Structs and Alignment

CSE 351 Autumn 2016

Instructor:

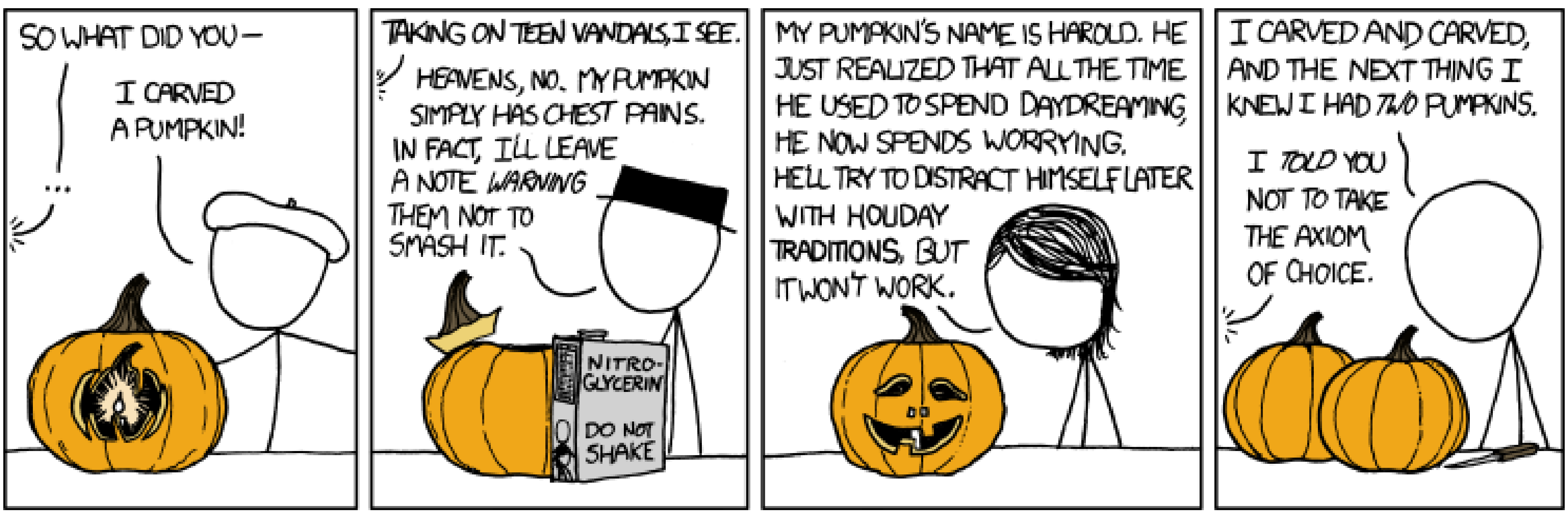
Justin Hsia

Teaching Assistants:

Chris Ma, Hunter Zahn, John Kaltenbach, Kevin Bi

Sachin Mehta, Suraj Bhat, Thomas Neuman, Waylon Huang

Xi Liu, Yufang Sun



http://xkcd.com/804/

Administrivia

- Homework 2 due Friday
- Lab 3 released today

- **Midterm** next lecture
  - Try to come early to settle in; starting promptly
  - Make a cheat sheet! – two-sided letter page, *handwritten*
  - Midterm details Piazza post: @225
- Review session tonight from 5-7pm in EEB 105
- Extra office hours
  - Justin Tue 11/1, 12:30-4:30pm, CSE 438

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8 Mac

Computer system:

Data Structures in Assembly

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structs
  - Alignment
- Unions

Structs in C

- Way of defining compound data types
- A structured group of variables, possibly including other structs

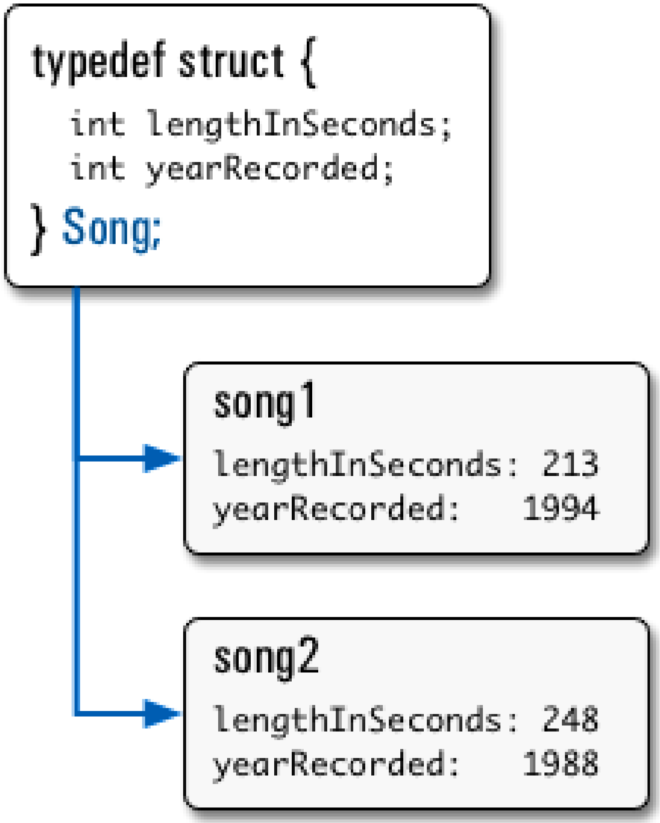
```
typedef struct {
  int lengthInSeconds;
  int yearRecorded;
} Song;

Song song1;

song1.lengthInSeconds =  213;
song1.yearRecorded    = 1994;

Song song2;

song2.lengthInSeconds =  248;
song2.yearRecorded    = 1988;
```



