

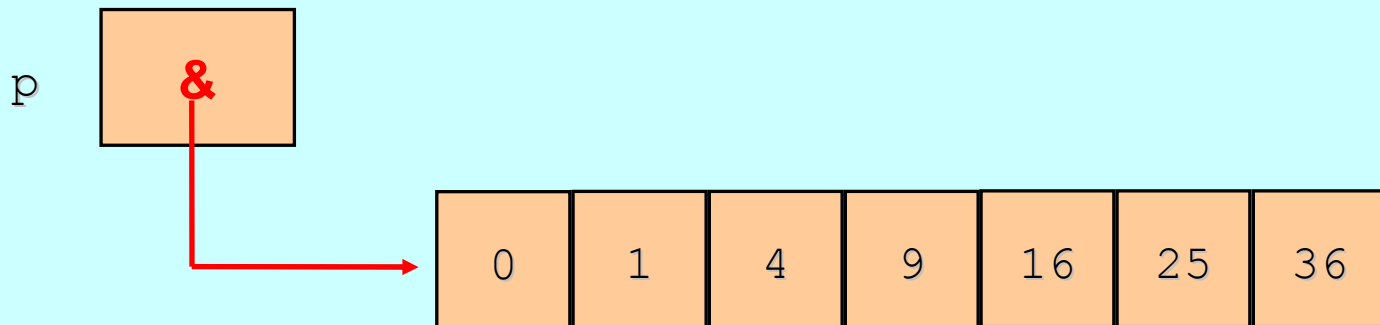
Arrays

1

C Foundation Part 2

- an aggregate initializer list can apparently be cast to an array type
 - ◆ known as a compound literal

```
int * p =  
    (int []) { 0, 1, 4, 9, 16, 25, 36 };
```

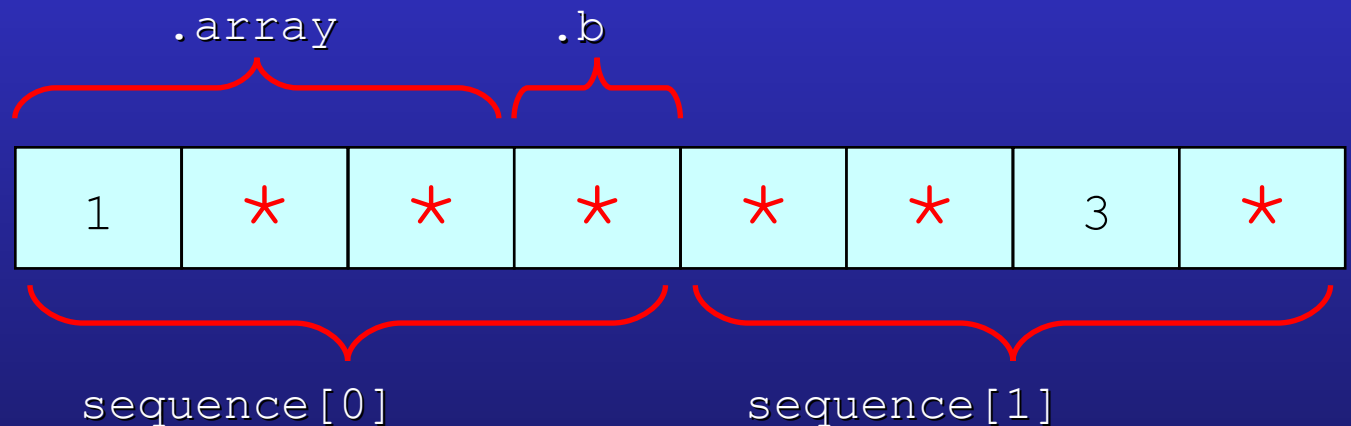


- arrays and struct may contain each other
 - ◆ [int] and .identifier designators can be combined

struct containing array

```
struct s
{
    int array[3];
    int b;
};
```

```
struct s sequence[] =
{
    [0].array = { 1 },
    [1].array[2] = 3
};
```



* default value

- only the top-level array decays into a pointer
 - ◆ the size of sub arrays remains part of the type

```
void print(int nrow, int matrix[2][3]);  
void print(int nrow, int matrix[ ][3]);  
void print(int nrow, int (*matrix)[3]);
```

equivalent

```
int main(void)  
{  
    int grid[2][3] = {{0,1,2},{3,4,5}};  
    ...  
    print(2, grid);  
}
```



```
void illegal(int matrix[ ][ ]) ...
```

