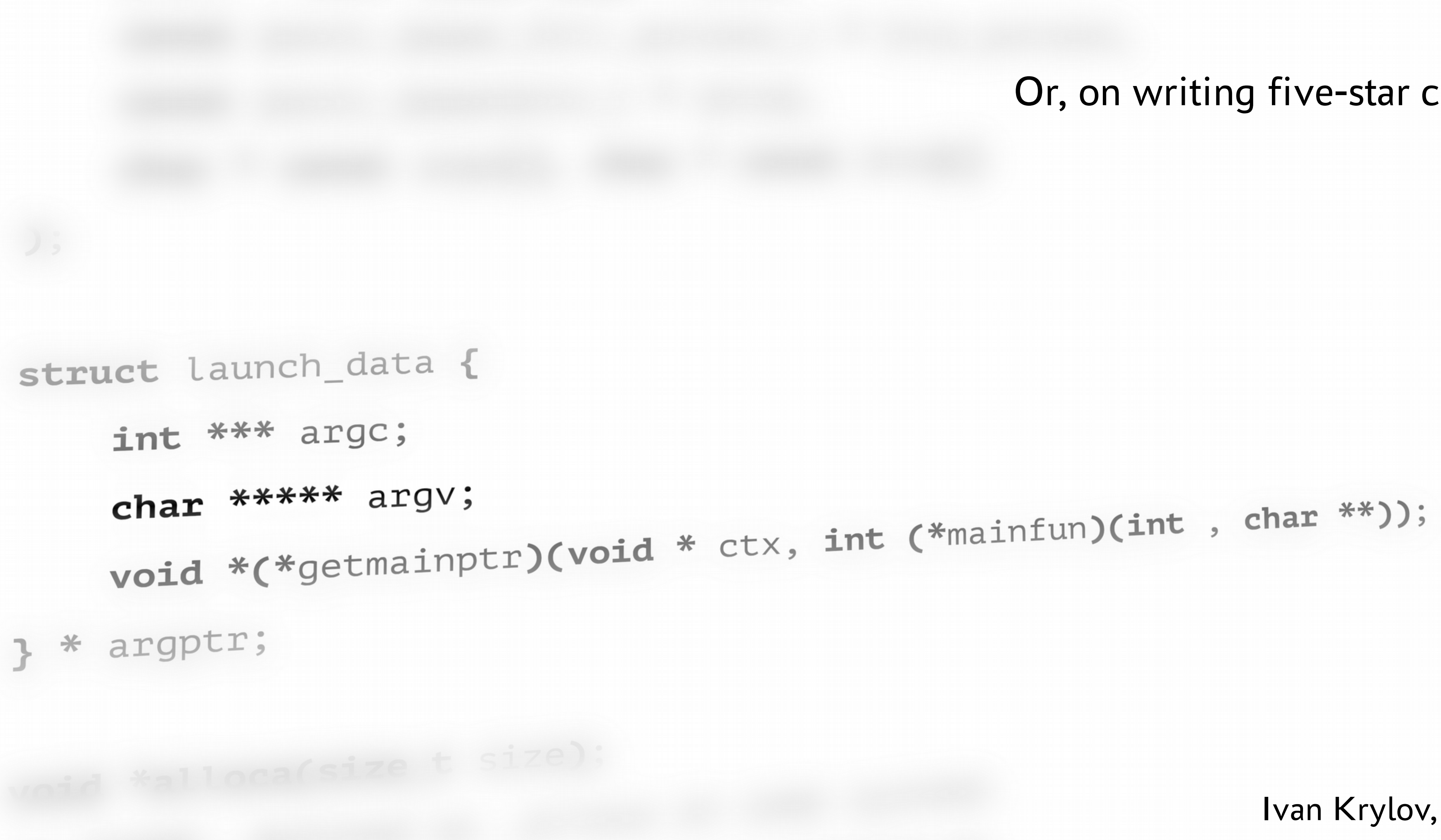
Pointers and the C memory model

Or, on writing five-star code

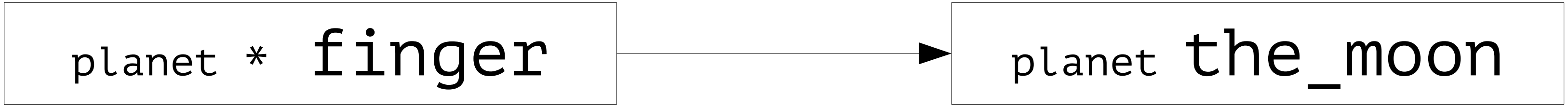


```
struct launch_data {
    int *** argc;
    char ***** argv;
    void *(*getmainptr)(void * ctx, int (*mainfun)(int , char **));
} * argptr;

void *alloca(size_t size);
```

Ivan Krylov, 2023

Pointers: what and why?



```
planet * finger
```

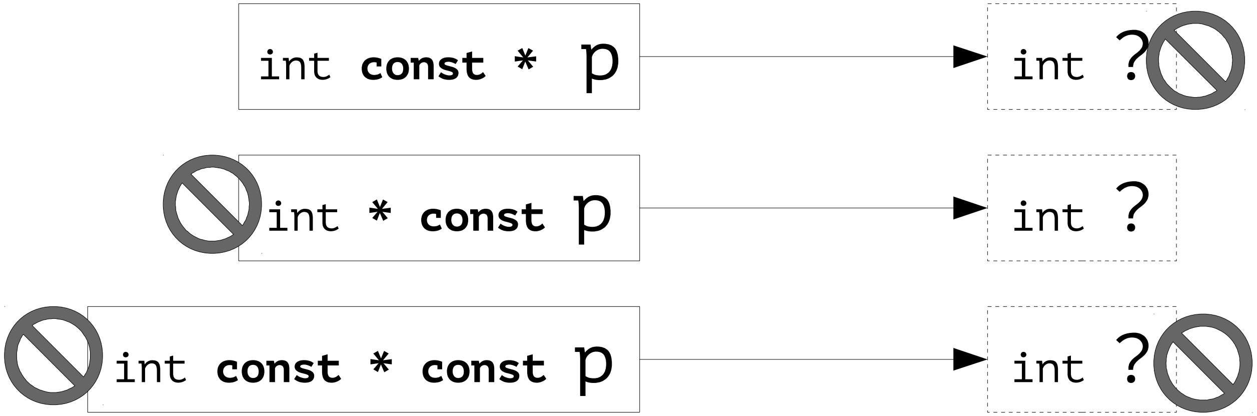
```
planet the_moon
```

- Non-local variable access (“impure” functions)
  - Anonymous variables
- Dynamic program behaviour (function pointers)
- Dynamic memory allocation (session 7)
- Underlying reality

