

# Pointers

“to get a deeper understanding of the language”



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

# pointers

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

```
int * stream;
```

stream  
is-a  
pointer  
to-an  
int

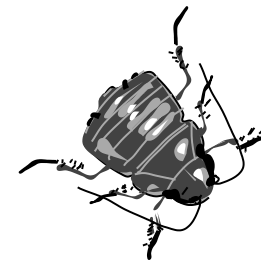
```
int * stream;
```

```
int * stream;
```

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

equivalent to

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



# 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

preferred

# 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

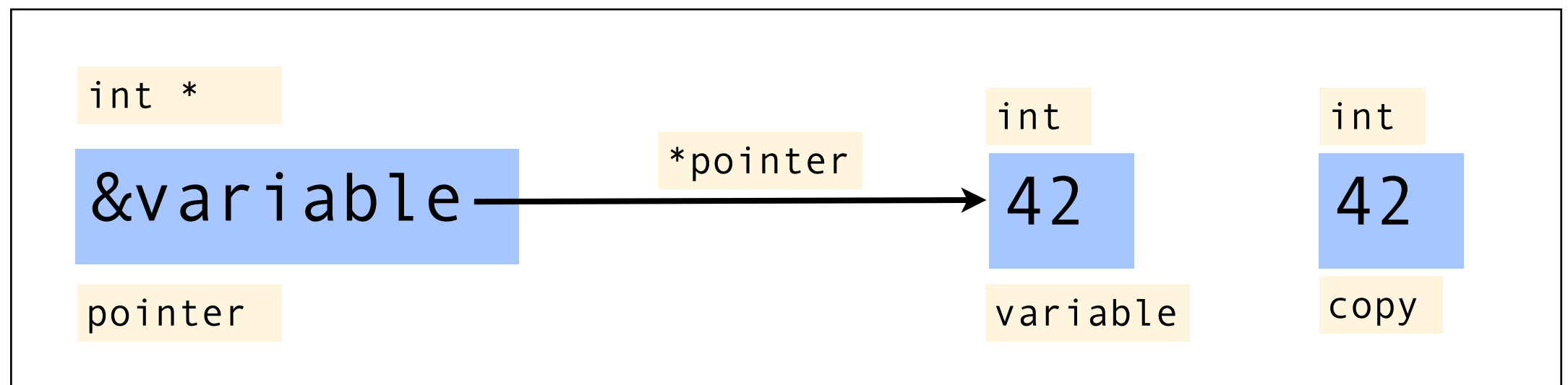
```
int variable = 42;
```

```
...
```

```
int * pointer = &variable; ← * used in a declarator
```

```
...
```

```
int copy = *pointer; ← * used in an expression
```



