ECE264: Advanced C Programming

Summer 2019

Week 2: Addresses, Pointers, Pointer Arithmetic

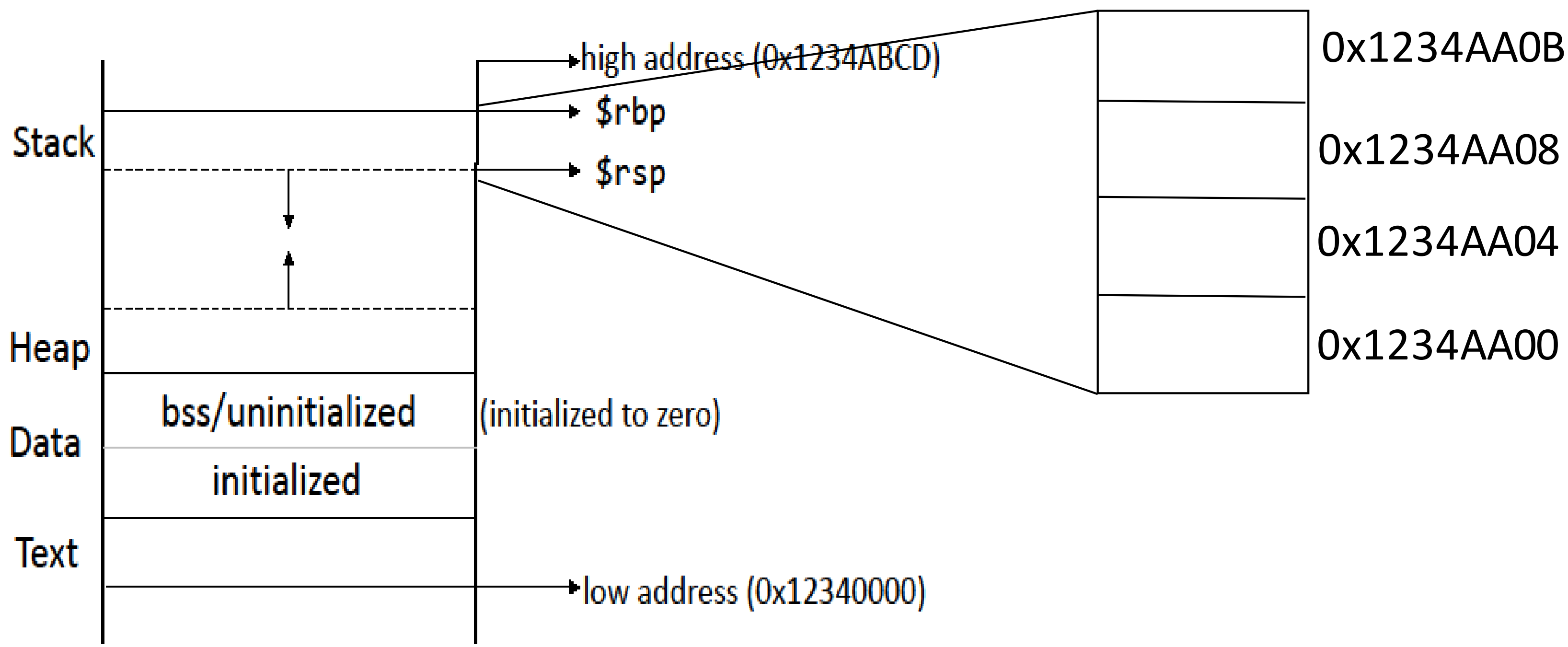
Addresses

- 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



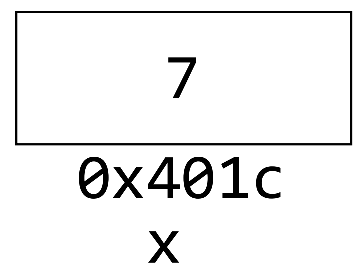
- Every memory location is a box holding data
- Each box has an address

