ECE264: Advanced C Programming

Summer 2019

Week 2: Addresses, Pointers, Pointer Arithmetic

Addresses

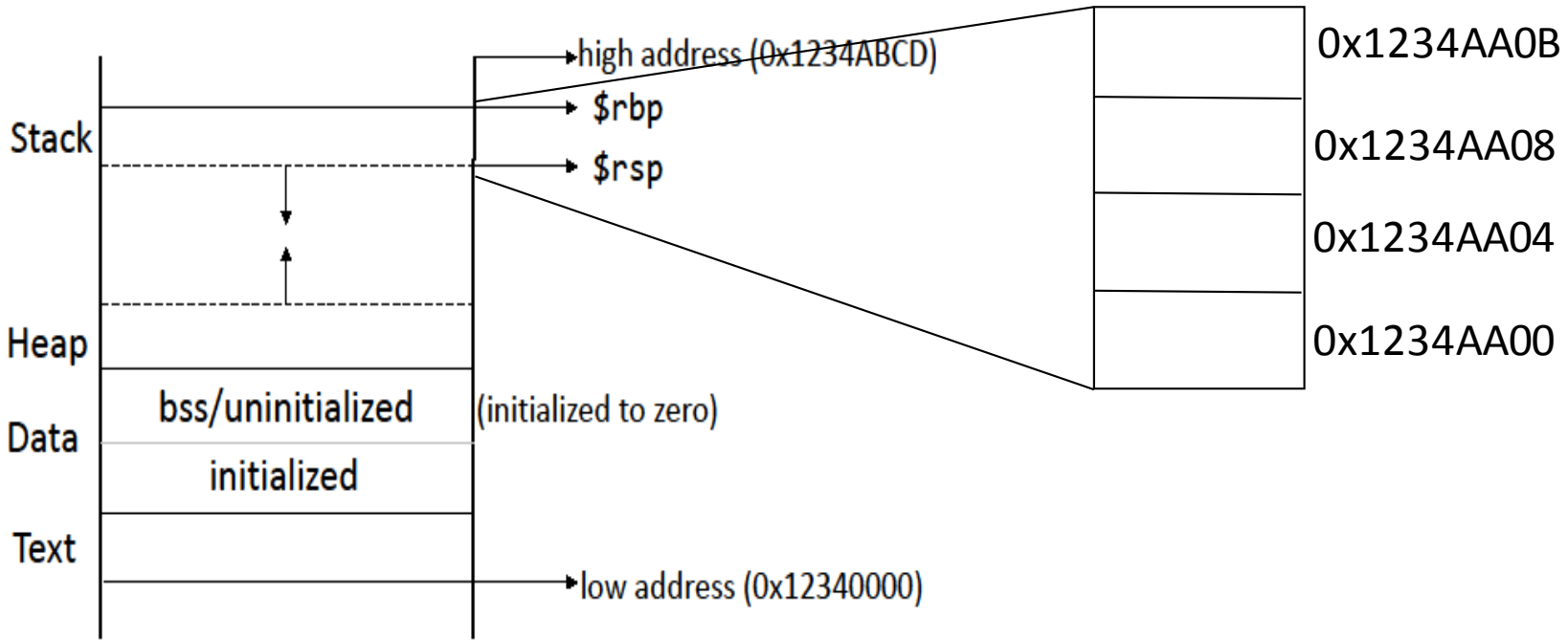
- Humans are not good at remembering numerical addresses.
  - What are the GPS coordinates (latitude and longitude) of your residence?

- Addresses in computer programs are just numbers.

- Addresses in computer programs identify memory locations.

- Computer programs think and live in terms of memory locations.

Program Memory Layout - Revisited



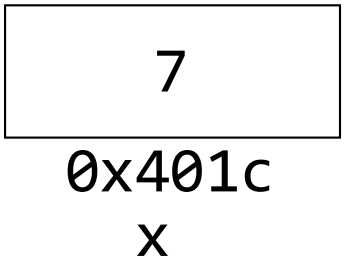
- Every memory location is a box holding data
- Each box has an address

Addresses Contd..