5 pointer to array

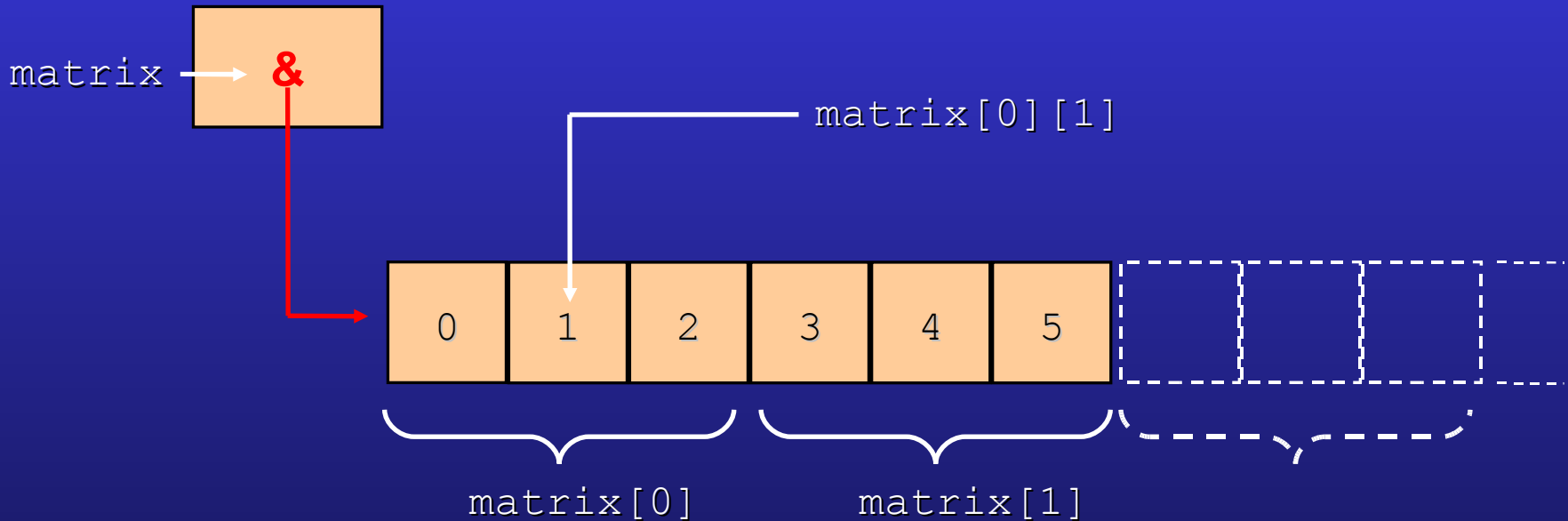
```
int (*matrix) [3]
```

```
int (*matrix) [3]
```

```
int (*matrix) [3]
```

matrix is a pointer
to zero, one, or more
array(s) of three ints

matrix points to a single chunk of memory



- an array of pointers can mimic a 2d array
 - ◆ each pointer points to an array
 - ◆ aka Illiffe vector aka dope vector

note this is `int*ragged[2]` and not `int (*ragged) [2]`

```
void print(int nrows, int ncols, int * ragged[2]);  
void print(int nrows, int ncols, int * ragged[ ]);  
void print(int nrows, int ncols, int * * ragged);
```

equivalent

```
int main(void)  
{  
    int vec1[] = { 0, 1, 2 };  
    int vec2[] = { 3, 4, 5 };  
    int * grid[2] = { vec1, vec2 };  
  
    print(2, 3, grid);  
}
```

