THEORY OF COMPILATION

LECTURE 10

ACTIVATION RECORDS
You are here

Source text → Compiler → Executable code

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Optimization
- Code Generation

+ Runtime
Supporting Procedures

- How is that done?
- What do we need from the compiler?

\[
\begin{align*}
n &= f(a[i]); \\
t_1 &= i \times 4 \\
t_2 &= a + t_1 \\
t_3 &= \ast t_2 \\
\text{param} \ t_3 \\
t_4 &= \text{call} \ f, \ 1 \\
n &= t_4
\end{align*}
\]
Supporting Procedures

• Extending our computing environment
  ‣ (at least) enough memory for local variables
• Passing information into the new environment
  ‣ call parameters
• Transfer of control to/from procedure
• Handling return values
Design Decisions

- Scoping rules
  - Static scoping vs. dynamic scoping

- Memory layout
  - Allocating space for local variables

- Caller/callee conventions
  - Saving and restoring register values
Static (Lexical) Scoping

A name refers to its (closest) enclosing scope known at compile time

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Scopes</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 0</td>
<td>B₀, B₁, B₃</td>
</tr>
<tr>
<td>b = 0</td>
<td>B₀</td>
</tr>
<tr>
<td>b = 1</td>
<td>B₁, B₂</td>
</tr>
<tr>
<td>a = 2</td>
<td>B₂</td>
</tr>
<tr>
<td>b = 3</td>
<td>B₃</td>
</tr>
</tbody>
</table>
Dynamic Scoping

• Every function call creates new definitions for variables in its scope
  – definitions are maintained in a global stack
• When entering scope where identifier is declared
  – push declaration on identifier stack
• When exiting scope where identifier is declared
  – pop identifier stack
• Evaluating the identifier in any context binds to the current top of stack
  ‣ determined at runtime
Example

```c
int x = 42;

int f() { return x; }
int g() { int x = 1; return f(); }
int main() { print g(); print x; }
```

- What values are output by `main`?
  - static scoping?
  - dynamic scoping?
Why do we care?

• **We need to generate code to access variables**

• **Static scoping**
  – identifier binding is known at *compile time*
  – address of the variable is known at *compile time*
  – assigning addresses to variables is part of code generation
  – no runtime errors of “access to undefined variable”
  – can check types of variables
Variable Addressing for Static Scoping
(first attempt)

int x = 42;
int f() { return x; }
int g() { int x = 1;
    return f(); }
int main() { print g(); }

<table>
<thead>
<tr>
<th>identifier</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>x (global)</td>
<td>0x42</td>
</tr>
<tr>
<td>x (inside g)</td>
<td>0x73</td>
</tr>
</tbody>
</table>
Variable Addressing for Static Scoping (first attempt)

```c
int x = 42;
int f() { return x; }
int g() { int x = 1;
    return f(); }
int main() { print g(); }
```

<table>
<thead>
<tr>
<th>identifier</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>x (global)</td>
<td>0x42</td>
</tr>
<tr>
<td>x (inside g)</td>
<td>0x73</td>
</tr>
</tbody>
</table>
Variable Addressing for Static Scoping
(first attempt)

int x = 42;
int f() { return x; }
int g() { int x = 1;
     return f(); }
int main() { print g(); }
Variable Addressing for Static Scoping (first attempt)


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

main() {
    ... 
    quicksort (1, 9) ;
}

<table>
<thead>
<tr>
<th>identifier</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a (global)</td>
<td>0x42</td>
</tr>
<tr>
<td>i</td>
<td>...</td>
</tr>
</tbody>
</table>

(inside quicksort)

what is the address of the variable “i” in the procedure quicksort?
Variable Addressing for Static Scoping (first attempt)

int a[11];

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

main() {
    ... 
    quicksort(1, 9);
}

<table>
<thead>
<tr>
<th>identifier</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a (global)</td>
<td>0x42</td>
</tr>
<tr>
<td>i (inside quicksort)</td>
<td>...</td>
</tr>
</tbody>
</table>
Activation Record (frame)

- Separate space for each procedure invocation

- Managed at runtime
  - But code for managing it generated by the compiler

- Desired properties
  - efficient allocation and deallocation
    - procedure calls are frequent
  - flexible size
    - different procedures may require different memory sizes
Memory Layout

- **Stack**: grows down (towards lower addresses)
- **Heap**: grows up (towards higher addresses)
- **Static Data**
- **Code**

