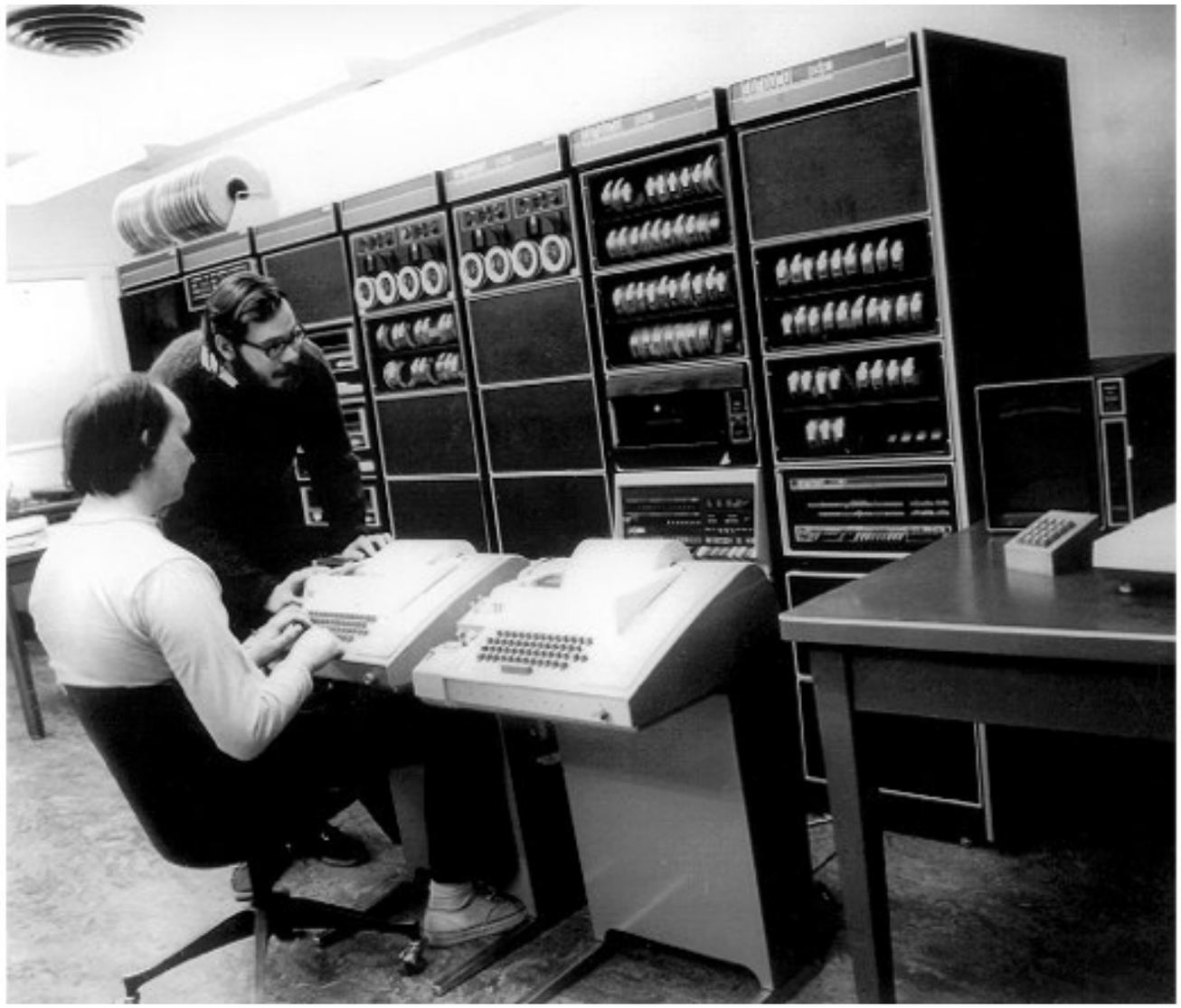
Arrays

"to get a deeper understanding of the language"

