Pointers

Office Hours

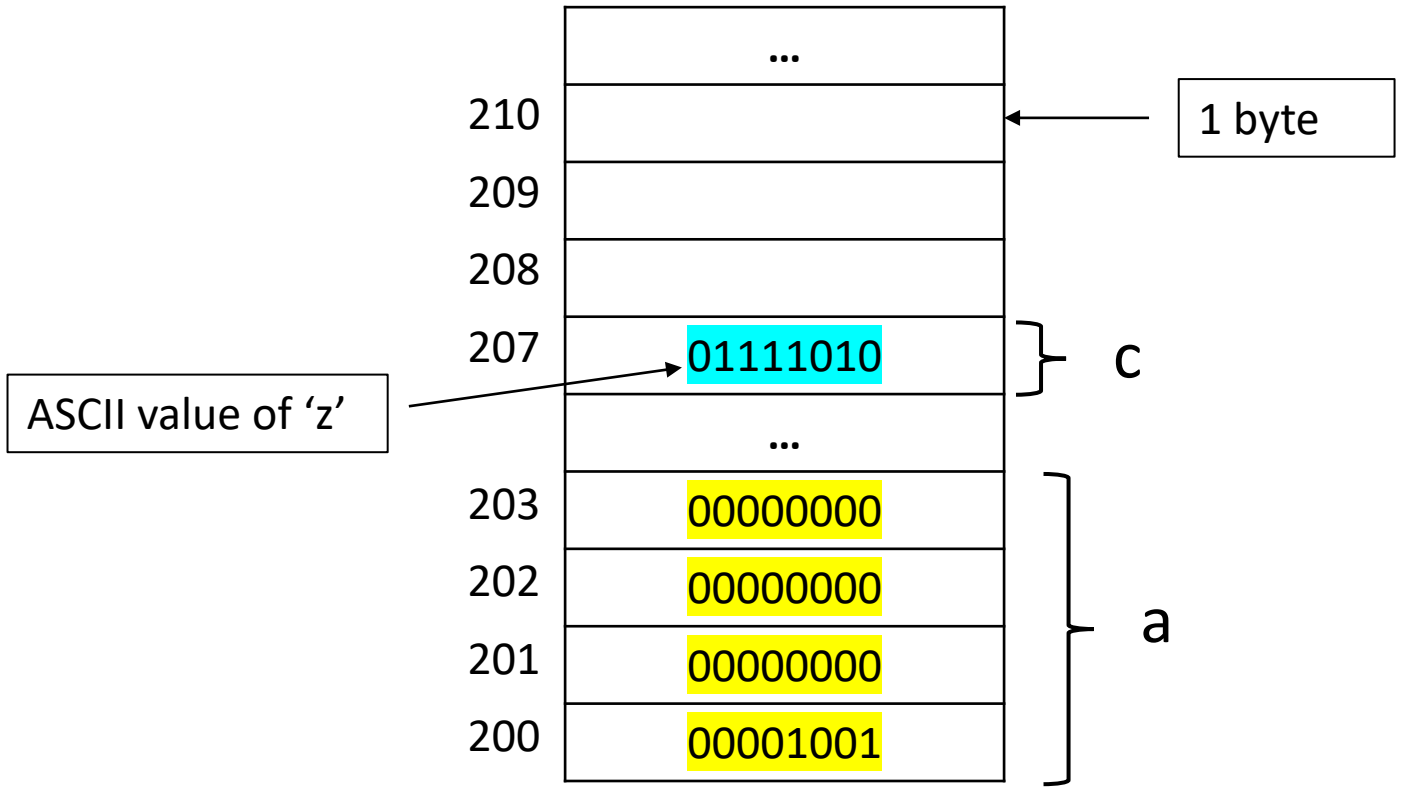
- Thursday 1 to 3 pm
- Friday 1 to 3 pm

Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
pointer	4	8	8

How a C variable is stored in memory

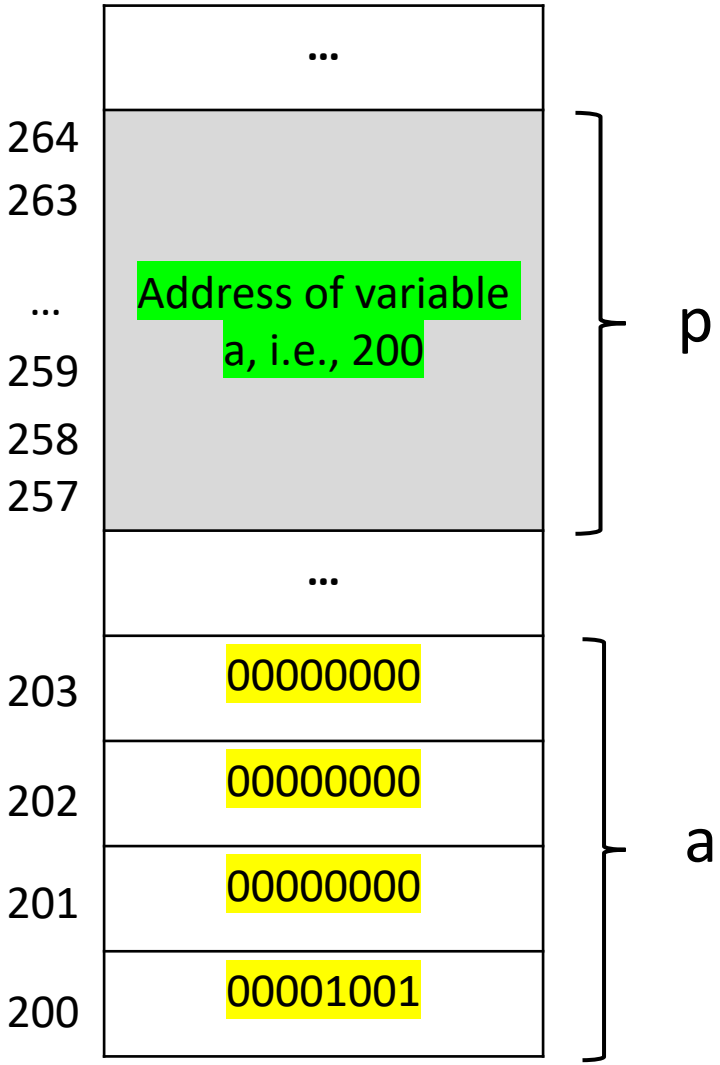
```
int main()
{
        int a = 9;
        char c = ‘z’;
}
```



Usually, addresses are represented as Hex values. For simplicity, I denote addresses as decimals

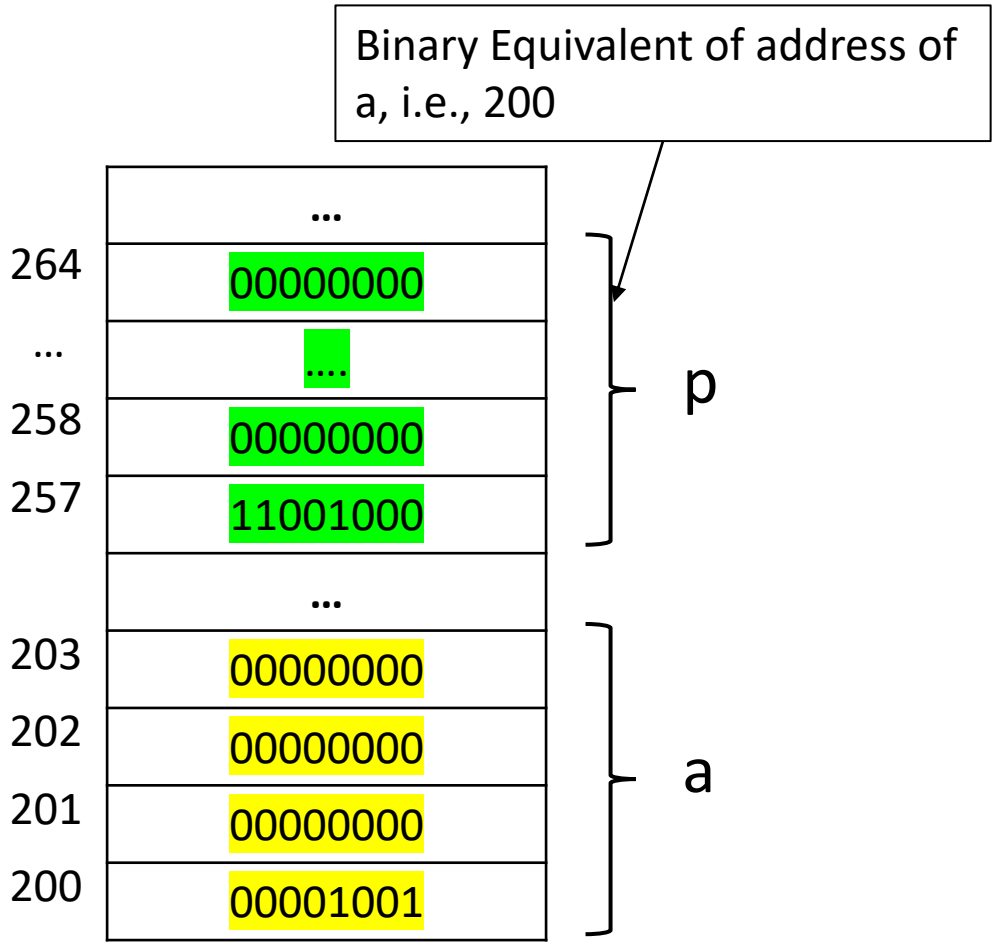
A pointer to variable ‘a’

```
int main()
{
        int a = 9;
        int *p = &a;

        printf(“&a = %d\n”, &a);
        printf(“p = %d\n”, p);
        printf(“*p = %d\n”, *p);

}
```