- A program navigates by visiting one address after another.
- We (humans) choose convenient ways to identify addresses so that we can give directions to a program
  - Variables

Handles to Addresses

- What is a variable?
  - Its just a handle to an address / program memory location
- `int x = 7;`



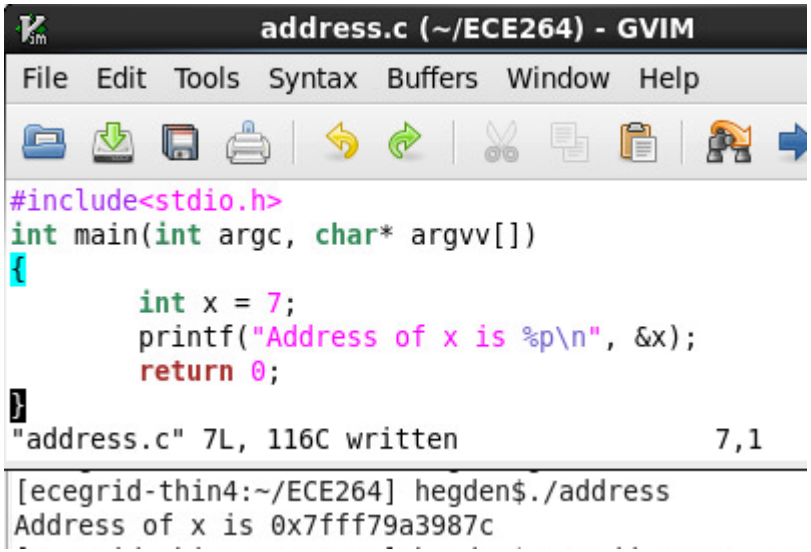
- Read `x` => Read the content at address `0x401C`
- Write `x`=> Write at address `0x401C`

Visualizing Addresses

- The *address of* (&) operator fetches a variable's address in C.
- &x would return the address `0x401C`.
- Format specifier `'p'`:

  `printf("%p\n",&x)`

  prints the Hexadecimal address of x



Pointers

- Pointer is a data type that *holds an address*.

  `<type>* <pointer_name>;`

  We read it as "pointer to `<type>`"

- Example:
  - `int* p;`

  is a variable named p whose type is pointer to `int`
  OR p is an integer pointer

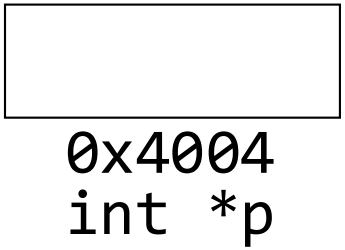
  Note that the variable declared is p, *not* *p

- A pointer always stores an address
- `<type>` of the pointer tells us what kind of data is stored at that address
- Example:
  - `int* p;`

    declares a pointer variable p holding an address, which identifies a memory location capable of storing an integer.

- `int* p;`

Remember p is a variable and all variables are just names identifying addresses.



Initializing Pointers

- `int* p=&x;`

  //p holds the address of a memory location that stores an integer.

0x401C	7
0x4004	0x401C
int *p	x

- We say `p` *points to* `x`

- Cannot assign arbitrary addresses to pointers.
- Example:

```
int* p=5;
```

- Operating system determines addresses available to each program.

The **NULL** address

- NULL is a special address
- Example

```
int* p=NULL; //p points to nowhere
```

- Useful when it is not yet known where p points to.
- Uninitialized pointers store garbage addresses

Using Pointers

- The *dereference* operator (*)
  - Lets us access the memory location at the address stored in the pointer

```
int x=7;
int *p = &x; //p now points to x
*p = 10; //this is the same as x=10
int  y=*p; //this is the same as y=x
```

The expression *p is equivalent to x

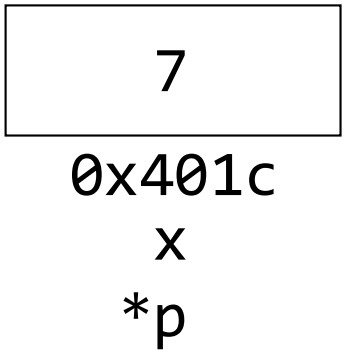
- Pointers as alternate names to memory locations

```
int x=7;
int *p = &x; //p now points to x
*p = 10; //this is the same as x=10
int  y=*p; //this is the same as y=x
```

The expression *p is equivalent to x

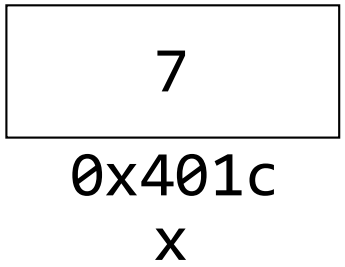
x is the name for an address

*p is the name for an address



- Pointers as “dynamic” names to memory locations

```
int x=7;
```




```
//x always names the location 0x401C
int *p = &x; //*p is now another name for x
int y = *p //like saying y=x
p = &y; //*p is now another name for y
*p=8; //like saying y=8
```

The **swap** function

```
int a = 8;
int b = 10;

void swap(int x, int y) {
   int tmp = x;
   x = y;
   y = tmp;
}

void main() {
   swap(a, b); //a is still 8, b is still 10
}
```

Pass by value

- C functions operate on ***copies*** of arguments.
- Change the data inside the function, you change the copy. Not the original.
- In swap, x and y are names of memory locations that are copies of a and b

What if x and y held addresses of a and b?