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;`



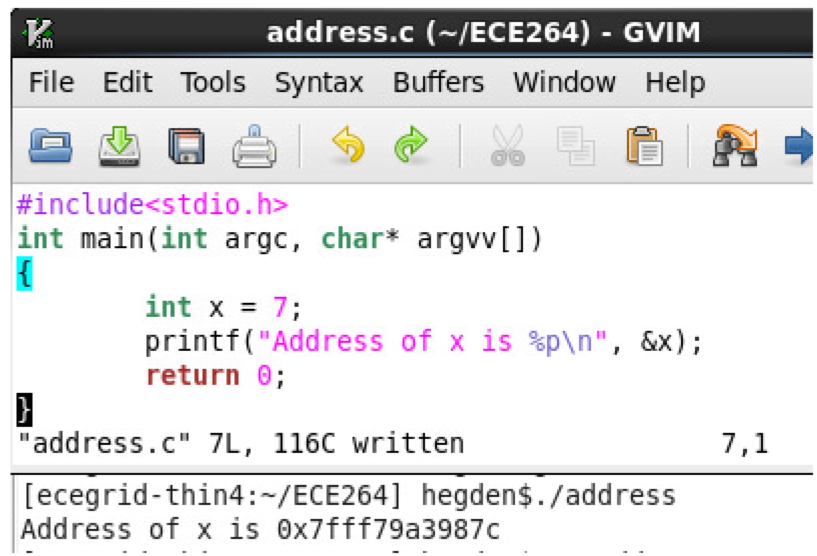
- Read `x` => Read the content at address `0x401C`
- Write `x`=> Write at address `0x401C`

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.

```
0x4004
int *p
```

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

7

0x401c

x

*p

- Pointers as “dynamic” names to memory locations

```
int x=7;        7
              0x401c
                 x
//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

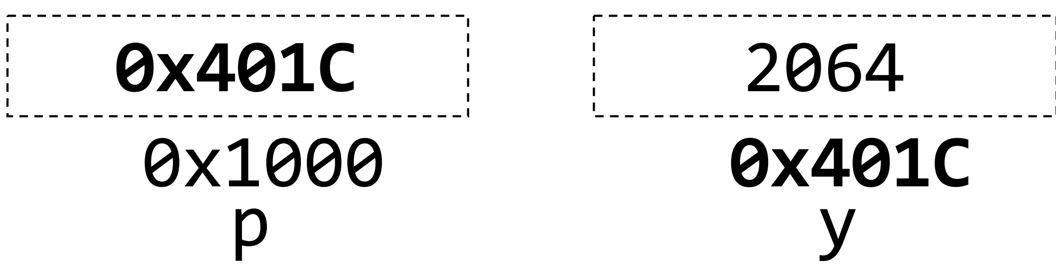
```
int y = 1040;
int* p= &y;
```

- 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*:

`10`	`08`	`00`	`00`
`0x401C`	`0x401D`	`0x401E`	`0x401F`

*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

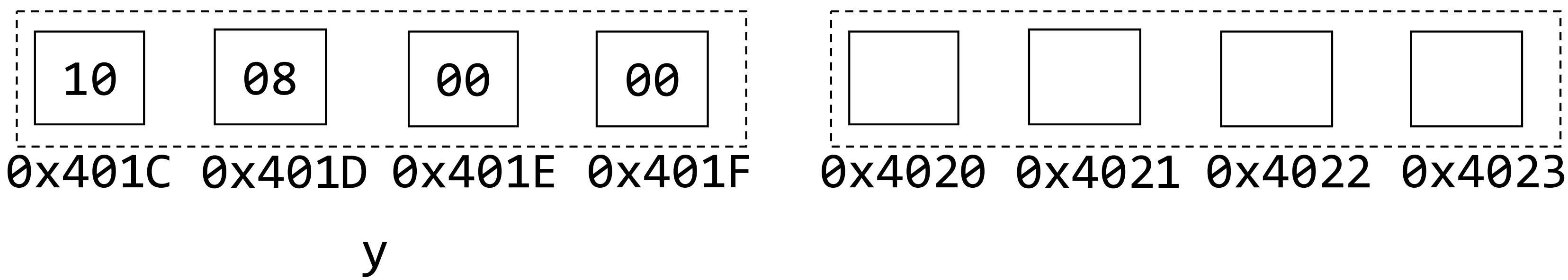
- (p+1) gets the “address of the next integer”

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

What is the address of the next integer?

Pointer Arithmetic

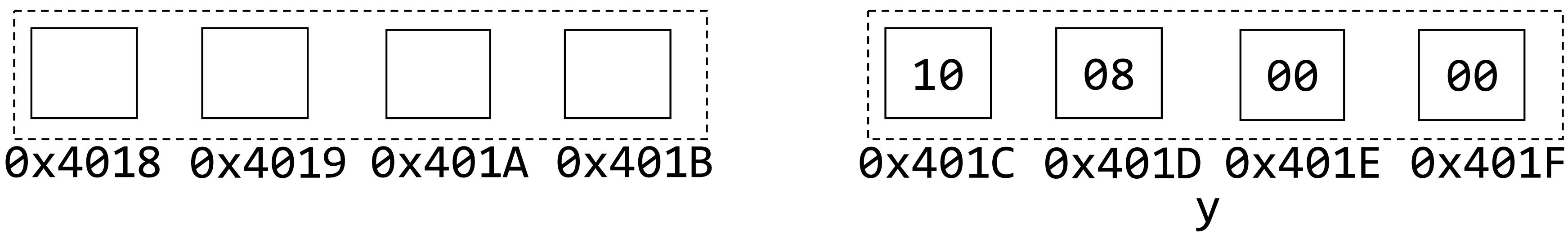
- What is the address of the next integer?
  - Add 4 to current value of p (0x401C) = `0x4020`



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;
```