Struct Definitions

- Structure definition:
  - Does NOT declare a variable
  - Variable type is "`struct name`"

```
struct name {
    /* fields */
};
```

Easy to forget semicolon!

```
struct name name1, *pn, name_ar[3];
```

pointer (→ `*pn`), array (→ `name_ar[3]`)

- Joint struct definition and typedef
  - Don't need to give struct a name in this case

```
struct nm {
   /* fields */
};
typedef struct nm name;
name n1;
```

→

```
typedef struct {
    /* fields */
} name;
name n1;
```

Scope of Struct Definition

- Why is placement of struct definition important?
  - What actually happens when you declare a variable?
    - Creating space for it somewhere!
  - Without definition, program doesn't know how much space

```
struct data {
   int ar[4];
   long d;
};
```

← Size = _____ bytes

Size = _____ bytes →

```
struct rec {
   int a[4];
   long i;
   struct rec* next;
};
```

- Almost always define structs in global scope near the top of your C file
  - Struct definitions follow normal rules of scope

Accessing Structure Members

```
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
```

- Given a struct instance, access member using the `.` operator:

```
struct rec r1;
r1.i = val;
```

- Given a *pointer* to a struct:

```
struct rec *r;
r = &r1;  // or malloc space for r to point to
```

  We have two options:
  - Use `*` and `.` operators: `(*r).i = val;`
  - Use `->` operator for short: `r->i = val;`

- **In assembly:** pointer holds address of the first byte
  - Access members with offsets

Review: Structs in Lab 0

```
// Use typedef to create a type: FourInts
typedef struct {
   int a, b, c, d;
} FourInts;   // Name of type is "FourInts"

int main(int argc, char* argv[]) {
   FourInts f1;  // Allocates memory to hold a FourInts
                 // (16 bytes) on stack (local variable)
   f1.a = 0;     // Assign first field in f1 to be zero

   FourInts* f2; // Declare f2 as a pointer to FourInts

   // Allocate space for a FourInts on the heap,
   //   f2 is a "pointer to"/"address of" this space.
   f2 = (FourInts*) malloc(sizeof(FourInts));
   f2->b = 17;   // Assign the second field to be 17
   ...
}
```

Java side-note

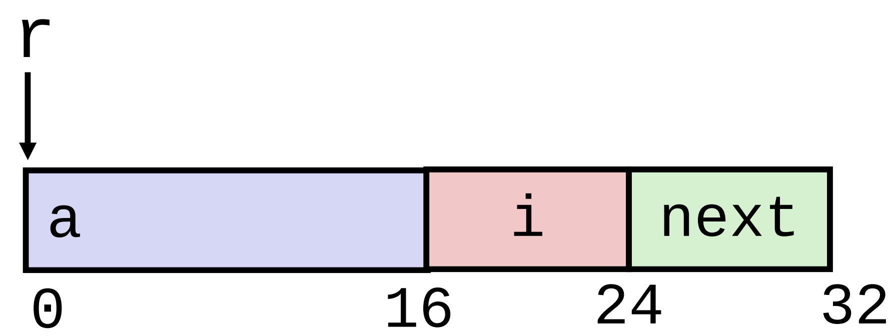
```
class Record { ... }
Record x = new Record();
```

- An instance of a class is like a *pointer to* a struct containing the fields
  - (Ignoring methods and subclassing for now)
  - So Java's `x.f` is like C's `x->f` or `(*x).f`

- In Java, almost everything is a pointer ("*reference*") to an object
  - Cannot declare variables or fields that are structs or arrays
  - Always a *pointer* to a struct or array
  - So every Java variable or field is ≤ 8 bytes (but can point to lots of data)

Structure Representation

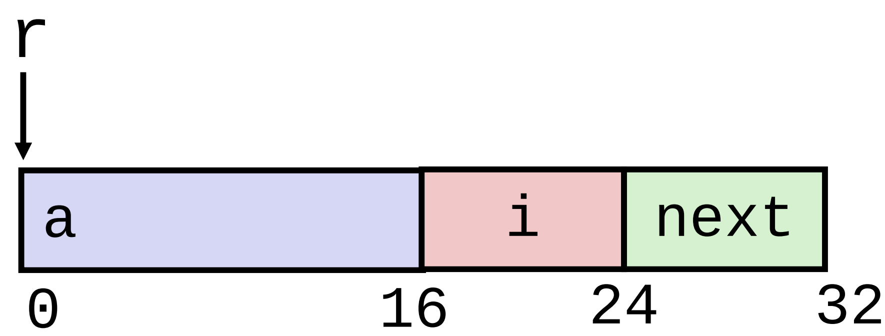
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} *r;
```



- Characteristics
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

Structure Representation

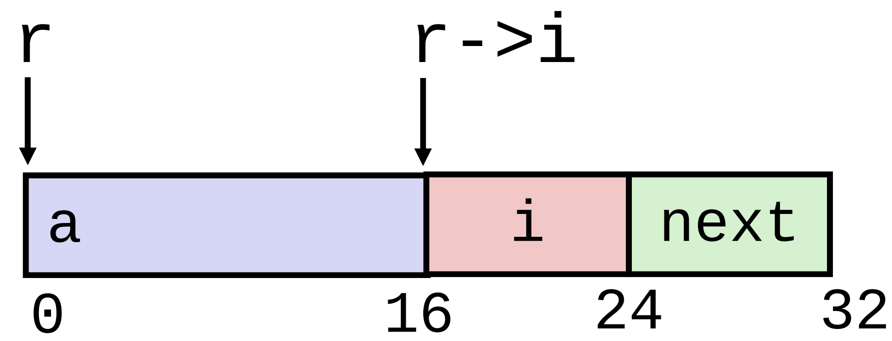
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} *r;
```



- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration order
  - Even if another ordering would be more compact
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

Accessing a Structure Member

```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} *r;
```



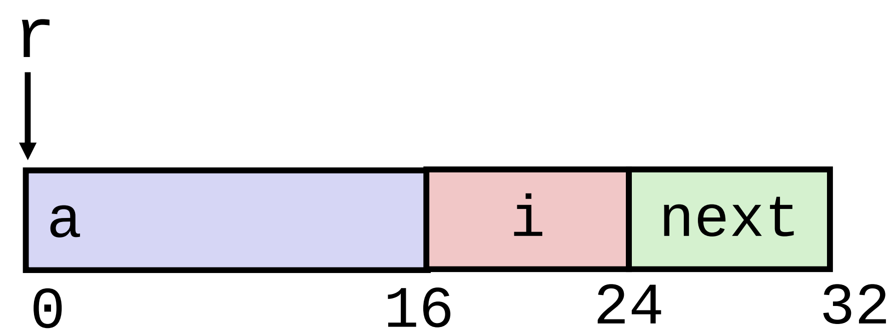
- Compiler knows the *offset* of each member within a struct
  - Compute as `*(r+offset)`
    - Referring to absolute offset, so no pointer arithmetic

```
long get_i(struct rec *r)
{
  return r->i;
}
```

```
  # r in %rdi, index in %rsi
  movq  16(%rdi), %rax
  ret
```

Exercise: Pointer to Structure Member

```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} *r;
```




```
long* addr_of_i(struct rec *r)
{
  return &(r->i);
}
```

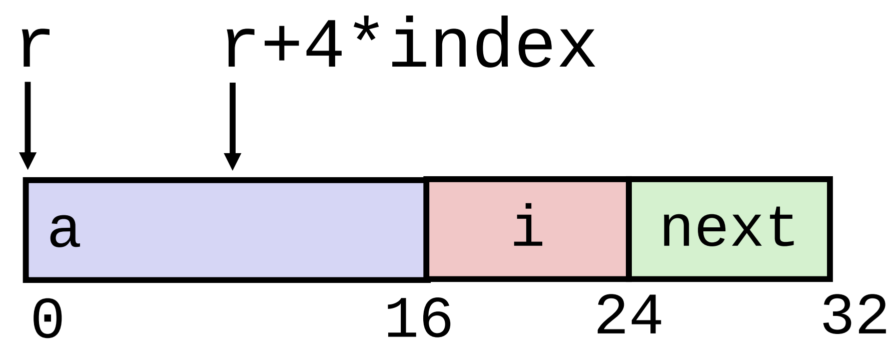
```
  # r in %rdi
  ______   __________,%rax
  ret
```

```
struct rec* addr_of_next(struct rec *r)
{
  return &(r->next);
}
```

```
  # r in %rdi
  ______   __________,%rax
  ret
```

Generating Pointer to Array Element

```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} *r;
```



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as: `r+4*index`

```
int* find_addr_of_array_elem
  (struct rec *r, long index)
{
  return &r->a[index];
}
```

