

# **Pointers and Arrays**

- 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

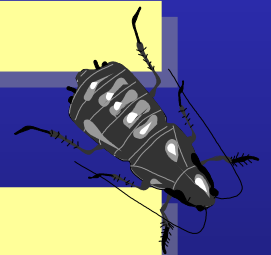
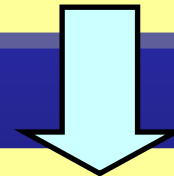
to an

int

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

Equivalent to

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



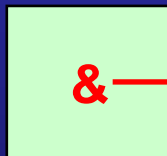
- many pointers can point to the same object
  - ◆ information can be shared across a program

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



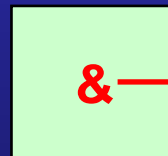
a wibble!

wibble\*



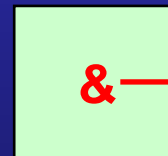
p1

wibble\*



p2

wibble\*



another

sharing

- 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

- 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

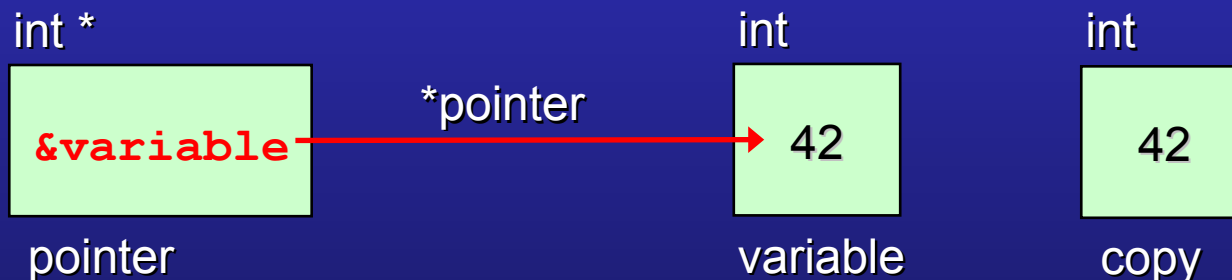
# address-of/dereference

- 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

7

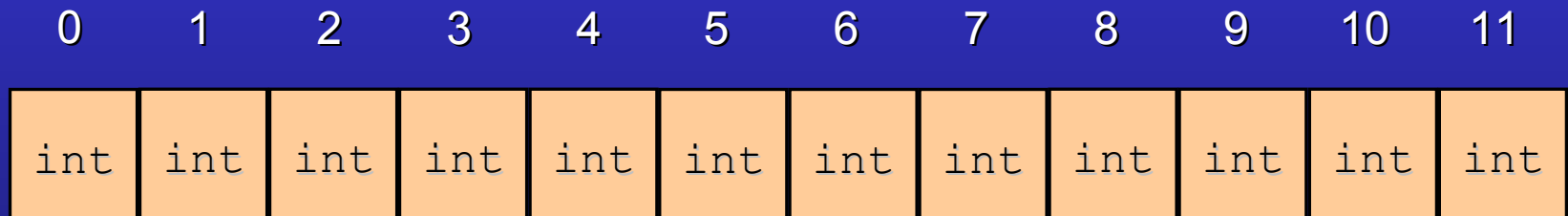
```
#include <stdio.h>

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);
}
```

- 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



1. a trailing comma is allowed
2. an empty list is not allowed (it is in C++)

