Pointers and Arrays

pointers

- a * in a declaration declares a pointer
  - read declarations from right to left
  - beware: the * binds to the identifier not the type

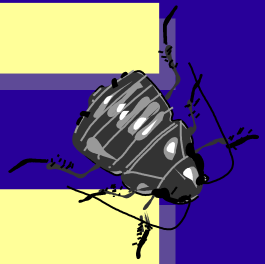
```
int * stream;

int * stream;

int * stream;
```

stream
is a
pointer
to an
int

```
int * pointer, value;
```

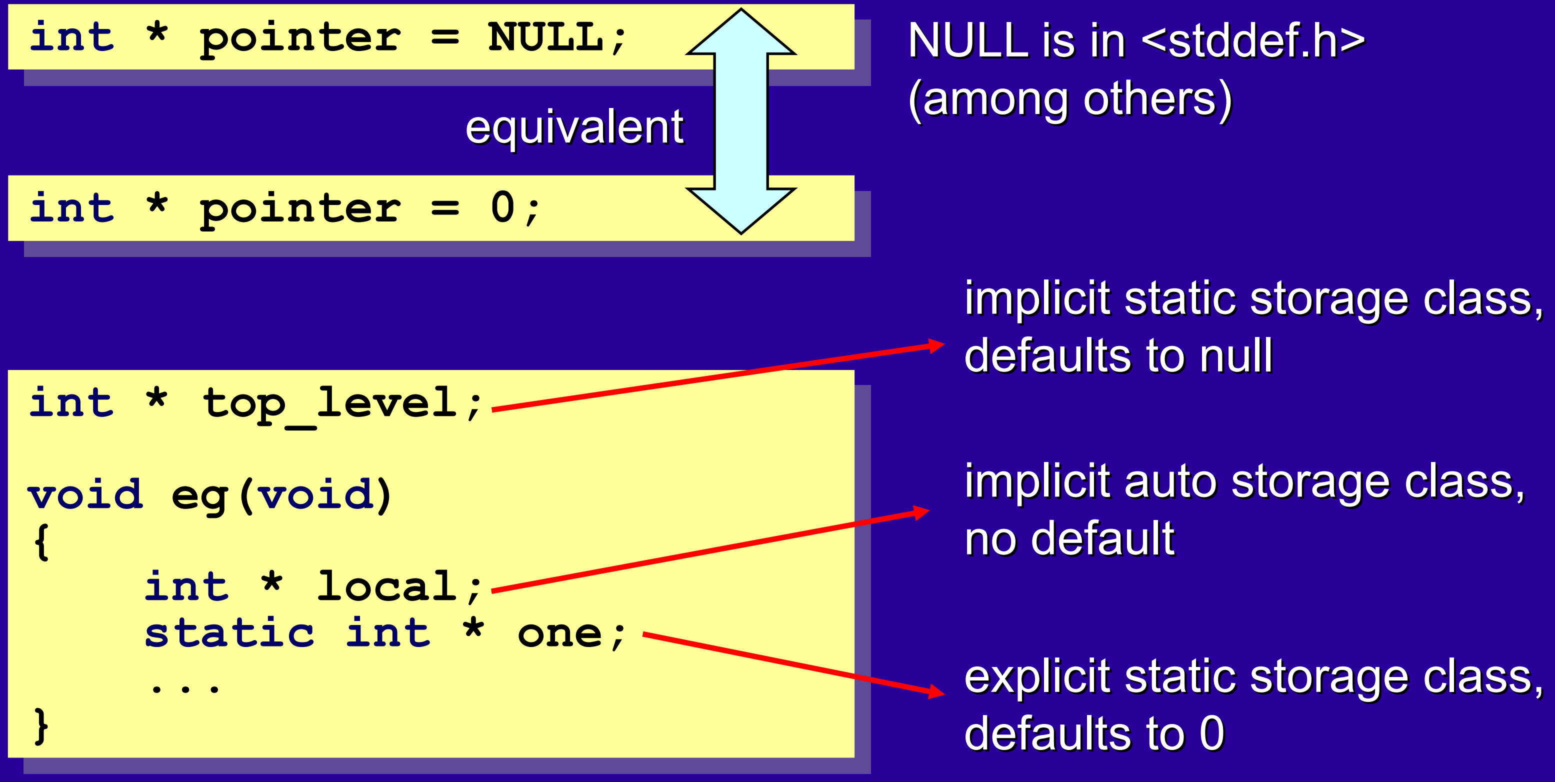
Equivalent to

```
int * pointer;
int value;
```



null pointer

- the null pointer (NULL or 0)
  - NULL never equals an objects address
  - the default for pointers with *static* storage class
  - no default for pointers with *auto* storage class



```
int * pointer = NULL;
```

equivalent

```
int * pointer = 0;
```

NULL is in <stddef.h> (among others)

```
int * top_level;

void eg(void)
{
    int * local;
    static int * one;
    ...
}
```

implicit static storage class, defaults to null

implicit auto storage class, no default

explicit static storage class, defaults to 0

pointer true/false

- a pointer expression can implicitly be interpreted as true or false
  - a null pointer is considered false
  - a non-null pointer is considered true

```
int * pos; ...
```

```
if (pos)
if (pos != 0)
if (pos != NULL)
```