Addresses:
- Lower addresses: 0x0000
- Higher addresses: 0xFFFF
Memory Layout

- Stack grows down (towards lower addresses)
- Heap grows up (towards higher addresses)

Address ranges:
- Stack: 0x0000 to 0xFFFF
- Heap: 0x0000 to 0xFFFF
- Static data: 0x0000 to 0xFFFF
- Code: 0x0000 to 0xFFFF
Runtime Stack

- FP – frame pointer
  - beginning of current frame + some offset
    - Sometimes called BP (base pointer)

- SP – stack pointer
  - end of current frame
Runtime Stack

• Stack of activation records
  ‣ Call = push new activation record
  ‣ Return = pop activation record
• Only one “active” record at a time (per thread)
  ‣ top of stack
• This allows to handle recursion
Activation Record (frame)

incoming parameters
- parameter k
- ...
- parameter 1

administrative part
- lexical pointer
- return information
- dynamic link (= prev. fp)
- saved register values
- local variables
- temporaries

next frame would be here

FP
SP

higher addresses

stack grows down

lower addresses
## x86 Runtime Stack

### Stack registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%ebp</td>
<td>Frame pointer</td>
</tr>
</tbody>
</table>

### Stack and subroutine instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>push, pusha, ...</td>
<td>push on runtime stack</td>
</tr>
<tr>
<td>pop, popa, ...</td>
<td>pop from runtime stack</td>
</tr>
<tr>
<td>call</td>
<td>transfer control to called routine</td>
</tr>
<tr>
<td>ret</td>
<td>transfer control back to caller</td>
</tr>
</tbody>
</table>
x86 Runtime Stack

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%ebp</td>
<td>Frame pointer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>push, pusha, ...</td>
<td>push on runtime stack</td>
</tr>
<tr>
<td>pop, popa, ...</td>
<td>pop from runtime stack</td>
</tr>
<tr>
<td>call</td>
<td>transfer control to called routine</td>
</tr>
<tr>
<td>ret</td>
<td>transfer control back to caller</td>
</tr>
</tbody>
</table>

Stack registers

Stack and subroutine instructions
# x86 Runtime Stack

## Stack Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%ebp</td>
<td>Frame pointer</td>
</tr>
</tbody>
</table>

## Stack and Subroutine Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>push, pusha, ...</td>
<td>push on runtime stack</td>
</tr>
<tr>
<td>pop, popa, ...</td>
<td>pop from runtime stack</td>
</tr>
<tr>
<td>call</td>
<td>transfer control to called routine</td>
</tr>
<tr>
<td>ret</td>
<td>transfer control back to caller</td>
</tr>
</tbody>
</table>

%eip: Instruction pointer
Call Sequences

- The processor does not save the content of registers on “call” instruction

- So who will?
  - There are several options:
    - Caller saves and restores registers
    - Callee saves and restores registers
    - But can also have both save/restore some registers
Call Sequences
Call Sequences

Caller push code

Push caller-save registers
Push actual parameters
(in reverse order)

Previous frame

FP

SP
Call Sequences

Caller push code

Push caller-save registers
Push actual parameters (in reverse order)

FP

Previous frame

Reg 1

...

Reg n

SP
Call Sequences

Caller push code

Push caller-save registers
Push actual parameters
(in reverse order)

FP

Previous frame

Reg 1
...Reg n

Param n
...Param 1

SP

...
Call Sequences

caller

<table>
<thead>
<tr>
<th>Caller push code</th>
</tr>
</thead>
</table>

Call

| : |
| : |

Param n

Param 1

Reg n

Reg 1

Previous frame

FP

SP

Push caller-save registers
Push actual parameters (in reverse order)

Push return address
Jump to call address
Call Sequences

Caller push code

Push caller-save registers
Push actual parameters  
(in reverse order)

Push return address
Jump to call address

FP
Previous frame
Reg 1
...  
Reg n
Param n
...  
Param 1
Return address

SP
Call Sequences

**Call**

- **Caller push code**
  - Push caller-save registers
  - Push actual parameters (in reverse order)
  - Push return address
  - Jump to call address

- **Callee push code (prologue)**
  - Push current base-pointer
  - FP = SP
  - Push local variables
  - Push callee-save registers

**Return address**

**Previous frame**

**FP**

- Reg 1
  - ...
  - Reg n

- Param n
  - ...
  - Param 1

**SP**

- Return address
Call Sequences

Caller push code

Call

Callee push code (prologue)

- Push caller-save registers
- Push actual parameters (in reverse order)
- Push return address
- Jump to call address

