Binding and Scoping

Read: Scott, Chapter 3.1, 3.2 and 3.3.1, 3.3.2 and 3.3.6
Lecture Outline

- Notion of binding time
- Object lifetime and storage management

- An aside: Stack Smashing 101

- Scoping
  - Static scoping
  - Dynamic scoping
Notion of Binding Time

- **Binding time** (Scott): the time an answer becomes associated to an open question
Notion of Binding Time

- **Static**
  - Before program execution

- **Dynamic**
  - During program executes
Examples of Binding Time Decisions

- **Binding time** (Scott): the time an answer becomes associated to an open question

- Binding a variable name to a memory location
  - Static or dynamic
  - Determined by **scoping rules**

- Binding a variable/expression to a type
  - Static or dynamic

- Binding a call to a target subroutine
  - Static (as it is in C, mostly) or dynamic (virtual calls in Java, C++)
Example: Binding Variables to Locations

- Map a variable to a location
  - Map variable at use to a location
  - Map subroutine at use to target subroutine
- Determined by scoping rules
  - Static scoping
    - Binding before execution
  - Dynamic scoping
    - Binding during execution
- More on scoping later…

```c
int x, y;
void foo(int x) {
    y = x;
    int y = 0;
    if (y) {
        y = 1;
    }
}
```
General View of Dynamic Binding

- Dynamic binding → FLEXIBILITY
  - What are the advantages of dynamic binding?
  - Disadvantages?

- An example: Cost of dynamic binding of call to target method in OO languages
Example: Cost of Dynamic Dispatch in C++

- Source: Driesen and Hölzle, OOPSLA’96

Virtual function tables (VFTs)
Capital characters denote classes, lowercase characters message selectors, and numbers method addresses

**C++**: \( \times 5.2\% \) in dispatch

**Java**: \( \times 13\% \)

Load: `[object_reg+#VFTOffset],table_reg`
Load: `[table_reg+#selectorOffset],method_reg`
Call: `method_reg`

Extra instructions: cost extra
Other Choices Related to Binding Time

- Pointers: introduce “heap variables”
  - Good for flexibility – allows dynamic structures
  - Bad for efficiency – direct cost: accessed indirectly; indirect cost: compiler unable to perform optimizations

- Most PLs support pointers
  - Issues of management of heap memory
    - Explicit allocation and deallocation
    - Implicit deallocation (garbage collection)

- PL design choices – many subtle variations
  - No pointers (FORTRAN 77)
  - Explicit pointers (C++ and C)
  - Implicit pointers (Java)
Lecture Outline

- Notion of binding time
- Object lifetime and storage management
- An aside: Stack Smashing 101
- Scoping
  - Static scoping
  - Dynamic scoping
Storage Allocation Mechanisms

- **Static storage** – an object is given absolute address, which is the same throughout execution
  - What is an example of static data?

- **Stack storage** – stack objects are allocated on a run-time stack at subroutine call and deallocated at return
  - Needs a stack management algorithm
  - What is an example of stack data?

- **Heap storage** - long-lived objects are allocated and deallocated at arbitrary times during execution
  - Needs the most complex storage management algorithm
Combined View

- **Static storage**: `.text` (program code), `.rodata`, `.data`, etc.

- **Stack** contains one **stack frame** per executing subroutine
  - Stack grows from higher towards lower memory addresses

- **Heap** contains objects allocated and not yet de-allocated
  - Heap grows from lower towards higher memory addresses

Memory graph courtesy of RPISEC/MBE class.
Examples of Static Data

- Program code
- Global variables
- Tables of type data (e.g., inheritance structure)
- Dispatch tables (VFTs) and other tables
- Other
Examples of Stack Data

- What data is stored on the stack?
- Local variables, including parameters
- Compiler-generated temporaries (i.e., for expression evaluation)
- Bookkeeping (stack management) information
- Return address
Run-time Stack