equivalent

```
if (!pos)
if (pos == 0)
if (pos == NULL)
```

equivalent

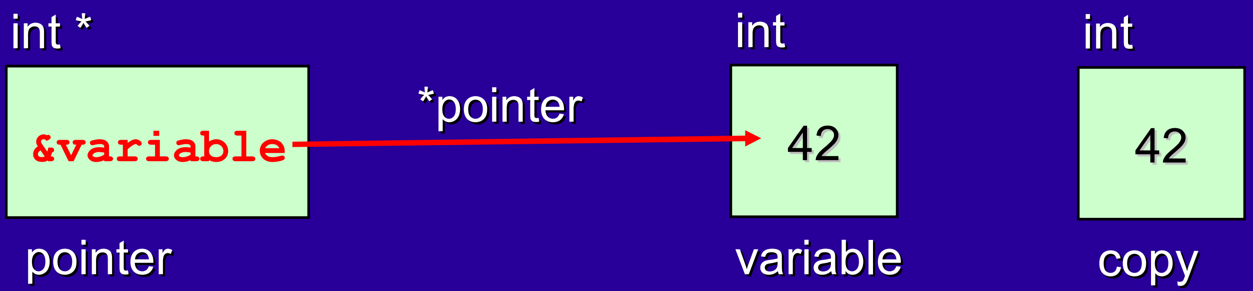
address-of/dereference

- unary & operator returns a pointer to its operand
- unary * operator dereferences a pointer
  - & and * are inverses of each other: *&x == x
  - *p is ***undefined*** if p is invalid or null

```
int variable = 42;
...
int * pointer = &variable;
...
int copy = *pointer;
```

* used in a declarator

* used in an expression



pointer fn arguments

```
#include <stdio.h>

void swap(int * lhs, int * rhs)
{
    int temp = *lhs;
    *lhs = *rhs;
    *rhs = temp;
}

int main(void)
{
    int a = 4;
    int b = 2;
    printf("%d,%d\n", a, b);
    swap(&a, &b);
    printf("%d,%d\n", a, b);
}
```

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];
```

0	1	2	3	4	5	6	7	8	9	10	11
int	int	int	int	int	int	int	int	int	int	int	int

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

1. a trailing comma is allowed
2. 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

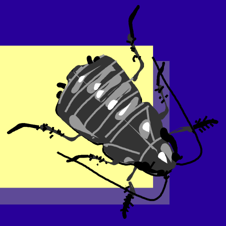
- **starts at zero and is not bounds-checked**
  - **out of bounds access is *undefined***

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

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

✓

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



pointers == arrays

- in an expression the name of an array "decays" into a pointer to element zero[†]
  - array arguments are not passed by copy

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

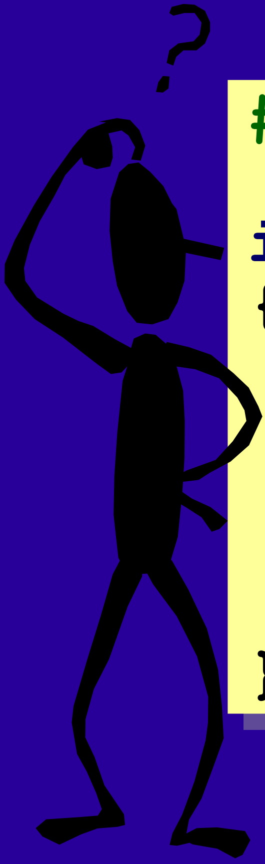
```
display(42, table);
display(42, &table[0]);
```

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

[†] except in a sizeof expression

exercise

- what does the following program print?
  - why?



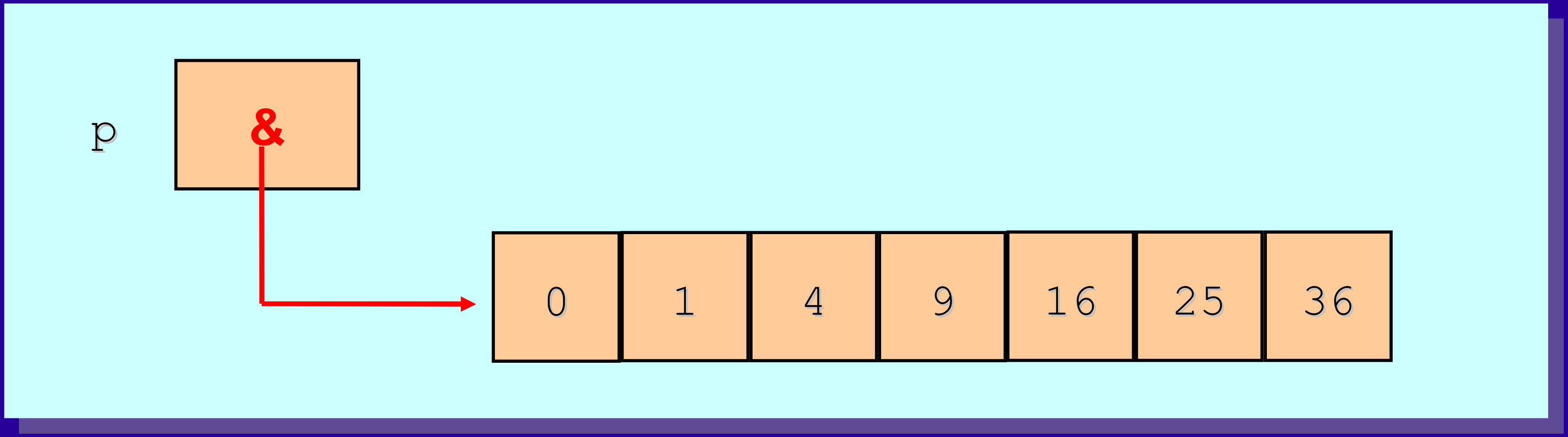
```
#include <stdio.h>