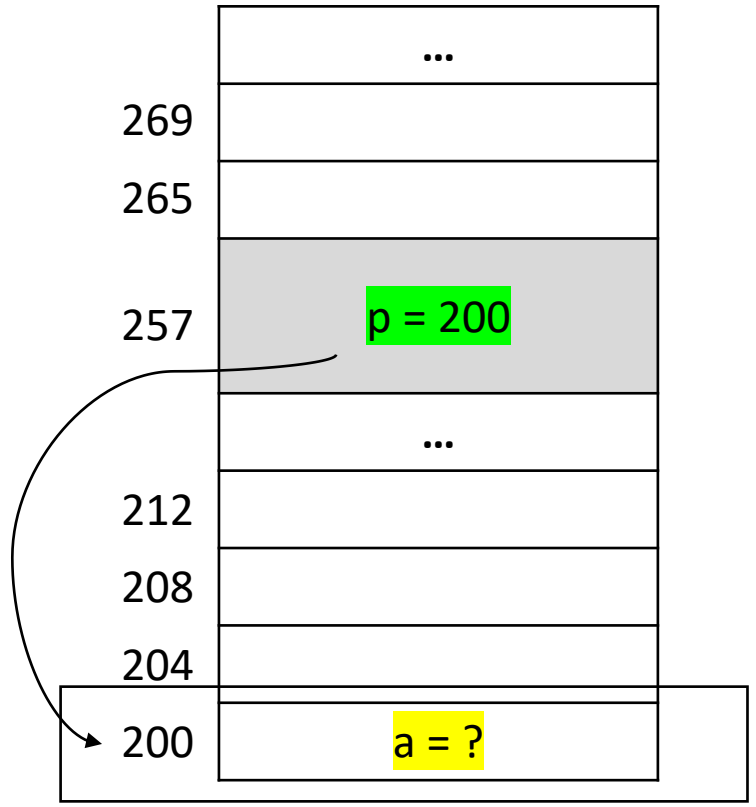
A pointer to variable ‘a’

```
int main()
{
        int a = 9;
        int *p = &a;

        printf(“&a = %d\n”, &a); //&a = 200
        printf(“p = %d\n”, p); //p = 200
        printf(“*p = %d\n”, *p); //*p = 9

}
```



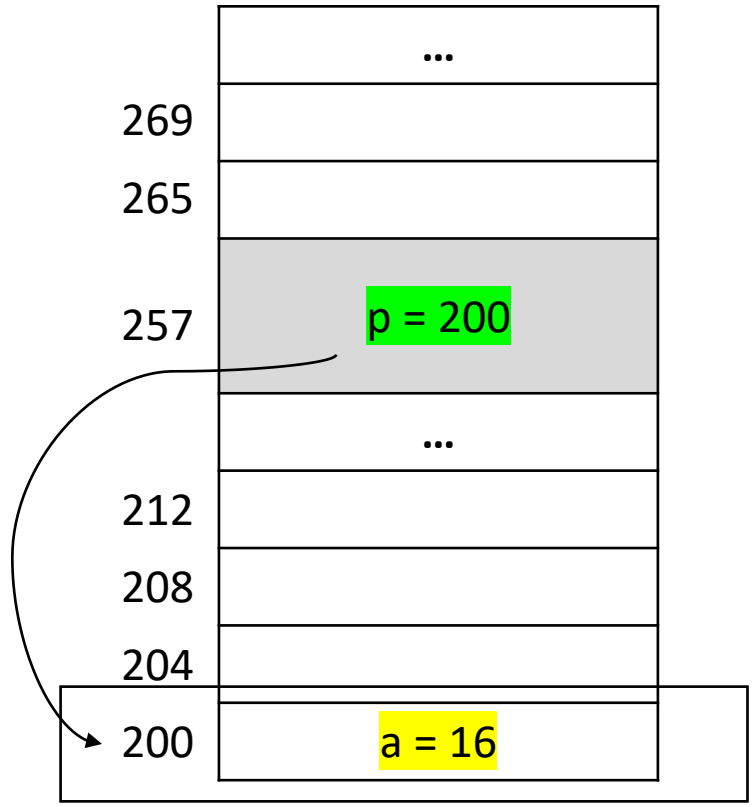
A pointer to variable ‘a’

```
int main()
{
        int a = 9;
        int *p = &a;

        printf(“&a = %d\n”, &a); //&a = 200
        printf(“p = %d\n”, p); //p = 200
        printf(“*p = %d\n”, *p); //*p = 9
        *p = 16;
        printf(“a = %d\n”, a);
}
```



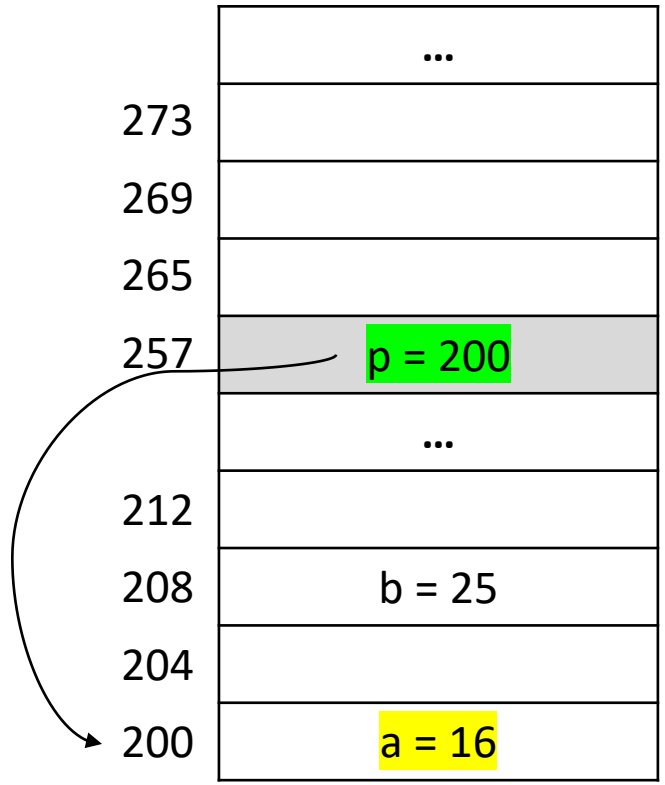
A pointer to variable ‘a’

```
int main()
{
        int a = 9;
        int *p = &a;

        printf(“&a = %d\n”, &a); //&a = 200
        printf(“p = %d\n”, p); //p = 200
        printf(“*p = %d\n”, *p); //*p = 9
        *p = 16;
        printf(“a = %d\n”, a); //a = 16

}
```



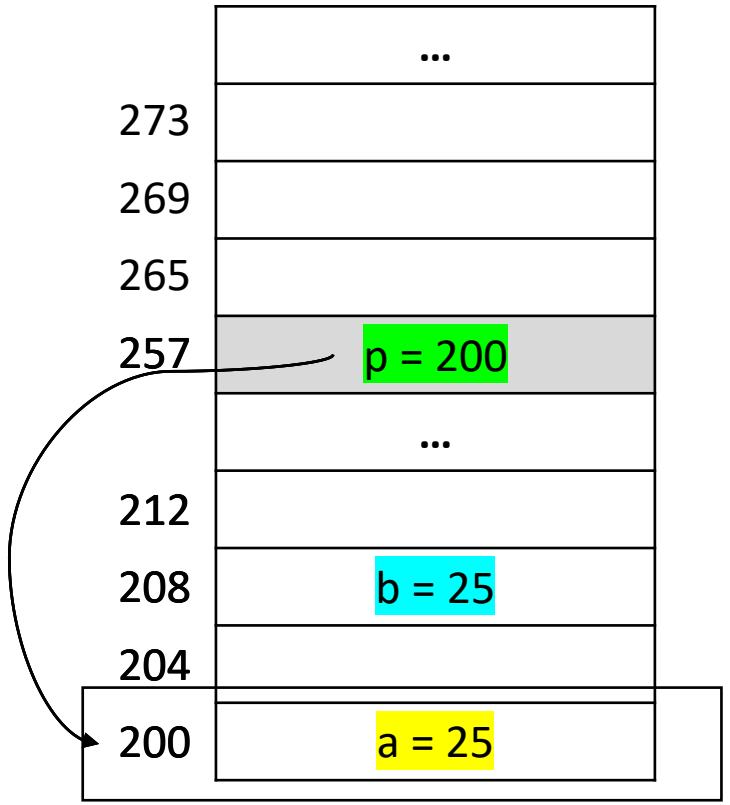
A pointer to variable ‘a’

```
int main()
{
        int a = 9;
        int *p = &a;
        int b = 25;

        printf(“&a = %d\n”, &a); //&a = 200
        printf(“p = %d\n”, p); //p = 200
        printf(“*p = %d\n”, *p); //*p = 9
        *p = 16;
        printf(“a = %d\n”, a); //a = 16
        *p = b;
        printf(“p = %d\n”, p);
        printf(“a = %d\n”, a);
}
```