- Stack contains frames of all subroutines that have been entered and not yet exited from.
- Frame contains all information necessary to update stack when subroutine is exited.
- Stack management uses two pointers: fp (frame pointer) and sp (stack pointer).
  - fp points to a location at the start of current frame
    - In higher memory (but lower on picture)
  - sp points to the next available location on stack (or the last used location on some machines)
    - In lower memory (but higher up on picture)
  - fp and sp define the beginning and the end of the frame
Run-time Stack

\[ \text{ff...ff} \]

\[ \text{sp} \]

\[ \text{fp} \]

\[ \text{B} \]

\[ \text{main} \]

\[ \text{etc.} \]
Run-time Stack Management

- Addresses for local variables are encoded as $sp + \text{offset}$
  - But may also have $fp - \text{offset}$

- Idea:
  - When subroutine is entered, its frame is placed on the stack. $sp$ and $fp$ are updated accordingly
  - All local variable accesses refer to this frame
  - When subroutine is exited, its frame is removed from the stack and $sp$ and $fp$ are updated accordingly
Frame Details

- Arguments to called routines
- Local variables, including parameters
- Temporaries
- Miscellaneous bookkeeping information
  - Saved address of start of caller’s frame (old fp)
  - Saved state (register values of caller), other
- Return address
Frame Example

```c
void foo(double rate, double initial) {
    double position; ...
    position = initial + rate*60.0; ...
    return;
}
```

Assume `bar` calls `foo`.

Frame for `foo`:

<table>
<thead>
<tr>
<th>sp -&gt;</th>
<th>Locals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>position</td>
</tr>
<tr>
<td></td>
<td>initial</td>
</tr>
<tr>
<td></td>
<td>rate</td>
</tr>
<tr>
<td></td>
<td>tmp</td>
</tr>
<tr>
<td>...</td>
<td>Misc bookkeeping info</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fp -&gt;</th>
<th>Return address in code of caller</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>old fp</td>
</tr>
<tr>
<td></td>
<td>return address</td>
</tr>
</tbody>
</table>
Lecture Outline

- Notion of binding time
- Object lifetime and storage management

- An aside: Stack Smashing 101
  - Slides courtesy of RPISEC/MBE

- Scoping
  - Static scoping
  - Dynamic scoping
Stack Frames

- In x86-64 RBP is `fp` and RSP is `sp`. Define the **stack frame** for the currently executing function
  - local variables
  - pointer to previous frame
  - return address

```c
void foo() {
    long long x = 0x1337;
    char str[16];
    strcpy(str, "ABCDEFGH0123456");
}
```

<table>
<thead>
<tr>
<th>RSP</th>
<th>RBP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1337</td>
<td></td>
</tr>
<tr>
<td>&quot;ABCDEFGH&quot;</td>
<td></td>
</tr>
<tr>
<td>0123456\0</td>
<td></td>
</tr>
<tr>
<td>0x7fff10203040</td>
<td></td>
</tr>
<tr>
<td>0x400134</td>
<td></td>
</tr>
</tbody>
</table>

**note:** for 64bit, each 'slot' is 8 bytes

```assembly
; _unwind {
    push rbp
    mov rbp, rsp
    push rbx
    sub rsp, 28h
    mov [rbp+var_28], rdi
    mov [rbp+var_30], rsi
    mov rax, [rbp+var_28]
    mov eax, [rax+128h]
    test eax, eax
}
```

```assembly
loc 30FB:
    mov [rax+0Ch], edx
    add edx, eax
    mov rax, [rbp+var_28]
    mov [rax+0Ch], edx
```

```assembly
lea rsi, aWannaCheatYes1; "wanna cheat"
lea rax, cs:ZSt14cout_ptr
mov rdi, rax
call _ZSt11char_traitsIcESt13char_traitsIcERKS11
```
What is corruption?

- What happens if a programmer makes a simple mistake:

```c
char foo[64];
int money = 0;
gets(foo);
```
gets()?

NAME

gets - get a string from standard input (DEPRECATED)

SYNOPSIS

#include <stdio.h>

char *gets(char *s);

DESCRIPTION

Never use this function.

gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).
Stack Smashing 101

main() has a stack frame:
- Contains local variables
- Pointer to previous frame
- Return address

Lower Memory

Not supposed to touch

Higher Memory

Start of char foo[64]

End of foo

money = 0

Base pointer

RETURN ADDRESS
Stack Smashing 101

As `gets()` continues to read input, we fill up the 64 bytes allocated for buffer `foo`
Stack Smashing 101

As gets() continues to read input, we fill up the 64 bytes allocated for foo.

