Appendix

bit twiddling

- sometimes you want to use an integer because of the bits it comprises
  - ~expression inverts the bits: 0-bit ← ~ → 1-bit
  - left-shift: integer-expression << bit-count
  - right-right: integer-expression >> bit-count

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

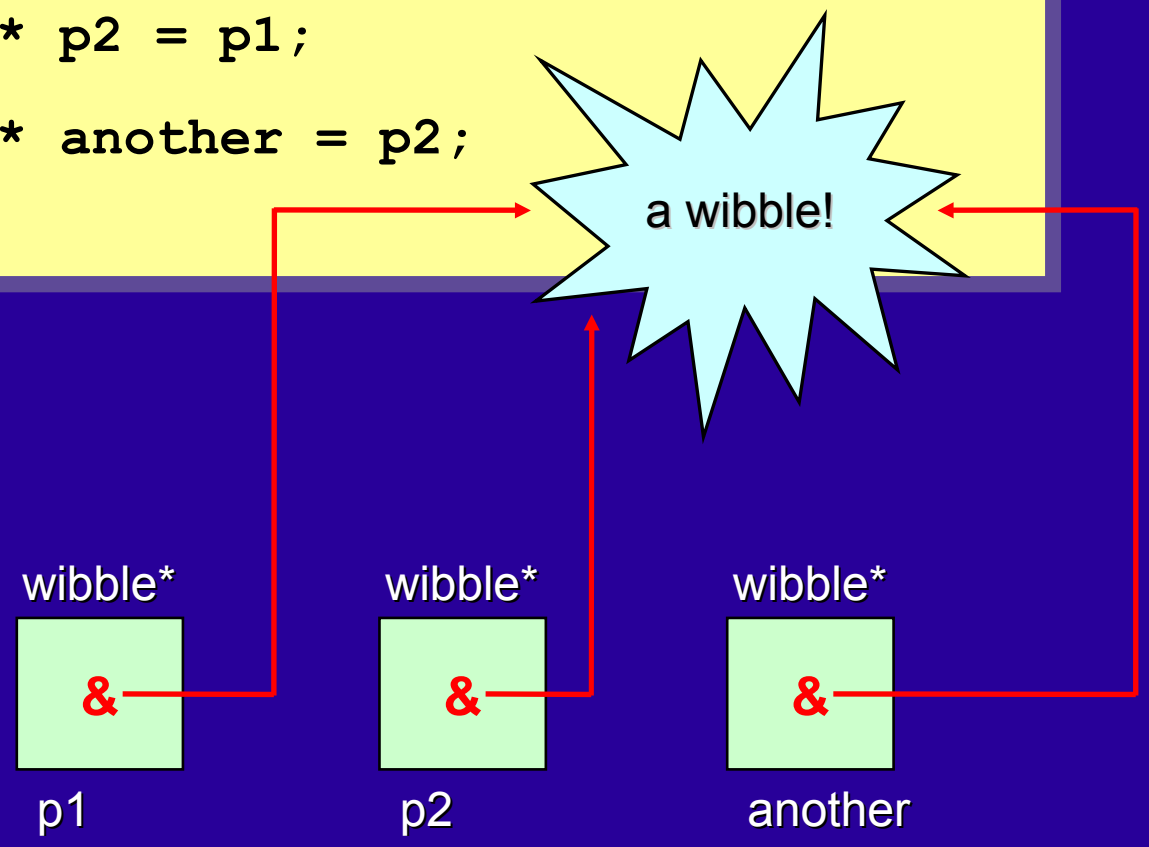
^	0	1
0	0	1
1	1	0

if the bit-count is negative or greater than or equal to the width of the left operand the behaviour is undefined

sharing

- many pointers can point to the same object
  - information can be shared across a program

```
void function(wibble * p1)
{
    ...
    wibble * p2 = p1;
    ...
    wibble * another = p2;
    ...
}
```



