Lecture 8: Data Structures in Assembly

CS 105

February 17, 2020

From Last Time…

```
int proc(int *p);

int example1(int x) {
  int a[4];
  a[3] = 10;
  return proc(a);
}
```

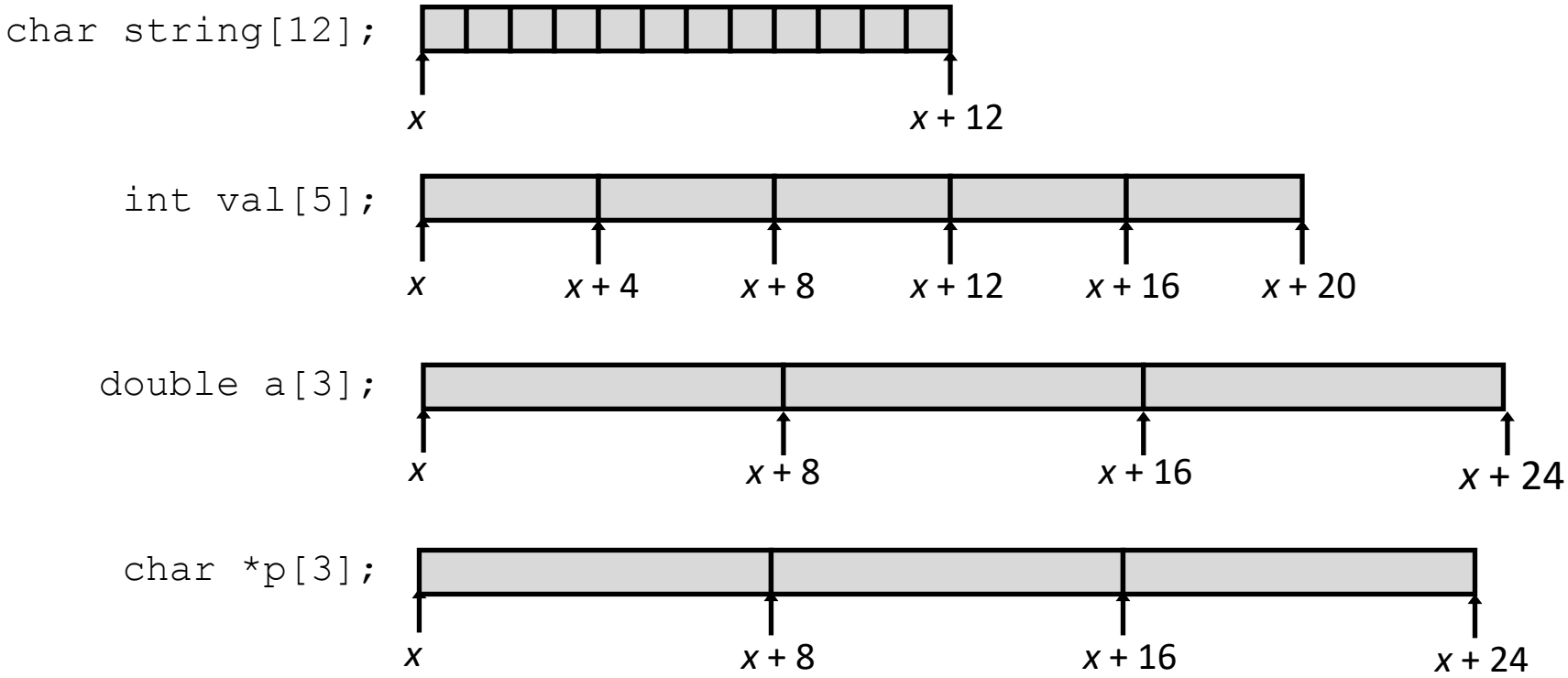
```
example1:
  subq   $16, %rsp
  movl   $10, 12(%rsp)
  movq   %rsp, %rdi
  call   0x400546 <proc>
  addq   $16, %rsp
  ret
```

Array Allocation

- Basic Principle

  T **`A[`***L***`];`**

  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **`sizeof`**`(`*T*`)` bytes in memory



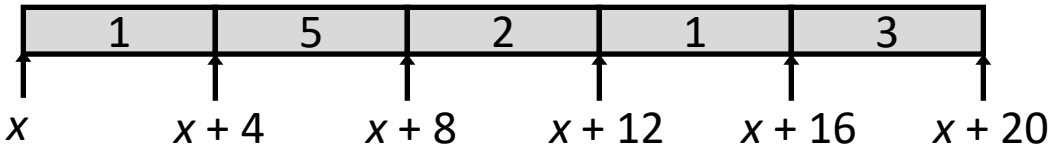
Array Access

- Basic Principle

  T **A[***L***];**

  - Array of data type *T* and length *L*
  - Identifier **A** can be used as a pointer to array element 0: Type *T**

```
int val[5];
```



