C-Programming Review

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I

C-Programming Review

POINTERS

Review of pointers

- A pointer is just a memory location.
- A memory location is simply an integer value, that we interpret as an address in memory.
- The contents at a particular memory location are just a collection of bits – there’s nothing special about them that makes them `ints`, `chars`, etc.
  - How you want to interpret the bits is up to you.

`0xfe4a10c5`

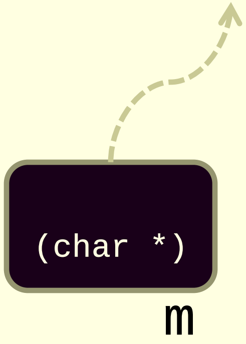
  - Is this... an `int` value?
    ... a pointer to a memory address?
    ... a series of `char` values?

Review of pointer variables

- A pointer variable is just a variable, that contains a value that we interpret as a memory address.
- Just like an uninitialized int variable holds some arbitrary “garbage” value,

  an uninitialized pointer variable points to some arbitrary “garbage address”

```
char *m;
```



How can you test whether a pointer points to something meaningful?

- There is a special pointer value `NULL`, that signifies “pointing to nothing”. You can also use the value `0`.

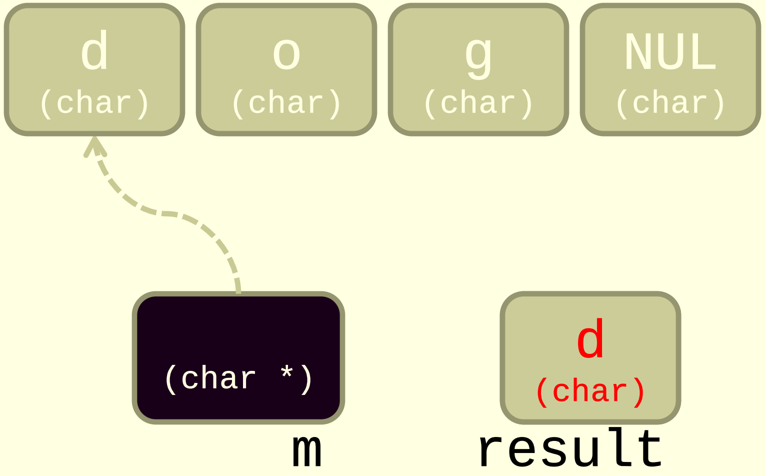
```
char *m = NULL;
...
if (m) { ... safe to follow the pointer ... }
```

- Here, `m` is used as a Boolean value
  - If `m` is “false”, aka `0`, aka `NULL`, it is not pointing to anything
  - Otherwise, it is (presumably) pointing to something good
  - Note: It is up to the *programmer* to assign `NULL` values when necessary

Indirection operator *

- Moves from address to contents

```
char  *m = ″dog″;

char result = *m;
```



`m` gives an address of a `char`

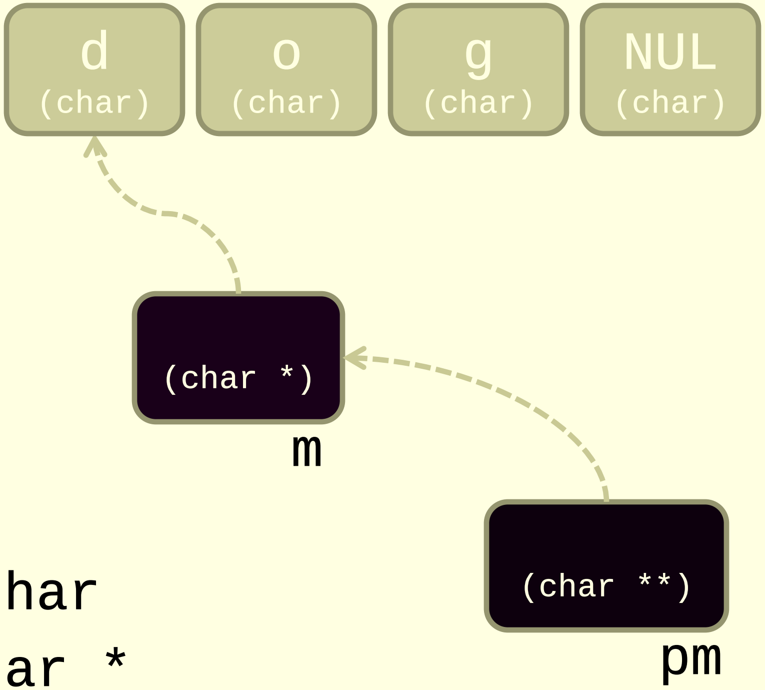
`*m` instructs us to take the contents of that address

`result` gets the value `′d′`

Address operator &

- Instead of contents, returns the address

```
char   *m = ″dog″,
       **pm = &m;
```



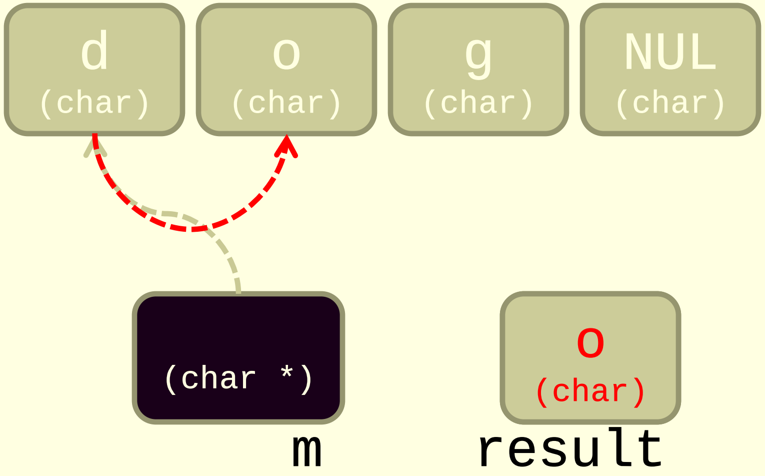
`pm` needs a value of type `char **`

- Can we give it `*m`? No – type is `char`
- Can we give it `m`? No – type is `char *`
- `&m` gives it the right value – the *address* of a `char *` value

Pointer arithmetic

- C allows pointer values to be incremented by integer values

```
char  *m = ″dog″;

char result = *(m + 1);
```