arrays and structs

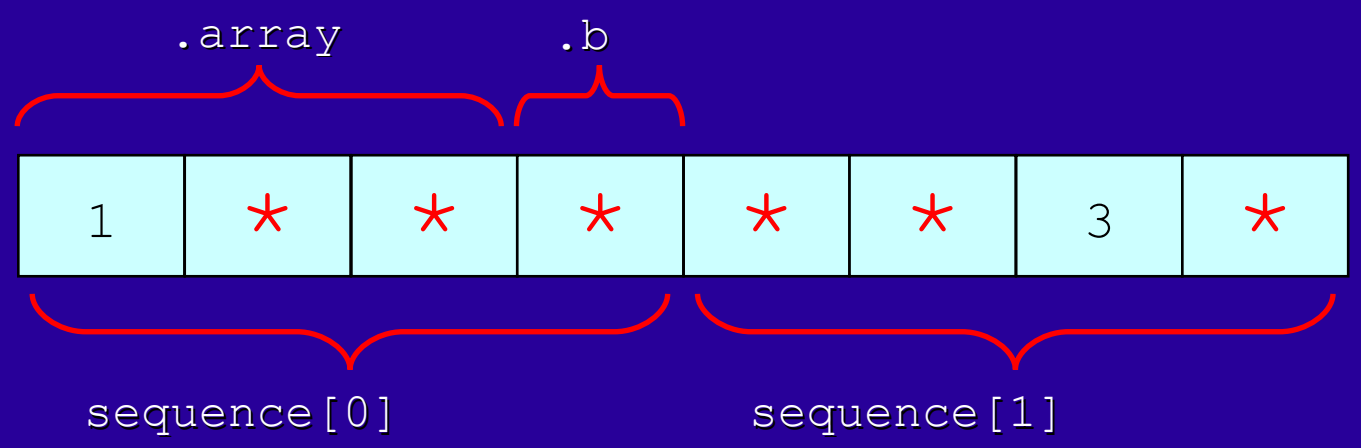
- arrays and struct may contain each other
  - [int] and .identifier designators can be combined

c99

struct containing array

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

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



* default value

rectangular arrays

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

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

equivalent

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

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

argument picture

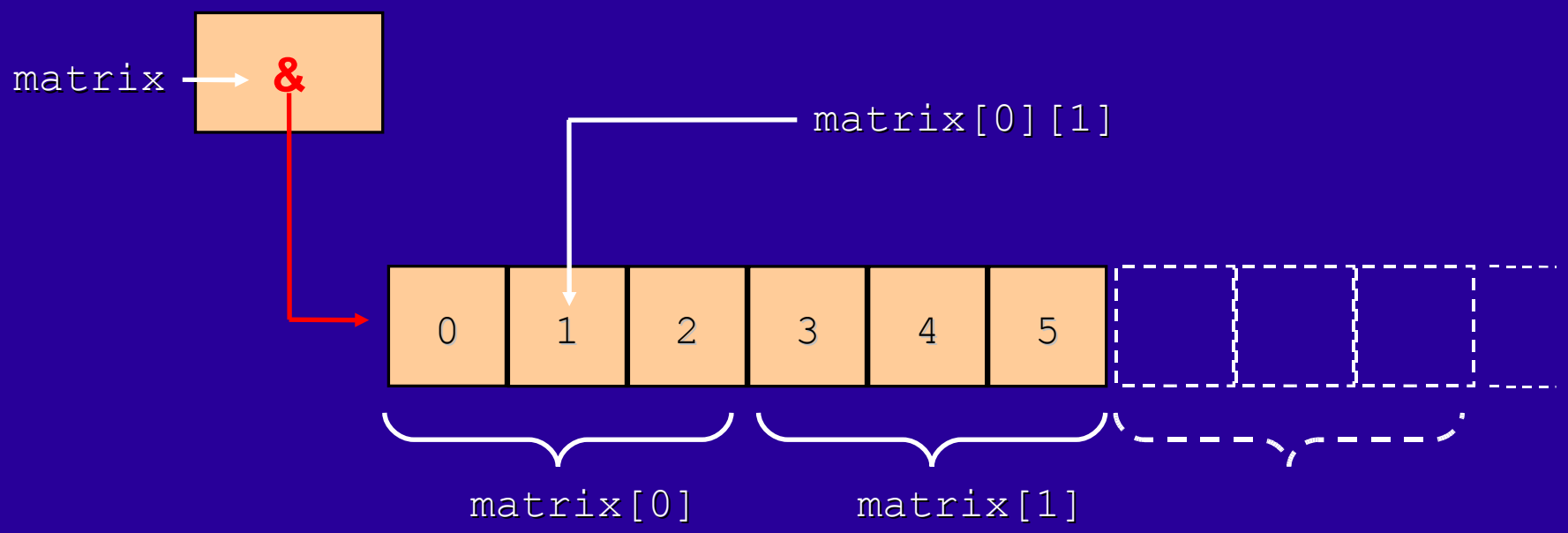
```
int (*matrix)[3]
```

```
int (*matrix)[3]

int (*matrix)[3]
```