Declaring a pointer

```
struct planet const * finger = &the_moon;
```

- “Declaration follows use”
- Qualifier on the left/right side of the *****



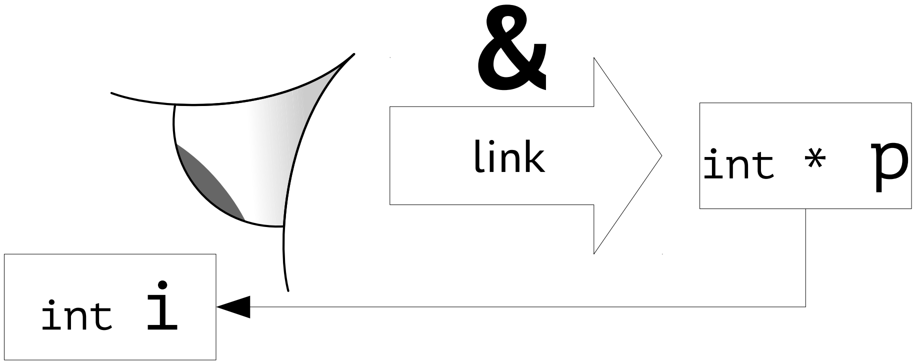
```
p = &i;

*p = i;
```

Address-of operator: taking a pointer

- Takes an object, returns a pointer
- Applicable to all objects except those marked **`register`**

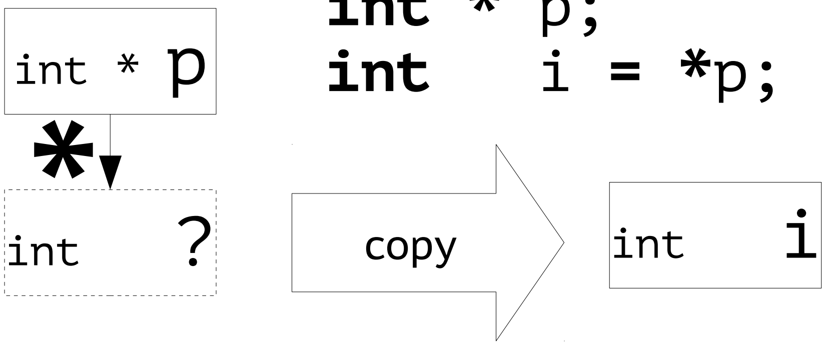
```
int   i;
int * p = &i;
```



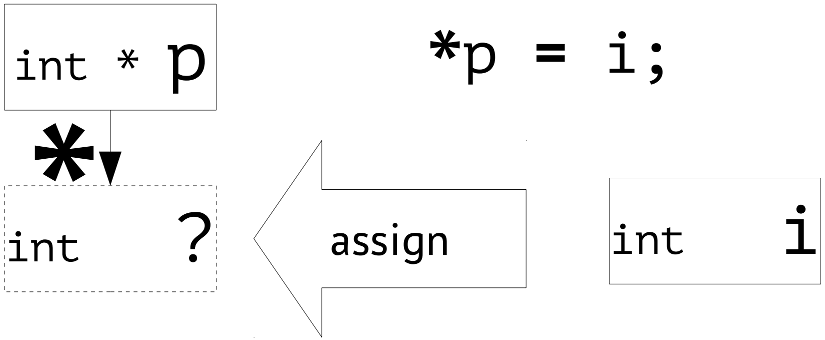
Object-of operator: following a pointer

- Also called “indirection operator”, “dereferencing”
- Accesses the pointed-to object, making a copy or overwriting it

```
int * p;
int   i = *p;
```

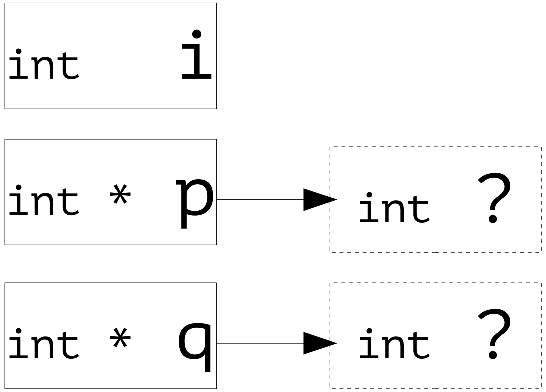


```
*p = i;
```



Exercise 11.3.2: assignments

Given `int i, *p, *q`, which assignments are valid?

