Programming Language Concepts

Programming Language Theory

Course Organization
Before Mid-term

- Now we're on Week 5, studying about PL Concepts.

- Based on Syllabus, we will study PL concepts one more week (Week 6), and move on to PL paradigm overview in Week 7.

- Due to the adjustment during holidays, Week 7 Online is actually Week 8 with the mid-term exam.

- Hence I'm considering to skip PL paradigm overview on Week 7.

New Plan for Week 7

- On Week 7 (Online), we will study little bit more about PL concepts.

- In addition, we will have ***mid-term summary session***.

- We need to discuss Details and Methods for the session.

    - So far I'm thinking of explaining more details about answers of assignment 1.

    - Maybe some Q & A session for pre-collected questions?

- Let me know if you have any request or suggestions.

Topics

- **Memory Management**
  - **Static Management**
- **Dynamic Management w/ Stack**
- Dynamic Management w/ Heap
- Scope Rule Implementation

Memory Management

- Memory management is one of the key functions of an interpreter.

- While a program is running, various information is produced and loaded from or stored to memory.

- e.g.) values of local variables, temporary values of expressions, arguments and return values of functions, and many others.

- Hence it is necessary to decide how to deal with such memory access of a programming language.

Terminology

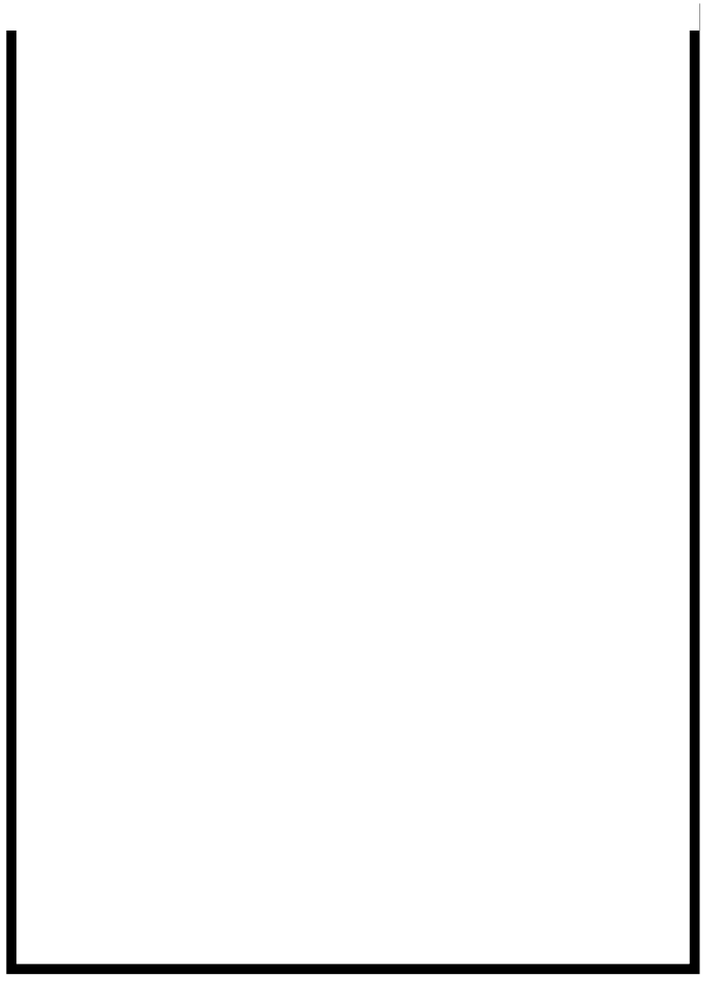
- We will use procedure, function, routine or subroutine as synonyms.
- They are all used to represent the concept of subprogram.
- In some PL, they might have different meaning (e.g. return value), but we will consider them as the same thing.
- Mostly, we will use ***procedure/function*** in this lecture.

Stack

- Stack is a data structure which literally **stacks** data.
- Consider the boxes as data.
- When you stacked the boxes, the **blue box** was placed before the **purple box**.
- You can’t simply pull out the **blue box**, until you pick the **purple box** first.

