Machine Program: Data

Jinyang Li

Last lecture

- How x86 supports procedure calls
  - call (pushes return address on stack; jump to function)
  - ret (pops return address from stack; jump to return address)
- C/UNIX calling convention (location of args/return val)
  - First 6 args are stored in regs: %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - Rest of arguments are stored on the stack
  - Return value (if there's one) is stored in %rax

Today's lesson plan

- C/UNIX calling convention (caller vs. callee-save reg)
- Program data storage and manipuation

Calling convention: Caller vs. callee-save registers

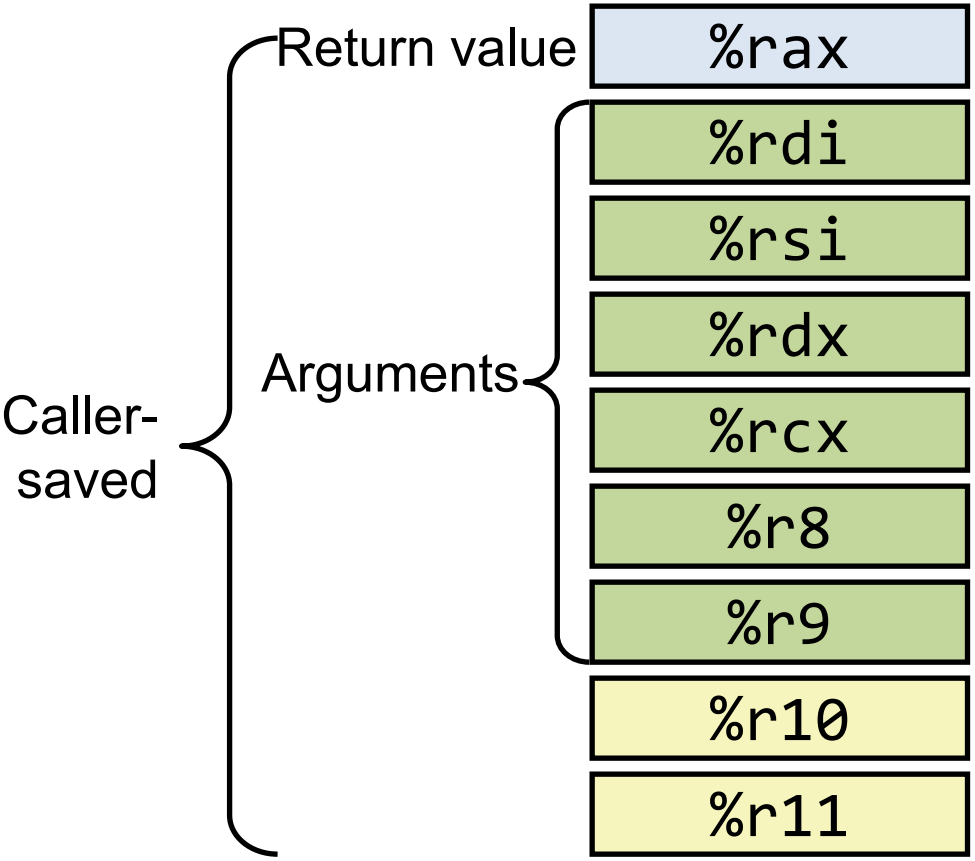
- What can the caller assume about the content of a register across function calls?

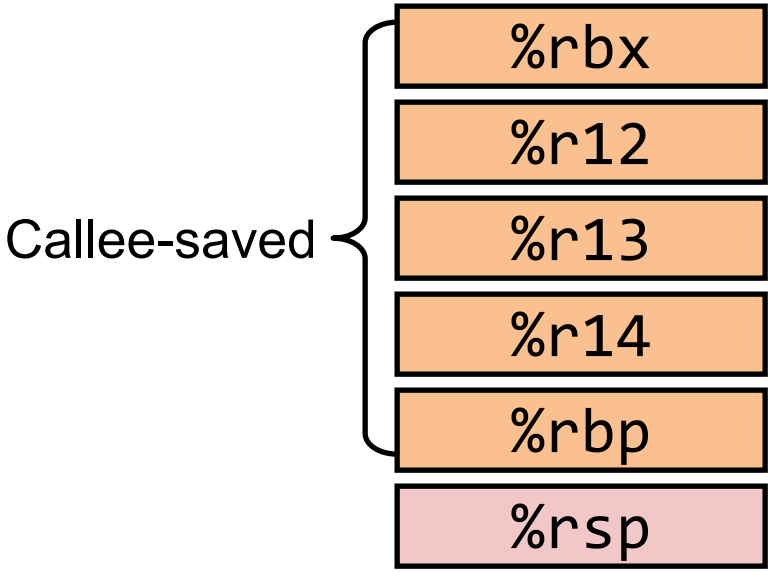
```
int foo() {
   int a;    // suppose a is stored in %r12
   a = .... // compute result of a

   int r = bar();

   int result = r + a; // does %r12 still store the value of a?
   return result;
}
```