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

matrix points to a *single* chunk of memory



ragged arrays

- 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);
}
```

argument picture

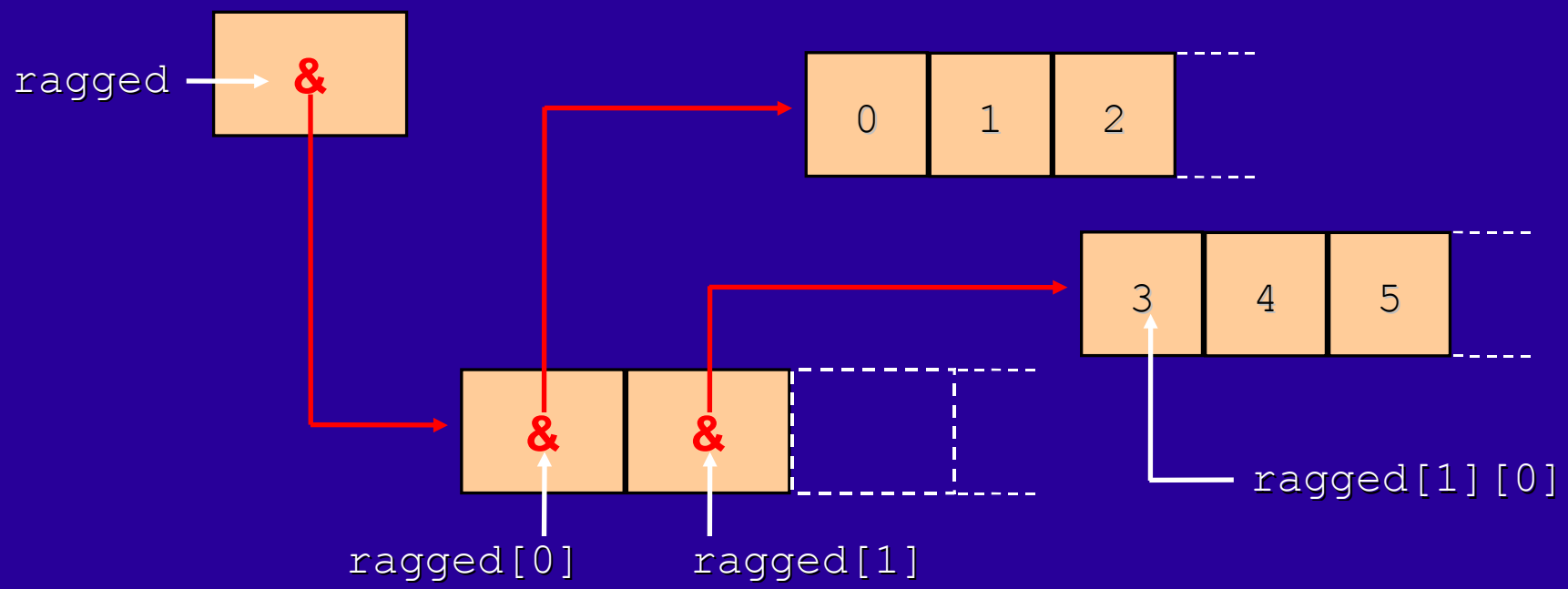
```
int * ragged[]
```

```
int * ragged[]

int * ragged[]

