

# CS 33

## Introduction to C Part 2

# Methods



- **C has functions**
- **Java has methods**
  - methods implicitly refer to objects
  - C doesn't have objects
- **Don't use the “M” word**
  - it's just wrong

# Swapping

Write a function to swap two ints

```
void swap(int i, int j) {
```



Arguments are  
passed by value

```
}
```

```
int main() {  
    int a = 4;  
    int b = 8;  
    swap(a, b);  
    printf("a:%d b:%d", a, b);  
}
```

# Swapping

Write a function to swap two ints

```
void swap(int i, int j) {  
    int tmp;  
    tmp = j; j = i; i = tmp;  
}  
  
int main() {  
    int a = 4;  
    int b = 8;  
    swap(a, b);  
    printf("a:%d b:%d", a, b);  
}
```



**Darn!**

```
$ ./a.out  
a:4 b:8
```

# Why “pass by value”?

- Fortran, for example, passes arguments “by reference”
- Early implementations had the following problem (shown with C syntax):

```
int main() {  
    function(2);  
    printf("%d\n", 2);  
}  
void function(int x) {  
    x = 3;  
}
```

```
$ ./a.out  
3
```

# Variables and Memory

What does

```
int x;
```

do?

- It tells the compiler:  
I want *x* to be the name of an area of memory that's big enough to hold an *int*.

## What's memory?

---

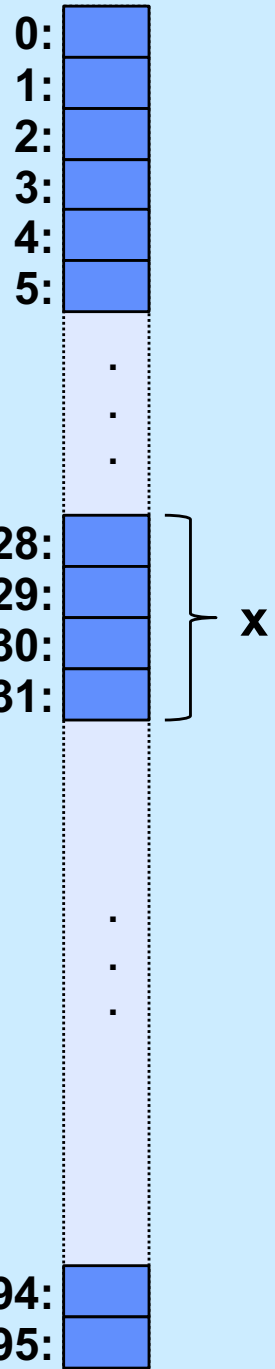
# Memory

- **“Real” memory**
  - it’s complicated
  - it involves electronics, semiconductors, physics, etc.
  - it’s not terribly relevant at this point
- **“Virtual” memory**
  - the notion of memory as used by programs
  - it involves logical concepts
  - it’s how you should think about memory (most of the time)

# Virtual Memory

- It's a large array of bytes
  - one byte is eight bits
  - an int is four consecutive bytes
  - so is a float
  - a char is one byte
- The array index of a byte is its *address*
  - the address of a larger item is the index of its first byte

virtual  
memory





# Variables

- **Where**
  - they refer to locations in memory
- **Size**
  - how much memory they refer to
- **Interpretation**
  - how to interpret the contents of memory
- **All determined when they are declared**
- **None of the above change after declaration**

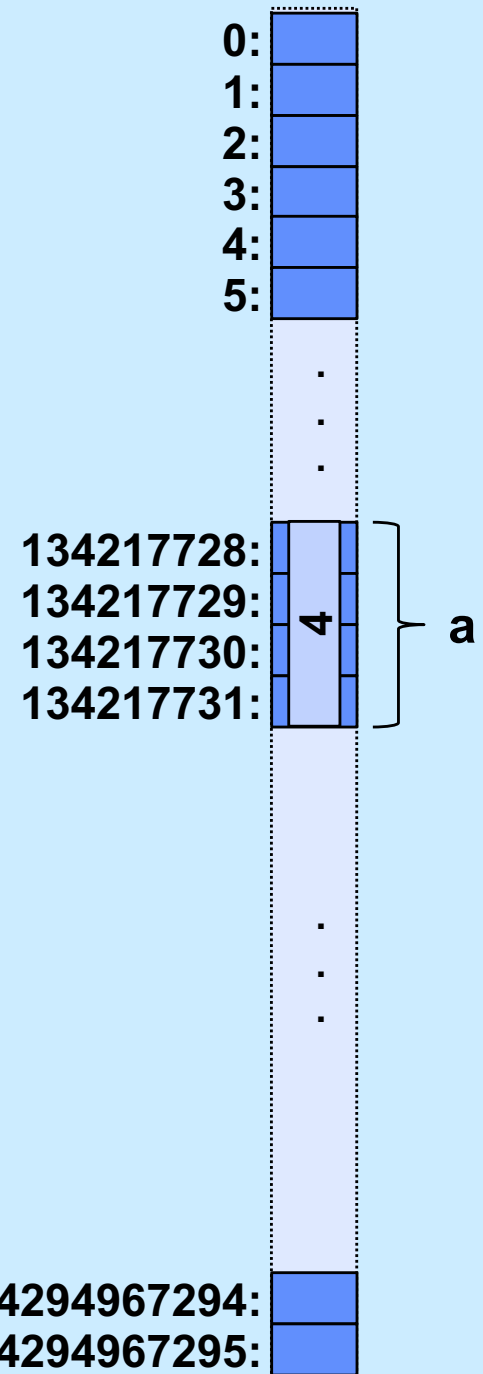
```
int x;    // sizeof(x) == 4
float y;  // sizeof(y) == 4
char z;   // sizeof(z) == 1
```

