Runtime Environments

Lecture 19
Section 7.1 - 7.4

Rokan Uddin Faruqui

Associate Professor
Dept of Computer Science and Engineering
University of Chittaong, Bangladesh
Email: *rokan@cu.ac.bd*

Runtime Environment

Imagine the environment where the target program code is being executed

Issues dealt with here:

- allocation of storage
- access to variables and data
- memory management: stack allocation, heap management, (and garbage collection)

Runtime Environment

Imagine the environment where the target program code is being executed

Issues dealt with here:

- allocation of storage
- access to variables and data
- memory management: stack allocation, heap management, (and garbage collection)

Runtime Environment

Imagine the environment where the target program code is being executed

Issues dealt with here:

- allocation of storage
- access to variables and data
- memory management: stack allocation, heap management, (and garbage collection)

Scanner (lexical analysis)

Parser (syntax analysis

Semantic analysis and intermediate code generation

Machine-independent code improvement (optional)

Target code generation

Machine-specific code improvement (optional)

Figure: Phases of compilation

Outline

1. Storage organization
2. Stack Allocation of Space
3. Non-local data
4. Heap Management

Outline

Static vs Dynamic Storage Allocation

int a;

Static means "at compile time"

Dynamic means "at runtime"

Stack has data local to (procedure) call.

- Layout known statically
- Managed automatically by generated code

Heap has data that survives between (procedure) calls.

- Layout not known statically
- Managed dynamically by garbage collector

Static vs Dynamic Storage Allocation

Static means “at compile time“

Dynamic means “at runtime“

Stack has data local to (procedure) call.

- Layout known statically
- Managed automatically by generated code

Heap has data that survives between (procedure) calls.

- Layout not known statically
- Managed dynamically by garbage collector

Static vs Dynamic Storage Allocation

Static means “at compile time“

Dynamic means “at runtime“

Stack has data local to (procedure) call.

- Layout known statically
- Managed automatically by generated code

Heap has data that survives between (procedure) calls.

- Layout not known statically
- Managed dynamically by garbage collector

Static vs Dynamic Storage Allocation

Static means “at compile time“

Dynamic means “at runtime“

Stack has data local to (procedure) call.

- Layout known statically
- Managed automatically by generated code

Heap has data that survives between (procedure) calls.

- Layout not known statically
- Managed dynamically by garbage collector

Static vs Dynamic Storage Allocation

Static means “at compile time“

Dynamic means “at runtime“

Stack has data local to (procedure) call.

- Layout known statically
- Managed automatically by generated code

Heap has data that survives between (procedure) calls.

- Layout not known statically
- Managed dynamically by garbage collector

Static vs Dynamic Storage Allocation

Static means “at compile time“

Dynamic means “at runtime“

Stack has data local to (procedure) call.

- Layout known statically
- Managed automatically by generated code

Heap has data that survives between (procedure) calls.

- Layout not known statically
- Managed dynamically by garbage collector

Static vs Dynamic Storage Allocation

Static means “at compile time“

Dynamic means “at runtime“

Stack has data local to (procedure) call.

- Layout known statically
- Managed automatically by generated code

Heap has data that survives between (procedure) calls.

- Layout not known statically
- Managed dynamically by garbage collector

Garbage Collector

Purpose: Enables the runtime system to detect useless data elements, and reuse their storage. Automated in most modern languages.

Storage Organization

"Runtime representation of an object program"

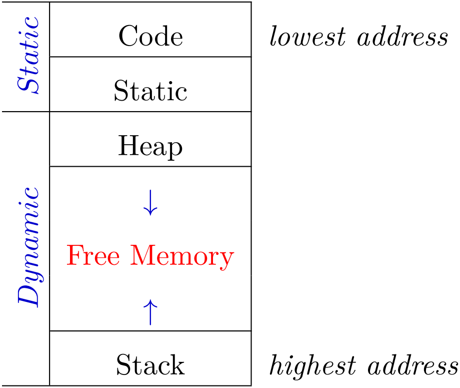
Code area instructions, address pointers.

Static area global constants.

Heap objects, records, arrays, variable strings.[1]

Stack activation records/stack frames;
objects, records, arrays, variable strings.[2]

Storage Organization



Storage Organization

Code area	executable target code, statically determined.
Static area	data generated by the compiler known at compile time.
Heap	grows towards higher addresses
Stack	grows towards lower addresses

Outline

Activation Tree

- Shows all calls at runtime.
- A call is a child of the callee node.
- Children are in call order from left to right.

