CSCB09: Software Tools and Systems Programming

Bianca Schroeder
bianca@cs.toronto.edu
IC 460

Does this change age?

```
#include <stdio.h>

void lie(int age) {
    printf("You are %d years old\n", age);
    age += 1;
    printf("You are %d years old\n", age);
}

int main() {
    int age = 18;
    lie(age);
    printf("But your age is still %d\n", age);

    return 0;
}
```

Fixing `lie()`

```
#include <stdio.h>

void lie(int *age) {
    printf("You are %d years old\n", *age);
    *age += 1;
    printf("You are %d years old\n", *age);
}

int main() {
    int age = 18;
    lie(&age);
    printf("And your age is still %d\n", age);

    return 0;
}
```

Passing an array to `lie()`

```
#include <stdio.h>

void lie(int *age, int size) {
// works also: void lie(int age[], int size) {

    for (int i= 0; i<size; i++)
         age[i] += 1;

}

int main() {
    int age[5] = {1,2,3,4,5};
    lie(age, 5);

    for (int i = 0; i<5; i++)
        printf("Age is %d\n", age[i]);

    return 0;
}
```

Will the values in the array change?

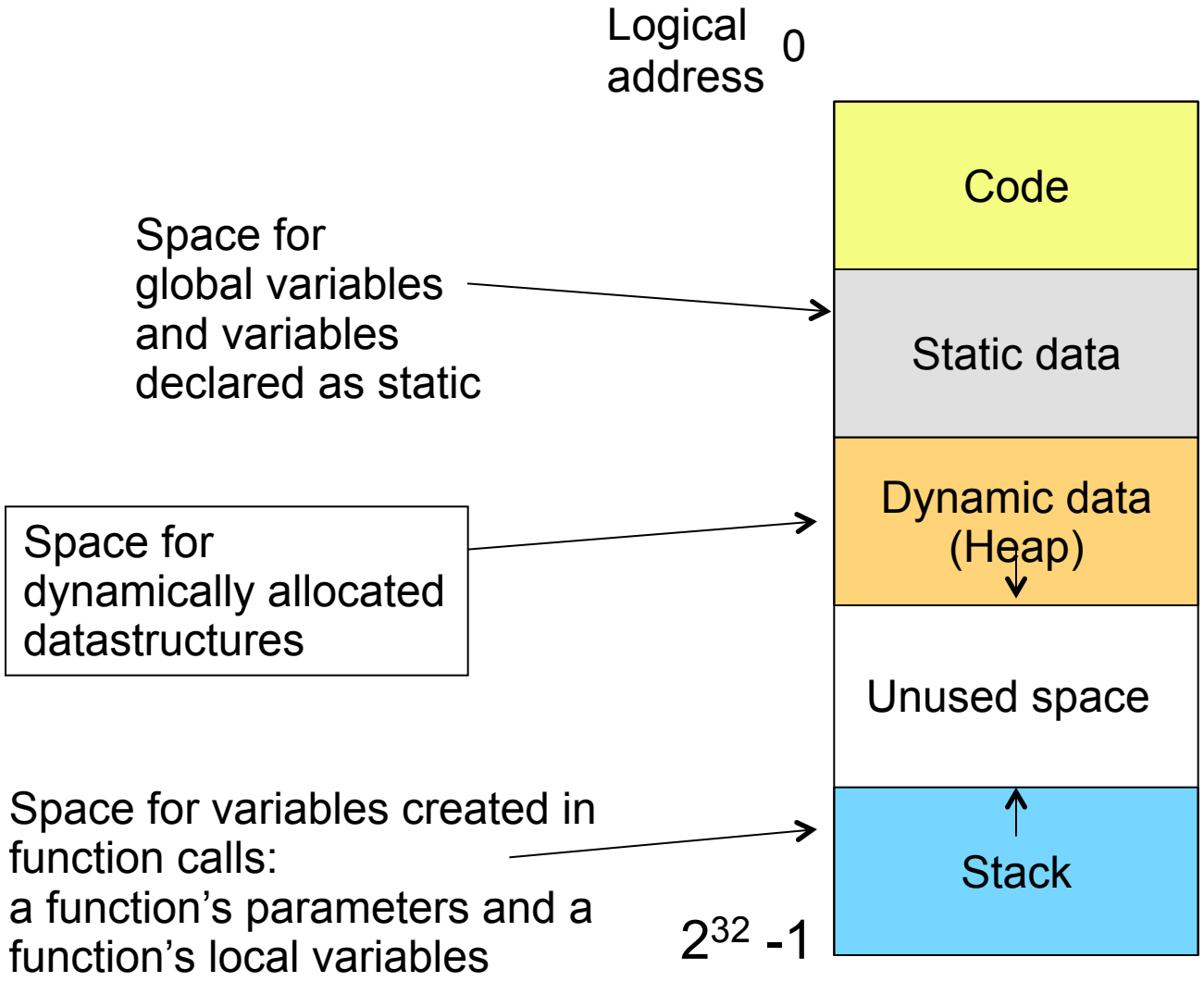
What do you think about C so far …?