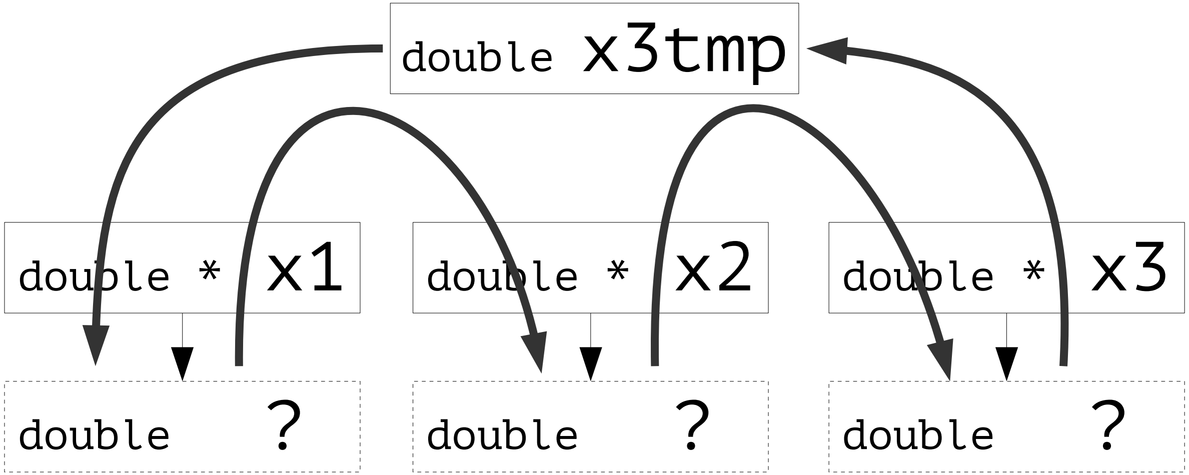


```
 p = i;      p = &q;     p = *q;
*p = &i;     p = *&q;   *p = q;
&p = q;      p = q;     *p = *q;
```

Exs 9: cyclically shifting three objects

Write a function that receives pointers to three objects and that shifts the values of these objects cyclically.

```
void cycle(double * x1, double * x2, double * x3) {
    // x1, x2, x3 -> x3, x1, x2
    double x3tmp = *x3;
    *x3 = *x2;
    *x2 = *x1;
    *x1 = x3tmp;
}
```



More or less the same as CPAMA exercise 11.4.1

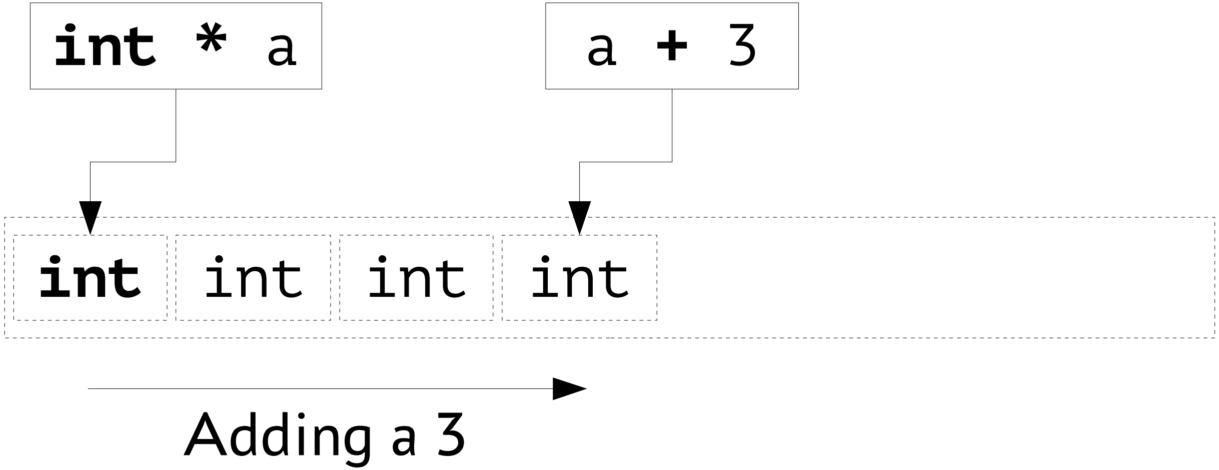
Exercise 11.4.6: two largest numbers

```
void find_two_largest(
    size_t n, const int a[n], int * largest, int *second_largest
) {
    if (n == 0) return;
    *largest = *second_largest = *a;
    for (size_t i = 1; i < n; ++i) {
        if (a[i] > *largest) {
            *second_largest = *largest;
            *largest = a[i];
        } else if (a[i] > *second_largest) {
            *second_largest = a[i];
        }
    }
}
```

Offsetting a pointer

$$\text{pointer} + \text{offset} \rightarrow \text{pointer}\,, \quad \text{offset} \in \mathbb{Z}$$

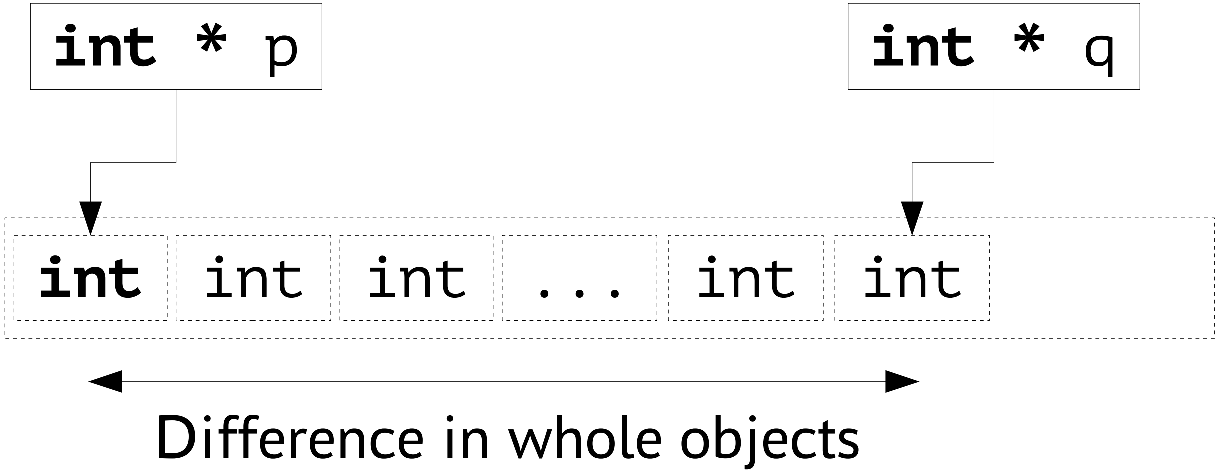
Adding an integer to a pointer advances it by whole pointed-to objects