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

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

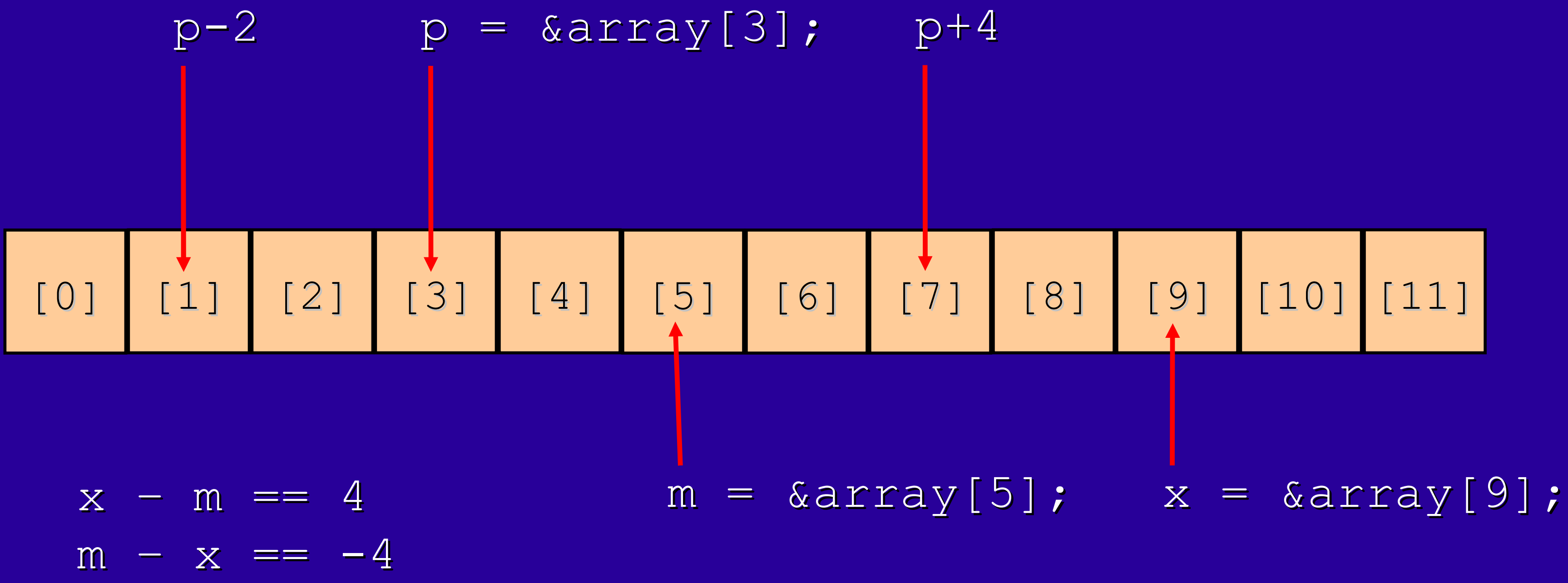
c99

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



pointer arithmetic

- is in terms of the target type, not bytes
  - p++ moves p so it points to the next element
  - p-- moves p so it points to the previous element
  - (pointer – pointer) is of type ptrdiff_t <stddef.h>