`m` gives an address of a `char`

`(m + 1)` gives the `char` one byte higher

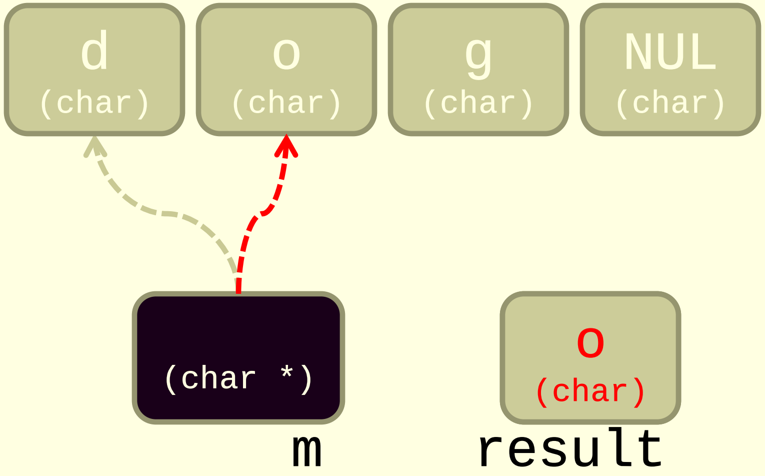
`*(m + 1)` instructs us to take the contents of that address

`result` gets the value `′o′`

Pointer arithmetic

- A slightly more complex example:

```
char  *m = "dog";

char result = *++m;
```



`m` gives an address of a `char`

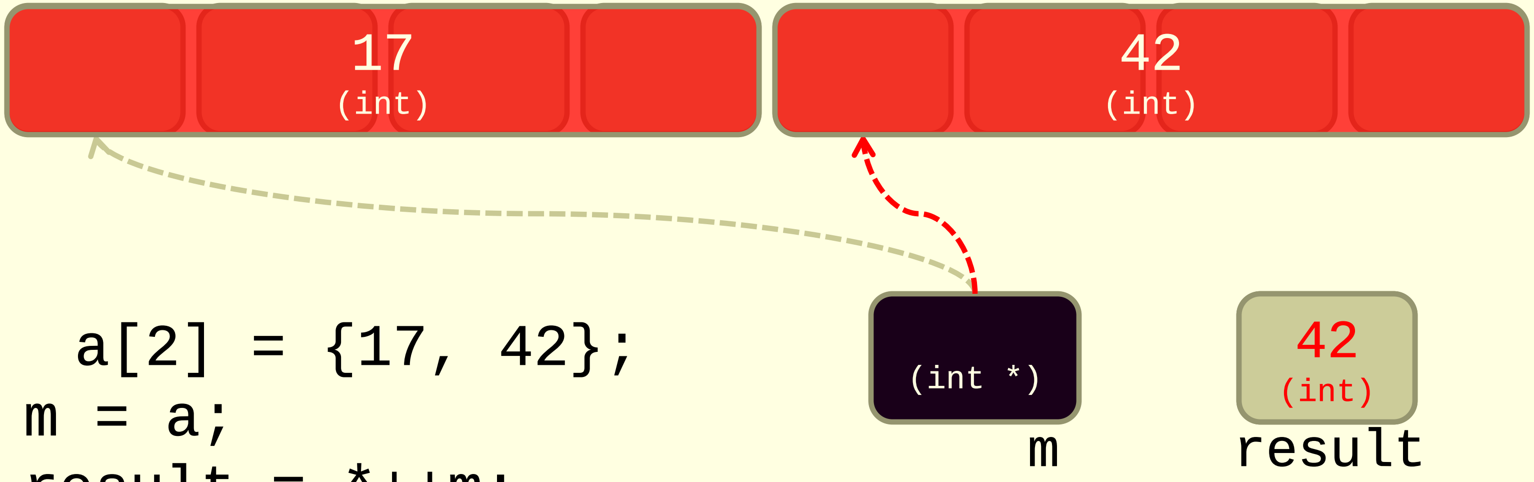
`++m` changes m, to the address one byte higher, and returns the new address

`*++m` instructs us to take the contents of that location

`result` gets the value `'o'`

Pointer arithmetic

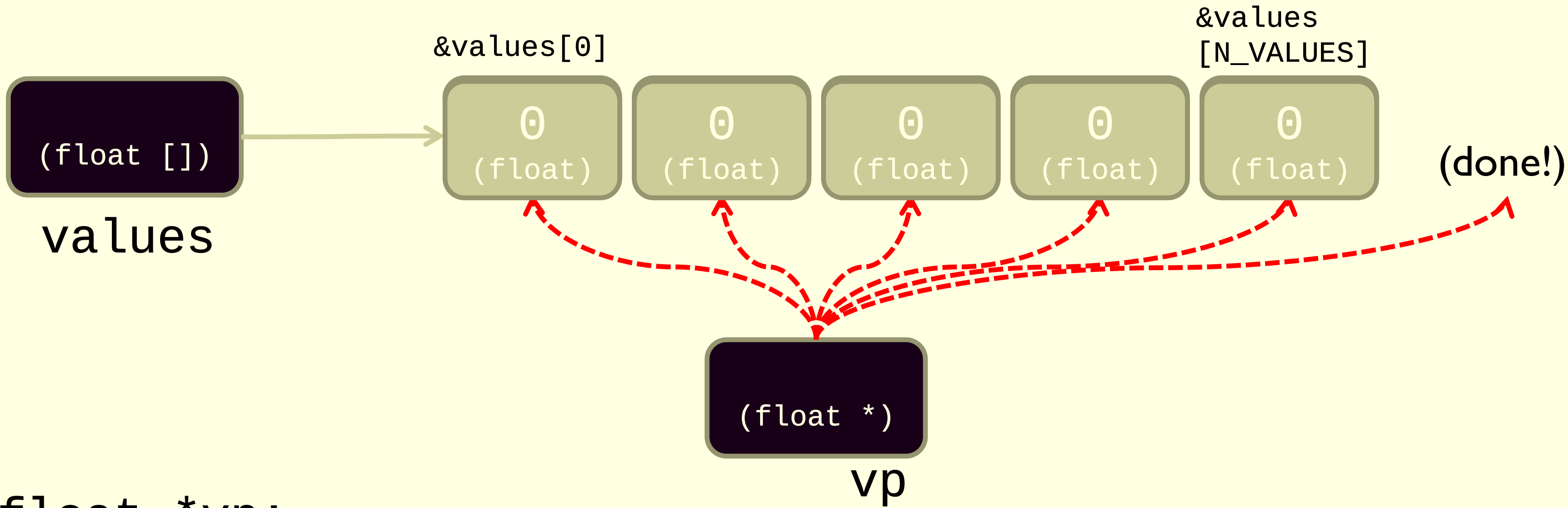
- How about multibyte values?
  - **Q:** Each `char` value occupies exactly one byte, so obviously incrementing the pointer by one takes you to a new `char` value... But what about types like `int` that span more than one byte?
  - **A:** C "does the right thing": increments the pointer by the size of one `int` value



```
int   a[2] = {17, 42};
int m = a;
int result = *++m;
```

