Pointers and arrays

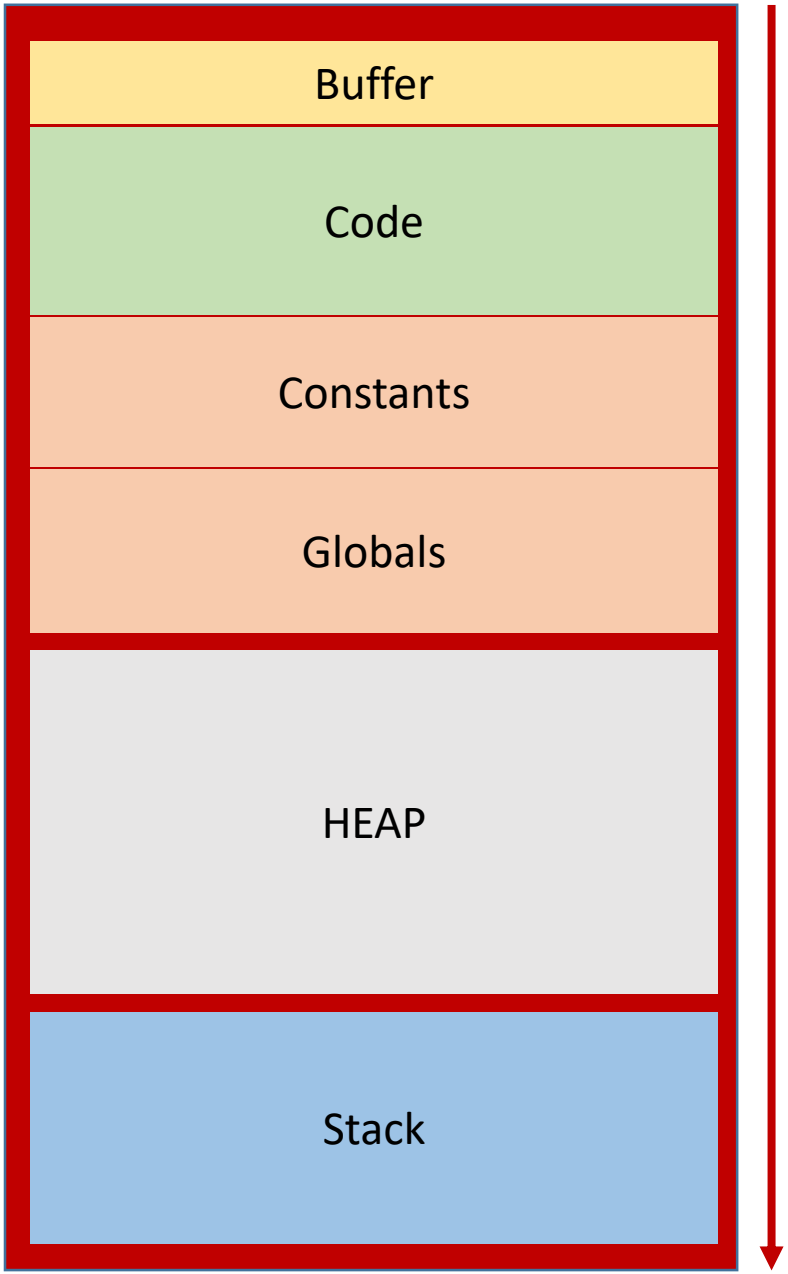
Lecture 03.01

Pointers

- Pointer is an address of a piece of data in memory
- Why pointers?
  - Avoid copies
  - Share data