&(r->a[index])

```
  # r in %rdi, index in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

Review: Memory Alignment in x86-64

- For good memory system performance, Intel recommends data be aligned
  - However the x86-64 hardware will work correctly regardless of alignment of data
- *Aligned* means that any primitive object of K bytes must have an address that is a multiple of K
- Aligned addresses for data types:

K	Type	Addresses
1	`char`	No restrictions
2	`short`	Lowest bit must be zero: ...0_2
4	`int, float`	Lowest 2 bits zero: ...00_2
8	`long, double, *`	Lowest 3 bits zero: ...000_2
16	`long double`	Lowest 4 bits zero: ...0000_2

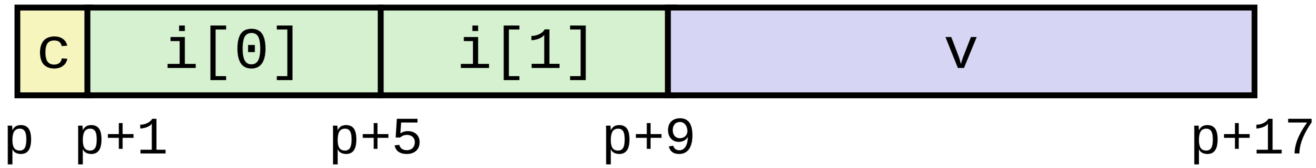
Alignment Principles

- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on x86-64

- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store value that spans quad word boundaries
    - Virtual memory trickier when value spans 2 pages (more on this later)

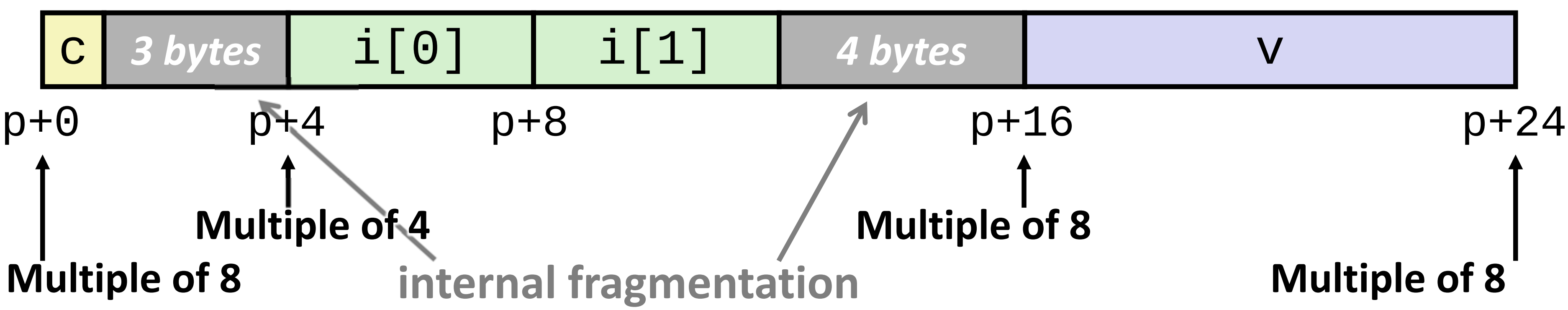
Structures & Alignment

- Unaligned Data




```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

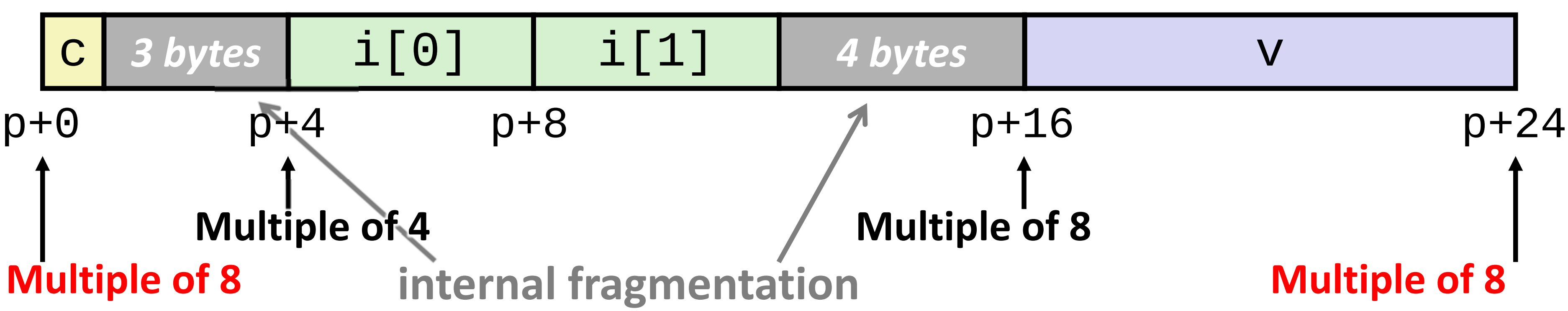
- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K



Satisfying Alignment with Structures (1)

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

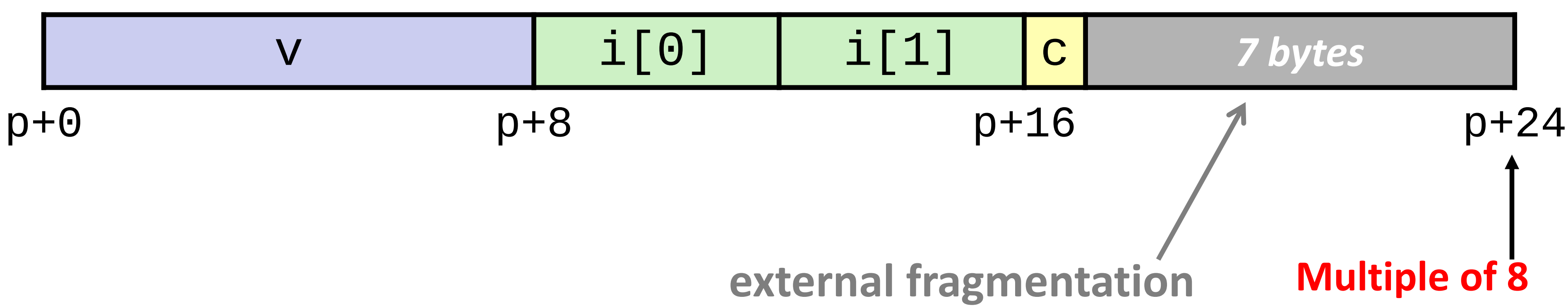
- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement K_{max}
    - K_{max} = Largest alignment of any element
    - Counts array elements individually as elements
  - **Address of structure & structure length must be multiples of K_{max}**
- Example:
  - K_{max} = 8, due to `double` element



Satisfying Alignment with Structures (2)