level diagram

# pointer function 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);
}
```

# array decay

- 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?

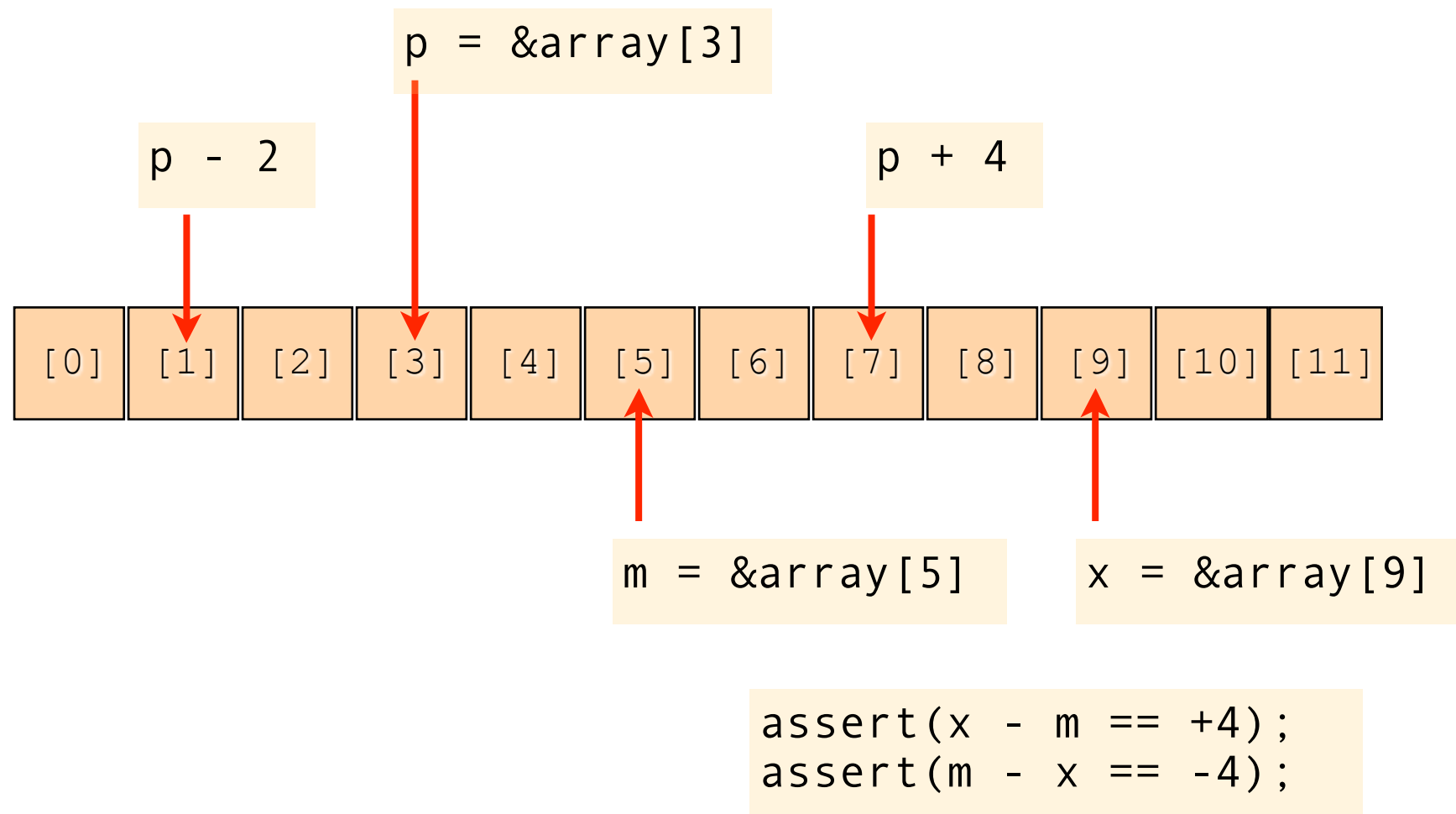
```
#include <stdio.h>