`0x401C`	`3.1428`
`0x1000`	`0x401C`
`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;
```

0x401C

0x1000

ptrModel

'S'

0x401C

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;
```

doublePtr	ptrModel	model
0x1000	0x401C	'S'
0x0500	0x1000	0x401C

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

Pointer Arithmetic

- Pointer to `struct`

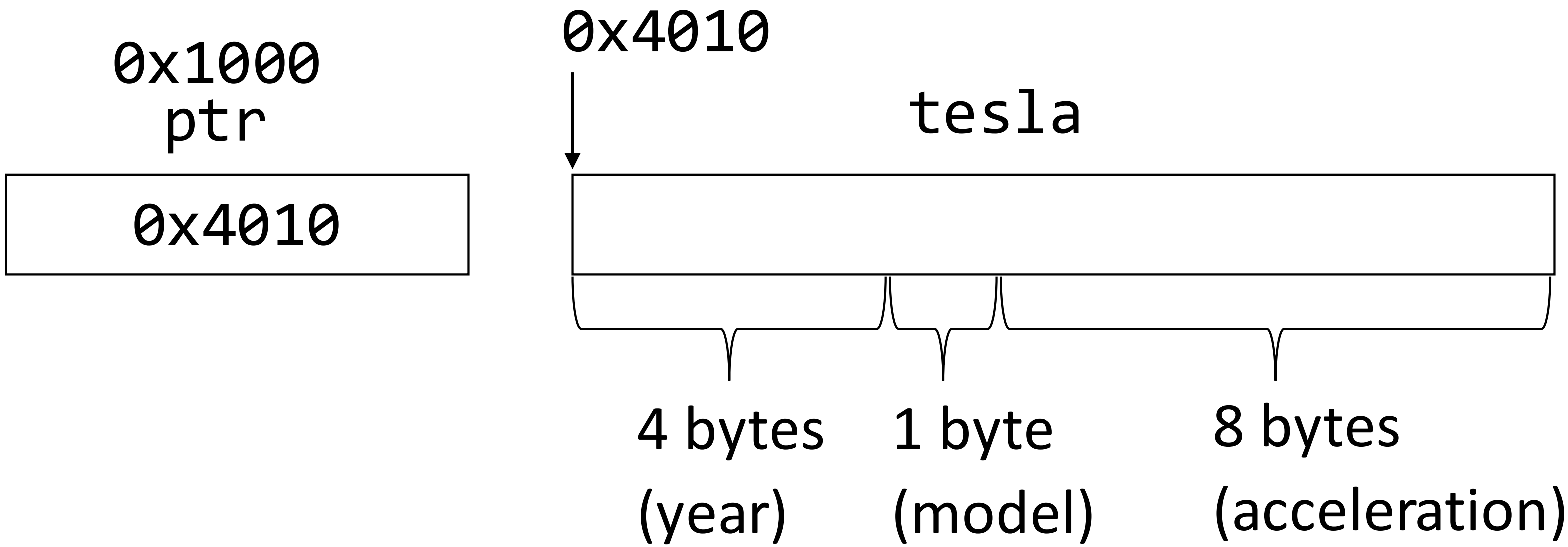
```
typedef struct {
   int year;
   char model;
   double acceleration; //0-60mph in seconds
}Car;

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

Car* ptr = &tesla;
```