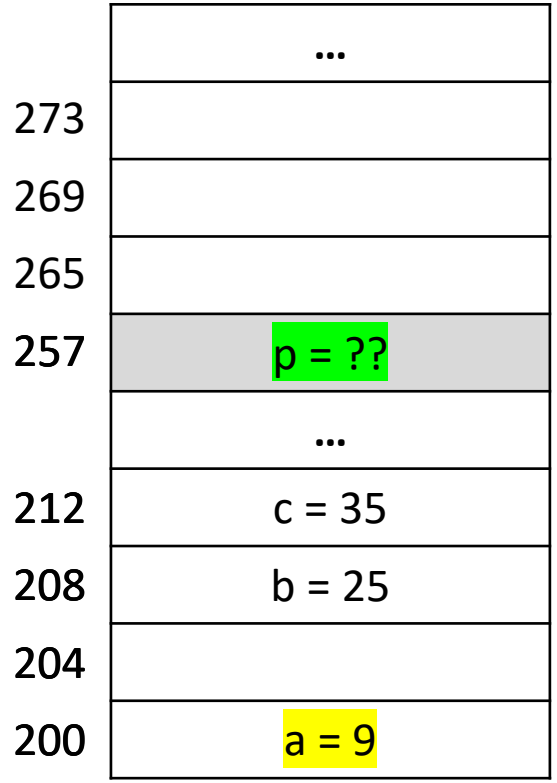
A pointer to variable ‘a’

```
int main()
{
        int a = 9;
        int *p = &a;
        int b = 25;

        printf(“&a = %d\n”, &a); //&a = 200
        printf(“p = %d\n”, p); //p = 200
        printf(“*p = %d\n”, *p); //*p = 9
        *p = 16;
        printf(“a = %d\n”, a); //a = 16
        *p = b;
        printf(“p = %d\n”, p); //p = 200
        printf(“a = %d\n”, a); //a = 25

}
```



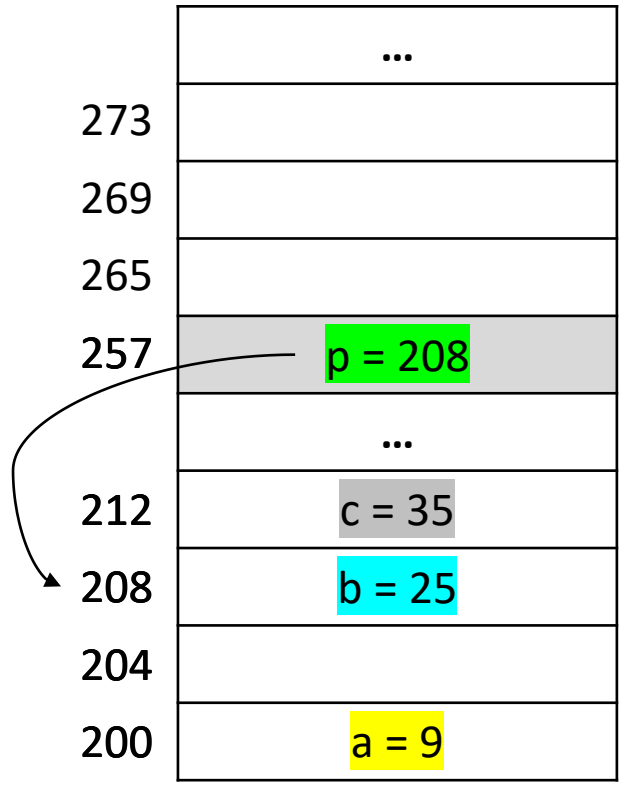
Pointer Arithmetic

```
int main()
{
        int a = 9;
        int *p = &a;
        int b = 25;
        int c = 35;
        p = p + 2;
        printf("p = %d\n", p);
        printf("*p = %d\n", *p);
        printf("b = %d\n", b);
}
```



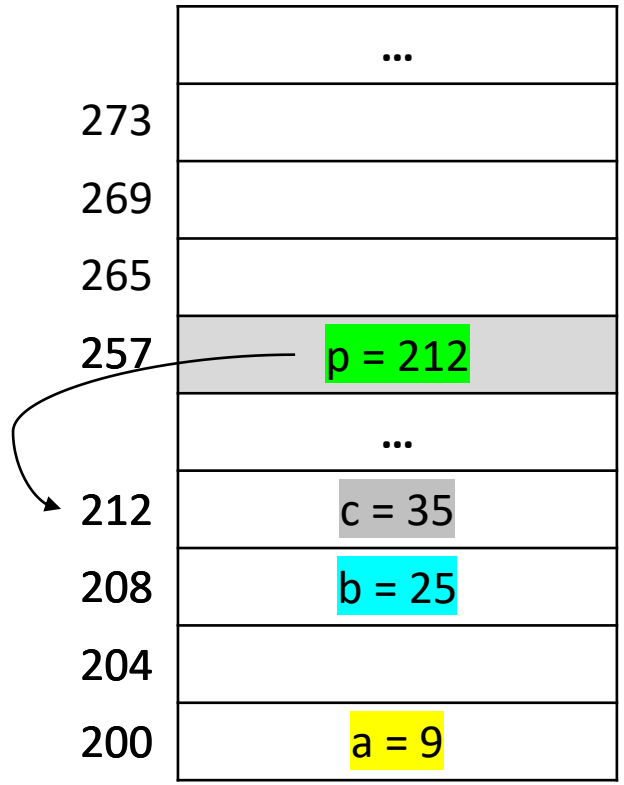
Pointer Arithmetic

```
int main()
{
        int a = 9;
        int *p = &a;
        int b = 25;
        int c = 35;
        p = p + 2; //p = p + 2*(sizeof(int))
        printf("p = %d\n", p); //p = 208
        printf("*p = %d\n", *p); //*p = 25
        printf("b = %d\n", b); //b = 25
}
```



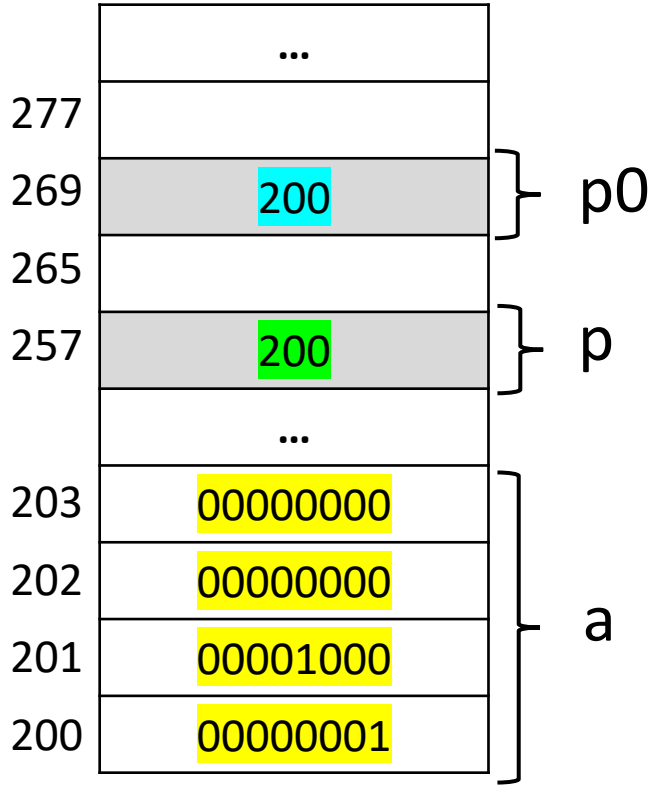
Pointer Arithmetic

```
int main()
{
        int a = 9;
        int *p = &a;
        int b = 25;
        int c = 35;
        p = p + 2; //p = p + 2*(sizeof(int))
        printf("p = %d\n", p); //p = 208
        printf("*p = %d\n", *p); //*p = 25
        printf("b = %d\n", b); //b = 25
        p = p + 1; //p = p + 1* (sizeof(int))
        printf("p = %d\n", p); //p = 212
        printf("*p = %d\n", *p); //*p = 35
}
```