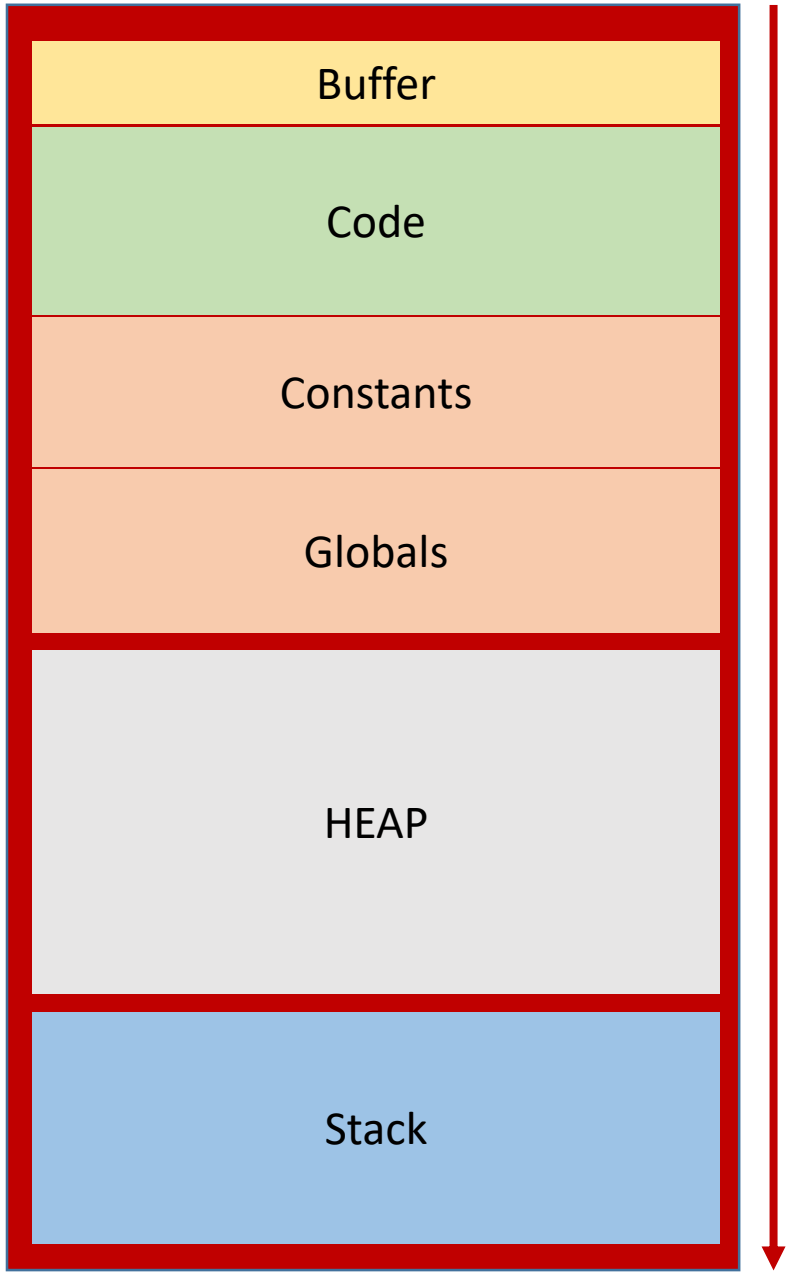
Memory addresses

- Memory is laid out in sequential order. Each position in memory has a number (called its address).
- The compiler associates your variable names with memory addresses
- In C, you can actually ask the computer for the address of a variable in memory. This is done using the ampersand &



Memory sections

- If you declare a variable inside function, it will have an address in the Stack area
- If you declare a variable outside the function, it will have an address in Globals section