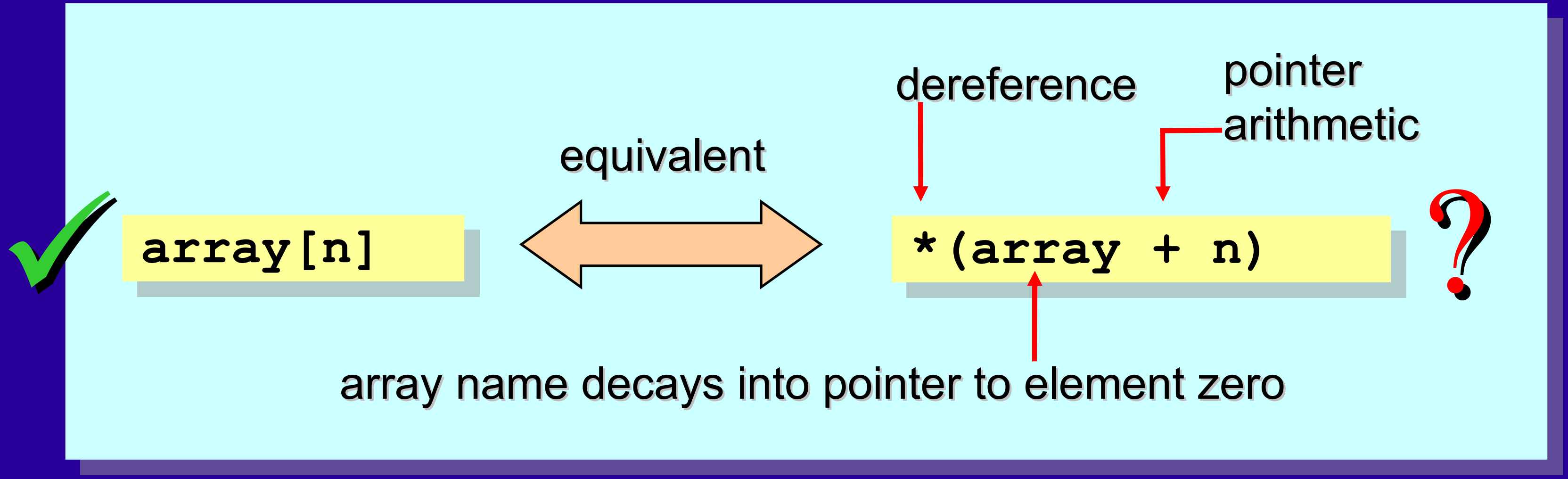
```
p-2        p = &array[3];    p+4
```

```
m = &array[5];    x = &array[9];
```

```
x - m == 4
m - x == -4
```

pointers == arrays

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



✓ `array[n]`

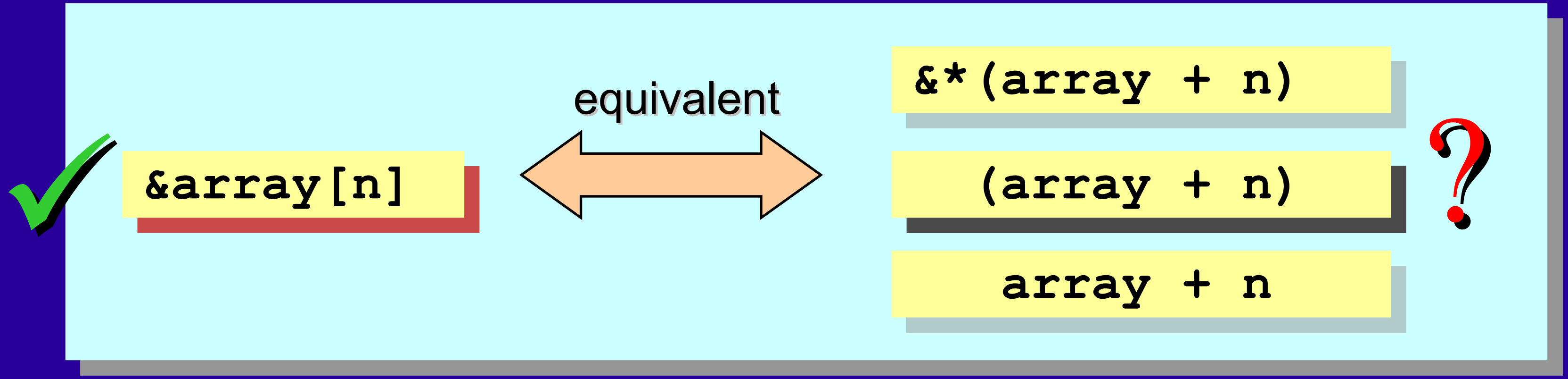
equivalent

`*(array + n)` ?

dereference

pointer arithmetic

array name decays into pointer to element zero



✓ `&array[n]`

equivalent

`&*(array + n)`

`(array + n)`

`array + n`

?

one beyond the end

- a pointer can point just beyond an array
  - can't be dereferenced
  - can be compared with

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

this is undefined

```
array[42]
```

this is not undefined

```
array + 42
```

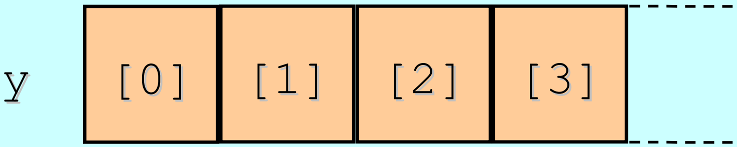
```
int * search(int * begin, int * end, int find)
{
    int * at = begin;
    while (at != end && *at != find) {
        at++;
    }
    return at;
}
```