- Disappointed about the lack of support for strings, arrays, etc…. ?
- Turns out you also have to do much of memory management manually ….

Your ride with Python:
- Cruise control
- Seat heating
- Cup holders
- Automatic transmission

Your ride with C:
- Performance and speed
- But, no amenities …
- And it only comes with a stickshift …

The address space continued ..



Dynamic memory management – why?

- When declaring an array we have to specify its size.
  - E.g. `int my_array[100];`

- What if we don't know the size of an array in advance?
- What if we realize later that we need a bigger array?

Dynamic memory management – why?

•Imagine we want to write a function `concat` that takes two strings, and returns their concatenation as a new string.

•We would use it as follows:

```
char *s;
s = concat (“abc”, “def”);
```

Dynamic memory management – why?

•Here is an attempt at writing `concat`:

```
char *concat(const char *s1, const char *s2) {
     char result[70];
     strcpy (result, s1);
     strcat (result, s2);
     return result;
}
```

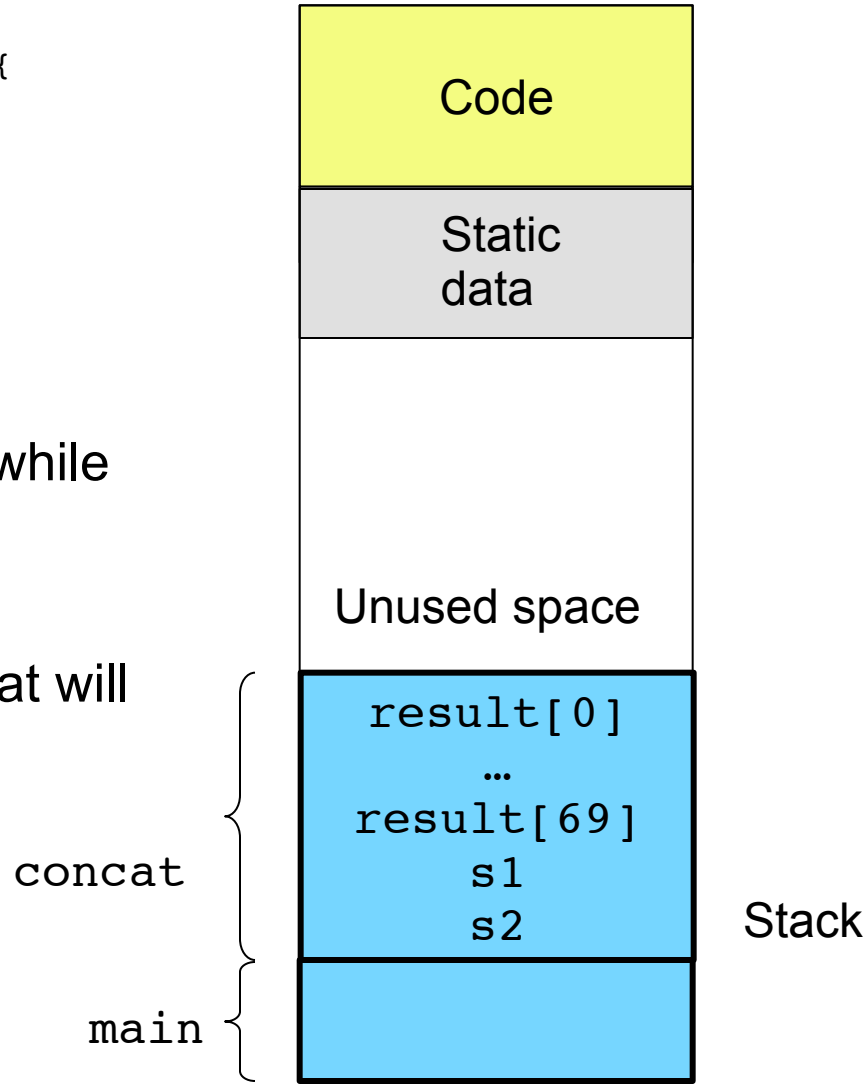
Any problems with this implementation?

