Memory Management

Madhavan Mukund

`https://www.cmi.ac.in/~madhavan`

Programming Concepts using Java

Week 1

Keeping track of variables

- Variables store intermediate values during computation
    - Typically these are local to a function
    - Can also refer to global variables outside the function
    - Dynamically created data, like nodes in a list

Keeping track of variables

- Variables store intermediate values during computation
  - Typically these are local to a function
  - Can also refer to global variables outside the function
  - Dynamically created data, like nodes in a list
- Scope of a variable
  - When the variable is available for use

Keeping track of variables

- Variables store intermediate values during computation
  - Typically these are local to a function
  - Can also refer to global variables outside the function
  - Dynamically created data, like nodes in a list

- Scope of a variable
  - When the variable is available for use
  - In the following code, the `x` in `f()` is not in scope within call to `g()`

```
def f(l):
  ...
  for x in l:
    y = y + g(x)
  ...
```

```
def g(m):
  ...
  for x in range(m):
    ...
```

Keeping track of variables

- Variables store intermediate values during computation
  - Typically these are local to a function
  - Can also refer to global variables outside the function
  - Dynamically created data, like nodes in a list

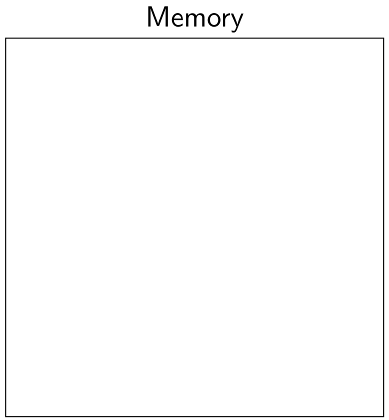
- Scope of a variable
  - When the variable is available for use
  - In the following code, the `x` in `f()` is not in scope within call to `g()`

```
def f(l):                    def g(m):
  ...                          ...
  for x in l:                  for x in range(m):
    y = y + g(x)                 ...
  ...
```

- Lifetime of a variable
  - How long the storage remains allocated
  - Above, lifetime of `x` in `f()` is till `f()` exits
  - “Hole in scope” — variable is alive but not in scope

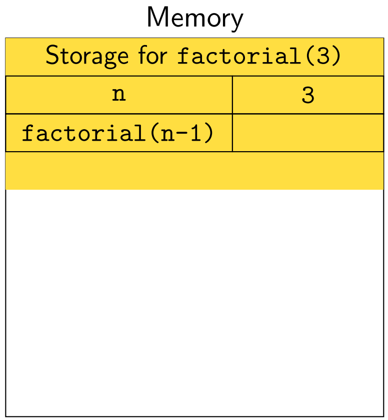
Memory stack

- Each function needs storage for local variables



Memory stack

- Each function needs storage for local variables
- Create activation record when function is called

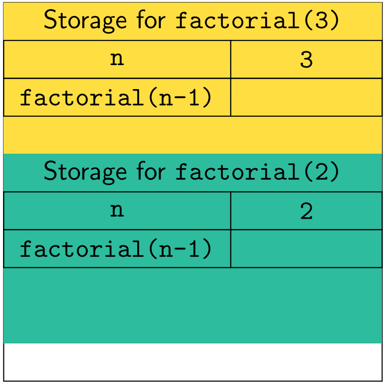


- Call `factorial(3)`

Memory stack

- Each function needs storage for local variables
- Create activation record when function is called
- Activation records are stacked
  - Popped when function exits

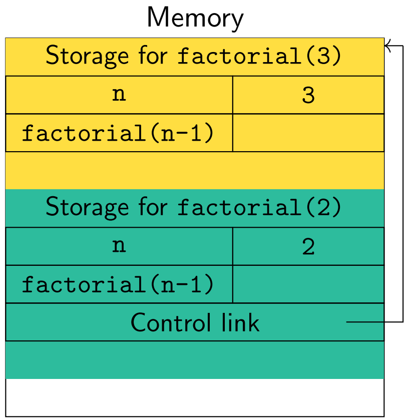
Memory



- Call factorial(3)
- factorial(3) calls factorial(2)

Memory stack

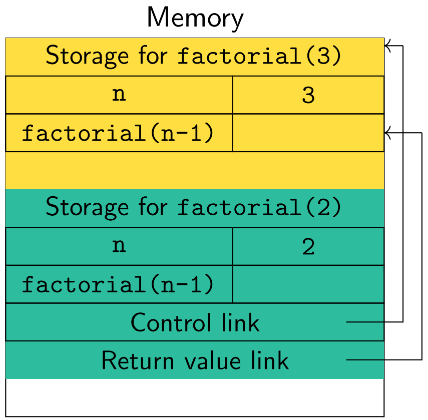
- Each function needs storage for local variables
- Create activation record when function is called
- Activation records are stacked
  - Popped when function exits
  - Control link points to start of previous record