# Memory addresses in C

- In C
  - you can get the memory address of any variable
  - just use the operator &

```
int main() {  
    int a = 4;  
    printf("%p\n", &a);  
}
```

```
$ ./a.out  
134217728
```



# C Pointers

- **What is a C pointer?**
  - a variable that holds an address
- **Pointers in C are “typed” (remember the promises)**
  - pointer to an int
  - pointer to a char
  - pointer to a float
  - pointer to <whatever you can define>
- **C has a syntax to declare pointer types**
  - things start to get complicated ...

# C Pointers

p is a pointer to an int

if you follow p, you find an int

```
int main() {  
    int *p;  
    int a = 4;  
    p = &a;  
    printf("%p\n", p);  
}
```

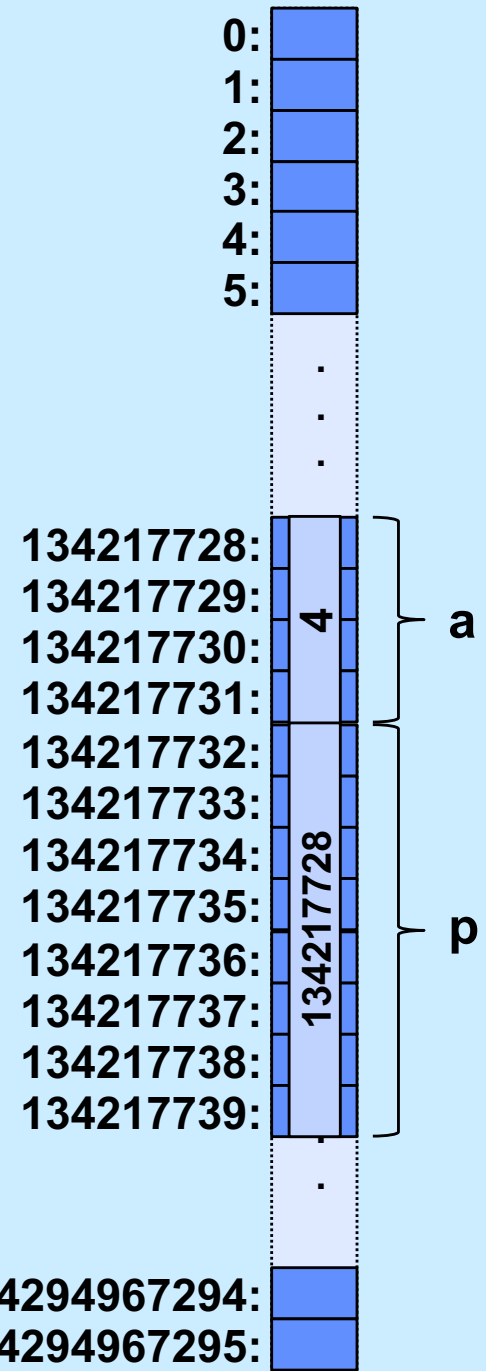
p is assigned the address of a

```
$ ./a.out  
134217728
```

# C Pointers

```
int main() {  
    int *p;  
    int a = 4;  
    p = &a;  
    printf("%p\n", p);  
}
```

```
$ ./a.out  
134217728
```



# C Pointers

- **Pointers are typed**
  - the types of the items they point to are known
  - there is one exception (discussed later)
- **Pointers are first-class citizens**
  - they can be passed to functions
  - they can be stored in arrays and other data structures
  - they can be returned by functions
- **Pointers have the properties of all variables**

```
sizeof(int *) == sizeof(char *) == 8
```

# Swapping

What does this do?

```
void swap(int *i, int *j) {  
    int *tmp;  
    tmp = j; j = i; i = tmp;  
}  
  
int main() {  
    int a = 4;  
    int b = 8;  
    swap(&a, &b);  
    printf("a:%d b:%d\n", a, b);  
}
```



