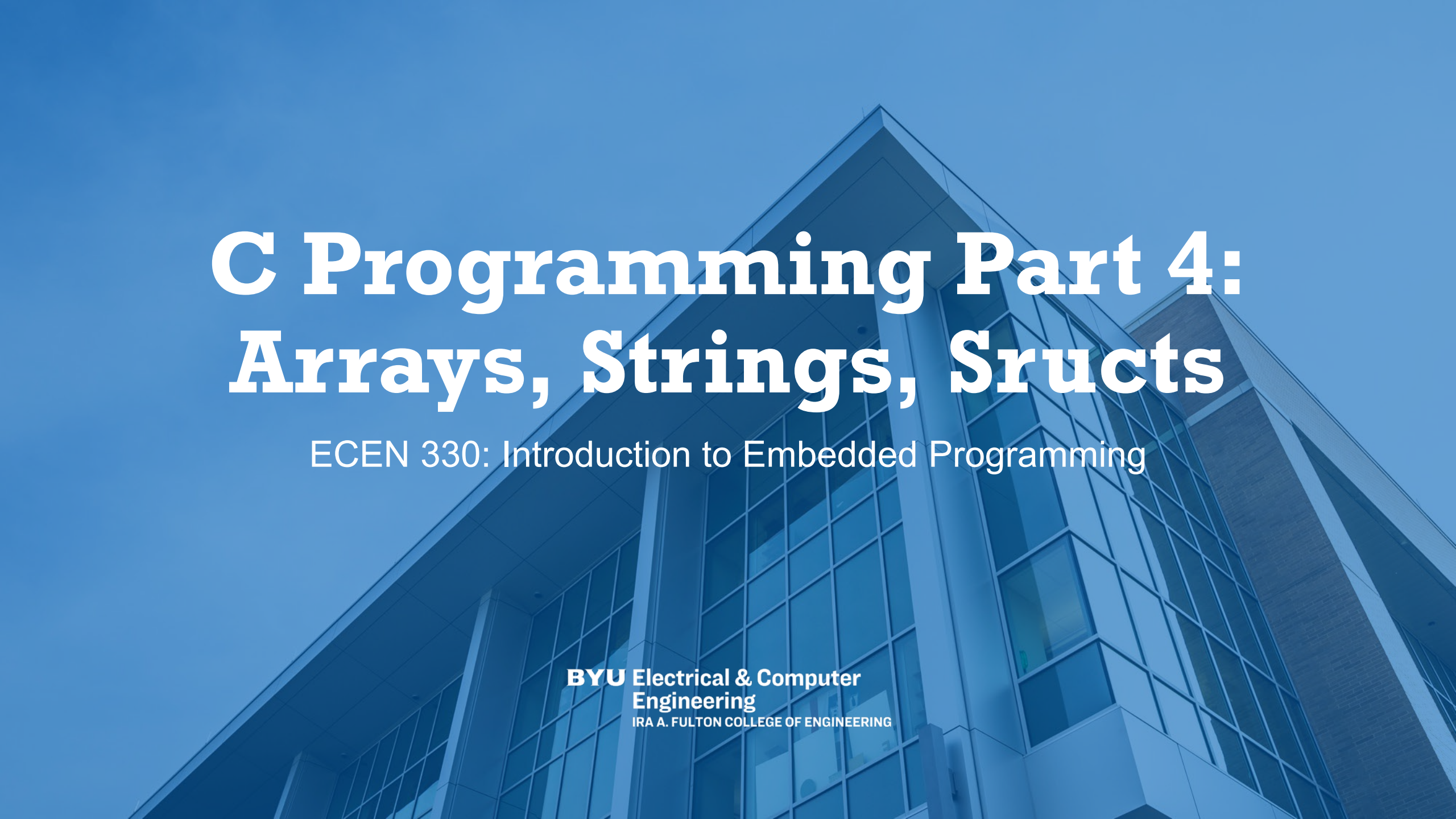


C Programming Part 4: Arrays, Strings, Sructs

ECEN 330: Introduction to Embedded Programming

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

C Arrays

- C arrays are declared in the following form:

```
type name[number of elements];
```

- For example, if we want an array of six integers :