Push current base-pointer
FP = SP
Push local variables
Push callee-save registers

FP

Previous frame

Reg 1
...  
Reg n

Param n
...
Param 1

Return address

Previous FP

SP

Param 1
Call Sequences

**Caller push code**
- Push caller-save registers
- Push actual parameters (in reverse order)
- Push return address
- Jump to call address

**Callee push code (prologue)**
- Push current base-pointer
- FP = SP
- Push local variables
- Push callee-save registers

**Call**
- Previous frame
  - Reg 1
  - ... (n-1)
  - Reg n
  - Param n
  - ... (n-1)
  - Param 1
  - Return address
  - Previous FP

**Return address**

---

**Call Sequences**

- Call
- Return
Call Sequences

Caller push code

Push caller-save registers
Push actual parameters (in reverse order)

Push return address
Jump to call address

Callee push code (prologue)

Push current base-pointer
FP = SP
Push local variables
Push callee-save registers

FP

SP

Return address
Previous FP
Callee Regs

Param 1
... Param n

Local n
... Local 2
Local 1

Reg n ...
Reg 1

Previous frame
Call Sequences

**Caller push code**
- Push caller-save registers
- Push actual parameters (in reverse order)

**Call**
- Jump to call address

**Callee push code** (prologue)
- Push current base-pointer
- FP = SP
- Push local variables
- Push callee-save registers

**Callee pop code** (epilogue)
- Pop callee-save registers
- Pop callee activation record
- Pop old base-pointer

**Previous frame**
- Previous FP
- Local 1
  - ...
  - Local n

**Callee Regs**
- FP
- SP
Call Sequences

**caller**
- Caller push code

**call**
- Callee push code (prologue)
- Callee push code
- Callee pop code (epilogue)

**callee**
- Push caller-save registers
- Push actual parameters (in reverse order)
- Push return address
- Jump to call address
- Push current base-pointer
- FP = SP
- Push local variables
- Push callee-save registers
- Pop callee-save registers
- Pop callee activation record
- Pop old base-pointer

**Return address**
- Previous FP
- Local 1
- Local 2
- Local n
- Callee Regs
- Reg 1
- ... Reg n
- Param 1
- ... Param n
- Previous frame

**FP**
- SP
### Call Sequences

#### Caller

- **Caller push code**

#### Call

- **Call**

#### Callee

- **Callee push code**
  - (prologue)
  - ...

- **Callee pop code**
  - (epilogue)

#### Return

- **Return**

#### Push caller-save registers
- Push actual parameters (in reverse order)

#### Push return address
- Jump to call address

#### Push current base-pointer
- FP = SP
- Push local variables
- Push callee-save registers

#### Pop callee-save registers
- Pop callee activation record
- Pop old base-pointer

#### Pop return address
- Jump to address
Call Sequences

**Caller**
- Push caller-save registers
- Push actual parameters (in reverse order)

**Callee**
- Push current base-pointer
- FP = SP
- Push local variables
- Push callee-save registers
- Callee push code
  - (prologue)
- Callee pop code
  - (epilogue)
- Return address
- Return address

**Return**
- Pop callee-save registers
- Pop callee activation record
- Pop old base-pointer

**Call**
- Call
  - Push return address
  - Jump to call address

**Stack Frame**
- Previous frame
- Previous FP
- Local 1
- Local 2
- Local n
- Param 1
- Param n
- Reg 1
- Reg n
- Callee Regs
Call Sequences

Caller push code

Call

Callee push code (prologue)

Callee push code (epilogue)

Callee pop code

Return address

Return address

Previous FP

Previous FP

Local 1

Local 2

... Local n

Callee Regs

Reg 1

... Reg n

Param n

... Param 1

Callee Regs

Param

Previous frame

Push caller-save registers

Push actual parameters (in reverse order)

Push return address

Jump to call address

Push current base-pointer

FP = SP

Push local variables

Push callee-save registers

Pop callee-save registers

Pop callee activation record

Pop old base-pointer

Pop return address

Jump to address

Pop parameters

Pop caller-save registers
Call Sequences

**caller**
- Caller push code

**call**
- Callee push code
  - (prologue)
  - ...
- Callee pop code
  - (epilogue)

**callee**
- Push caller-save registers
- Push actual parameters (in reverse order)
- Push return address
- Jump to call address
- Push current base-pointer
- FP = SP
- Push local variables
- Push callee-save registers
- Pop callee-save registers
- Pop callee activation record
- Pop old base-pointer

