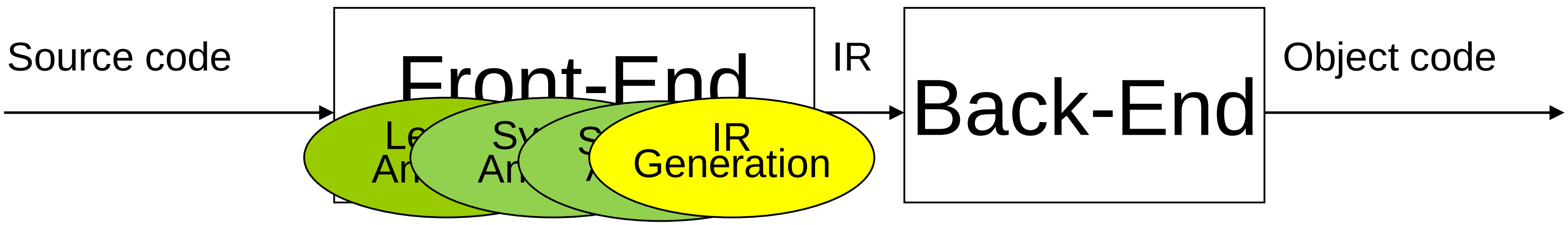
Compiler Design

Lecture 8: Runtime Environments

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book

Intermediate Representation Generation.



- IR Generation
  - Goal: Translate the program into the format expected by the compiler back-end.

Outline

- **Introduction**
- Runtime Environments
- Encoding in Runtime Environments
- Control-flow

What is IR Generation?

- Intermediate Representation Generation.
- The final phase of the compiler front-end.
- Translate the program into the format expected by the compiler back-end.
- Generated code need not be optimized
  - It will be handled by later passes.
- Generated code need not be in assembly
  - It can be handled by later passes.

Why Do IR Generation?

- Simplify certain optimizations.
  - Machine code has many constraints that inhibit optimization.
  - Working with an intermediate language makes optimizations easier and clearer.
- Have many front-ends into a single back-end.
  - gcc can handle C, C++, Java, Fortran, Ada, and many other languages.
  - Each front-end translates source to the GENERIC language.
- Have many back-ends from a single front-end.
  - Do most optimization on intermediate representation before emitting code targeted at a single machine.

Designing a Good IR

- IRs are like type systems – they're extremely hard to get right.
- Need to balance needs of high-level source language and low-level target language.
- Too high level: can't optimize certain implementation details.
- Too low level: can't use high-level knowledge to perform aggressive optimizations.
- Often have multiple IRs in a single compiler.

Steps

- Runtime Environments
- Three-Address Code IR

Outline

- Introduction
- **Runtime Environments**
- Encoding in Runtime Environments
- Control-flow

An Important Duality

- Programming languages contain high-level structures

- The physical computer only operates in terms of several primitive operations

An Important Duality

- High-level structures in programming languages:
  - Functions
  - Objects
  - Exceptions
  - Dynamic typing
  - …

An Important Duality

- Primitive operations of physical computer:
  - Arithmetic
  - Data movement
  - Control jumps

Runtime Environments

- We need to come up with a representation of these high-level structures using the low-level structures of the machine.

- A runtime environment is a set of data structures maintained at runtime to implement these high-level structures.
  - e.g. the stack, the heap, static area, virtual function tables, etc.

Runtime Environments

- Strongly depends on the features of both the source and target language. (e.g compiler vs. cross-compiler)

- Our IR generator will depend on how we set up our runtime environment.

Runtime Environments

- Need to consider
  - What do objects look like in memory?
  - What do functions look like in memory?
  - Where in memory should they be placed?

- There are no right answers to these questions.
  - Many different options and tradeoffs.
  - We will see several approaches.

Data Representations

- What do different types look like in memory?
- Machine typically supports only limited types:
  - Fixed-width integers: 8-bit, 16-bit- 32-bit, signed, unsigned, etc.
  - Floating point values: 32-bit, 64-bit, 80-bit IEEE 754.
- How do we encode our object types using these types?

Outline

- Introduction
- Runtime Environments
- **Encoding in Runtime Environments**
- Control-flow

Encoding Primitive Types

- Primitive integral types (byte, char, short, int, long, unsigned, uint16_t, etc.) typically map directly to the underlying machine type.
- Primitive real-valued types (float, double, long double) typically map directly to underlying machine type.
- Pointers typically implemented as integers holding memory addresses.
  - Size of integer depends on machine architecture; hence 32-bit compatibility mode on 64-bit machines.

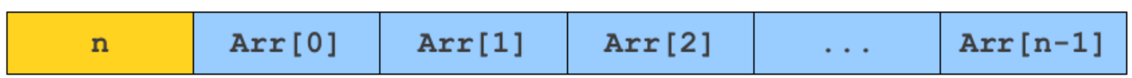
Encoding Arrays

- C-style arrays:
  - Elements laid out consecutively in memory.

`Arr[0]`	`Arr[1]`	`Arr[2]`	`...`	`Arr[n-1]`

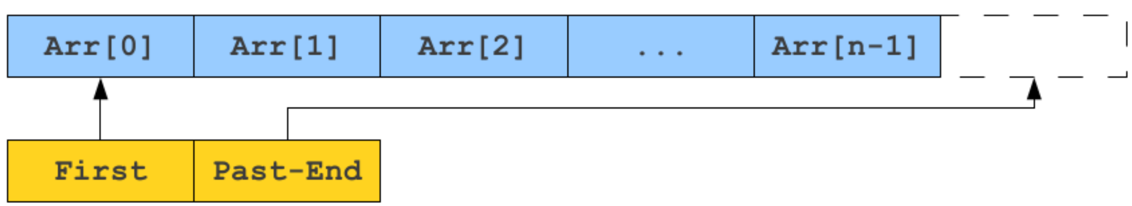
Encoding Arrays

- Java-style arrays:
  - Elements laid out consecutively in memory with size information prepended.