**Damn!**

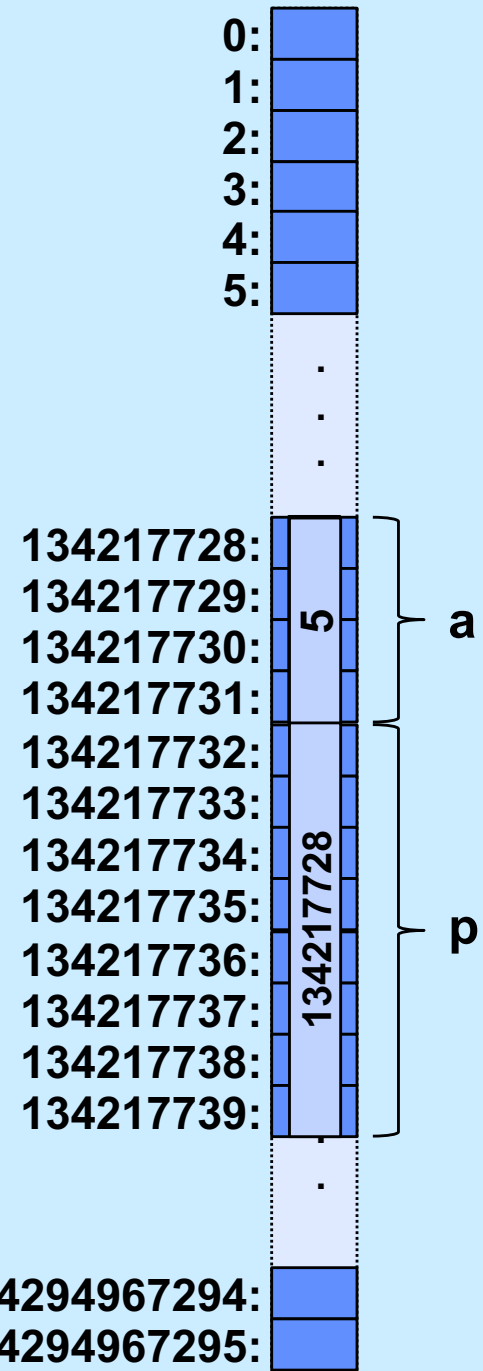
```
$ ./a.out  
a:4 b:8
```

# C Pointers

- **Dereferencing pointers**
  - accessing/modifying the value pointed to by a pointer

```
int main() {  
    int *p;  
    int a = 4;  
    p = &a;  
    printf("%d\n", *p);  
    *p = *p + 1;  
    printf("%d\n", *p);  
}
```

```
$ ./a.out  
4  
5
```





# Dereferencing C Pointers

```
int main() {  
    int *p;  
    int a = 4;  
    p = &a;  
    printf("%d\n", *p);  
    *p = *p + 1;  
    *p += 3;  
    printf("%d\n", a);  
}
```

```
$ ./a.out  
4  
8
```

# Swapping

```
void swap(int *i, int *j) {  
    int tmp;  
    tmp = *j; *j = *i; *i = tmp;  
}  
int main() {  
    int a = 4;  
    int b = 8;  
    swap(&a, &b);  
    printf("a:%d b:%d\n", a, b);  
}
```