Pointer Arithmetic

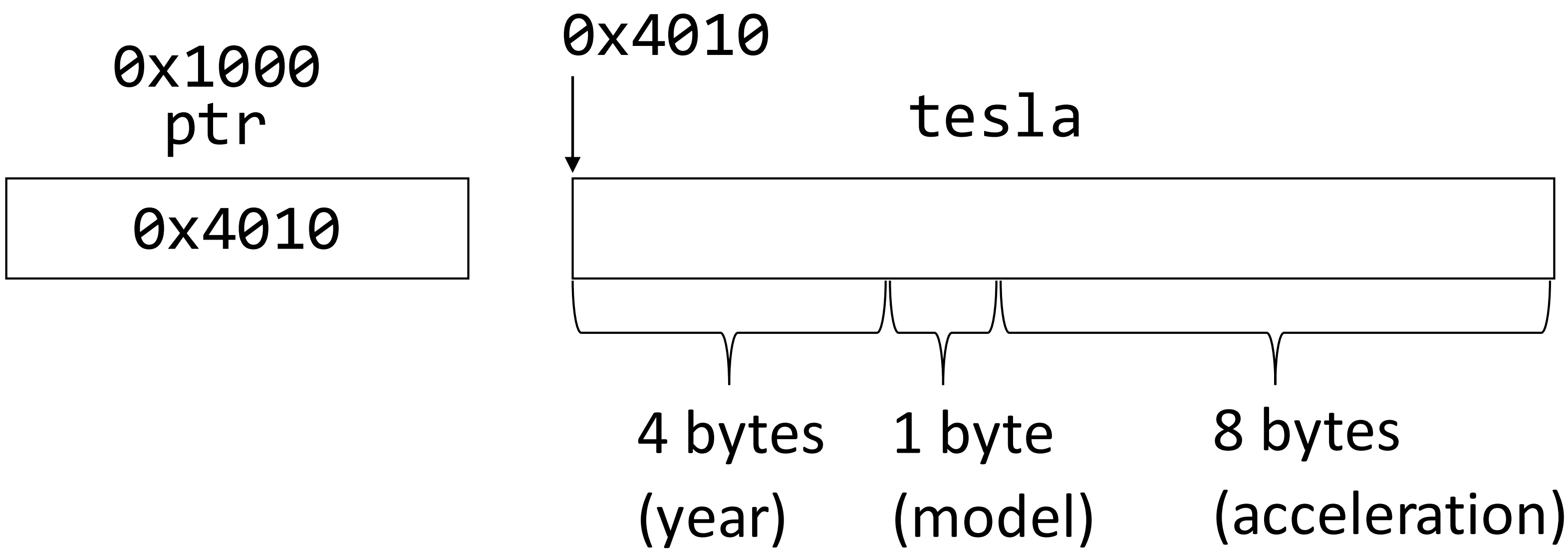
- Pointer to `struct`



- With `#pragma pack(1)`

Pointer Arithmetic

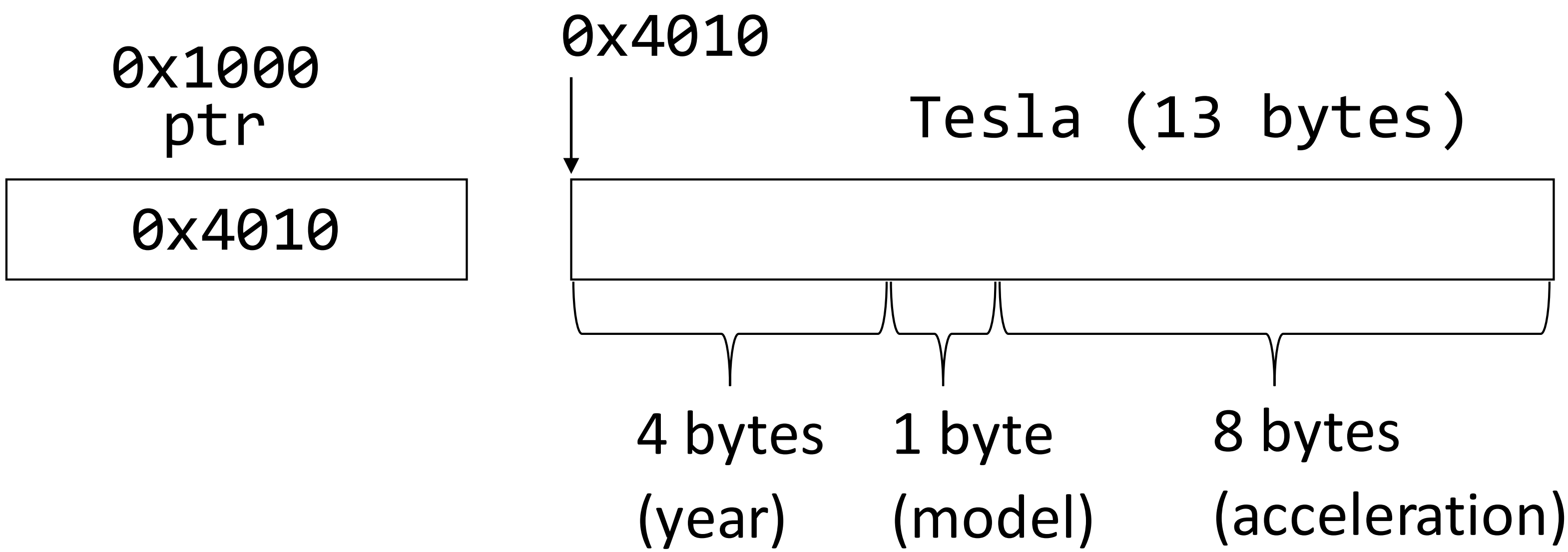
- What address does `(ptr+1)` evaluate to?
  - Add 13 (4+1+8) to the value at `ptr`