Encoding Arrays

- D-style arrays:
  - Elements laid out consecutively in memory
  - Array variables store pointers to first and past-the-end elements.



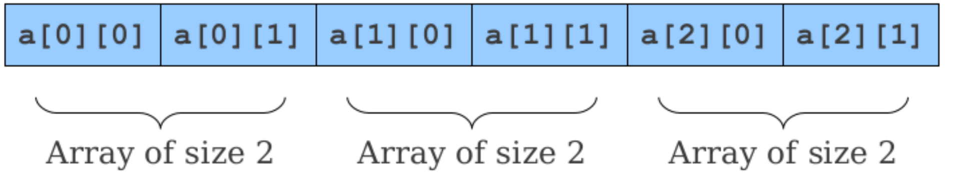
Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.

Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.
- C-style arrays:

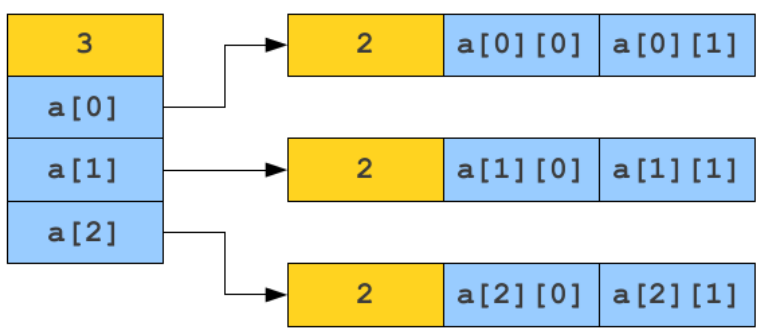
```
int a[3][2];
```



Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.
- Java-style arrays:

    int[][] a = new int [3][2];



Encoding Functions

- Many questions to answer:
  - What does the dynamic execution of functions look like?
  - Where is the executable code for functions located?
  - How are parameters passed in and out of functions?
  - Where are local variables stored?

- The answers strongly depend on what the language supports.

The Procedure

- Procedures are the key to building large systems; they provide:
  - Control abstraction: well-defined entries & exits.
  - Name Space: has its own protected name space.
  - External Interface: access is by name & parameters.
- Requires system wide-compact:
  - Broad agreement on memory layout, protection, etc…
  - Must involve compiler, architecture, OS

The Procedure

- Establishes the need for private context:
  - Create a run-time “record” for each procedure to encapsulate information about control & data abstractions.
- Separate compilation:
  - Allows us to build large systems; keeps compile-time reasonable

The Procedure: a more detailed view

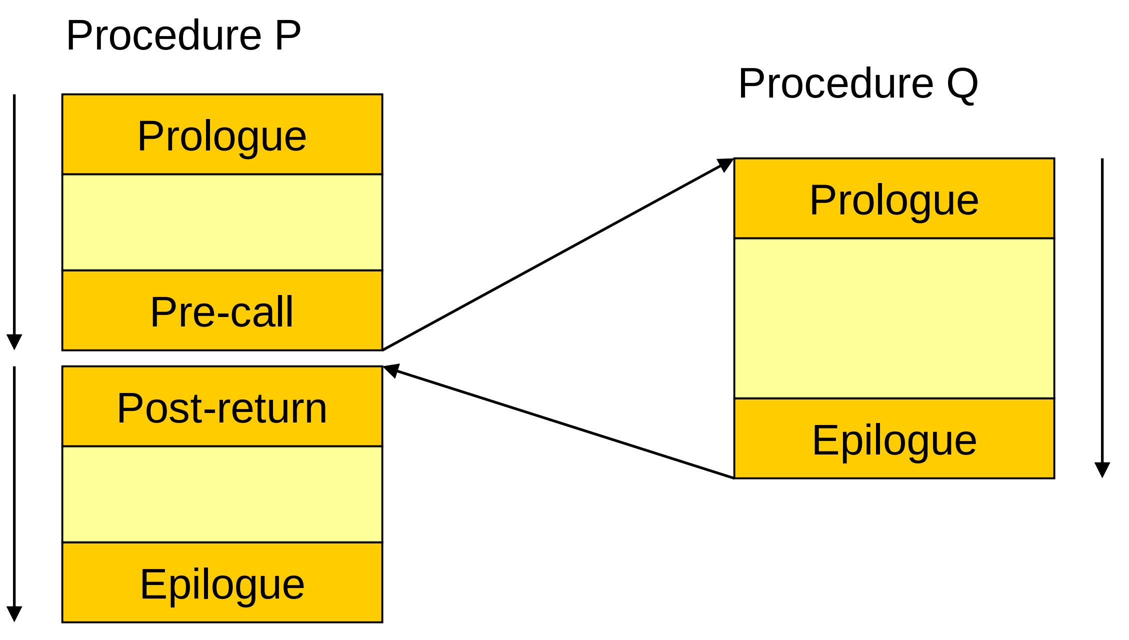
- A procedure is a collection of commands.
- The commands in a procedure are running line by line.

Outline

- Introduction
- Runtime Environments
- Encoding in Runtime Environments
- **Control-flow**

The linkage convention

- Procedures have well-defined control-flow behaviour:
  - A protocol for passing values and program control at procedure call and return is needed.
  - The linkage convention ensures that procedures inherit a valid run-time environment and that they restore one for their parents.
- Linkages execute at run-time.
- Code to make the linkage is generated at compile-time.