Calling convention: register saving

- Caller-save registers
  - Register X's value may change across the function call.
  - It is caller's responsibility to save X on the stack and restore X after function returns (if caller needs X's value)
- Callee-save registers
  - Register Y's value must remain unchanged across the function call
  - It is callee's responsibility to save Y on the stack and restore Y before function return (if callee wants to change Y)

Calling convention: Register saving





Callee can use these regs without save/restore

Caller can assume these regs are unchanged across function calls

Example

```
int add2(int a, int b)
{
  return a + b;
}
```

```
add2:
    leal    (%rdi,%rsi), %eax
    ret
```

```
int add3(int a, int b, int c)
{
  int r = add2(a, b);
  r = r + c;
  return r;
}
```

```
add3:
    pushq   %rbx
    movl    %edx, %ebx
    movl    $0, %eax
    call    add2
    addl    %ebx, %eax
    popq    %rbx
    ret
```

Registers

First 6 Arguments: %rdi, %rsi, %rdx, %rcx, %r8, %r9

Return value: %rax

Example

```
int add2(int a, int b)
{
  return a + b;
}
```

```
int add3(int a, int b, int c)
{
  int r = add2(a, b);
  r = r + c;
  return r;
}
```

```
add2:
    leal    (%rdi,%rsi), %eax
    ret
```

```
add3:
    pushq   %rbx
    movl    %edx, %ebx
    movl    $0, %eax
    call    add2
    addl    %ebx, %eax
    popq    %rbx
    ret
```

save %rbx (callee-save) before overwriting it

%rdx (contains c) is caller save, i.e. may be changed by add2

c is copied to %ebx, which is callee save

restore %rbx before ret

Registers

First 6 Arguments: %rdi, %rsi, %rdx, %rcx, %r8, %9

Return value: %rax

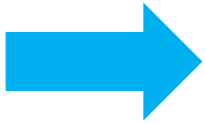
Today's lesson plan

- C/UNIX calling convention (caller vs. callee-save reg)
- Program data storage and manipuation

Local variables

- For primitive data types, use registers whenever possible
- Allocate local array/struct variables on the stack

```
int main() {
        int a[10];
        clear_array(a, 10);
        return 0;
}
```



```
main:
        subq    $48, %rsp
        movl    $10, %esi
        movq    %rsp, %rdi
        call    clear_array
        movl    $0, %eax
        addq    $48, %rsp
        ret
```

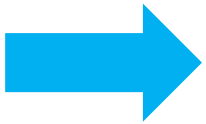
array allocation

array de-allocation

Global variables

- Allocated in a memory region called “data” segment
  - Statically allocated; compiler determines each global variable’s location in data segment.

```
int count = 0;

void inc() {
  count++;
}

int main() {
    inc();
}
```

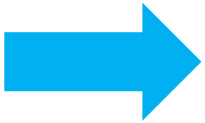


```
inc:
    addl $0x1, count(%rip)
    ret
main:
    ...
   call     add
   movl $0, %eax
   ...
```

Dynamically allocated space

- Allocated in a memory region called “heap”
  - Allocated by malloc library using sophisticated algorithms (discussed in later lecture)

```
int main() {
    int *x;
    x=malloc(100*sizeof(int));
    …
}
```