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

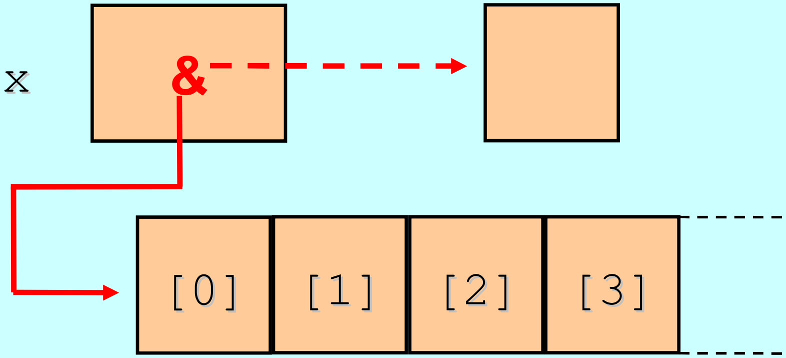
y is an array of int (of unspecified size)

```
...
extern int y[];
...
```



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

```
...
extern int * x;
...
```



syntax

syntax of declaration mirrors syntax of use

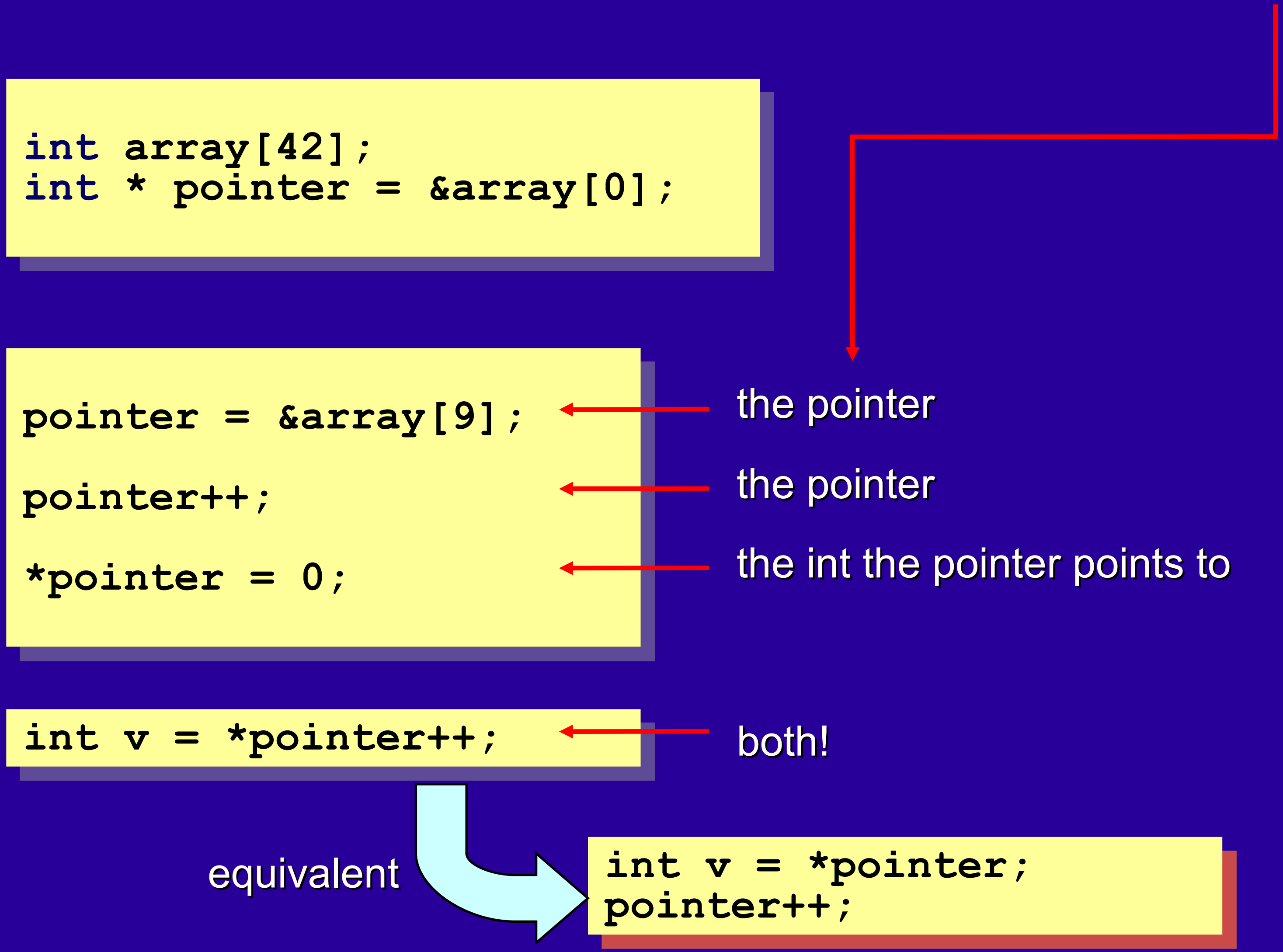
```
int days_in_month[12];

 ...days_in_month[at]...
```

```
       int *pointer = &variable;

int copy = *pointer;

           *pointer = 42;
```

```
        int identifier;

typedef int identifier;
```

pointer confusion

- be clear what your expression refers to
  - the pointer, the thing the pointer points to, both?



```
int array[42];
int * pointer = &array[0];
```

```
pointer = &array[9];

pointer++;

*pointer = 0;
```

the pointer

the pointer

the int the pointer points to

```
int v = *pointer++;
```

both!