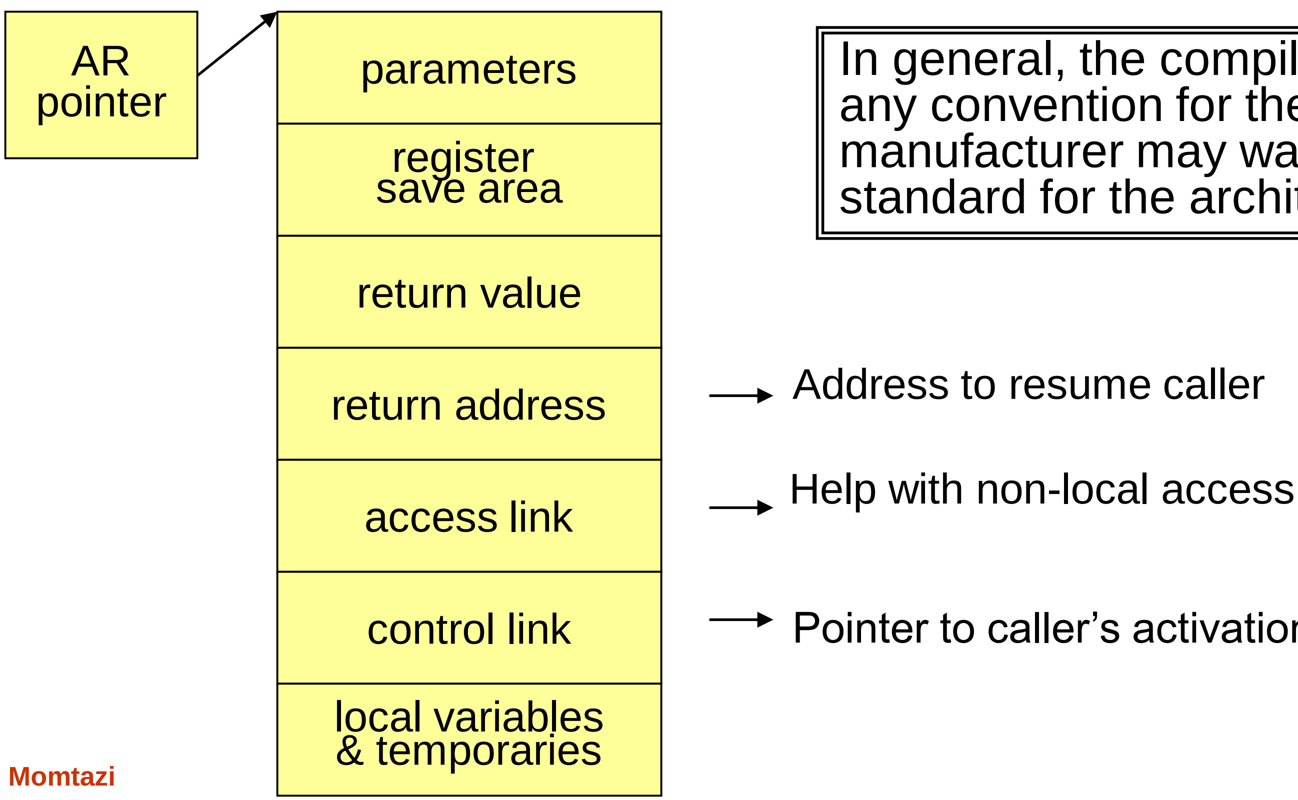


Control-flow as a Tree

- The control-flow in a program can be considered as an tree: Activation Tree.
- The root is the main program.
- The control-flow of the program is derived by search over this tree.

Storage Organisation: Activation Records

- Local variables require storage during the lifetime of the procedure invocation at run-time.
- The compiler arranges to set aside a region of memory for each individual call to a procedure (run-time support): <u>Activation Record</u>



In general, the compiler is free to choose any convention for the AR. The manufacturer may want to specify a standard for the architecture.

Activation Records are also known as stack frames.

Procedure linkages

Caller (pre-call):
- allocate AR
- evaluate and store parameters
- store return address
- store self's AR pointer
- set AR pointer to child
- jump to child

Callee (prologue):
- save registers, state
- extend AR for local data
- get static data area base address
- initialise local variables
- fall through to code

Callee (epilogue):
- store return value
- restore registers, state
- unextend basic frame
- restore parent's AR pointer
- jump to return address

Caller (post-return):
- copy return value
- deallocate callee's AR
- restore parameters (if used for call-by reference)

the procedure linkage convention is a machine-dependent contract between the compiler, the OS and the target machines to divide clearly responsibility

Parameters in Procedure linkages

- Parameters:
  - Formal
    - The arguments used in definition of a procedure
  - Actual
    - The arguments used when calling a procedure

- When calling a procedure, the actual parameters are passed to the procedure. The formal parameters are replaced with the actual parameters.

```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```

Activation Trees

- An activation tree is a tree structure representing all of the function calls made by a program on a particular execution.
  - Depends on the runtime behavior of a program; can't always be determined at compile-time.
  - (The static equivalent is the call graph).
- Each node in the tree is an activation record.
- Each activation record stores a control link to the activation record of the function that invoked it.

Activation Trees

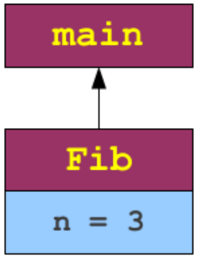
```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```

Activation Trees

```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```

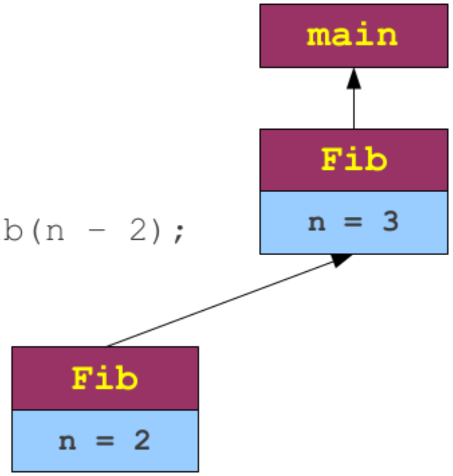
Activation Trees

