

Arrays

Suppose I need to compute statistics on class marks?

```
int mark_student0, mark_student1, mark_student2, ...;  
mark_student0 = 73;  
mark_student1 = 42;  
mark_student2 = 99;  
...
```

- cumbersome, need hundreds of individual variables
- can't write while loop which executes for each student
- becomes unfeasible if dealing with a lot of values

Solution use an array

```
int mark[930];  
mark[0] = 73;  
mark[1] = 42;  
mark[2] = 99;  
...
```

C Arrays

- C array is a collection of variables called **array elements**.
- All array elements must be the same type.
- Array elements don't have a name
- Array elements accessed by a number called the **array index**.
- Valid array indices for array with n elements are $0 \dots n - 1$
- Array can have millions/billions of elements.
- Array elements must be initialized.
- Can't assign scanf/printf whole arrays.
- Can assign scanf/printf array elements.

Arrays

```
// Declare an array with 10 elements  
// and initialises all elements to 0.  
int myArray[10] = {0};
```

	myArray
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Arrays

```
// Declare an array with 10 elements  
// and initialises all elements to 0.
```

```
int myArray[10] = {0};
```

```
// Put some values into the array.
```

```
myArray[0] = 3;
```

	myArray
0	3
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Arrays

```
// Declare an array with 10 elements  
// and initialises all elements to 0.
```

```
int myArray[10] = {0};
```

```
// Put some values into the array.
```

```
myArray[0] = 3;
```

```
myArray[5] = 17;
```

	myArray
0	3
1	0
2	0
3	0
4	0
5	17
6	0
7	0
8	0
9	0

Arrays

```
// Declare an array with 10 elements  
// and initialises all elements to 0.
```

```
int myArray[10] = {0};
```

```
// Put some values into the array.
```

```
myArray[0] = 3;
```

```
myArray[5] = 17;
```

```
myArray[10] = 42; // <-- Error
```

	myArray
0	3
1	0
2	0
3	0
4	0
5	17
6	0
7	0
8	0
9	0

Reading Arrays

Scanf can't read an entire array. This will read only 1 number:

```
#define ARRAY_SIZE 42  
...  
int array[ARRAY_SIZE];  
scanf("%d", &array);
```

Instead you must read the elements one by one:

```
i = 0;  
while (i < SIZE) {  
    scanf("%d", &array[i]);  
    i = i + 1;  
}
```

Printing Arrays

printf can't print an entire array. This won't compile:

```
#define ARRAY_SIZE 42  
...  
int array[ARRAY_SIZE];  
printf("%d", array);
```

Instead must print the elements one by one:

```
i = 0;  
while (i < ARRAY_SIZE) {  
    printf("%d\n", array[i]);  
    i = i + 1;  
}
```


Copying Arrays

Suppose we have the following:

```
int array1[5] = {1, 2, 3, 4, 5};  
int array2[5];
```

Array assignment not allowed in C. This won't compile:

```
array2 = array1;
```

Instead must must copy the elements one by one:

```
i = 0;  
while (i < 5) {  
    array2[i] = array1[i];  
    i = i + 1;  
}
```

Copying Arrays

Suppose we have the following:

```
int array1[5] = {1, 2, 3, 4, 5};  
int array2[5];
```

Array assignment not allowed in C. This won't compile:

```
array2 = array1;
```

Instead must must copy the elements one by one:

```
i = 0;  
while (i < 5) {  
    array2[i] = array1[i];  
    i = i + 1;  
}
```

Static Array Initialisers

Other ways to define arrays:

```
// If no size is given C counts how many elements  
// you have specified to determine the size  
int myArray1[] = {3,12,9,12,8,17,33,22,43,10};  
int myArray2[10] = {3,12}; //The rest is padded with 0's  
int myArray3[10] = {3};    //The rest is padded with 0's  
//A common way to initialise the whole array to 0  
int myArray4[10] = {0};    //The rest is padded with 0's
```

Each definition creates a **int** array with 10 elements.

