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):

```c
int main() {
    function(2);
    printf("%d\n", 2);
}

void function(int x) {
    x = 3;
}
```

$ ./a.out
3
In C
- you can get the memory address of any variable
- just use the magical operator &

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

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

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

$ ./a.out
3221224352
```
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%u\n", p);
}

$ ./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?

```c
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);
}
```

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

Damn!
C Pointers

• Dereferencing pointers
  – accessing/modifying the value pointed to by a pointer

```c
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

```c
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

```c
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
```
int doubleit(int *p) {
    *p = 2*(*p);
    return *p;
}

int main() {
    int a = 3;
    int b;
    b = doubleit(&a);
    printf("%d\n", a*b);
}
Pointers and Arrays

```c
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

```c
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

\[ p = &a[0]; \quad \text{can also be written as} \quad p = a; \]

\[ a[i]; \quad \text{really is} \quad *(a+i) \]

- **This makes sense, yet is weird and confusing ...**
  - \( p \) is of type \( \text{int}^* \)
    - it can be assigned to
      \[ \text{int}^* q; \]
      \[ p = q; \]
  - \( a \) sort of behaves like an \( \text{int}^* \)
    - but it can’t be assigned to
      \[ \underline{a} = q; \]
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:
    \[
    x = y + 2;
    \]
    \[
    \text{“2” is a literal that represents the value 2}
    \]
  – can’t do
    \[
    2 = x + y;
    \]
Literals and Procedures

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

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

initialized with a copy of the argument
Arrays and Procedures

```c
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 ...

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

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

No need for array size, since all that’s used is pointer to first element.
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;
}
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, 5);
    printf("%d\n", array[1]);
    return 0;
}
```

This program prints:

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