```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```



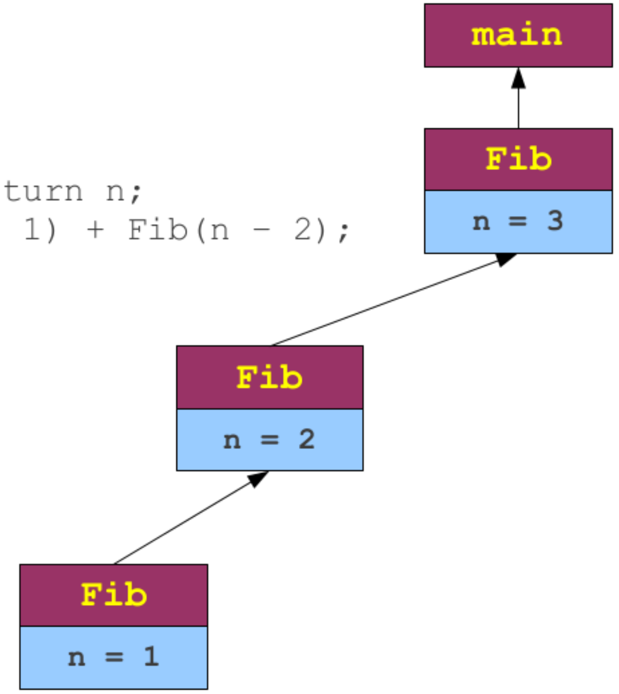
Activation Trees

```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```



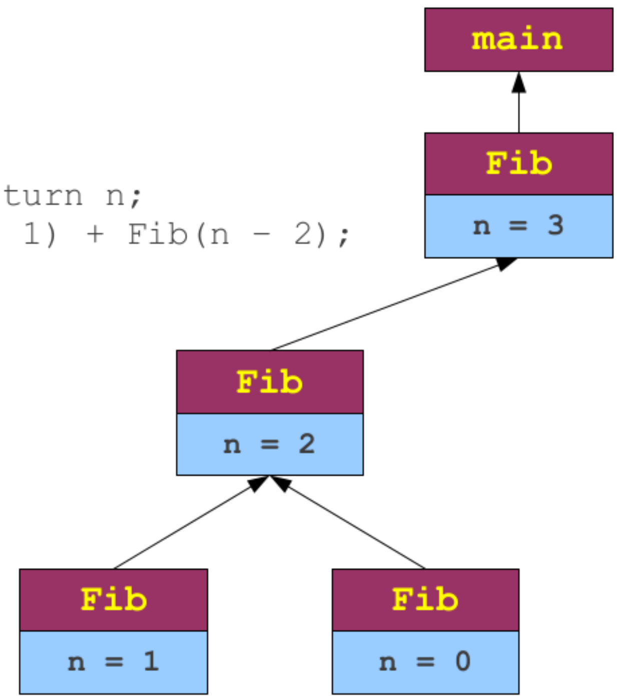
Activation Trees

```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```



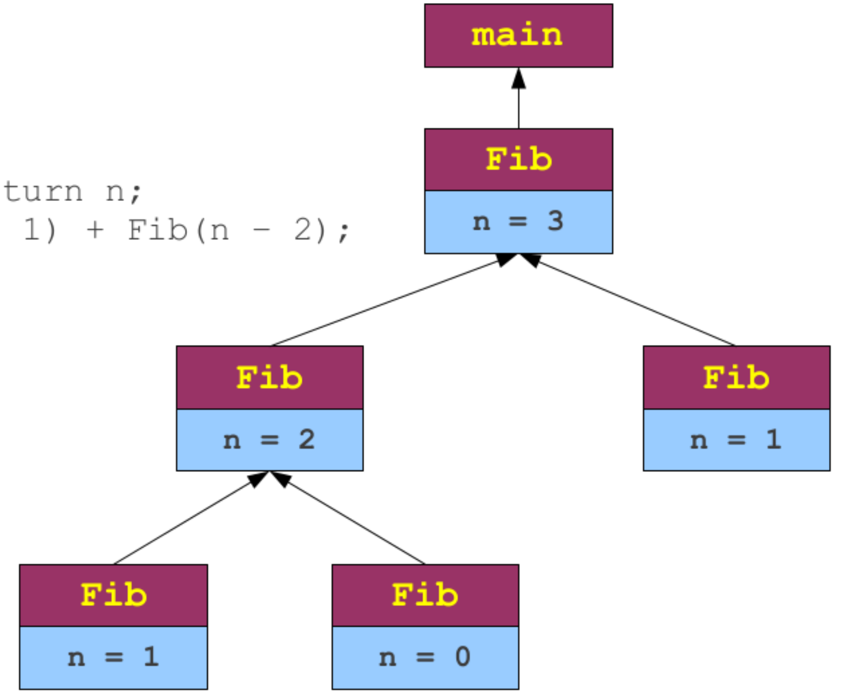
Activation Trees

```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```



Activation Trees

```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```



Placing run-time data structures

Single logical address space:

- Code, static, and global data have known size.
  - They are defined at compile time.
- Heap & stack are used for allocating dynamic memory
  - They are defined at runtime
- Heap & stack grow towards each other
- The usage of heap and stack depends on the way the variables needs to be allocated: first-in-first-out or last-in-first-out

Stack of Activation Records

- Function calls are often implemented using a stack of activation records (or stack frames).
- Calling a function pushes a new activation record onto the stack.
- Returning from a function pops the current activation record from the stack.

- The runtime stack is an optimization of the activation tree.

Why Can We Optimize the Stack?

- Once a function returns, its activation record cannot be referenced again.
  - Every activation record has either finished executing or is an ancestor of the current activation record.
- We don't need to store old nodes in the activation tree.
  - We don't need to keep multiple branches alive at any one time.
- These are not always true!

Run-time storage organisation

- The compiler must ensure that each procedure generates an address for each variable that it references:
  - Static and Global variables:
    - Addresses are allocated statically (at compile-time). (relocatable)
  - Procedure local variables:
    - Put them in the activation record if: sizes are fixed and values are not preserved.
  - Dynamically allocated variables:
    - Usually allocated and deallocated explicitly.
    - Handled with pointers.

Links in Activation Records

- The access link of a function is to access variables from the outer scope.
  - Also called static link.

- The control link of a function is a pointer to the activation record of the function that called it.
  - Also called dynamic link.
  - Used to determine where to resume execution after the function returns.

