CS 449 Review

Pointers

- A pointer is a **variable** that holds an **address**

- Declaration:

```
type *pointer_var;
```

```
int *p;
```

- Why do pointers have types?
  1. So that we know the type (size & interpretation) when dereferencing
  2. Pointer arithmetic is scaled by the size of this type

Pointers continued

- The **address-of** operator gets the address of the right hand side:

```
int x = 2;
int *p = &x;
```

- The **dereference** operator goes to the address of the pointer and gets the value:

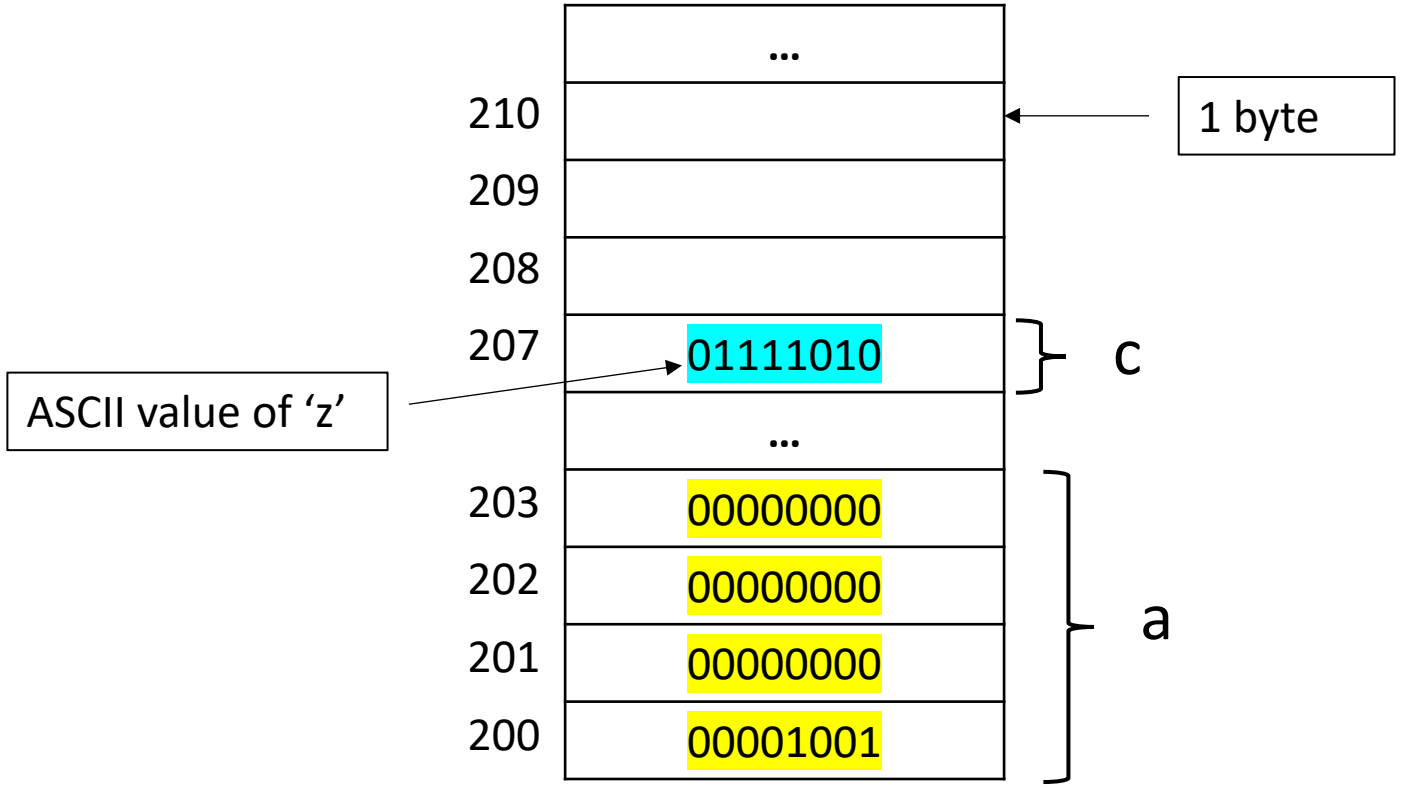
```
printf(“%d\n”, *p);    //prints 2 since *p is an alias of x
```