Pointer difference

$$\text{pointer} - \text{pointer} \rightarrow \text{offset}\,,\ \text{offset} \in \mathbb{Z}$$

- Subtracting pointers from the same array gives an integer offset of type **`ptrdiff_t`**



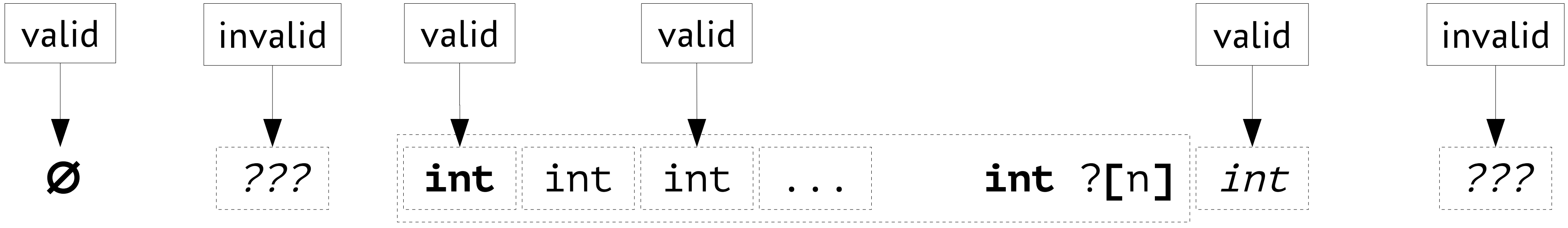
Exercise 12.1.2: middle of array

```
type *high, *low, *middle;
middle = (low + high) / 2;
```

- Pointer addition, division not valid
- Arithmetically, $(a + b)/2 = a + (b - a)/2$
- Pointer subtraction; addition, division of offsets all valid
- `middle = low + (high - low) / 2;`

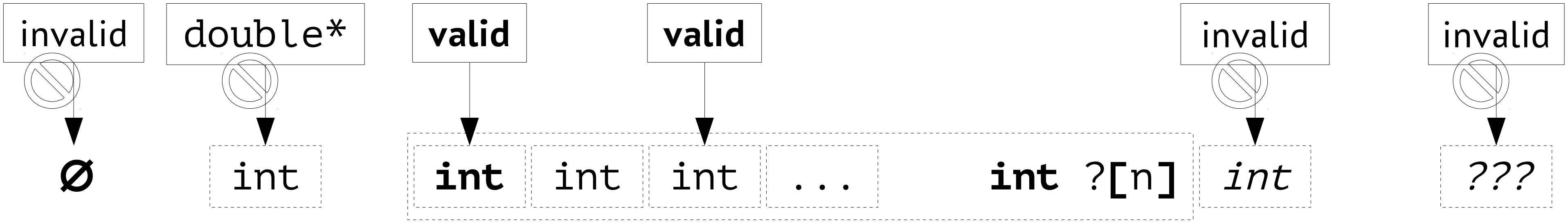
Pointer validity

- Pointers must either:
  - Be null: `ptr = 0;`
  - Point to a valid object
  - Point one position beyond a valid object



Pointer access validity

- Accessing a pointer is undefined behaviour unless all of the following is true:
  - Pointer isn’t null
  - Object is of designated type
  - Object is not a trap representation



Exercise 12.3.6: array to pointer arithmetic

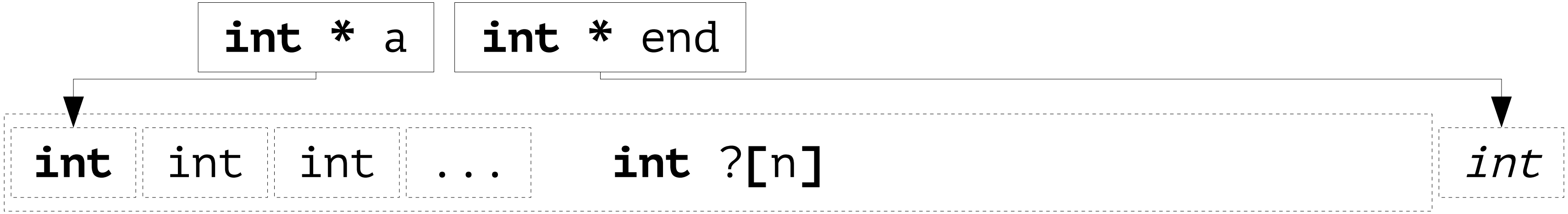
```c
int sum_array(
  const int a[], int n
) {
  int i, sum;
  sum = 0;




  for (i = 0; i < n; i++)
      sum += a[i];
  return sum;
}
```

```c
int sum_array(
  size_t n, const int a[n]
) {
    int sum = 0;

    for (
      int *end = a+n;
      a < end;
      ++a
    ) sum += *a;
    return sum;
}
```



See also: “On doing without HW multipliers” by Ido Gendel

Other operations on pointers

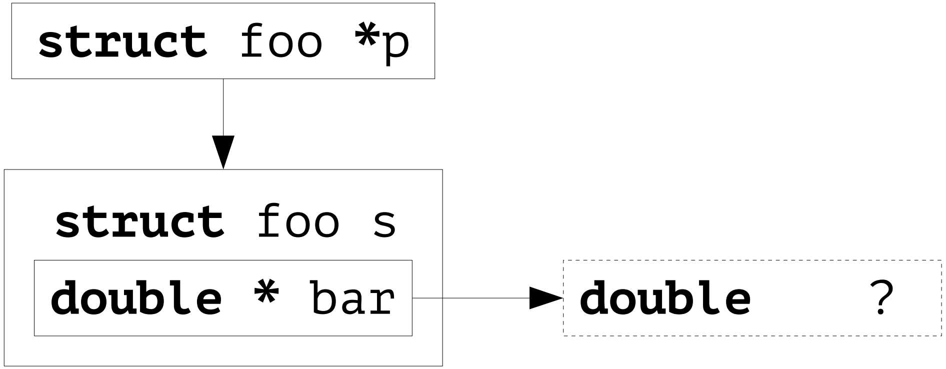
- Checking a pointer for truth:
  `if (``ptr`**`)`** checks that `ptr` is not null
- Printing a pointer
  `printf(``"%p", `**`(void*)`**`ptr`**`);`**