- **array support [int] designators**
  - ◆ **int must be a constant-expression**

```
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

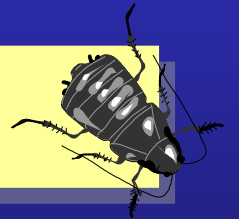
- starts at zero and is not bounds-checked
  - ◆ out of bounds access is undefined

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

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



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



- in an expression the name of an array "decays" into a pointer to element zero<sup>†</sup>
  - ♦ 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

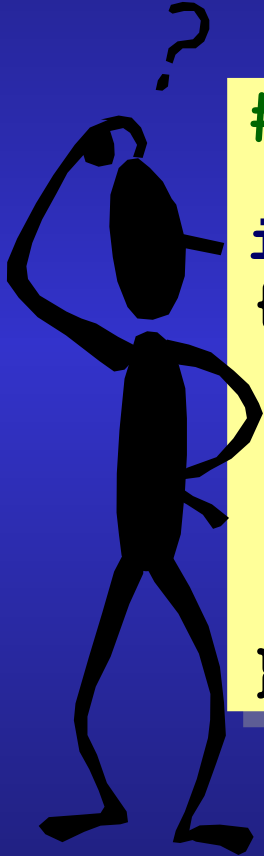
```
display(42, table);  
display(42, &table[0]);
```



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

<sup>†</sup>except in a sizeof expression

- what does the following program print?
  - ♦ why?

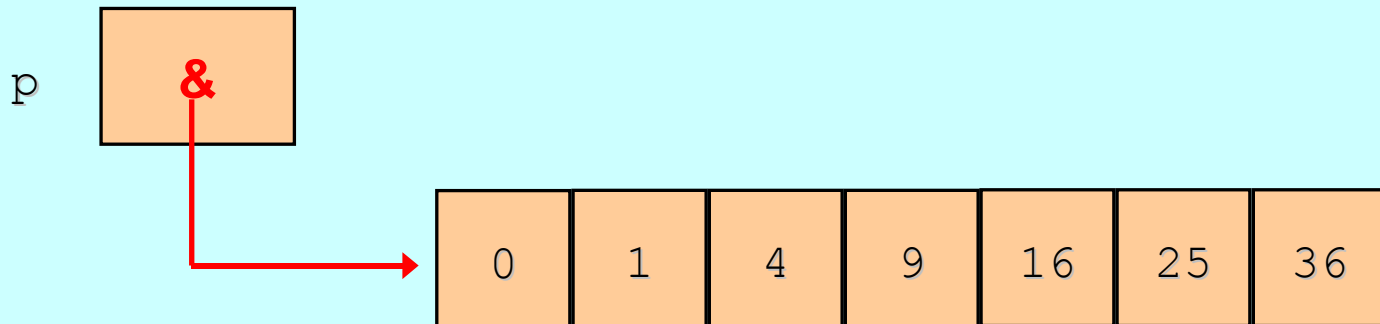