Stack

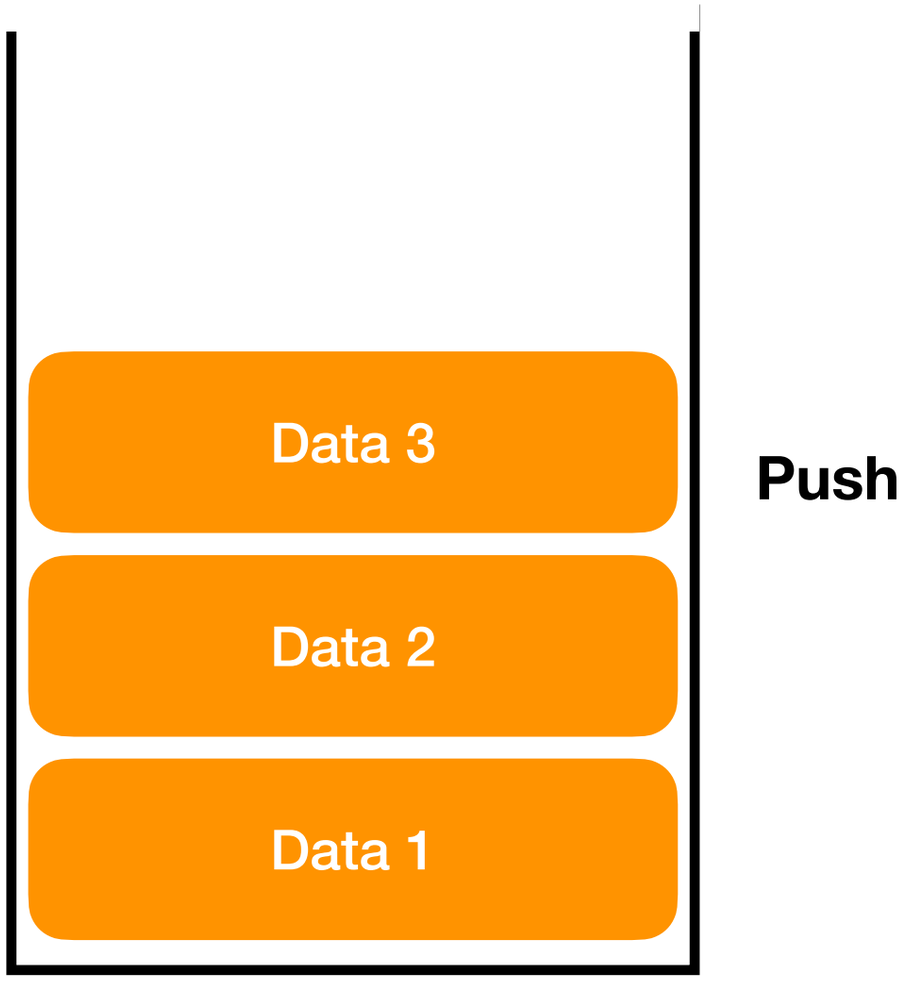
- Stack works similarly.
- The data inserted first, will be the one pulled out the last.
- It is called **LIFO** - **L**ast **I**n, **F**irst **O**ut.
- We can **push** the data to stack, and **pop** the last pushed data from the stack.



Stack

Stack

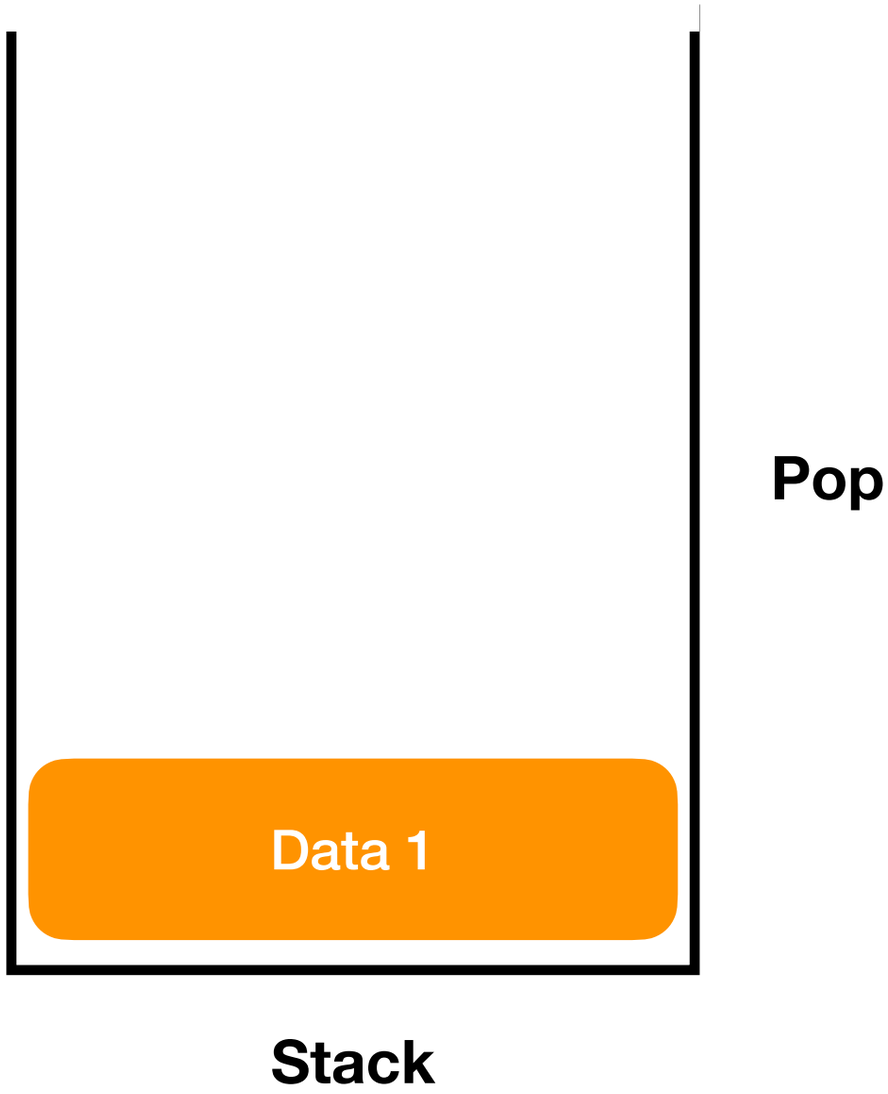
- Stack works similarly.
- The data inserted first, will be the one pulled out the last.
- It is called **LIFO** - **L**ast **I**n, **F**irst **O**ut.
- We can **push** the data to stack, and **pop** the last pushed data from the stack.