array of pointers

7

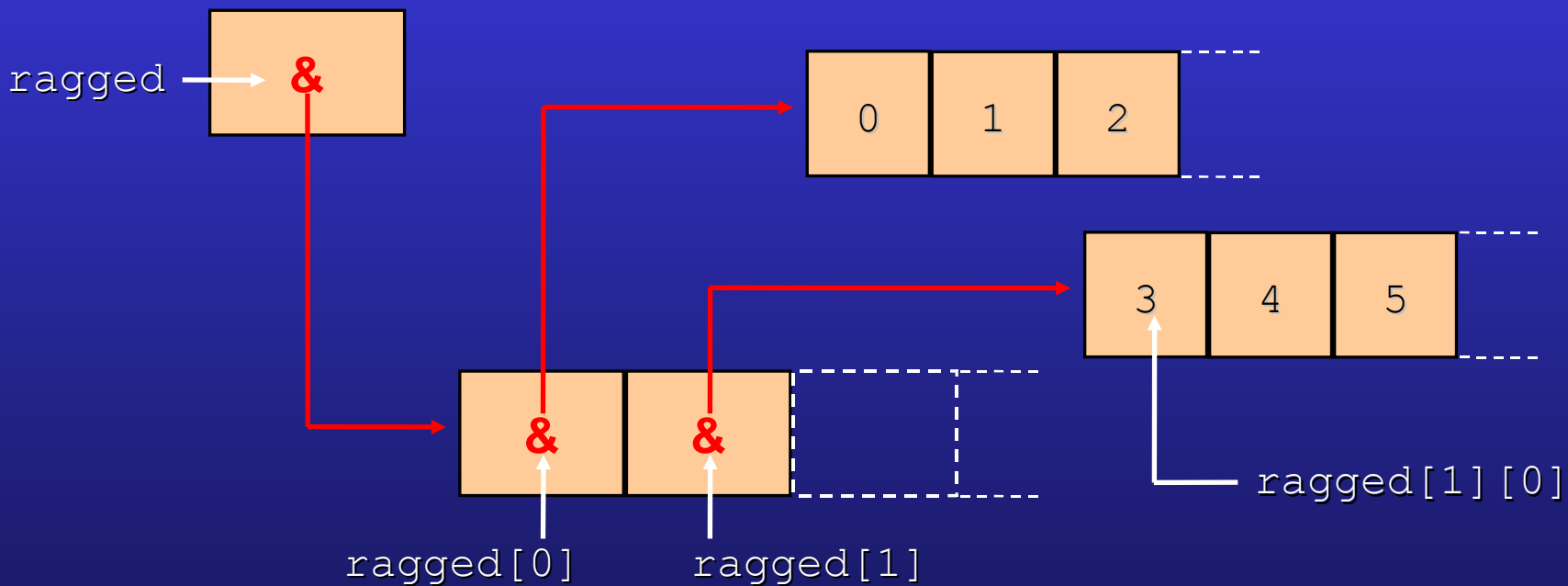
```
int * ragged[]
```

```
int * ragged[]
```

```
int * ragged[]
```

```
int * ragged[]
```

ragged is an array of pointers to zero, one, or more int(s)



