Raw Pointers

EECS 211

Winter 2018

Addresses in memory

	0_	1_	2_
0	17	798	13
1	5	-4	0
2	0	50	-1
3	0	12	-1
4	65	2	-1
5	98	4	-1
6	99	6	87
7	20	8	4
8	66	10	16
9	0	-9	255

Addresses in memory

	0_	1_	2_
0	17	798	13
1	5	-4	0
2	0	50	-1
3	0	12	-1
4	65	2	-1
5	98	4	-1
6	99	6	87
7	20	8	4
8	66	10	16
9	0	-9	255

Addresses in memory

	0_	1_	2_
0	17	798	13
1	5	-4	0
2	0	50	-1
3	0	12	-1
4	65	2	-1
5	98	4	-1
6	99	6	87
7	20	8	4
8	66	10	16
9	0	-9	255

```
int x = 50;
```

Addresses in memory

	0_	1_	2_
0	17	798	13
1	5	-4	0
2	0	50	-1
3	0	12	-1
4	65	2	-1
5	98	4	-1
6	99	6	87
7	20	8	4
8	66	10	16
9	0	-9	255

```
int x = 50;
// int x @ 12
```

Addresses in memory

	0_	1_	2_
0	17	798	13
1	5	−4	0
2	0	50	−1
3	0	12	−1
4	65	2	−1
5	98	4	−1
6	99	6	87
7	20	8	4
8	66	10	16
9	0	−9	255

```
int x = 50;
// int x @ 12

int* px = &x;
```

Addresses in memory

	0_	1_	2_
0	17	798	13
1	5	-4	0
2	0	50	-1
3	0	12	-1
4	65	2	-1
5	98	4	-1
6	99	6	87
7	20	8	4
8	66	10	16
9	0	-9	255

```
int x = 50;
// int x @ 12

int* px = &x;
// int* px @ 13
```

Addresses in memory

	0_	1_	2_
0	17	798	13
1	5	−4	0
2	0	50	−1
3	0	12	−1
4	65	2	−1
5	98	4	−1
6	99	6	87
7	20	8	4
8	66	10	16
9	0	−9	255

```
int x = 50;
// int x @ 12

int* px = &x;
// int* px @ 13
```

Addresses in memory

	0_	1_	2_
0	17	798	13
1	5	-4	0
2	0	50	-1
3	0	12	-1
4	65	2	-1
5	98	4	-1
6	99	6	87
7	20	8	4
8	66	10	16
9	0	-9	255

```
int x = 50;
// int x @ 12

int* px = &x;
// int* px @ 13

int a[] = { 2, 4, 6, 8, 10 };
```

Addresses in memory

	0_	1_	2_
0	17	798	13
1	5	-4	0
2	0	50	-1
3	0	12	-1
4	65	2	-1
5	98	4	-1
6	99	6	87
7	20	8	4
8	66	10	16
9	0	-9	255

```
int x = 50;
// int x @ 12

int* px = &x;
// int* px @ 13

int a[] = { 2, 4, 6, 8, 10 };
// int a[5] @ 14
```

Addresses in memory

	0_	1_	2_
0	17	798	13
1	5	-4	0
2	0	50	-1
3	0	12	-1
4	65	2	-1
5	98	4	-1
6	99	6	87
7	20	8	4
8	66	10	16
9	0	-9	255

```
int x = 50;
// int x @ 12

int* px = &x;
// int* px @ 13

int a[] = { 2, 4, 6, 8, 10 };
// int a[5] @ 14

int** ppx = &px;
```

Addresses in memory

	0_	1_	2_
0	17	798	13
1	5	-4	0
2	0	50	-1
3	0	12	-1
4	65	2	-1
5	98	4	-1
6	99	6	87
7	20	8	4
8	66	10	16
9	0	-9	255

```
int x = 50;
// int x @ 12

int* px = &x;
// int* px @ 13

int a[] = { 2, 4, 6, 8, 10 };
// int a[5] @ 14

int** ppx = &px;
// int** ppx @ 20
```

Addresses in memory

	0_	1_	2_
0	17	798	13
1	5	-4	0
2	0	50	-1
3	0	12	-1
4	65	2	-1
5	98	4	-1
6	99	6	87
7	20	8	4
8	66	10	16
9	0	-9	255

```
int x = 50;
// int x @ 12

int* px = &x;
// int* px @ 13

int a[] = { 2, 4, 6, 8, 10 };
// int a[5] @ 14

int** ppx = &px;
// int** ppx @ 20
```

Addresses in memory

	0_	1_	2_
0	17	798	13
1	5	-4	0
2	0	50	-1
3	0	12	-1
4	65	2	-1
5	98	4	-1
6	99	6	87
7	20	8	4
8	66	10	16
9	0	-9	255

```
int x = 50;
// int x @ 12

int* px = &x;
// int* px @ 13

int a[] = { 2, 4, 6, 8, 10 };
// int a[5] @ 14

int** ppx = &px;
// int** ppx @ 20
```

Using raw pointers

To get the address of a variable `x`, write `&x`

Using raw pointers

To get the address of a variable x, write &x

To *dereference* (get the value of) a raw pointer p, write *p

Using raw pointers

To get the address of a variable x, write &x

To *dereference* (get the value of) a raw pointer p, write *p

(You can assign raw pointers too: *p = x;)

Using raw pointers

To get the address of a variable x, write &x

To *dereference* (get the value of) a raw pointer p, write *p

(You can assign raw pointers too: *p = x;)

As operators, & and * are inverses!

