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:

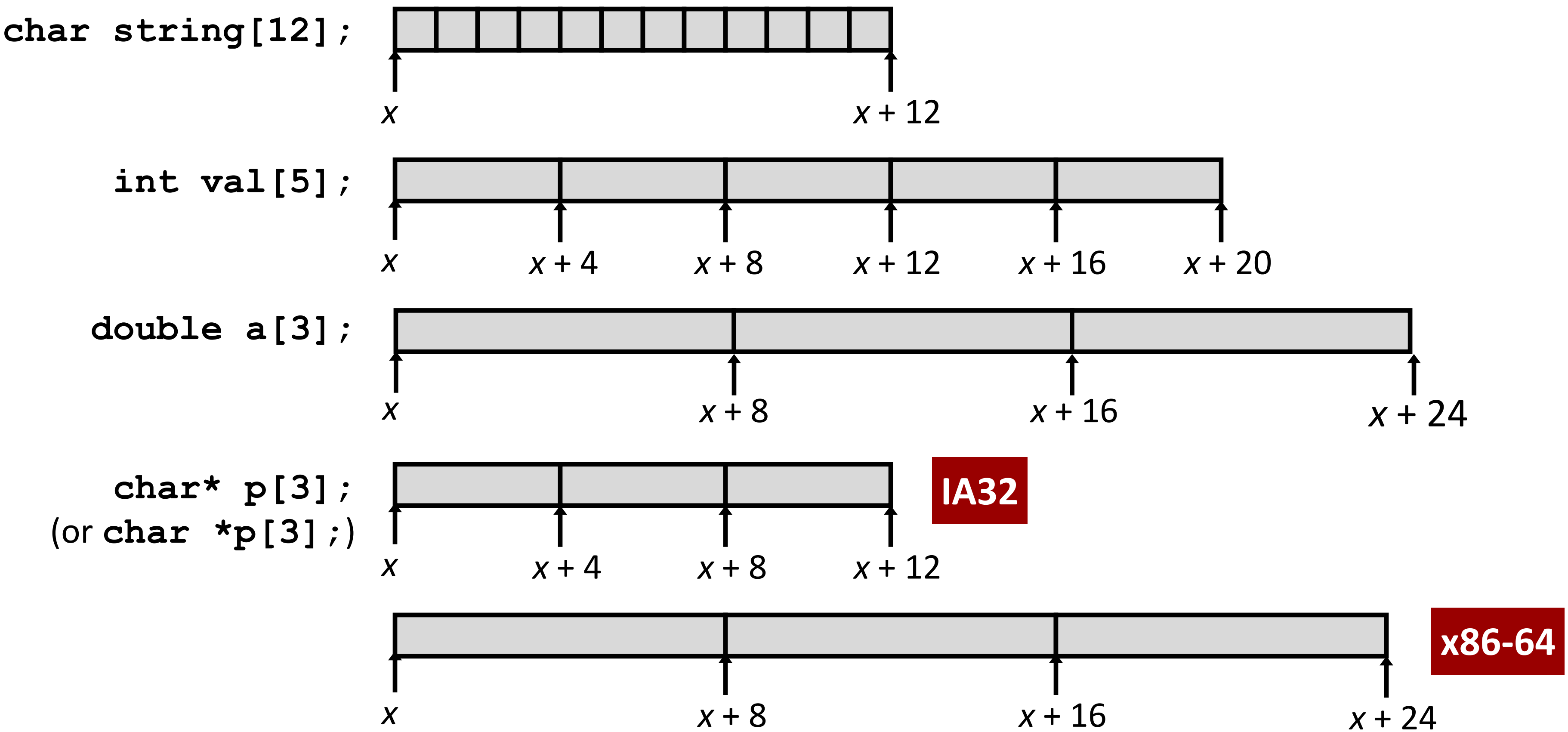
- Memory & data
- Integers & floats
- Machine code & C
- x86 assembly
- Procedures & stacks
- **Arrays & structs**
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