Stack

Stack

- Stack works similarly.
- The data inserted first, will be the one pulled out the last.
- It is called **LIFO** - **L**ast **I**n, **F**irst **O**ut.
- We can **push** the data to stack, and **pop** the last pushed data from the stack.



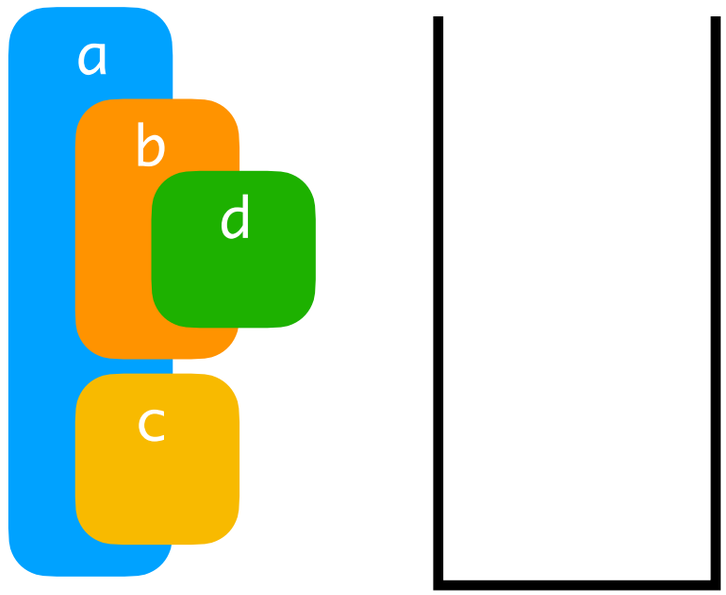
Stack and Procedure

- Stack naturally fits to procedures, since they are also activated in **LIFO**.

- The last procedure or block entered, is the first one which is exited.

- Environment changes can be also handled in this way.

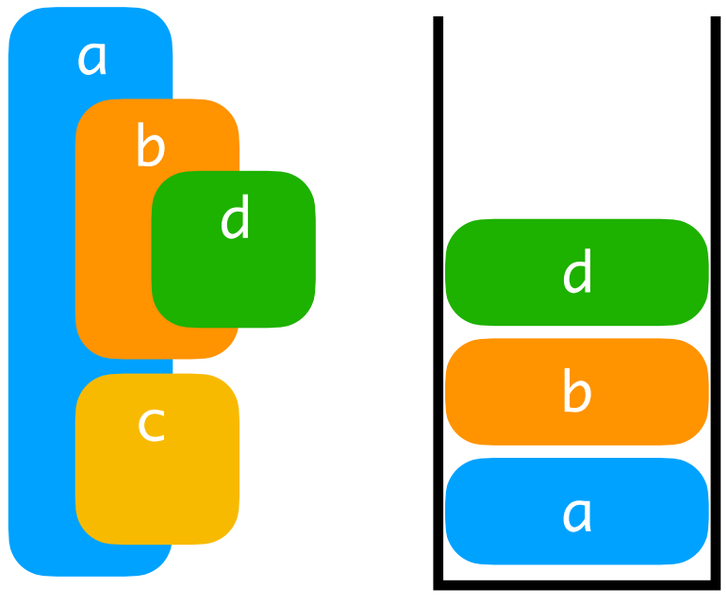
```
a(){
   b(){
      d(){
      }
      d();
   }
   c(){
   }
   b();
   c();
}
```



Stack and Procedure

- Stack naturally fits to procedures, since they are also activated in **LIFO**.
- The last procedure or block entered, is the first one which is exited.
- Environment changes can be also handled in this way.

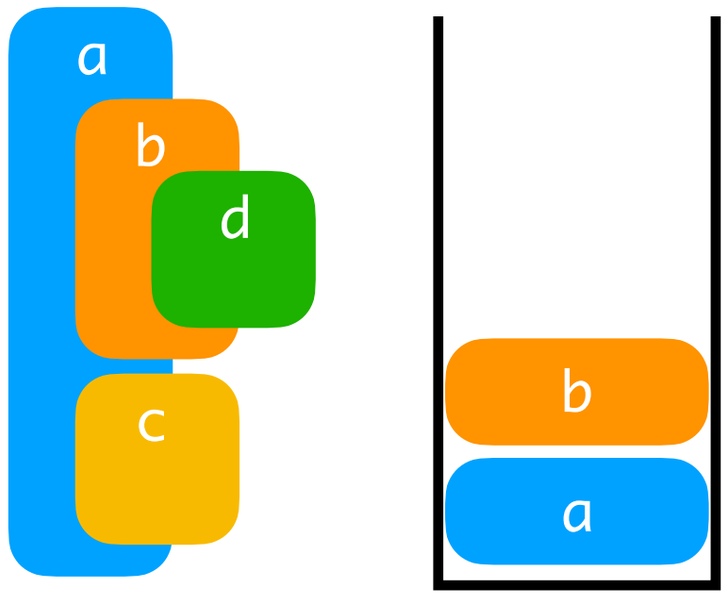
```
a(){
   b(){
      d(){
      }
      d();
   }
   c(){
   }
   b();
   c();
}
```



Stack and Procedure

- Stack naturally fits to procedures, since they are also activated in **LIFO**.
- The last procedure or block entered, is the first one which is exited.
- Environment changes can be also handled in this way.