Go far enough, it corrupts important data!
Stack Smashing 101

- We can give ourselves money!
- If we want to set money to 0x1337beef we need to know:
  - Most x86 machines are little endian (little byte goes first)
  - Meaning the byte order for numbers is "backwards" in memory
  - 0x01020304 would be

<table>
<thead>
<tr>
<th>0x04</th>
<th>0x03</th>
<th>0x02</th>
<th>0x01</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x04</td>
<td>0x03</td>
<td>0x02</td>
<td>0x01</td>
</tr>
</tbody>
</table>
Stack Smashing 201

- What else can we corrupt?
- What happens if you corrupt further?
  When does it segfault?

- What was that about a return address?
Stack Smashing 201

When `func()` is called, runtime stores the return address on the stack (i.e., the address of the instruction that immediately follows `call func in main`)
Stack Smashing 201

Before the call:

```
0x40051a <main+13>:  call 0x4004f6 <func>
0x40051f <main+18>:  mov DWORD PTR [rbp-0x4],eax
0x400522 <main+21>:  mov eax,0x0
0x400527 <main+26>:  leave
0x400528 <main+27>:  ret
```

No argument
Stack Smashing 201

Before the call:

```assembly
=> 0x40051a <main+13>: call 0x4004f6 <func>
0x40051f <main+18>: mov DWORD PTR [rbp-0x4],eax
0x400522 <main+21>: mov eax,0x0
0x400527 <main+26>: leave
0x400528 <main+27>: ret
No argument
```

After the call:

```assembly
=> 0x4004f6 <func>:
0x4004f7 <func+1>: lea rdi,[rip+0xb3]
0x4004fa <func+4>: call 0x4003f0 <puts@plt>
0x400501 <func+11>: mov eax,0x11
0x400506 <func+16>: 

Return address points back to where it left off in main
```
Stack Smashing 201

Returning just takes whatever is on the top of the stack, and jumps there, equivalently: `pop rip`

About to return:
Stack Smashing 201

Returning just takes whatever is on the top of the stack, and jumps there, equivalently: pop rip

About to return:

Returned back to main:

09/09/2019
Stack Smashing 201

Returning just takes whatever is on the top of the stack, and jumps there, equivalently: \texttt{pop rip}

About to return:

Returned back to main:

What if we change this???
Stack Smashing 201

Without corruption:

- At the end of the function, it returns
- 0x40051f is popped off the stack and stored in rip
- Control goes to that address

We want to change this

gets (foo)
Corrupted:

- At the end of the function, it returns
- \textbf{0x4141414141414141} is popped off the stack and stored in rip
- Control goes to that address
- but it's invalid memory...

Segmentation fault

MALICIOUS ADDRESS

Lower Memory

0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141

Start of char foo[64]

0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141

Higher Memory

0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141

MALICIOUS ADDRESS

0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141
0x4141414141414141

End of foo

money = 0

Base pointer

RETURN ADDRESS
Lecture Outline

- Notion of binding time
- Object lifetime and storage management
- An aside: Stack Smashing 101

- Scoping
  - Static scoping
  - Dynamic scoping
Scoping

- In most languages the same variable name can be used to denote different memory locations
- **Scoping rules**: map variable to declaration (location)
- **Scope**: region of program text where a declaration is visible
- Most languages use **static scoping**
  - Mapping from variable to declaration (or location) is made at compile time
- **Block-structured** programming languages
  - Nested subroutines (Pascal, ML, Scheme, etc.)
  - Nested blocks (C, C++ { … })
Static Scoping in Block Structured Programming Languages

- Also known as lexical scoping
- Block structure and nesting of blocks gives rise to the closest nested scope rule
  - There are local variable declaration within a block
  - A block inherits variable declarations from enclosing blocks
  - Local declarations take precedence over inherited ones
    - Hole in scope of inherited declaration
    - In other words, inherited declaration is hidden

- Lookup for non-local variables proceeds from inner to outer enclosing blocks
Example - Block Structured PL

```plaintext
main
  a, b, c: integer
  procedure P
    c: integer
    procedure S
      c, d: integer
      procedure R
      ...
      end R
    R()
  end S
  S()
end P
procedure R
  a: integer
  ...
  = a, b, c
end R
P()
end main
```