Pointers and structures

- The **->** operator follows the structure pointer and accesses its member

```
struct foo {
    double * bar;
} s, *p = &s;

s.bar;     // -> double *
*s.bar;    // -> double
(*p).bar;  // -> double *
p->bar;    // -> double *
*p->bar;   // -> double
```



Exs 14: implementing `rat_print`

```
typedef struct {
  bool sign;
  size_t num;
  size_t denom;
} rat;
```

$$q=\frac{a}{b}\in\mathbb{Q} \Leftrightarrow a\in\mathbb{Z}, b\in\mathbb{N}$$

```
char const * rat_print (size_t len, char tmp[len], rat const * x) {
    snprintf(
        tmp, len, "%c%zu/%zu",
        x->sign ? '-' : '+', x->num, x->denom
    );
    return tmp;
}
```

```
char const * str = rat_print(SIZE, (char[SIZE]){0,}, &x);
```

Exs 15: implementing `rat_print_normalized`

```c
char const* rat_normalize_print(
    size_t len, char tmp[len], rat const* x
) {
    rat xnorm = rat_get_normal(*x);
    return rat_print(len, tmp, &xnorm);
}
```

- Not allowed to modify x
- Therefore have to make a copy to normalise

Exs 16: implementing `rat_dotproduct`

Given vectors **a**, **b** of rationals, compute their scalar product

$$\sum_i a_i \cdot b_i$$

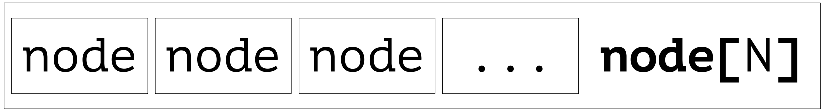
```c
rat* rat_dotproduct(
    rat rp[static 1], size_t n, rat const A[n], rat const B[n]
) {
    rat_init(rp, 0LL, 1ULL);
    for (size_t i = 0; i < n; ++i)
        rat_rma(rp, A[i], B[i]);
    return rp;
}
```

$\mathrm{rp} \leftarrow 0/1$

$\mathrm{rp} \leftarrow \mathrm{rp} + a_i \cdot b_i$

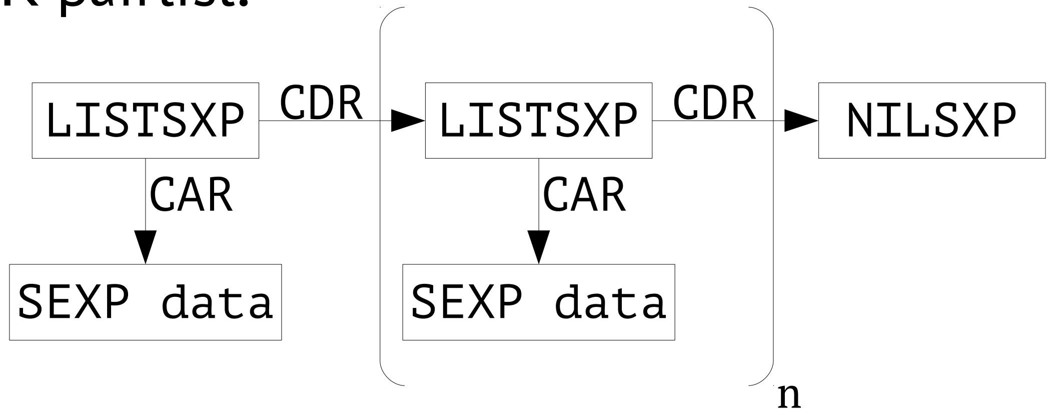
Interlude: linked lists

Vector (array):



node node node ... **node[N]**

R pairlist:



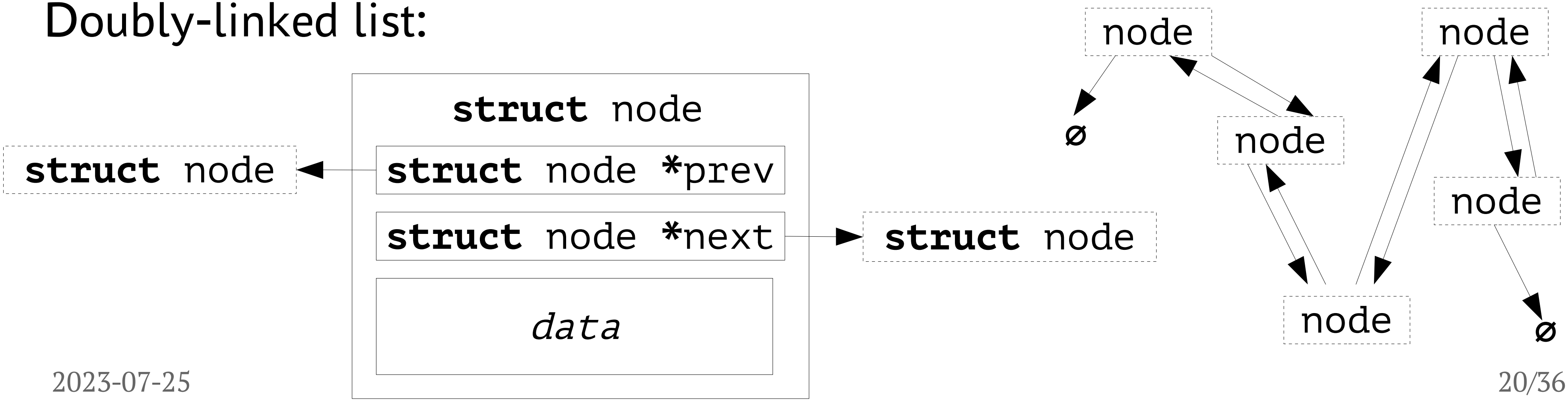
LISTSXP CDR LISTSXP CDR NILSXP

CAR CAR

SEXP data SEXP data

n

Doubly-linked list:



struct node

struct node *****prev → **struct** node

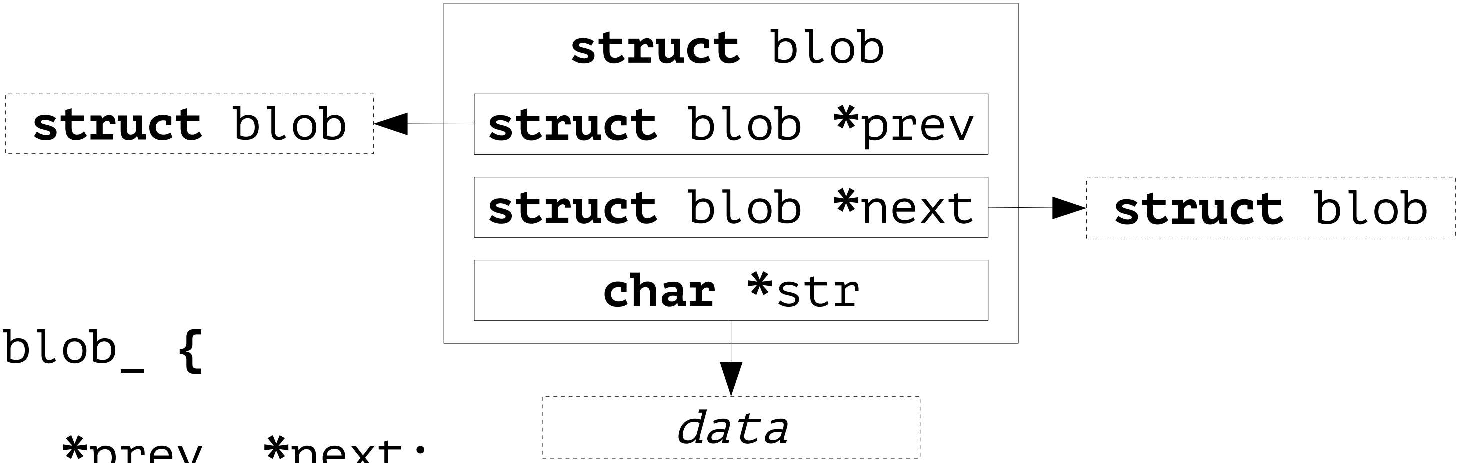
struct node *****next → **struct** node

data

node node node node node ∅ ∅

Challenge 12: doubly-linked list of strings[†]

For a text processor, can you use a doubly linked list to store text? The idea is to represent a "blob" of text through a `struct` that contains a string (for the text) and pointers to preceding and following blobs.



```
typedef struct blob_ {
    char * str;
    struct blob_ *prev, *next;
} blob;
```

Pointers and arrays

- Arrays “decay” to pointer to first element
- Arrays in function declaration are actually pointers
  - Compiler takes array size as a hint

$$\mathit{object}\ \mathtt{A[];}$$

$$\mathtt{A} \Rightarrow \mathtt{\&A}[\mathtt{0}]$$

$$\mathtt{a}[\mathtt{i}] \equiv \mathtt{*}(\mathtt{a+i}),$$

$$\mathtt{a} \in \text{pointer},\ \mathtt{i} \in \mathbb{Z}$$

Array arguments

```
void matrix_mult (
    size_t n, size_t k, size_t m,
    double C[n][m],
    double /* const */ A[n][k], double /* const */ B[k][m]
);
void matrix_mult (
    size_t n, size_t k, size_t m,
    double (C[n])[m],
    double (A[n])[k], double (B[k])[m]
);
void matrix_mult (
    size_t n, size_t k, size_t m,
    double (*C)[m],
    double (*A)[k], double (*B)[m]
);
```

```
warning: pointers to arrays with different qualifiers are
incompatible in ISO C [-Wpedantic]
```

See also: GNU C extension making them work as expected

Function pointers

- A function decays to a pointer to its start

```
// define a function derived type
typedef void atexit_function(void);
// define a function pointer derived type
typedef atexit_function * atexit_function_pointer;
typedef void (*atexit_function_pointer)(void);
// declare a function taking an atexit_function
void atexit(void f(void));
void atexit(void (*f)(void));
void atexit(atexit_function f);
void atexit(atexit_function * f);
void atexit(atexit_function_pointer f);
```

Using function pointers†

```
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    puts("Help! I'm being called as an atexit() handler!");
}

int main(void) {
    atexit(f);
    atexit(&f);
    atexit(&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*f);
    atexit(****************************f);
    puts("About to return from main()");
    return 0;
}
```

Exercise 18.4.8

Describe the type of `x` as specified by the declarations:

`char (*x[10])(int);`	Array `x` of 10 pointers to functions. Each function takes an `int` and returns a `char`.
`int (*x(int))[5];`	Function `x` taking an `int` and returning a pointer to an array of 5 `int`s.
`float *(*x(void))(int);`	Function `x` taking nothing and returning a pointer to a function taking an `int` and returning a pointer to `float`.

`void (*x(`
`    int, void (*y)(int)`
`))(int);`
	Function `x` taking an `int` and a pointer to a function `y` (which takes an `int` and returns nothing). The function `x` returns a pointer to a function that takes an `int` and returns nothing.