**Hooray!**

```
$ ./a.out  
a:8 b:4
```

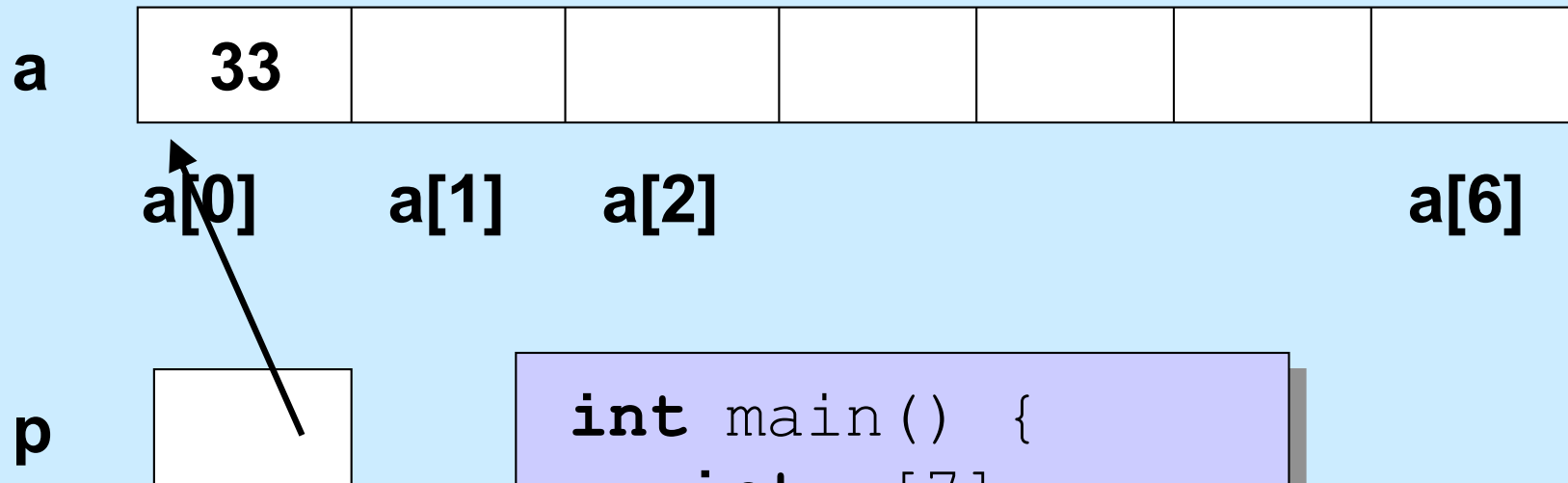
# Quiz 1

```
int doubleit(int *p) {  
    *p = 2*(*p);  
    return *p;  
}  
int main() {  
    int a = 4;  
    int b;  
    b = doubleit(&a);  
    printf("%d\n", a*b);  
}
```

**What's printed?**

- a) 8
- b) 16
- c) 32
- d) 64

# Pointers and Arrays

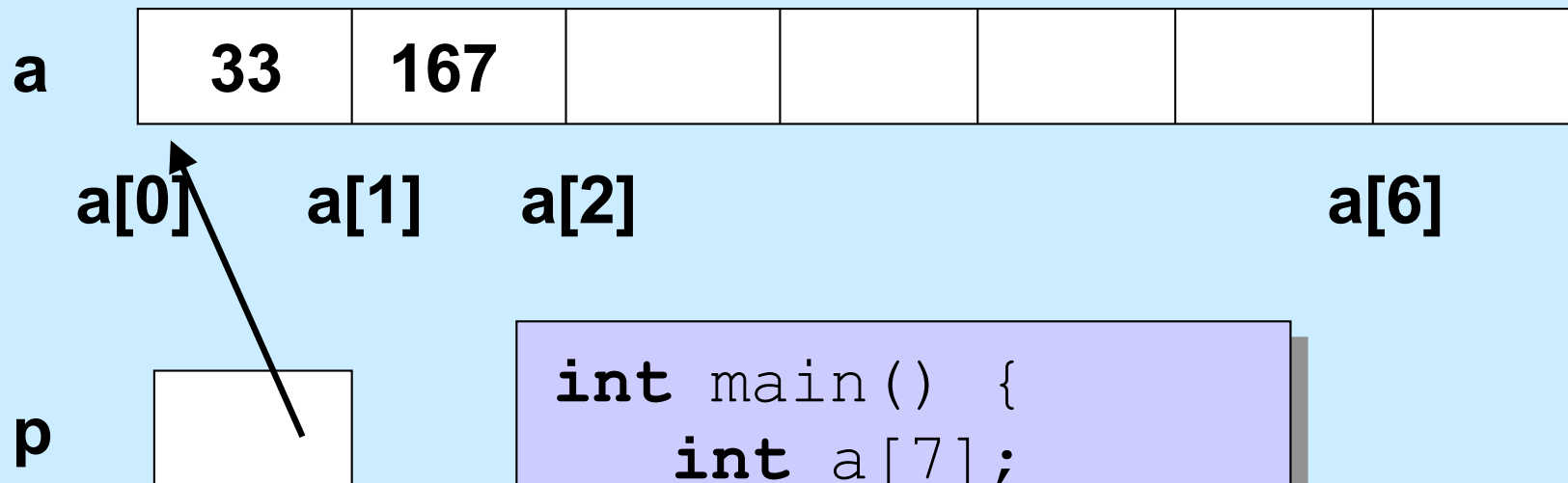


```
int main() {  
    int a[7];  
    int *p;  
    p = &a[0];  
    *p = 33;  
}
```