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```

- Can find offset of individual fields using `offsetof()`
  - Need to `#include <stddef.h>`
  - Example: `offsetof(struct S2,c)` returns 16
- For largest alignment requirement K_{max}, overall structure size must be multiple of K_{max}
  - Compiler will add padding at end of structure to meet overall structure alignment requirement



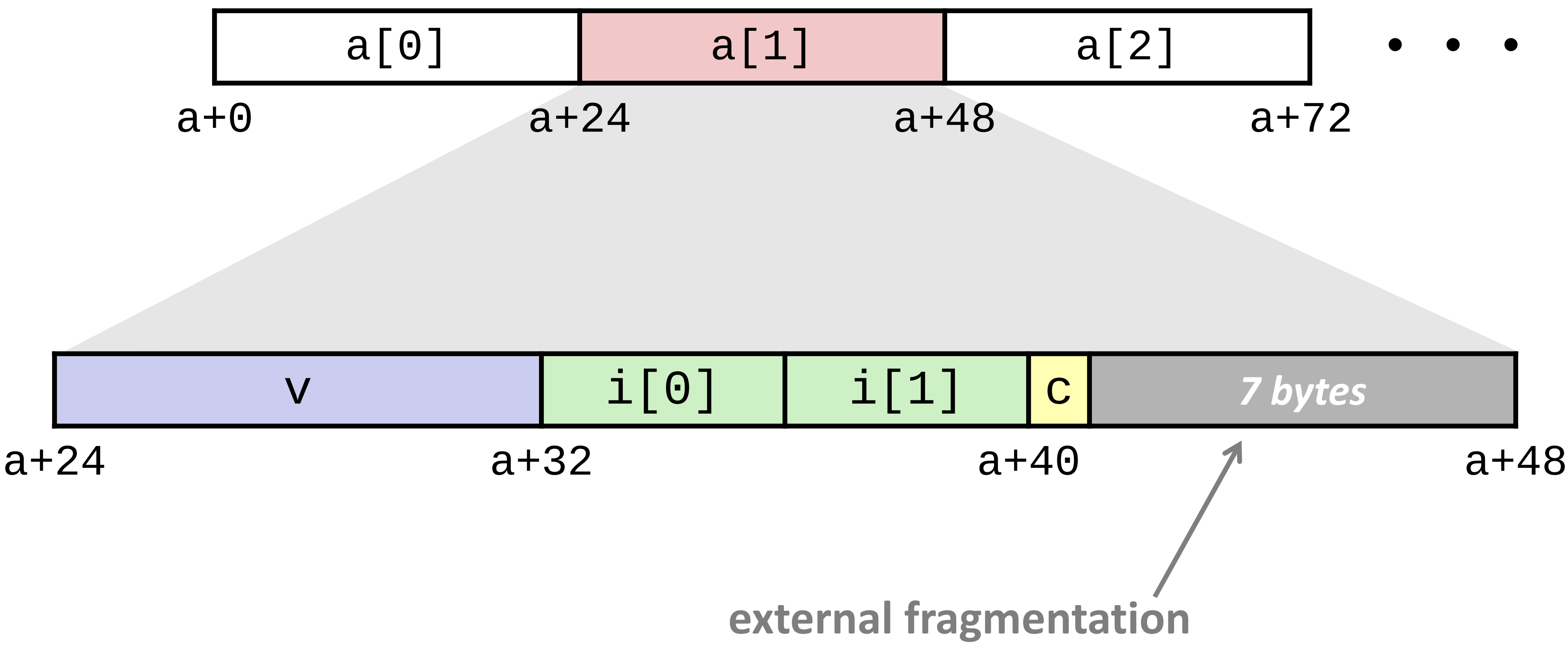
Alignment of Structs

- Compiler will do the following:
  - Maintains declared *ordering* of fields in struct
  - Each ***field*** must be aligned *within* the struct *(may insert padding)*
    - `offsetof` can be used to get actual field offset
  - Overall struct must be ***aligned*** according to largest field
  - Total struct ***size*** must be multiple of its alignment *(may insert padding)*
    - `sizeof` should be used to get true size of structs

Arrays of Structures

- Overall structure length multiple of K_{max}
- Satisfy alignment requirement for every element in array

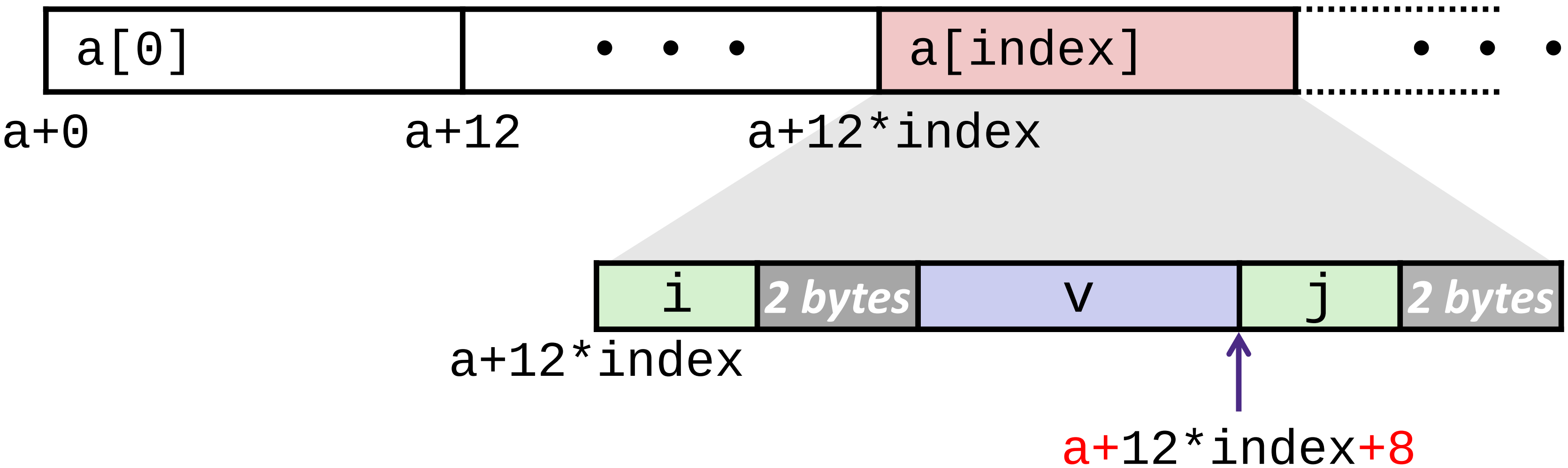
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