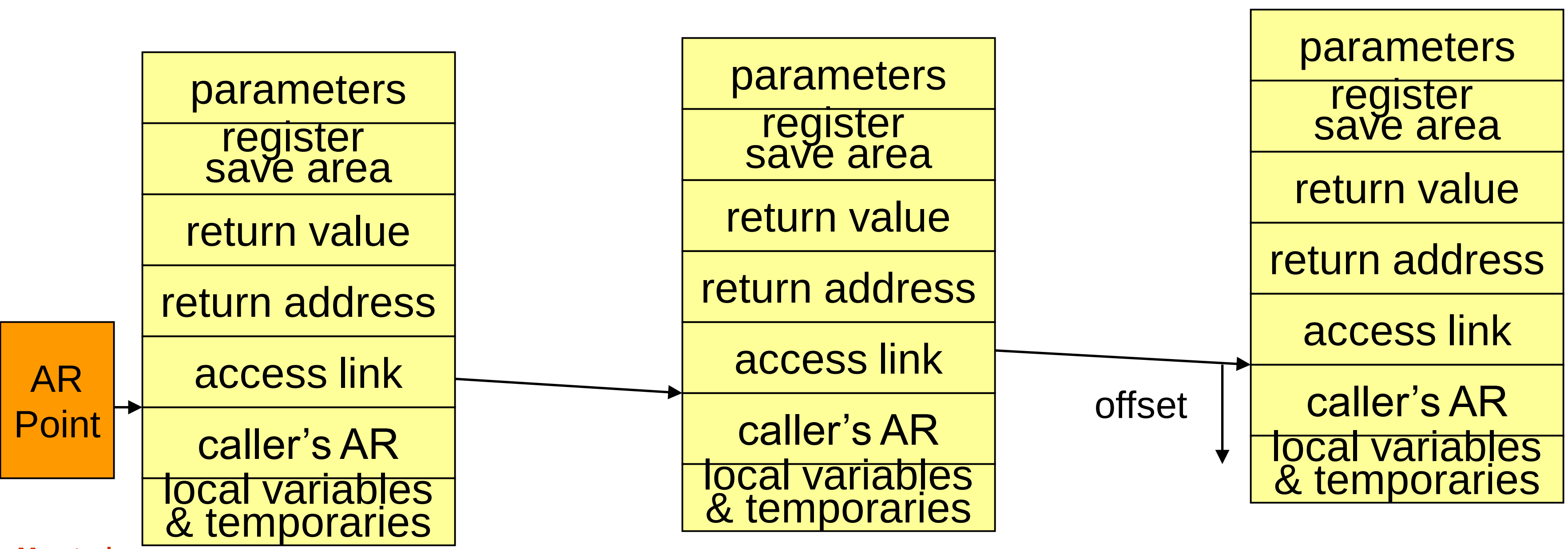
parameters
register save area
return value
return address
access link
control link
local variables & temporaries

Addressing non-local data

- In a language that supports nested lexical scopes, the compiler must provide a mechanism to map variables onto addresses.
- The compiler knows current level of lexical scope and of variable in question and offset (from the symbol table).
- Needs code to:
  - Track lexical ancestry (not necessarily the caller) among ARs.
  - Interpret difference between levels of lexical scope and offset.
- Two basic mechanisms:
  - Access links
  - Global display.

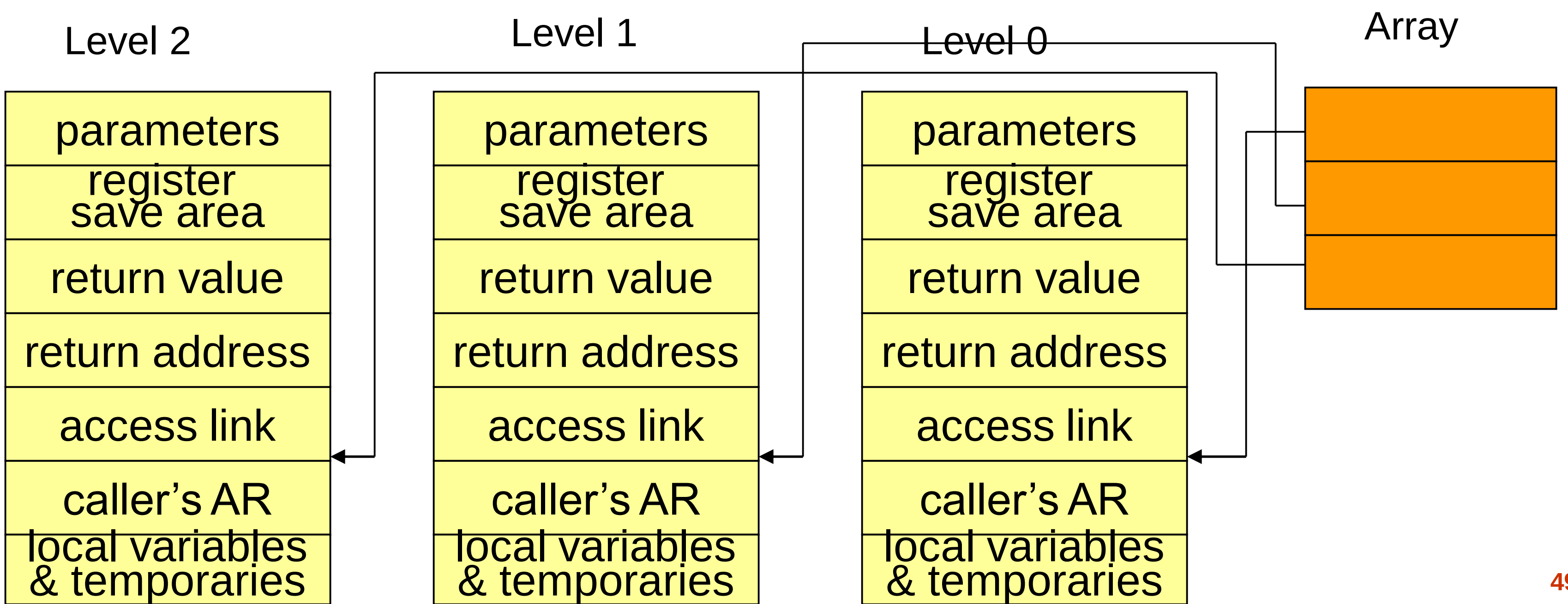
Access Links

- Idea: Each AR contains a pointer to its lexical ancestor.
- Compiler needs to emit code to find lexical ancestor (if caller's scope=callee's scope+1 then it is the caller; else walk through the caller's ancestors)
- Cost of access depends on depth of lexical nesting



Global Display

- Idea: keep a global array to hold ARPs for each level.
- Compiler needs to emit code (when calling and returning from a procedure) to maintain the array.
- Cost of access is fixed (table lookup + AR)
- Display vs access links trade-off. conventional wisdom: use access links when tight on registers; display when lots of registers.