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



# 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>`





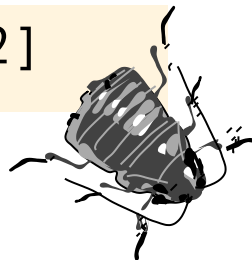
# one beyond the end

- 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]
```

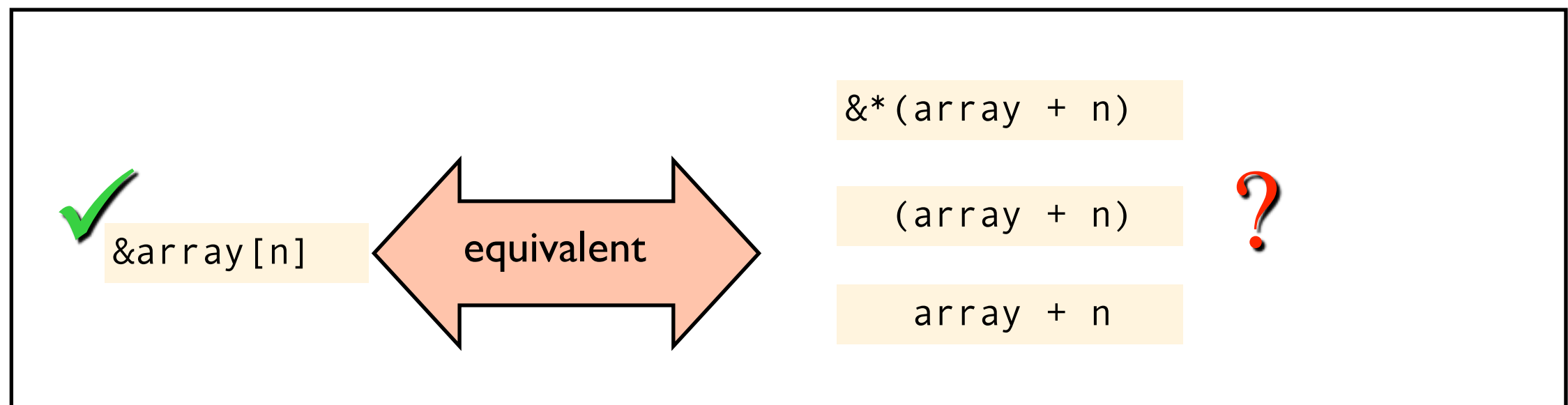
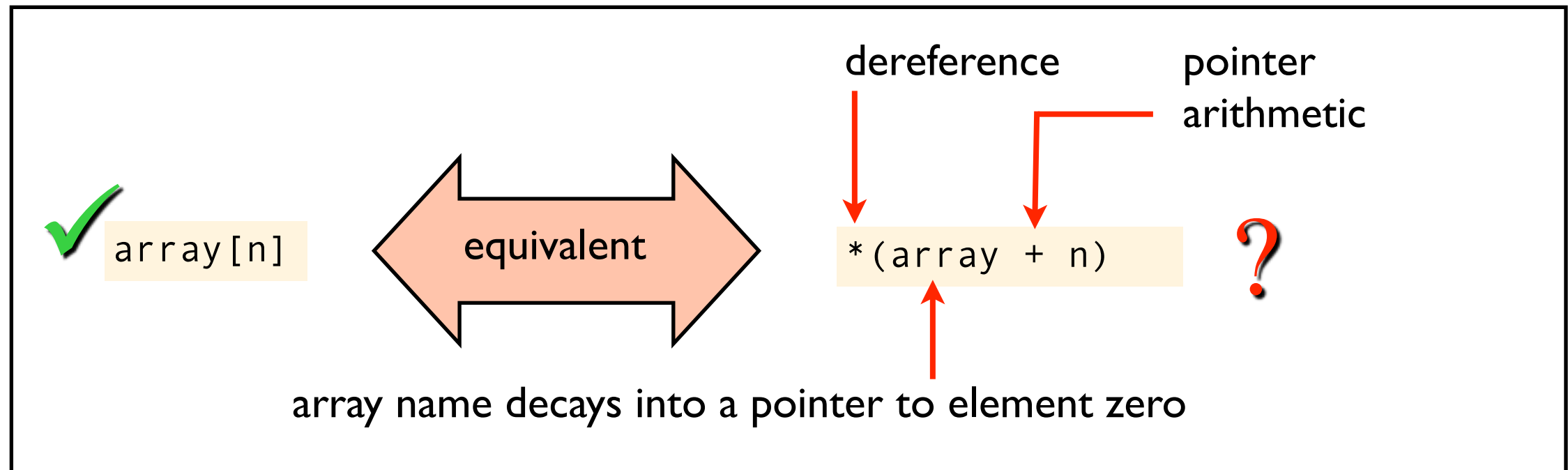


```
int * p = &array[0];  
int * q = &array[42];  
assert(q - p == 42);
```



# pointers $\longleftrightarrow$ arrays

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





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

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

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

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

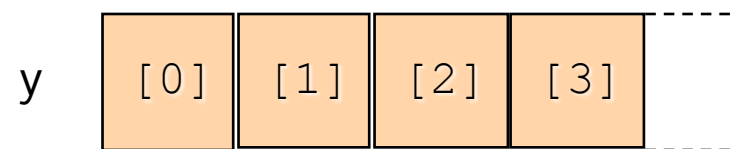
So....  
 $a[n] == n[a]$

# 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

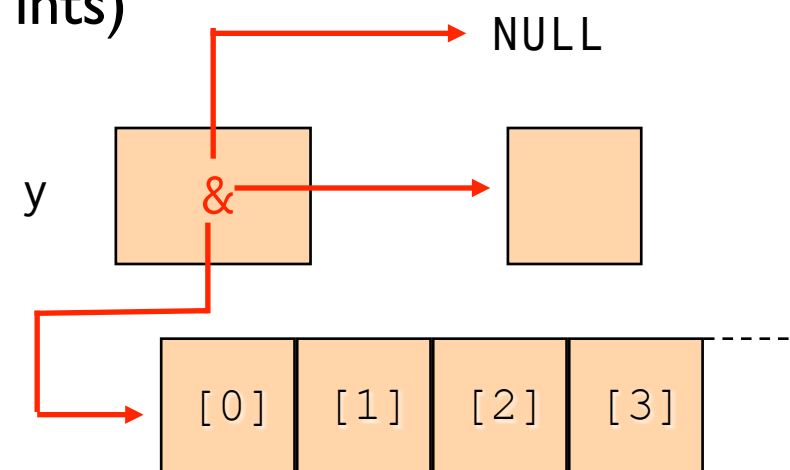
y is an array of int (of unspecified size)

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



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

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



# pointer confusion

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

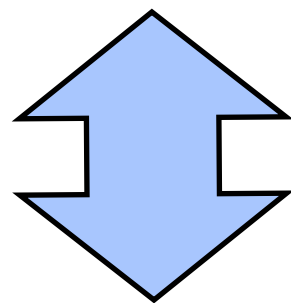
```
int array[42];  
int * pointer = &array[0];
```