int * ragged[]
```

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



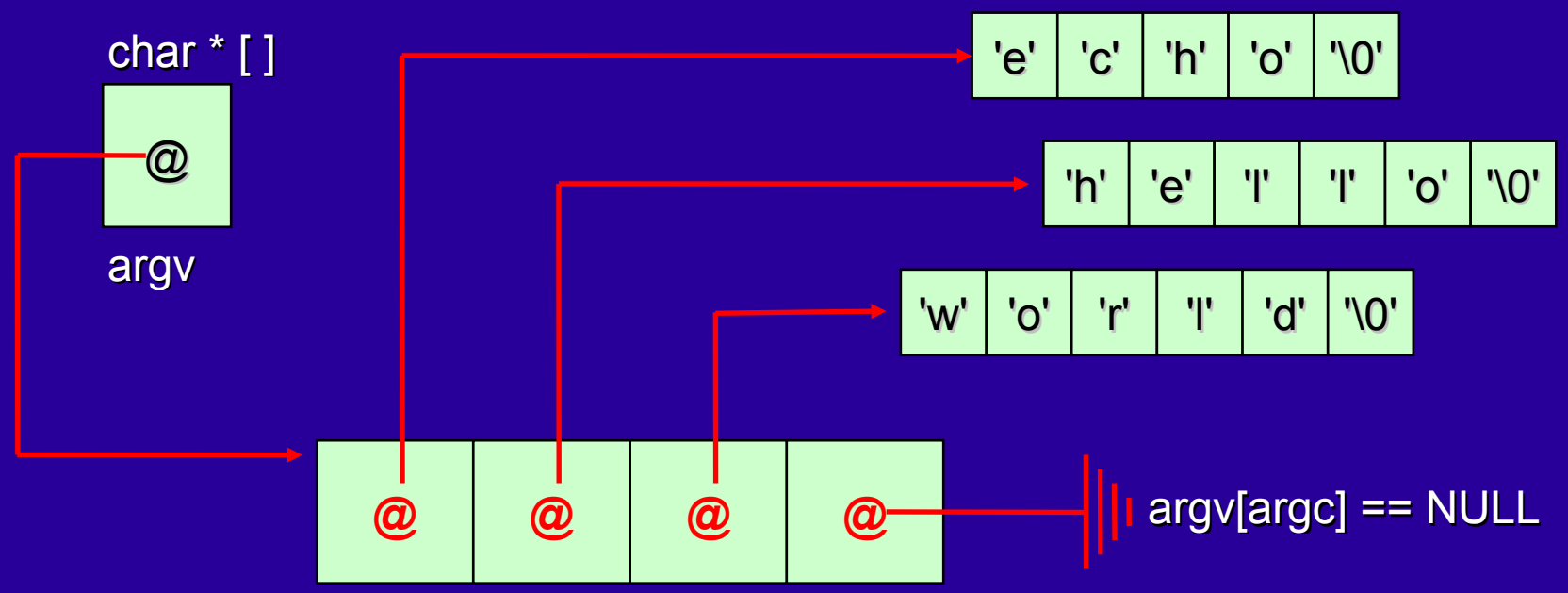
ragged array example

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

- make multi-dimensional array parameters much more useful and reusable
  - several restrictions (see notes)

c99

```
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);
}
```

#line

- **#line changes the apparent line number and name of the source file**
  - **useful mostly to code generators**

s-char == any character except " \ or newline

```
# line digit-sequence "s-chars"opt
```

the line number (required)

the filename (optional)

example.c

```
#line 999 "fake.c"

bug++;
```

```
>gcc example.c
fake.c:1000: error: syntax error before '++' token
```

testability

- functions using internally wired file scope data are hard to test
  - the Parameterize from Above (PfA) pattern can help to loosen the tight coupling

```
void not_easy_to_test(void)
{
    printf(...);
    printf(...);
    ...
}
```

?

```
void easier_to_test(FILE * stream)
{
    fprintf(stream, ...);
    fprintf(stream, ...);
    ...
}
```

✓

```
void not_easy_to_test(void)
{
    easier_to_test(stdout);
}
```

a global

testability

better – but still not a *unit* test

```
size_t fsize(FILE * stream);
int freadall(FILE * stream, char buffer[], size_t n);

void example_test(void)
{
    FILE * stream = fopen("test.txt", "w");
    easier_to_test(stream);
    fclose(stream);

    stream = fopen("test.txt", "r");
    size_t n = fsize(stream);
    char * actual = malloc(n);
    freadall(stream, actual, n);
    fclose(stream);

    const char * expected = "42";
    bool same = strcmp(expected, actual) == 0;
    free(actual);
    assert(same);
}
```

- a unit test cannot touch the file system
  - use PfA again; make the parameter a function ptr
  - note the external API is again unchanged

```
void not_easy_to_test(FILE * stream)
{
    int result = fputc('4', stream);
    ...
}
```

```
void easy_to_test(void * v, int putter(int, void *))
{
    int result = putter('4', v);
    ...
}
```

```
int fputc_stub(int ch, void * stream)
{
    return fputc(ch, (FILE*)stream);
}
```

```
void not_easy_to_test(FILE * stream)
{
    easy_to_test(stream, fputc_stub);
}
```

testability

- now a proper unit test
  - isolated, repeatable, automatable, fast

```
void easy_to_test(void * v, int putter(int, void *))
{
    int result = putter(v, ch);
    ...
}
```