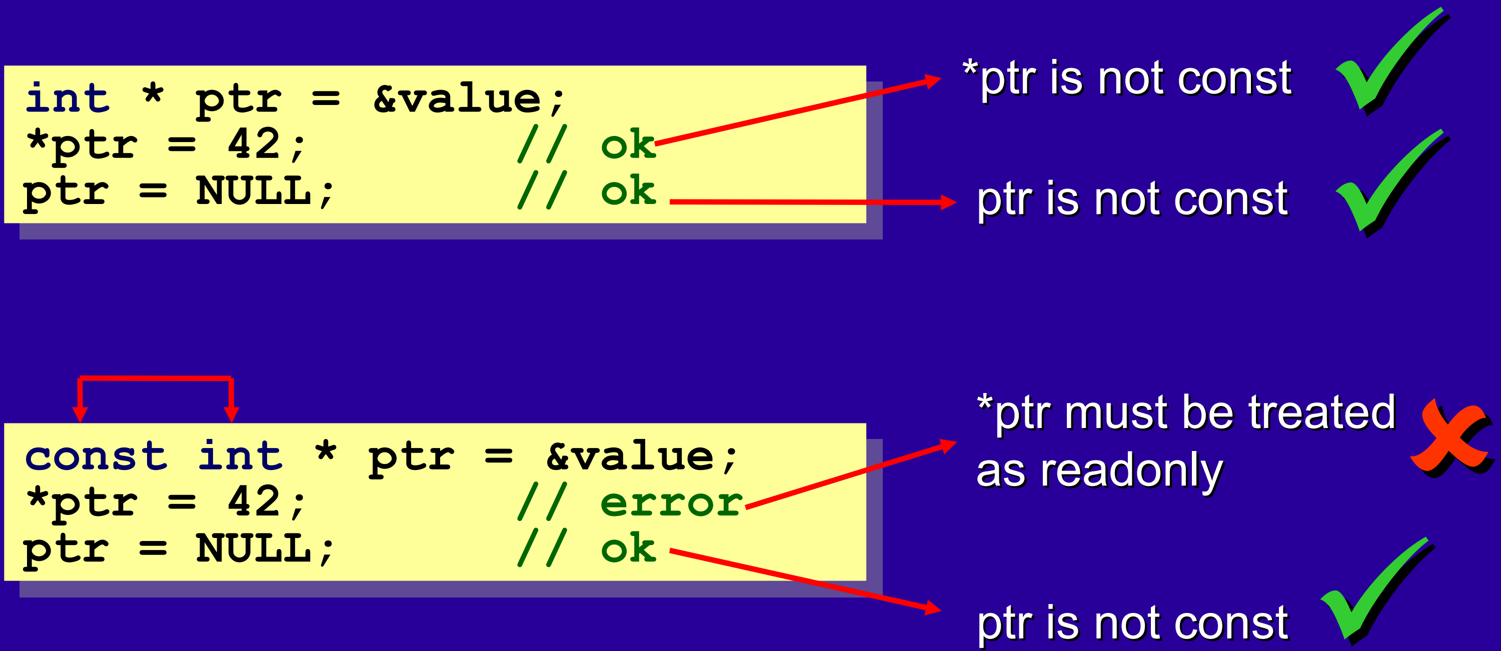
equivalent

```
int v = *pointer;
pointer++;
```

const + pointer

- another notorious source of confusion
  - again, be clear what your expression refers to
  - read const on the pointer's target as readonly

```
int value = 0;
```



```
int * ptr = &value;
*ptr = 42;          // ok
ptr = NULL;         // ok
```

*ptr is not const ✓

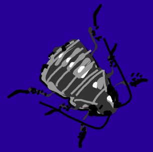
ptr is not const ✓

```
const int * ptr = &value;
*ptr = 42;          // error
ptr = NULL;         // ok
```

*ptr must be treated as readonly ✗

ptr is not const ✓

const + pointer



- another notorious source of confusion
  - again, be clear what your expression refers to
  - read const on the pointer's target as readonly

```
int value = 0;
```

```
int * const ptr = &value;
*ptr = 42;          // ok
ptr = NULL;         // error
```

*ptr is not const ✓

ptr is const ✗

```
const int * const ptr = &value;
*ptr = 42;          // error
ptr = NULL;         // error
```

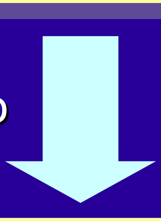
*ptr must be treated
as readonly ✗

ptr is const ✗

string literals

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

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



equivalent to

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

note the terminating null character

- strcpy: in <string.h>, copies a string
  - why is this a dangerous function to call?