- Compute start of array element as: `12*index`
  - `sizeof(S3) = 12`, including alignment padding
- Element `j` is at offset 8 within structure
- Assembler gives offset `a+8`

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```




```
short get_j(int index)
{
  return a[index].j;
}
```

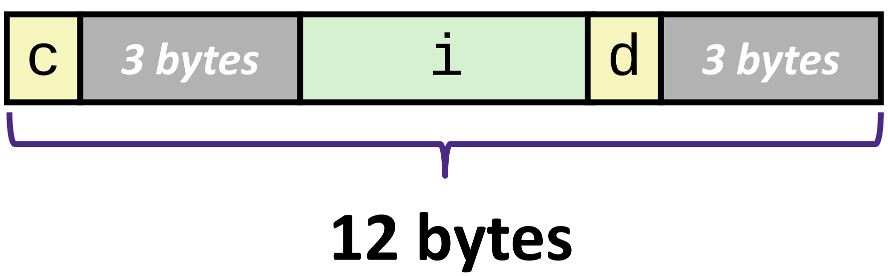
```
# %rdi = index
leaq (%rdi,%rdi,2),%rax   # 3*index
movzwl a+8(,%rax,4),%eax
```

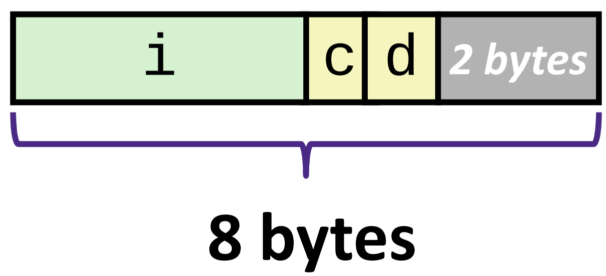
How the Programmer Can Save Space

- Compiler must respect order elements are declared in
  - Sometimes the programmer can save space by declaring large data types first

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```





Peer Instruction Question

- Minimize the size of the struct by re-ordering the vars

```
struct old {
  int i;

  short s[3];

  char *c;

  float f;
};
```

→

```
struct new {
  int     i;

  ______ ______;

  ______ ______;

  ______ ______;
};
```

- What are the old and new sizes of the struct?

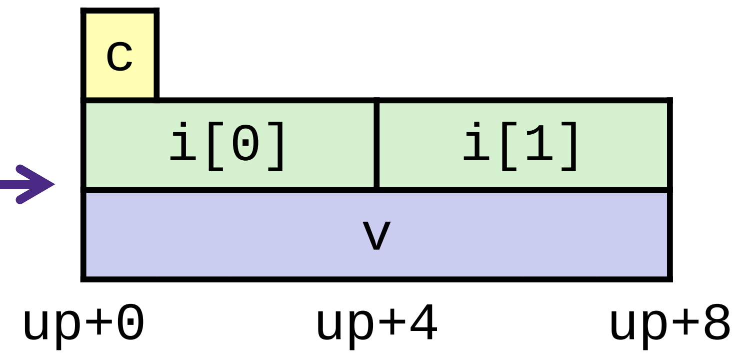
  sizeof(struct old) = _____   sizeof(struct new) = _____

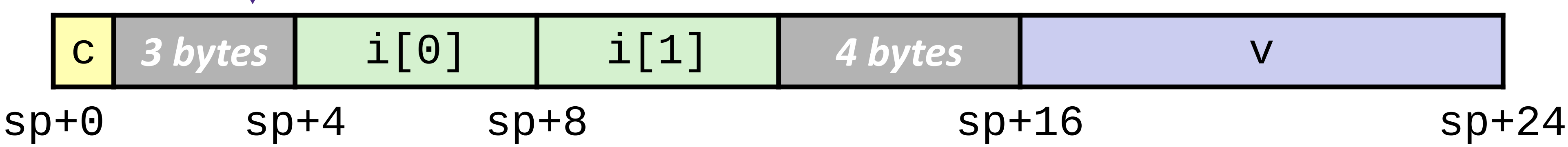
Unions

- Only allocates enough space for the largest element in union
- Can only use one member at a time

```
struct S {
  char c;
  int i[2];
  double v;
} *sp;
```

```
union U {
  char c;
  int i[2];
  double v;
} *up;
```





What Are Unions Good For?