Nested block structure allows locally defined variables and subroutines

- `main.a, main.b, P.c`  
  `main.P, P.S, main.R`

- `main.a,main.b,S.c,S.d`  
  `main.P, P.S, S.R`

- `S.R,P.S,main.P`  
  `main.P, P.S, S.R`

- `R.a, main.b, main.c`  
  `main.R, main.P`
main
  a, b, c: integer
procedure P
    c: integer
procedure S
    c, d: integer
procedure R
    ... = a, b, c
    end R
R()
end S
R()
end P
procedure R
P()
end main

**Rule:** a variable is visible if it is declared in its own block or in a textually surrounding block and it is not ‘hidden’ by a binding in a closer block (i.e., there is no hole in scope)
Example with Frames

main
  a, b, c: integer /*1*/
procedure P /*3*/
  c: integer
procedure S /*8*/
  c, d: integer
procedure R /*10*/
  ...
  end R /*11*/
  R() /*9*/
end S /*12*/
end P /*13*/
procedure R /*5*/
  a: integer
  ...
  = a, b, c
end R /*6*/
procedure R /*5*/
  end S /*12*/
  R() /*4*/
  S() /*7*/
end P /*13*/
end main /*14*/

---
---
---

at /*1*/

main
  ---
  ---
  a
  b
  c

fp - currently active frame

at /*2*/, main calls main.P

sp
Example

main

a, b, c: integer /*1*/
procedure P /*3*/
  c: integer
  procedure S /*8*/
    c, d: integer
    procedure R /*10*/
      ...
      end R /*11*/
  R() /*9*/
end S /*12*/
R() /*4*/
S() /*7*/
end P /*13*/
procedure R /*5*/
  a: integer
  ... = a, b, c
end R /*6*/
P() /*2*/ ...
end main /*14*/

at /*3*/

static link
(access link; static environment)

dynamic link
(control link; called-by chain)
Example

main
a, b, c: integer /*1*/
procedure P /*3*/
c: integer
procedure S /*8*/
c, d: integer
procedure R /*10*/
...end R /*11*/
R() /*9*/
end S /*12*/
R() /*4*/
S() /*7*/
end P /*13*/
procedure R /*5*/
a: integer
... = a, b, c
end R /*6*/
P() /*2*/ ...
end main /*14*/
Example

at /*6*/ main.R exits
sp ← fp
fp ← old fp
in main.R’ s frame

```r
main
  a, b, c: integer /*1*/
  procedure P /*3*/
    c: integer
    procedure S /*8*/
      c, d: integer
      procedure R /*10*/
        ...
        end R /*11*/
        R() /*9*/
    end S /*12*/
    R() /*4*/
    S() /*7*/
  end P /*13*/
  procedure R /*5*/
    a: integer
    ... = a, b, c
  end R /*6*/
  P() /*2*/ ...
end main /*14*/
```

diagram of frame structure:

```
  main
    ---
    ---
    a
    b
    c
  main.P

  top
  sp ← fp
  fp ← old fp
```

"Example"

at /*6*/ main.R exits
sp ← fp
fp ← old fp
in main.R’ s frame
Example

main

a, b, c: integer /*1*/
procedure P /*3*/
c: integer
procedure S /*8*/
c, d: integer
procedure R /*10*/
...
end R /*11*/
R() /*9*/
end S /*12*/
R() /*4*/
S() /*7*/
end P /*13*/
procedure R /*5*/
a: integer
... = a, b, c
end R /*6*/
P() /*2*/ ...
end main /*14*/

at /*7*/,
P calls P.S;
at /*8*/:

\[main\]

---
---
---
a
b
c
---
main.P

c
P.S

c
---
d
fp
sp
Example

main
  a, b, c: integer /*1*/
  procedure P /*3*/
    c: integer
    procedure S /*8*/
      c, d: integer
      procedure R /*10*/
        ...
      end R /*11*/
    end S /*12*/
  end P /*13*/
  procedure R /*5*/
    a: integer
    ... = a, b, c
  end R /*6*/
P() /*2*/ ...
end main /*14*/

at /*9*/ S calls in S.R; at /*10*/
Example

main

a, b, c: integer /*1*/
procedure P /*3*/
  c: integer
procedure S /*8*/
  c, d: integer
  procedure R /*10*/
  ...
  end R /*11*/
R() /*9*/
end S /*12*/
R() /*4*/
S() /*7*/
end P /*13*/
procedure R /*5*/
  a: integer
  … = a, b, c
end R /*6*/
P() /*2*/ ...
end main /*14*/

