CSCI 2132: Software Development

Pointers

Norbert Zeh

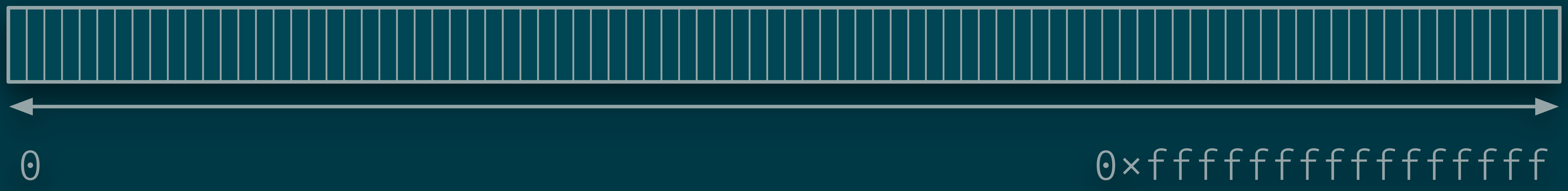
Faculty of Computer Science
Dalhousie University

Winter 2019

Pointers

Pointer = memory address
(e.g., of another variable)

- Hardware indexes memory addresses linearly.
- Addresses on modern processors more complicated



Pointer Variables

Pointer variable = variable that can store a pointer

Declaration:

```
type_to_be_referenced * variable_name;
```

Examples:

- `int *p;`
- `int* q;`
- `char **argv;`
- Careful: `int* a, b;`

Retrieving Addresses and Dereferencing

Address operator &:

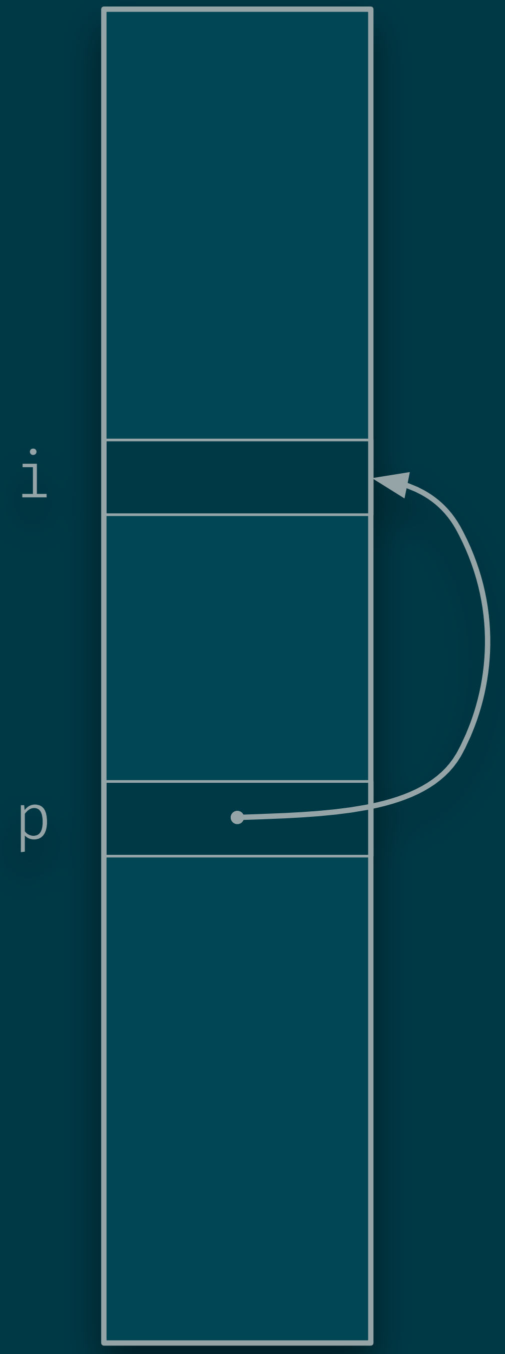
- Takes the address of a variable

```
int i, *p;
p = &i;
```

Indirection operator or dereference operator *:

- Accesses the memory location referenced by a pointer

```
int i, *p;
p = &i;
printf(“%d\n”, *p);
```



Common Pitfalls with Pointers

- Forgetting to dereference the pointer

```
int i = 1, *p;
p = &i;
p = 5;
```

- Dereferencing an un-initialized pointer

```
int *p;
*p = 5;
```

- **Dangling pointer**
  - Dereference pointer after object no longer exists on stack or heap

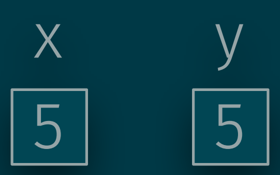
```
int *f() {
  int i = 4;
  return &i;
}

int *p;
p = f();
++(*p);
```

Pointers in Java?