- `ptr+1 = 0x401D`

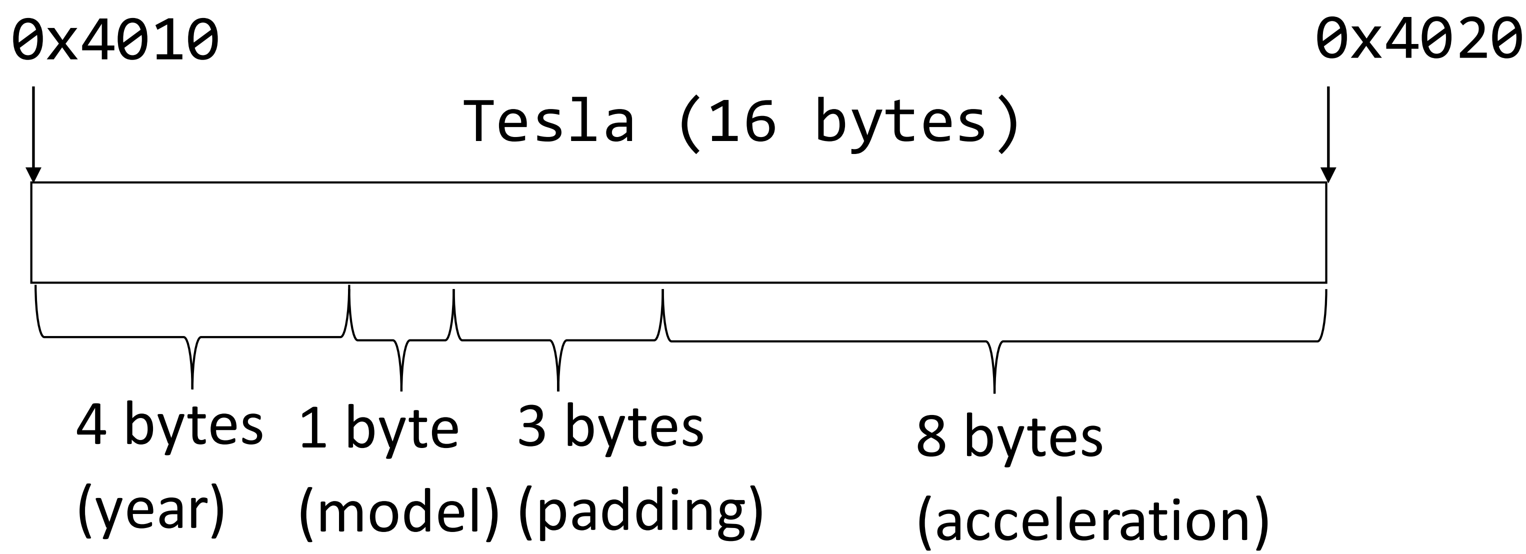
Detour - #pragma pack

- 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

```c
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?

Exercise

```
int u[3] = {5, 6, 7};
int* p=u;
p[0]=7;
p[1]=6;
p[2]=5;
```

//At this line, u *would contain the numbers in reverse order.*
u[0] = 7, u[1]=6, u[2]=5.

```
char *str = “Hello”;
char* p=str;
p[0]=‘Y’;
```

//At this line, what would `str` contain?

Array of Strings

- How do we creating them?
  - Declare types

1. `char* strArray1[3]; //declares an array of 3 pointers to char.`
2. `char strArray2[3][10]; //declares a two dimensional array. This can hold 3 strings, each of a maximum length of 10 bytes.`

Array of Strings

- Initializing (method 1)

```
char* strArray1[3]; //declares an array of 3
pointers to char.
strArray1[0]="RED";
strArray1[1]="BLUE";
strArray1[2]="GREEN";
```

OR

```
char* strArray1[3]={"RED", "BLUE", "GREEN"};
```

OR

```
char* strArray1[]={"RED", "BLUE", "GREEN"};
```

Array of Strings

- Modifying (method 1)

```
char* strArray1[3]; //declares an array of 3 pointers to char.
strArray1[0]="RED";
strArray1[1]="BLUE";
strArray1[2]="GREEN";
strArray1[1]="CLUE"; //modifies strArray1 by changing the 2nd string
```

NOT ALLOWED TO MODIFY `strArray1 as in:`

```