/*11*/ pop S.R’s frame
Example

main

a, b, c: integer /*1*/
procedure P /*3*/
c: integer
procedure S /*8*/
c, d: integer
procedure R /*10*/
...
end R /*11*/
R() /*9*/
end S /*12*/
R() /*4*/
S() /*7*/
end P /*13*/
procedure R /*5*/
a: integer
… = a, b, c
end R /*6*/
P() /*2*/ ...
end main /*14*/

/*12*/ pop S’s frame

fp
sp
Example

main

a, b, c: integer /*1*/
procedure P /*3*/
c: integer
procedure S /*8*/
c, d: integer
procedure R /*10*/
...
end R /*11*/
R() /*9*/
end S /*12*/
R() /*4*/
S() /*7*/
end P /*13*/
procedure R /*5*/
a: integer
... = a, b, c
end R /*6*/
P() /*2*/ ...
end main /*14*/

at /*13*/

main

fp

sp

/*13*/ pop P’s frame
/*14*/ pop main’s frame
so that sp ← fp
Static Link vs. Dynamic Link

- **Static link** for a frame of subroutine P points to the most recent frame of P’s lexically enclosing subroutine
  - Bookkeeping required to maintain the static link
  - If subroutine P is enclosed $k$-levels deep from main, then the length of the static chain that begins at a frame for P, is $k$
  - To find non-local variables, follow static chain

- **Dynamic link** points to the caller frame, this is essentially old fp stored on frame
Observations

- Static link of a subroutine $P$ points to the frame of the most recent invocation of subroutine $Q$, where $Q$ is the lexically enclosing subroutine of $P$.

- Dynamic link may point to a different subroutine’s frame, depending on where the subroutine is called from.
An Important Note!

- For now, we assume languages that do not allow subroutines to be passed as arguments or returned from other subroutines, i.e., subroutines (functions) are third-class values.
  - When subroutines (functions) are third-class values, it is guaranteed the static reference environment is on the stack.
  - I.e., a subroutine cannot outlive its reference environment.
An Important Note!

- Static scoping rules become more involved in languages that allow subroutines to be passed as arguments and returned from other subroutines, i.e., subroutines (functions) are first class values.

- We will return to scoping later during our discussion of functional programming languages.
Dynamic Scoping

- Allows for local variable declaration
- Inherits non-local variables from subroutines that are live when current subroutine is invoked
  - Use of variable is resolved to the declaration of that variable in the most recently invoked and not yet terminated frame. I.e., lookup proceeds from closest predecessor on stack to furthest
- (old) Lisp, APL, Snobol, Perl
Example

main

procedure Z
  a: integer
  a := 1
  Y()
  output a
end Z

procedure W
  a: integer
  a := 2
  Y()
  output a
end W

procedure Y
  a := 0 /*1*/
end Y

Z()
W()
end main

Which a is modified at /*1*/ under dynamic scoping? Z.a or W.a or both?
Example

main

procedure Z
  a: integer
  a := 1
  Y()
  output a
end Z

procedure W
  a: integer
  a := 2
  Y()
  output a
end W

procedure Y
  a := 0; /*1*/
end Y

Z()
W()
end main

main calls Z, Z calls Y, Y sets Z.a to 0.
Example

main
  procedure Z
    a: integer
    a := 1
    Y()
    output a
  end Z
procedure W
  a: integer
  a := 2
  Y()
  output a
end W
procedure Y
  a := 0; /*1*/
end Y
Z()
W()
end main

main calls W, W calls Y, Y sets W.a to 0.
Static vs. Dynamic Scoping

Static Scoping:
- a bound to R.a, b to main.b, c to main.c

Dynamic Scoping:
- a bound to R.a, b to main.b, c to P.c
Dynamic Scoping

- Dynamic scoping is considered a bad idea. Why?

- More on static and dynamic scoping to come!
The End