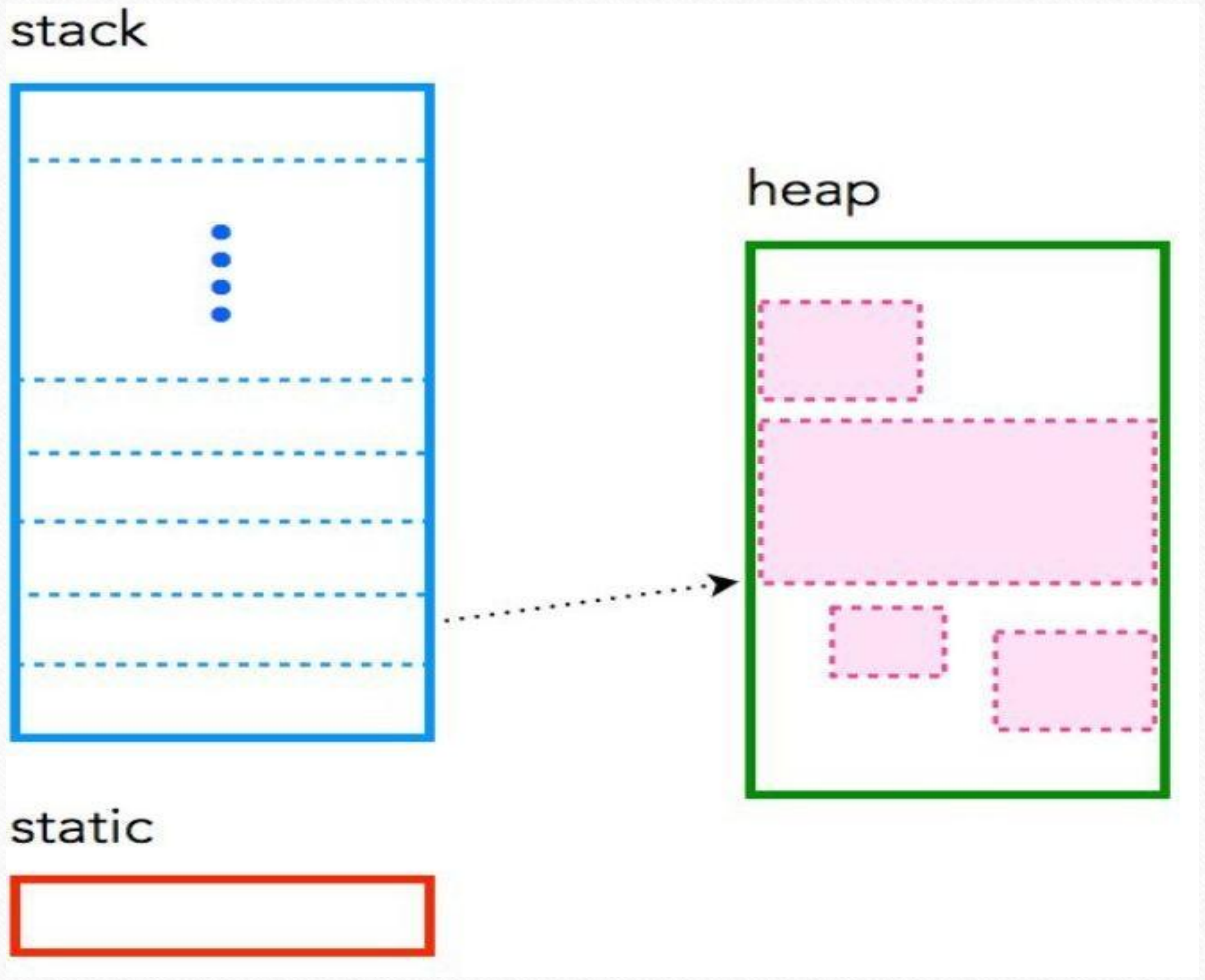
heap segment

-It provides more storage of data for a program; memory allocated in the heap remains in existence for the duration of a program

-global variables (storage class external), and dynamically allocated variables are stored on the heap

3 principal Storage Allocation mechanisms

- Static Allocation is appropriate when the storage requirements are known at compile time.
- Stack Allocation is appropriate when the storage requirements are not known at compile time,uses last-in, first-out discipline.
- Heap Allocation is appropriate where storage cells are allocated and deallocated dynamically


stack
heap
static

2.Stack Based storage Allocation

- Stack-based storage allocation is appropriate when the storage requirements are not known at compile time, but the requests obeys LIFO.
- Stack-based allocation is normally used in C/C++, Ada, Algol, and Pascal for local variables in a procedure and for procedure call information.
- It allows for recursive procedures, and also allocates data only when the procedure or function has been called ..
- Examples:

  -local variables in a procedure in C/C++, Ada, Algol, or Pascal

  -procedure call information (return address etc.)

Stack Based storage Allocation contd..