- `strncpy & strncat` would be safer..
- But what else?

Recall memory management from last week

```
char *concat(const char *s1, const char *s2) {
     char result[70];
     strcpy (result, s1);
     strcat (result, s2);
     return result;
}
```

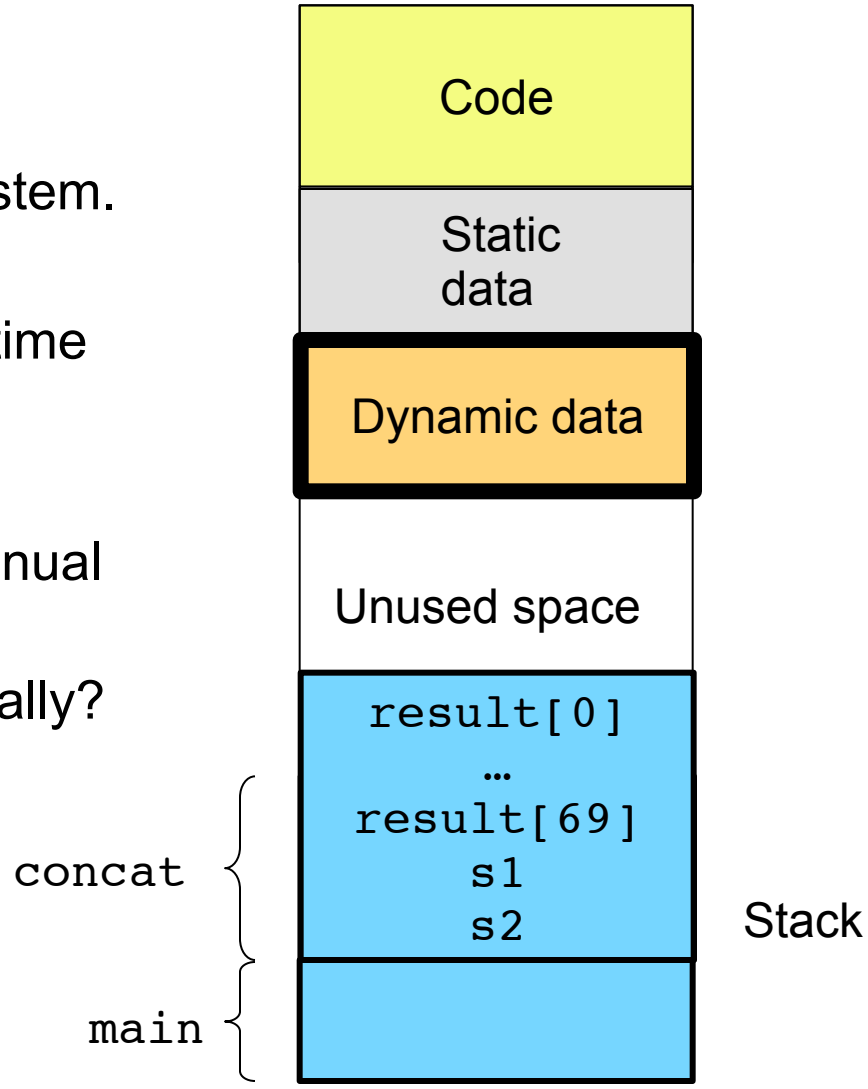
- The memory for `results` exists only while `concat` is running.
- Need to be able to allocate memory that will not be lost when concat finishes.
- How?



Remember the dynamic data segment in the address space?