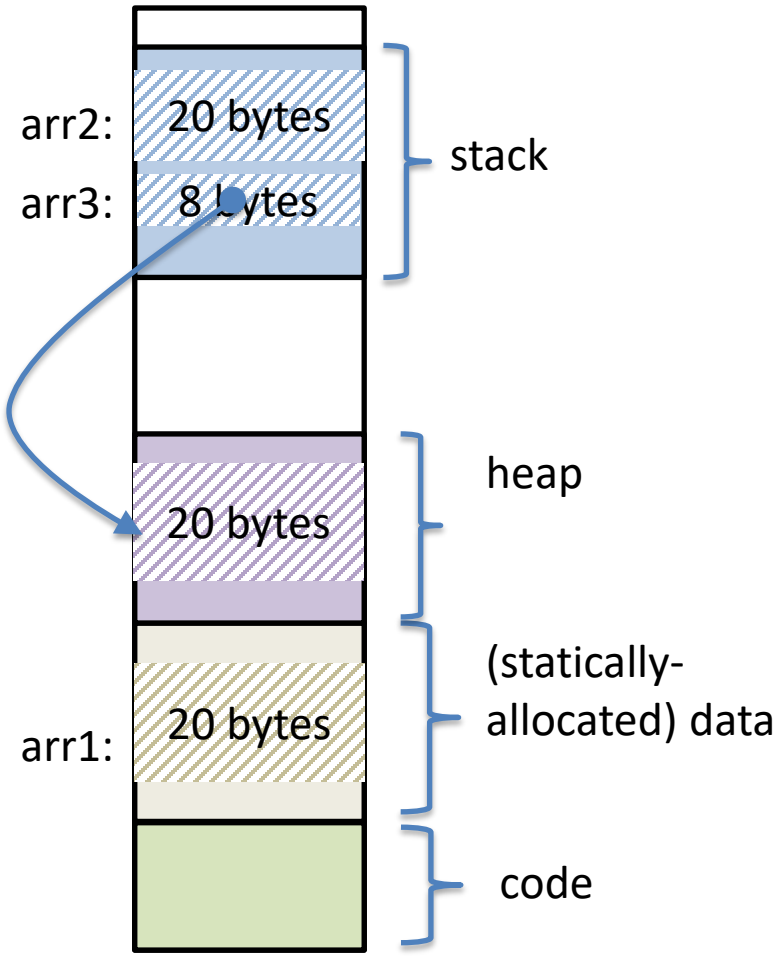
```
main:
    movl   $400 %edi
    call   malloc
    ...
```

Lots of code in this function

A process' memory regions

- A running program (process)'s memory consists of code, data, stack, heap (and code/data of its shared libraries)

```
int arr1[5];

int main() {

    int arr2[5];
    int *arr3;
    arr3 = malloc(sizeof(int)*5);
}
```



Accessing program data: primitive types

- Local variables of primitive data types are commonly stored in regs

count is a local variable (stored in %edi)