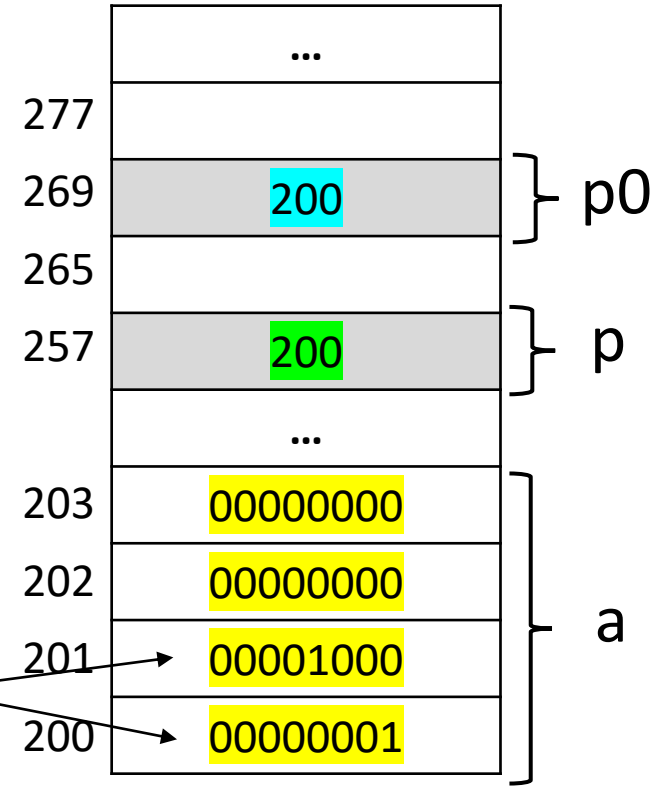
Give output of the following:

```
int main()
{
        int a = 2049;
        int *p = &a;
        printf("Address = %d, Value = %d\n", p, *p);
        printf("Address = %d, Value = %d\n", p+1, *(p+1));
        char *p0;
        p0 = (char*)p;
        printf("Address = %d, Value = %d\n", p0, *p0);
        printf("Address = %d, Value = %d\n", p0+1, *(p0+1));
}
```



Give output of the following:

```
int main()
{
        int a = 2049;
        int *p = &a;
        printf("Address = %d, Value = %d\n", p, *p);
        //Address = 200, Value = 2049
        printf("Address = %d, Value = %d\n", p+1, *(p+1));
        //Address = 204, Value = -867181191
        char *p0;
        p0 = (char*)p;
        printf("Address = %d, Value = %d\n", p0, *p0);
        //Address = 200, Value = 1
        printf("Address = %d, Value = %d\n", p0+1, *(p0+1));
        //Address = 201, Value = 8
}
```

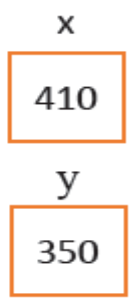


Exercise:

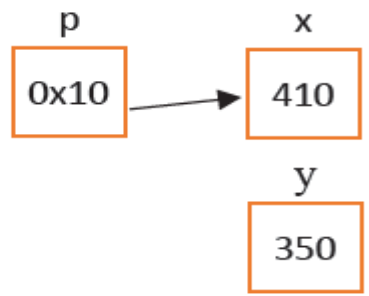
Draw out the memory diagram after sequential execution of each of the lines below:

```
int main(int argc, char **argv) {
  int x = 410, y = 350;   // assume &x = 0x10, &y = 0x14
  int *p = &x;            // p is a pointer to an integer
  *p = y;
  p = p + 4;
  p = &y;
  x = *p + 1;
}
```

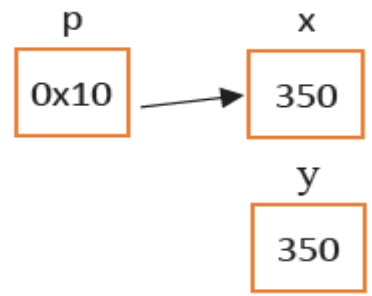
Line 1:



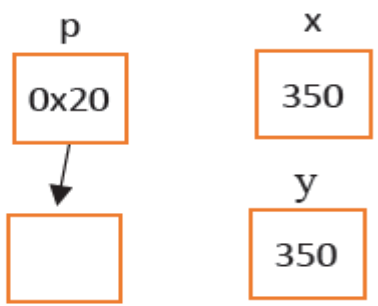
Line 2:



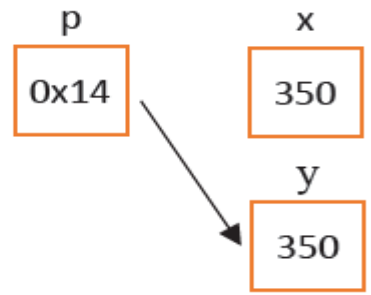
Line 3:



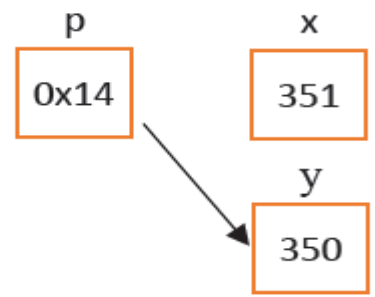
Line 4:



Line 5:

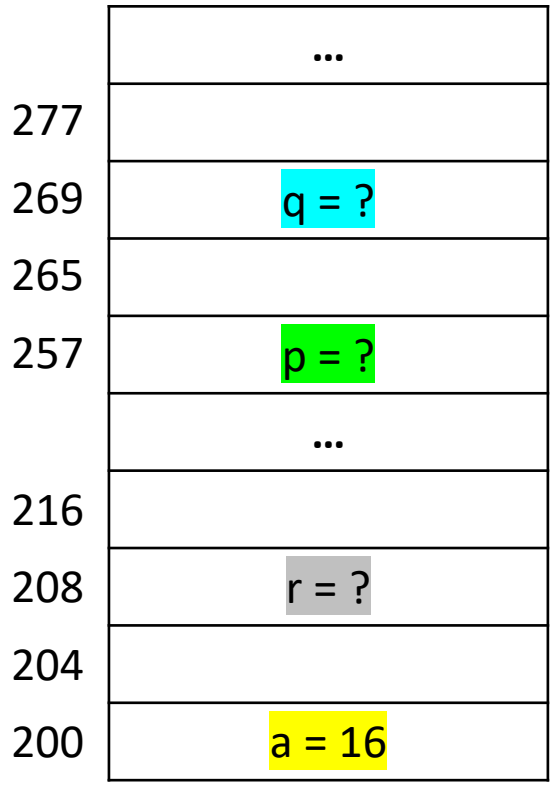


Line 6:



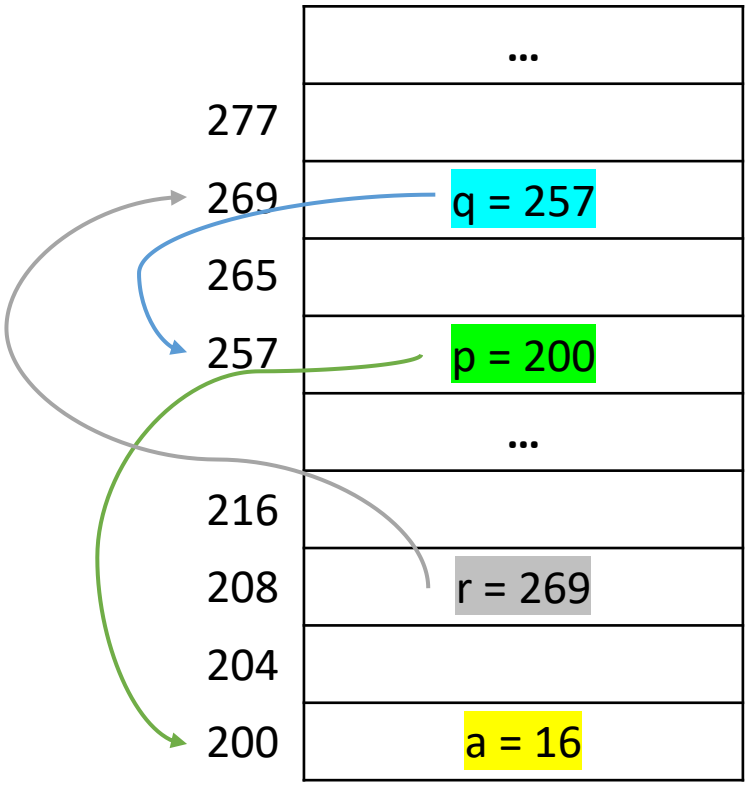
Pointer to pointer to pointer

```
int main()
{
	int a = 16;
	int *p = &a;
	int **q = &p;
	int ***r = &q;
}
```