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

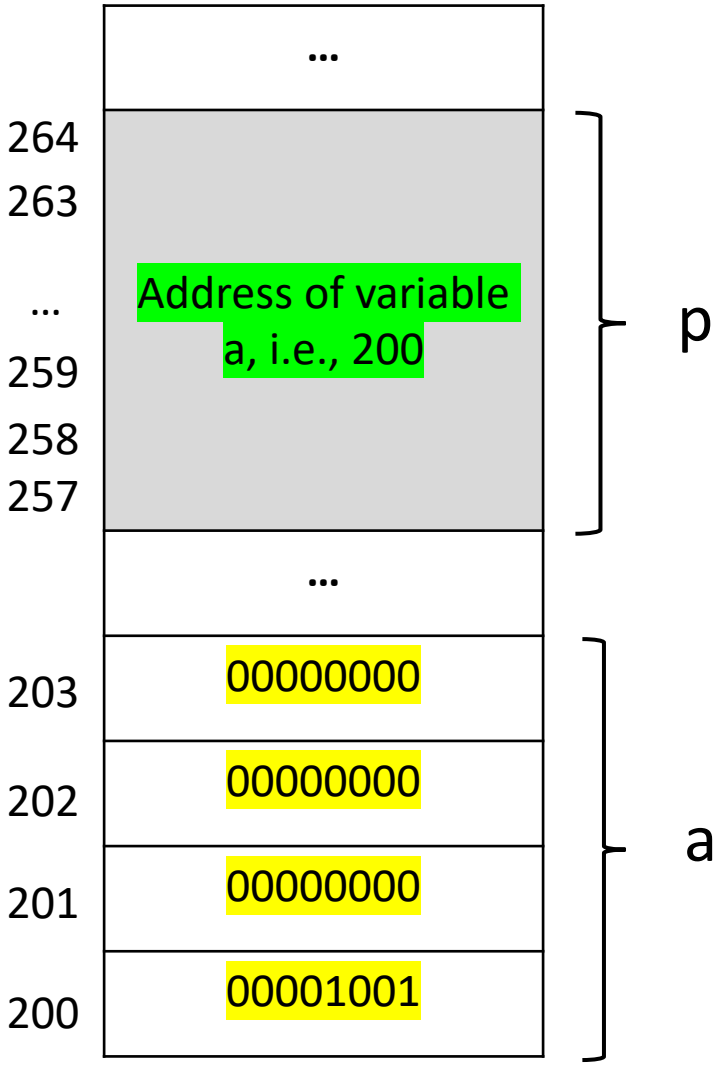
```
int main()
{
        int a = 9;
        char c = ‘z’;
}
```



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

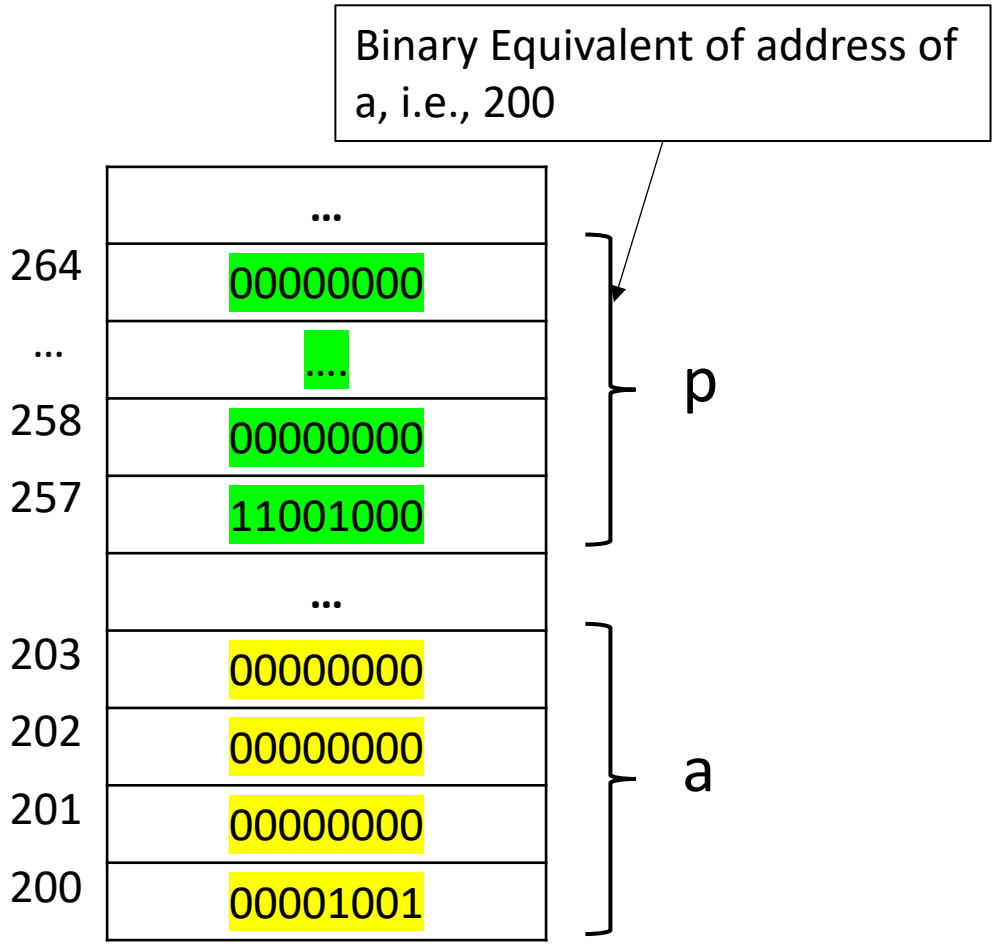
A pointer to variable ‘a’

