

# CS 33

## Introduction to C Part 2

# Why “pass by value”?

- Fortran, for example, passes parameters “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
```

# Memory addresses

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

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

```
$ ./a.out  
3221224352
```

**a:3221224352**

**4**

**Memory**

# 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("%u\n", p);  
}
```

p takes the address of a

```
$ ./a.out  
3221224352
```

# C Pointers

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

*a:3221224352*

*p*

**3221224352**

**4**

```
$ ./a.out  
3221224352
```

**Can you guess what &p is?**

# C Pointers

- **Pointers are typed**
  - the type of the objects they point to is 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

# 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
```

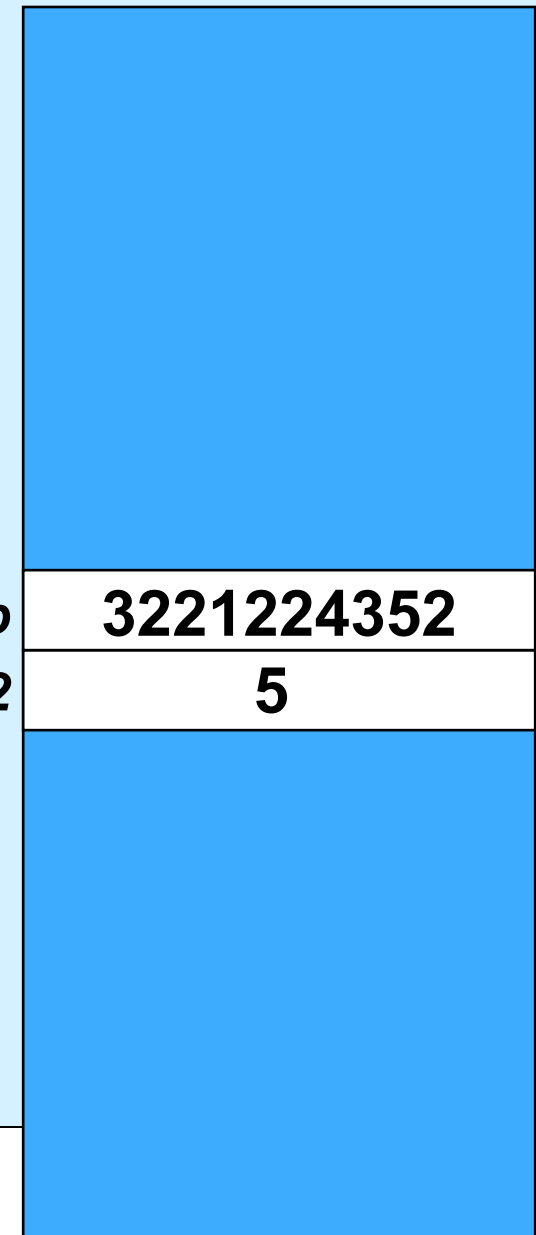


# C Pointers

- **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);  
}
```

*p*  
*a*:3221224352



```
$ ./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
```

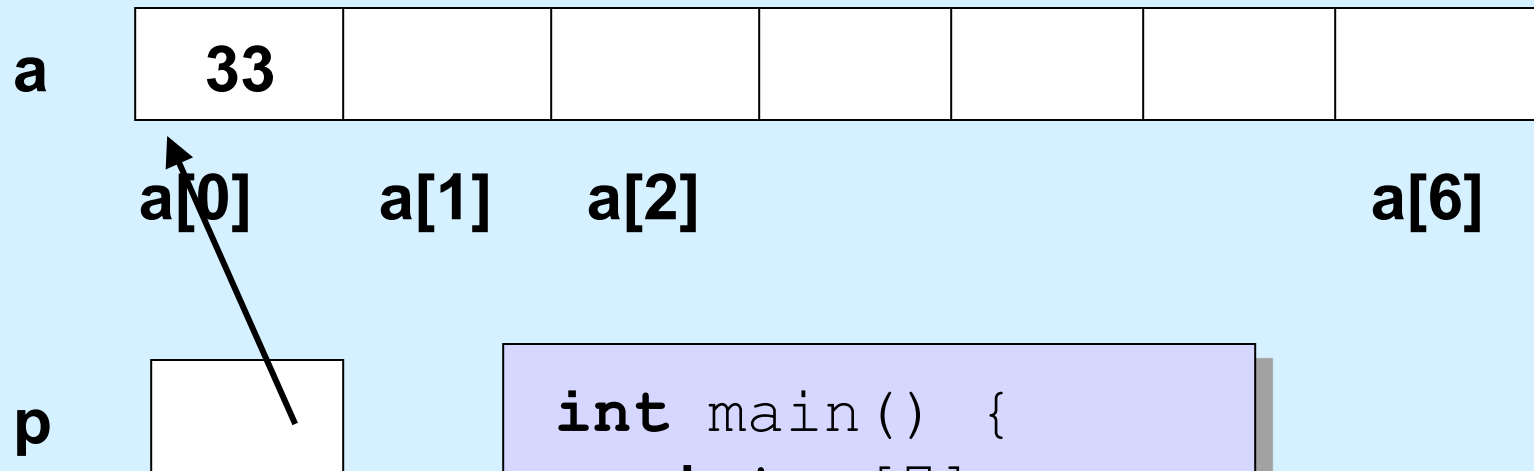
# Quiz 1

```
int doubleit(int *p) {  
    *p = 2*(*p);  
    return *p;  
}  
  
int main() {  
    int a = 3;  
    int b;  
    b = doubleit(&a);  
    printf("%d\n", a*b);  
}
```