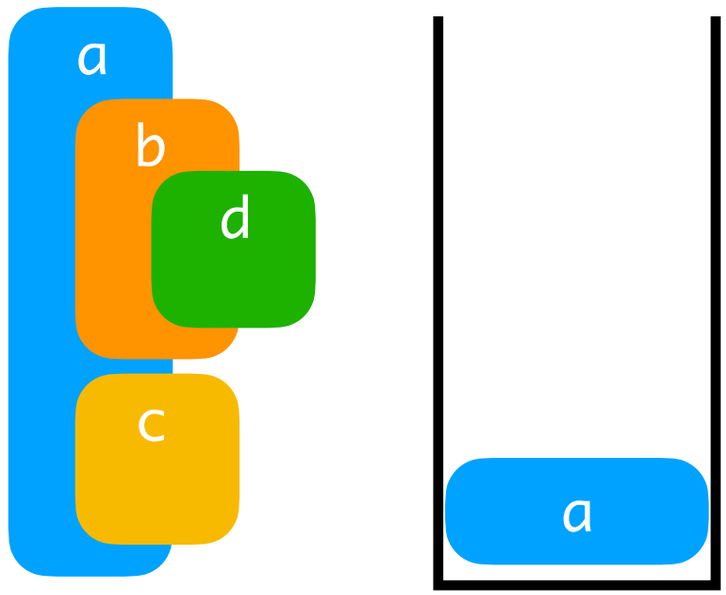
```
a(){
   b(){
      d(){
      }
      d();
   }
   c(){
   }
   b();
   c();
}
```



Stack and Procedure

- Stack naturally fits to procedures, since they are also activated in **LIFO**.
- The last procedure or block entered, is the first one which is exited.
- Environment changes can be also handled in this way.

```
a(){
   b(){
      d(){
      }
      d();
   }
   c(){
   }
   b();
   c();
}
```



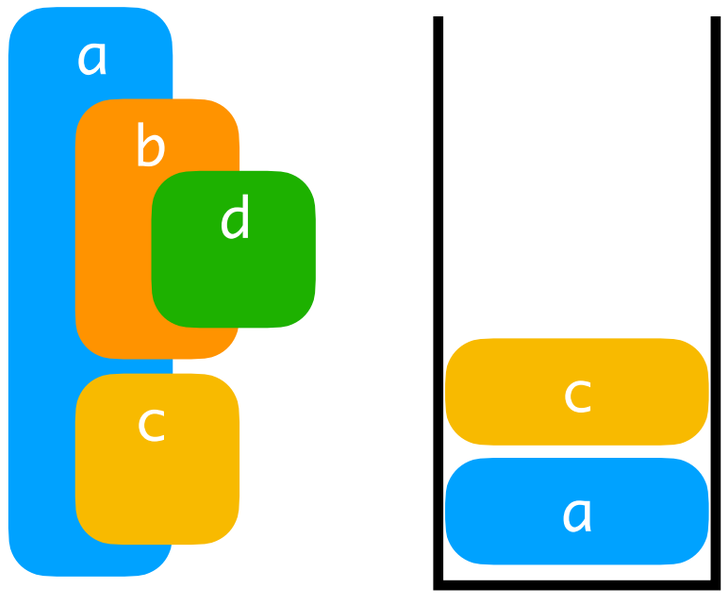
Stack and Procedure

- Stack naturally fits to procedures, since they are also activated in **LIFO**.

- The last procedure or block entered, is the first one which is exited.

- Environment changes can be also handled in this way.

```
a(){
   b(){
      d(){
      }
      d();
   }
   c(){
   }
   b();
   c();
}
```



Heap

- How about Heap?
- Heap is also a data structure related to priority queue or heap sort.
- However, in PL, ***Heap is just a certain place in memory*** which is allocated to a program.
- So there is zero connection with the data structure heap in this case.

Static Management

- Static memory management is performed by the compiler, before program execution.

- Statically allocated objects are located in a fixed zone of memory.

- These objects stay in there for the entire program execution.

Which can be allocated statically?

- ***Global Variables***: they are available for entire program.
- ***Object Code***: machine instructions generated by the compiler.
- ***Constants***: only if their values can be decided during compile time.
- ***Compiler-generated Tables***: they are used to runtime support of a program.

Without Recursion

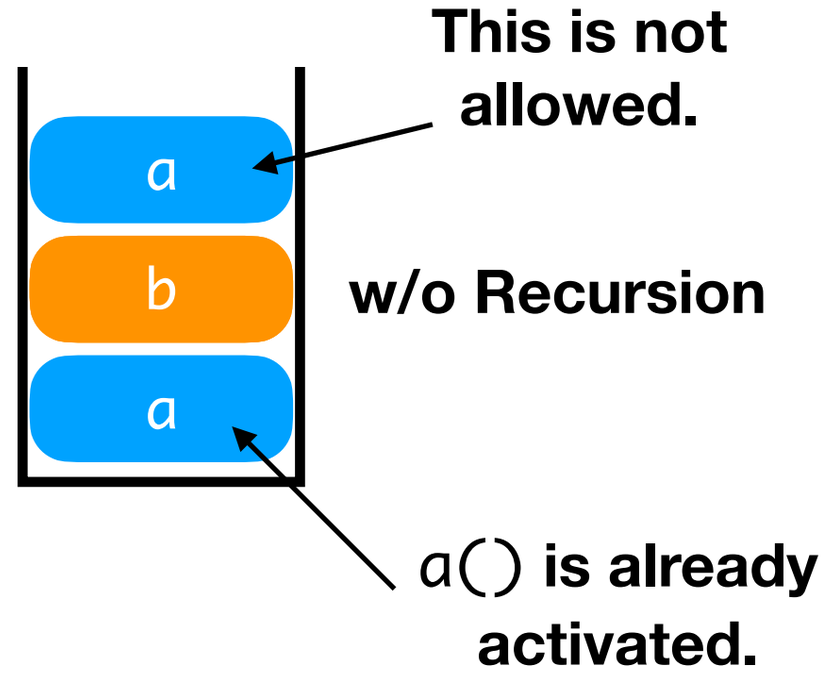
- Without recursion, more than one procedures cannot be activated at the same time.