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

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

}
```



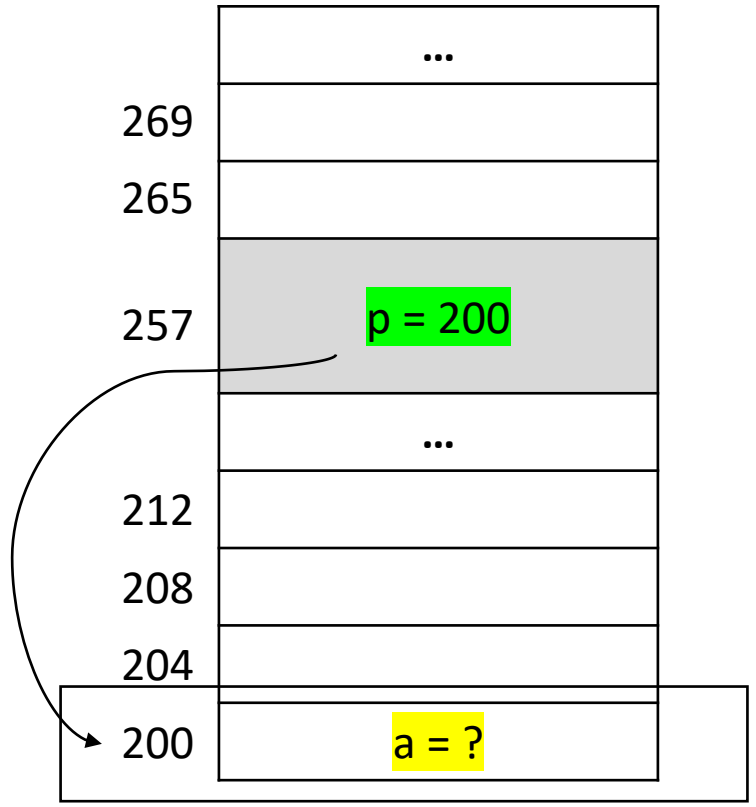
A pointer to variable ‘a’

```
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

}
```



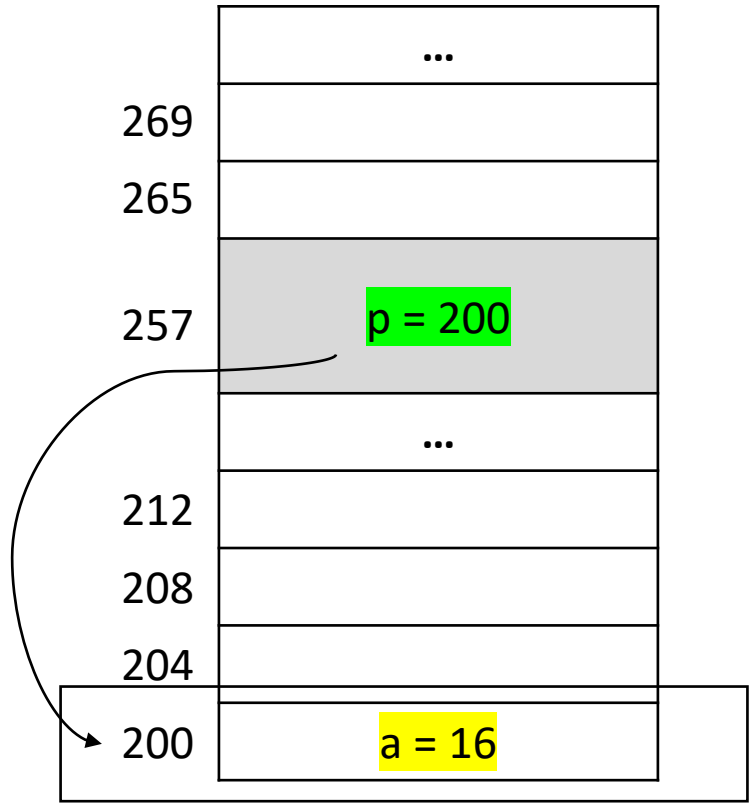
A pointer to variable ‘a’