```
#include <stdio.h>

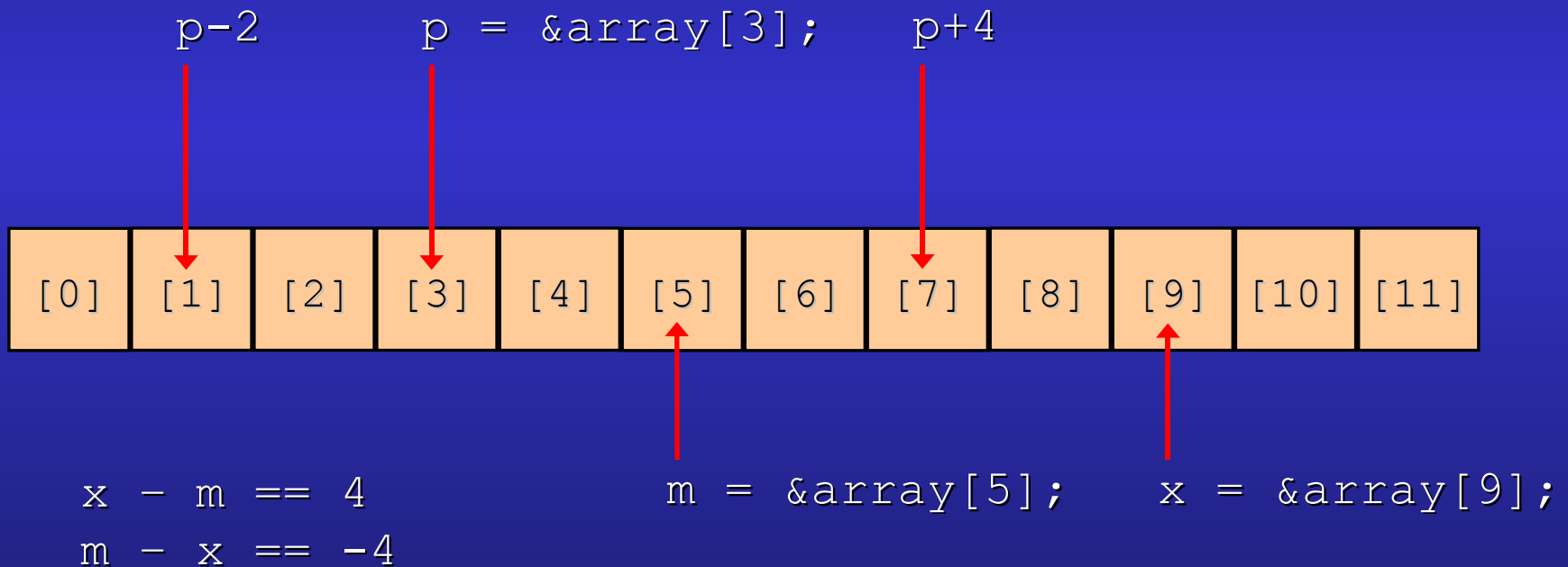
int main(void)
{
    int array[] = { 0,1,2,3 };
    int clone[] = { 0,1,2,3 };
    puts(array == clone
         ? "same" : "different");
    return 0;
}
```

- 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

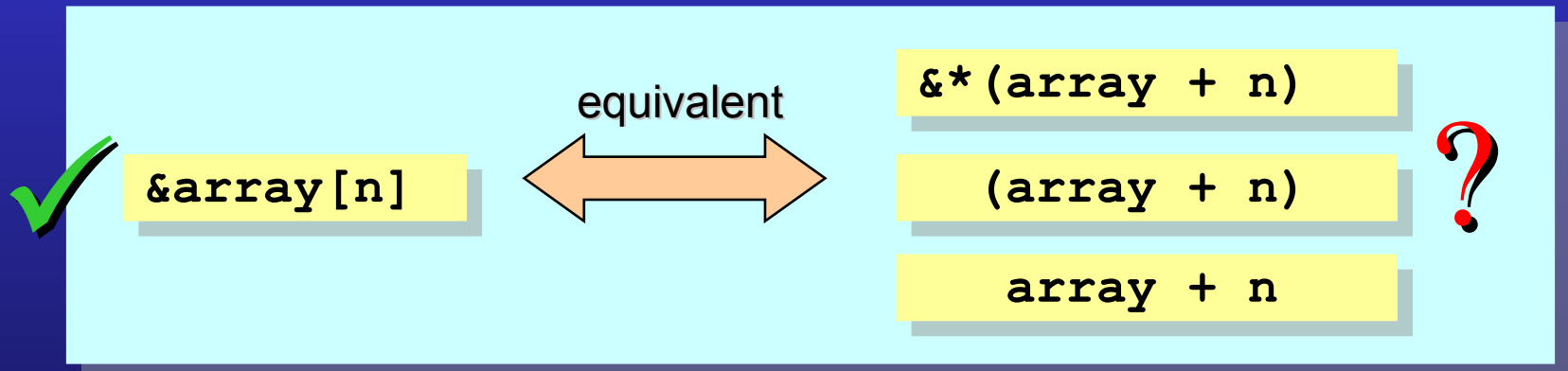
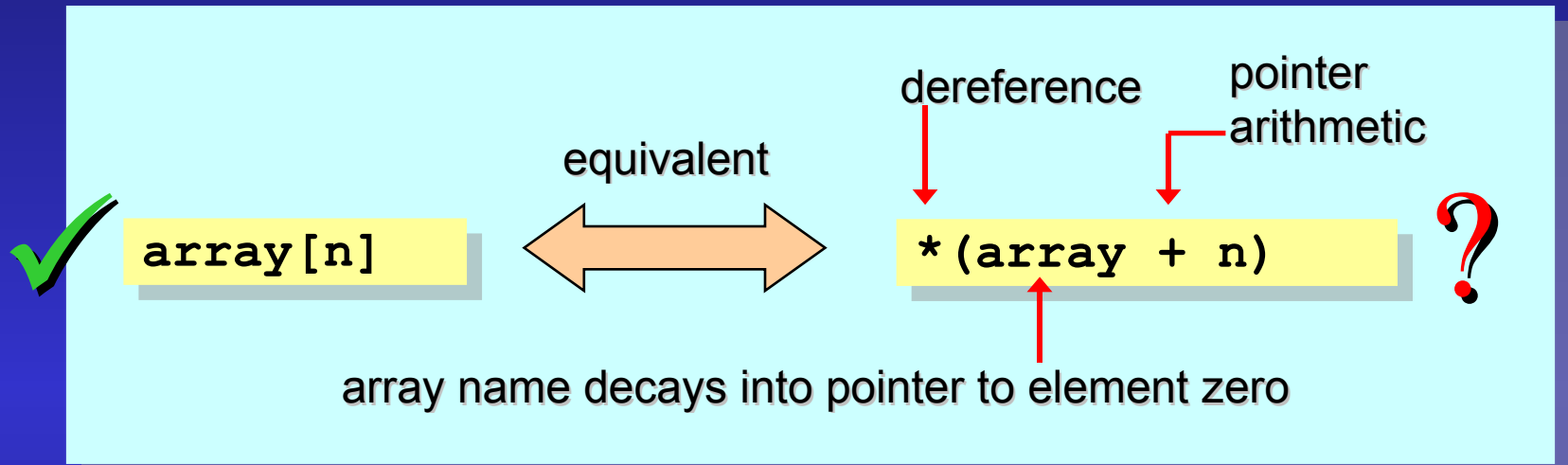
```
int * p =  
    (int []) { 0, 1, 4, 9, 16, 25, 36 };
```



- 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>`



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



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

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

this is *undefined*