```
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++;
```

C was designed so that the syntax of use  
mirrors the syntax of declaration



```
int identifier;  
↑      ↑  
typedef int identifier;
```

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

```
int days_in_month[12];  
↑      ↑  
...days_in_month[at]...
```

```
void func(int a, int b);  
↑      ↑      ↑  
func(   4,    2);
```

# syntax trick

- syntax of use mirrors syntax of declaration
- declaration tells you the type and any qualifiers

```
int value = 42;  
const int *ptr = &value;
```

```
*ptr = 42;
```

**x**

```
const int = 42;
```

# pointer + const

- often causes 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





# pointer + const



- often causes 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 ✗

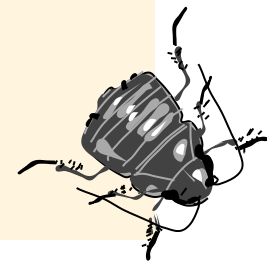
# restrict

- applies only to pointer declarations
- `type* restrict p`  $\rightarrow$  `*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
}
```



## remember this?

```
int f(int * p, int * q)
{
    return (*p) - (*q)++;
}
```

\*p (value computation)  
is unsequenced  
relative to  
(\*q)++ (side effect)

undefined if p and q point to the same object

```
int f(int * restrict p, int * restrict q)
{
    return (*p) - (*q)++;
}
```

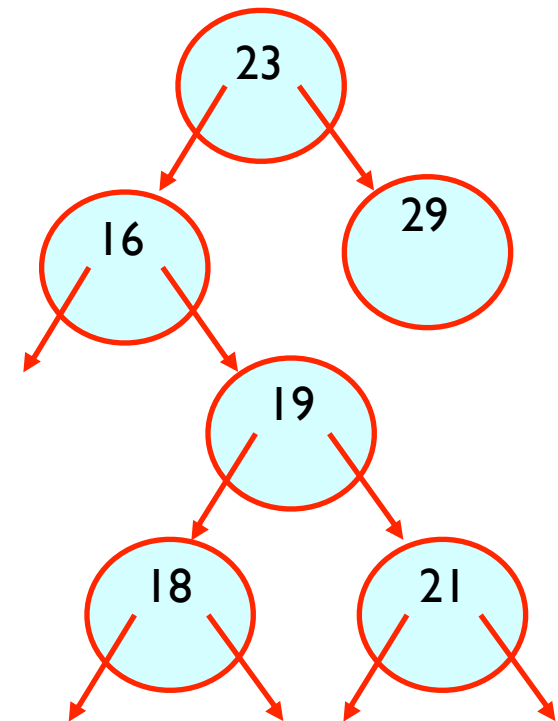
p and q do not point to the same object

# restrict

c99

- restrict can be used on pointer data members
- enables the same no-alias optimizations

```
struct tree_node
{
    int count;
    struct tree_node * restrict left;
    struct tree_node * restrict right;
};
```



# 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
- pointer arithmetic is scaled
- pointers and arrays share many similarities
  - but they are not the same
  - the differences are as important as the similarities
- be clear about what you can do with a pointer
  - be clear about what's const
  - respect restrict