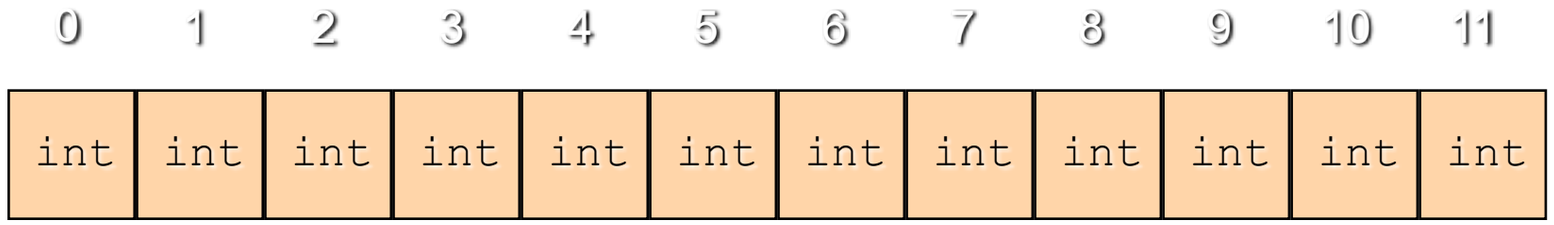


Deep C - a 3 day course
Jon Jagger & Olve Maudal

arrays

- an array is a fixed-size contiguous sequence of elements
- all elements have the same type
- default initialization when `static` storage class
- no default initialization when `auto` storage class

```
int days_in_month[12];
```



the type of `days_in_month` is `int[12]`

array initialization

- arrays support ={...} aggregate initialization
- syntax not permitted for assignment
- any missing elements are default initialized
- arrays cannot be initialized/assigned from another array

```
const int days_in_month[12] =
{
   31, // January
   28, // February
   31, // March
   ...
   31, // October
   30, // November
   31  // December
};
```

size is optional

- a trailing comma is allowed
- an empty list is not allowed (it is in C++)

[designators]

- arrays support [int] designators
- int must be a constant-expression

c99

```
enum { january, february, march, ...
   october, november, december };


const int days_in_month[] =
{
   [january] = 31,
   [february] = 28,
   [march] = 31,
   ...
   [october] = 31,
   [november] = 30,
   [december] = 31
};
```

these initializer list elements can now appear in any order

array indexing

- indexing is zero based
- indexing is *not* bounds-checked
- out of bounds access is *undefined*

```
int days_in_month[12];
```

```
printf("%d", days_in_month[january]);
```

```
printf("%d", days_in_month[-1]);
printf("%d", days_in_month[12]);
```

one beyond the end

- a pointer ***can*** point just beyond an array
- can't be dereferenced
- can be compared with
- can be used in pointer arithmetic

```
int array[42];
```

undefined

```
array[42]
```

not undefined

```
&array[42]
```

```
int * search(int * begin, int * end, int find)
{
    int * at = begin;
    while (at != end && *at != find) {
        at++;
    }
    return at;
}
```

array decay

- in an expression the name of an array "decays" into a pointer to element zero†
- arrays are *not* passed by copy as function arguments

these two declarations are equivalent

```
void display(size_t size, wibble * first);
void display(size_t size, wibble first[]);
```

```
wibble table[42] = { ... };
```

these two statements are equivalent

```
display(42, table);
display(42, &table[0]);
```

```
const size_t size =
    sizeof array / sizeof array[0];
```

†except in a `sizeof` expression

exercise

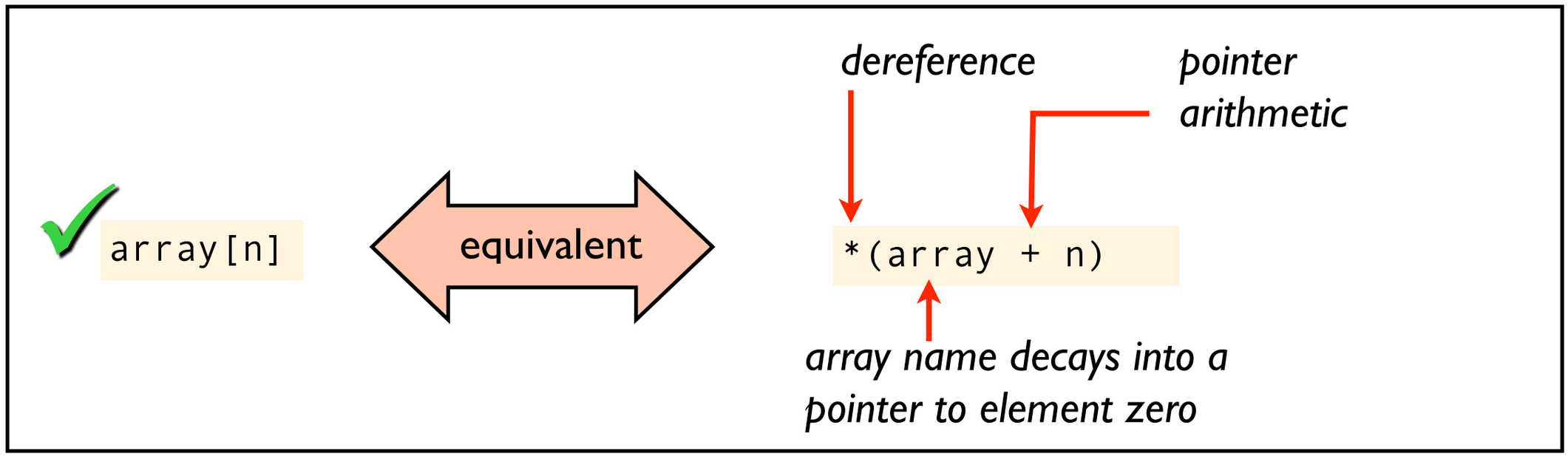
- what does the following program print?
- why?