- Call `factorial(3)`
- `factorial(3)` calls `factorial(2)`

Memory stack

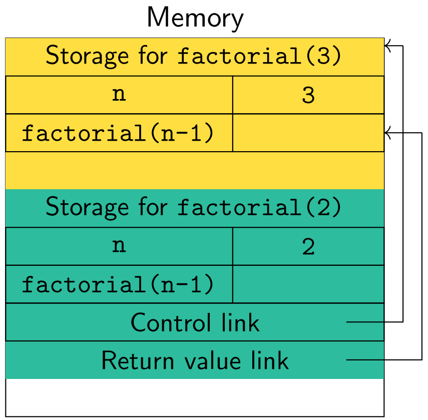
- Each function needs storage for local variables
- Create activation record when function is called
- Activation records are stacked
  - Popped when function exits
  - Control link points to start of previous record
  - Return value link tells where to store result



- Call `factorial(3)`
- `factorial(3)` calls `factorial(2)`

Memory stack

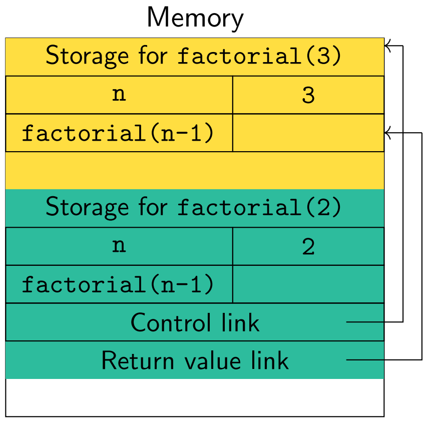
- Each function needs storage for local variables
- Create activation record when function is called
- Activation records are stacked
  - Popped when function exits
  - Control link points to start of previous record
  - Return value link tells where to store result
- Scope of a variable
  - Variable in activation record at top of stack
  - Access global variables by following control links



- Call `factorial(3)`
- `factorial(3)` calls `factorial(2)`

Memory stack

- Each function needs storage for local variables
- Create activation record when function is called
- Activation records are stacked
  - Popped when function exits
  - Control link points to start of previous record
  - Return value link tells where to store result
- Scope of a variable
  - Variable in activation record at top of stack
  - Access global variables by following control links
- Lifetime of a variable
  - Storage allocated is still on the stack



- Call `factorial(3)`
- `factorial(3)` calls `factorial(2)`

Passing arguments to functions

- When a function is called, arguments are substituted for formal parameters

```
def f(a,l):         x = 7
    ...             myl = [8,9,10]
    ...             f(x,myl)
```

Passing arguments to functions

- When a function is called, arguments are substituted for formal parameters

```
def f(a,l):         x = 7
    ...             myl = [8,9,10]
    ...             f(x,myl)
```

- Parameters are part of the activation record of the function
    - Values are populated on function call

Passing arguments to functions

- When a function is called, arguments are substituted for formal parameters

```
def f(a,l):          x = 7                  a = x
    ...              myl = [8,9,10]         l = myl
    ...              f(x,myl)               ... code for f() ...
```

- Parameters are part of the activation record of the function
  - Values are populated on function call
  - Like having implicit assignment statements at the start of the function

Passing arguments to functions

- When a function is called, arguments are substituted for formal parameters

```
def f(a,l):          x = 7                   a = x
    ...              myl = [8,9,10]          l = myl
    ...              f(x,myl)                ... code for f() ...
```

- Parameters are part of the activation record of the function
  - Values are populated on function call
  - Like having implicit assignment statements at the start of the function
- Two ways to initialize the parameters

Passing arguments to functions

- When a function is called, arguments are substituted for formal parameters

```
def f(a,l):          x = 7                  a = x
    ...              myl = [8,9,10]         l = myl
    ...              f(x,myl)               ... code for f() ...
```

- Parameters are part of the activation record of the function
  - Values are populated on function call
  - Like having implicit assignment statements at the start of the function
- Two ways to initialize the parameters
  - Call by value — copy the value
    - Updating the value inside the function has no side-effect