**return**
- Pop return address
- Jump to address
- Pop parameters
- Pop caller-save registers

**caller**
- Caller pop code

**frame**
- Previous frame
  - FP
  - SP
  - Reg 1
  - ... Reg n
  - Param n
  - ... Param 1
  - Return address
  - Previous FP
  - Local 1
  - Local 2
  - ... Local n
  - Callee Regs
Call Sequences

foo(42, 21)

x86 (32 bit), cdecl calling convention

Push caller-save registers
Push actual parameters (in reverse order)
Push return address
Jump to call address
Push current base-pointer
FP = SP
Push local variables
Push callee-save registers
Pop callee-save registers
Pop callee activation record
Pop old base-pointer
Pop return address
Jump to address
Pop parameters
Pop caller-save registers

Call Sequences

20

Previous frame
Reg 1
... Reg n
Param n
... Param 1
Return address
Previous FP
Local 1
Local 2
... Local n
Callee Regs
“To Callee-save or to Caller-save?”

- That is indeed the question
  - Callee-saved registers need only be saved when callee modifies their value
  - Caller-saved registers need only be saved if the caller needs their value after the call returns

- Some conventions and heuristics are followed
  - x86 cdecl: %eax, %ecx, %edx are caller-saved; the rest are callee-saved.
Accessing Stack Variables

- Use offset from FP (%ebp)
- Remember – stack grows downwards
- Above FP = parameters
- Below FP = locals
- Examples
  - FP + 4 = return address
  - FP + 8 = first parameter
  - FP – 4 = first local
Factorial – \texttt{fact(int n)}

\begin{verbatim}
int fact(int n) {
    if (n <= 1) {
        return 1;
    }
    else {
        return fact(n - 1) * n;
    }
}
\end{verbatim}
Factorial – \texttt{fact(int n)}

\texttt{fact:}

\begin{verbatim}
pushl %ebp           # save ebp
movl %esp,%ebp       # ebp = esp
pushl %ebx           # save ebx
movl 8(%ebp),%ebx    # ebx = n
cmpl $1,%ebx         # n <= 1 ?
jle .lresult         # then done
leal -1(%ebx),%eax   # eax = n-1
pushl %eax           #
call fact            # eax = fact(n-1)
addl $4,%esp         #
imull %ebx,%eax      # eax = eax* n
jmp .lreturn         #
.lresult:
    movl $1,%eax      # return 1
.lreturn:
    movl -4(%ebp),%ebx # restore ebx
    movl %ebp,%esp    # restore esp
    popl %ebp         # restore ebp
    ret
\end{verbatim}

Parameter = n
Return address
old %ebp
old %ebx (=n)

(stack after prologue)
Factorial – \texttt{fact(int n)}

\textbf{fact:}

\begin{verbatim}
pushl %ebp           # save ebp
movl %esp,%ebp       # ebp = esp
pushl %ebx           # save ebx
movl 8(%ebp),%ebx    # ebx = n
cmpl $1,%ebx         # n <= 1 ?
jle .lresult         # then done
leal -1(%ebx),%eax   # eax = n-1

pushl %eax           #
call fact            # eax = fact(n-1)
addl $4,%esp         #
imull %ebx,%eax      # eax = eax * n
jmp .lreturn         #

.lresult:
movl $1,%eax         # return 1

.lreturn:
movl -4(%ebp),%ebx   # restore ebx
movl %ebp,%esp       # restore esp
popl %ebp            # restore ebp
ret
\end{verbatim}
Factorial – \texttt{fact(int n)}

\begin{verbatim}
fact:
pushl %ebp          # save ebp
movl %esp,%ebp      # ebp = esp
pushl %ebx           # save ebx
movl 8(%ebp),%ebx   # ebx = n
cmpl $1,%ebx        # n <= 1 ?
jle .lresult        # then done
lea -1(%ebx),%eax   # eax = n-1
pushl %eax          #
call fact           # eax = fact(n-1)
imull %ebx,%eax     # eax = eax * n
jmp .lreturn        #

.lresult:
movl $1,%eax        # return 1

.lreturn:
movl -4(%ebp),%ebx  # restore ebx
movl %ebp,%esp      # restore esp
popl %ebp           # restore ebp
ret
\end{verbatim}
Nested Procedures