**What's printed?**

- a) 0
- b) 12
- c) 18
- d) 36

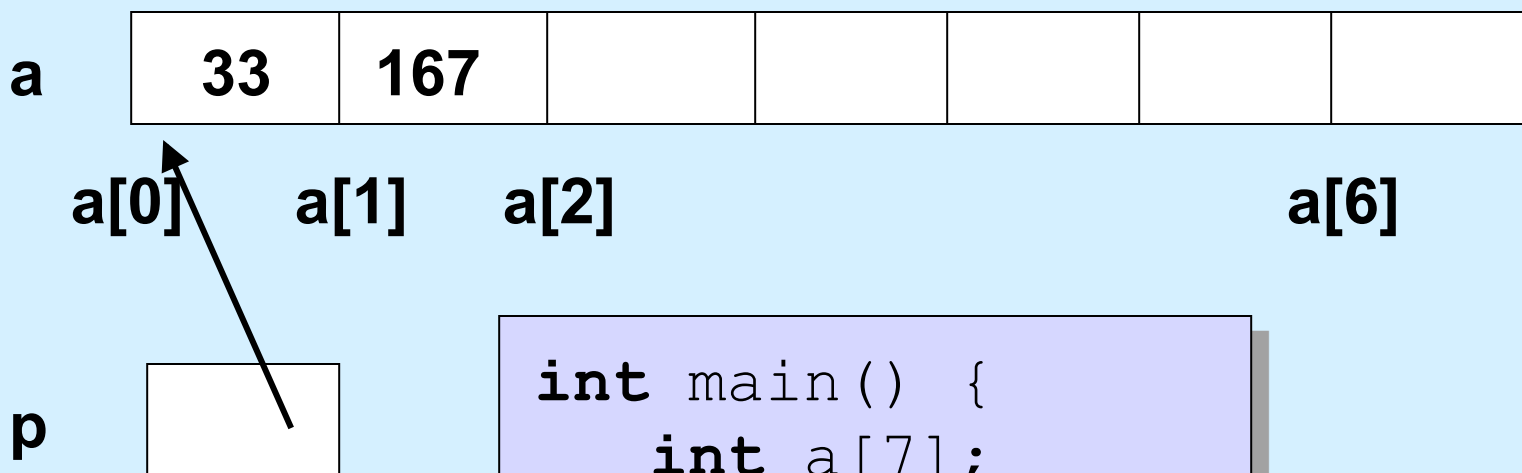
# Pointers and Arrays



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

# Pointer Arithmetic

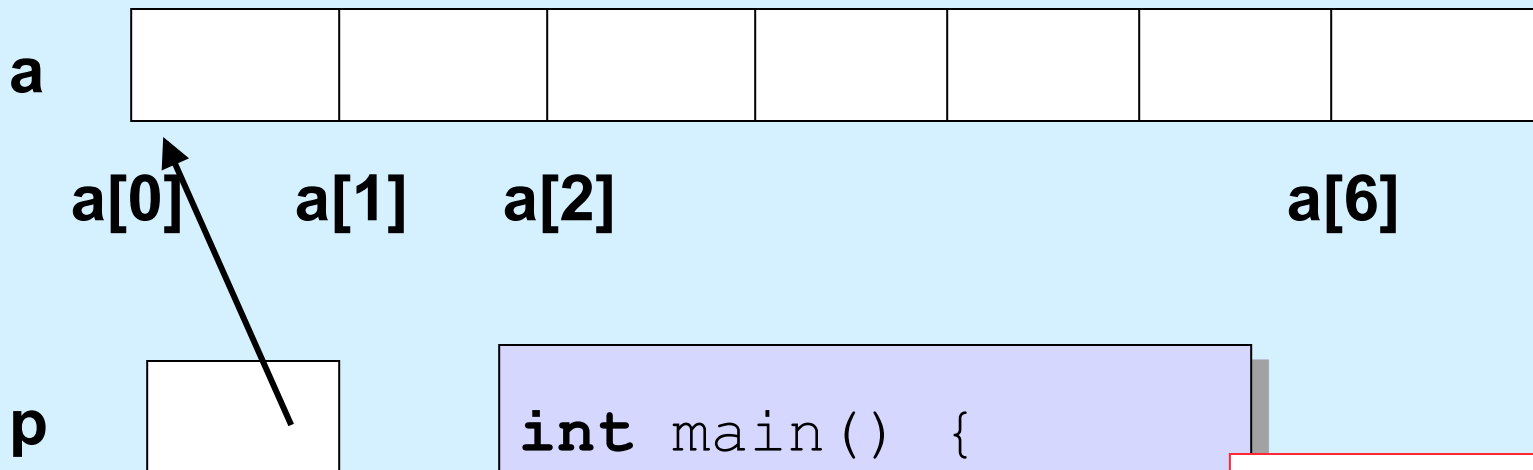
Pointers can be incremented/decremented  
– what this does depends on its type



```
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

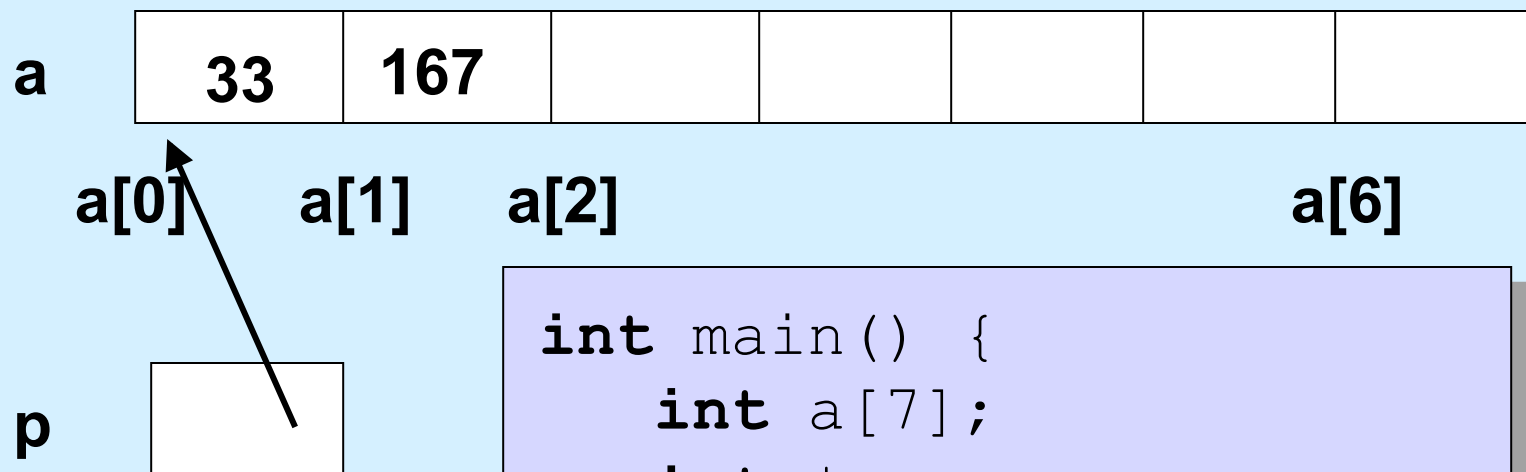


```
int main() {  
    int a[7];  
    int *p;  
    p = &a[0];  
}
```

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;  
}
```



# 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 and confusing ...**
  - **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

```
a = q;
```

# Pointers and Arrays

- An array name represents a pointer to the first element of the array
- Just like a literal represents its associated value

– in:

```
x = y + 2;
```

» “2” is a *literal* that represents the value 2

– can't do

```
2 = x + y;
```

# Literals and Procedures

```
int proc(int x) {  
    x = x + 4;  
    return x * 2;  
}
```

initialized with a copy  
of the argument

```
int main() {  
    result = proc(2);  
    printf("%d\n", result);  
    return 0;  
}
```

# Arrays and Procedures

```
int proc(int *a, int nelements) {  
    // sizeof(a) == sizeof(int *)  