Argument Passing: Array Variables

Array arguments are passed by [reference](#)

- The array is not copied so changes to array elements are visible the outside function
- Full explanation will have to wait until we cover pointers

Arrays as Function Arguments

```
int main(void) {  
    int nums[5] = {0};  
    f(nums,5);           // pass nums as argument  
    printf("%d\n", nums[0]); // what is printed?  
}  
  
void f(int nums[], int size) {  
    nums[0] = 42;        // modify argument  
}
```

Arrays as Function Arguments

```
int main(void) {  
    int nums[5] = {0};  
    f(nums,5);           // pass nums as argument  
    printf("%d\n", nums[0]); // what is printed?  
}  
void f(int nums[], int size) {  
    nums[0] = 42;        // modify argument  
}
```

`printf()` $\Rightarrow$ 42. Why?

Because a **reference** to the original copy of the array `nums` is passed to `f()`, any changes to the **referenced** `nums` in `f()` is reflected in the original `nums`.

Arrays as Function Arguments

Examples of how the prototypes can be declared:

```
void f1(double ff[]);  
void f2(double ff[SIZE]);  
void f3(double ff[], int size);
```

Warning:

Notice that the size may be left unspecified.

In these cases it is up to the programmer to manage the number of elements in its array argument.

Arrays as Function Arguments

Examples of how the prototypes can be declared:

```
void f1(double ff[]);  
void f2(double ff[SIZE]);  
void f3(double ff[], int size);
```

Warning:

Notice that the size may be left unspecified.

In these cases it is up to the programmer to manage the number of elements in its array argument.

Options:

- by using a size constant
- by passing in a size variable, like in the `f3()` example

Arrays as Function Arguments

Consider the following:

```
#define SIZE 10
int sum1(int nums[]);
int sum2(int nums[SIZE]);
int sum3(int nums[], int size);

int main(void) {
    int nums[10] = {1, 2, 3};
    sum1(nums);
    sum2(nums);
    sum3(nums, 3);
    return 0;
}
```

Arrays as Function Arguments

The functions

- `sum1` and `sum2` uses `SIZE` to iterate through its array argument
- `sum3` uses the supplied `size` argument.

Why is `sum3` better?

Arrays as Function Arguments

The functions

- `sum1` and `sum2` uses `SIZE` to iterate through its array argument
- `sum3` uses the supplied `size` argument.

Why is `sum3` better? You can pass in arrays of different sizes, or tell the function to just sum the first 'size' elements in the array.

Beware: Don't Try to Return an Array

It might be tempting to try returning an array from a function:

```
int[] foo(void) {  
    int nums[] = {1,2,3};  
    return nums;  
}
```

Beware: Don't Try to Return an Array

It might be tempting to try returning an array from a function:

```
int[] foo(void) {  
    int nums[] = {1,2,3};  
    return nums;  
}
```

This looks ok but **fails** spectacularly!

Arrays are passed **by reference**, but the array is destroyed immediately after the **return** statement. Using it in the caller then becomes a **run-time error**!

It is possible to return dynamically allocated arrays, which we will learn later in the course.

Beware: Don't Try to Return an Array

Instead of returning an array you can pass in an array ,fill it with values.

```
int main(void){
    int numbers[SIZE];
    //foo fills the numbers array with with values
    foo(numbers,SIZE);
    //Now you can use the numbers array
    //etc
}

void foo(int nums[], int size) {
    int i = 0;
    while(i < size){
        nums[i] = 42;
    }
}
```

Arrays of Arrays

- C supports arrays of arrays.
- Useful for multi-dimensional data.

```
int matrix[3][3] = { {1, 2, 3},  
                     {4, 5, 6},  
                     {7, 8, 9} };  
  
printf("%d\n", matrix[1][1]);
```

Read a Two-dimensional Array

```
#define SIZE 42

...
int matrix[SIZE][SIZE];
int i, j;

i = 0
while (i < SIZE) {
    j = 0;
    while (j < SIZE) {
        scanf("%d", &matrix[i][j]);
        j = j + 1;
    }
    i = i + 1;
}
```


Print a Two-dimensional Array

```
...  
  
while (i < SIZE) {  
    j = 0;  
    while (j < SIZE) {  
        print("%d", &matrix[i][j]);  
        j = j + 1;  
    }  
    printf("\n");  
    i = i + 1;  
}
```