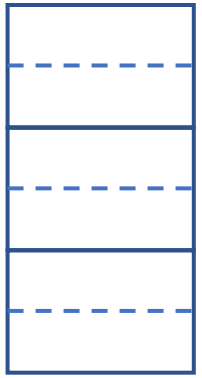
```
int numbers[6];
```

Memory

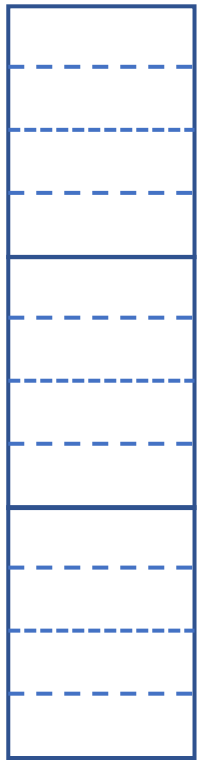
In C, arrays are always stored in contiguous memory

`uint8_t vals [3]`

`int16_t vals [3]`



`uint32_t vals [3]`



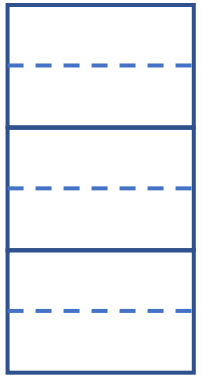
sizeof()

sizeof() returns the size of the array in bytes:

`uint8_t vals [3]`

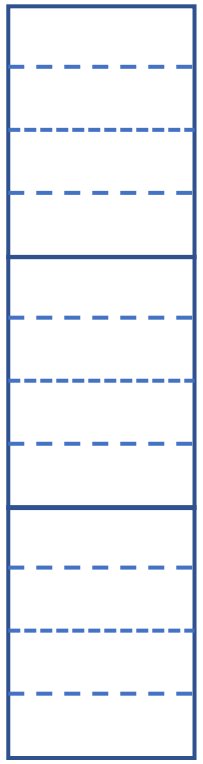
`sizeof(vals) is 3`

`int16_t vals [3]`



`sizeof(vals) is 6`

`uint32_t vals [3]`



`sizeof(vals) is 12`

Initializers


```
uint8_t vals [4] = {1, 2, 3, 10};
```

1
2
3
10

```
uint8_t vals [6] = {13};
```

13
0
0
0
0
0

```
uint8_t vals [] = {1, 2, 3, 10};
```

1
2
3
10

Accessing

- Access using [ ]

```
uint8_t point [4] = {1, 2, 3, 10};
int x;
x = point[2];
```

- There is no checking to make sure the index is within the array:

```
char y;
int z = 9;
char point[6] = { 1, 2, 3, 4, 5, 6 };
//examples of accessing outside the array. A compile error is not always raised
y = point[15];
y = point[-4];
y = point[z];
```

2D Arrays


```
char two_d[3][5];
```

To access/modify a value in this array we need two subscripts:

```
char ch;
ch = two_d[2][4];
```

or

```
two_d[0][0] = 'x';
```

Similarly, a multi-dimensional array can be initialized like this:

```
int two_d[2][3] = {{ 5, 2, 1 },
                   { 6, 7, 8 }};
```

Strings

- There is no string data type in C
- Strings are arrays of 1 byte ASCII values, ended with a 0 (null terminator).
- String constants inside " ", implicitly include the null terminator.

```
char s[3] = "hi";
```

‘h’
‘i’
‘\0’

104
105
0

```
char s[6] = "hi";
```

‘h’
‘i’
0
0
0
0

What is sizeof(s)?

What is strlen(s)?

Strings

- Is this a string?

```
char s[5] = "hello";
```

‘h’
‘e’
‘l’
‘l’
‘o’

- What will this print?

```
printf("%d\n", strlen(s));
```

- What will this print?

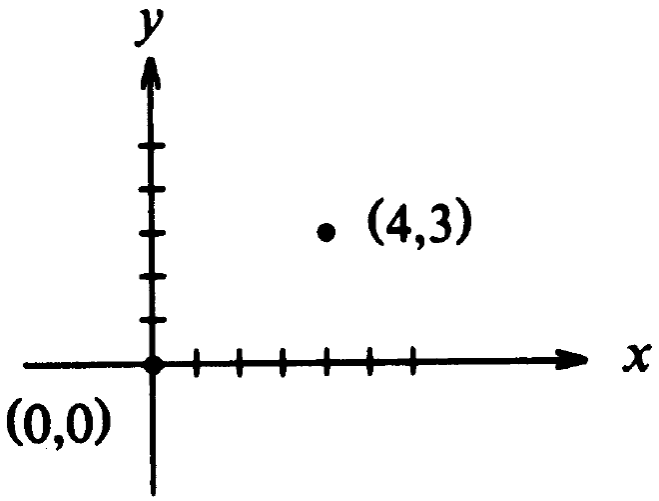
```
printf("%s\n", s);
```

structs

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

Basics of Structures

- A structure is a collection of one or more variables
- Structures help to organize complicated data
- Related variables can be treated as a unit
- The keyword **`struct`** introduces a structure declaration



structure tag (optional)

```
struct point {
    int x;
    int y;
}; /* reserves no storage */
```

members

- Variables may follow after right brace

```
struct point { ... } a, b, c; /* space reserved */
```