Example: initializing an array

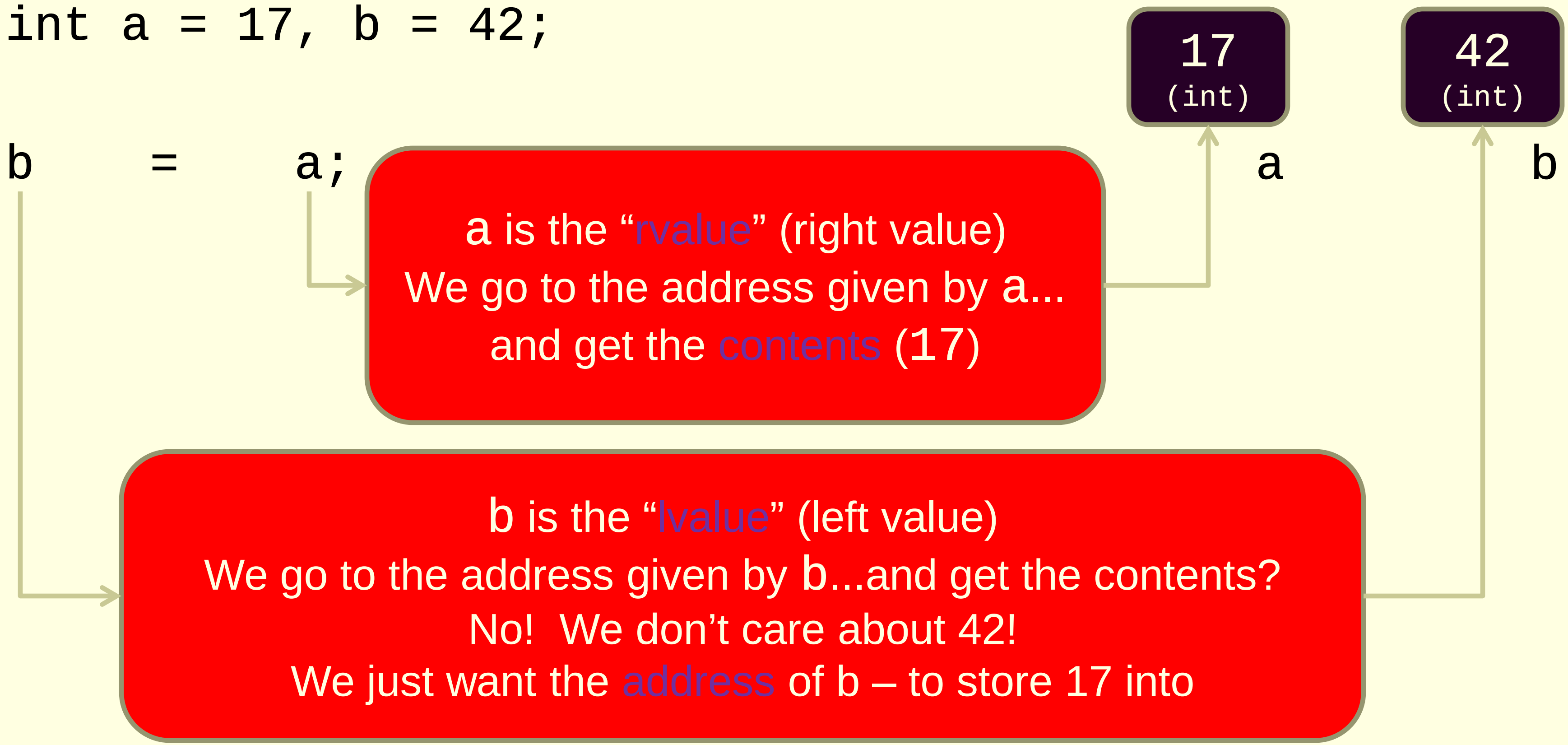
```
#define N_VALUES 5
float values[N_VALUES];
```



```
float *vp;
for ( vp = &values[0]; vp < &values[N_VALUES]; )
   *vp++ = 0;
```

A note on assignment: Rvalues vs. Lvalues

- What’s really going on in an assignment?
  - Different things happen on either side of the =



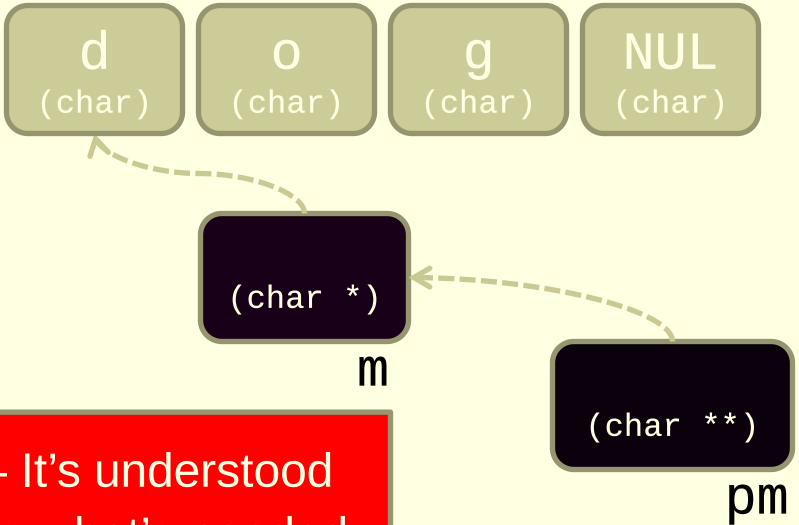
```
int a = 17, b = 42;

b     =     a;
```

a is the “rvalue” (right value)
We go to the address given by **a**…
and get the contents (17)

b is the “lvalue” (left value)
We go to the address given by **b**…and get the contents?
No! We don’t care about 42!
We just want the address of b – to store 17 into