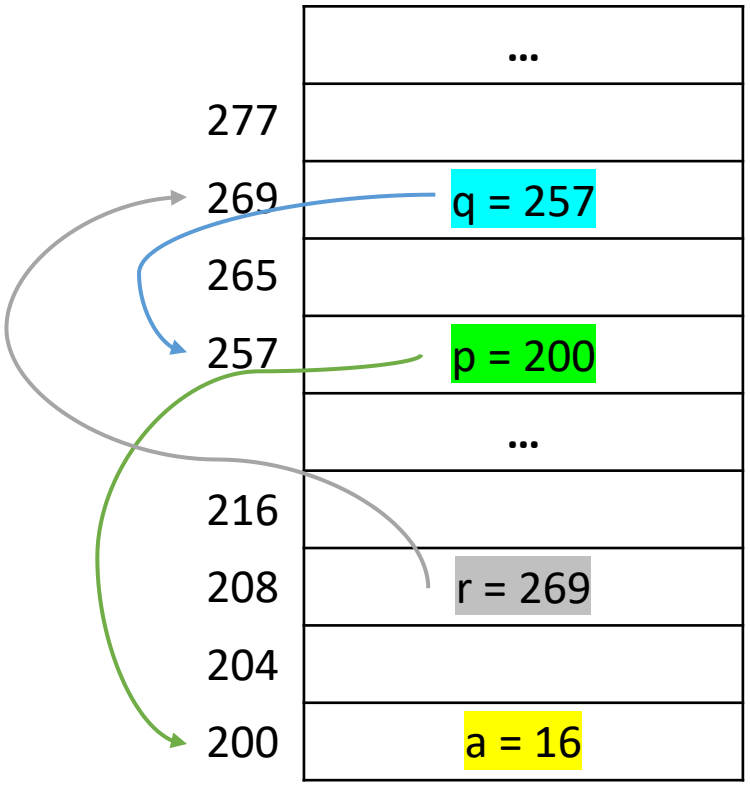
Pointer to pointer to pointer

```
int main()
{
        int a = 16;
        int *p = &a;
        int **q = &p;
        int ***r = &q;
}
```



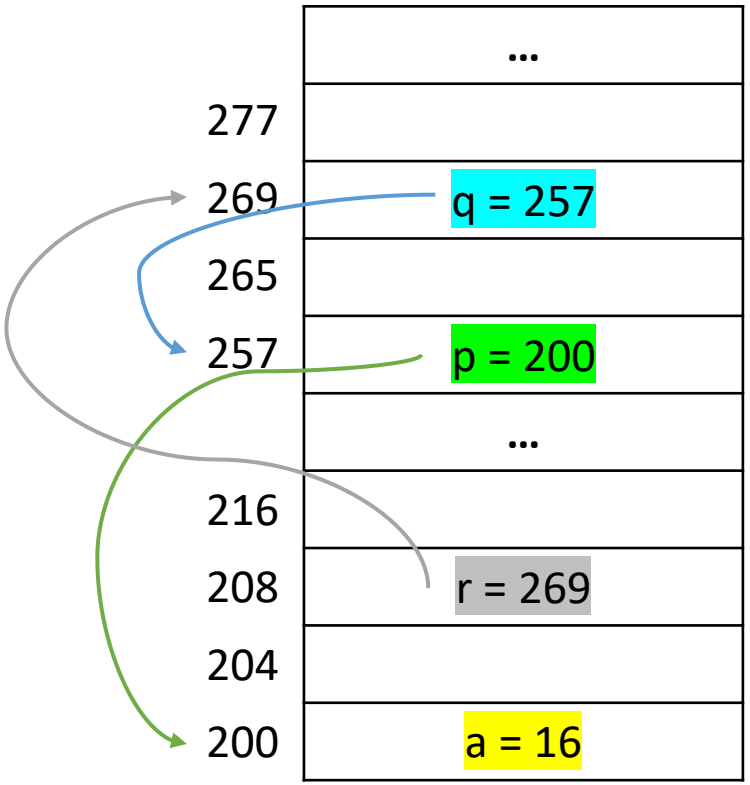
Pointer to pointer to pointer

```
int main()
{
        int a = 16;
        int *p = &a;
        int **q = &p;
        int ***r = &q;
        printf(“%d\n”, *p);
        printf(“%d\n”, *q);
        printf(“%d\n”, *r);
        printf(“%d\n”, **q);
        printf(“%d\n”, ***r);
}
```



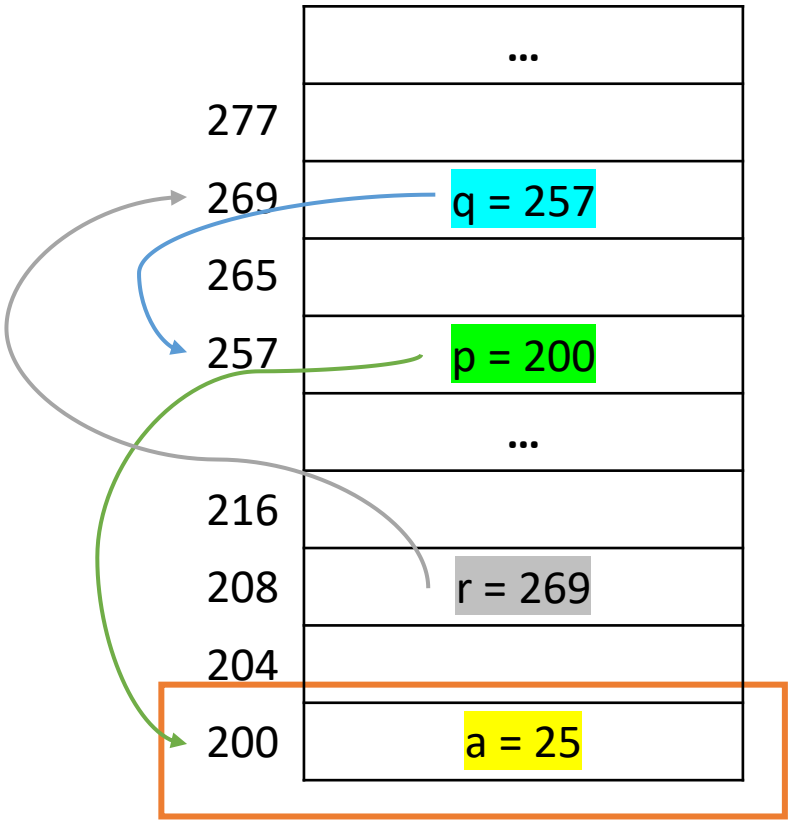
Pointer to pointer to pointer

```
int main()
{
        int a = 16;
        int *p = &a;
        int **q = &p;
        int ***r = &q;
        printf("%d\n", *p); //16
        printf("%d\n", *q); //200
        printf("%d\n", *r); //257
        printf("%d\n", **q); //16
        printf("%d\n", ***r); //16
}
```



Pointer to pointer to pointer

```
int main()
{
        int a = 16;
        int *p = &a;
        int **q = &p;
        int ***r = &q;
        printf("%d\n", *p); //16
        printf("%d\n", *q); //200
        printf("%d\n", *r); //257
        printf("%d\n", **q); //16
        printf("%d\n", ***r); //16
        ***r = 25;
        printf("%d\n", a); //25
}
```



Exercise

- Give output of the following code snippet

```
int main()
{
	int a = 10;
	int *p = &a;
	int **q = &p;
	printf("value of q: %d, value of q+1: %d\n", q, q+1);
	//Value of q: 2000, q+1: 2008
}
```

Assume that the address of variable p is 2000.

Call by value

```
void increment(int a)
{
        printf(“Address of a = %d”, &a);
        a = a + 1;
        printf(“a = %d\n”, a);
}
int main()
{
        int a = 10;
        increment(a);
        printf(“a = %d\n”, a);
        printf(“Address of a = %d”, &a);
}
```

Call by value

```
void increment(int a)
{
        printf("Address of a = %d", &a);
        //"Address of a = 234000
        a = a + 1;
        printf("a = %d\n", a); //a = 11
}
int main()
{
        int a = 10;
        increment(a);
        printf("Address of a = %d", &a);
        //"Address of a = 222000
        printf("a = %d\n", a); //a = 10
}
```