- *x and *y would name the same memory locations that a and b did.

The **swap** function

```
int a = 8;
int b = 10;

void swap(int* x, int* y) {
   int tmp =*x; //tmp = whatever is in the
location that x points to.
   *x = *y;
   *y = tmp;
}

void main() {
   //remember, we have to pass addresses now,
not ints.
   swap(&a, &b); //a is now 10, b is 8
}
```

Pointers to Different Types

- What can pointers point to? any data type!
  - Basic data types,
  - Structures,
  - Functions, and
  - even Pointers!

Pointer Chains

```
int x = 7;
int *p = x; //p points to x; *p is same
as x.

int * * q; //q is a pointer to pointer
to int
```

*q is same as p.
*(*q) is the same as *p, which is same as x

Pointers to Structures

```
typedef struct {
   int year;
   char model;
   float acceleration; //0-60mph in seconds
}Car;

Car t1 = {.year = 2017, .model = ‘S’,
.acceleration = 2.8 };

Car * pt1 = &t1; //now you can use *pt1
anywhere you use t1
```

```
(*pt1).acceleration = 2.3;
(*pt1).year = 2019;
(*pt1).model = ‘X’;
float avg_acceleration = ((*pt1).acceleration
+ (*pt2).acceleration) / 2.0;
```

We can also use the -> operator to access structure members.

```
pt1->acceleration = 2.3;
pt1->year = 2019;
pt1->model = ‘X’
float avg_acceleration = (pt1->acceleration +
pt2->acceleration) / 2.0;
```

Address of (&) operator and Type

- Adding & to a variable adds * to its type
- Example:
  - if a is an int, then &a is an int*
  - if b is an int*, then &b is an int**
  - if c is an int**, then &c is an int***
  - …

Dereference (*) operator and Type

- Adding * to a variable subtracts * from its type
- Example:
  - if a is an int*, then *a is an int
  - if b is an int**, then *b is an int*
  - if c is an int***, then *c is an int**
  - …

Pointers to Functions (Function Pointers)

- Every function in a C program refers to a specific address (remember disassembling code during buffer overflow attack)
- Function pointers store addresses of functions
- Syntax:

```
typedef type (*name) (argument types)
```

Function Pointers - Example

```
typedef void (*myfuncptr) (int, int)
```

- `myfuncptr` is a pointer to a function that returns a `void` and accepts two arguments of type `int`.

Function Pointers - Example

```
void swap(int x, int y) {
   int tmp = x;
   x = y;
   y = tmp;
}

myfuncptr ptrswap = swap; //initialization.

int main(int argc, char* argv[]) {
   int a=10;
   int b=20;
   ptrswap(a,b); //swap called by a function
pointer
}
```

Function Pointers

How about these?

```
(*ptrswap)(a,b);

(****ptrswap)(a, b)
```

C says dereferencing a function pointer returns a function pointer. Behavior different from normal '&' and '' operators.*

Pointer Arithmetic

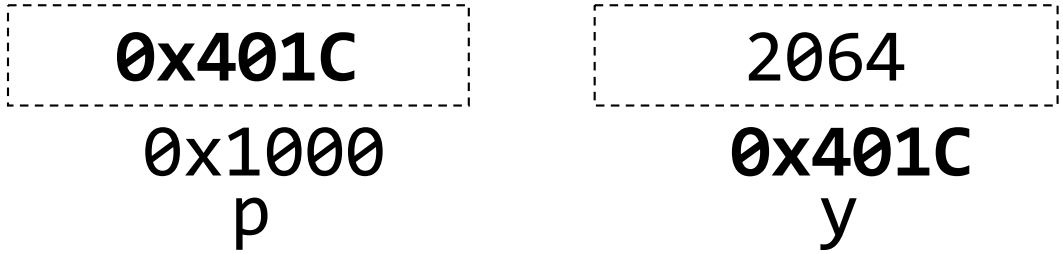
```
int y = 1040;
int* p= &y;
```

- What does *(p+1) mean?
  - Data at "one element past" p
- What does "one element past" mean?
  - p is a pointer, so holds the address of a memory location
  - p is an `int` pointer, so that memory location holds an integer
  - p+1 is interpreted as address of the next integer