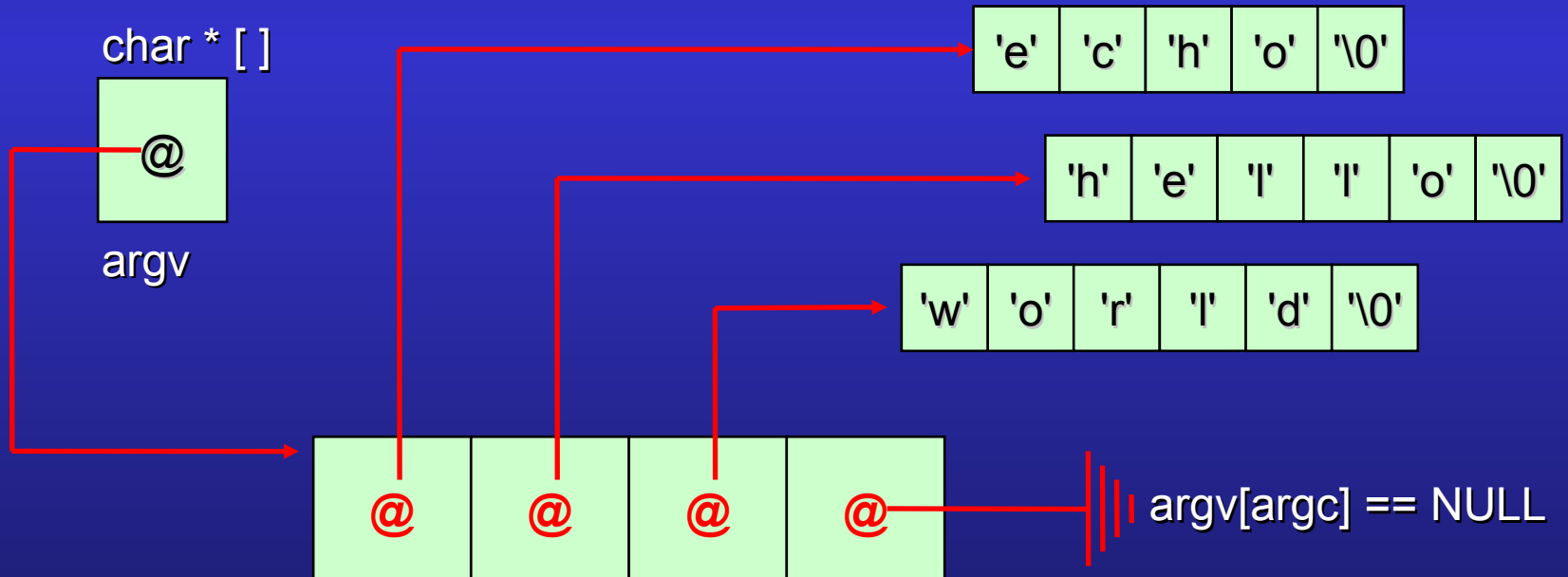
ragged array example

echo.c

```
#include <stdio.h>

int main(int argc, char * argv[])
{
    for (int at = 0; at != argc; at++)
        printf("%s ", argv[at]);
    putchar('\n');
}
```

>echo hello world



variable length arrays

- VLA's - four restrictions ✓

```
void vla(int n)
{
    int ok[n];

    static int g[n];

    extern int f[n];

    struct tag
    {
        int (*y)[n];
        int x[n];
    };

    extern int size;
    int array[size];
```

An object with static storage duration cannot have a variable length array type

An object with linkage cannot have a variable length array type

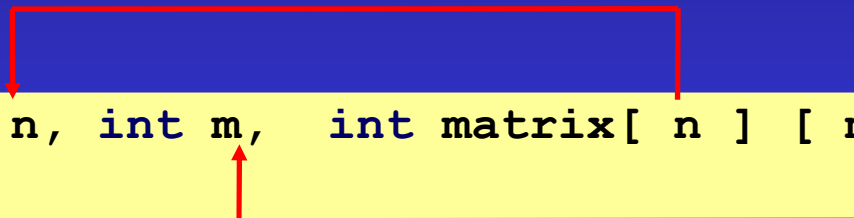
An identifier other than a simple identifier cannot have a variable length array type

A file scope identifier cannot have a variable length array type

- VLA's make multi-dimensional array parameters much more useful and reusable

```
void print(int n, int m, int matrix[ n ] [ m ] );
```

```
void print(int n, int m, int matrix[ n ] [ m ] )  
{  
    ...  
}
```



n and m must be declared before matrix

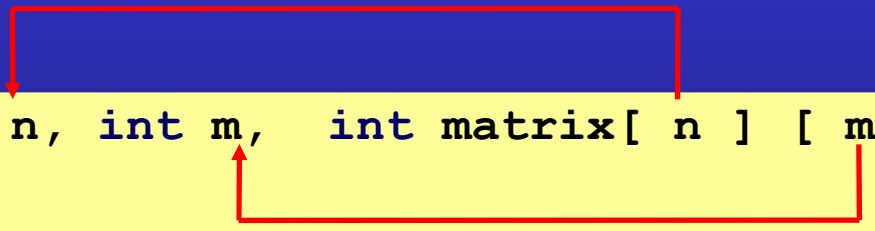
```
int main(void)  
{  
    int matrix[][3] = {{ 0, 1, 2 }, { 3, 4, 5 }};  
    print(2, 3, matrix);  
}
```

- 6.7.5.2 Array declarators
 - ◆ para 4 – If the size is * instead of being an expression, the array type is a variable length array type of unspecified size, which can only be used in declarations with function prototype scope

