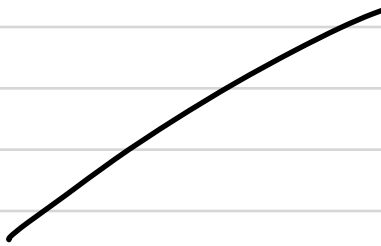


Context through Declarations

- context sensitivity through declarations
 - a variable declaration lies in the context of a statement that uses the variable
 - how does the parser handle declarations?
 - symbol tables!

- every declaration results in a new symbol table entry
 - every occurrence of an identifier in a statement requires a search of the symbol table to determine the attributes (properties) of the object denoted by the identifier
- 

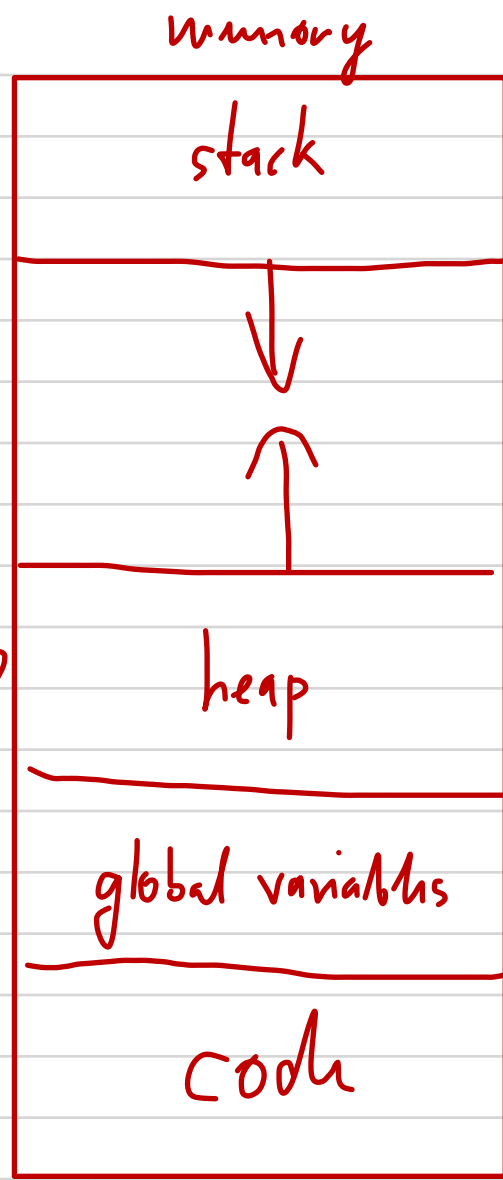
• Required!

Arrays

• `int i;`
`i = 15;`
`f(i);`

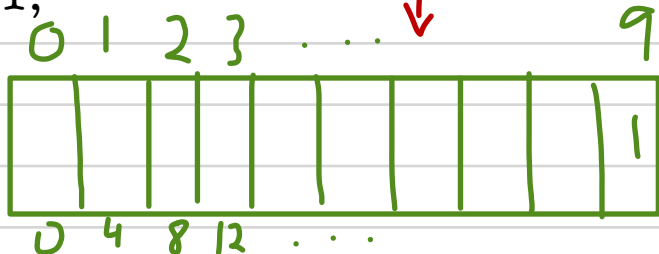
4 bytes
 $0 \times 000F$
32 bit

byte-addressed
word-aligned

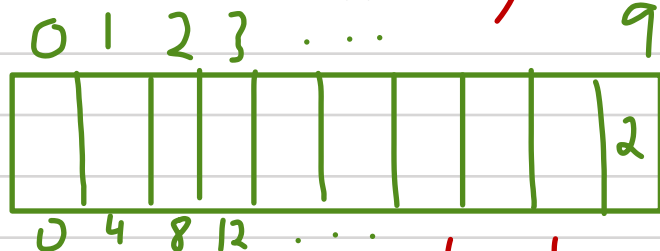


`int a1[10];`
`a1[9] = 1;`
`f(a1);`

sequential allocation



`int *a2;` ← anonymous type!
`a2 = malloc(10 * sizeof(int));`
`a2[9] = 2;`
`f(a2);`



no size

• every type has a size
• every variable has an address!