Pointer Arithmetic

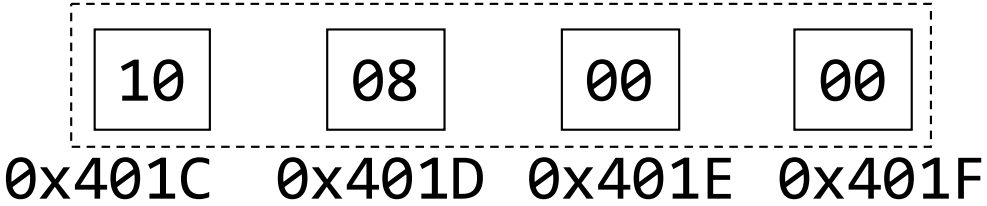
- Our representation of

```
int y=2064;
int* p = &y;
```



Pointer Arithmetic

- `ints` occupy 4 bytes. `0x401C` is the address of the first byte*:



*2064 = 0x810 (=0x00,00,08,10 when written using 8 digits and x86 is little-endian)

- (*p) = data at 0x401C
  - *returns the correct value of 2064 and not 0x10. Why?*

Pointer Arithmetic

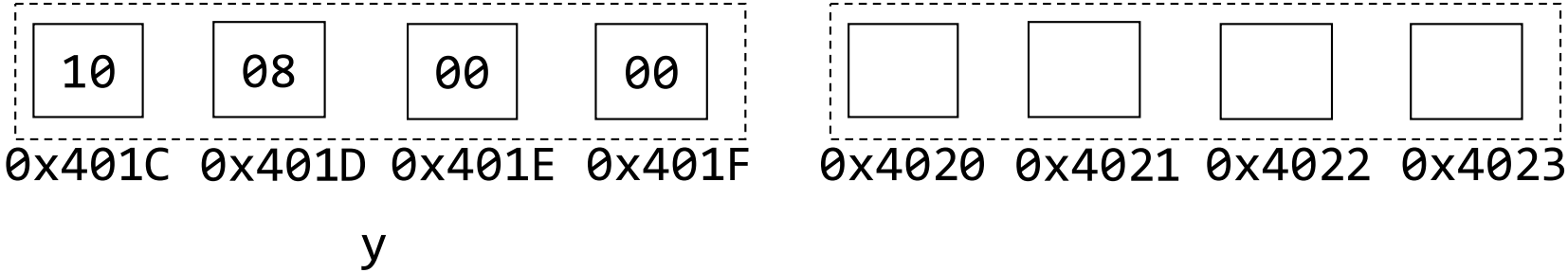
- (p+1) gets the “address of the next integer”

0x401C	2064
0x1000 p	**0x401C** y

What is the address of the next integer?

Pointer Arithmetic

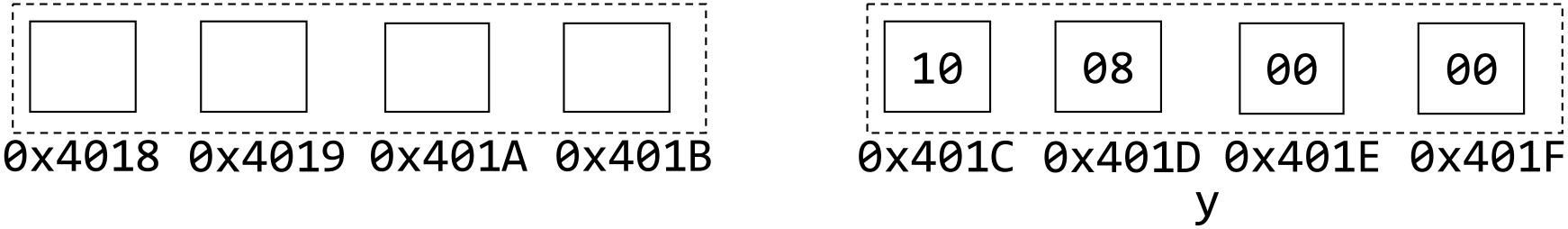
- What is the address of the next integer?
  - Add 4 to current value of p (0x401C) = `0x4020`