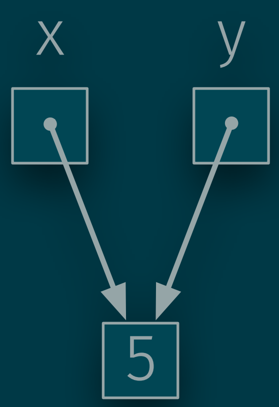
Java’s variable model:

- Primitive types (`int`, `char`, …) stored variables (**value model**).
- Objects (anything allocated with `new`) stored on heap, variable stores reference (pointer) to object (**reference model**).
- Pointers cannot be manipulated explicitly.
- Assignment in reference model makes two variables point to the same object (careful!).

```
int x = 5;
int y = x;
```



```
Integer x = new Integer(5);
Integer y = x;
```



Pointer Assignment

- Pointers can be passed around and stored in variables just as any other type.
- Only pointers of matching type can be assigned to pointer variables.

```
int i = 8, j = 15;
int *p = &i;
int *q;
int *r = &j;

*r = *p;
q = p;
(*q)++;

printf(“%d %d %d %d %d\n”, i, j, *p, *q, *r);
```

Pointer and Arrays

From the programmer’s point of view, C does not distinguish between an array and its first element!

```
int a[10];
*a = 15;
printf(“%d\n”, a[0]);
```

Pointer Arithmetic

Not a real operator

- Assume type *p, *q and int offset
- p + offset points to address addr(p) + offset * sizeof(type)
- p - offset points to address addr(p) - offset * sizeof(type)
- p < q if addr(p) < addr(q)
- p == q, p != q
- q - p = (addr(q) - addr(p)) / s

This is *(p++), not (*p)++.

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *start, *end, *p, sum;
start = a + 3;
end = a + 7;
for (sum = 0, p = start; p < end; sum += *p++);
printf("%d\n", sum);
```

Pointer Arithmetic or Array Indexing?

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *start, *end, *p, sum;
start = a + 3;
end = a + 7;
for (sum = 0, p = start; p < end; sum += *p++);
printf(“%d\n”, sum);
```

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int start, end, p, sum;
start = 3;
end = 7;
for (sum = 0, p = start; p < end; sum += a[p++]);
printf(“%d\n”, sum);
```

Which one is faster?

Pointer Arithmetic or Array Indexing?

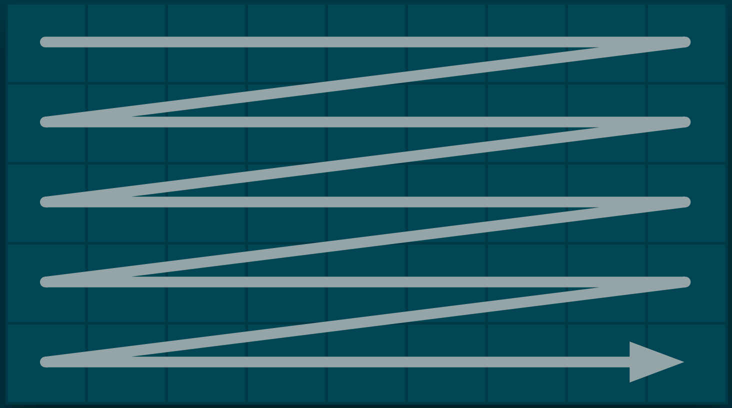
Traditionally, pointer arithmetic was faster than array indexing:

- **Array indexing:**
  - Access two variables: array and index
- **Pointer arithmetic:**
  - Access only pointer

Modern compilers (with `-O3` optimization option) translate array indexing into pointer arithmetic ⟶ no difference in efficiency.

A 2D Arrays Using Pointers

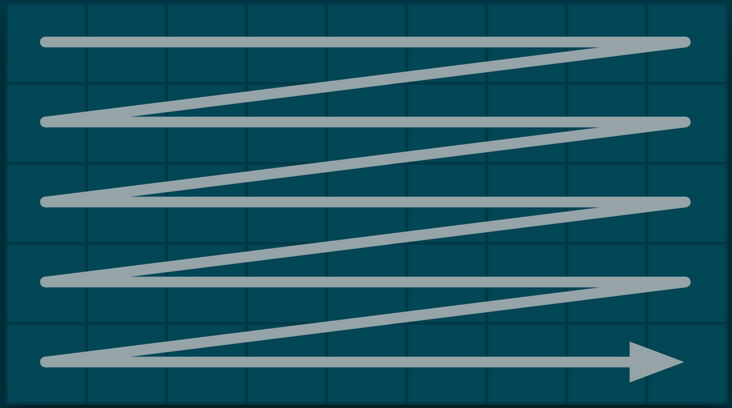
Memory is linear. How do we store 2D arrays?



```
#define WIDTH 20
#define HEIGHT 10
```

A 2D Arrays Using Pointers

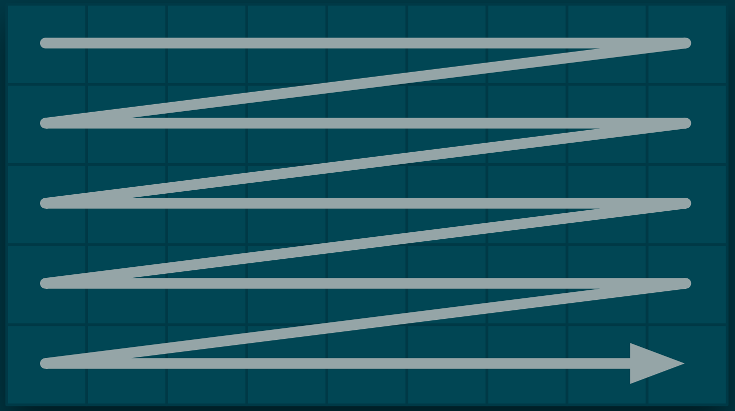
Memory is linear. How do we store 2D arrays?



```
#define WIDTH 20
#define HEIGHT 10

int a[WIDTH * HEIGHT];
```

A 2D Arrays Using Pointers

Memory is linear. How do we store 2D arrays?



```
#define WIDTH 20
#define HEIGHT 10

int a[WIDTH * HEIGHT];

// Access element in row i and column j
a[WIDTH * i + j] = ...
```

This will become important once we allocate dynamic arrays on the heap.