# Pointer Arithmetic

Pointers can be incremented/decremented

– what this does depends on its type

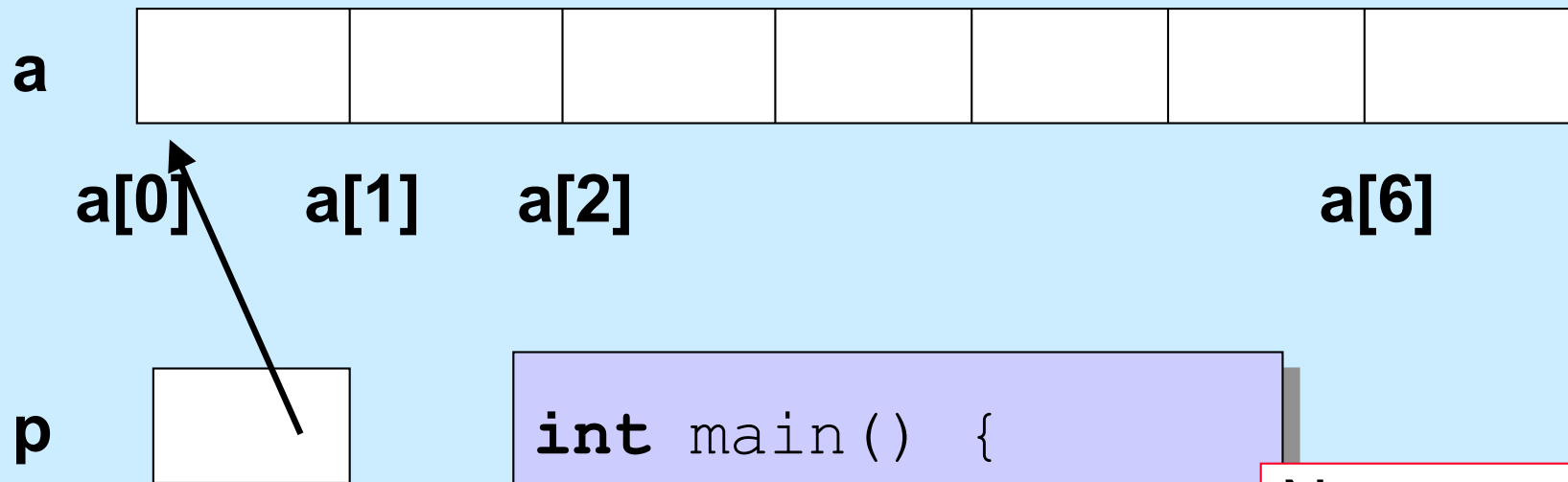


```
int main() {  
    int a[7];  
    int *p;  
    p = &a[0];  
    *p = 33;  
    *(p+1) = 167;  
}
```

# Pointer Arithmetic

Pointers can be incremented/decremented

– what this does depends on its type



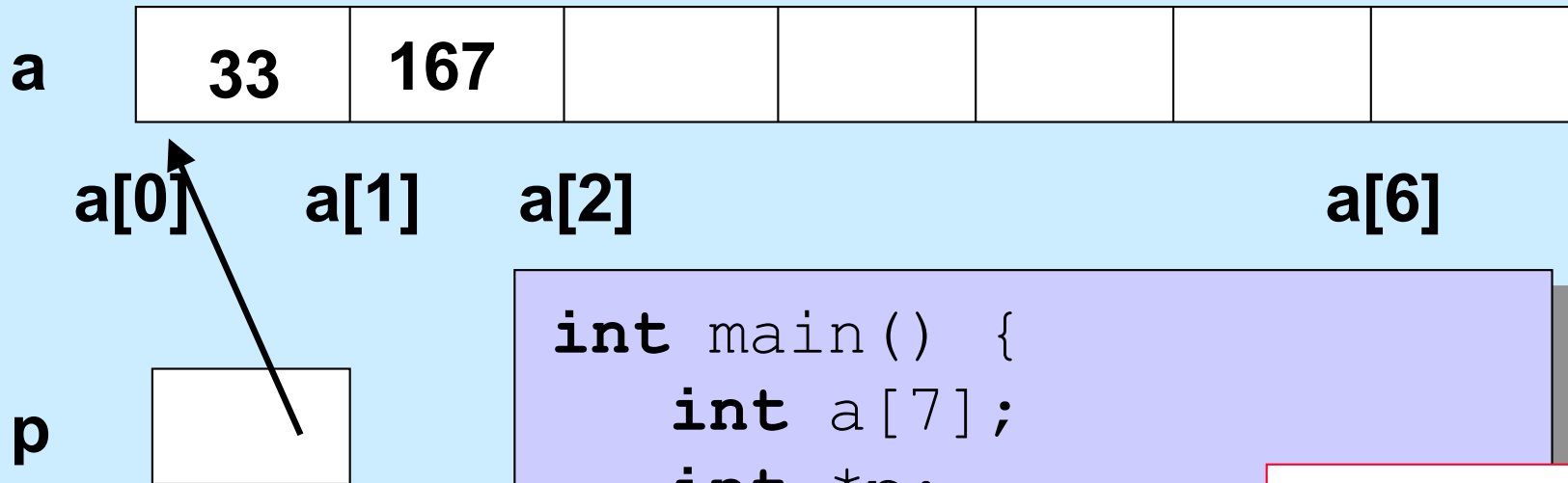
```
int main() {  
    int a[7];  
    int *p;  
    p = &a[0];  
}
```