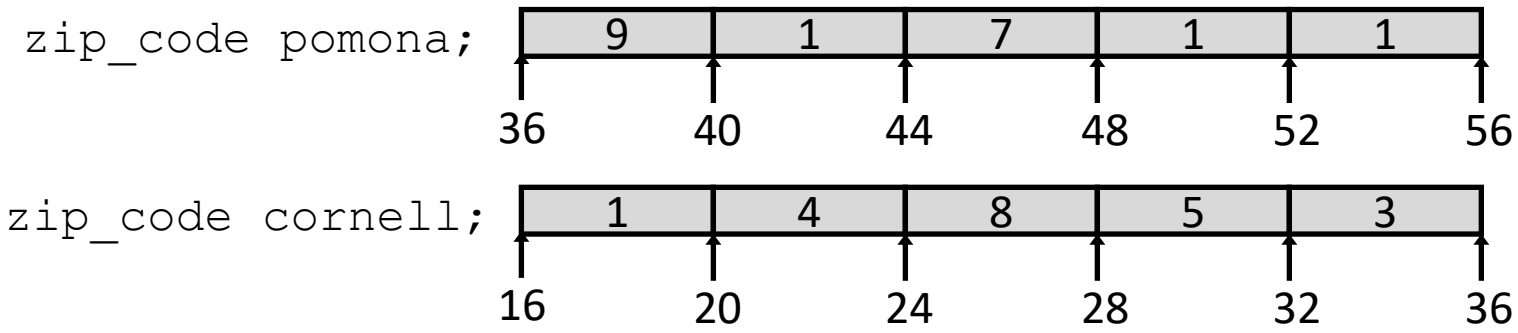
- Reference Type Value

```
val[4]
val
val+1
&val[2]
val[5]
*(val+1)
val + i
```

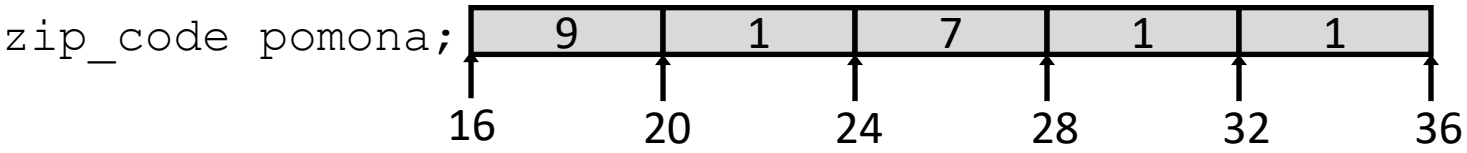
Array Example

```
#define ZLEN 5
typedef int zip_code[ZLEN];

zip_code pomona  = { 9, 1, 7, 1, 1 };
zip_code cornell = { 1, 4, 8, 5, 3 };
```

- Declaration "`zip_code pomona`" equivalent to "`int pomona[5]`"



Array Accessing Example




```
int get_digit(zip_code z, int digit){
  return z[digit];
}
```

```
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi + 4*%rsi`
- Use memory reference `(%rdi,%rsi,4)`

Array Loop Exercise

```
array_loop:
  xorl    %esi, %esi
  xorl    %eax, %eax
L1:
  addl    (%rdi,%rsi,4), %eax
  incq    %rsi
  cmpq    $5, %rsi
  jne     L1
  retq
```

```
int array_loop(zip_code z) {
  int sum = _______;
  int i;
  for(i = ___; i < ___; ___ )
    sum = ________;
  }

  return ________;
}
```

```
int array_loop(zip_code z){
  int sum = _______;
  int *p;
  for(p = ___; p < ___; ___ )
    sum = ________;
  }

  return ________;
}
```

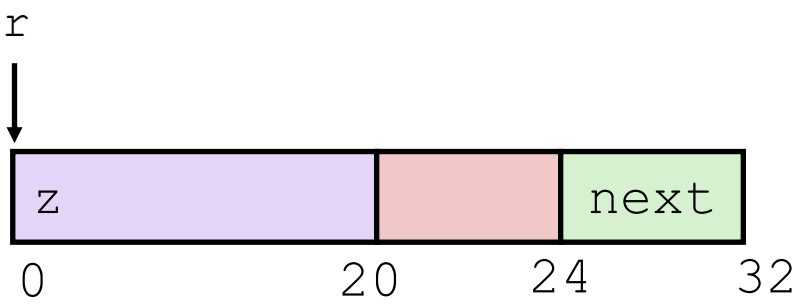
Array Loop Exercise

```
array_loop:
  xorl    %esi, %esi
  xorl    %eax, %eax
L1:
  addl    (%rdi,%rsi,4), %eax
  incq    %rsi
  cmpq    $5, %rsi
  jne     L1
  retq
```