Section 5: Arrays & Other Data Structures

- Array allocation and access in memory
- Multi-dimensional or nested arrays
- Multi-level arrays
- Other structures in memory
- Data structures and alignment

Array Allocation

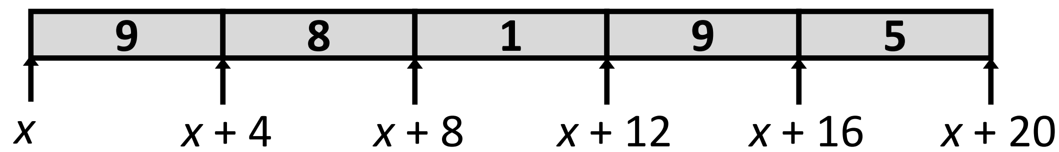
- **Basic Principle**
  - T A[N];
  - Array of data type T and length N
  - *Contiguously* allocated region of N * sizeof(T) bytes



Array Access

- **Basic Principle**
  - T A[N];
  - Array of data type T and length N
  - Identifier A can be used as a pointer to array element 0: Type T*

`int val[5];`



- **Reference** **Type** **Value**

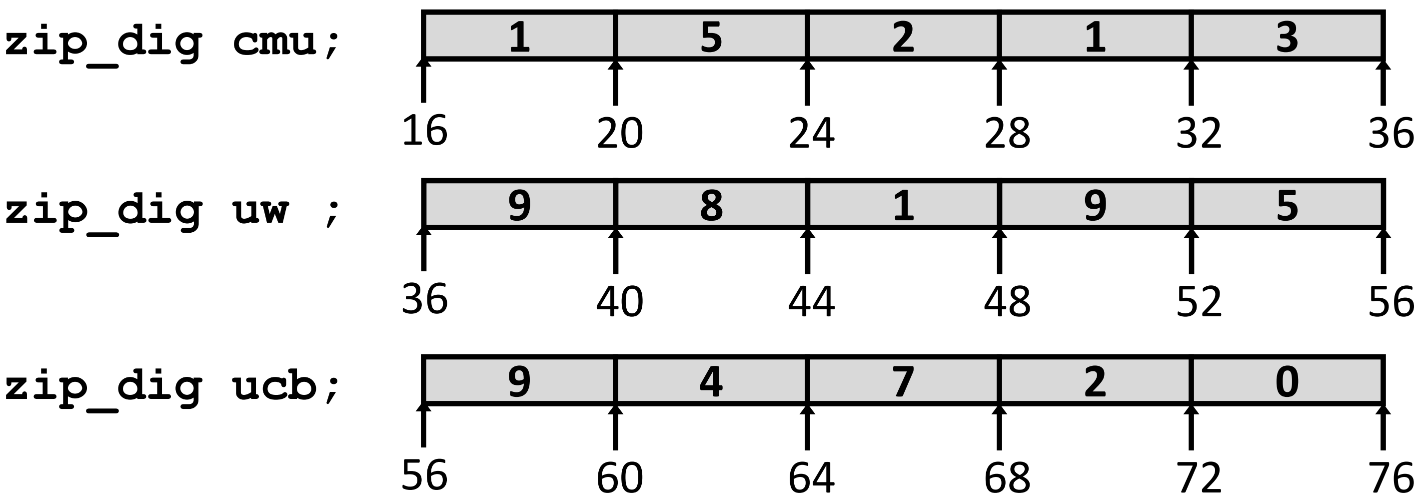
Reference	Type	Value
val[4]	int	5
val	int *	*x*
val+1	int *	*x* + 4
&val[2]	int *	*x* + 8
val[5]	int	?? (whatever is in memory at address *x* + 20)
*(val+1)	int	8
val + i	int *	*x* + 4*i