Now **p** and **a**  
have the  
same value

# Pointer Arithmetic

Pointers can be incremented/decremented

– what this does depends on its type



```
int main() {  
    int a[7];  
    int *p;  
    p = a;  
    *p = 33;  
    p[1] = 167;  
}
```

The array name represents a pointer to its first element

# Pointers and Arrays

```
p = &a[0];
```

can also be written as

```
p = a;
```

```
a[i];
```

really is

```
*(a+i)
```

- **This makes sense, yet is weird ...**
  - **p is of type `int *`**
    - it can be assigned to

```
int *q;  
p = q;
```
  - **a sort of behaves like an `int *`**
    - but it can't be assigned to in the same way

```
a = q;
```



# Non-Array Variables

- **`int i`**
  - **four bytes of memory are allocated for `i`**  
`sizeof(i) == 4`
  - **`i` represents the contents of this memory, interpreted as an `int`**
  - **it makes sense to do, for example**  
`i = 7; // changes the contents of i`
- **`int *p`**
  - **8 bytes of memory are allocated for `p`**  
`sizeof(p) == 8`
  - **`p` represents the contents of this memory, interpreted as an `int *`**
  - **it makes sense to do, for example**  
`p = &i; // changes the contents of p`

# Array Variables

- **int A[6]**
  - **24 bytes of memory are allocated for A**  
`sizeof(A) == 24`
  - **A represents the address of the first byte**
  - **\*A is the value of the first int (as if A were an int \*)**
  - **it does not make sense to do**  
`A = &i; // would change the location of A`
- **int \*p = A;**
  - **8 bytes of memory are allocated for p**  
`sizeof(p) == 8`
  - **p represents the contents of this memory**
  - **\*p is the same as A[0]**
  - **it makes sense to do, for example**  
`p = &i;`

# Arrays and Functions

```
int func(int *a, int nelements) {  
    int i;  
    int result;  
    for (i=0; i<nelements; i++) {  
        *(a+i) = i;  
    }  
    return result;  
}
```

initialized with a copy  
of the argument

```
int main() {  
    int array[1000000000] = ... ;  
    printf("result = %d\n", func(array, 1000000000));  
    return 0;  
}
```

# Equivalently

```
int func(int a[], int nelements) {  
    int i;  
    int result;  
    for (i=0; i<nelements; i++) {  
        a[i] = i;  
    }  
    return result;  
}
```

initialized with a copy  
of the argument

```
int main() {  
    int array[1000000000] = ... ;  
    printf("result = %d\n", func(array, 1000000000));  
    return 0;  
}
```

# Equivalently

ignored

```
int func(int a[500], int nelements) {  
    int i;  
    int result;  
    for (i=0; i<nelements; i++) {  
        ...  
    }  
    return result;  
}  
  
int main() {  
    int array[1000000000] = ... ;  
    printf("result = %d\n", func(array, 1000000000));  
    return 0;  
}
```

# Parameter passing

## Passing arrays to a function

```
int average(int a[], int size) {  
    int i; int sum;  
    for(i=0, sum=0; i<size; i++)  
        sum += a[i];  
    return sum/size;  
}  
int main() {  
    int a[100];  
    ...  
    printf("%d\n", average(a, 100));  
}
```

# Swapping

**Write a function to swap two entries of an array**

```
void swap(int a[], int i, int j) {  
    int tmp;  
    tmp = a[j];  
    a[j] = a[i];  
    a[i] = tmp;  
}
```