Advantages:

- Data is automatically pushed onto the stack at the beginning of a function call.
- Data is automatically popped from the stack when a function terminates making stack-based memory management highly reliable.
- In the absence of recursion it conserves storage b/c all subprograms share the same memory space for their locals.

Disadvantages:

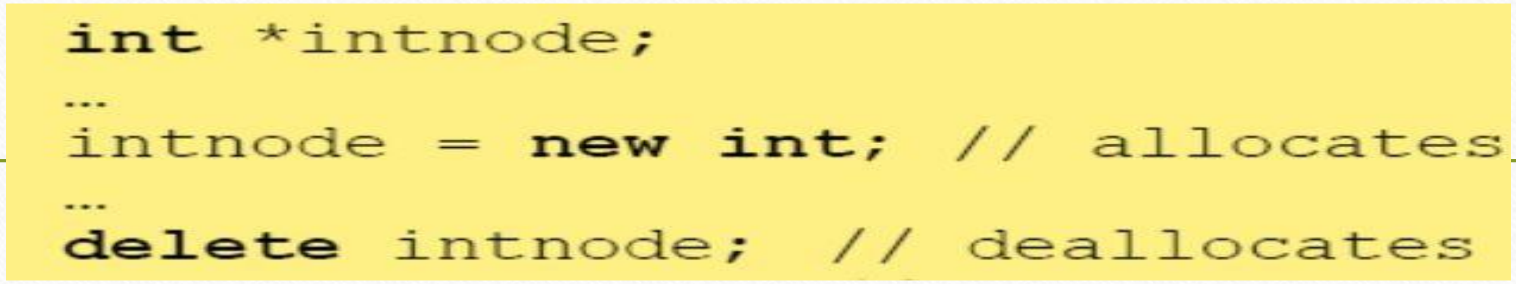
- Overhead of allocation and deallocation.
- Subprograms cannot be history sensitive.

3.Heap Based storage Allocation

- Heap is a region of memory where storage cells are allocated and deallocated dynamically