- Hence it's possible to handle other components of PL statically.

  - e.g.) local variable, arguments, temporary values, return values and return address.

  - Because the same local variables can only be appeared in the stack once.

```
a(){
    b();
    a();
}
```



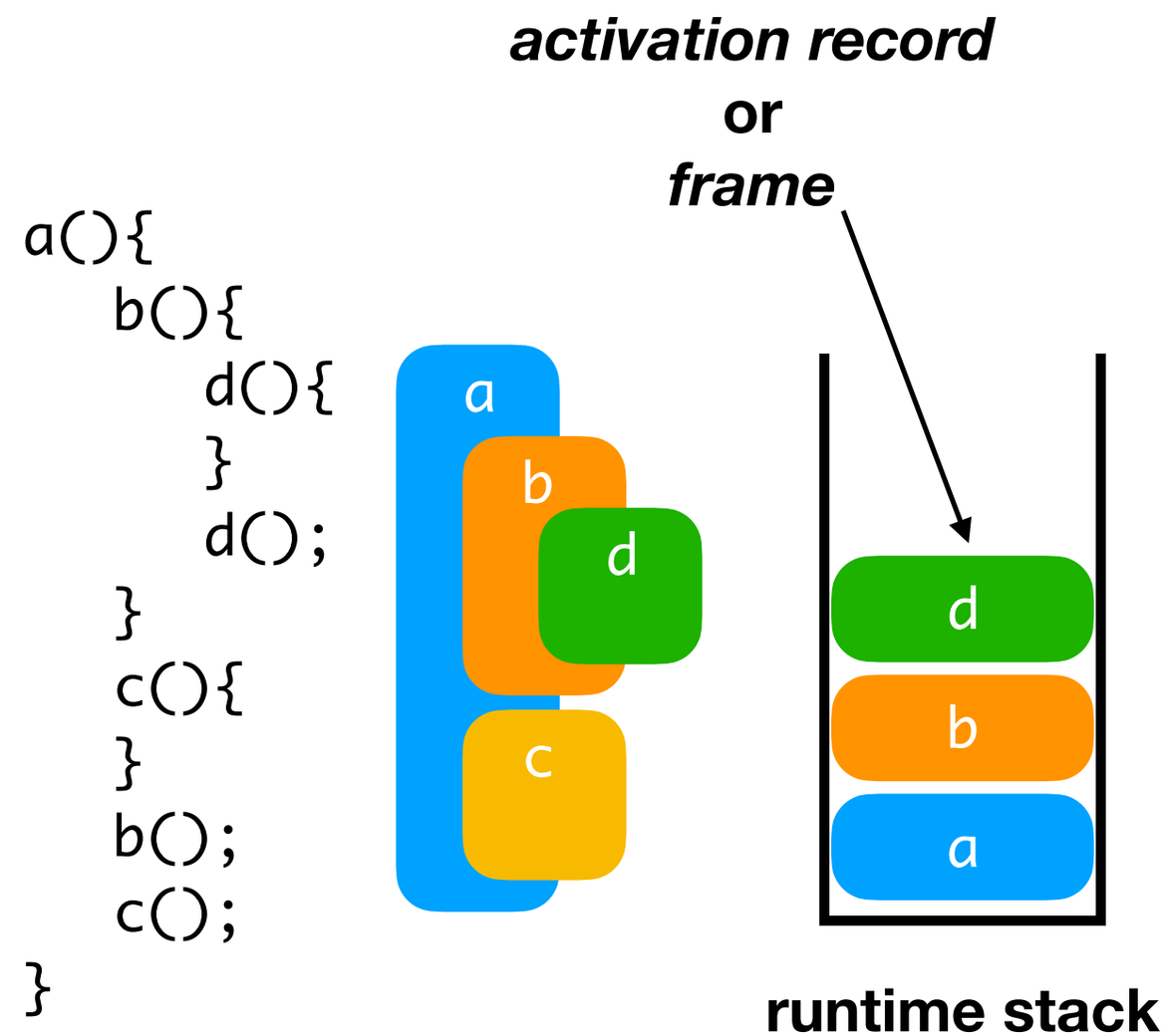
Dynamic Management

- Static management is not sufficient, since not all program components can be determined before runtime.

- We can use stack and heap for dynamic memory management.

Dynamic Management w/ Stack

- We have already seen the basic concepts using stack.
- Each memory space allocated to a procedure activation (or an inline block) is called ***activation record*** or ***frame***.
- The stack containing activation records is called ***runtime stack***.