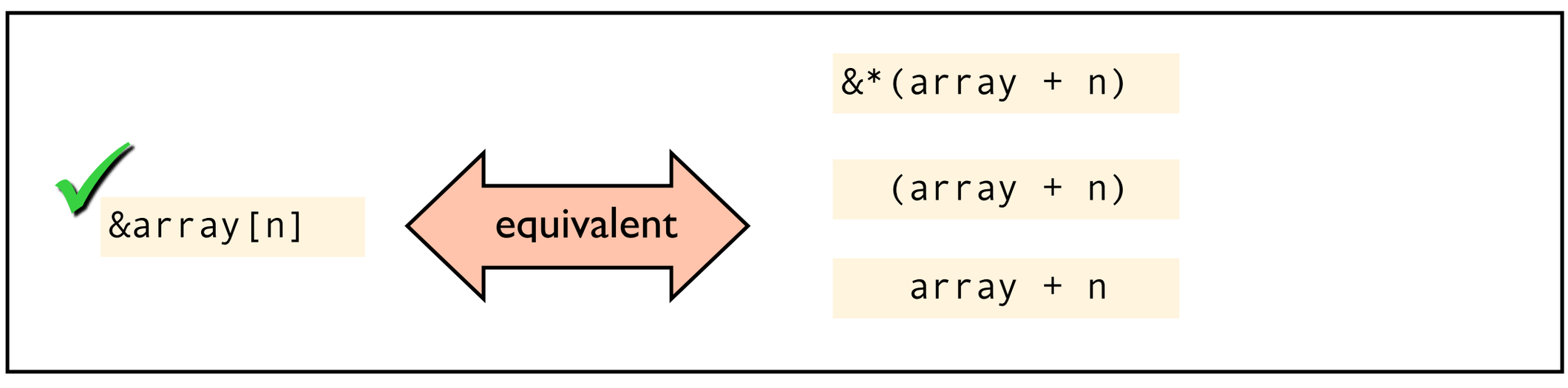
```
#include <stdio.h>

int main(void)
{
    int array[] = { 0,1,2,3 };
    int clone[] = { 0,1,2,3 };
    puts(array == clone
        ? "same" : "different");
    return 0;
}
```

pointers ←→ arrays

- array indexing is syntactic sugar
- the compiler converts `a[i]` into `*(a + i)`

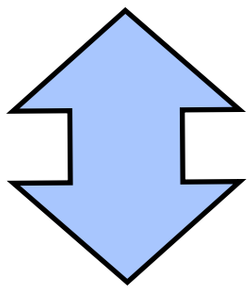




string literals

- strings are arrays of char
- automatically terminated with a nul character, `'\0'`
- a convenient string literal syntax

```
char greeting[] = "Bonjour";
```



equivalent

```
char greeting[] =
    { 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0' };
```

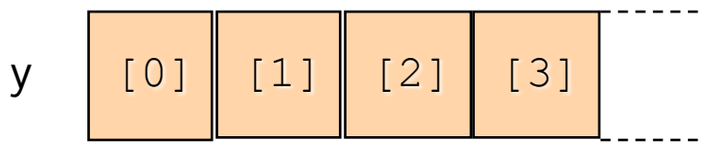
terminating nul character

pointers != arrays

- very closely related but *not* the same
- declare as a pointer → define as a pointer
- declare as an array → define as an array