count++;

```
addl $1, %edi
```

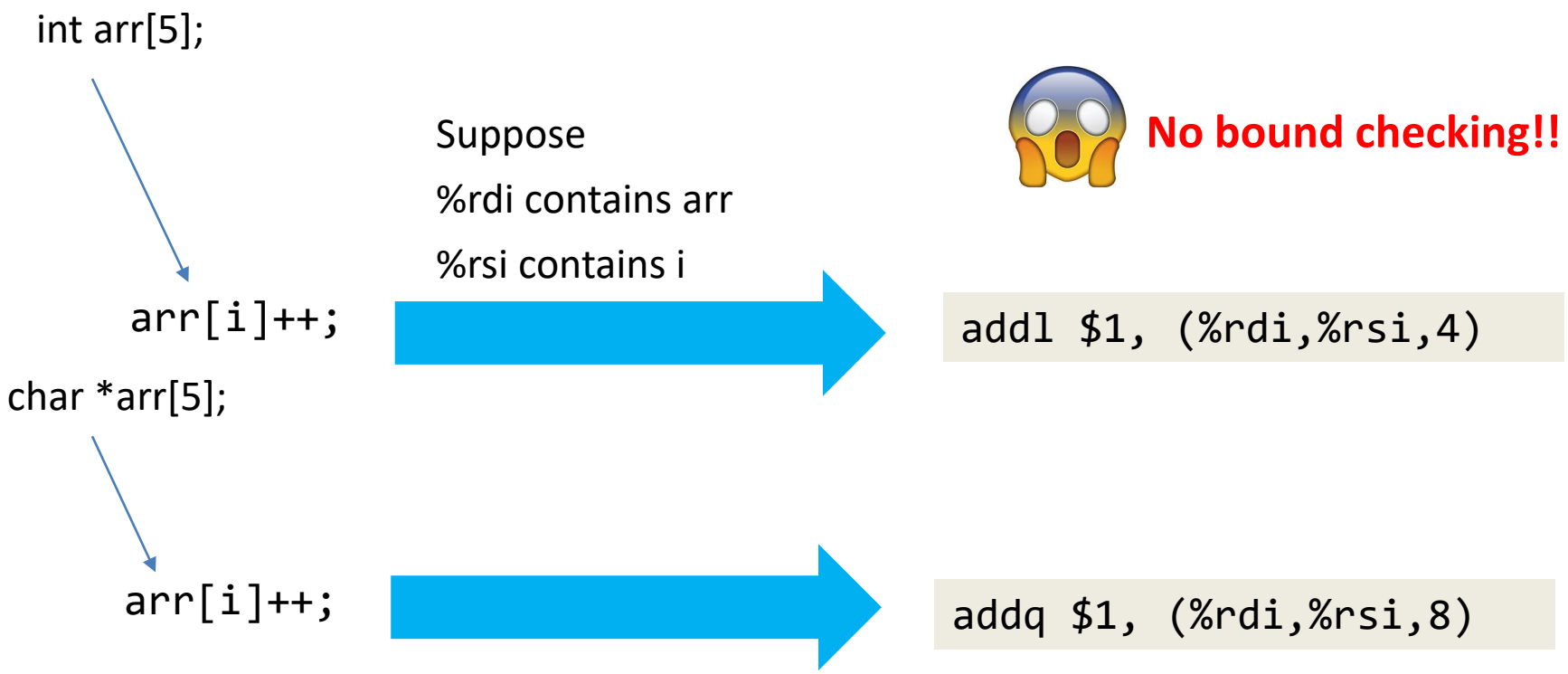
count is a global variable

count++;

```
addl $1, 0x200a13(%rip)
```

Accessing program data: arrays

- Arrays are always stored in the memory (stack, heap or data)



Binary Puzzle 1

```
void mystery(int *arr, int n) {
    ???
}
```

```
  movl $0, %eax
  jmp  .L3
.L4:
  movslq %eax, %rdx
  addl $1, (%rdi,%rdx,4)
  addl $1, %eax
.L3:
  cmpl %esi, %eax
  jl   .L4
  ret
```

%rdi has the value of arr
%esi has the value of n

Binary Puzzle 1

```
void mystery(int *arr, int n) {
    ???
}
```

```
  movl $0, %eax
  jmp  .L3
.L4:
  movslq %eax, %rdx
  addl $1, (%rdi,%rdx,4)
  addl $1, %eax
.L3:
  cmpl %esi, %eax
  jl   .L4
  ret
```

```
  a = 0;
  goto .L3
```

%rdi has the value of arr
%esi has the value of n

Binary Puzzle 1

```
void mystery(int *arr, int n) {
   ???
}
```

```
  movl $0, %eax
  jmp  .L3
.L4:
  movslq %eax, %rdx
  addl $1, (%rdi,%rdx,4)
  addl $1, %eax
.L3:
  cmpl %esi, %eax
  jl   .L4
  ret
```

```
  a = 0;
  goto .L3



.L3:
  if a < n
       goto .L4

  return
```

%rdi has the value of arr
%esi has the value of n

Binary Puzzle 1

```
void mystery(int *arr, int n) {
   ???
}
```

```
  movl $0, %eax
  jmp  .L3
.L4:
  movslq %eax, %rdx
  addl $1, (%rdi,%rdx,4)
  addl $1, %eax
.L3:
  cmpl %esi, %eax
  jl   .L4
  ret
```

```
  a = 0;
  goto .L3
.L4
  arr[a] = arr[a] + 1
  a++

.L3:
  if a < n
     goto .L4
  return
```

%rdi has the value of arr
%esi has the value of n

Binary Puzzle 1

type of a?

```
void mystery(int *arr, int n) {
   for(    int i = 0; i < n; i++)
   {
     arr[i] = arr[i] + 1;
   }
}
```

```
  movl $0, %eax
  jmp  .L3
.L4:
  movslq %eax, %rdx
  addl $1, (%rdi,%rdx,4)
  addl $1, %eax
.L3:
  cmpl %esi, %eax
  jl   .L4
  ret
```

```
  a = 0;
  goto .L3
.L4
  arr[a] = arr[a] + 1
  a++
.L3:
  if a < n
     goto .L4
  return
```

