Arrays

Arrays are just sequential chunks of memory:

```cpp
char s[5] = {'H','e','l','l','o'};
```

Arrays and Pointers

Array variables are secretly pointers:

```cpp
char *p = s;
cout << *p << endl; // prints 'H'
cout << *s << endl; // also prints 'H'
```

Arrays and Pointers

Pointers are also secretly array variables:

```cpp
for (int j = 0; j < 5; j++) {
    cout << p[j] << endl;
}
```

To the compiler:

```cpp
p[j] == *(p + j)
```

More on this in a bit...

Incrementing Pointers

Here's a neat trick:

```cpp
char s[5] = {'H','e','l','l','o'};
char *p = s;
p++; // What does this print?
cout << *p << endl;
```
Looping with Pointers

```cpp
// Try this next:
for (int j = 0; j < 5; j++) {
    cout << *p;
    p++;
}
```

// or this…
```cpp
for (p = s; p < s + 5; p++) {
    cout << *p;
}
```

POINTER ARITHMETIC

Pointer Arithmetic: char

Suppose:
```cpp
char s[] = {'H', 'e', 'l', 'l', 'o'};
char* p = s;
```
We’ve stated that:
```cpp
p[j] == *(p + j)
```
Another way to look at it is:
```cpp
p[j] == s[j] when p == s
```
Thus:
```cpp
```

Pointer Arithmetic: int

Now, suppose we have:
```cpp
int arr[] = {42, 17, 33, 6};
int* q = arr;
```
```cpp
if
```cpp
q[j] == *(q + j) == arr[j]
```
This implies that:
```cpp
q[1] == *(q + 1) == arr[1] == 17
```
Then q + 1 is not simply 1 byte address beyond q, but must be 4 bytes beyond q.

POINTER ARITHMETIC

- Pointer arithmetic depends on type
  - char* p → p++ advances by 1 byte
  - int* q → q++ advances by 4 bytes (size of int)

- You can add or subtract:
  - q-- → go back 4 bytes
  - q + 3 → q plus 12 bytes

- Just keep in mind the array-pointer equivalence:
  type* ptr;
  *(ptr + k) == ptr[k]
  i.e.
  (ptr + k) == &ptr[k] // k * sizeof(type)
```

POINTER NOTES

Miscellaneous
C-style Strings

In C, strings are simply arrays of char:

```c
char *s = "Hello!"; // valid in C; for C++ add const
```

This array has size 7, not 6: the last entry stores byte value 0, or '\0':

```
H e l l o ! \0
```

Without this value, there's no way to detect the end of a string! With it, though, we can do:

```c
for (char* p = s; *p != '\0'; p++) { … }
```

String literals in C++ are still stored this way, but convert to the string type:

```c
string foo = "Hello";
```

or

```c
string("Hello")
```

Pointers and Reference Parameters

Reference parameters are not pointers!

Reference parameters are not pointers!

Reference parameters are not pointers!

If you have a function

```c
void foo(int &x) { … }
```

Inside `foo`, you cannot do

```c
*x = 10; // incorrect!
```

You just do:

```c
x = 10; // correct
```

Sources of confusion:

- & denotes a reference parameter
- & also used as address-of operator
- References use pointers "under the covers"

Up Next

- Wednesday, Sept. 27
  - Dynamic Memory Allocation
- Friday, Sept. 29
  - Lab 6 – Memory
- Monday, Oct. 2
  - Midterm Review
  - Project 2 Due
  - Lab 6 Due
- Wednesday, Oct. 4
  - Midterm 1 (in class)