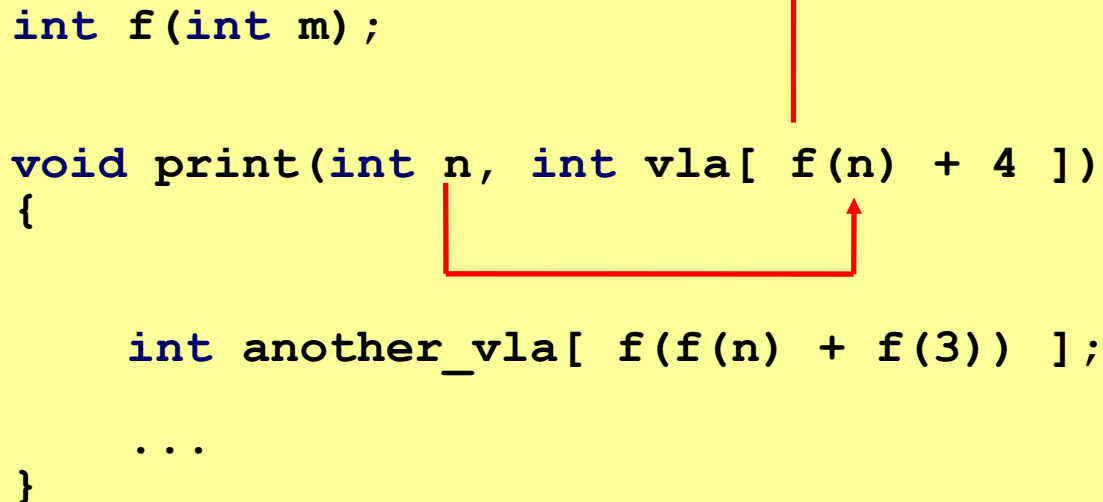
```
void print(int n, int m, int matrix[ * ] [ * ] );
```



```
void print(int n, int m, int matrix[ n ] [ m ] )  
{  
    ...  
}
```



- 6.7.5.2 Array declarators
 - ◆ para 5 – if the size is an expression that is not an integer constant expression ... each time it is evaluated it shall have a value greater than zero.



```
int f(int m);  
  
void print(int n, int vla[ f(n) + 4 ])  
{  
  
    int another_vla[ f(f(n) + f(3)) ];  
  
    ...  
}
```

The image shows a C code snippet on a yellow background. Red arrows highlight the variable length array declarations. One arrow points from the function call `f(n)` in the `print` function signature to the `f` function definition. Another arrow points from the `f(n)` in the `another_vla` declaration to the `f` function definition. A third arrow points from the `f(3)` in the `another_vla` declaration to the `f` function definition.

- 6.5.3.4 The sizeof operator
 - ◆ para 2 – If the type of the operand is a variable length array type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant

```
void examples(int n, int vla[ f(n) ])  
{  
    ...sizeof vla  
    int fla[42];  
    ...sizeof fla  
}
```

compile-time evaluation

run-time evaluation

- 6.7.5.3 Function declarators
 - ◆ para 7 - If the keyword `static` also appears within the `[and]` of the array type derivation, then for each call to the function, the value of the corresponding argument *shall* provide access to the first element of an array with at least as many elements as specified by the size expression

```
void function(double f[static 16])  
{  
    ...  
}
```

f is non-null and points to at least 16 doubles

- 6.7.5.3 Function declarators

- para 7 – A declaration of a parameter as "array of type" shall be adjusted to "qualified pointer to type", where the type qualifiers (if any) are those specified within the [and] of the array type derivation



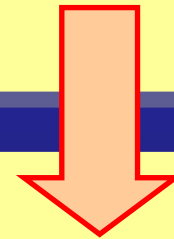
```
void function(double f[])  
{  
    f = 0;  
}
```



```
void function(double f[const])  
{  
    f = 0;  
}
```



```
void function(double * const f)  
{  
    f = 0;  
}
```



'adjustment'

- complete arrays can be typedef'd

```
typedef int table[42];  
void example(void)  
{  
    table t;  
    printf("%zd\n", sizeof(t));  
}
```

The diagram consists of red arrows. One arrow starts from the '42' in the typedef line and points to the 't' in the variable declaration 'table t;'. Another arrow starts from the 't' in 'table t;' and points to the 't' in 'sizeof(t)'. A third arrow starts from the 'sizeof(t)' expression and points to the text '42 * sizeof(int)' below the code block.

42 * sizeof(int)

- incomplete arrays can be typedef'd

```
typedef int table[];  
  
void example(void)  
{  
    table t1 = { 1,2 };  
    printf("%zd\n", sizeof(t1));  
    table t2 = { 1,2,3,4 };  
    printf("%zd\n", sizeof(t2));  
}
```

2 * sizeof(int)

4 * sizeof(int)