```
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);
}
```



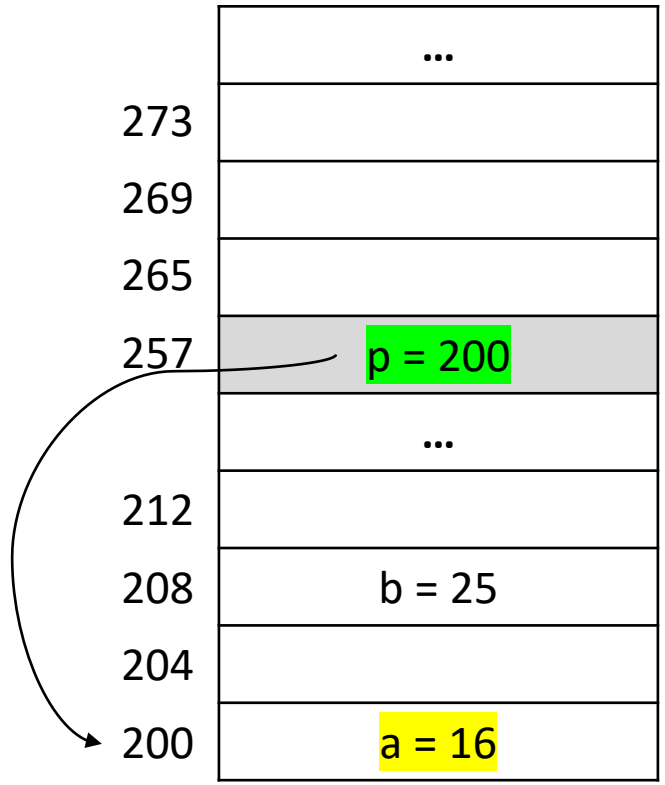
A pointer to variable ‘a’

```
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

}
```



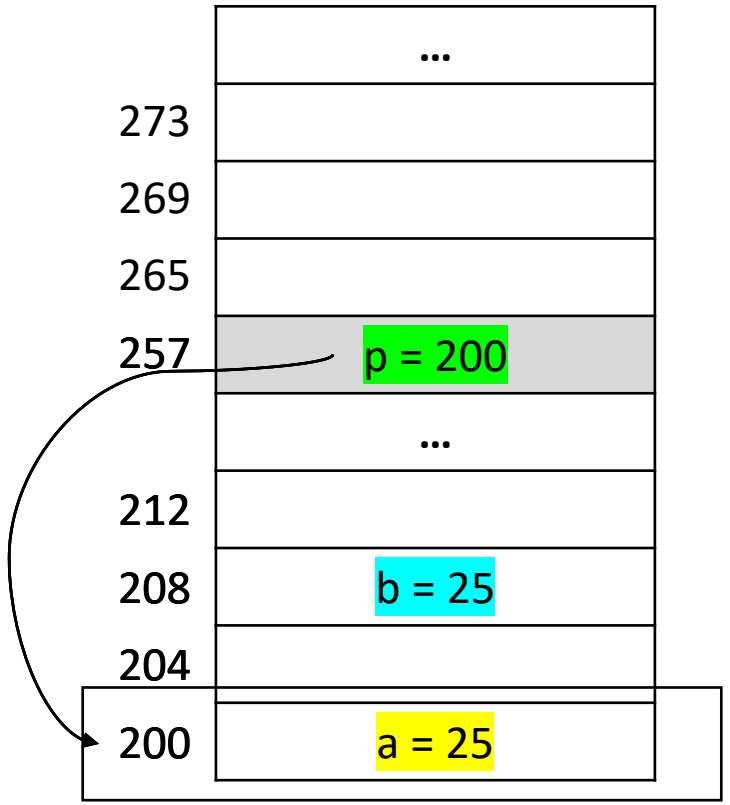
A pointer to variable ‘a’

```
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);
}
```



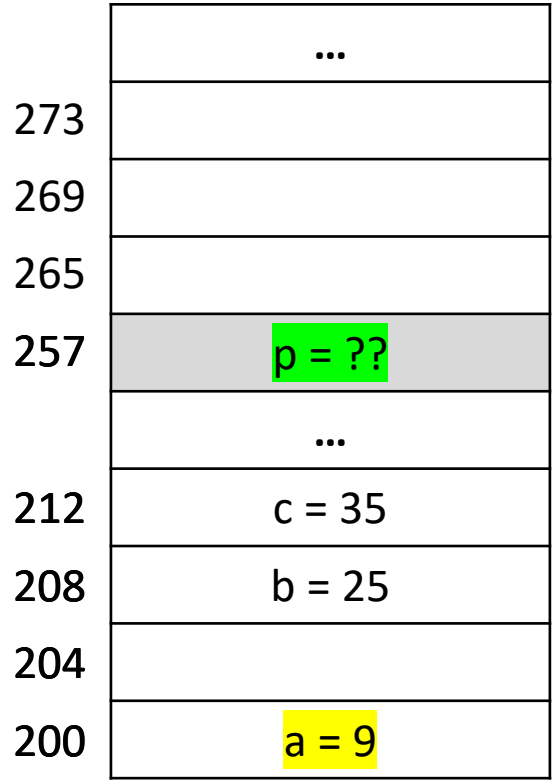
A pointer to variable ‘a’