A note on assignment: Rvalues vs. Lvalues

- This explains a certain “asymmetry” in assignments involving pointers:



```
char *m = NULL, **pm = NULL;

m = “dog”;

pm = &m;
```

Here, `m` is an *lvalue* – It’s understood that the *address* of `m` is what’s needed

Once again, we need the *address* of `m` – but since it’s an *rvalue*, just plain `m` will give the *contents* of `m` – use & to get the address instead

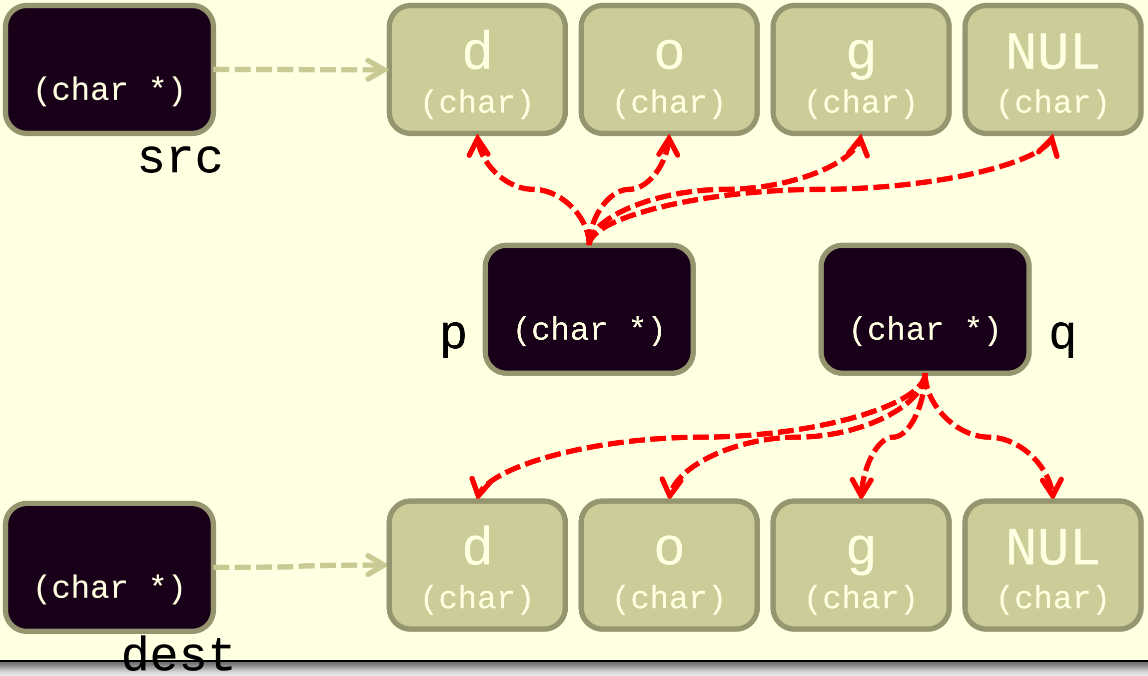
Example: `strcpy` “string copy”

```
char *strcpy(char *dest, const char *src)
```

- (assume that) `src` points to a sequence of `char` values that we wish to copy, terminated by NUL
- (assume that) `dest` points to an accessible portion of memory large enough to hold the copied chars
- `strcpy` copies the char values of `src` to the memory pointed to by `dest`
- `strcpy` also gives `dest` as a return value

Example: `strcpy` “string copy”

```
char *strcpy(char *dest, const char *src) {
   const char *p;
   char *q;
   for(p = src, q = dest; *p != '\0'; p++, q++)
      *q = *p;
   *q = '\0';
   return dest;
}
```



C-Programming Review

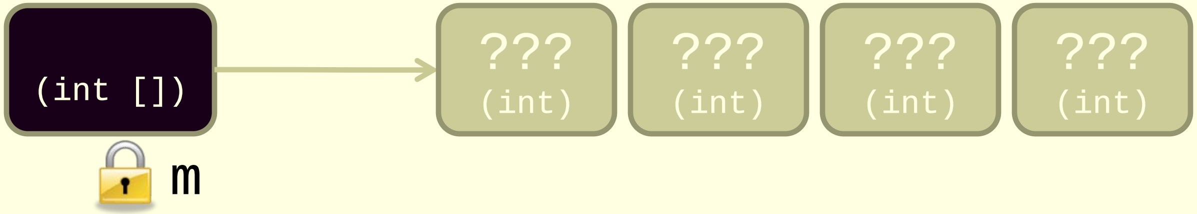
ARRAYS

Review of arrays

- There are no array variables in C – only array *names*
  - Each name refers to a constant pointer
  - Space for array elements is allocated at declaration time
- Can’t change where the array name refers to…
  - but you can change the array elements,

    via pointer arithmetic

`int m[4];`



(int [])

m

??? (int)

??? (int)

??? (int)

??? (int)

Subscripts and pointer arithmetic

- `array[subscript]` equivalent to `*(array + (subscript))`

- Strange but true: Given earlier declaration of `m`, the expression `2[m]` is legal!
  - Not only that: it's equivalent to `*(2+m)`