Exercise 18.4.10

p is a pointer to a function with a character pointer argument that returns a character pointer.

```
char *(*p)(char*);
```

f is a function with 2 arguments:

1) p, a pointer to a structure with tag t
2) n, a long integer

f returns a pointer to a function that has no arguments and returns nothing.

```
void (*f(struct t *p, long n))
(void);
```

a is an array of 4 pointers to functions that have no arguments and return nothing. The elements of a initially point to functions named insert, search, update, and print.

```
void (*a[4])(void) = {
    &insert, &search,
    &update, &print
};
```

b is an array of 10 pointers to functions with two int arguments that return structures with tag t.

```
struct t (*b[10])(int, int);
```

Exercise 18.4.9

Use common types to build up complex types
from easy-to-understand parts

`char (*x[10])(int);`	`typedef char (*f_ic)(int);`
	`f_ic x[10];`

`int (*x(int))[5];`	`typedef int (*ap)[5];`
	`ap x(int);`

`float *(*x(void))(int);`	`typedef float *(*f_if)(int);`
	`f_if x(void);`

`void (*x(int, void (*y)(int)))(int);`	`typedef void (*f_i)(int);`
	`f_i x(int, f_i);`

cdecl

- Original program: David R. Conrad, 1996
  - Available in GNU/Linux distros
- Also a website now: https://cdecl.org/
- `cdecl> declare fptab as array of pointer to function returning pointer to char`
  - `char *(*fptab[])()`
- `cdecl> explain char *(*fptab[])()`
  - `declare fptab as array of pointer to function returning pointer to char`

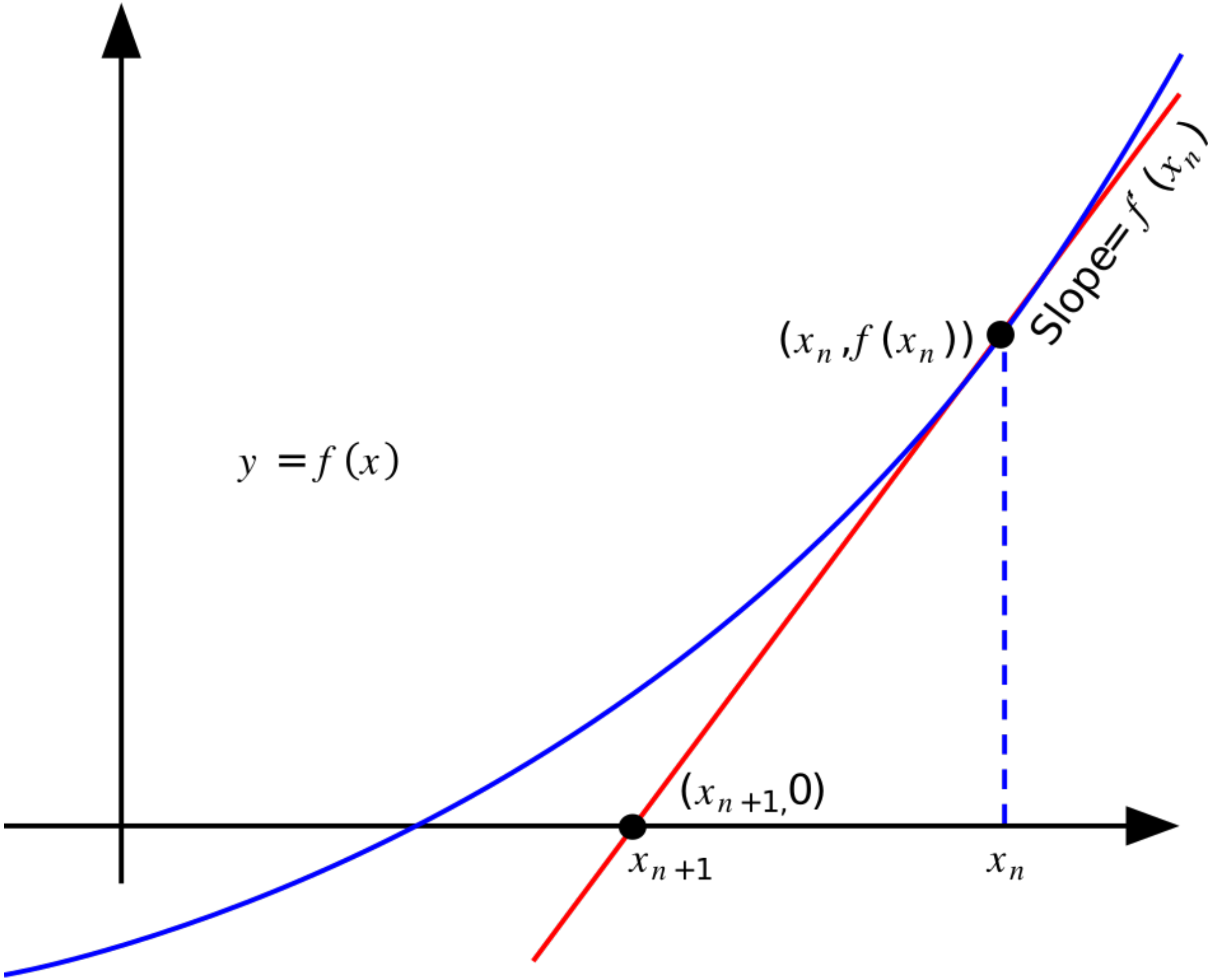
Challenge 13: generic derivative and Newton's method[†]

$$f\left(x\right)=f\left(\Re x,\boldsymbol{i}\cdot\Im x\right),\ x\in\mathbb{C}$$