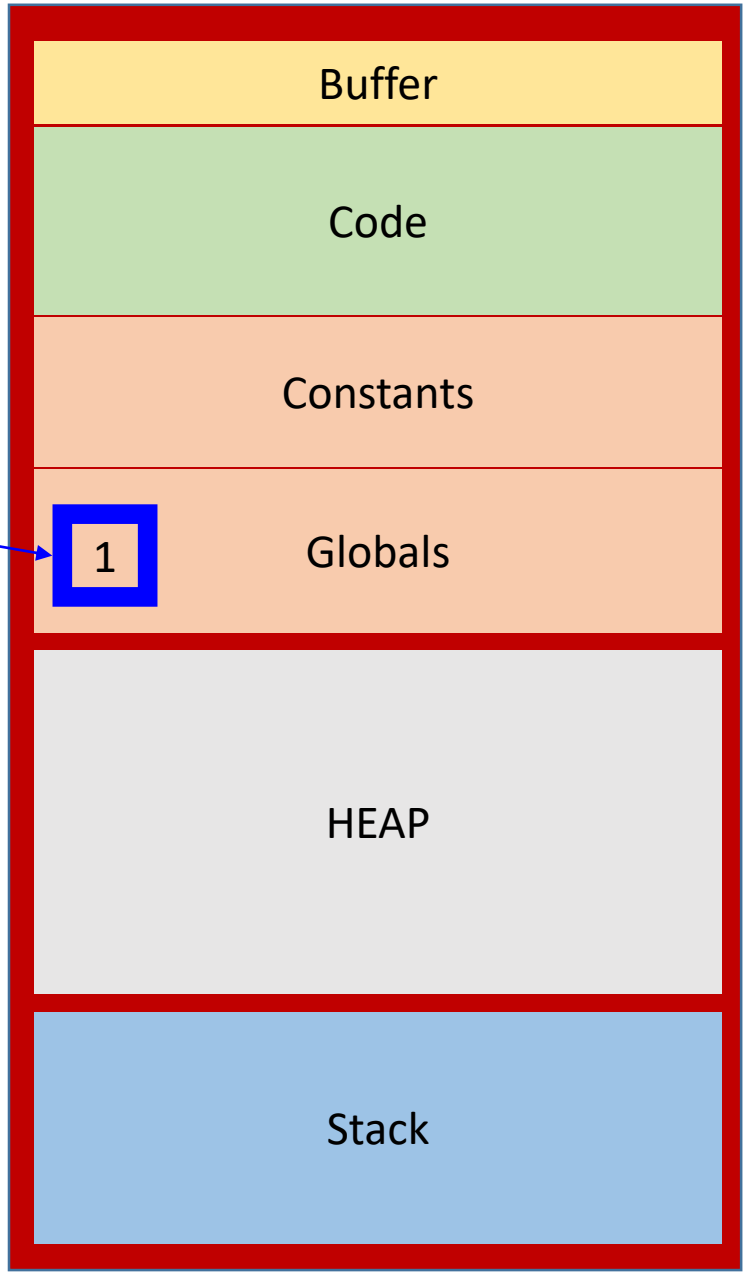
Memory diagram of a single process

Where *y* lives?

```
int y=1;

int main () {
 int x=4;
 printf ("y lives at address %p\n", &y);
 return 0;
}
```

Prints something like 0xF4240 –
which corresponds to address 1,000,0000



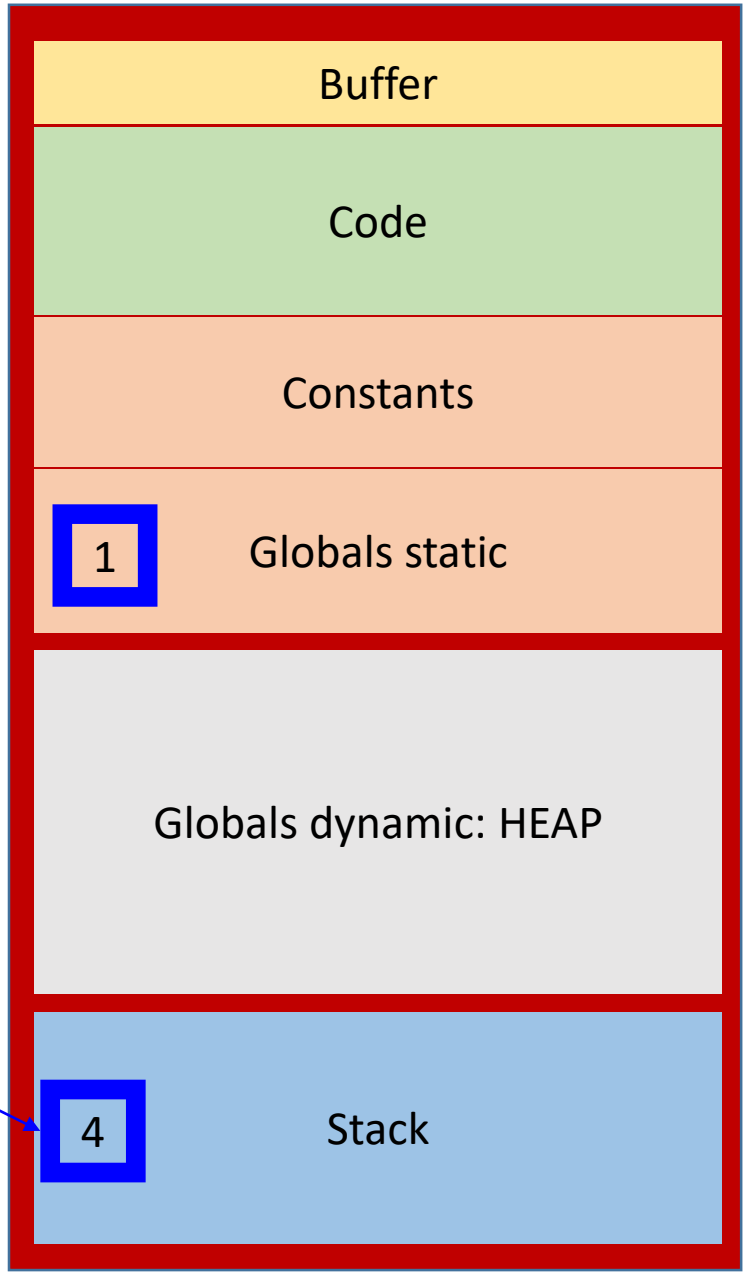
Memory diagram of a single process

Where *x* lives?

```
int y=1;

int main () {
 int x=4;
 printf ("x lives at address %p\n", &x);
 return 0;
}
```

Prints something like 0x3E8FA0 –
which corresponds to address 4,100,0000



Memory diagram of a single process

3 things to remember

```
int x = 9;
```

1. Get the address of x and store it in a variable:

```
int * addr_x = &x;   // addr_x now stores some long number – say 4,200,000
```