Activation Tree

- Shows all calls at runtime.
- A call is a child of the callee node.
- Children are in call order from left to right.

Activation Tree

- Shows all calls at runtime.
- A call is a child of the callee node.
- Children are in call order from left to right.

Activation Tree Example - QuickSort

```
int a[11];

void readArray() {
int i;
}
int partition(int m, int n) {

}
void quicksort(int m, int n){
int i;
if (n > m){
i = partition(m,n);
quicksort(m, i−1);
quicksort(i+1, n);
}
```

```
void quicksort(int m, int n) {
int i;
if (n > m) {
i = partition(m, n);
quicksort(m, i−1);
quicksort(i+1, n);
}
}
main() {
readArray();
a[0] = −9999;
a[10] = 9999;
quicksort(1,9);
}
```

Activation Tree Example

```
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
       enter partition(1,9)
       leave partition(1,9)
       enter quicksort(1,3)
          ...
       leave quicksort(1,3)
       enter quicksort(5,9)
         ...
       leave quicksort(5,9)
   leave quicksort(1,9)
leave main()
```

i = 4

Activation Tree Example

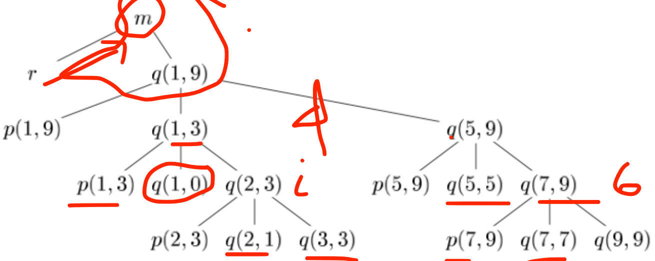


Figure: Activation tree representing calls during an execution of quicksort

Contents of Activation Record

1. Actual parameters (commonly in registers instead).
2. Space for return value.
3. Control link (caller's activation record, "dynamic link").
4. Access link ("static link").
5. Saved machine state (notably return address).
6. Local data for called procedure.
7. Temporary values that do not fit in registers.

Contents of Activation Record

1. Actual parameters (commonly in registers instead).
2. Space for return value.
3. Control link (caller's activation record, "dynamic link").
4. Access link ("static link").
5. Saved machine state (notably return address).
6. Local data for called procedure.
7. Temporary values that do not fit in registers.

Contents of Activation Record

1. Actual parameters (commonly in registers instead).
2. Space for return value.
3. Control link (caller's activation record, "dynamic link").
4. Access link ("static link").
5. Saved machine state (notably return address).
6. Local data for called procedure.
7. Temporary values that do not fit in registers.

Contents of Activation Record

1. Actual parameters (commonly in registers instead).
2. Space for return value.
3. Control link (caller's activation record, "dynamic link").
4. Access link ("static link").
5. Saved machine state (notably return address).
6. Local data for called procedure.
7. Temporary values that do not fit in registers.

Contents of Activation Record

1. Actual parameters (commonly in registers instead).
2. Space for return value.
3. Control link (caller's activation record, "dynamic link").
4. Access link ("static link").
5. Saved machine state (notably return address).
6. Local data for called procedure.
7. Temporary values that do not fit in registers.

Contents of Activation Record

1. Actual parameters (commonly in registers instead).
2. Space for return value.
3. Control link (caller's activation record, "dynamic link").
4. Access link ("static link").
5. Saved machine state (notably return address).
6. Local data for called procedure.
7. Temporary values that do not fit in registers.

Contents of Activation Record

1. Actual parameters (commonly in registers instead).
2. Space for return value.
3. Control link (caller's activation record, "dynamic link").
4. Access link ("static link").
5. Saved machine state (notably return address).
6. Local data for called procedure.
7. Temporary values that do not fit in registers.