- Unions allow the same region of memory to be referenced as different types
  - Different "views" of the same memory location
  - Can be used to circumvent C's type system (bad idea and technically not guaranteed to work)
- Better idea: use a struct inside a union to access some memory location either as a whole or by its parts
  - But watch out for endianness at a small scale…
- Layout details are implementation/machine-specific…

```
union int_or_bytes {
    int i;
    struct bytes {
      char b0, b1, b2, b3;
    }
}
```

Example: Simulated Condition Flags

- Simulating an x86-64 processor in C
  - Each flag only requires 1 bit, no need to use more space
  - Set after most instructions (e.g. arithmetic, `test`, `cmp`)

```
typedef union {
   char all;
   struct {
      unsigned char unused : 4;
      unsigned char CF : 1;
      unsigned char ZF : 1;
      unsigned char SF : 1;
      unsigned char OF : 1;
    } flags;
} FLAGS;
FLAGS cond_reg;
```

specified bit widths

CF *Carry Flag* ZF *Zero Flag* SF *Sign Flag* OF *Overflow Flag*

Example: Simulated Condition Flags

- Simulating an x86-64 processor in C
  - Each flag only requires 1 bit, no need to use more space
  - Set after most instructions (e.g. arithmetic, `test`, `cmp`)

```
void set_flags(long a, long b, long r) {
        // condition for CF is complicated
        //    without access to ALU,
        //    so omitted from this demo.
        cond_reg.flags.ZF = !r;
        cond_reg.flags.SF = (r<0);
        cond_reg.flags.OF = (a>0 && b>0 && r<0)
                         || (a<0 && b<0 && r>0);
}
```

CF	*Carry Flag*	ZF	*Zero Flag*	SF	*Sign Flag*	OF	*Overflow Flag*

Summary

- Arrays in C
  - Aligned to satisfy every element’s alignment requirement
- Structures
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- Unions
  - Provide different views of the same memory location

BONUS SLIDES

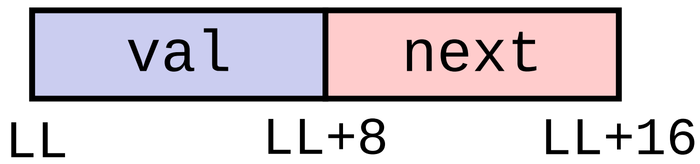
Overview of a basic linked list. You may have encountered this during Lab 2.

- Compiler Explorer link: https://godbolt.org/g/Sbsd1r
- Linked lists are a common example of structs and pointers (both in C and assembly)
- You won't be tested on assembly directives

Linked List Example

- Generate a (singly-) linked list of values:

```
typedef struct N {
   long val;
   struct N *next;
} Node;
typedef Node * List
// "head" of linked list
List LL = NULL;
```



- Creating and destroying Nodes:

```
// dynamically allocate - don't know how many
Node *newNode = (Node *)malloc(sizeof(Node));

// get rid of Node by freeing it (ptr still exists)
free(newNode);
newNode = NULL; // optional
```

Example Use of Linked List

```
int i;
List LL = NULL;
LL = addNode(LL,1); // add node at start of list
LL = addNode(LL,5);
LL = addNode(LL,3);

for(i=-1;i<4;i++)
   printf("node %d = %ld\n",i,getNode(LL,i));
```

```
unix> ./linkedlist
node -1 = -1
node 0 = 3
node 1 = 5
node 2 = 1
node 3 = -1
```

Add a Node at Head of List

- Returns new head of list (the added node)

```
List addNode(List list, long v)
{
   Node *node = (Node *)
      malloc(sizeof(Node));
   node->val = v;
   node->next = list;
   return node;
}
```

- Let's examine how this works for the 3rd call:
  `LL = addNode(LL, 3);`

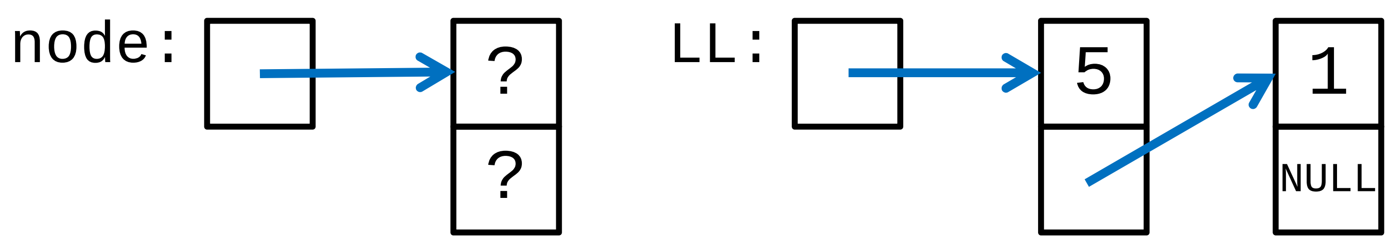
```
addNode(N*, long):
   pushq   %rbp
   pushq   %rbx
   subq    $8, %rsp
   movq    %rdi, %rbx
   movq    %rsi, %rbp
   movl    $16, %edi
   call    malloc
   movq    %rbp, (%rax)
   movq    %rbx, 8(%rax)
   addq    $8, %rsp
   popq    %rbx
   popq    %rbp
   ret
```