```
int my_putter(int c, void * v)
{
   char * ptr = v;
   *ptr = c;
   return c;
}

void example_test(void)
{
    char put[] = { '\0' };
    easy_to_test(put, my_putter);
    assert(put[0] == '4');
}
```

testability

- testing
  - no one aspect of software development is more encompassing than testing
  - make unit testing compulsory
- dependencies
  - unit testing helps to ensure active management of dependencies
  - it helps to reveal the dependency horizon
  - Parameterize from Above
  - Don’t Talk To Strangers

TESTING TESTING

Appendix

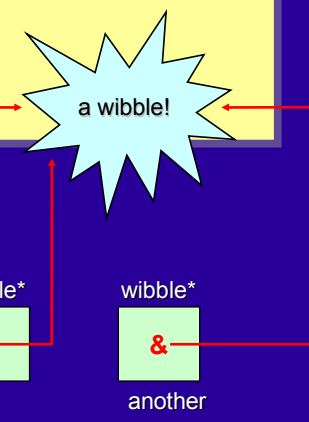
Here is an example of using bit twiddling to count the number of bits in an unsigned int that are set to one:

```
int bit_count(unsigned int n)
{
    int count = 0;
    while (n) {
        count += n & 0x1U;
        n >>= 1;
    }
    return count;
}
```

sharing

- many pointers can point to the same object
  - information can be shared across a program

```
void function(wibble * p1)
{
    ...
    wibble * p2 = p1;
    ...
    wibble * another = p2;
    ...
}
```



arrays and structs

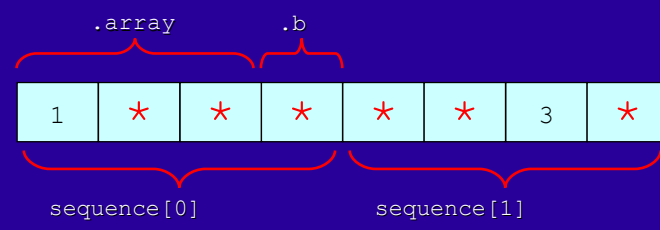
- arrays and struct may contain each other
  - [int] and .identifier designators can be combined

struct containing array

c99

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

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



* default value

rectangular arrays

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

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

equivalent

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

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

Note that the memory layout for a 2d array is a single block of memory.

To understand the third function prototype for pass remember that a pointer to something uses the same syntax as a pointer to an array of something. In:

void print(int nrows, int (*matrix)[3]);

matrix is a pointer to an int[3] or to the first element in an array of int[3].

argument picture

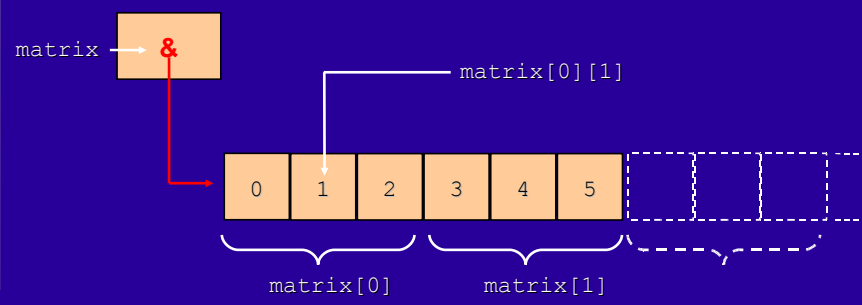
```
int (*matrix)[3]
```

```
int (*matrix)[3]

int (*matrix)[3]
```

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

matrix points to a *single* chunk of memory



ragged arrays

- 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);
}
```

argument picture

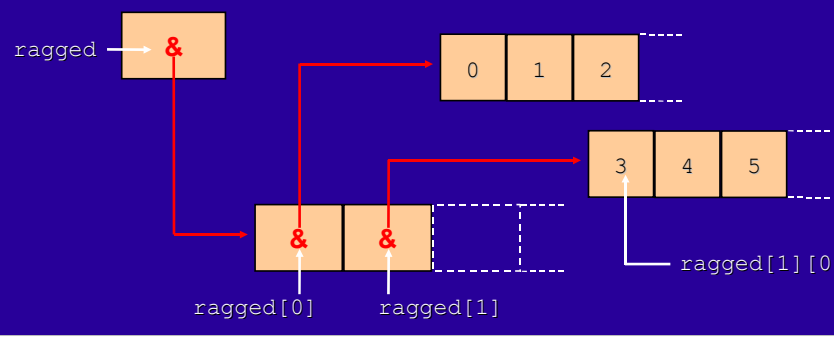
```
int * ragged[]
```

```
int * ragged[]

int * ragged[]

int * ragged[]
```