Activation Record or "Frame"

...
Actual Parameters
Return Values
Control Link
Access Link
Saved Machine State
Local Data
Temporaries
↓

Activation Record or “Frame”

a = b + c / 4
t1 = c / 4
a = b + t1

1. Temporary values, such as those arising from the evaluation of expressions
2. Local data belonging to the procedure whose activation record this is.
3. A saved machine status just before the call to the procedure. Such as *return address*
4. An access link may be needed to locate data needed by the called procedure but found elsewhere
5. A control link, pointing to the activation record of the caller.
6. Space for the return value of the called function, if any.
7. The actual parameters used by the calling procedure

Snapshot of run-time stack for 'quicksort'

- Dashed lines in the partial trees go to activations that have ended.
- Since array a is global, space is allocated for it before execution begins with an activation of procedure main, as shown in

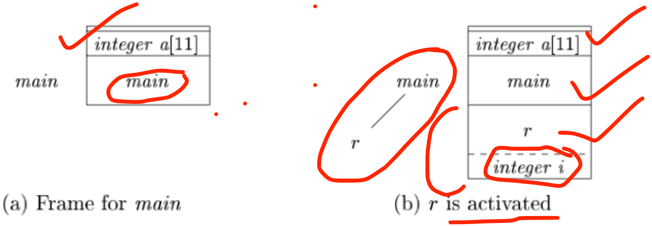


Figure: Downward growing stack of activation records

Snapshot of run-time stack for 'quicksort'

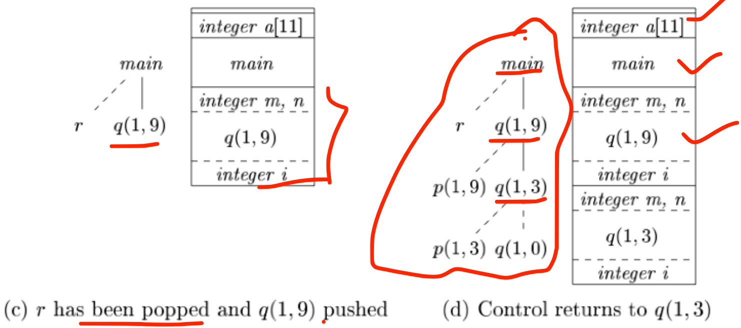


Figure: Downward growing stack of activation records

Where Is My Activation Record?

- Keep actual (top) frame's Frame Pointer (FP) in a register.
- Keep track of Local Stack Size (SP) in a register.
    - Variable-length data on stack.

	...
	Actual Parameters
	Return Values
$FP \rightarrow$	Control Link
	Access Link
	Saved Machine State
	Local Data
$SP \rightarrow$	Temporaries
	$\downarrow$

Where Is My Activation Record?

- Keep actual (top) frame's Frame Pointer (FP) in a register.
- Keep track of Local Stack Size (SP) in a register.
    - Variable-length data on stack.

	...
	Actual Parameters
	Return Values
$FP \rightarrow$	Control Link
	Access Link
	Saved Machine State
	Local Data
$SP \rightarrow$	Temporaries
	$\downarrow$

Where Is My Activation Record?

- Keep actual (top) frame's Frame Pointer (FP) in a register.
- Keep track of Local Stack Size (SP) in a register.
  - Variable-length data on stack.

	...
	Actual Parameters
	Return Values
$FP \rightarrow$	Control Link
	Access Link
	Saved Machine State
	Local Data
$SP \rightarrow$	Temporaries
	$\downarrow$

Calling Sequence

Calling sequences consists of code that allocates an activation record on the stack and enters information into its fields:

1. caller allocates space for and stores actual parameters.
2. caller allocates space for return value.
3. caller pushes the control/dynamic link (FP).
4. callee allocates space for and stores access/static link.
5. callee allocates space for and stores saved machine state.
6. callee allocates space for local data.
7. callee runs and might use temporaries while it runs

Calling Sequence

Calling sequences consists of code that allocates an activation record on the stack and enters information into its fields:

1. caller allocates space for and stores actual parameters.
2. caller allocates space for return value.
3. caller pushes the control/dynamic link (FP).
4. callee allocates space for and stores access/static link.
5. callee allocates space for and stores saved machine state.
6. callee allocates space for local data.
7. callee runs and might use temporaries while it runs

Calling Sequence

Calling sequences consists of code that allocates an activation record on the stack and enters information into its fields:

1. caller allocates space for and stores actual parameters.
2. caller allocates space for return value.
3. caller pushes the control/dynamic link (FP).
4. callee allocates space for and stores access/static link.
5. callee allocates space for and stores saved machine state.
6. callee allocates space for local data.
7. callee runs and might use temporaries while it runs

Calling Sequence

Calling sequences consists of code that allocates an activation record on the stack and enters information into its fields:

1. caller allocates space for and stores actual parameters.
2. caller allocates space for return value.
3. caller pushes the control/dynamic link (FP).
4. callee allocates space for and stores access/static link.
5. callee allocates space for and stores saved machine state.
6. callee allocates space for local data.
7. callee runs and might use temporaries while it runs

Calling Sequence

Calling sequences consists of code that allocates an activation record on the stack and enters information into its fields:

1. caller allocates space for and stores actual parameters.
2. caller allocates space for return value.
3. caller pushes the control/dynamic link (FP).
4. callee allocates space for and stores access/static link.
5. callee allocates space for and stores saved machine state.
6. callee allocates space for local data.
7. callee runs and might use temporaries while it runs

Calling Sequence

Calling sequences consists of code that allocates an activation record on the stack and enters information into its fields:

1. caller allocates space for and stores actual parameters.
2. caller allocates space for return value.
3. caller pushes the control/dynamic link (FP).
4. callee allocates space for and stores access/static link.
5. callee allocates space for and stores saved machine state.
6. callee allocates space for local data.
7. callee runs and might use temporaries while it runs

Calling Sequence

Calling sequences consists of code that allocates an activation record on the stack and enters information into its fields:

1. caller allocates space for and stores actual parameters.
2. caller allocates space for return value.
3. caller pushes the control/dynamic link (FP).
4. callee allocates space for and stores access/static link.
5. callee allocates space for and stores saved machine state.
6. callee allocates space for local data.
7. callee runs and might use temporaries while it runs

Calling Sequence, continued.

Calling sequences consists of code that allocates an activation record on the stack and enters information into its fields:

1. callee cleans up local material (pops local data and temporaries).
2. callee restores caller's machine state (includes "jumping back").
3. caller extracts return value and recovers space used for parameters.

Calling Sequence, continued.

Calling sequences consists of code that allocates an activation record on the stack and enters information into its fields:

1. callee cleans up local material (pops local data and temporaries).
2. callee restores caller's machine state (includes "jumping back").
3. caller extracts return value and recovers space used for parameters.

Outline

Easy: C (no nested procedures, static scoping)

- how procedures access their data, also data from other procedures
- Global and static data goes in static section.
- Everything else is by default on the stack.
- Explicit malloc() and free() manage the heap

Harder: C++, C#, Java, Scala, ML, Haskell, ...

- Has nested procedures.
- Uses access link ("static link") or display.

Outline

The Heap

Static	Code	*lowest address*
	Static	
Dynamic	**Heap**	
	↓ Free Memory ↑	
	Stack	*highest address*

The Heap

Heap *objects* may live indefinitely. “Life span” is managed automatically or from program (memory manager).

	Java	**C**	**C++**
Allocation	new	malloc	new
Deallocation	garbage c.	free	delete

Memory Manager: function and properties

Allocation get contiguous chunk of bytes (sometimes virtual memory).

Deallocation release previously allocated chunk for reuse.

Desired properties of a memory manager:

Space efficiency Use the available memory well (avoid fragmentation).

Program efficiency Make access to the allocated memory fast (use locality aspects).

Low overhead Reduce administrative cost of allocation.

Memory Manager: function and properties

Allocation get contiguous chunk of bytes (sometimes virtual memory).

Deallocation release previously allocated chunk for reuse.

Desired properties of a memory manager:

Space efficiency Use the available memory well (avoid fragmentation).

Program efficiency Make access to the allocated memory fast (use locality aspects).

Low overhead Reduce administrative cost of allocation.

Memory Manager: function and properties

Allocation get contiguous chunk of bytes (sometimes virtual memory).

Deallocation release previously allocated chunk for reuse.

Desired properties of a memory manager:

Space efficiency Use the available memory well (avoid fragmentation).

Program efficiency Make access to the allocated memory fast (use locality aspects).

Low overhead Reduce administrative cost of allocation.

Memory Hierarchy

Registers	10^3 bytes @ 0.2 ns.
L1 cache	10^5 bytes @ 5 ns.
L2 cache	10^7 bytes @ 20 ns.
Physical Memory	10^{10} bytes @ 100 ns.
Flash Storage	10^{12} bytes @ 1 ms.
Disk	10^{14} bytes @ 10 ms.

Locality

- Memory is loaded into cache in chunks.
- Multiple access to same chunk improves cache rate.
- Reorganizing data can drastically improve performance.
    - Allocator and garbage collector can help!