```
array[42]
```

but these are ok

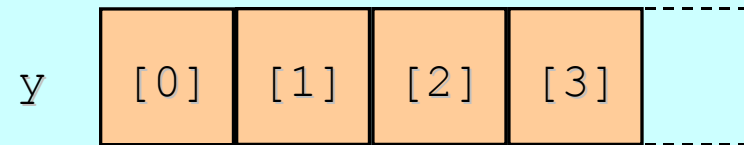
```
array + 42  
&array[42]
```

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

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

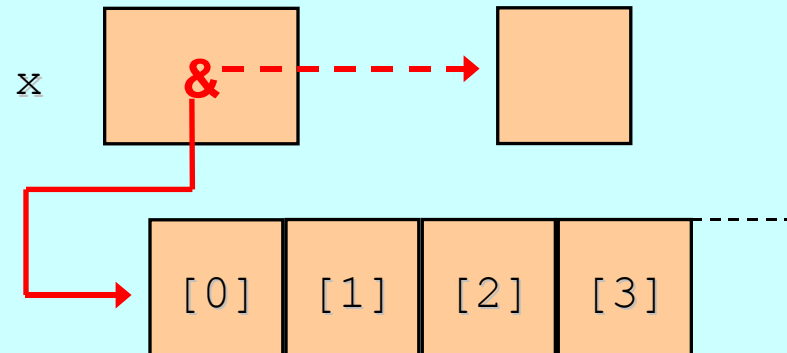
y is an array of int (of unspecified size)

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



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

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



- **syntax of declaration mirrors syntax of use**

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

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



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

```
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++;
```



- 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 ✓



- 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 ✗

- 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

why are the parameters in this order?

```
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

- 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

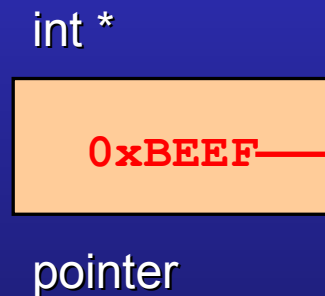
```
generic_pointer = int_specific_pointer;  
int_specific_pointer = generic_pointer;
```

these don't compile

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

- 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>

```
#include <stdint.h>
intptr_t address = 0xBEEF;
int * pointer = (int*)address;
```



- **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**

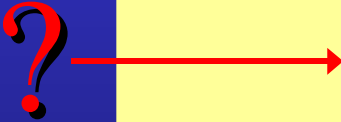
```
void f(int n, int * restrict p, int * restrict q)
{
    while (n-- > 0) {
        *p++ = *q++;
    }
}
```

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

- dynamic memory 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`

- **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

- This course was written by

---

Expertise: Agility, Process, OO, Patterns  
Training+Designing+Consulting+Mentoring

{ JSL }

*Jon Jagger*

jon@jaggersoft.com

www.jaggersoft.com