Pointer Arithmetic

- (p-1) computes the address before y

```
int y=2064;
int* p = &y;
```



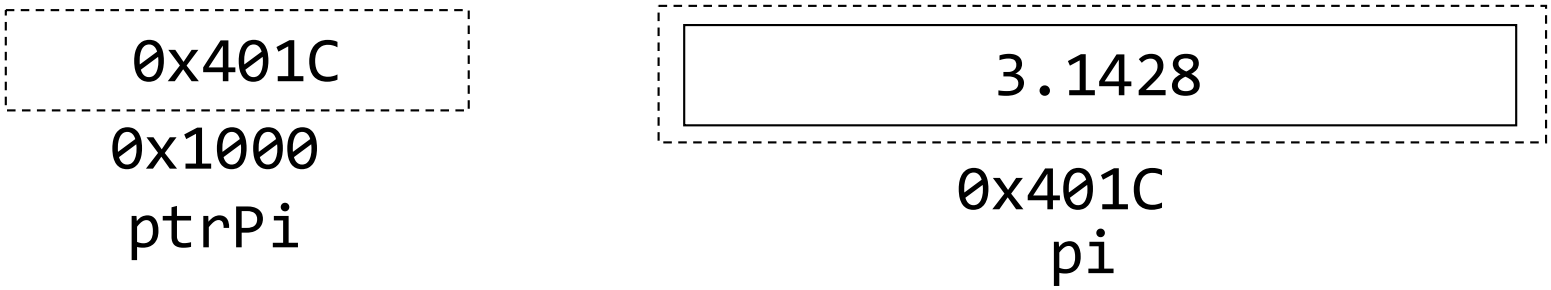
subtract 4 from the current value of p `(0x401C)` = `0x4018`

- Similarly we can add/subtract any number to/from a pointer variable.
- Compare to a specific address (E.g. `if(p == NULL)`)

Pointer Arithmetic

- Pointer to `double` (`double` occupies 8 bytes)

```
double pi=3.1428;
double* ptrPi = &pi;
```



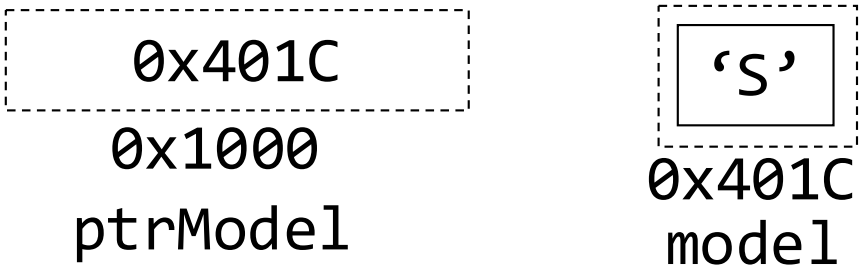
What is the address computed for `(ptrPi+1)`*?* `0x4024`

What is the address computed for `(ptrPi-1)`*?* `0x4014`

Pointer Arithmetic

- Pointer to `char`

```
char model='S';
char* ptrModel = &model;
```

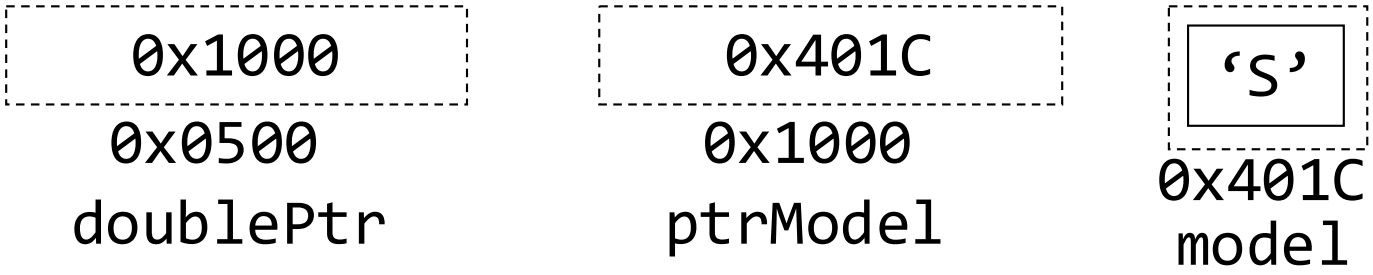


What is the address computed when we do `(ptrModel+1)`*?*

Pointer Arithmetic

- Pointer to pointer

```
char model='S';
char* ptrModel = &model;
char** doublePtr = &ptrModel;
```



Bonus: what is the address computed when we do `(doublePtr+1)`*?* (assuming we are using 32-bit machines)