array version

```
char * strcpy(char dst[], const char src[])
{
    int at = 0;
    while ((dst[at] = src[at]) != '\0')
        at++;
    return dst;
}
```

why are the parentheses needed?
why is the comparison with '\0' optional?
note the empty statement here

```
char * strcpy(char * dst, const char * src)
{
    char * destination = dst;
    while (*dst++ = *src++)
        ;
    return destination;
}
```

equivalent pointer version – very terse – typical of C code

void*

- a generic object pointer
  - any object pointer can be converted to a void* and back again
  - const void * is allowed
  - dereferencing a void* is not allowed

```
void * generic_pointer;
int * int_specific_pointer;
char * char_specific_pointer;
```

these compile

```
generic_pointer = int_specific_pointer;
int_specific_pointer = generic_pointer;
```

these don't compile

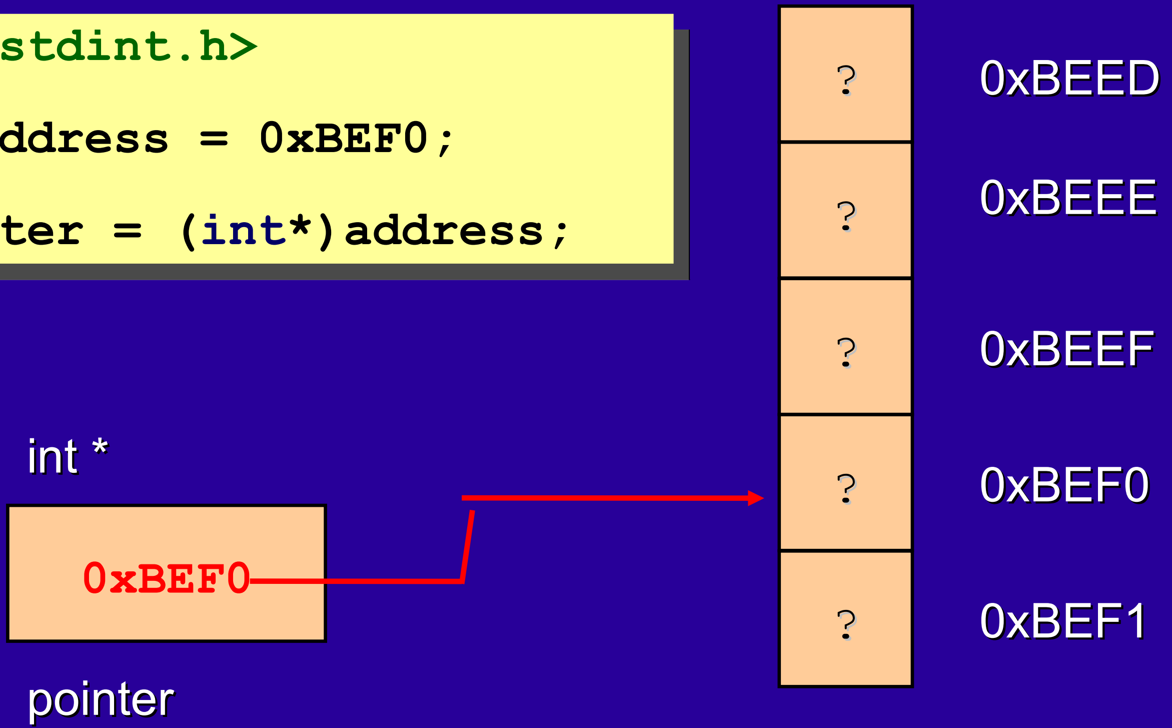
```
*generic_pointer;           ✗
generic_pointer[0];
```

int pointer

- any object pointer can safely be held in…
  - intptr_t - a signed integer typedef
  - uintptr_t - an unsigned integer typedef
  - both declared in <stdint.h>

c99

```
#include <stdint.h>

intptr_t address = 0xBEF0;

int * pointer = (int*)address;
```



restrict

- applies only to pointer declarations
  - type * restrict p → *p is accessed only via p in the surrounding block
  - enables pointer no-alias optimizations
  - a compiler is free to ignore it

c99

```
void f(int n, int * restrict p, int * restrict q)
{
    while (n-- > 0) {
        *p++ = *q++;
    }
}
```

```
void g(void)
{
    int d[100];
    f(50, d + 50, d);  // ok
    f(50, d + 1,  d);  // undefined-behaviour
}
```

dynamic memory

- dynamic memory can be requested using malloc() and released using free()
  - both functions live in <stdlib.h>
  - one way to create arrays whose size is not a compile-time constant

```
#include <stdlib.h>

void dynamic(int n)
{
    void * raw = malloc(sizeof(int) * n);
    if (raw != NULL)
    {
        int * cooked = (int *)raw;
        cooked[42] = 99;
        ...
        free(raw);
    }
}
```

?

see also calloc, realloc

summary

- pointers can point to...
  - nothing, i.e., null (expressed as NULL or 0)
  - a variable whose address has been taken (&)
  - a dynamically allocated object in memory (from malloc, calloc or realloc – don't forget to free)
  - an element within or one past the end of an array
- pointers and arrays share many similarities
  - but they are not the same and the differences are as important as the similarities
- strings are conventionally expressed as arrays of char (or wchar_t)
  - <string.h> supports many common string-handling operations
- be clear about what you can do with a pointer
  - respect restrict and be clear about what's const

As we will see shortly, a pointer such as int*stream; can also be a pointer to an array (of unspecified size) of ints. C has a single syntax for these two constructs.