- Nameless memory cells that are **allocated and deallocated by "run-time instructions"**, specified by the programmer, which take effect during execution.
- These variables, which are allocated from and deallocated to the heap, can only be referenced through pointers or reference variables.
- **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;
...
intnode = new int; // allocates
...
delete intnode;  // deallocates
```

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

Heap Based storage Allocation concept

- Memory allocation is the process of assigning blocks of memory on request.
- The allocator receives memory from the operating system in a small number of large blocks that **it must divide** up to satisfy the requests for **smaller blocks**.
- **Any returned blocks must also made available for reuse**.
- When an allocation demand is made, the program searches the heap for a free block of at least the requested size.

Referencing environment

Referencing environment

- It is the **collection of all names that are visible in the statement**.
- A subprogram is active if its execution has begun but has not yet terminated
- In a **static-scoped language**, it is the **local variables plus all of the visible variables in all of the enclosing scopes**.
- In a **dynamic-scoped language**, the referencing environment is the **local variables plus all visible variables in all active subprograms**.

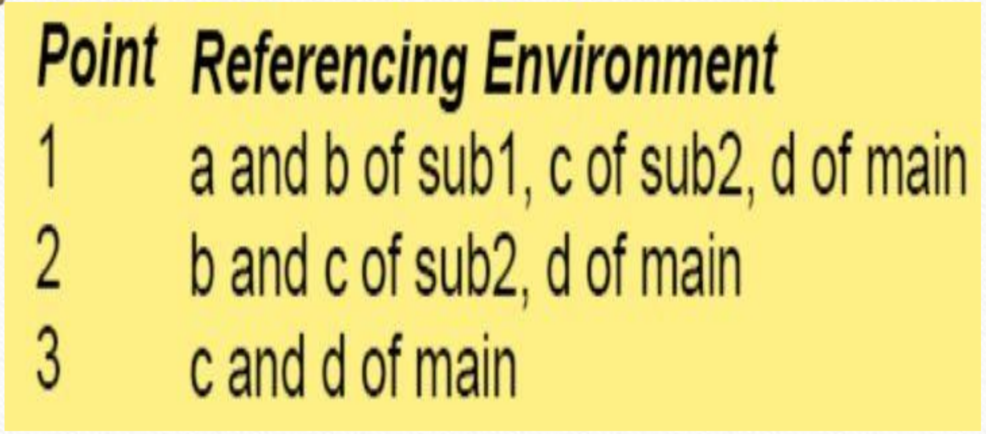
Ex, **static-scoped language**

Point	Referencing Environment
1	X and Y of Sub1, A & B of Example
2	X of Sub3, (X of Sub2 is hidden), A and B of Example
3	X of Sub2, A and B of Example
4	A and B of Example

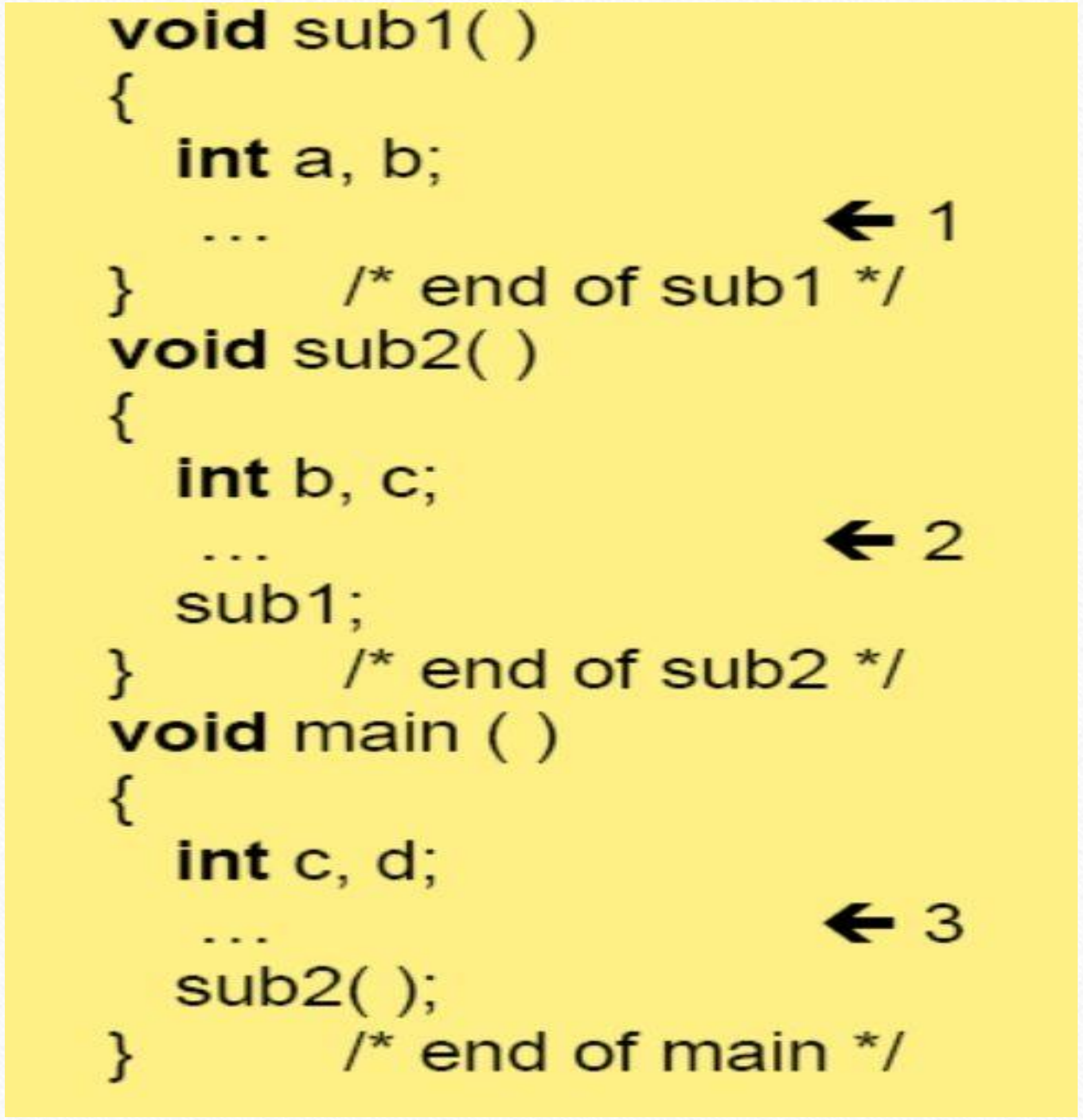
```
procedure Example is
    A, B : Integer;
    ...
    procedure Sub1 is
        X, Y : Integer;
        begin       -- of Sub1
        ...                                ← 1
         end        -- of Sub1
    procedure Sub2 is
        X : Integer;
        ...
        procedure Sub3 is
            X : Integer;
            begin   -- of Sub3
            ...                            ← 2
            end;    -- of Sub3
     begin   -- of Sub2
     ...                                   ← 3
      end;    { Sub2}
begin
    ...                                    ← 4
end;        {Example}
```

Ex, **dynamic-scoped language**

Consider the following program; assume that the only function calls are the following: *main* calls *sub2*, which calls *sub1*



Point	Referencing Environment
1	a and b of sub1, c of sub2, d of main
2	b and c of sub2, d of main
3	c and d of main



```
void sub1( )
{
  int a, b;
                              ← 1
   ...
}        /* end of sub1 */
void sub2( )
{
  int b, c;
                              ← 2
   ...
  sub1;
}        /* end of sub2 */
void main ( )
{
  int c, d;
                              ← 3
   ...
  sub2( );
}        /* end of main */
```

Binding of Referencing environment

- When a subroutine is passed as a parameter to a function, which referencing environment it uses when it is called :
  - **when the subroutine is passed as a parameter, or**
  - **when the subroutine is actually called**.
- There are two types of binding of referencing environment..
  - Deep binding corresponds to an early binding of the referencing environment
  - Shallow binding corresponds to a late binding of the referencing environment.