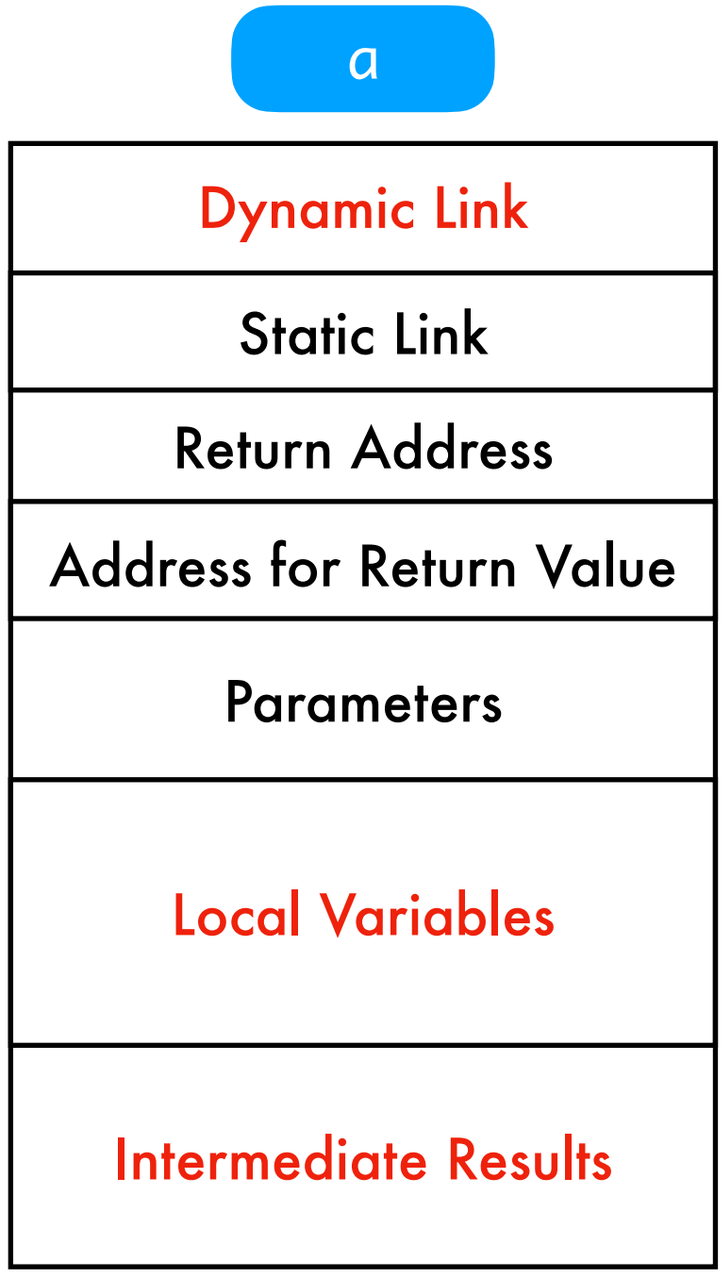
What's in Activation Record?

- Activation records for in-line blocks are much simpler than those of procedures.

- We will consider procedures only.

- Information in for in-line blocks' activation records is a subset of those in procedures' activation records.

- Red color means that it's common for in-line blocks and procedures.

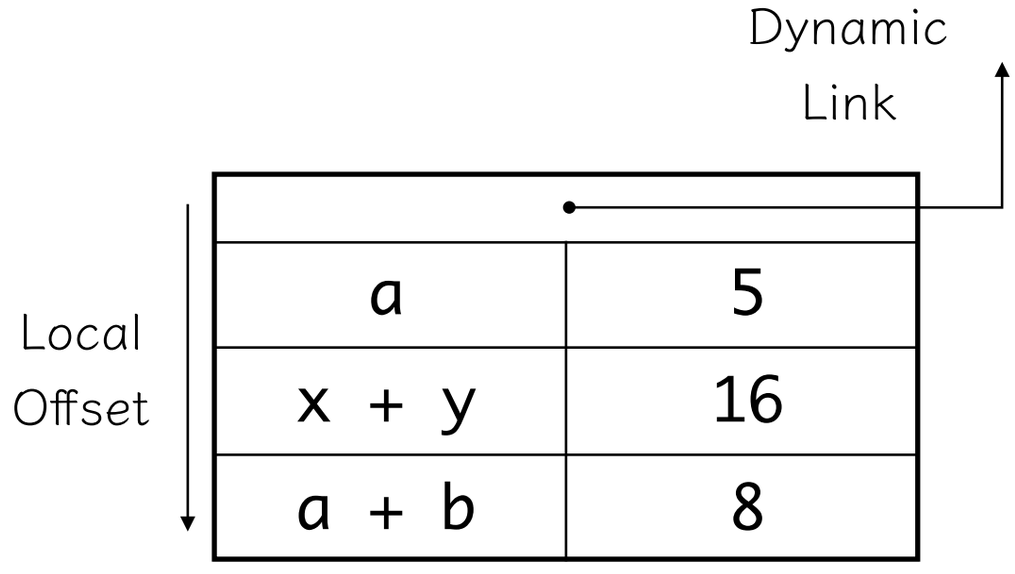


Dynamic Link

- An activation record with a local variable and intermediate results.
- **Dynamic Link** points to the start of *previous activation record* on the stack.
- This link is necessary since activation records have different sizes in general.
- From the start of activation record, we can use local offset to find a specific local variable.

In-line block

```
{
   int a = 5;
   b = (x+y) / (a+b);
}
```



Activation Record

Others

- **Return Address**: the next instruction after procedure call is finished.