The use of spaces (or not) around the * on a pointer declaration does not affect the meaning of the declaration but it can subtly alter the emphasis. Placing the * next to the type naturally emphasizes the identifier's type, and is common in C++ (it is the style used by Bjarne Stroustrup) which is a language dominated by types.

```
int* pointer; // type emphasis
```

In contrast, placing the * next to the identifier emphasizes the use of the identifier in an expression, and is common in C (it is the style used by Dennis Ritchie) which is a language dominated by expressions.

```
int *pointer; // expression emphasis
```

Placing spaces on both sides of the * has the effect of choosing not to emphasize either the type or the expression over the other, and is an increasingly common style in both C and C++.

```
int * pointer;
```

The null pointer is guaranteed not to be equal to the address of any object or function in your program.

Note that *&x and x are not completely identical since *&x is not an lvalue.

```
x = 42; // ok
*&x = 42; // illegal
```

With a trailing comma the array on the slide looks like this:

```
int days_in_month[12] = { 31,28,31, …  31,30,31,};
```

The main use of the trailing comma is as a convenience for programs generated from other programs.

You can use a trailing comma in hand written code in cases where the aggregate initialization list is likely to change. And similarly, choose not to use a trailing comma where the aggregate initialization list is not likely to change. For example, it's a reasonable bet that we aren't going to get a thirteenth month any time soon so days_in_months does not use a trailing comma.

Designators were added in C99.

An array element designator specifies not only the array index for the accompanying value but also the base index for subsequent values if they do not have a designator.

In other words this:

```
int x[4] = { [1]=99, 42, -4, [0]=0 };
```

is the same as this:

```
int x[4] = { [1]=99, [2]=42, [3]=-4, [0]=0 };
```

Because arrays are passed by pointer functions often need a second argument to indicate the size of the array the pointer points to. The size argument should be passed before the array argument since there is prior art for this ordering – main(int argc, char * argv[]). Sometimes the length of an array is not required because a special element value signifies the end of the array – for example '\0' in strings.

Note that the following is also permitted:

```
void display(wibble array[9], size_t size);
```

The integer constant 9 here is simply noise and does in any way shape or form indicate the size of the array.

The answer is it prints "different". The reason is that the == operator is not comparing the arrays! Remember, the name of an array decays into a pointer into its initial element. The == operator is therefore comparing the addresses of two arrays. The two arrays may contain the same content but they are nevertheless two different arrays and cannot live at exactly the same memory location.

The compound-literal feature was added in C99.

If one pointer is subtracted from another pointer and they do not point into the same object the behaviour is undefined.

The ptrdiff_t type is the signed version of the size_t type.

Because addition is commutative (that is to say: a+b == b+a) the designers of C decided to make *(array + n) be the same as *(n + array). The creates the unusual situation that array[n] can also be written as n[array] which is only of passing interest; array[n] is idiomatic whereas n[array] is not.

Things can get even more bizarre though! Given

```
int a[];
int b[];
int c;
```

then

```
a[b[c]] == c[b][a]
```

Note that a pointer pointing one beyond the last element is allowed but a pointer pointing just before the first element is not allowed.

```
int array[42];
int * beyond = &array[42];
int * nope = &array[-1];  // undefined behaviour
```

Here is a rewritten version of main that uses the new version of search:

```
int main(void)
{
    int v[] = { 32,13,19,56,17,42,89,43 };
    const size_t size = sizeof v / sizeof v[0];
    const int * begin = &v[0];
    const int * end = &v[size];
    bool found = search(begin, end, 42) != end;
    puts(found ? "found" : "not found");
}
```

Another difference is that y is implicitly "const" (in the sense it cannot be reassigned) whereas x is not.

The declaration extern int y[]; declares y to be an incomplete type.

const int * ptr; be written with the const and the int in either order.
int const * ptr; // also allowed

The former is the most common style (the latter is quite common in C++).

const int * ptr; be written with the const and the int in either order.
int const * ptr; // also allowed

The former is the most common style (the latter is quite common in C++).

The function is dangerous to call because it does not know how long the arrays are. Consequently the dst[at] and src[at] expressions could result in undefined behaviour.

The parameters are ordered dst,src and not src,dst deliberately. This is to mimic assignment – viz the left expression changes, the right expression does not.

The parentheses are needed because of precedence.

The comparison with '\0' is optional because it is the same as != 0 which is how the compiler will convert an integer expression in a boolean context.

Pre-ANSI C did not have the `void` keyword. In K&R C `char*` was used as a synonym for `void*`.

The conversion from a `void*` pointer to a specific pointer would require a cast in C++, as would conversions between other pointer types.

Note that a function pointer is not an object pointer.

The standard also says (in an example)…

<quote>

The file scope declarations:

```
int * restrict a;
int * restrict b;
extern int c[];
```

asserts that if an object is accessed using the value of a, b, or c, then it is never accessed using the value of either of the other two.

</quote>

Note that accessing an object using the value of a means *a.

The line cooked[42] = 99; is dangerous since it could easily be out of bounds and cause undefined behaviour (since the size of the array is the runtime value n).