```
array_r:
  xorl    %eax, %eax
  cmpl    $5, %esi
  je      L2
  pushq   %rbx
  movl    (%rdi,%rsi,4), %ebx
  incq    %rsi
  callq   array_r
  addl    %ebx, %eax
  popq    %rbx
L2:
  retq
```

Structure Representation

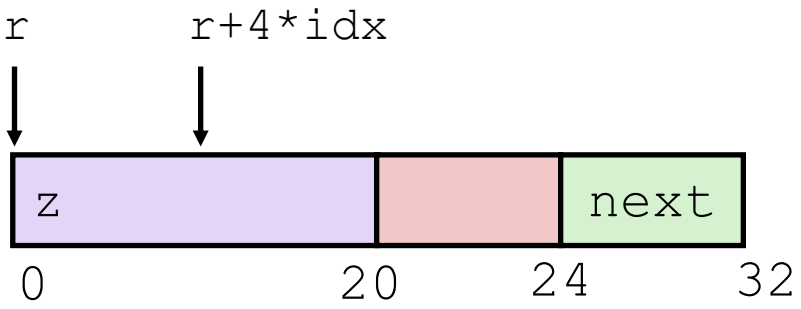
```
struct rec {
  zip_code z;
  struct rec *next;
};
```



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

Generating Pointer to Structure Member

```
struct rec {
  zip_code z;
  struct rec *next;
};
```



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

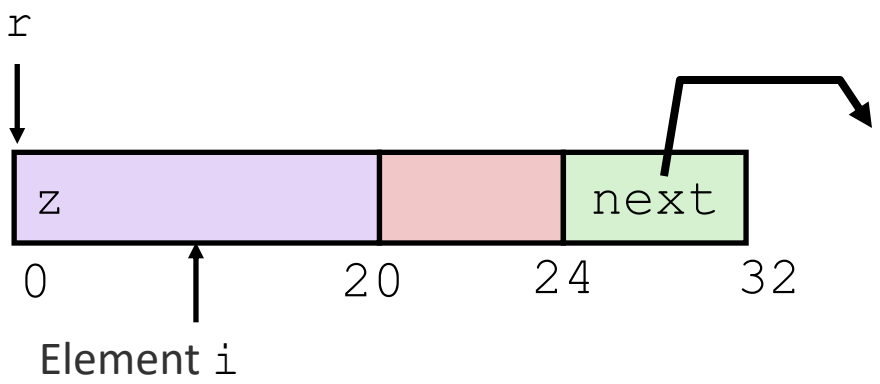
```
int *get_digit_ptr(struct rec *r, int idx){
  return &(r->z[idx]);
}
```

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

Register	Value
%rdi	p

Following Linked List

```
typedef struct rec {
  zip_code z;
  struct rec *next;
} zip_node;
```



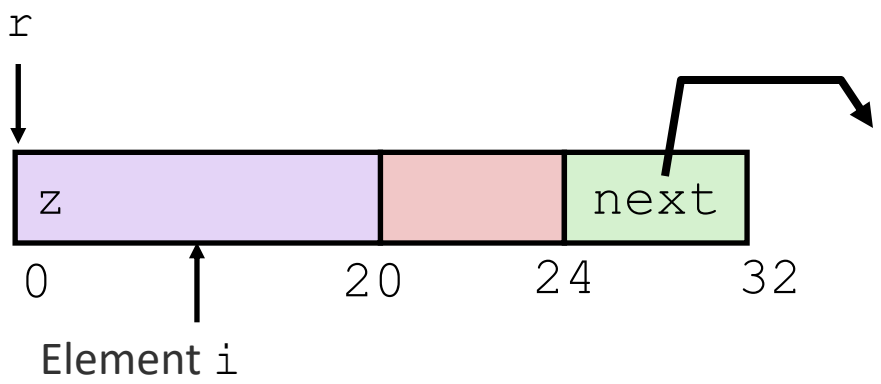

```
zip_node*get_tail_ptr(zip_node *p){
  if(p == NULL){
    return NULL;
  }

  while(p->next != NULL){
    p = p->next;
  }

  return p;
}
```

```
get_tail_ptr:
  testq   %rdi, %rdi
  jne      L1
  xorl    %eax, %eax
  retq
L1:
  movq    %rdi, %rax
  movq    24(%rax), %rdi
  testq   %rdi, %rdi
  jne     L1
  retq
```

Register	Value
%rdi	p

Linked List Exercise

```
typedef struct rec {
  zip_code z;
  struct rec *next;
} zip_node;
```




```
ll_exercise:
  movq    %rdi, %rax
  testq   %rax, %rax
  je      L1
  movq    24(%rax), %rdi
  testq   %rdi, %rdi
  je      L2
  pushq   %rax
  callq   ll_exercise
  addq    $8, %rsp
L2:
  retq
L1:
  xorl    %eax, %eax
  retq
```

```
zip_node *exercise(zip_node *p){




}
```