Basics of Structures

- If a structure is tagged, it can be used later in definitions

```
struct point pt;
```

Defines a variable `pt` which is a structure of type `struct point`

- A structure can be initialized with a list of initializers

```
struct point maxpt = { 320, 200 };
```

One for each member

- A structure member is referred to with a “dot” operator

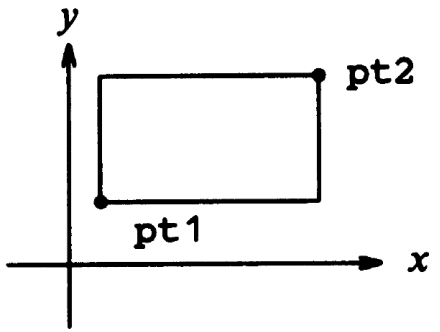
`structure_name.member`

```
printf("%d,%d\n", pt.x, pt.y);
if (pt.x > maxpt.x) …
```

Basics of Structures

- Structures can be nested, consider a pair of points

```
struct rect {
    struct point pt1;
    struct point pt2;
};
```



- Declare `block` as `rect` structure

```
struct rect block;
```

- Refer to the `x` coordinate of the `pt1` member of `block`

```
block.pt1.x
```

Structures and Functions

- Legal operations on a structure
  - Copying it or assigning to it as a unit ← Includes function arguments and function return values
  - Accessing its members
  - Taking its address with `&`
- Structures may not be compared
- Three approaches to passing data
  - Pass components separately
  - Pass an entire structure
  - Pass a pointer to a structure (we won't cover this until we go over pointers)

Pass Components Separately

- Take two integers and return a point structure

```
/* makepoint: make a point from x and y components */
struct point makepoint(int x, int y)
{
    struct point temp;
    temp.x = x;
    temp.y = y;
    return temp;
}
```

No conflict between the argument name and the member with the same name

- `makepoint` can be used in place of a struct variable

```
struct rect block; /* define block */
block.pt1 = makepoint(0, 0); /* initialize pt1 */
block.pt2 = makepoint(XMAX, YMAX); /* init pt2 */
```

Pass an Entire Structure

- Both the arguments and the return value are structures

```
/* addpoints: add two points */
struct point addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

Passed by value

Increment the components in `p1` (a copy) rather than make another temporary variable

- Add two points

```
struct point p1, p2, ptsum;
…
ptsum = addpoint(p1, p2);
```

Typedef

- C provides **`typedef`** for creating new data type names

```
typedef int length_t;
```

New type name

- `length_t` can now be used the same way as `int`

```
length_t len, maxlen;
length_t lengths[];
```

_t is a convention used to indicate a type

- Main reasons for using `typedef`
  - Parameterize a program against portability problems
    - If many variables use the same type, we can change it in the future by only changing one place.
  - Provide better documentation / readability

```
typedef char status_t;
typedef int16_t minimax_score_t;
```

Struct with Typedef


```
struct point {
    int x;
    int y;
};
struct point p1;
p1.x = 5;
p1.y = 6;
```

Notice that we had to type "**struct** point"
- Programmers are lazy

```
struct point {
    int x;
    int y;
};
typedef struct point point_t;

point_t p1;
```

OR

```
typedef struct point {
    int x;
    int y;
} point_t ;
```

OR

```
typedef struct {
    int x;
    int y;
} point_t ;
```

```
struct point {
   int x;
   int y;
};
typedef struct point point_t;

point_t p1;
```

```
typedef struct point {
   int x;
   int y;
} point_t ;
```

```
typedef struct {
   int x;
   int y;
} point_t ;
```

It's okay to have the typdef name match the struct name…

```
struct point {
   int x;
   int y;
};
typedef struct point point;

point p1;
struct point p2;
```

```
typedef struct point {
   int x;
   int y;
} point ;
```

```
typedef struct {
   int x;
   int y;
} point ;
```

Watch out! …These are very different


```
struct point {
    int x;
    int y;
} myPoint;
```

This defines a variable variable myPoint
(and allocates memory)

```
typedef struct {
    int x;
    int y;
} point_t;
```

This defines a new type point_t
(point_t isn't a variable)

```
myPoint.x = 0;
myPoint.y = 13;
```

```
point_t myPoint;

myPoint.x = 0;
myPoint.y = 13;
```