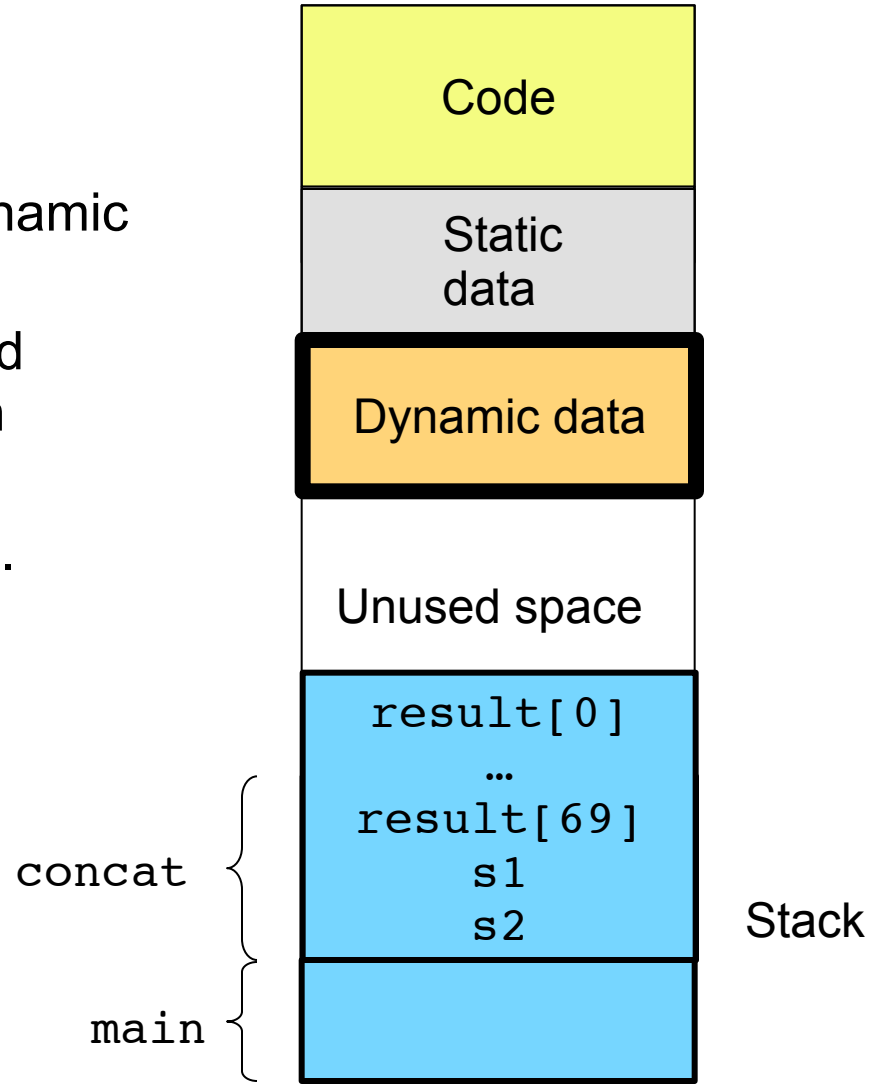
- Memory allocated here will never be released /freed automatically by the system.
  - Completely under programmers control.
- Memory here can be allocated at any time during the run of a program.

- Dynamic memory allocation allows manual control of memory allocation!
- How do you allocate memory dynamically?



Malloc()