Pointer Arithmetic

- Pointer to `struct`

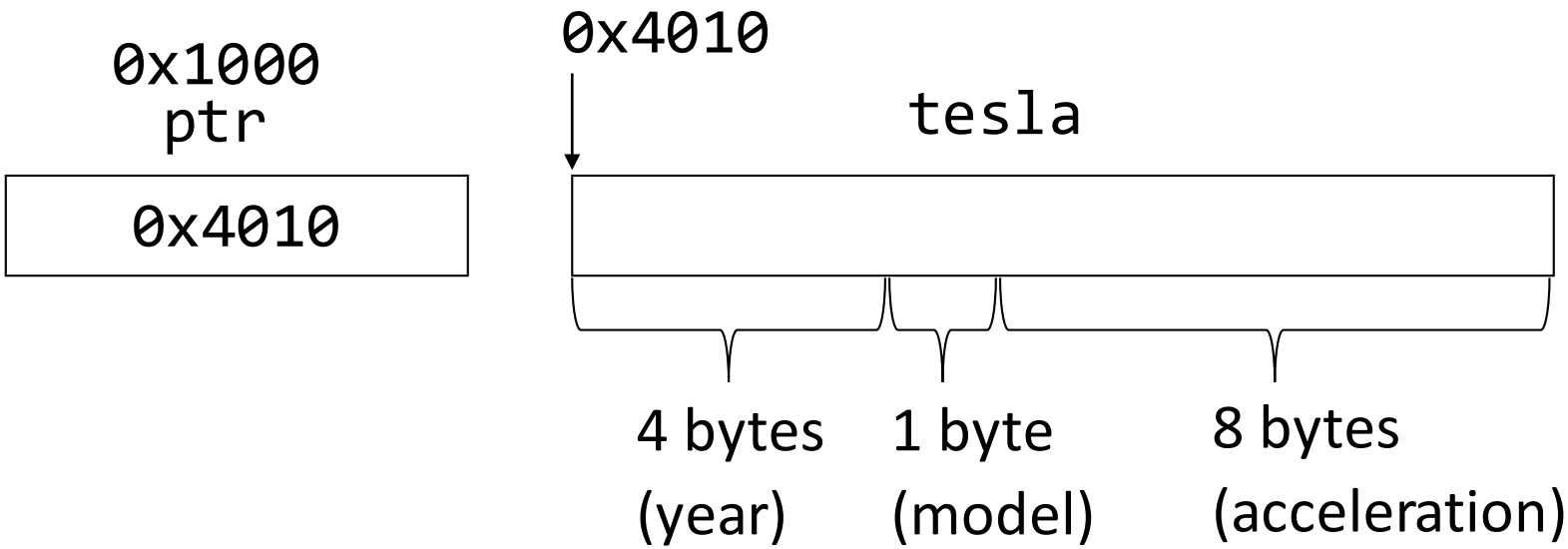
```
typedef struct {
   int year;
   char model;
   double acceleration; //0-60mph in seconds
}Car;

Car tesla = {.year = 2017, .model = ‘S’,
.acceleration = 2.8 };

Car* ptr = &tesla;
```