%rdi has the value of arr
%esi has the value of n

Binary puzzle 2

```
?? mystery(char *s) {

      ???
}
```

%rdi contains s

```
    movl    $0x0,%eax
    jmp     L1.
L2.
    addl    $0x1,%eax
L1.
    movslq %eax,%rdx
    cmpb   $0x0,(%rdi,%rdx,1)
    jne    L2.
    ret
```

Binary puzzle 2

```
?? mystery(char *s) {

      ???

}
```

%rdi contains s

```
    movl    $0x0,%eax
    jmp     L1.
L2.
    addl    $0x1,%eax
L1.
    movslq %eax,%rdx
    cmpb   $0x0,(%rdi,%rdx,1)
    jne     L2.
    ret
```

```
int a = 0;
goto L1;
```

Binary puzzle 2

```
?? mystery(char *s) {

      ???

}
```

%rdi contains s

```
    movl    $0x0,%eax
    jmp     L1.
L2.
    addl    $0x1,%eax
L1.
    movslq %eax,%rdx
    cmpb   $0x0,(%rdi,%rdx,1)
    jne     L2.
    ret
```

```
        int a = 0;
        goto L1;


L1.
        long d = a;
```

Binary puzzle 2

```
?? mystery(char *s) {

      ???

}
```

%rdi contains s

```
    movl    $0x0,%eax
    jmp     L1.
L2.
    addl    $0x1,%eax
L1.
    movslq %eax,%rdx
    cmpb   $0x0,(%rdi,%rdx,1)
    jne     L2.
    ret
```

```
    int a = 0;
    goto L1;


L1.
    long d = a;
    if(0 != s[d])
        goto L2;
```

Binary puzzle 2

```
?? mystery(char *s) {

      ???

}
```

%rdi contains s

```
    movl    $0x0,%eax
    jmp     L1.
L2.
    addl    $0x1,%eax
L1.
    movslq %eax,%rdx
    cmpb   $0x0,(%rdi,%rdx,1)
    jne     L2.
    ret
```

```
    int a = 0;
    goto L1;
L2.
    a = a  + 1;
L1.
    long d = a;
    if(0 != s[d])
       goto L2;
```

Binary puzzle 2

```
int mystery(char *s) {

    int a = 0;
    while(s[a]) {
        a = a + 1;
    }
    return a;
}
```

%rdi contains s

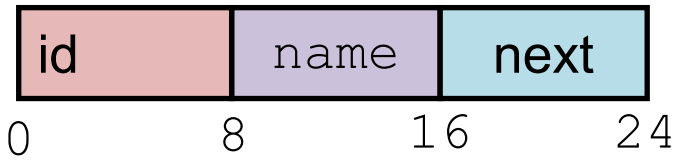
```
      movl    $0x0,%eax
      jmp     L1.
  L2.
      addl    $0x1,%eax
  L1.
      movslq %eax,%rdx
      cmpb   $0x0,(%rdi,%rdx,1)
      jne     L2.
      ret
```

```
    int a = 0;
    goto L1;
L2.
    a = a  + 1;
L1.
    long d = a;
    if(0 != s[d]) {
       goto L2;
    }
    ret;
```

Accessing Program Data: struct

- Struct is stored in the memory
  - Fields are contiguous in the order they are declared in struct
  - There may be padding (gaps) between fields

```
typedef struct node {
    long id;
    char *name;
    struct node *next;
}node;
```




```
n->id = 10;
n->name = NULL;
n->next = n;
```

%rdi contains n

```
movq    $10, (%rdi)
movq    $0, 8(%rdi)
movq    %rdi, 16(%rdi)
```

Binary Puzzle 3

```
?? mystery(node *n, long id) {
   ???
}
```

```
  jmp     .L1
.L3:
  cmpq    %rsi, (%rdi)
  jne     .L2
  movq    8(%rdi), %rax
  ret
.L2:
  movq    16(%rdi), %rdi
.L1:
  testq   %rdi, %rdi
  jne     .L3
  movq    $0, %rax
  ret
```