    `*(m+2)`

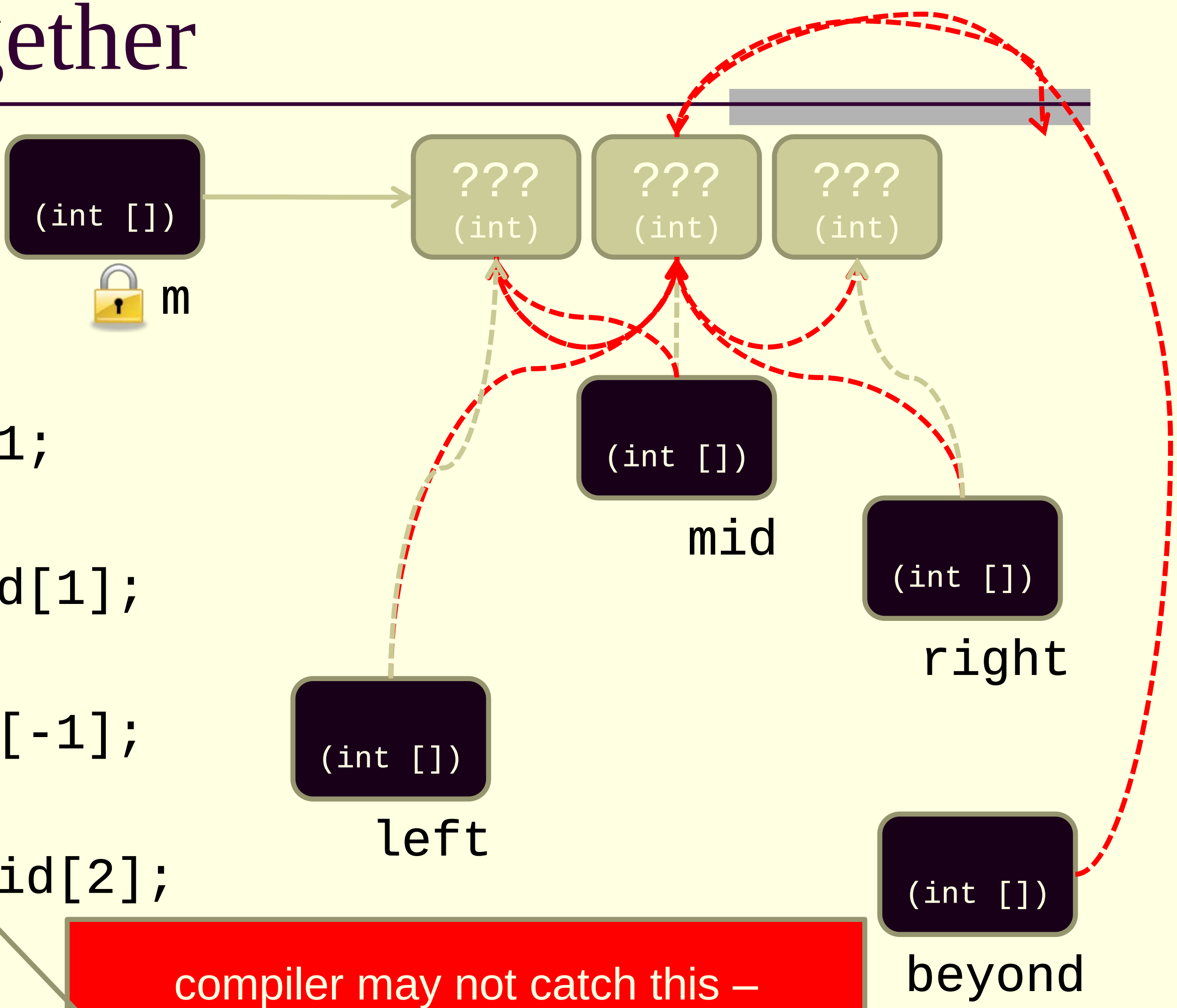
    `m[2]`

Array names and pointer variables, playing together

```
int m[3];
```

subscript OK with pointer variable

```
int *mid = m + 1;

int *right = mid[1];

int *left = mid[-1];

int *beyond = mid[2];
```

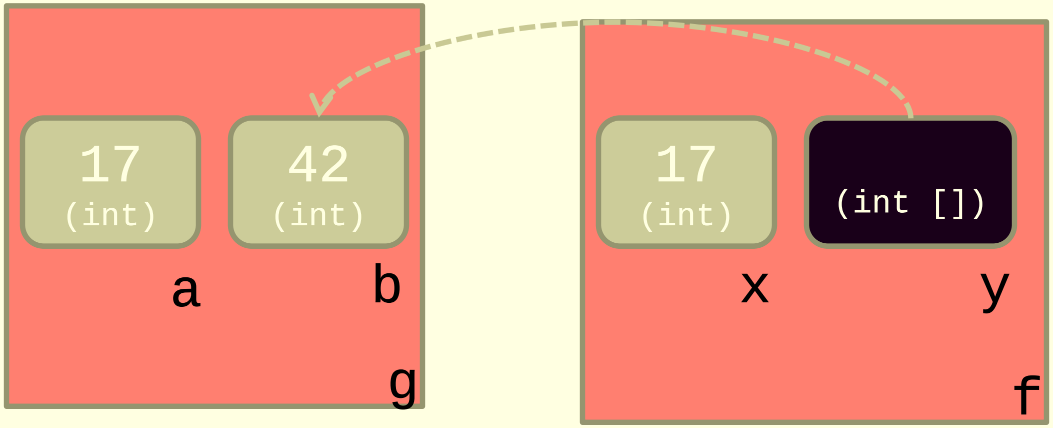


compiler may not catch this – runtime environment certainly won’t

Array names as function arguments

- In C, arguments are passed “by value”
  - A temporary copy of each argument is created, solely for use within the function call

```
void f(int x, int *y) { … }

void g(…) {
  int a = 17, b = 42;
  f(a, &b);
  …
}
```



- Pass-by-value is “safe” in that the function plays only in its “sandbox” of temporary variables –
  - can’t alter the values of variables in the callee (except via the return value)

Array names as function arguments

- But, functions that take arrays as arguments can exhibit *what looks like* “pass-by-reference” behavior, where the array passed in by the callee does get changed
  - Remember the special status of arrays in C –