- **Address for Return Value**: Return value will be stored in caller’s frame.

- **Parameters**: passed from caller.

Stack Management

- Activation records are stored to and removed from the stack at runtime.

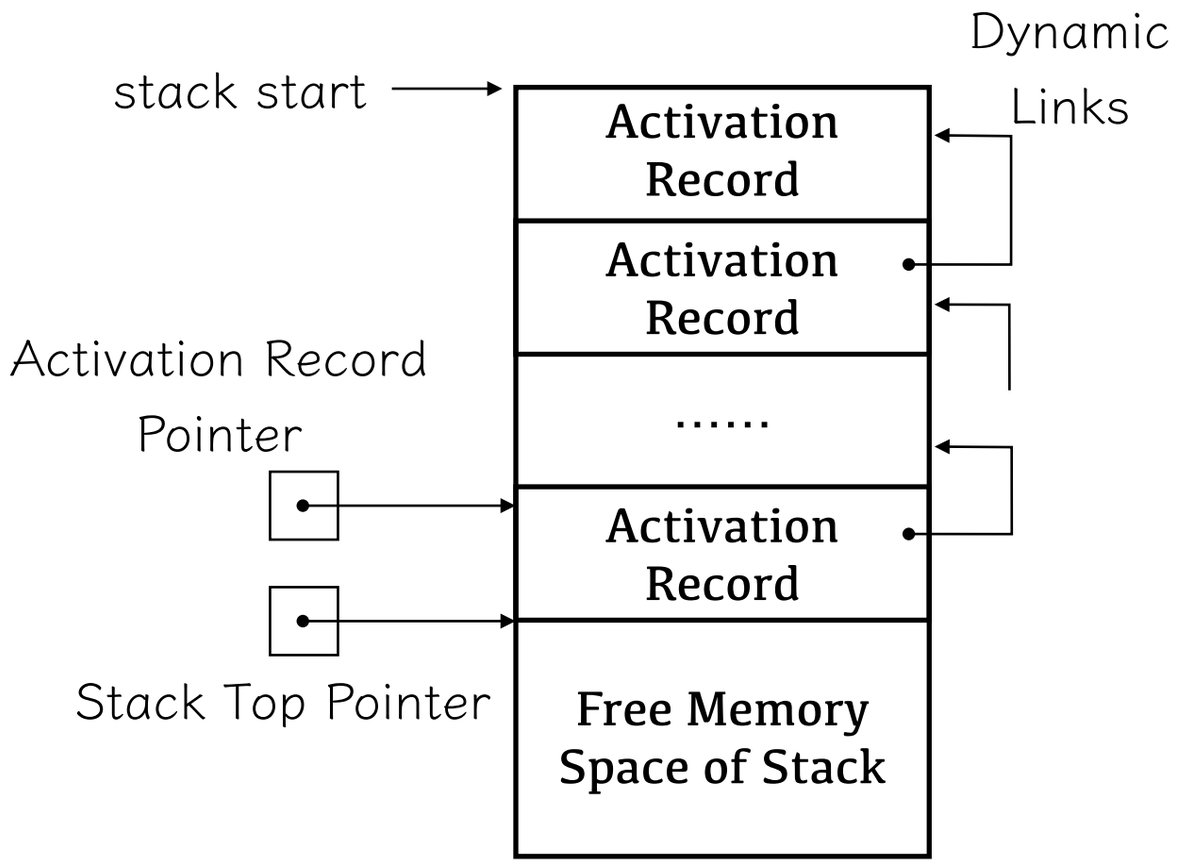
- When ***procedure B is called by procedure A***, both ***A (caller)*** and ***B (callee)*** manage such operations on the stack.

Stack Management

- Stack of Activation Records

- ***Activation Record Pointer***: Also called ***frame pointer*** or ***current environment pointer***.

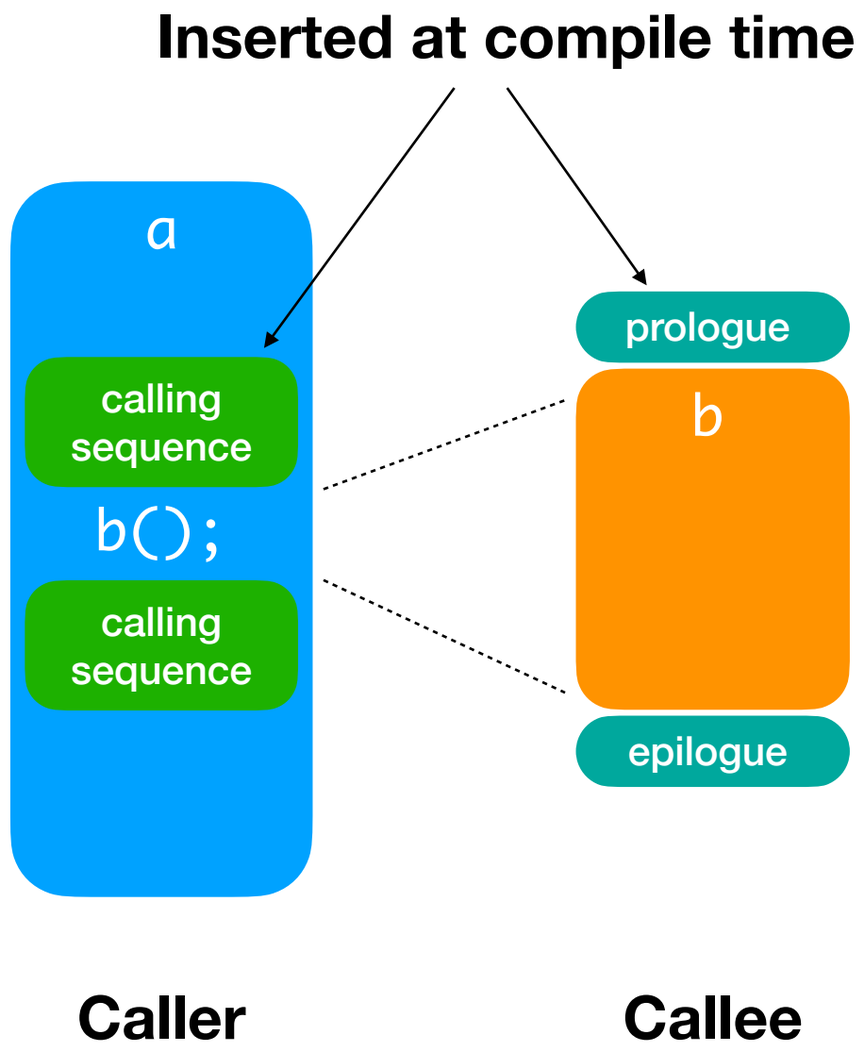
- **Stack Top Pointer**: it shows where is the start of “free space”.