NO CHANGE IN value of a

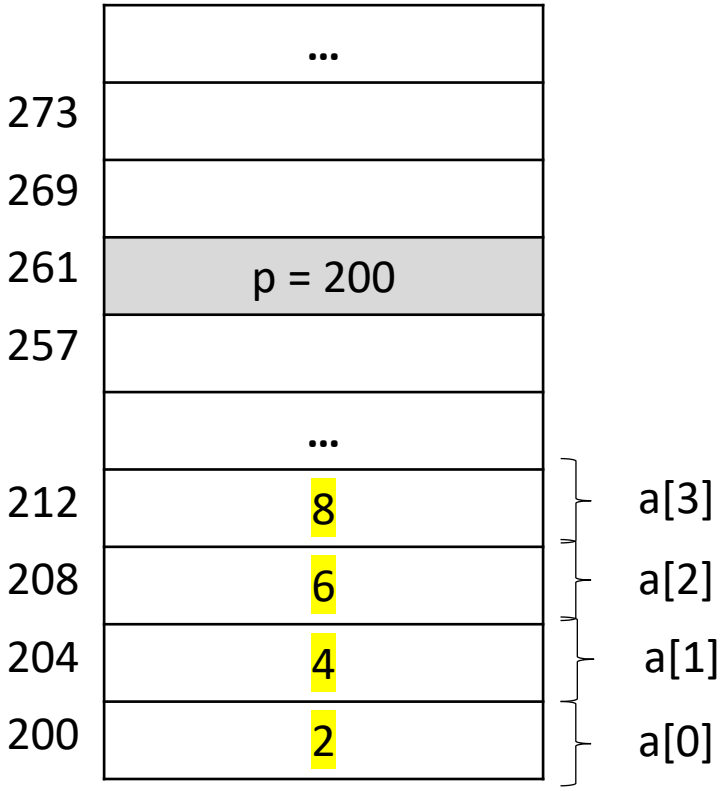
Pointers as function arguments – Call by Reference

```
void increment(int *p)
{
        *p = (*p) + 1;
}
int main()
{
        int a = 10;
        increment(&a);
        printf("a = %d\n", a); //a = 11
}
```

CHANGE IN value of a

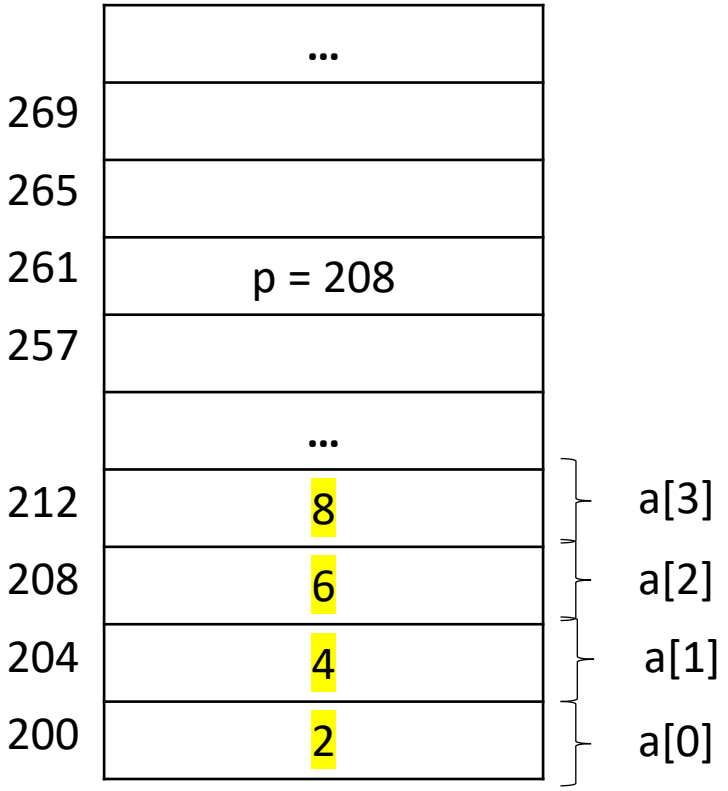
Pointers and Arrays

```
int a[] = {2, 4, 6, 8};

int *p = a; //Equivalent to *p = &a[0]

printf("%d , %d", (p+1), *(p+1)); //204, 4
printf("%d , %d", (a+1), *(a+1)); //204, 4
printf("%d , %d", &a[1], a[1]); //204, 4
```



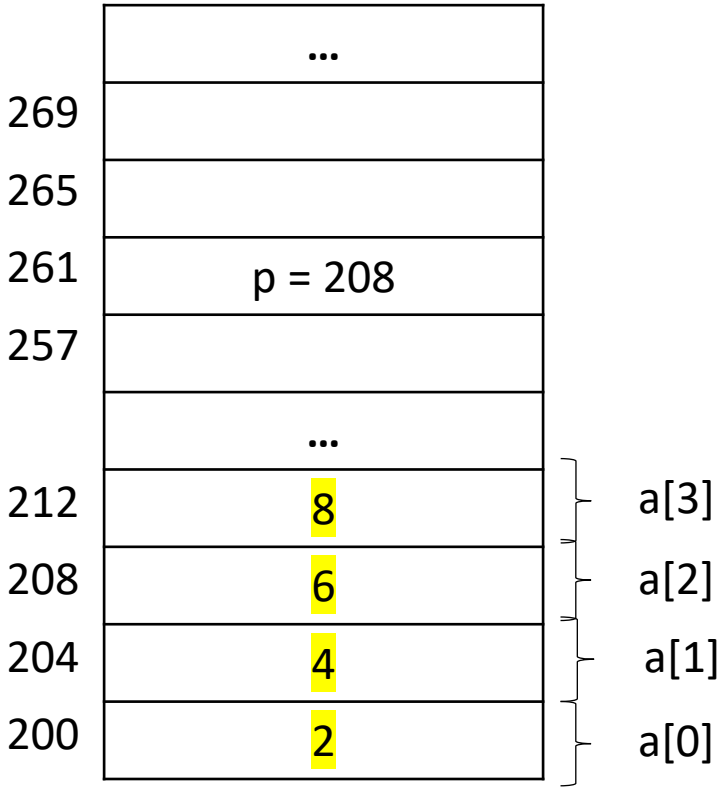
Pointers and Arrays

```
int a[] = {2, 4, 6, 8};

int *p = a; //Equivalent to int *p = &a[0]

printf("%d , %d", (p+1), *(p+1)); //204, 4
printf("%d , %d", (a+1), *(a+1)); //204, 4

p = p + 2
printf("%d , %d", p, *p);

a = a + 2;
```



Pointers and Arrays

```
int a[] = {2, 4, 6, 8};

int *p = a; //Equivalent to *p = &a[0]

printf("%d , %d", (p+1), *(p+1)); //204, 4
printf("%d , %d", (a+1), *(a+1)); //204, 4

p = p + 2
printf("%d , %d", p, *p); //208, 6

a = a + 2; //INVALID. a always points to the
base address
```



Character arrays and strings

```
#include <stdio.h>
#include <string.h>
int main()
{
	char c[] = “Mary”;
	char d[3] = “Tea”;
	char e[6] = “Chair”;
	printf(“Size in bytes: %d\n”, sizeOf(c));

	int len = strlen(c);
	printf(“Length of string: %d\n”, len);

}
```

Character arrays and strings

```
#include <stdio.h>
#include <string.h>
int main()
{
        char c[] = “Mary”; //Equivalent to char c[] = {‘M’, ‘a’, ‘r’, ‘y’, ‘\0’}
        char d[3] = “Tea”; //Invalid because it cannot accommodate ‘\0’
        char e[6] = “Chair”; //Equivalent to char e[6] = {‘C’, ‘h’, ‘a’, ‘i’, ‘r’,‘\0’}
        printf(“Size in bytes: %d\n”, sizeOf(c));
        //Size in bytes: 5

        int len = strlen(c);
        printf(“Length of string: %d\n”, len);
        //Length of string: 4
}
```

Character arrays as pointers

```
char c1[10] = “Hello”;
char *c2 = c1;
printf(“%c, %c”, c2[1], *(c2 + 1)); //e, e

*(c2) = z;
printf(“%s”, c1); //zello

c1 = c2; //Invalid because c1 points to base address of the array c1[]
c2 ++; //Valid. Now c2 points to the address of the second character
printf(“%c”, *c2); //e
```

Arrays as function arguments

```
void sum(int A[])
{
        int i, sum = 0;
        printf("Size of A: %d, Size of A[0]: %d \n", sizeof(A), sizeof(A[0]));
}
int main()
{
        int A[] = {1, 2, 3, 4, 5};
        sum(A);
        printf("Size of A: %d, Size of A[0]:%d \n", sizeof(A), sizeof(A[0]));
        return 0;
}
```

Arrays as function arguments

```
void sum(int A[])
{
        int i, sum = 0;
        printf("Size of A: %d, Size of A[0]: %d \n", sizeof(A), sizeof(A[0]));
        //Size of A: 8, Size of A[0]: 4
}
int main()
{
        int A[] = {1, 2, 3, 4, 5};
        sum(A);
        printf("Size of A: %d, Size of A[0]:%d \n", sizeof(A), sizeof(A[0]));
        //Size of A: 20, Size of A[0]: 4
        return 0;
}
```

same as **void sum(int *A)**

Base address of array A

2.2 The following functions may contain logic or syntax errors. Find and correct them.

(a) Returns the sum of all the elements in `summands`.

```
1  int sum(int* summands) {
2      int sum = 0;
3      for (int i = 0; i < sizeof(summands); i++)
4          sum += *(summands + i);
5      return sum;
6  }
```

(a) Returns the sum of all the elements in summands.