```
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

}
```



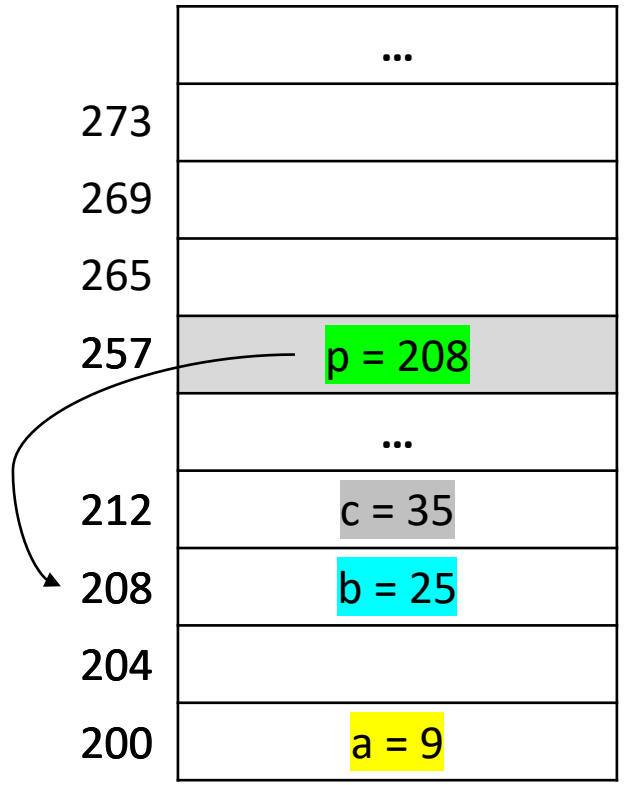
Pointer Arithmetic

```
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
}
```



Pointers and Functions

- Pointer parameters to functions
  1. Avoid the cost of copying a large object (since C is pass-by-value)
  2. Allows the function to change the contents of memory it could not access by name

- Primitive types and structs are passed by value, so a function only changes the copy sent to the function, not the original

- Since arrays in C are represented by the address of their first (0^{th}) element, all arrays can be changed (only the address is copied to the pointer parameter; the array is not cloned like a struct is)

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

What is the output of the following two code segments?

```c
#include <stdio.h>

struct example {
        int a;
        int b;
};

void add(struct example param) {
        param.a++;
}

int main() {
        struct example instance;
        instance.a = 3;
        add(instance);
        printf("%d\n", instance.a);
        return 0;
}
```

```c
#include <stdio.h>

struct example {
        int a;
        int b;
};

void add(struct example *param) {
        param->a++;
}

int main() {
        struct example instance;
        instance.a = 3;
        add(&instance);
        printf("%d\n", instance.a);
        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]));
}
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)**

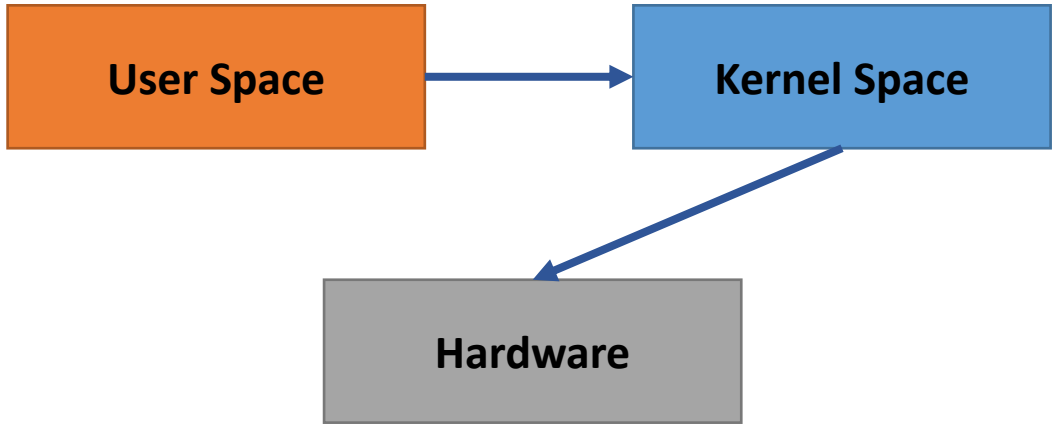
Base address of array A

Scope vs Lifetime

	Scope	Lifetime
Local (automatic) variables	From point declared to end of block	Function call (stored on stack in activation record)
Global (File scope) variables	File declared in (unless externed into other files)	Program (stored in .data segment)
Static local variable	From point declared to end of block	Program (stored in .data segment)
Static global variable	File declared in (cannot be externed into other files)	Program (stored in .data segment)
Malloc'ed space	Scopes of variables (pointers) that point to it. If they go out of scope, their lifetime is over. If we have no pointers to something, it cannot be free()ed, leading to a memory leak	From call of malloc() to call of free()