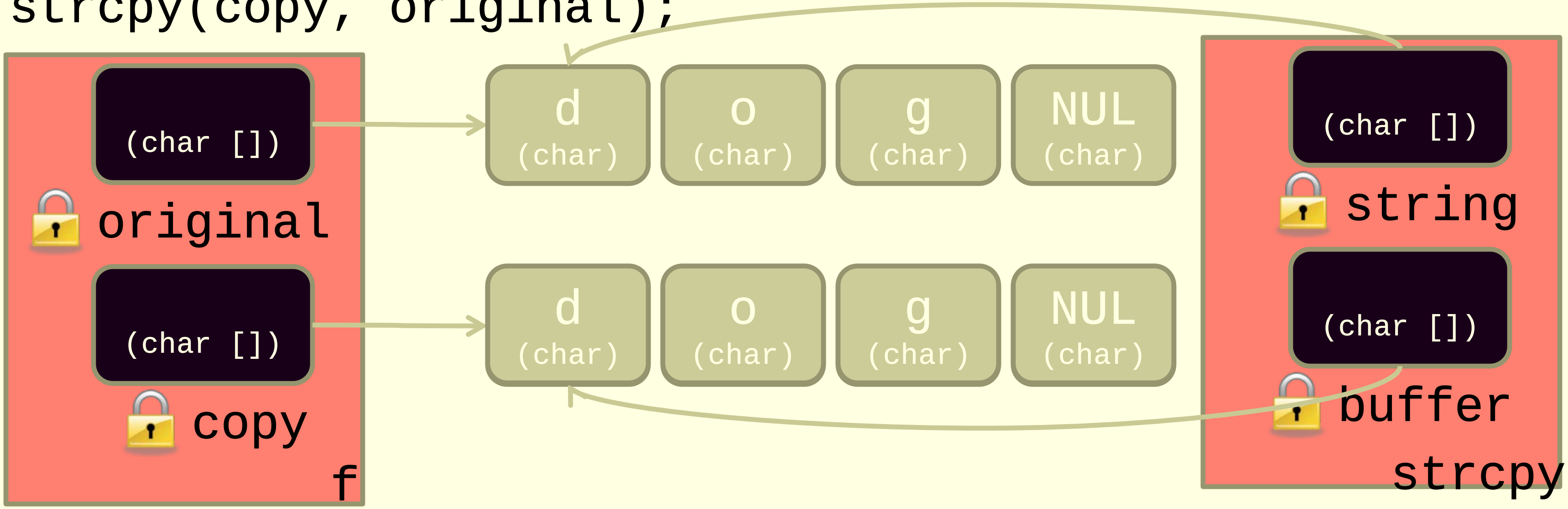
    They are basically just pointers.
  - So arrays are indeed passed by value –

    but only the pointer is copied, not the array elements!
    - Note the advantage in efficiency (avoids a lot of copying)
    - But – the pointer copy points to the same elements as the callee’s array
    - These elements can easily be modified via pointer manipulation

Array names as function arguments

- The `strcpy` “string copy” function puts this “pseudo” call-by-reference behavior to good use

```
void strcpy(char *buffer, char const *string);
void f(…) {
  char original[4] = ″dog″;
  char copy[4];
  strcpy(copy, original);
}
```



When can array size be omitted?

- There are a couple of contexts in which an array declaration need not have a size specified:
  - Parameter declaration:

    ```
    int strlen(char string[]);
    ```

    - As we’ve seen, the elements of the array argument are not copied, so the function doesn’t need to know how many elements there are.
  - Array initialization:

    ```
    int vector[] = {1, 2, 3, 4, 5};
    ```

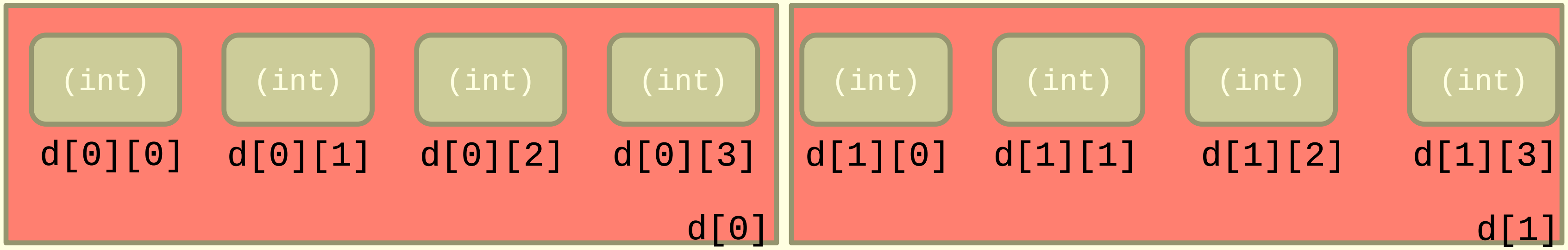
    - In this case, just enough space is allocated to fit all (five) elements of the initializer list

Multidimensional arrays

- How to interpret a declaration like:

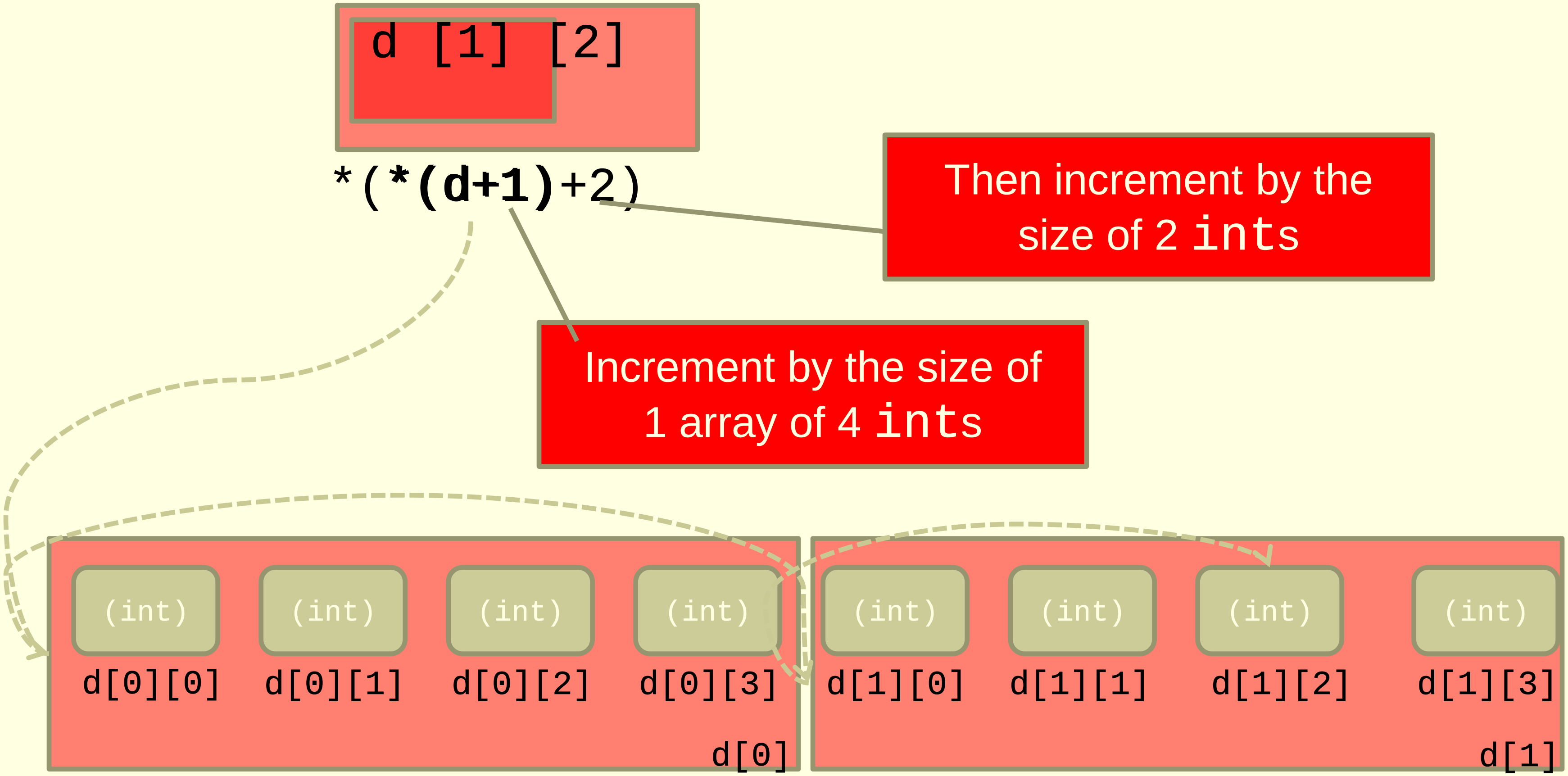
  `int d[2][4];`

- This is an array with two elements:
  - Each element is an array of four `int` values
- The elements are laid out sequentially in memory, just like a one-dimensional array
  - Row-major order: the elements of the *rightmost* subscript are stored contiguously



Subscripting in a multidimensional array

```
int d[2][4];
```



`d [1] [2]`

`*(*(d+1)+2)`

Then increment by the size of 2 `ints`

Increment by the size of 1 array of 4 `ints`

Why do we care about storage order?

- If you keep within the “paradigm” of the multidimensional array, the order doesn’t matter…
- But if you use tricks with pointer arithmetic, it matters a lot
- It also matters for initialization
    - To initialize `d` like this:

0	1	2	3
4	5	6	7

        - use this:

```
int d[2][4] = {0, 1, 2, 3, 4, 5, 6, 7};
```

        - rather than this

```
int d[2][4] = {0, 4, 1, 5, 2, 6, 3, 7};
```

Multidimensional arrays as parameters

- Only the first subscript may be left unspecified

```
void f(int matrix[][10]);      /* OK */
void g(int (*matrix)[10]);     /* OK */
void h(int matrix[][]);        /* not OK */
```

- Why?
  - Because the other sizes are needed for scaling when evaluating subscript expressions (see slide 10)
  - This points out an important drawback to C:

    Arrays do not carry information about their own sizes!

    If array size is needed, you must supply it somehow

    (e.g., when passing an array argument, you often have to pass an additional “array size” argument) – bummer

C-Programming Review

STRINGS

Review of strings

- Sequence of zero or more characters, terminated by `NUL` (literally, the integer value `0`)
- `NUL` terminates a string, but isn’t part of it
  - important for `strlen()` – length doesn’t include the `NUL`
- Strings are accessed through pointers/array names
- `string.h` contains prototypes of many useful functions

String literals

- Evaluating `″dog″` results in memory allocated for three characters `′d ′`, `′ o ′`, `′ g ′`, plus terminating `NUL`

```
char *m = ″dog″;
```

- Note: If `m` is an array name, subtle difference:

```
char m[10] = ″dog″;
```

10 bytes are allocated for this array

This is not a string literal;
It’s an array initializer in disguise!
Equivalent to
`{′d′,′o′,′g′,′\0′}`

String manipulation functions

- Read some “source” string(s), possibly write to some “destination” location

```
char *strcpy(char *dst, char const *src);
char *strcat (char *dst, char const *src);
```

- Programmer’s responsibility to ensure that:
  - destination region large enough to hold result
  - source, destination regions don’t overlap
    - “undefined” behavior in this case –

according to C spec, anything could happen!

```
char m[10] = ″dog″;
strcpy(m+1, m);
```

Assuming that the implementation of `strcpy` starts copying left-to-right without checking for the presence of a terminating NUL first, what will happen?

`strlen()` and `size_t`

```
size_t strlen(char const *string);
/* returns length of string */
```

- `size_t` is an unsigned integer type, used to define sizes of strings and (other) memory blocks
  - Reasonable to think of “size” as unsigned”...
  - But beware! Expressions involving `strlen()` may be unsigned (perhaps unexpectedly)

```
if (strlen(x) – strlen(y) >= 0) ...
```

always true!

- avoid by casting:

  `((int) (strlen(x) – strlen(y)) >= 0)`

  - Problem: what if `x` or `y` is a very large string?
- a better alternative: `(strlen(x) >= strlen(y))`

`strcmp()` "string comparison"

```
int strcmp(char const *s1, char const *s2);
```

  - returns a value less than zero if `s1` precedes `s2` in lexicographical order;
  - returns zero if `s1` and `s2` are equal;
  - returns a value greater than zero if `s1` follows `s2`.
- Source of a common mistake:
  - seems reasonable to assume that `strcmp` returns "true" (nonzero) if `s1` and `s2` are equal; "false" (zero) otherwise
  - In fact, *exactly the opposite* is the case!