It is necessary to pass a size alongside the pointer.

```
1  int sum(int* summands, size_t n) {
2      int sum = 0;
3      for (int i = 0; i < n; i++)
4          sum += *(summands + i);
5      return sum;
6  }
```

Lab 2

4 Programming Task

Your assignment is to complete each function skeleton according to the following rules:

- Only straight-line code (i.e., no loops or conditionals) unless otherwise stated. Look for "Control Constructs" under ALLOWED in `pointer.c` file comments.
- A limited number of C arithmetic and logical operators (described in `pointer.c` comments)
- No constants larger than 8 bits (i.e., `0` − `255` inclusive) are allowed
- Feel free to use `(`, `)`, and = as much as you want
- You are permitted to use casts for these functions

Problem 1

- **Pointer Arithmetic:** The first three functions in `pointer.c` ask you to compute the size (how much memory a single one takes up, in bytes) of various data elements (ints, doubles, and pointers). You will accomplish this by noting that arrays of these data elements allocate contiguous space in memory so that one element follows the next.

Problem 2

- **Manipulating Data Using Pointers**: The next two functions in `pointer.c` challenge you to manipulate data in new ways with your new knowledge of pointers. The `swapInts` function asks you to swap the values that two given pointers point to, without changing the pointers themselves (i.e. they should still point to the same memory addresses). The `changeValue` function asks you to change the value of an element of an array using only the starting address of the array. You will add the appropriate value to the pointer to create a new pointer to the data element to be modified. **You are not permitted to use [ ] syntax to access or change elements in the array anywhere in the `pointer.c` file.**

Problem 3

- **Pointers and Address Ranges**: The next two functions in `pointer.c` ask you to determine whether pointers fall within certain address ranges, defined by aligned memory blocks or arrays. For the first of these two functions, you will determine if the addresses stored by two pointers lie within the same block of 64-byte aligned memory. The following are some examples of parameters and returns for calls to this function.

Problem 3

```
- ptr1:  0x0
  ptr2:  0x3F
  return:  1
- ptr1:  0x0
  ptr2:  0x40
  return:  0
- ptr1:  0x3F
  ptr2:  0x40
  return:  0
- ptr1:  0x3CE
  ptr2:  0x3EF
  return:  1
- ptr1:  0x3CE
  ptr2:  0x404
  return:  0
```

withinSameBlock() Hints

- What is the size of an address?
  - 8 Bytes = 64 bits
  - So, how many possible unique addresses can you have in Memory?
    - 2^{64}

- How many bytes does a block contain for *"64 bytes aligned memory"*?
  - 64 Bytes = 2^6 Bytes

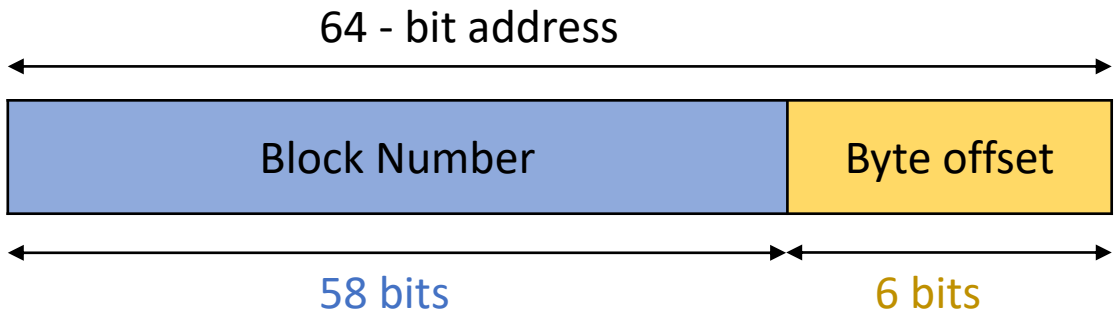
- So, how many blocks are there?
  - (Total No. of Addresses)/(Block Size) = $2^{64}/\ 2^6 = 2^{58}$

Blocks

- How many unique blocks are there?
  - 2^{58}
- How many bits do you need to represent a block Number?
  - 58 bits

Bytes in a Block

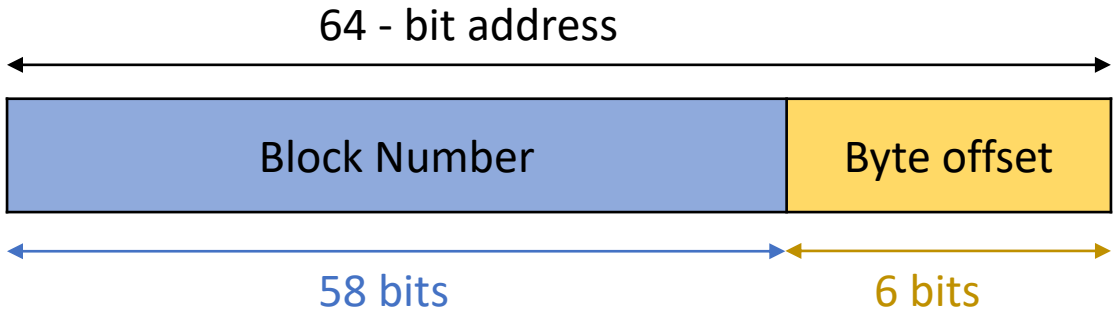
- How many unique Bytes within a block?
  - $64 = 2^6$
- How bits do you need to represent a Byte within a block?
  - 6 bits



Memory

Block 0	B0	B1	…	B63
Block 1	B64	B65	…	B127
Block 2	B128	B129	…	B191
Block 3	B192	B193	…	B255
	… … …			
Block $2^{58} - 2$	…	…	…	…
Block $2^{58} - 1$	…	…	…	$B2^{64} - 1$

withinSameBlock() Hints



- So, you do not need to worry about the least significant 6 bits
  - If two addresses are in the same block, then **their block numbers will be same!!**

- How do you figure if the block numbers are the same?

withinArray

- Size is the number of ints contained in the array
  - Can assume size != 0
- Pointing anywhere in the array is fair game, ptr does not have to point to the beginning of an element.
- Hint
  - Use the size to calculate the range of address within which the ptr should lie

stringLength

- Return the length of a string, given a pointer to its beginning
- Can use loops
  - When do you stop the loop?
- Null terminator character does not count as part of the string length

Little Endian and Big Endian Data Storage

- Big-endian: is an order in which the "big end" (most significant value in the sequence) is stored first (at the lowest storage address).

- Little-endian is an order in which the "little end" (least significant value in the sequence) is stored first (at the lowest storage address).

- Example: For Hex Number 0x4F52
  - **Big Endian:** if 4F is stored at storage address 1000, 52 will be at address 1001)

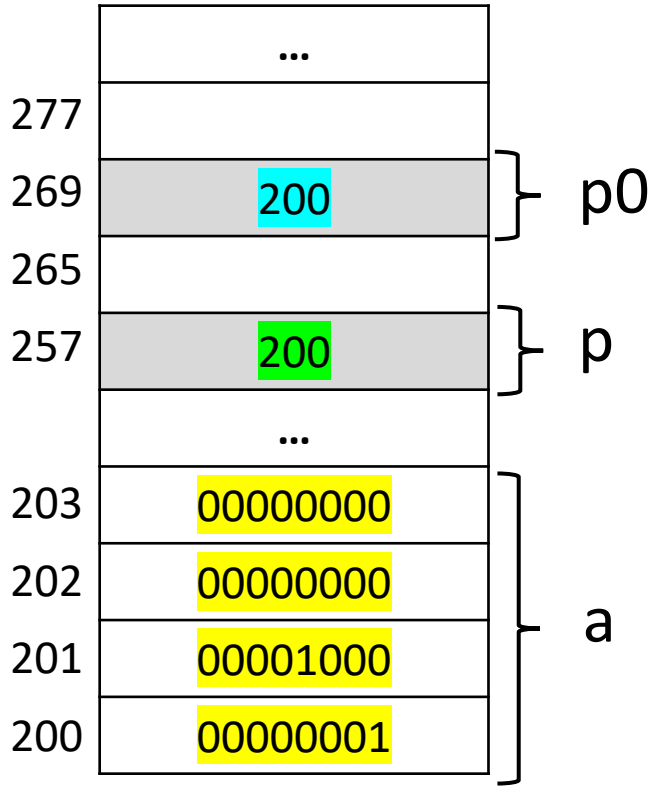
  - **Little Endian:** it would be stored as 524F (52 at address 1000, 4F at 1001)