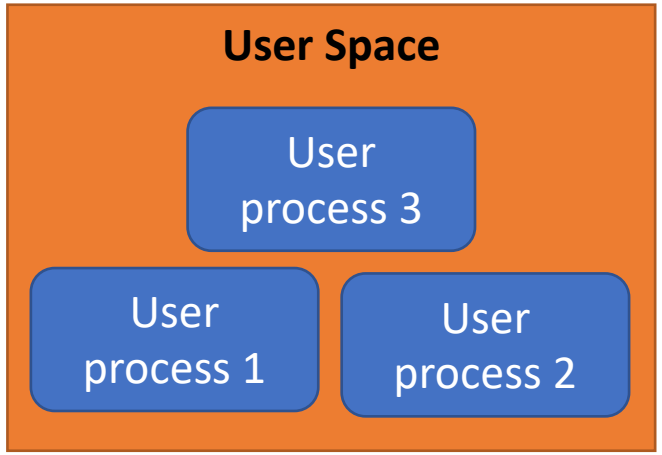
CS 1550 – Kernel Space vs User Space

- User space
  - **Less privileged memory space** where user processes execute
- Kernel space
  - **Privileged memory space** where the OS main process resides
  - **No User application should be able to change**



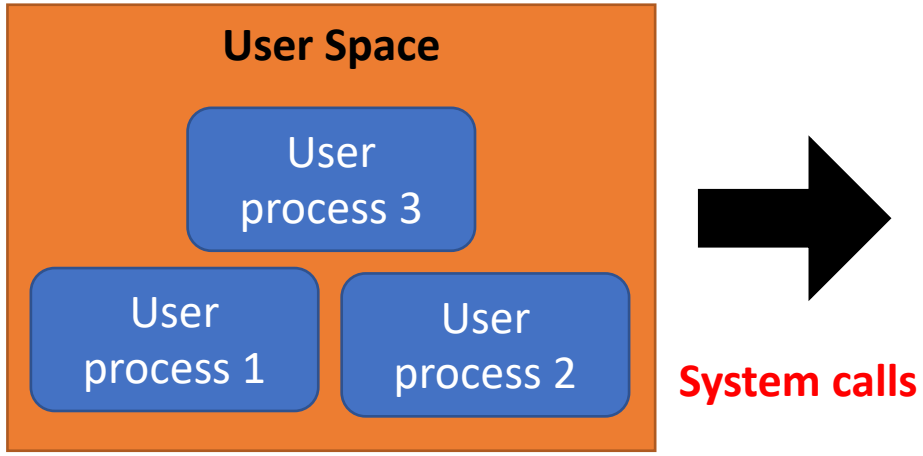
CS 1550 – Kernel Space vs User Space

- **System Call**
  - User processes must execute system calls to access OS resources and hardware