C-Programming Review

STRUCTS

C structures: aggregate, yet scalar

- aggregate in that they hold multiple data items at one time
  - named *members* hold data items of various types
  - like the notion of class/field in C or C++

    – but without the data hiding features
- scalar in that C treats each structure as a unit
  - as opposed to the "array" approach: a pointer to a collection of members in memory
  - entire structures (not just pointers to structures) may be passed as function arguments, assigned to variables, etc.
  - Interestingly, they cannot be compared using ==

    (rationale: too inefficient)

Structure declarations

- Combined variable and type declaration

```
struct tag {member-list} variable-list;
```

- Any one of the three portions can be omitted

```
struct {int a, b; char *p;} x, y;  /* omit tag */
```

  - variables `x, y` declared with members as described: `int` members `a, b` and `char` pointer `p`.
  - `x` and `y` have same type, but differ from all others – even if there is another declaration:

```
struct {int a, b; char *p;} z;
/* z has different type from x, y */
```

Structure declarations

```
struct S {int a, b; char *p;};  /* omit variables */
```

- No variables are declared, but there is now a type `struct s` that can be referred to later

```
struct S z;  /* omit members */
```

  - Given an earlier declaration of `struct S`, this declares a variable of that type

```
typedef struct {int a, b; char *p;} S;
  /* omit both tag and variables */
```

  - This creates a simple type name `s`

    (more convenient than `struct S`)

Recursively defined structures

- Obviously, you can’t have a structure that contains an instance of itself as a member – such a data item would be infinitely large
- But within a structure you can *refer* to structures of the same type, via pointers

```
struct TREENODE {
  char *label;
  struct TREENODE *leftchild, *rightchild;
}
```

Recursively defined structures

- When two structures refer to each other, one must be declared in incomplete (prototype) fashion

```
struct HUMAN;
struct PET {
  char name[NAME_LIMIT];
  char species[NAME_LIMIT];
  struct HUMAN *owner;
} fido = {″Fido″, ″Canis lupus familiaris″};
struct HUMAN {
  char name[NAME_LIMIT];
  struct PET pets[PET_LIMIT]
} sam = {″Sam″, {fido}};
```

We can't initialize the `owner` member at this point, since it hasn't been declared yet

Member access

- Direct access operator `s.m`
  - subscript and dot operators have same precedence and associate left-to-right, so we don't need parentheses for `sam.pets[0].species`
- Indirect access `s->m`: equivalent to `(*s).m`
  - Dereference a pointer to a structure, then return a member of that structure
  - Dot operator has higher precedence than indirection operator , so parentheses are needed in (*s).m

`(*fido.owner).name` or `fido.owner->name`

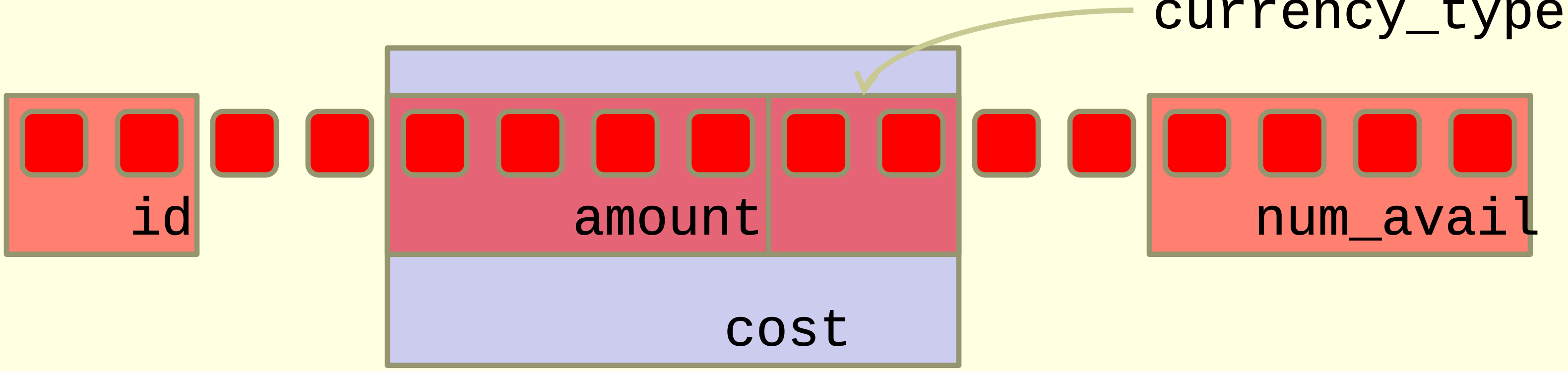
`.` evaluated first: access `owner` member
`*` evaluated next: dereference pointer to `HUMAN`

`.` and `->` have equal precedence and associate left-to-right

Memory layout

```
struct COST { int amount;
              char currency_type[2]; }
struct PART { char id[2];
              struct COST cost;
              int num_avail; }
```

layout of `struct PART`:

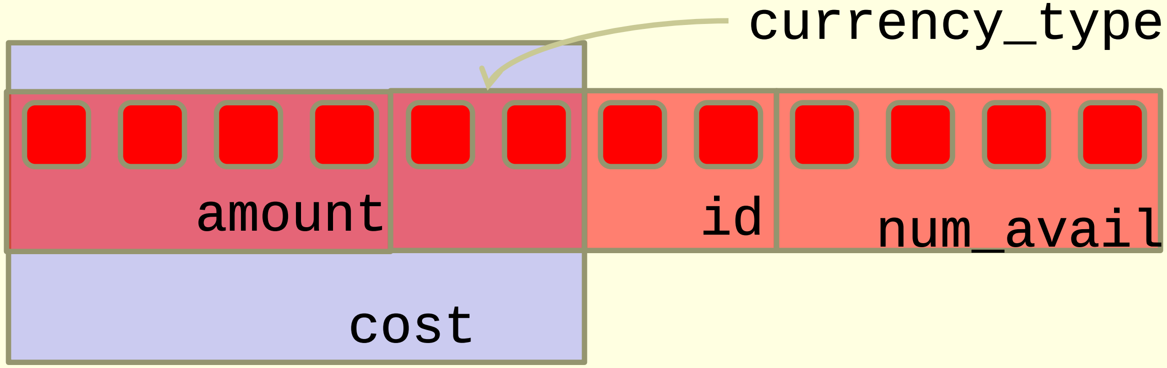


Here, the system uses 4-byte alignment of integers, so `amount` and `num_avail` must be aligned
Four bytes wasted for each structure!

Memory layout

A better alternative (from a space perspective):

```
struct COST { int amount;
              char currency_type; }
struct PART { struct COST cost;
              char id[2];
              int num_avail;
}
```



Structures as function arguments

- Structures are scalars, so they can be returned and passed as arguments – just like `int`s, `char`s

```
struct BIG changestruct(struct BIG s);
```

  - Call by value: temporary copy of structure is created
  - Caution: passing large structures is inefficient – involves a lot of copying
- avoid by passing a pointer to the structure instead:

```
void changestruct(struct BIG *s);
```

- What if the `struct` argument is read-only?
  - Safe approach: use `const`

C-Programming Review

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I