Pointer Arithmetic

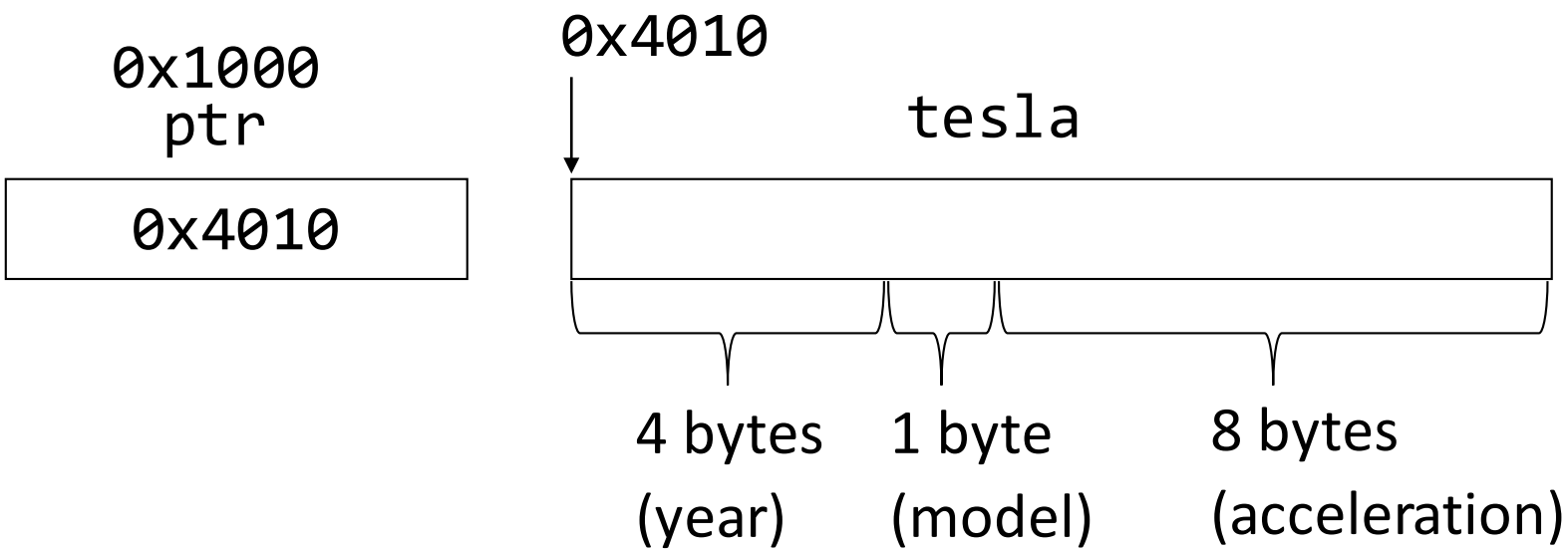
- Pointer to `struct`



- With `#pragma pack(1)`

Pointer Arithmetic

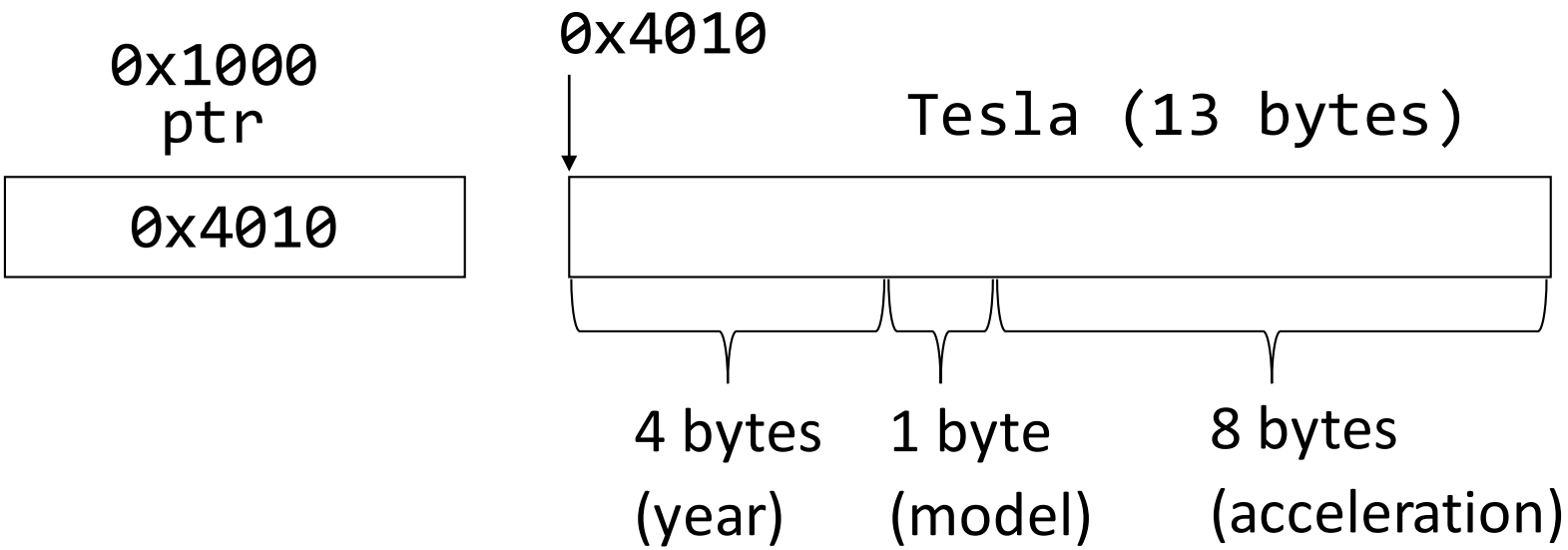
- What address does `(ptr+1)` evaluate to?
  - Add 13 (4+1+8) to the value at `ptr`