• For example – in Pascal, JavaScript

• Any routine can have sub-routines

• Any sub-routine can access anything that is defined in its containing scope or inside the sub-routine itself
  – “non-local” variables
Example: Nested Procedures

```plaintext
program p;

var x: Integer;

procedure a

var y: Integer;

procedure b begin ... ... end;

function c

var z: Integer;

procedure d begin ... y ... end;

begin ... ... end;

begin ... ... end;

begin ... ... end;

begin ... ... end;

begin ... ... end.
```

possible call sequence:

```
p→a→a→c→b→c→d
```
Example: Nested Procedures

program p;
  var x: Integer;
procedure a
  var y: Integer;
  procedure b begin ... ... end;
function c
  var z: Integer;
  procedure d begin ... y ... end;
  begin ... ... end;
  begin ... ... end;
begin ... ... end;
begin ... ... end.

possible call sequence:
p~a~a~c~b~c~d

what is the address of variable “y” in procedure d?
Nested Procedures

- A routine can call a sibling or an ancestor
- When “c” uses (non-local) variables from “a”, which instance of “a” is it?
- How do you find the right activation record at runtime?

Possible call sequence:

\[ p \rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d \]
Nested Procedures

- **Goal**: find the closest routine in the stack from a given nesting level
- A routine may occur more than once in a sequence of calls
  - if a routine of level \(k\) uses variables of the same nesting level \(\Rightarrow\) it refers to the current activation record
  - if it uses variables of nesting level \(j < k\) \(\Rightarrow\) it must be the last routine called at level \(j\)
- If a procedure is last at level \(j\) on the stack, then it must be an ancestor of the current routine

Possible call sequence:
\[p \rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d\]
Nested Procedures

- **Problem**: a routine may need to access variables of another routine that contains it statically
- **Solution**: store a lexical pointer (a.k.a. access link) in the activation record
  - Lexical pointer points to the last activation record of the nesting level above it
    - According to what we just said, this is enough to reach any variable being referenced
- Lexical pointers are created *at runtime*
- Number of links to be traversed is always *known at compile time*
Lexical Pointers

program p;
var x: Integer;
procedure a
    var y: Integer;
    procedure b
        begin ... end;
    function c
        var z: Integer;
        procedure d
            begin ... y ... end;
            begin ... end;
            begin ... end;
            begin ... ... end;
            begin ... ... end.
    begin ... ... end;
begin ...
possible call sequence:
p\rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d

possible call sequence:

\[
p \rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d
\]
Lexical Pointers

program p;
var x: Integer;
procedure a
var y: Integer;
procedure b
begin ... end;
function c
var z: Integer;
procedure d
begin ... y ... end;
begin ... end;
begin ... end;
begin ... end.

possible call sequence:
p \rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d
Lexical Pointers

program p;
var x: Integer;
procedure a
var y: Integer;
procedure b
begin ... end;
function c
var z: Integer;
procedure d
begin ... y ... end;
begin ... end;
begin ... end;
begin ... end.

possible call sequence:
\[ p \rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d \]
Security Exploit(s)
Buffer Overflow

void foo (char *x) {
    char buf[4];
    strcpy(buf, x);
}

int main (int argc, char *argv[]) {
    foo(argv[1]);
}

% ./a.out abracadabra
Segmentation fault

(YMMV)
Security Exploit(s)
Buffer Overflow

```c
void foo (char *x) {
    char buf[4];
    strcpy(buf, x);
}

int main (int argc, char *argv[]) {
    foo(argv[1]);
}
```

% ./a.out abracadabra
Segmentation fault

(YMMV)
int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[16];

copy(password_buffer, password);
if(strcmp(password_buffer, "brillig") == 0)
    auth_flag = 1;
if(strcmp(password_buffer, "outgrabe") == 0)
    auth_flag = 1;
return auth_flag;
}
int main(int argc, char *argv[]) {
    if(argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0); }
if(check_authentication(argv[1])) {
    printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
" );
    printf("Access Granted.\n");
    printf("Access Denied.\n");
}
else {
    printf("Access Denied.\n");
}
int check_authentication(char *password) {
    char password_buffer[16];
    int auth_flag = 0;

    strcpy(password_buffer, password);
    if(strcmp(password_buffer, "brillig") == 0)
        auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0); }
    if(check_authentication(argv[1])) {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("      Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
    } else {
        printf("\nAccess Denied.\n");
    }
}
Activation Records: Summary

✓ Compile time memory management for procedure data
  – Used to pass parameters, store local variables, and restore registers

✓ Works well for data with well-scoped lifetime
  – deallocation when procedure returns
  – no need to call free() for stack data
Coming Up

MEMORY and ERROR MGMT