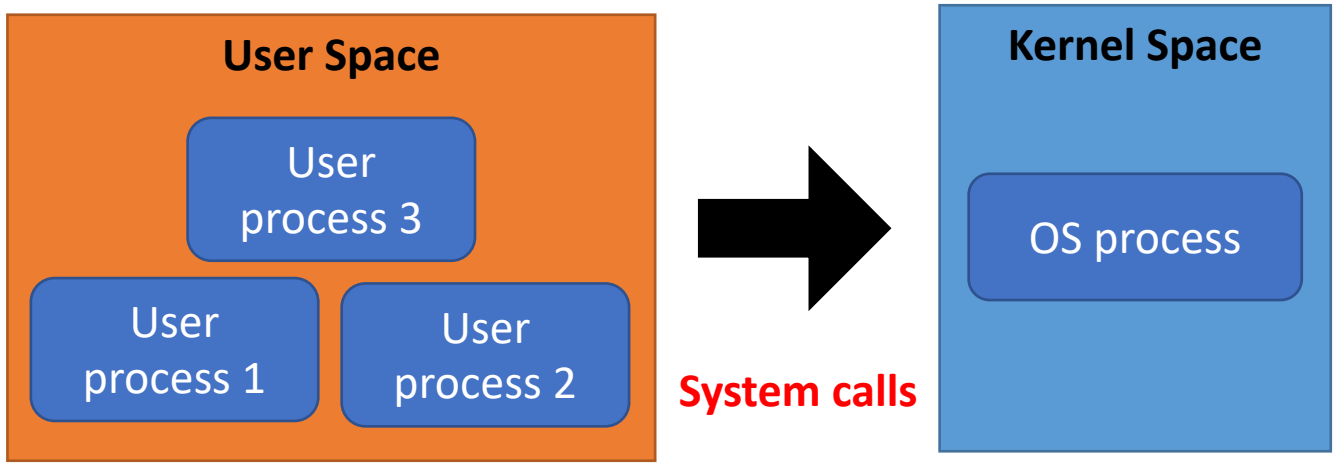
CS 1550 – Kernel Space vs User Space

- **System Call**
  - User processes must execute system calls to access OS resources and hardware



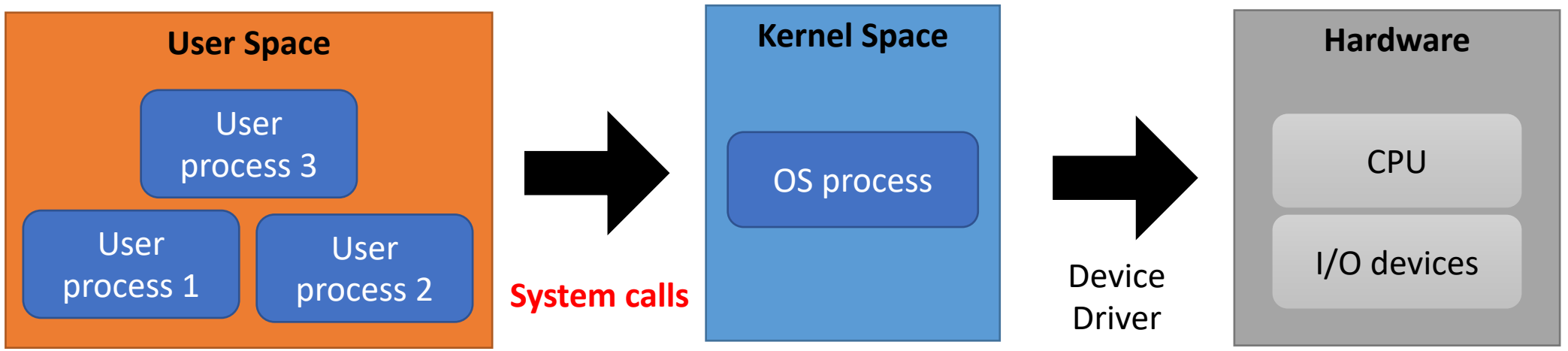
CS 1550 – Kernel Space vs User Space

- **System Call**
  - User processes must execute system calls to access OS resources and hardware



CS 1550 – Kernel Space vs User Space

- **System Call (OS function)**
  - User processes must execute system calls to access OS resources and hardware



Linux Kernel Syscalls

- **Kernel -** The core process of the operating system
- **System Call -** An operation (function) that an OS provides for running applications to use

Syscall	Purpose
exit	Causes a process to terminate
fork	Creates a new process, identical to the current one
read	Reads data from a file or device
write	Writes data to a file or device
open	Opens a file
close	Closes a file
creat	Creates a file

Using Syscalls

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
        int fd;
        char buffer[100];
        strcpy(buffer, "Hello,World!\n");
        fd = open("hello.txt",O_WRONLY|O_CREAT);
        write(fd, buffer, strlen(buffer));
        close(fd);
        exit(0);
        return 0;
}
```

OR-ing Flags

- Define constants as powers of 2
- Bitwise OR to combine one or more flags
- Bitwise AND to test whether a particular flag is set