Calling Sequence

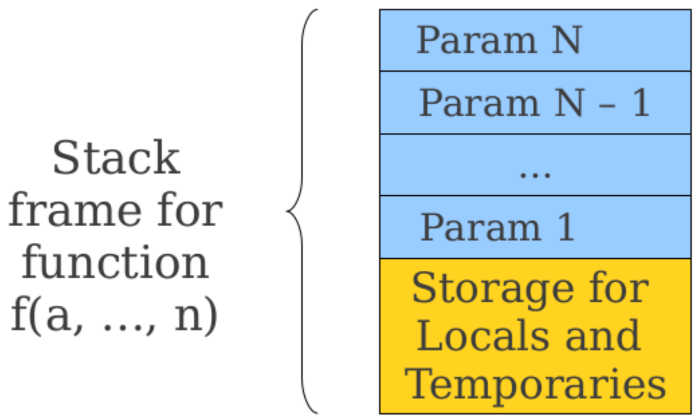
- Assumptions
  - “Once a function returns, its activation record cannot be referenced again.”
  - “Every activation record has either finished executing or is an ancestor of the current activation record.”

Calling Sequence

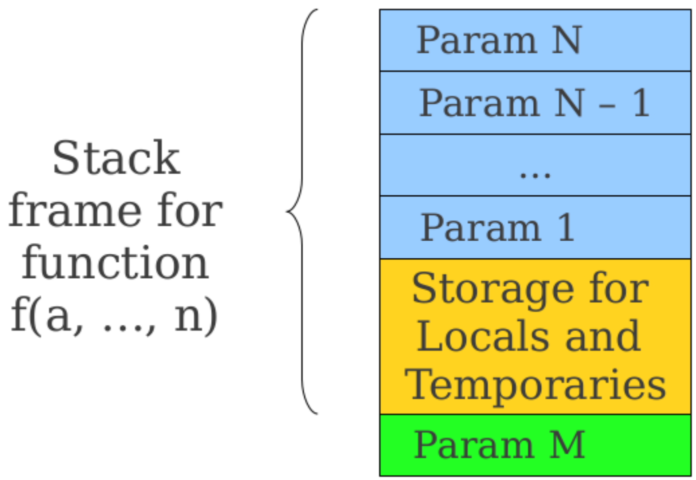
- Caller responsible for pushing and popping space for callee's arguments.
- Callee responsible for pushing and popping space for its own temporaries.

- Example:
  - Function f() is caller and function g() is callee.