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



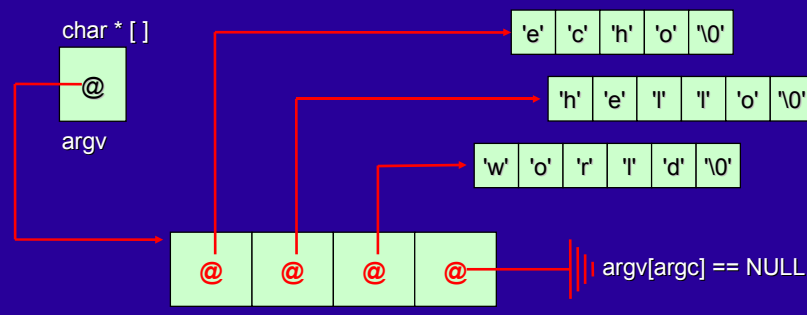
ragged array example

echo.c

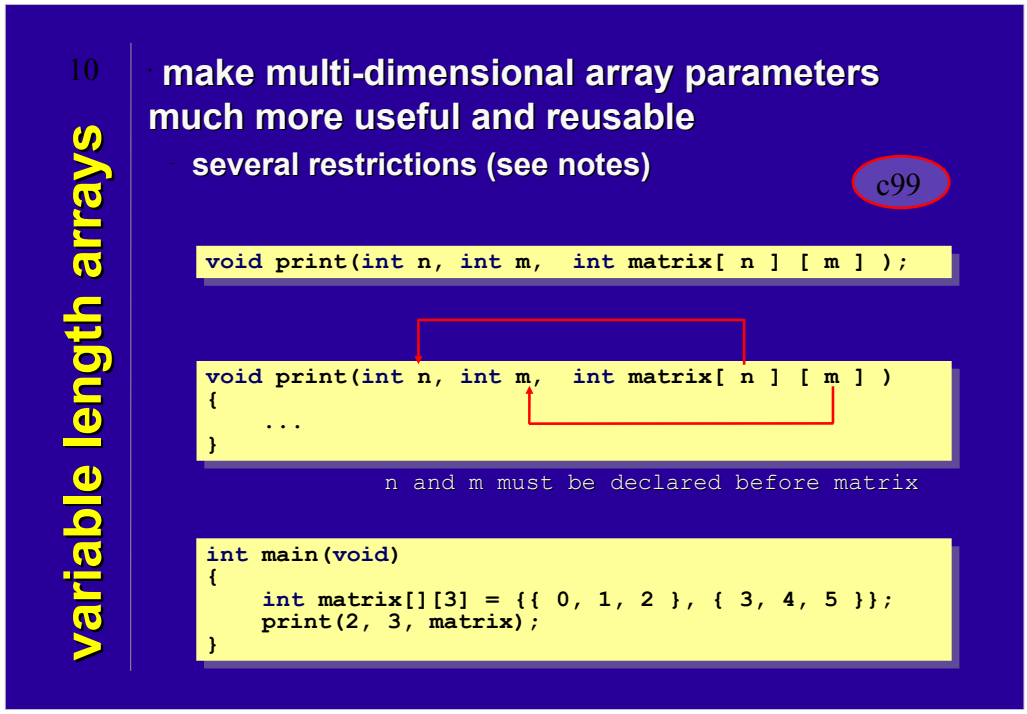
```
#include <stdio.h>

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

```
>echo hello world
```



Note that argv[0] is often the name of the program itself by strictly speaking this is not required by the standard.



Variable length arrays (VLA's) were added in C99.

- An object with static storage duration cannot have a variable length array type. However an object with static storage duration can have a pointer to 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 type.
- A file-scope identifier cannot have a variable length type.

```
extern int n;
int A[n];    // error 4.

void vla(int m)
{
    struct tag
    {
        int (*y)[m]; // error 3
        int x[m];    // error 3
    };
    extern int f[m];    // error 2
    static int g[m];// error 1
```

#line

- #line changes the apparent line number and name of the source file
  - useful mostly to code generators

s-char == any character except " \ or newline

```
# line digit-sequence "s-chars"opt
```

the line number (required)

the filename (optional)

example.c

```
#line 999 "fake.c"

bug++;
```

```
>gcc example.c
fake.c:1000: error: syntax error before '++' token
```

The printf function from <stdio.h> implicitly prints to the stdout FILE*. This makes a function that uses printf hard to test properly since its output is not easily accessible.

The code in the example is simplified. For example, malloc and fopen could both fail.