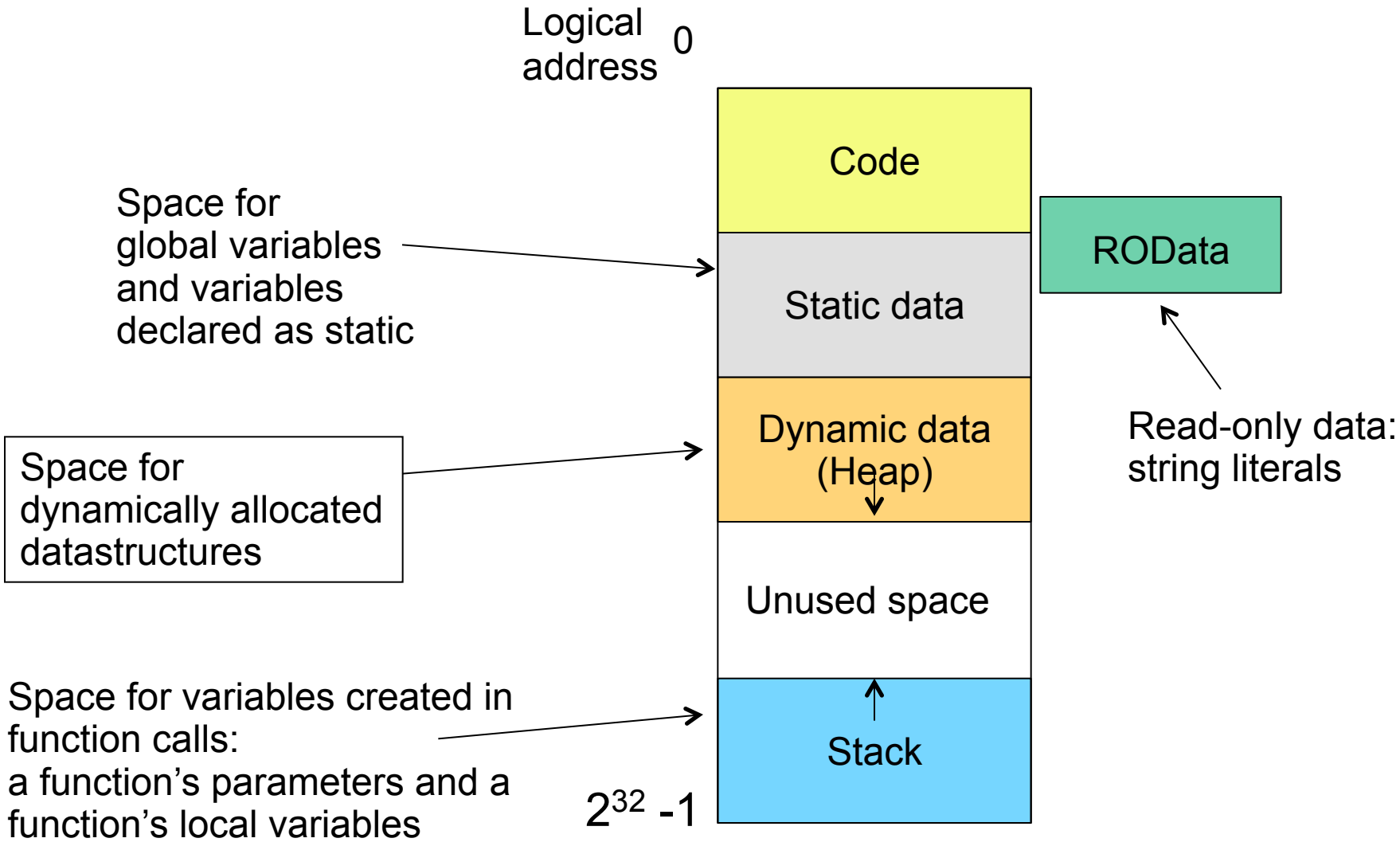
- void *malloc (`size_t size`);
- Malloc allocates `size` bytes in the Dynamic Data segment
- Returns a pointer to the newly acquired memory, or NULL if there is not enough available memory.
- The allocated memory is **uninitialized**.



A correct `concat`

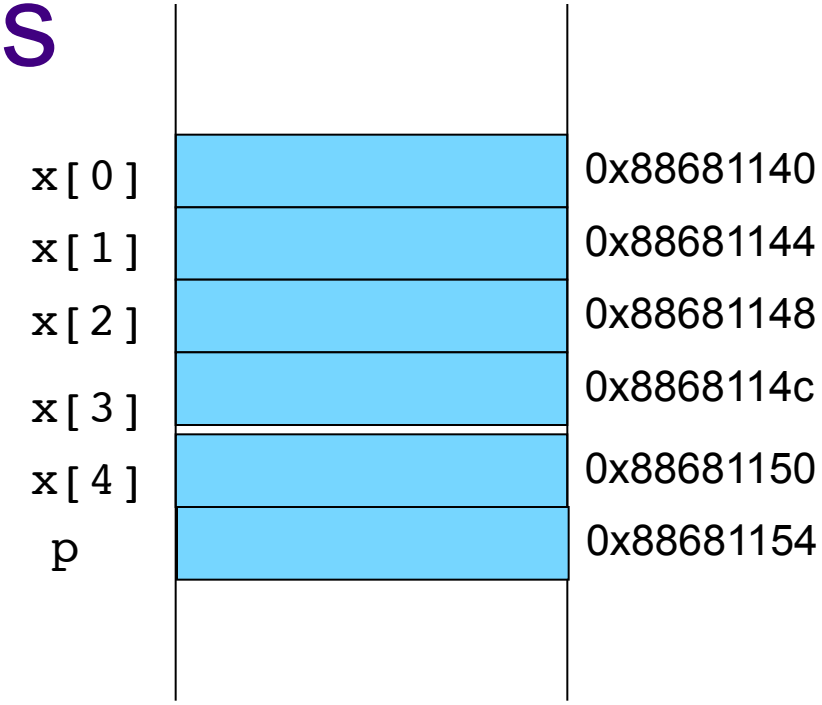
```
char *concat(const char *s1, const char *s2) {
     char *result;
     result = malloc(strlen(s1) + strlen(s2) + 1);
     if (result == NULL) {
         printf ("Error: malloc failed\n");
         exit(1);
     }
     strcpy (result, s1);
     strcat (result, s2);
     return result;
}
```