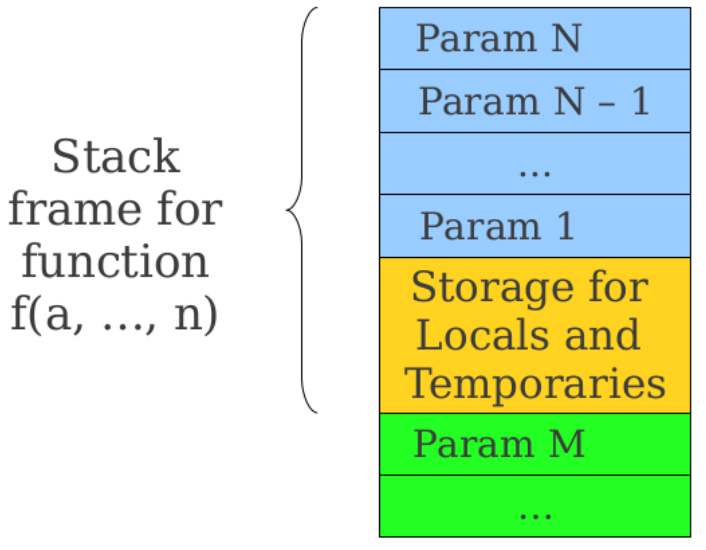
Calling Sequence



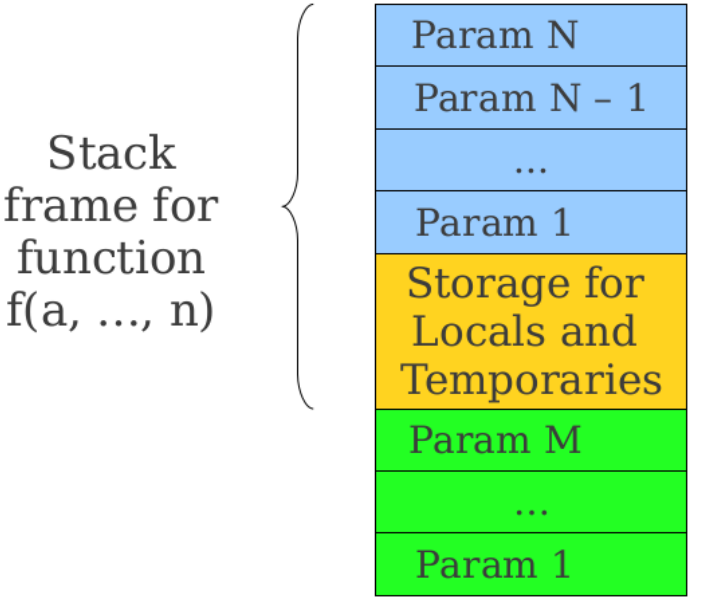
Calling Sequence



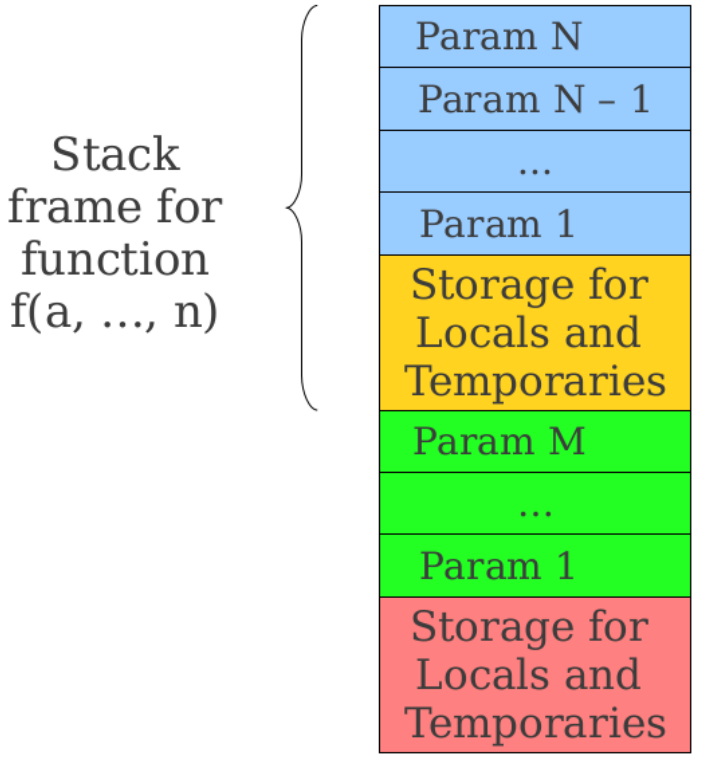
Calling Sequence



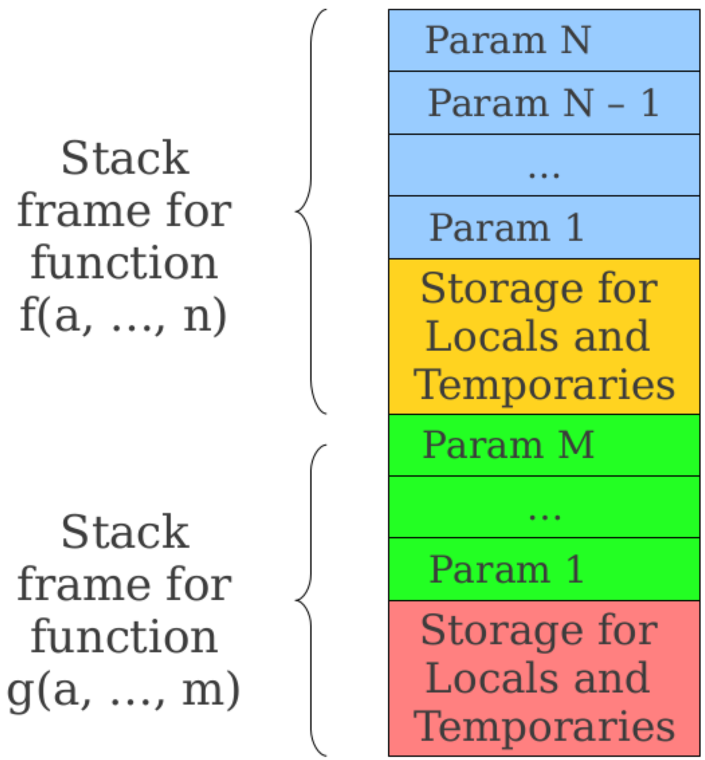
Calling Sequence



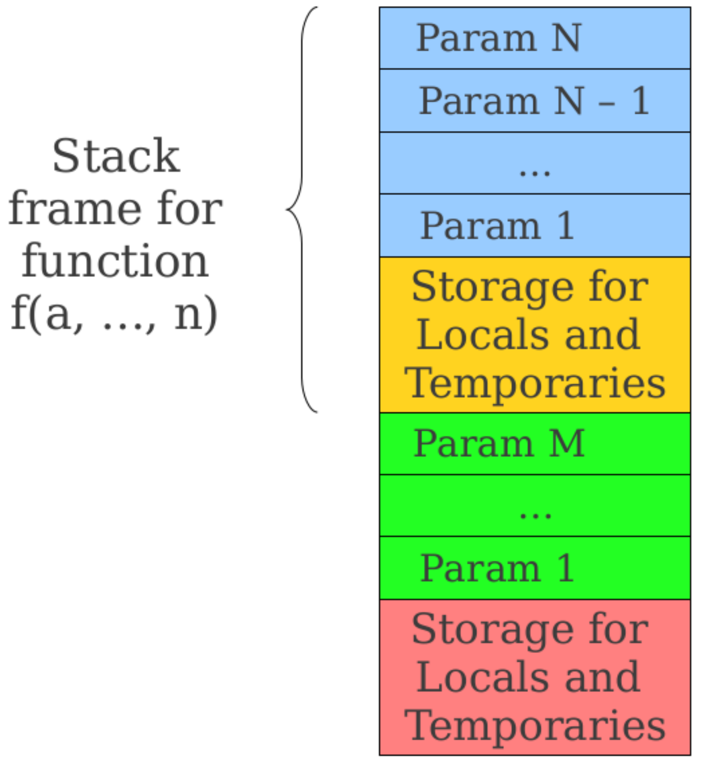
Calling Sequence



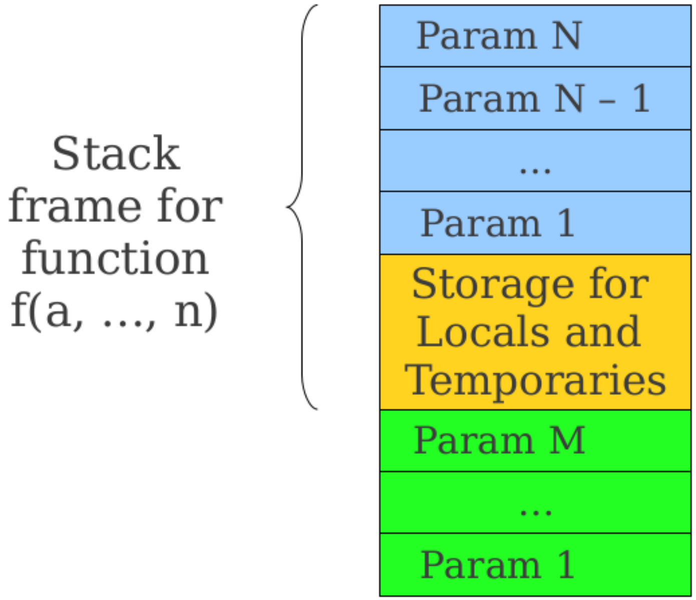
Calling Sequence



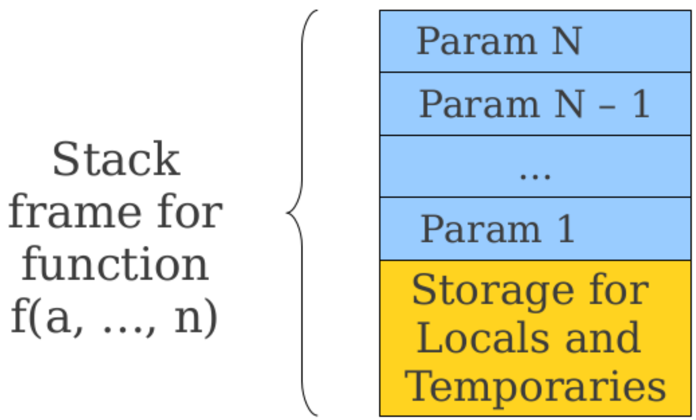
Calling Sequence



Calling Sequence



Calling Sequence



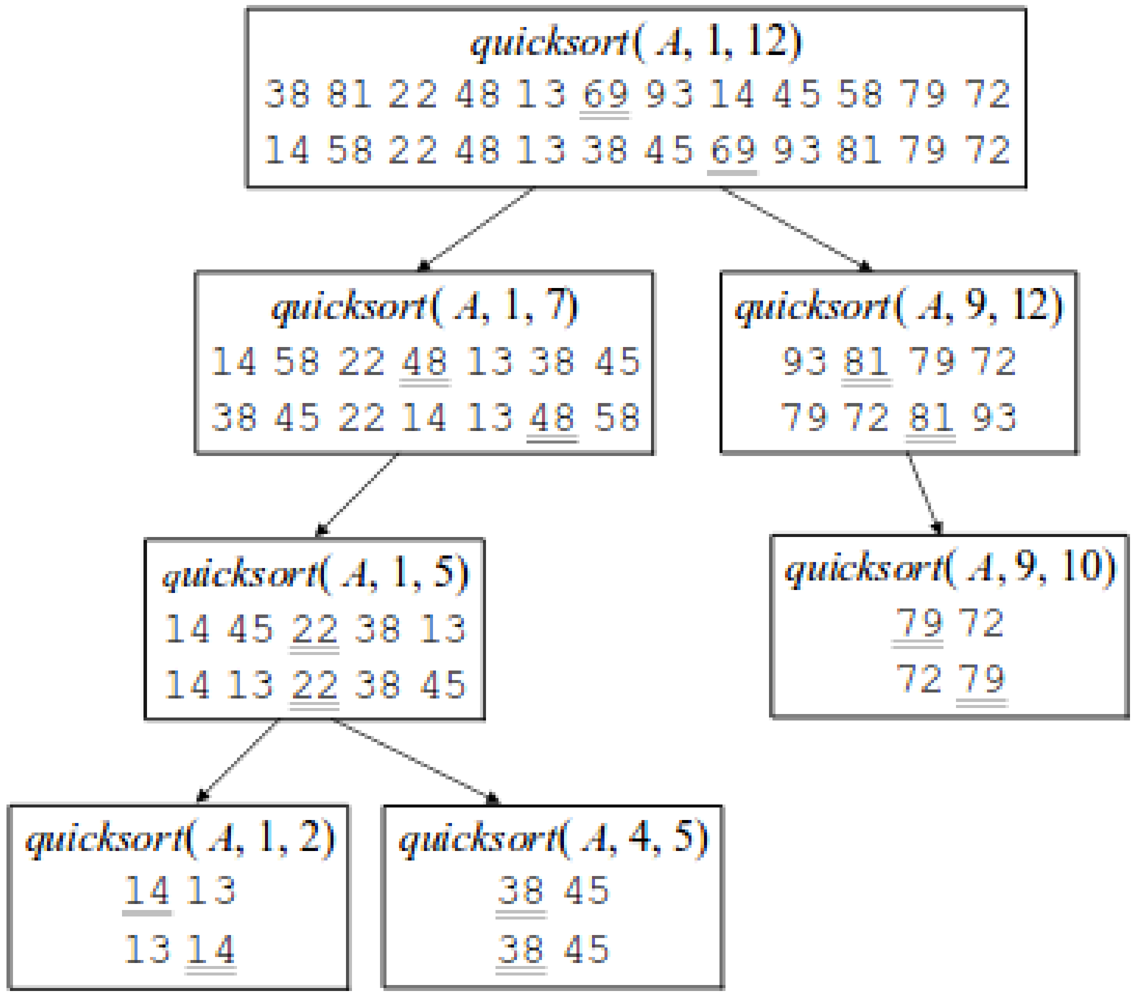
Example

- Quicksort Code:

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Example

- Data:



Example

- Quicksort Execution:
  - push quicksort (1,12)
  - push partition (1,12)
  - pop partition (1,12)
  - …

Summary

- An activation records arranges to set aside a region of memory for each individual call to a procedure
- An Activation tree is a tree structure representing all of the function calls made by a program on a particular execution.
- The runtime stack is an optimization of the activation tree stack.
- Activation records logically store a control link to the calling function and an access link to the function in which it was created.

Question?