Raw pointer example

```
int x = 4;
int y = 6;
```

Raw pointer example

```
int x = 4;
int y = 6;

int* p = &x;
CHECK( *p == 4 );
```

Raw pointer example

```
int x = 4;
int y = 6;

int* p = &x;
CHECK( *p == 4 );

x = 5;
CHECK( *p == 5 );
```

Raw pointer example

```
int x = 4;
int y = 6;

int* p = &x;
CHECK( *p == 4 );

x = 5;
CHECK( *p == 5 );

p = &y;
CHECK( *p == 6 );
```

Raw pointer example

```
int x = 4;
int y = 6;

int* p = &x;
CHECK( *p == 4 );

x = 5;
CHECK( *p == 5 );

p = &y;
CHECK( *p == 6 );

*p = 7;
CHECK( y == 7 );
```

& versus *

	*	&
as type (postfix)	`int*` means pointer to `int`	`int&` means reference to `int`
as expression (prefix)	`*p` dereferences pointer `p` to get value	`&x` takes address of variable `x` to get pointer

Raw arrays

This defines an uninitialized *raw array*:

```
int arr[5];
```

Raw arrays

This defines an uninitialized *raw array*:

```
int arr[5];
```

This defines an initialized raw array:

```
int arr[ ] = { 1, 2, 3, 4, 5 };
```

Raw arrays

This defines an uninitialized *raw array*:

```
int arr[5];
```

This defines an initialized raw array:

```
int arr[ ] = { 1, 2, 3, 4, 5 };
```

Raw arrays can be indexed just like vectors:

```
arr[n] = arr[m] + 6;
```

Raw arrays

This defines an uninitialized *raw array*:

```
int arr[5];
```

This defines an initialized raw array:

```
int arr[ ] = { 1, 2, 3, 4, 5 };
```

Raw arrays can be indexed just like vectors:

```
arr[n] = arr[m] + 6;
```

Unlike vectors, raw arrays don't know their size (so they can't bounds check):

```
arr.size();       // error!
```

Pointer arithmetic

Raw arrays are raw pointers in disguise:

```
int arr[] = { 2, 3, 4 };
```

Arrays can *decay* to pointers:

```
int* p = arr;
CHECK( arr[0] == *p );
```

Pointer arithmetic

Raw arrays are raw pointers in disguise:

```
int arr[] = { 2, 3, 4 };
```

Arrays can *decay* to pointers:

```
int* p = arr;
CHECK( arr[0] == *p );
```

Pointers are just addresses—numbers—so we can do arithmetic on them:

```
CHECK( p + 1 == &arr[1] );
CHECK( p + 2 == &arr[2] );
```

Pointer arithmetic

Raw arrays are raw pointers in disguise:

```
int arr[] = { 2, 3, 4 };
```

Arrays can *decay* to pointers:

```
int* p = arr;
CHECK( arr[0] == *p );
```

Pointers are just addresses—numbers—so we can do arithmetic on them:

```
CHECK( p + 1 == &arr[1] );
CHECK( p + 2 == &arr[2] );
CHECK( *(p + 1) == arr[1] );
CHECK( *(p + 2) == arr[2] );
```

Array indexing *is* pointer arithmetic

That is,

$\mathsf{arr}[\mathsf{i}]$ means the same thing as $*(\mathsf{arr} + \mathsf{i})$

Can’t return pointers to stack variables

This is fundamentally broken:

```
int* ptr_to_3()
{
        int x = 3;
        return &x;
}
```

Can’t return pointers to stack variables

This is fundamentally broken:

```
int* ptr_to_3()
{
        int x = 3;
        return &x;
}
```

So is this:

```
int* ptr_to_array()
{
        int arr[] = { 3, 4, 5 };
        return arr;
}
```

But we can allocate raw pointers on the free store

```
int* p = new int(3);
```

But we can allocate raw pointers on the free store

```
int* p = new int(3);

int* q = new int[]{ 3, 4, 5 };
```

But we can allocate raw pointers on the free store

```
int* p = new int(3);

int* q = new int[]{ 3, 4, 5 };

int* r = new int[32];
```

But we can allocate raw pointers on the free store

```
int* p = new int(3);

int* q = new int[]{ 3, 4, 5 };

int* r = new int[32];

int* s = new int[w * h];
```

But we can allocate raw pointers on the free store

C++ doesn't know when we are done with a raw pointer; we have to free the pointer with delete.

```
int* p = new int(3);

int* q = new int[]{ 3, 4, 5 };

int* r = new int[32];

int* s = new int[w * h];
```

But we can allocate raw pointers on the free store

C++ doesn't know when we are done with a raw pointer; we have to free the pointer with delete.

```
int* p = new int(3);                delete p;

int* q = new int[]{ 3, 4, 5 };

int* r = new int[32];

int* s = new int[w * h];
```

But we can allocate raw pointers on the free store

C++ doesn't know when we are done with a raw pointer; we have to free the pointer with delete.

```
int* p = new int(3);              delete p;

int* q = new int[]{ 3, 4, 5 };    delete [] q;

int* r = new int[32];             delete [] r;

int* s = new int[w * h];          delete [] s;
```

A rudimentary vector

```
struct Int_vector
{
        int* data;
        size_t capacity;    // amount allocated
        size_t size;        // amount used
};
```

– To CLion! –