$$\begin{aligned}
f\left(\Re\left(x+\Delta\right),\boldsymbol{i}\Im\left(x+\Delta\right)\right) \quad &\approx \quad f\left(\Re x,\boldsymbol{i}\Im x\right)+\frac{\partial f}{\partial\Re x}\cdot\Re\Delta+\boldsymbol{i}\frac{\partial f}{\partial\left(\boldsymbol{i}\Im x\right)}\cdot\Im\Delta \ + \\
&+ \quad \frac{\partial^2 f}{\partial\left(\Re x\right)^2}\cdot\frac{\left(\Re\Delta\right)^2}{2}-\frac{\partial^2 f}{\partial\left(\boldsymbol{i}\Im x\right)^2}\cdot\frac{\left(\Im\Delta\right)^2}{2} \ + \ O\left(\Delta^3\right) \\
f\left(\Re\left(x-\Delta\right),\boldsymbol{i}\Im\left(x-\Delta\right)\right) \quad &\approx \quad f\left(\Re x,\boldsymbol{i}\Im x\right)-\frac{\partial f}{\partial\Re x}\cdot\Re\Delta-\boldsymbol{i}\frac{\partial f}{\partial\left(\boldsymbol{i}\Im x\right)}\cdot\Im\Delta \ + \\
&+ \quad \frac{\partial^2 f}{\partial\left(\Re x\right)^2}\cdot\frac{\left(\Re\Delta\right)^2}{2}-\frac{\partial^2 f}{\partial\left(\boldsymbol{i}\Im x\right)^2}\cdot\frac{\left(\Im\Delta\right)^2}{2} \ + \ O\left(\Delta^3\right)
\end{aligned}$$

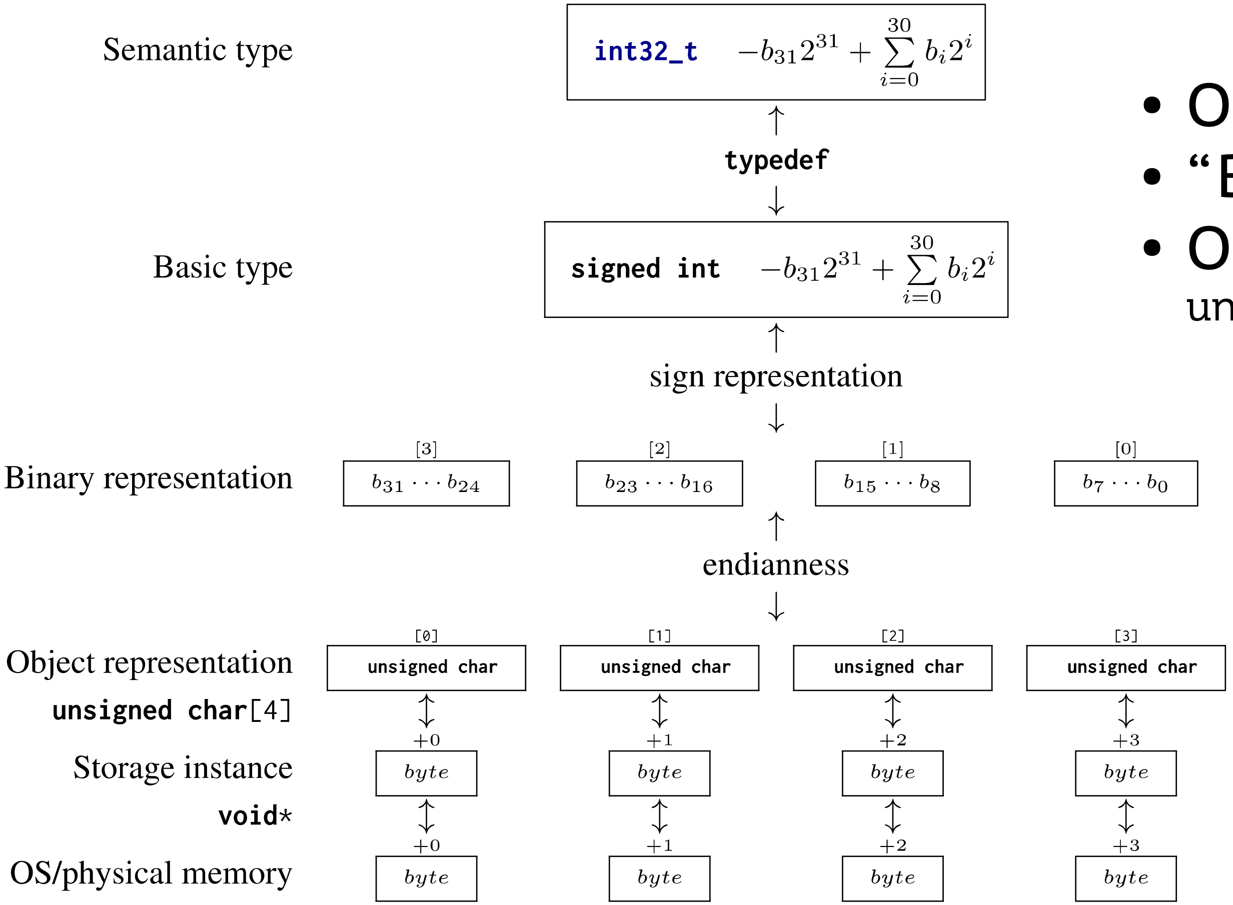
Newton's method

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$



Source: https://en.wikipedia.org/wiki/File:Newton_iteration.svg

The C memory model



- Objects consist of bytes
- “Byte” is whatever `char` is
- Objects can be viewed as `unsigned char A[sizeof object]`

See also: The Descent to C by Simon Tatham

Exs 23-25: Unions[†]

- Unions are overlays, sharing memory
- Byte-order representation is implementation-defined

Aliasing

- “Strict aliasing”: only pointers of same type may alias
  - Memory access through variable of different type may not change an object
  - Except character types (bytes)

See also: the Rcomplex / complex*16 bug

Pointers to void

- Pointers implicitly convert to and from `void*`
  - Except function pointers
    - but see POSIX, Windows
- Aliasing rules still apply, even if pointer is “laundered” through `void*`

Alignment

- Processors may require multi-byte objects to align to a number of bytes
- Rule of thumb: address must be divisible by size of object
- x86 processors: no error, minor slowdown
  - but will crash for SIMD
- Some ARM chips: will crash
- Other CPUs: may access different object