The address space continued ..



Arrays

```
int *p;
int x[5];
for (i = 0; i < 5; i++) {
    x[i] = i;
}
```

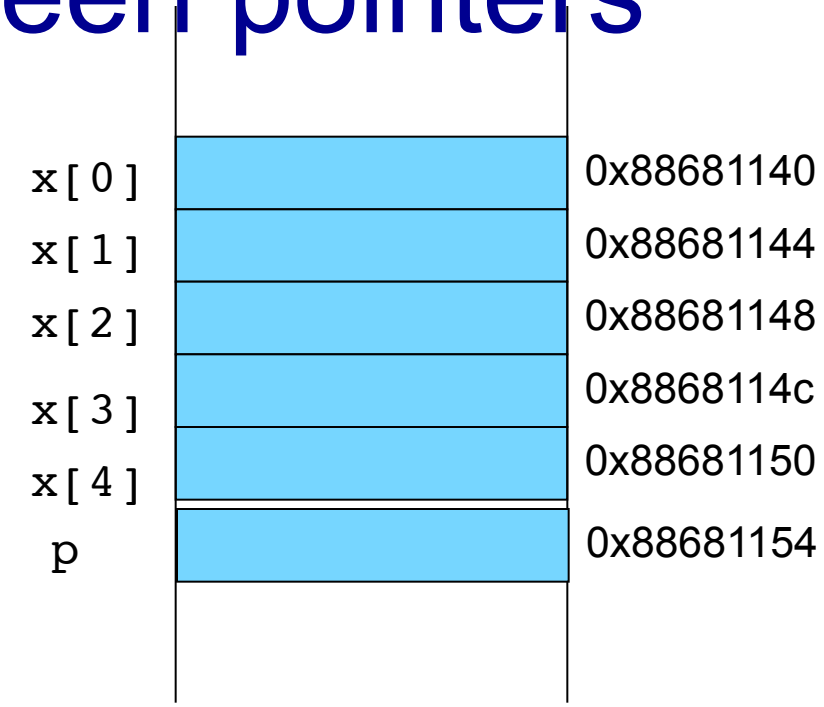


- Remember from last time:
  - In many ways an array name can be treated like a pointer
  - E.g.: `*x == x[0]`
  - E.g.: `void x(int *a)` is same as `void x(int a[])`

- So are there differences between arrays and pointers?

Differences between pointers and arrays

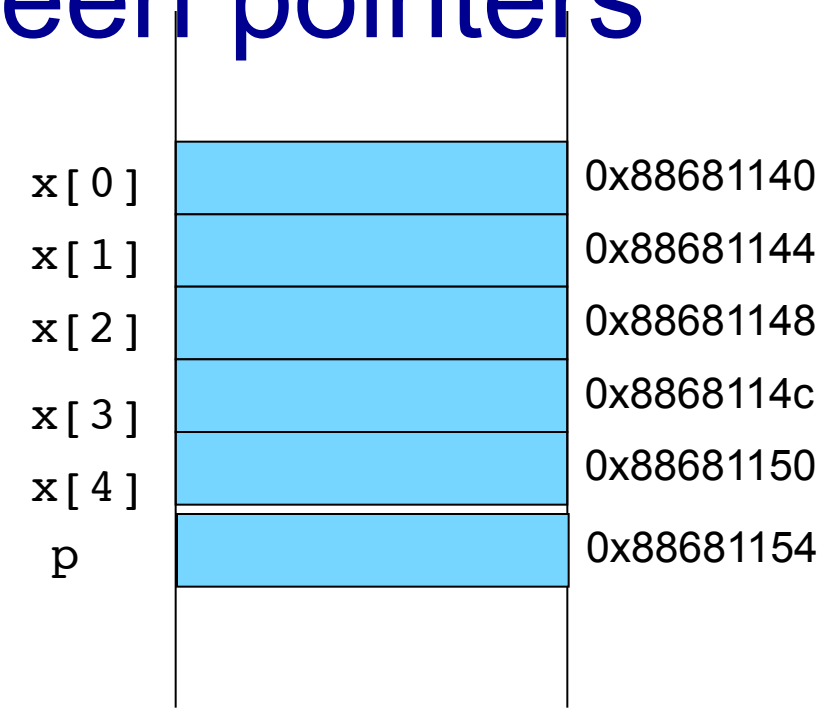
```
int *p;
int x[5];
for (i = 0; i < 5; i++) {
    x[i] = i;
}
```