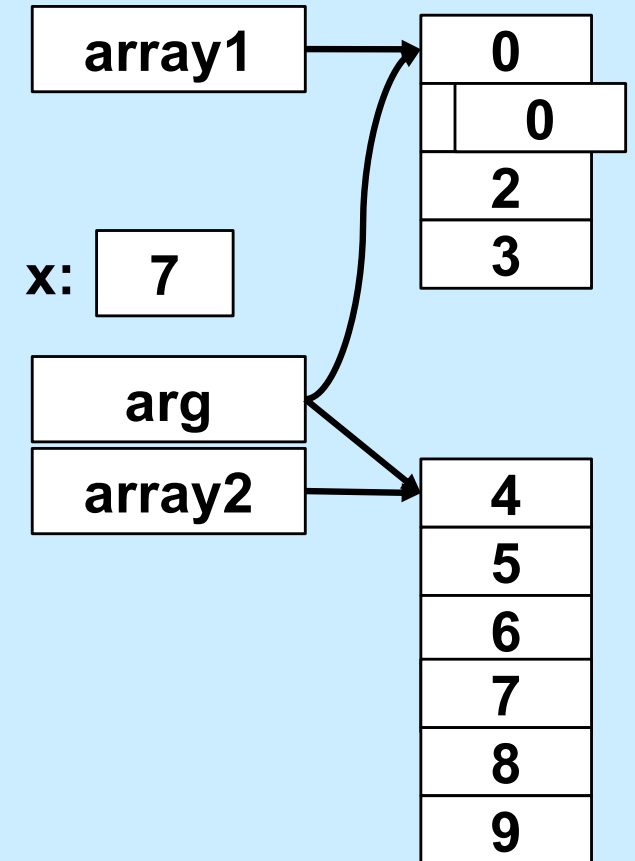
# Selection Sort

```
void selectsort(int array[], int length){
    int i, j, min;
    for (i = 0; i < length; ++i){
        /* find the index of the smallest item from i onward */
        min = i;
        for (j = i; j < length; ++j) {
            if (array[j] < array[min])
                min = j;
        }
        /* swap the smallest item with the i-th item */
        swap(array, i, min);
    }
    /* at the end of each iteration, the first i slots have the i
       smallest items */
}
```



# Arrays and Arguments

```
int func(int arg[]) {  
    int array2[6] = {4, 5, 6, 7, 8, 9};  
    arg[1] = 0;  
    arg = array2;  
    return arg[3];  
}  
  
int main() {  
    int array1[4] = {0, 1, 2, 3};  
    int x = func(array1);  
    printf("%d, %d\n", x, array1[1]);  
    return 0;  
}
```



```
$ ./a.out  
7 0
```

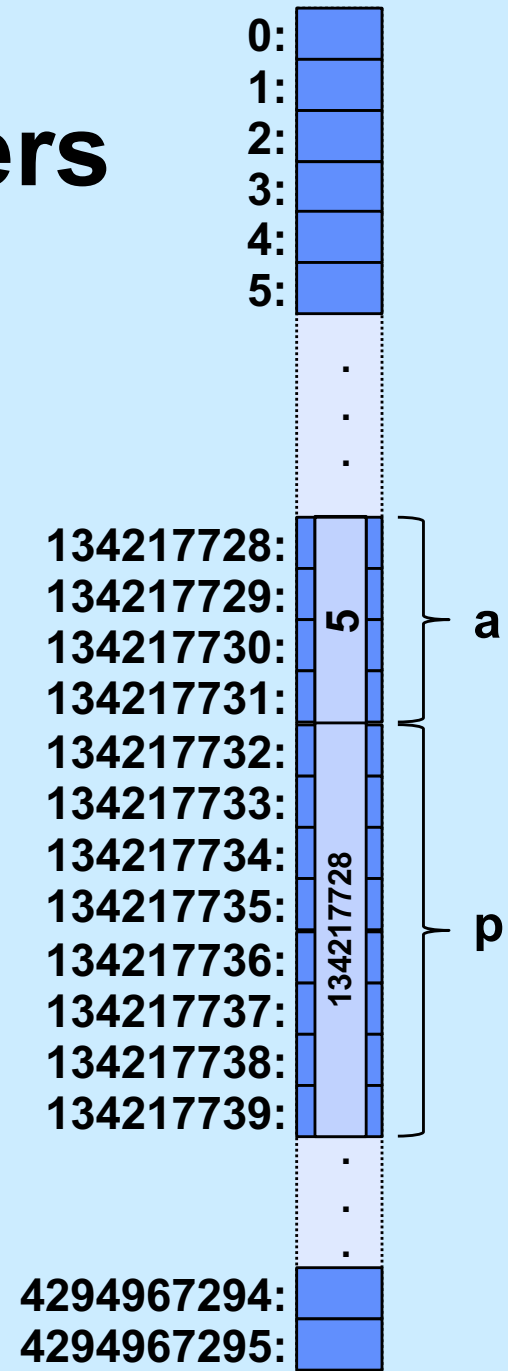
# Arrays and Arguments

```
void func(int arg[]) {  
    /* arg points to the caller's array */  
    int local[7];      /* seven ints */  
    arg++;              /* legal */  
    arg = local;        /* legal */  
    local++;           /* illegal */  
    local = arg;        /* illegal */  
}
```

# Dereferencing C Pointers

```
int main() {  
    int *p; int a = 4;  
    p = &a;  
    (*p)++;  
    printf("%d %p\n", *p, p);  
}
```