Array Example

```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw  = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

Array Example

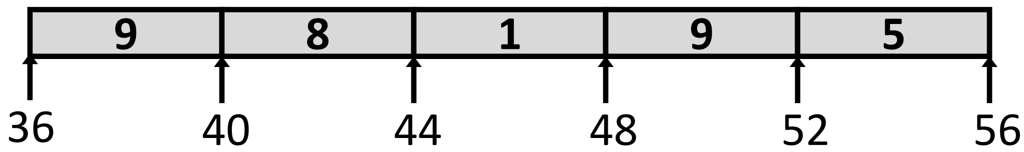
```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw  = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- **Declaration "`zip_dig uw`" equivalent to "`int uw[5]`"**
- **Example arrays were allocated in successive 20 byte blocks**
  - Not guaranteed to happen in general

Array Accessing Example

```
zip_dig uw;
```




```
int get_digit
  (zip_dig z, int dig)
{
  return z[dig];
}
```

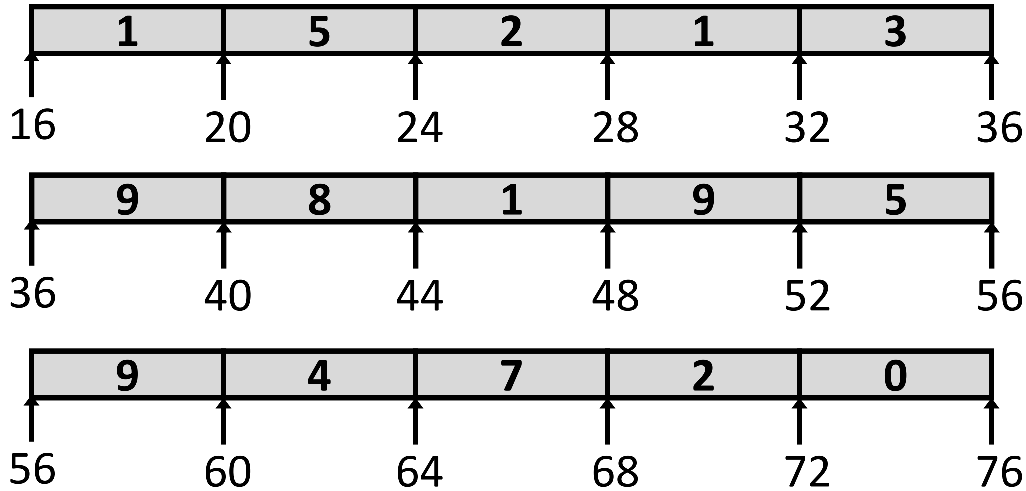
IA32

```
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax  # z[dig]
```

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

Referencing Examples

```
zip_dig cmu;
zip_dig  uw;
zip_dig ucb;
```



Reference	Address	Value	Guaranteed?
`uw[3]`	`36 + 4* 3 = 48`	`9`	Yes
`uw[6]`	`36 + 4* 6 = 60`	`4`	No
`uw[-1]`	`36 + 4*-1 = 32`	`3`	No
`cmu[15]`	`16 + 4*15 = 76`	`??`	No

- No bounds checking
- Location of each separate array in memory is not guaranteed

Array Loop Example

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

Array Loop Example

- **Original**

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

- **Transformed**
  - Eliminate loop variable **`i`**, use pointer **`zend`** instead
  - Convert array code to pointer code
    - Pointer arithmetic on **`z`**
  - Express in do-while form (no test at entrance)

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while (z <= zend);
  return zi;
}
```

Array Loop Implementation (IA32)

- **Registers**

  `%ecx  z`
  `%eax  zi`
  `%ebx  zend`

- **Computations**
  - `10*zi + *z` implemented as `*z + 2*(5*zi)`
  - `z++` increments by 4

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

```
  # %ecx = z
  xorl %eax,%eax              # zi = 0
  leal 16(%ecx),%ebx          # zend  = z+4
.L59:
  leal  (%eax,%eax,4),%edx    # zi + 4*zi = 5*zi
  movl  (%ecx),%eax           # *z
  addl  $4,%ecx               # z++
  leal  (%eax,%edx,2),%eax    # zi = *z + 2*(5*zi)
  cmpl  %ebx,%ecx             # z : zend
  jle  .L59                   # if <= goto loop
```