%rdi has the value of n
%rsi has the value of id
%rax is to contain return value

Binary Puzzle 3

```
?? mystery(node *n, long id) {
   ???
}
```

```
  jmp     .L1
.L3:
  cmpq    %rsi, (%rdi)
  jne     .L2
  movq    8(%rdi), %rax
  ret
.L2:
  movq    16(%rdi), %rdi
.L1:
  testq   %rdi, %rdi
  jne     .L3
  movq    $0, %rax
  ret
```

```
goto .L1
```

%rdi has the value of n
%rsi has the value of id
%rax is to contain return value

Binary Puzzle 3

```
?? mystery(node *n, long id) {
   ???
}
```

```
  jmp     .L1
.L3:
  cmpq    %rsi, (%rdi)
  jne     .L2
  movq    8(%rdi), %rax
  ret
.L2:
  movq    16(%rdi), %rdi
.L1:
  testq   %rdi, %rdi
  jne     .L3
  movq    $0, %rax
  ret
```

```
goto .L1
```

```
.L1:
  if (n != 0)
     goto .L3
```

%rdi has the value of n
%rsi has the value of id
%rax is to contain return value

Binary Puzzle 3

```
?? mystery(node *n, long id) {
   ???
}
```

```
  jmp     .L1
.L3:
  cmpq    %rsi, (%rdi)
  jne     .L2
  movq    8(%rdi), %rax
  ret
.L2:
  movq    16(%rdi), %rdi
.L1:
  testq   %rdi, %rdi
  jne     .L3
  movq    $0, %rax
  ret
```

```
  goto .L1

.L1:
  if (n != 0)
     goto .L3
  return 0;
```

%rdi has the value of n
%rsi has the value of id
%rax is to contain return value

Binary Puzzle 3

```
?? mystery(node *n, long id) {
   ???
}
```

```
  jmp     .L1
.L3:
  cmpq    %rsi, (%rdi)
  jne     .L2
  movq    8(%rdi), %rax
  ret
.L2:
  movq    16(%rdi), %rdi
.L1:
  testq   %rdi, %rdi
  jne     .L3
  movq    $0, %rax
  ret
```

```
  goto .L1
.L3:
  if (*((long *)n) != id)
     goto .L2

.L1:
  if (n != 0)
     goto .L3
  return 0;
```

n->id != id

%rdi has the value of n
%rsi has the value of id
%rax is to contain return value

Binary Puzzle 3

```
?? mystery(node *n, long id) {
   ???
}
```

```
  jmp     .L1
.L3:
  cmpq    %rsi, (%rdi)
  jne     .L2
  movq    8(%rdi), %rax
  ret
.L2:
  movq    16(%rdi), %rdi
.L1:
  testq   %rdi, %rdi
  jne     .L3
  movq    $0, %rax
  ret
```

```
  goto .L1;
.L3:
  if (n->id != id)
     goto .L2;
  return n->name;

.L1:
  if (n != 0)
     goto .L3;
  return 0;
```

%rdi has the value of n
%rsi has the value of id
%rax is to contain return value

Binary Puzzle 3

```
?? mystery(node *n, long id) {
   ???
}
```

```
  jmp     .L1
.L3:
  cmpq    %rsi, (%rdi)
  jne     .L2
  movq    8(%rdi), %rax
  ret
.L2:
  movq    16(%rdi), %rdi
.L1:
  testq   %rdi, %rdi
  jne     .L3
  movq    $0, %rax
  ret
```

```
  goto .L1;
.L3:
  if (n->id != id)
     goto .L2;

   return n->name;
.L2
    n = n->next;

.L1:
  if (n != 0)
     goto .L3;
  return 0;
```

%rdi has the value of n
%rsi has the value of id
%rax is to contain return value

Binary Puzzle 3

```
char *mystery(node *n, long id) {
   while (n) {
      if (n->id == id)
         return n->name;
      n= n->next;
   }
   return NULL;
}
```

```
  jmp     .L1
.L3:
  cmpq    %rsi, (%rdi)
  jne     .L2
  movq    8(%rdi), %rax
  ret
.L2:
  movq    16(%rdi), %rdi
.L1:
  testq   %rdi, %rdi
  jne     .L3
  movq    $0, %rax
  ret
```

```
  goto .L1;
.L3:
  if (n->id != id)
     goto .L2;

   return n->name;
.L2
   n = n->next;
.L1:
  if (n != 0)
     goto .L3;
  return 0;
```

Summary

- How program data is stored and accessed
  - Primitive data types
  - Arrays
  - Structs
- Separate memory regions for stack, heap, data