Procedure Call

- ***Calling Sequence*** is inserted immediately before and after the procedure call by the compiler.

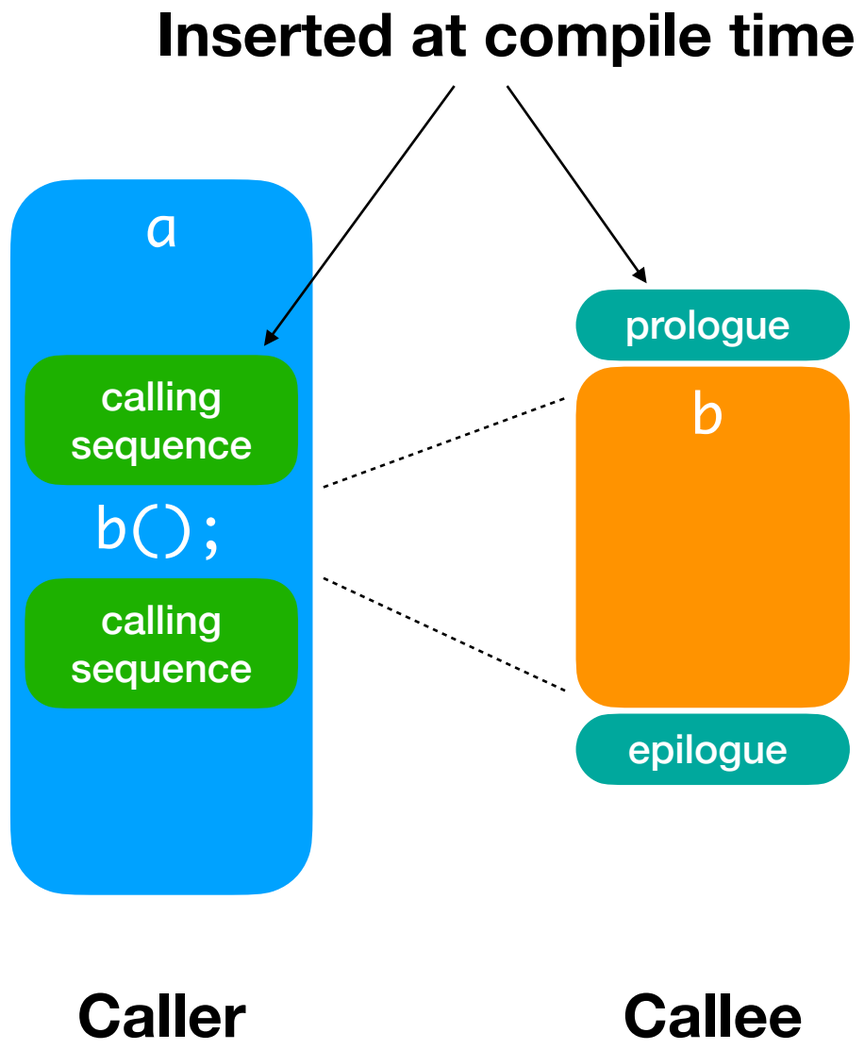
- ***Prologue*** and ***Epilogue*** are also inserted before and after procedure (callee) execution.



Procedure Call

- The exact operations of three code fragments depend on the compiler and implementation.

- To optimize the size of code, a large part of such code is given to callee.
  - Since it is inserted only once at declaration.
  - Otherwise, we need to add many times for each call.



Tasks before Procedure Call

- The following tasks will be done by ***calling sequence*** *and* ***prologue***.

- Program Counter Modification
- Stack Space Allocation
- Activation Record Pointer Modification
- Parameter Passing
- Register Save
- Initialization Code Execution

Tasks after Procedure Call

- The following tasks will be done by ***epilogue*** *and* ***calling sequence***.

- Update Program Counter
- Value Return
- Return of Registers
- Finalization Code Execution
- Stack Space Deallocation

Summary

- Stack and Heap
- Static Memory Management
- Dynamic Memory Management w/ Stack