```
#define O_RDONLY         0
#define O_WRONLY         1
#define O_RDWR           2
#define O_CREAT         16
```

File Descriptors

- Integer identifying a unique open file
  - Similar to FILE *

- OS maintains additional information about the file to do things such as clean up on process termination

- Three standard file descriptors opened automatically:

0 – stdin

1 – stdout

2 – stderr

Header Files

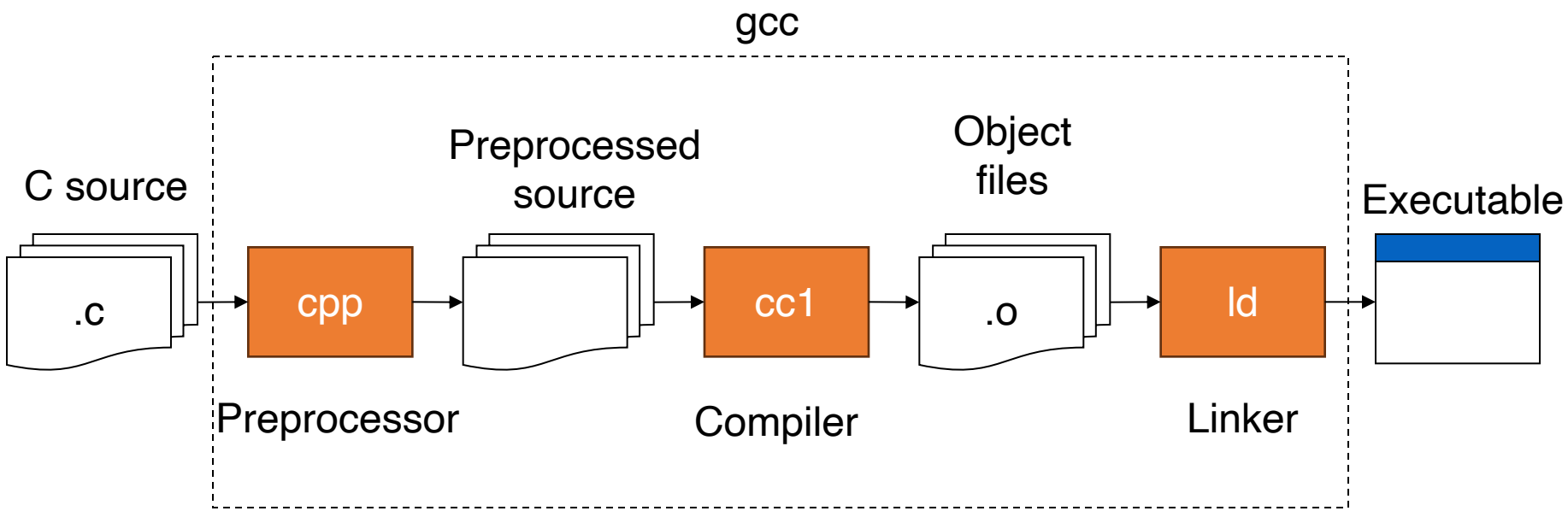
- Usually only contain declarations
  - Variables
  - Functions
  - #defined macros

- Paired with an implementation file

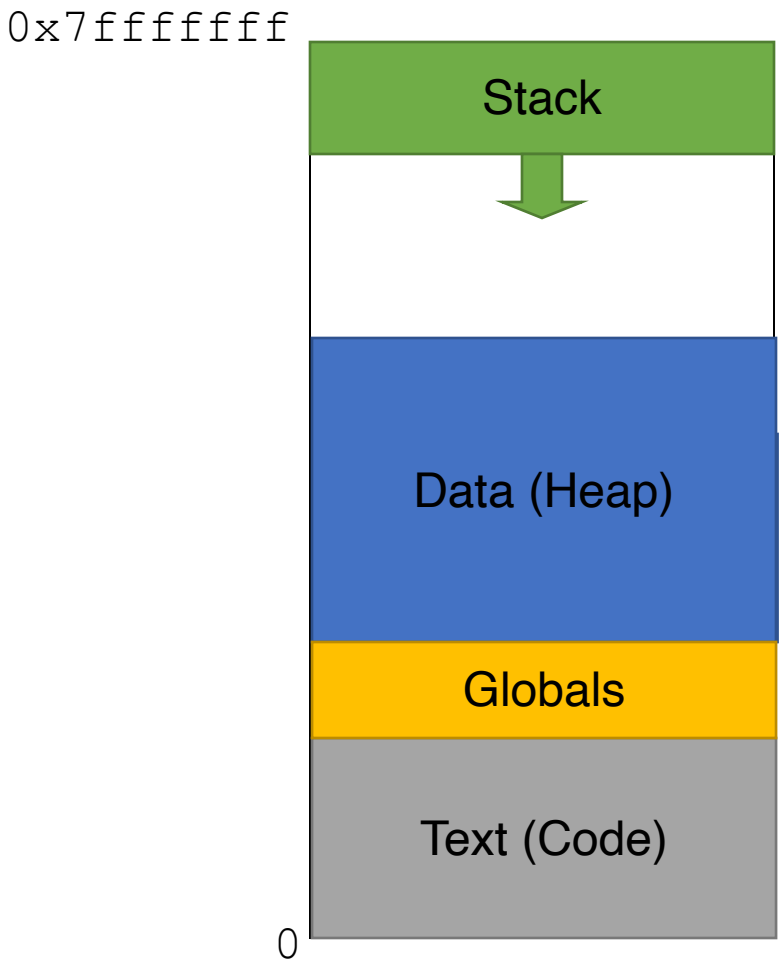
Including a Header File Once

```
#ifndef _MYHEADER_H_
#define _MYHEADER_H_

…Definitions of header to only be included once


#endif
```

C Compilation



Process's Address Space



Linux Address Space