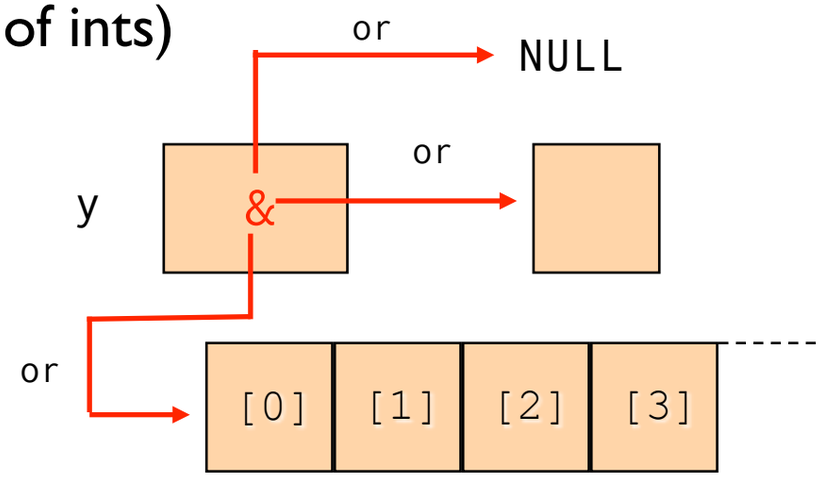
y is an array of int (of unspecified size)

```
extern int y[];
y[n]
```



y is a pointer to an int (or to an array of ints)

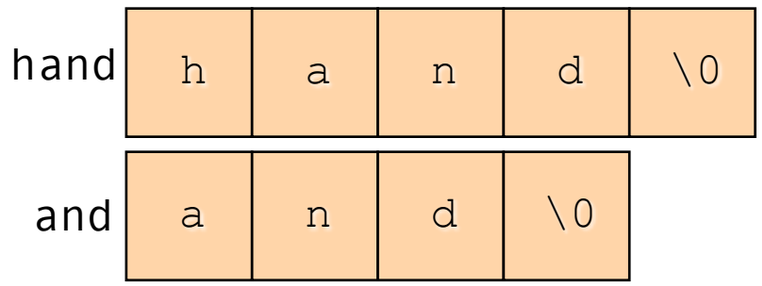
```
extern int * y;
y[n]
```



pointers != arrays

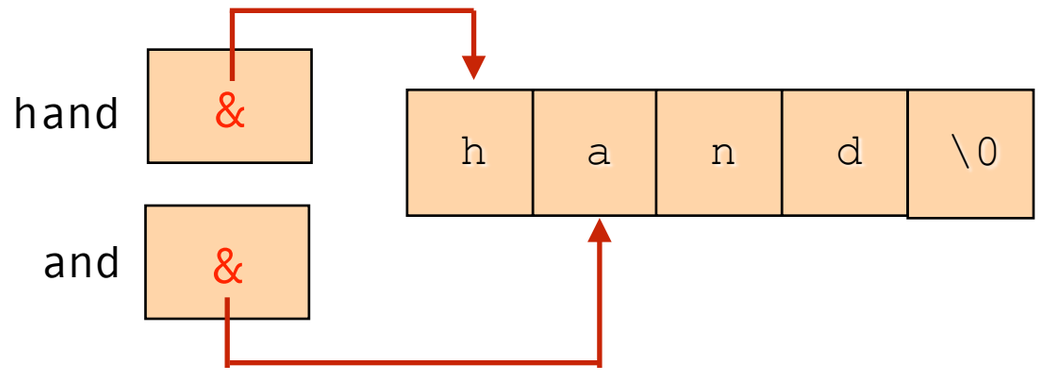
```
char hand[] = "hand";
```

```
char and[] = "and";
```




```
char * hand = "hand";
```

```
char * and = "and";
```

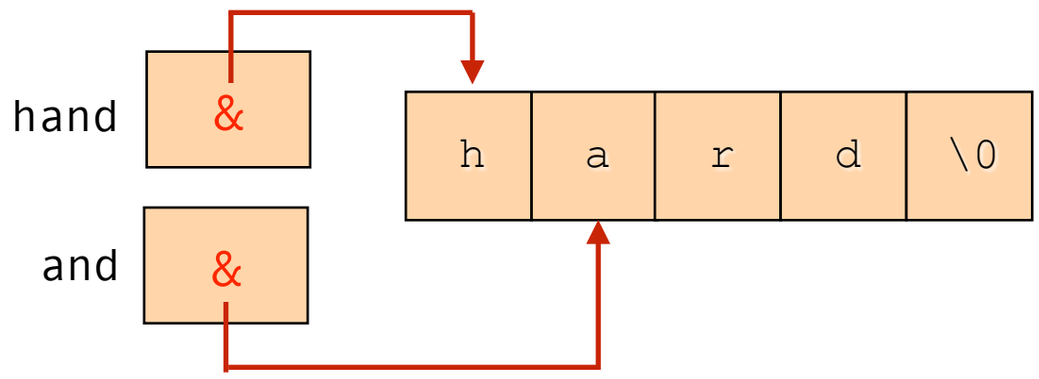


array literals are allowed to 'overlap'