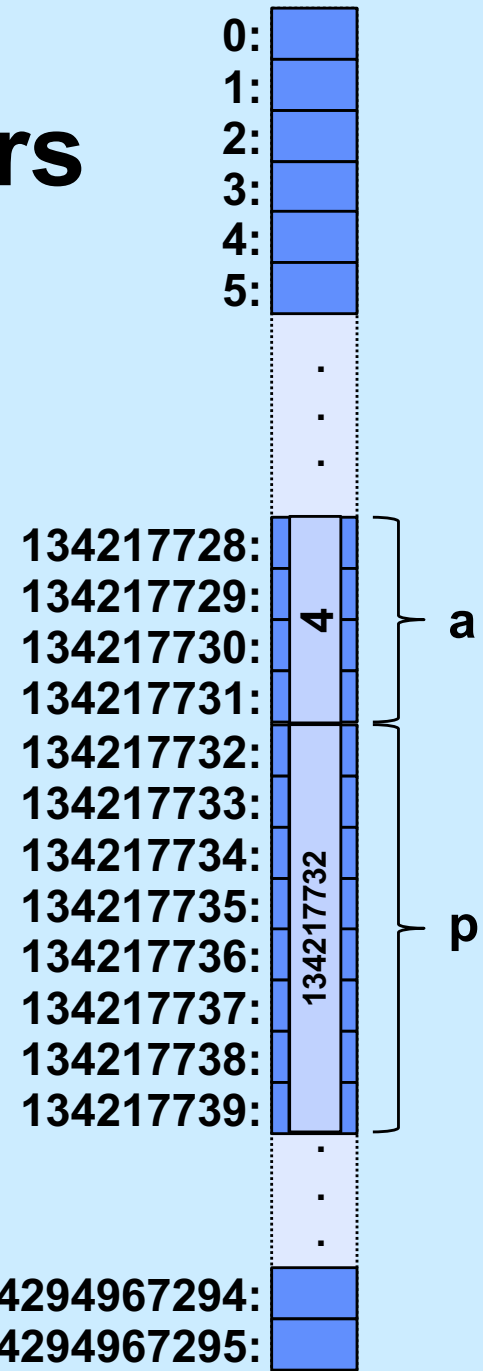
```
$ ./a.out  
5 134217728
```



# Dereferencing C Pointers

```
int main() {  
    int *p; int a = 4;  
    p = &a;  
    *p++;  
    printf("%d %p\n", *p, p);  
}
```

```
$ ./a.out  
134217732 134217732
```



# Dereferencing C Pointers

```
int main() {  
    int *p; int a = 4;  
    p = &a;  
    ++*p;  
    printf("%d %p\n", *p, p);  
}
```

```
$ ./a.out  
5 134217728
```

# Quiz 2

```
int func(int arg[]) {  
    arg++;  
    return arg[0];  
}  
  
int main() {  
    int A[3]={10, 11, 12};  
    printf("%d\n",  
        func(A) );  
}
```

**What's printed?**

- a) 9
- b) 10
- c) 11
- d) 12

# Quiz 3

```
int func(int a[]) {  
    int b[5] = {10, 11, 12, 13, 14};  
    a = b;  
    return a[1];  
}  
  
int main() {  
    int array[50];  
    array[1] = 0;  
    printf("result = %d\n",  
        func(array));  
    return 0;  
}
```

**This program prints:**

- a) 0
- b) 10
- c) 11
- d) **nothing: it doesn't compile because of a syntax error**

# Quiz 4

```
int func(int a[]) {  
    int b[5] = {10, 11, 12, 13, 14};  
    a = b;  
    return a[1];  
}  
  
int main() {  
    int array[5] = {9, 8, 7, 6, 5};  
    func(array);  
    printf("%d\n", array[1]);  
    return 0;  
}
```

**This program prints:**

- a) 7
- b) 8
- c) 10
- d) 11



# The Preprocessor

`#include`

- calls the preprocessor to include a file

What do you include?

- your own *header* file:

`#include "fact.h"`

– look in the current directory

- standard *header* file:

`#include <assert.h>`

`#include <stdio.h>`

– look in a standard place

Contains declaration of  
*printf* (and other things)

# Function Declarations

**fact.h**

```
float fact(int i);
```

**main.c**

```
#include "fact.h"  
int main() {  
    printf("%f\n", fact(5));  
    return 0;  
}
```

# #define

```
#define SIZE 100
int main() {
    int i;
    int a[SIZE];
}
```

## #define

- defines a substitution
- applied to the program by the preprocessor

# #define

```
#define forever for(;;)
int main() {
    int i;
    forever {
        printf("hello world\n");
    }
}
```

# assert

```
#include <assert.h>
float fact(int i) {
    int k, res;
    assert(i >= 0);
    for(res=1, k=1; k<=i; k++)
        res = res * k;
    return res;
}
int main() {
    printf("%f\n", fact(-1));
}
```

## assert

- verify that the assertion holds
- abort if not

```
$ ./fact
main.c:4: failed assertion 'i >= 0'
Abort
```