Arrays

Arrays

- Arrays are a fundamental **data structure**, and they are extremely useful!

- We use arrays to hold values of the same type at contiguous memory locations.

- One way to analogize the notion of array is to think of your local post office, which usually has a large bank of post office boxes.

Arrays

Arrays	Post Office Boxes
An **array** is a block of contiguous space in memory…	A **mail bank** is a large space on the wall of the post office…

Arrays

Arrays	Post Office Boxes
An **array** is a block of contiguous space in memory…	A **mail bank** is a large space on the wall of the post office…
…which has been partitioned into small, identically-sized blocks of space called **elements**…	…which has been partitioned into small, identically-sized blocks of space called **post office boxes**…

Arrays

Arrays	Post Office Boxes
An **array** is a block of contiguous space in memory…	A **mail bank** is a large space on the wall of the post office…
…which has been partitioned into small, identically-sized blocks of space called **elements**…	…which has been partitioned into small, identically-sized blocks of space called **post office boxes**…
…each of which can store a certain amount of **data**…	…each of which can hold a certain amount of **mail**…

Arrays

Arrays	Post Office Boxes
An **array** is a block of contiguous space in memory…	A **mail bank** is a large space on the wall of the post office…
…which has been partitioned into small, identically-sized blocks of space called **elements**…	…which has been partitioned into small, identically-sized blocks of space called **post office boxes**…
…each of which can store a certain amount of **data**…	…each of which can hold a certain amount of **mail**…
…all of the same data type such as **`int`** or **`char`**…	…all of a similar type such as **letters** or **small packages**…

Arrays

Arrays	Post Office Boxes
An **array** is a block of contiguous space in memory…	A **mail bank** is a large space on the wall of the post office…
…which has been partitioned into small, identically-sized blocks of space called **elements**…	…which has been partitioned into small, identically-sized blocks of space called **post office boxes**…
…each of which can store a certain amount of **data**…	…each of which can hold a certain amount of **mail**…
…all of the same data type such as **`int`** or **`char`**…	…all of a similar type such as **letters** or **small packages**…
…and which can be accessed directly by an **index**.	…and which can be accessed directly by a **mailbox number**.

Arrays

- In C, the elements of an array are indexed starting from 0.
  - This is one of the major reasons we count from zero!
- If an array consists of *n* elements, the first element is located at index 0. The last element is located at index *(n*-1).
- C is very lenient. It will not prevent you from going "out of bounds" of your array; be careful!

Arrays

- Array declarations

```
type name[size];
```

  - The `type` is what kind of variable each element of the array will be.
  - The `name` is what you want to call your array.
  - The `size` is how many elements you would like your array to contain.

Arrays

- Array declarations

```
int student_grades[40];
```

  - The `type` is what kind of variable each element of the array will be.
  - The `name` is what you want to call your array.
  - The `size` is how many elements you would like your array to contain.

Arrays

- Array declarations

```
double menu_prices[8];
```

  - The `type` is what kind of variable each element of the array will be.
  - The `name` is what you want to call your array.
  - The `size` is how many elements you would like your array to contain.

Arrays

- If you think of a single element of an array of type `data-type` the same as you would any other variable of type `data-type` (which, effectively, it is) then all the familiar operations make sense.

```
bool truthtable[10];

truthtable[2] = false;
if(truthtable[7] == true)
{
   printf(“TRUE!\n”);
}
truthtable[10] = true;
```

Arrays

- If you think of a single element of an array of type `data-type` the same as you would any other variable of type `data-type` (which, effectively, it is) then all the familiar operations make sense.

```
bool truthtable[10];

truthtable[2] = false;
if(truthtable[7] == true)
{
   printf("TRUE!\n");
}
truthtable[10] = true;
```

Arrays

- When declaring and initializing an array simultaneously, there is a special syntax that may be used to fill up the array with its starting values.

```
// instantiation syntax
bool truthtable[3] = { false, true, true };

// individual element syntax
bool truthtable[3];
truthtable[0] = false;
truthtable[1] = true;
truthtable[2] = true;
```

Arrays

- When declaring and initializing an array simultaneously, there is a special syntax that may be used to fill up the array with its starting values.

```
// instantiation syntax
bool truthtable[] = { false, true, true };

// individual element syntax
bool truthtable[3];
truthtable[0] = false;
truthtable[1] = true;
truthtable[2] = true;
```

Arrays

- Arrays can consist of more than a single dimension. You can have as many size specifiers as you wish.

```
bool battleship[10][10];
```

- You can choose to think of this as either a 10x10 grid of cells.
  - In memory though, it's really just a 100-element one-dimensional array.
  - Multi-dimensional arrays are great **abstractions** to help visualize game boards or other complex representations.

Arrays

- While we can treat individual elements of arrays as variables, we cannot treat entire arrays themselves as variables.

- We cannot, for instance, assign one array to another using the assignment operator. That is not legal C.

- Instead, we must use a loop to copy over the elements one at a time.

Arrays

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

bar = foo;
```

Arrays

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

bar = foo;
```

Arrays

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

for(int j = 0; j < 5; j++)
{
    bar[j] = foo[j];
}
```

Arrays

- Recall that most variables in C are **passed by value** in function calls.

- Arrays do not follow this rule. Rather, they are **passed by reference**. The callee receives the actual array, not a copy of it.
  - What does that mean when the callee manipulates elements of the array?

- For now, we'll gloss over why arrays have this special property, but we'll return to it soon enough!

Arrays

```
void set_array(int array[4]);
void set_int(int x);

int main(void)
{
   int a = 10;
   int b[4] = { 0, 1, 2, 3 };
   set_int(a);
   set_array(b);
   printf("%d %d\n", a, b[0]);
}

void set_array(int array[4])
{
   array[0] = 22;
}

void set_int(int x)
{
   x = 22;
}
```

Arrays

```
10, 22
```