https://searchnetworking.techtarget.com/definition/big-endian-and-little-endian

endianExperiment

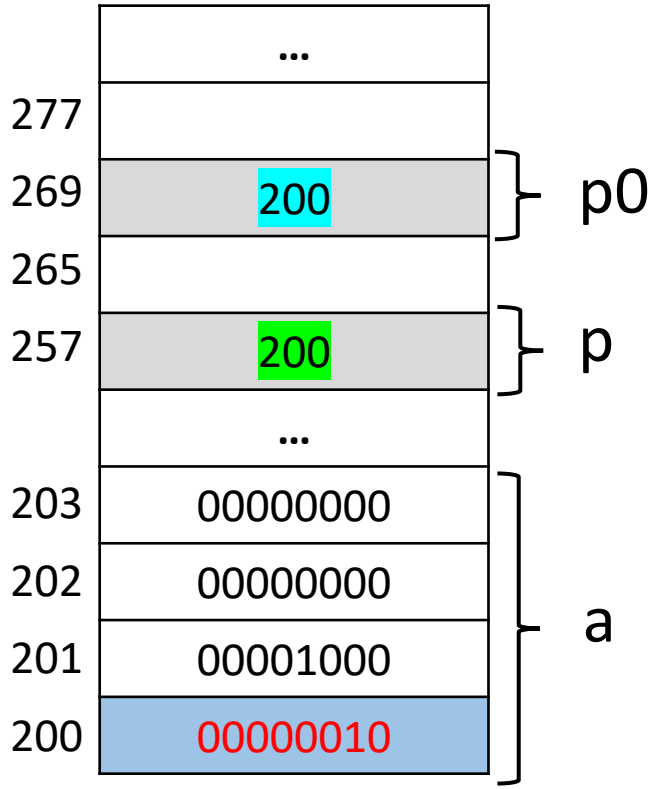
Consider this example from Slide 14

```
int *func()
{
        int a = 2049;
        int *p = &a;
        char *p0;
        p0 = (char*)p;
        printf("Address = %d, Value = %d\n", p0, *p0);
        printf("Address = %d, Value = %d\n", p0+1, *(p0+1));
        return p;
}
```



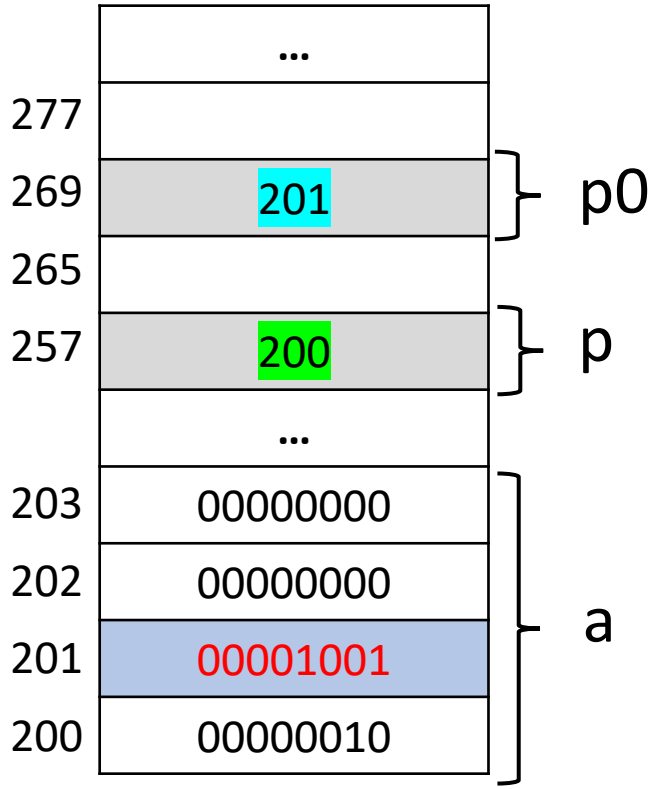
Add 1 to each of the bytes of the value of *a*

```
int *func()
{
        int a = 2049;
        int *p = &a;
        char *p0;
        p0 = (char*)p;
        *p0 = (*p0) + 1;
        return p;
}
```



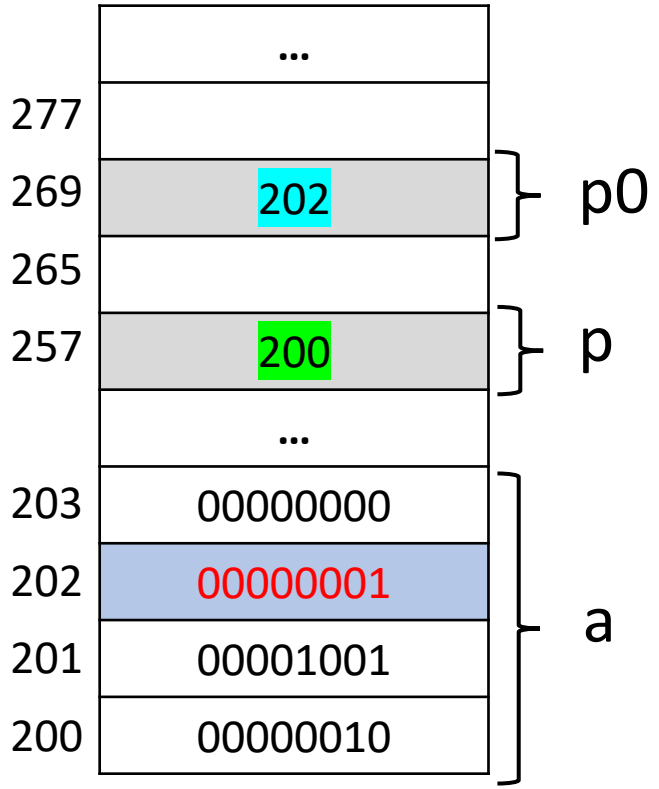
Add 1 to each of the bytes of the value of *a*

```
int *func()
{
        int a = 2049;
        int *p = &a;
        char *p0;
        p0 = (char*)p;
        (*p0)++;
        p0++;
        (*p0)++;
        return p;
}
```



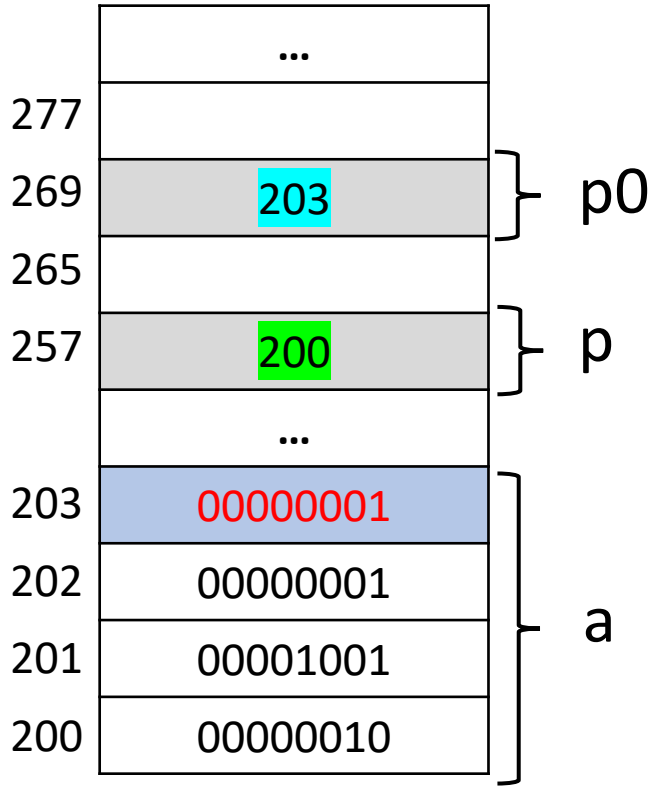
Add 1 to each of the bytes of the value of *a*

```
int *func()
{
        int a = 2049;
        int *p = &a;
        char *p0;
        p0 = (char*)p;
        (*p0)++;
        p0++;
        (*p0)++;
        p0++;
        (*p0)++;
        return p;
}
```



Add 1 to each of the bytes of the value of *a*

```
int *func()
{
        int a = 2049;
        int *p = &a;
        char *p0;
        p0 = (char*)p;
        (*p0)++;
        p0++;
        (*p0)++;
        p0++;
        (*p0)++;
        p0++;
        (*p0)++;
        return p;
}
```



stringSpan Hints

- Returns the length of the initial portion of str1 which consists only of characters that are part of str2.
  - stringSpan("abcdefgh", "abXXcdeZZh"); // returns 5
    - Uninterrupted spans from the beginning of string 1
  - stringSpan (“123456”, “156”);
    - Returns 1 (Because the span has to be uninterrupted)
  - stringSpan(“aaaab”, “a”);
    - What should it return ?
      - 4

Lab 2 Problem 5

- **Selection Sort**: The final part of the lab has you implement `selectionSort`. Selection sort works by effectively partitioning an array into a sorted section, followed by an unsorted section. It repeatedly finds (and selects) the minimum element in the unsorted section and moves it to the end of the sorted section (`swapInts` might be useful for this). The pseudo code might look something like this:

```
arr - an array
n - the length of arr

for i = 0 to n - 1
   minIndex = i
   for  j = i + 1 to n
       if arr[minIndex] > arr[j]
           minIndex = j
       end if
   end for
   Swap(arr[i], arr[minIndex])
end for
```

Note that you **are** allowed to use loops and if statements in this one.

References

- https://www.youtube.com/playlist?list=PL2_aWCzGMAwLZp6LMUKI3cc7pgGsasm2_