pointers != arrays

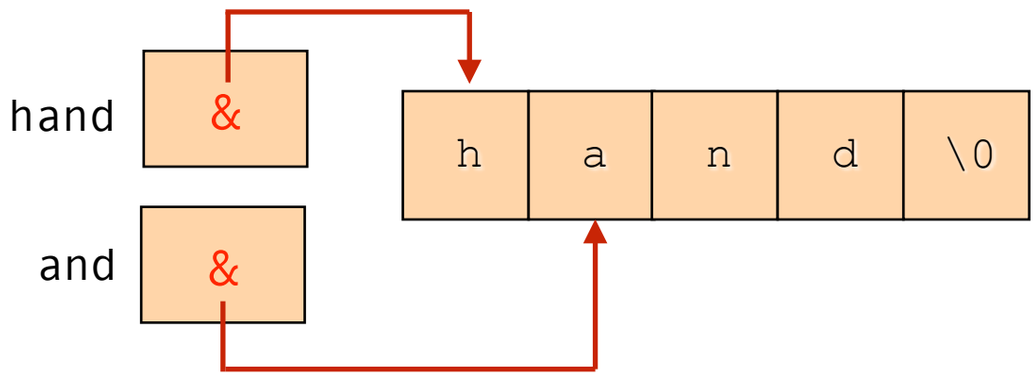
```
char * hand = "hand";
char * and = "and";

hand[2] = 'r';
```




```
const char * hand = "hand";
const char * and = "and";

hand[2] = 'r';
```



We know `a[n]` is syntactic sugar for `*(a + n)`

We also know that
`a+n == n+a`

Therefore
`*(a + n) == *(n + a)`

But
`*(n + a) == n[a]`

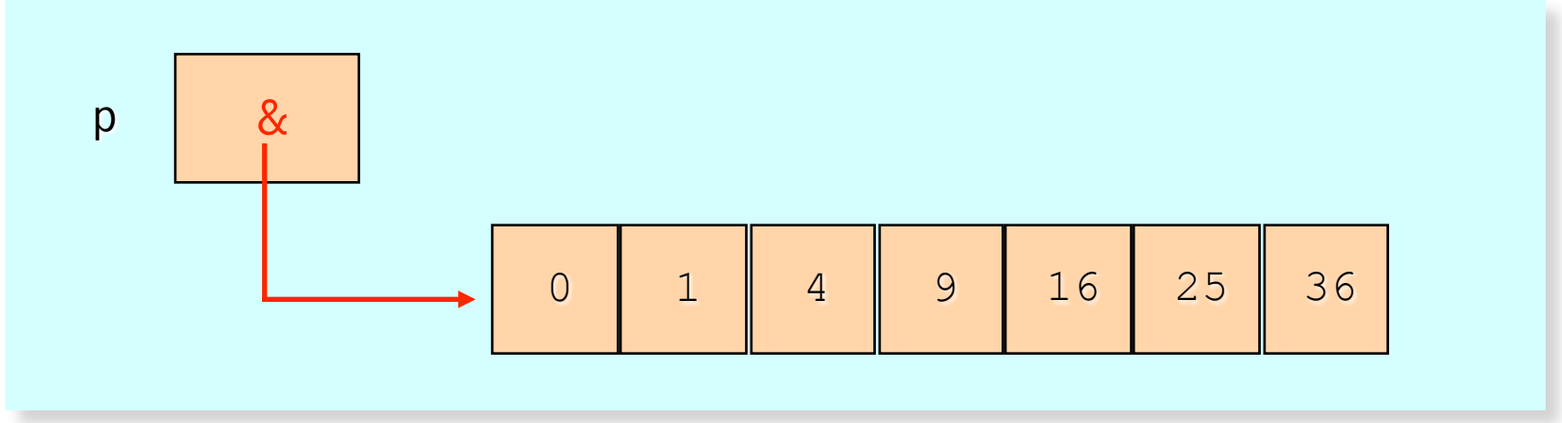
So....
`a[n] == n[a]`

array literal

- an aggregate initializer list can be "cast" to an array type
- known as a compound literal
- can be useful in both testing and production code

c99

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



summary

- an array is a contiguous block of memory
- designators (c99), eg [constant] = value
- array indexing is not checked
- out of bounds access caused undefined behaviour
- arrays and pointers closely related but are not the same
- in an expression the name of an array "decays" into a pointer to the first element
- compound literals, eg (type[]){...}