Passing arguments to functions

- When a function is called, arguments are substituted for formal parameters

```
def f(a,l):          x = 7               a = x
    ...              myl = [8,9,10]      l = myl
    ...              f(x,myl)            ... code for f() ...
```

- Parameters are part of the activation record of the function
  - Values are populated on function call
  - Like having implicit assignment statements at the start of the function
- Two ways to initialize the parameters
  - Call by value — copy the value
    - Updating the value inside the function has no side-effect
  - Call by reference — parameter points to same location as argument
    - Can have side-effects
    - Be careful: can update the contents, but cannot change the reference itself

Heap

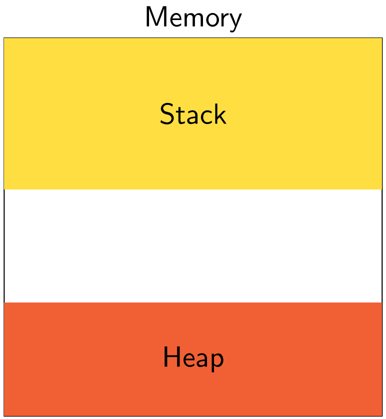
- Function that inserts a value in a linked list
  - Storage for new node allocated inside function
  - Node should persist after function exits
  - Cannot be allocated within activation record

Heap

- Function that inserts a value in a linked list
  - Storage for new node allocated inside function
  - Node should persist after function exits
  - Cannot be allocated within activation record
- Separate storage for persistent data
  - Dynamically allocated vs statically declared
  - Usually called the heap
    - Not the same as the heap data structure!

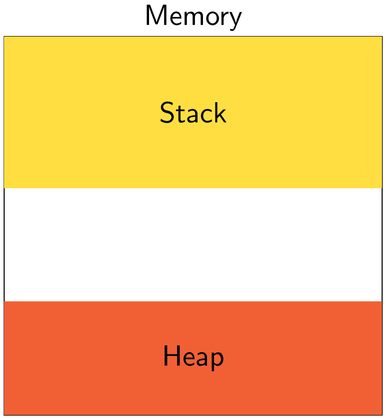
Heap

- Function that inserts a value in a linked list
  - Storage for new node allocated inside function
  - Node should persist after function exits
  - Cannot be allocated within activation record
- Separate storage for persistent data
  - Dynamically allocated vs statically declared
  - Usually called the heap
    - Not the same as the heap data structure!
  - Conceptually, allocate heap storage from "opposite" end with respect to stack



Heap

- Function that inserts a value in a linked list
  - Storage for new node allocated inside function
  - Node should persist after function exits
  - Cannot be allocated within activation record
- Separate storage for persistent data
  - Dynamically allocated vs statically declared
  - Usually called the heap
    - Not the same as the heap data structure!
  - Conceptually, allocate heap storage from "opposite" end with respect to stack
- Heap storage outlives activation record
  - Access through some variable that is in scope



Managing heap storage

- On the stack, variables are deallocated when a function exits

Managing heap storage

- On the stack, variables are deallocated when a function exits
- How do we “return” unused storage on the heap?
  - After deleting a node in a linked list, deleted node i now dead storage, unreachable

Managing heap storage

- On the stack, variables are deallocated when a function exits
- How do we "return" unused storage on the heap?
  - After deleting a node in a linked list, deleted node i now dead storage, unreachable
- Manual memory management
  - Programmer explicitly requests and returns heap storage
    - `p = malloc(...)` and `free(p)` in C
  - Error-prone — memory leaks, invalid assignments

Managing heap storage

- On the stack, variables are deallocated when a function exits
- How do we “return” unused storage on the heap?
  - After deleting a node in a linked list, deleted node i now dead storage, unreachable
- Manual memory management
  - Programmer explicitly requests and returns heap storage
    - `p = malloc(...)` and `free(p)` in C
  - Error-prone — memory leaks, invalid assignments
- Automatic garbage collection (Java, Python, . . .)
  - Run-time environment checks and cleans up dead storage — e.g., mark-and-sweep
    - Mark all storage that is reachable from program variables
    - Return all unmarked memory cells to free space
  - Convenience for programmer vs performance penalty

Summary

- Variables have scope and lifetime
    - Scope — whether the variable is available in the program
    - Lifetime — whether the storage is still allocated
- Activation records for functions are maintained as a stack
    - Control link points to previous activation record
    - Return value link tells where to store result
- Heap is used to store dynamically allocated data
    - Outlives activation record of function that created the storage
    - Need to be careful about deallocating heap storage
    - Explicit deallocation vs automatic garbage collection