Add a Node at Head of List

- Line 1: Create new node (and pointer to it)
  - Uninitialized space in the Heap returned by `malloc()`

```
List addNode(List list, long v)
{
   Node *node = (Node *)
      malloc(sizeof(Node));
   node->val = v;
   node->next = list;
   return node;
}
```

```
addNode(N*, long):
   ...
   subq     $8, %rsp
   ...
   movl     $16, %edi
   call     malloc
   ...
```

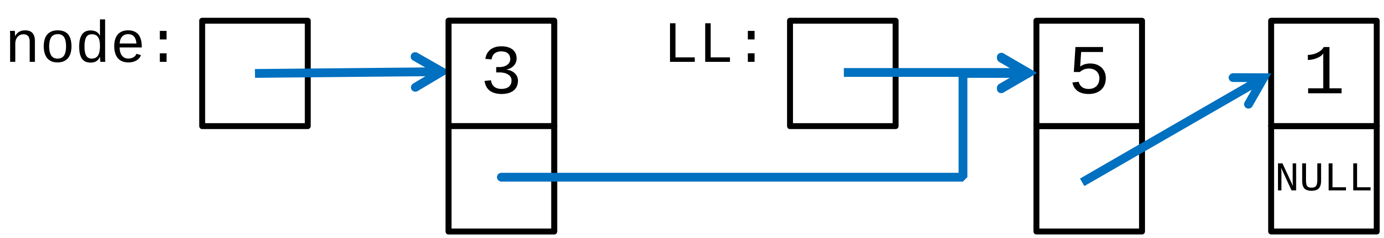


Add a Node at Head of List

- Line 2 & 3: Initialize new node

```
List addNode(List list, long v)
{
   Node *node = (Node *)
      malloc(sizeof(Node));
   node->val = v;
   node->next = list;
   return node;
}
```

```
addNode(N*, long):
   ...
   movq    %rbp, (%rax)
   movq    %rbx, 8(%rax)
   ...
```

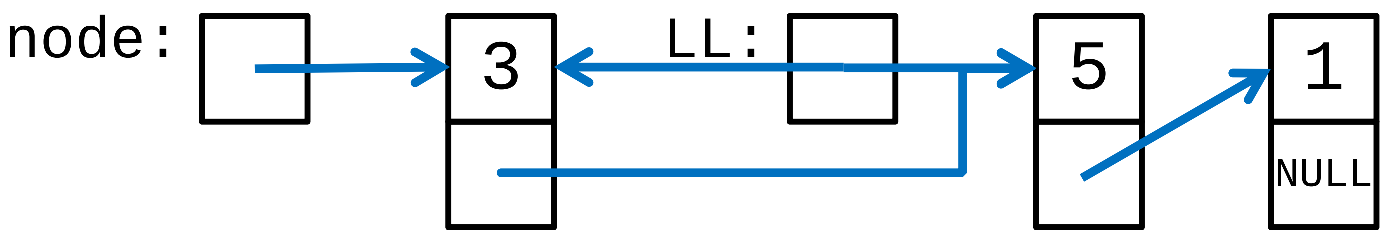


Add a Node at Head of List

- Line 4: Store new head of list back into LL variable
  - Local pointer `node` gets deallocated

```
List addNode(List list, long v)
{
    Node *node = (Node *)
        malloc(sizeof(Node));
    node->val = v;
    node->next = list;
    return node;
}
```

```
addNode(N*, long):
    ...
    addq    $8, %rsp
    ...
    ret
```



Get the n-th Value on Linked List

- Follow nodes in memory
  - End of list indicated when `next` field = NULL

```
long getNode(List list, int i) {
   int count = 0;
   while (list) {
      if (count==i)
         return list->val;
      count++;
      list = list->next;
   }
   return -1;
}
```

```
getNode:
   movl    $0, %eax
   jmp     .L4
.L7:
   cmpl    %esi, %eax
   jne     .L5
   movq    (%rdi), %rax
   ret
.L5:
   addl    $1, %eax
   movq    8(%rdi), %rdi
.L4:
   testq   %rdi, %rdi
   jne     .L7
   movq    $-1, %rax
   ret
```

- `setNode` to change value of n-th node looks very similar

Manually Creating Linked List in Assembly

- Initial data (e.g. global vars) placed in memory using assembly directives

Old list (3→5→1) using `addNode()`

```
   movl    $1, %esi
   movl    $0, %edi
   call    addNode
   movl    $5, %esi
   movq    %rax, %rdi
   call    addNode
   movl    $3, %esi
   movq    %rax, %rdi
   call    addNode
   movq    %rax, %rbp
```

New list (1→2→3) using assembly directives and labels (see `linkedlist.s`)

```
   movq     $N1, %rbp
   ...
   .data         # Static Data
   .align 16     # struct size
N1:
   .quad    1   # N1->val
   .quad    N2  # N1->next
N3:
   .quad    3   # N3->val
   .quad    0   # N3->next (NULL)
N2:
   .quad    2   # N2->val
   .quad    N3  # N2->next
```

Additional Linked List Functionality

- Think about how you might implement the following functions in C and what the x86-64 code probably looks like:
  - Remove a node from the list
  - Append a node to the *end* of the list
  - Delete/free and entire list
  - Join two lists together
  - Sort a list
- How would the functions change if the “value” we were storing in each node was a string instead of an integer?