    int i;  
    for (i=0; i<nelements-1; i++)  
        a[i+1] += a[i];  
    return a[nelements-1];  
}  
  
int main() {  
    int array[50] = ... ;  
    // sizeof(array) == 50*sizeof(int)  
    printf("result = %d\n", proc(array, 50));  
    return 0;  
}
```

initialized with a copy  
of the argument

# Equivalently ...

```
int proc(int a[], int nelements) {  
    // sizeof(a) == sizeof(int *)  
    ...  
}
```

No need for array size,  
since all that's used is  
pointer to first element

```
int main() {  
    int array[50] = ... ;  
    // sizeof(array) == 50*sizeof(int)  
    printf("result = %d\n", proc(array, 50));  
    return 0;  
}
```

# Quiz 2

```
int proc(int a[], int nelements) {
    int b[5] = {0, 1, 2, 3, 4};
    a = b;
    return a[1];
}

int main() {
    int array[50];
    array[1] = 0;
    printf("result = %d\n",
        proc(array, 50));
    return 0;
}
```

**This program prints:**

- a) 0
- b) 1
- c) 2
- d) **nothing: it doesn't compile because of a syntax error**

# Quiz 3

```
int proc(int a[], int nelements) {
    int b[5] = {0, 1, 2, 3, 4};
    a = b;
    return a[1];
}

int main() {
    int array[5] = {4, 3, 2, 1, 0};
    proc(array, 50);
    printf("%d\n", array[1]);
    return 0;
}
```

**This program prints:**

- a) 0
- b) 1
- c) 2
- d) 3