2. Given an address – read value stored at this address:

```
int val = *addr_x; // val is now equal ?
```

3. Write a new value at a given address:

```
*addr_x = 99; //x is now equal ?, val is equal ?
```

Pointer is just a variable that stores an address

```
int * ip;
long * lp;
double *dp;
```

- **sizeof(ip) = sizeof(lp) =sizeof (dp)**

- Each variable stores an address (unsigned long on 64-bit systems)
- The address is stored in a variable and the variable itself has an address:

&ip

Examples of using pointers in C

C:

Incrementing int by calling *increment*

```
void increment (int a) {
    a++;
}

int main () {
    int a = 5;
    increment (a);
    printf ("%d\n", a);

    return 0;
}
```

Passing by value – the copy of *a* is created and processed

Prints ?

C:

Incrementing int by passing an *address*

```
void increment (int *p) {
    (*p)++;
}

int main () {
    int a = 5;
    increment (&a);
    printf ("%d\n", a);

    return 0;
}
```

Copy of address of a is created, but the copy points to the same location in memory

Prints ?

Java: no way of incrementing *int* by calling *increment*

```
static void increment (int p) {
        p++;
}

public static void  main (String [] args) {
    int a = 5;
    increment (a);
    System.out.println (a);
}
```

Java solves this problem with objects

```
static void increment (MyInt a){
    a.value ++;
}
```

```
class MyInt {
    public int value;
}
```

```
public static void main (String [] args) {
    MyInt b = new MyInt();
    b.value = 5;
    increment (b);
    System.out.println(b.value);
}
```

Passes reference to an object

Working on exercises 1, 2, 3

Arrays are just like pointers

- The compiler associates the address of the first byte with variable *drinks*
- You can read elements of an array with subscripts or with pointer arithmetic:

```
int drinks[] = {4, 2, 3};
printf("1st order: %i drinks\n", drinks[0]);
printf("1st order: %i drinks\n", *drinks);

printf("3rd order: %i drinks\n", drinks[2]);
printf("3rd order: %i drinks\n", *(drinks + 2));
```

The same

The same

Why arrays really start with 0

```
int drinks[] = {4, 2, 3};
printf("1st order: %i drinks\n", drinks[0]);
printf("1st order: %i drinks\n", *drinks);

printf("3rd order: %i drinks\n", drinks[2]);
printf("3rd order: %i drinks\n", *(drinks + 2));
```

- The index is just the number that's added to the pointer to find the location of the element.

Arrays and pointers are interchangeable as function parameters

```
int func1 ( int [ ] numbers) {
    return *(numbers + 3);
}
int func2 ( int * numbers) {
    return *(numbers + 3);
}
int main () {
    int numbers = {1,2,3,4,5};
    int forth = func1(numbers);
    Int another_forth = func2(numbers);
}
```

Honey, who shrunk the numbers?

```
void func1 ( int [ ] numbers) {
    printf (“size of array is %ld\n”, sizeof (numbers));
}
```

Prints 4 or 8

```
int main () {
    int numbers = {1,2,3,4,5};
    printf (“size of array is %ld\n”, sizeof (numbers));
```

Prints 20

```
    func1(numbers);
}
```

Array variables are not quite pointer variables: 1

- sizeof(an array) is...the size of an array – the total number of bytes allocated for an array

- When array is passed as a parameter to the function, the function receives only array name – which is an address of the first byte of the array

- Thus the sizeof inside the function becomes the size of the memory address (4 bytes on 32-bit, and 8 bytes on 64-bit machines)

- This is called ***pointer decay***

Array variables are not quite pointer variables: 2

int numbers = {1,2,3,4,5};

int * p_numbers = numbers;

- Pointer variable stores a value of address, but it is another variable, which has its own address:

&p_numbers≠ p_numbers

- Array variable stores the address of the first byte of the array. The computer will allocate space to store the array, but it won't allocate *any* memory to store the array variable. The compiler simply plugs in the address of the start of the array.

&numbers = numbers

Array variables are not quite pointer variables: 3

```
int numbers = {1,2,3,4,5};
int * p_numbers = numbers;
```

- Because **array variables** don't have allocated storage, it means you **can't point** them **at anything else**.

```
int numbers2 = {1,2,3,4,5};
int * pp_numbers = numbers2;
pp_numbers = numbers1;
numbers  = numbers2;
numbers = pp_numbers;
```

Illegal !

Summary

- Array variables are different from pointer variables because:
  - They cannot point to anything else
  - The address of an array variable is not stored in another variable, but array variable is substituted by the address of the first byte
  - Passing an array variable to the function decays it to the pointer