- `ptr+1 = 0x401D`

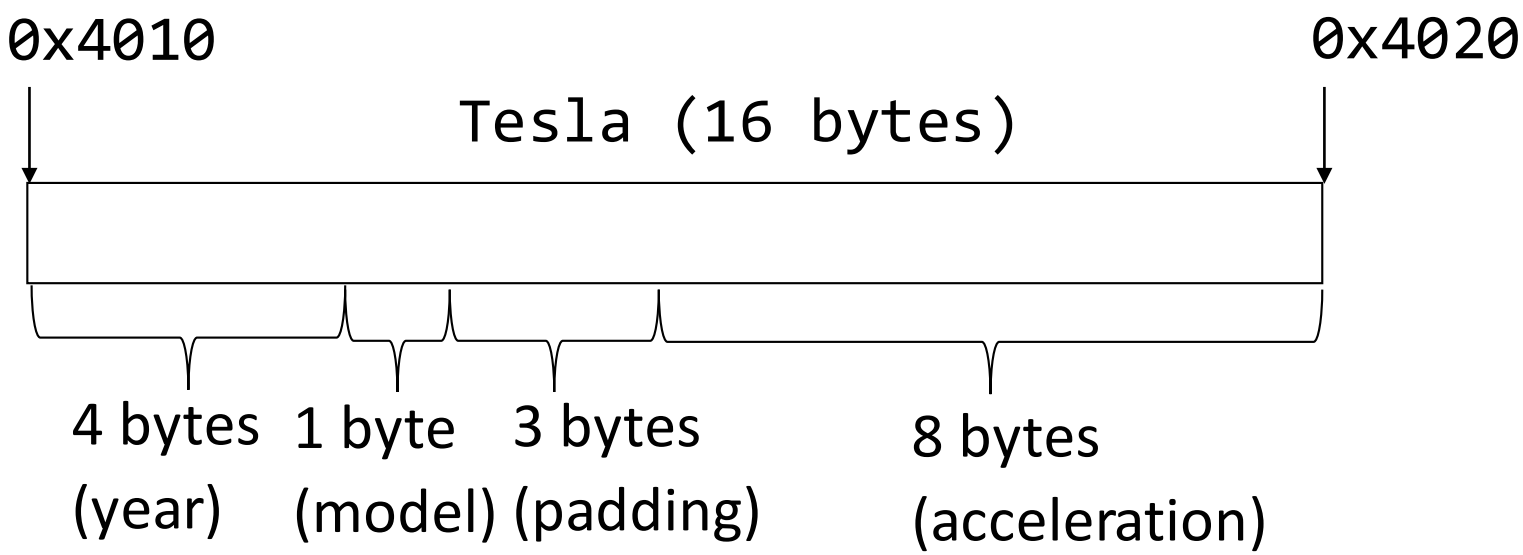
Detour - #pragma pack

- Preprocessor directive (starts with '#')
  - Preprocessor specifies instructions for the compiler on how to *pack* structure members in memory.
  - Varies from compiler to compiler



#pragma pack

- Normally (without #pragma pack) structure members are padded to create an alignment of the structure size with memory addresses.



Arrays

- Another data type!
  - Array of `ints, structs` etc.
  - Array of `chars` (strings in C)
- Work a little bit like pointers

```
int a[10]={1,2,3,4,5,6,7,8,9,10};
//array of 10 integers
```

1	2	3	4	5	6	7	8	9	10
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

10 elements guaranteed to be next to each other in memory

Arrays

```
int a[10]={1,2,3,4,5,6,7,8,9,10};
```

1	2	3	4	5	6	7	8	9	10
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

a

0x4001

- 0x4001 is starting address of the array = address of `a[0]` = **`&a[0]`**
- Fetch the address of a = `&a` = `0x4001`

Arrays

- Array name in C is the address of the first element of the array

```
int a[10]={1,2,3,4,5,6,7,8,9,10};
```

Therefore, **`a == &a[0]`**

`a, &a, &a[0]` are the same and have values `0x4001`.

Arrays

- Array name in C is the address of the first element of the array

Array names are converted to pointers (in most cases) but a's type is not a pointer.

```
int* ptr=a; //ptr holds the address of the
first element of the array (also &a[0]).

ptr[1] gets a[1]
ptr[2] gets a[2]
...
```

How is this possible?

Arrays

- Array dereferencing operator [ ] is implemented in terms of pointers.
  - `a[3]` means: start at the address a, go forward 3 elements, fetch the *data at* that address.
  - In pointer arithmetic syntax, this is equivalent to:

    *(a+3)

  So,

  a[0] really means: *(a+0)
  a[1] really means: *(a+1)

Arrays

- So, when

```
int* ptr = a;
```

  - `ptr[0]` really means `*(ptr+0)`, which is the same as `*(a+0)`, which is `a[0]`
  - `ptr[1]` really means `*(ptr+1)`, which is the same as `*(a+1)`, which is `a[1]`

```
...
```

Exercise

```
char s[3] = “Hi”;
char *t = “Si”;
int u[3] = {5, 6, 7};
int n=8;
```

Expression	Type	Comments
s	char[3]	array of 3 chars
t	char*	address of a char
u	int[3]	array of 3 ints
&u[0]	int*	address of an int

Exercise

```
char s[3] = “Hi”;
char *t = “Si”;
int u[3] = {5, 6, 7};
int n=8;
```

Expression	Type	Comments
*&n	int	value at n
*t	char	data at address Held by t

Exercise

- Array initializers:

```
1. int u[3] = {5, 6};
```

Is this valid?

If yes, what is the value held in the third element u[2]?

```
2. int u[3] = {5, 6, 7, 8};
```

Is this valid?

```
3. char s1[]="Hi";
```

What is the size of s1? (how many bytes are reserved for s1)

```
4. char s2[3]="Si";
```

Is this valid?