- For a pointer variable (e.g. p in the example) space is reserved to store an address.
- For an array, space is reserved to store array contents, but not the array address.

Differences between pointers and arrays (I)

```
int *p;
int x[5];
for (i = 0; i < 5; i++) {
    x[i] = i;
}
```



- You can do

```
p = 0x88681144;
```

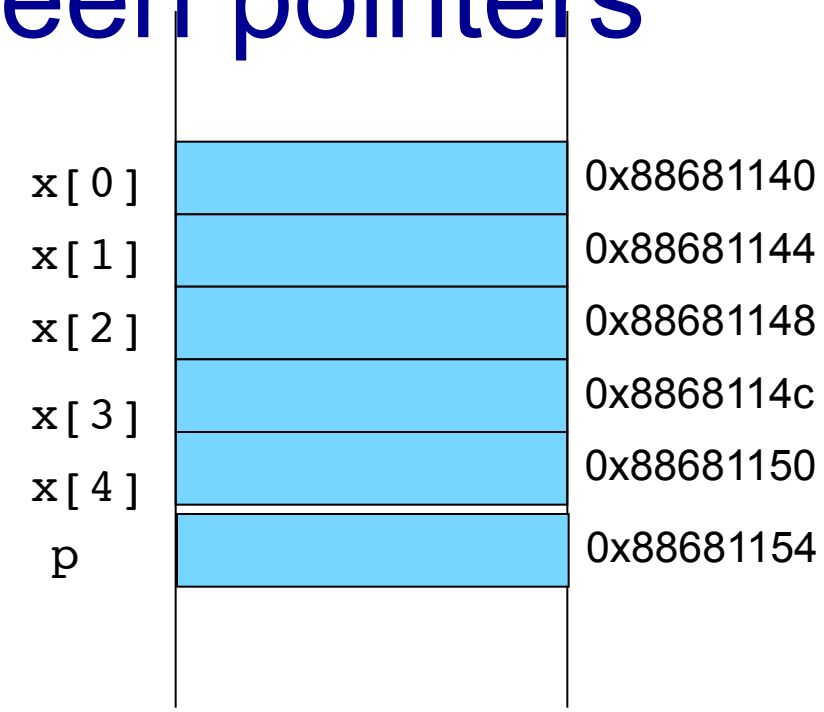
- But the following is illegal:

```
x = 0x88681144;
```

- There is no space allocated for x to hold an address

Differences between pointers and arrays (II)

```
int *p;
int x[5];
for (i = 0; i < 5; i++) {
    x[i] = i;
}
```



- Inside `main:` `sizeof(p)` `!=` `sizeof(x)`

Size of an address (4 or 8 bytes)

5 * size of an int (typically 20 bytes)

- BUT, inside `void func(int x[]):`

  `sizeof(x) == size of an address`

Differences between char pointers and char arrays (I)

```
char a[] = “array”;
char *p = “pointer”;
```

- As for all arrays, space is reserved to store contents of a, but not the array address.

```
a = p;    // Illegal!
p = a;    // OK!
```

- Can assign new value to `p`, but not to `a`.

Differences between char pointers and char arrays (II)

```
char a[] = “array”;
char *p = “pointer”;
```

- `“pointer”` is stored in read-only memory. No other space is reserved for `p`, except to store a memory address.

```
p[0] = ‘z’;    // Illegal!
a[0] = ‘z’;    // OK!
```

With all this in mind try Q1..