char* cptr= strArray1[1];
cptr[1]='C'; //to change "BLUE" to "CLUE"
```

OR `strcpy(strArray[1],"CLUE");`

Array of Strings

- Initializing (method 2)

```
char strArray2[3][10]; //declares a two
dimensional array.
strcpy(strArray2[0],“RED”);
strcpy(strArray2[1],“BLUE”);
strcpy(strArray2[2],“GREEN”);
```

OR

```
char strArray2[3][10]={“RED”, ”BLUE”, “GREEN”};
```

OR

```
char strArray2[][10]={“RED”, ”BLUE”, “GREEN”};
```

BUT NOT

```
char strArray2[][]={“RED”, ”BLUE”, “GREEN”};
```

- *Second and subsequent dimensions must be given.*

Array of Strings

- Modifying (method 2)

```
char strArray2[3][10]; //declares a two dimensional array.
strcpy(strArray2[0],“RED”);
strcpy(strArray2[1],“BLUE”);
strcpy(strArray2[2],“GREEN”);
strcpy(strArray2[1],“CLUE”);
```

OR

```
strArray2[1][0]=‘C’;
```

BUT NOT

```
strArray2[1]=“CLUE”;
```

Array name `strArray2` *does not convert (decay) into a pointer (exception 1)*

Array of Strings - Exercise

1. `char* strArray1[3];`

What is the type of `strArray1`? char* [3]

2. `char strArray2[3][10];`

What is the type of `strArray2`? char [3][10]

3. Give an example of string array that you saw in `PA01main.c`?

Command Line Arguments

```
bash-4.1$./pa01 input1
```

//this is how we ran pa01 (the Makefile did it for us)

- The `main` function is defined as:

```
int main(int argc, char* argv[])
{
	//some code here.
}
```

Command Line Arguments

```
bash-4.1$./pa01 input1
```

```
int main(int argc, char* argv[])
{
      //some code here.
}
```

Identifier	Comments	Value
`argc`	Number of command-line arguments (including the executable)	2
`argv`	each command-line argument stored as a string	`argv[0]=“./pa01”` `argv[1]=“input1”`

Command Line Arguments - Exercise

`char* argv[]`

1. is method1 of declaring string arrays.
2. In method1, we can only assign string literals (constants) to array elements. (“`./pa01`” and “`input1`” are string literals here)
3. string literals reside on read-only data segment.
4. In an earlier lecture we learnt that command-line arguments passed to `main` reside on stack segment.

is there a contradiction?

Array of Strings - Comparison

- Method 2 (strArray2)
  - Wastes space (*how?*)
  - Modification is easy
- Method 1 (strArray1)
  - Does not waste space
  - Modification is not possible
- How to get the best of both worlds?
  - *Dynamic memory allocation*

`sizeof` operator

- Returns the size of a type or variable in bytes.
- The return value is of type size_t.
  - unsigned integer of at least 16 bits.
- Unary operator
  - Takes a single operand
- Computes results at compile time

`sizeof` operator

- Example:

```
1.printf("sizeof(int)=%zu\n",sizeof(int));
2.printf("sizeof(double)=%zu\n",sizeof(double));
3.printf("sizeof(char*)=%zu\n",sizeof(char*));
4.printf("sizeof(int[10])=%zu\n",sizeof(int[10]));

int x=2064;
double y=3.142832;
char cArr[10];
5.printf("sizeof(x)=%zu\n",sizeof(x));
6.printf("sizeof(y)=%zu\n",sizeof(y));
7.printf("sizeof(cArr[10])=%zu\n",sizeof(cArr));
```

- *What is %z?*
  - Introduced in C99 for portability of code

`sizeof` operator

- Example:

```
char cArr[10]="Hi";

char* cPtr = cArr; //array name converted to pointer

printf("sizeof(cPtr)=%zu\n",sizeof(cPtr));

printf("sizeof(cArr)=%zu\n",sizeof(cArr)); //array
name NOT converted to pointer
```

The array name `cArr` does not convert (decay) into a pointer when used as an operand of `sizeof` operator (exception 2).

`sizeof` operator - uses

- Computing array length:

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

int numElements = sizeof(iArr) / sizeof(iArr[0]);
```

- In dynamic memory allocation

What does `sizeof(1000000)` return?

Dynamic Memory Allocation

- Statically allocated arrays:

```
int arr[3]={1, 2, 3};
```

Must be known at compile time

- Can't expand `arr` once defined
- Memory for `arr` is invalid when the function returns

Dynamic Memory Allocation

- What if we don’t know the array length?
  - Option 1: Variable length arrays.

  Not an option with `-Wvla, -Wall,` and `-Werror` flags

  - Option 2: use heap.

  Preferred option

Dynamic Memory Allocation

- We interact with heap using
  - `malloc`

    "Give us X bytes of storage space (memory) from the heap so that we can use it to store data"
  - `free`

    "take back this memory so that it can be used for something else"

malloc

```
void * malloc(size_t X)
```

//Gives us access to X bytes of memory from the heap. Returns the address of the first byte of the memory location"

- What is `void*`
  - A generic pointer that can hold the address of a variable of any type
  - cannot dereference (*) or do pointer arithmetic.
  - Must convert to appropriate type before use.

Detour - type casting

- Way to convert from one type to another.
  - We saw an example of implicit conversion: array names to pointers (`int*  p=arr;`)

  - type enclosed in brackets is a typecast operator:

```
(type) expression
E.g. (int) (2.3+1.5)
```

  - Use case: e.g. force floating point division.

```
int numMiles=41;
int numGallons = 2;
double mpg = (double) numMiles/numGallons;
```

malloc

```
int N=10;
int * arr=malloc(N * sizeof(int))
```

- Find 40 bytes of heap and reserve it for program's use.
- Return the address of the beginning of the chunk.
- `arr` is guaranteed to be 40 bytes of contiguous memory.
- We can now treat `arr` just like an array:

`arr[0]` accesses the first integer element

`arr[1]` accesses the second integer element

….

`malloc`

Suggestions:

1. `malloc` returns `void *`. So, to convert the return address to `int *` in the above example, you need **not** typecast the return value to an `int *`

   `int *arr = ~~(int *)~~malloc(N * sizeof(int))`

2. Use `sizeof(expression)` instead of `sizeof(int)`

   ```
   int *arr = malloc(N * sizeof(*arr))
   ```

   Later when you change `int` to `long long`, you just need to change at one place.

3. Always check if the return value is NULL:

   ```
   if(arr == NULL) {}
   ```

`free`

- When we no longer need the heap memory chunk reserved for us:

  `free(arr);`

- `free(void *ptr)` `//take back the chunk of memory, where ptr points to the beginning of that chunk`

- Next time you call `malloc` you may get the same address as earlier or an entirely new address

`free` – Don’ts

- Forget to call `free`
- Use the memory after calling `free`
- Call `free` twice (or multiple times)
- Call `free` on a different address

- `IMPORTANT:`
  `malloc`'ed memory remains with the program until we `free` it;

What happens if we don't call `free`?

Memory Leaks

- What happens when you call `malloc` inside a function `foo`

```
void foo(int N) {
    //allocate an array of N integers
    int * p = malloc(N * sizeof(int));

    //code to do something with the array

    return;
}
```

Memory Leaks

- When `foo` returns, local variable `p` goes away.

```
void foo(int N) {
    //allocate an array of N integers
    int * p = malloc(N * sizeof(int));

    //code to do something with the array

    return;
}
```

- We can no longer reach the block of memory allocated inside `foo`!
  - We have no way of getting the address of that block. (can't free it).

Memory Leaks

```
void foo(int N) {
    //allocate an array of N integers
    int * p = malloc(N * sizeof(int));

    //code to do something with the array

    free(p); //avoid memory leak

    return;
}
```

Memory Leaks

- Memory leaks are bugs
- Eat up memory space as long as program is running
- When program terminates (that memory space is made available to other programs by *most* operating systems (OS))

Calling `free` Early

```
int* foo(int N) {
    //allocate an array of N integers
    int * p = malloc(N * sizeof(int));
    //code to do something with the array
    free(p); //THIS IS TOO EARLY!
    return p;
}
```

Calling `free` – is it safe?

```
int ** bar(int N) {
//allocate an array of N integers
int * p = malloc(N * sizeof(int));
//allocate space for an int*
int ** q = malloc(sizeof(int *));
//the box q now holds the address of the array
* q = p;
//return the address of the box q points to.
return q;
}

int** i = bar(10); //i points to a box which points
to the array
(* i)[5] = 12; //this sets the 6th element of the
array.
(* i) = NULL; //now i points to a box which contains
NULL
```

Calling **free** – is it safe?

```
int ** bar(int N) {
//allocate an array of N integers
int * p = malloc(N * sizeof(int));
//allocate space for an int*
int ** q = malloc(sizeof(int *));
//the box q now holds the address of the array
* q = p;
//return the address of the box q points to.
return q;
}

int** i = bar(10); //i points to a box which points
to the array
(* i)[5] = 12; //this sets the 6th element of the
array.
free(*i); //free the array.
(* i) = NULL; //now i points to a box which contains
NULL
```

Calling **free** twice

```
int* foo(int N) {
     int * f_p = malloc(N * sizeof(int));
     //..some code here
     free(f_p);
     return f_p;
}
void bar(int* x) {
     int * b_p = malloc(N * sizeof(int));
     if(x != NULL)
       free(x) //freeing twice. Frees b_p as well if
b_p == x
    return;
}
main(){
    int* f_x=foo(10);
    bar(f_x);
}
```

Detecting Memory Leaks

- Detecting memory leaks can be tricky
    - Not free the memory early
    - Free the memory late enough
        - Be absolutely sure that you are done with it (is it safe)?
- Use `valgrind`

valgrind

- Does more than just memory leak detection.
  - profiling, memory analysis

- Options that we use to detect memory leaks:

```
--tool=memcheck --leak-check=full
```