`typedef int *array_t;`
`array_t a3;`

`a3 = malloc(10 * sizeof(int));`
`a3[9] = 3;`
`f(a3);`

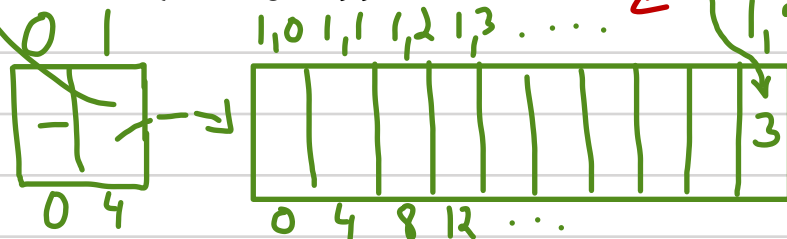


n : number of array elements
 $Size("array") = n \cdot size(element)$

$address(a[i]) = address(a) + i \cdot size(element)$

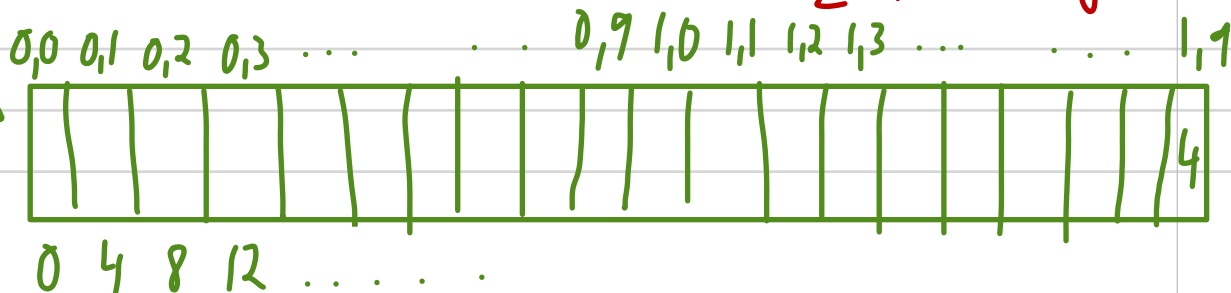
`typedef array_t *array_of_arrays_t;`
`array_of_arrays_t a4;`

`a4 = malloc(2 * sizeof(array_t));`
`a4[1] = a3;`
`i = a4[1][9];`
`f(a4);`



multidimensional:
1. array of arrays
2. contiguous (row-major)

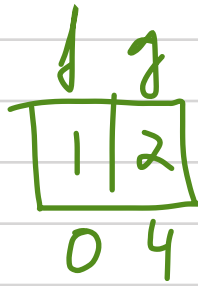
`int a5[2][10];`
`a5[1][9] = 4;`
`f(a5);`



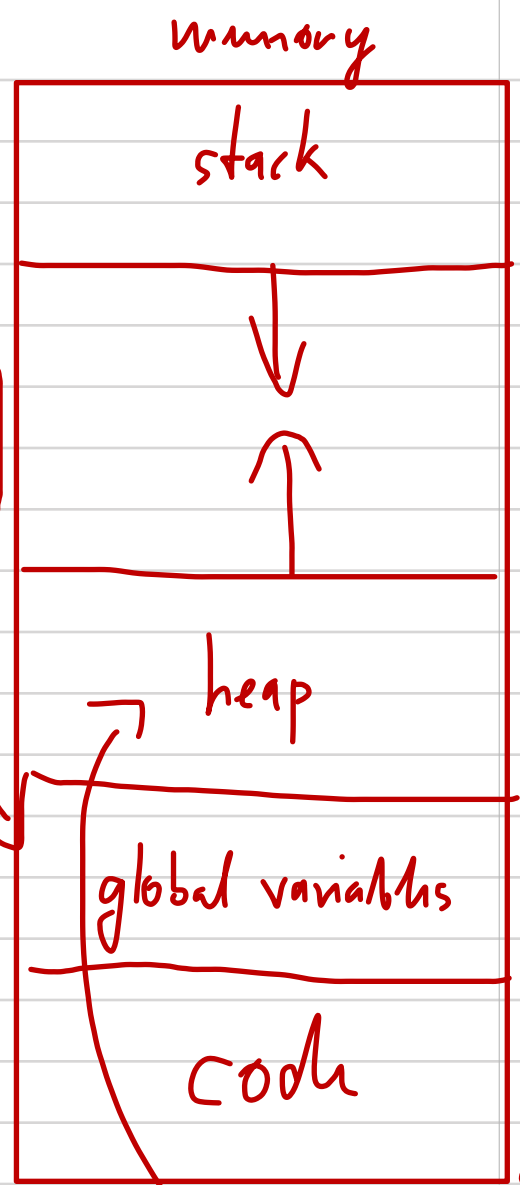
• required!

Records

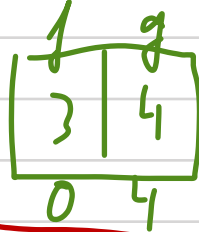
```
struct record_t {
    int f;
    int g;
};
struct record_t r1;
r1.f = 1;
r1.g = 2;
f(r1);
```



← copies r1 onto the stack!



```
struct record_t *r2;
r2 = malloc(sizeof(struct record_t));
r2->f = 3;
r2->g = 4;
f(r2);
```

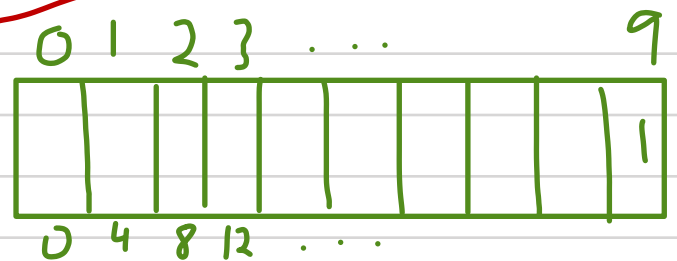
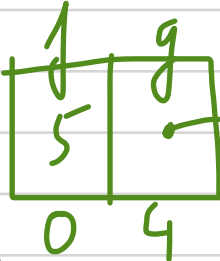


pushes only
the memory
onto stack

```
struct record_of_record_t;
struct record_of_record_t {
    int f;
    struct record_of_record_t *g;
};
```

forward declaration

```
struct record_of_record_t *r3;
r3 = malloc(sizeof(struct record_of_record_t));
r3->f = 5;
r3->g = r3;
f(r3);
```

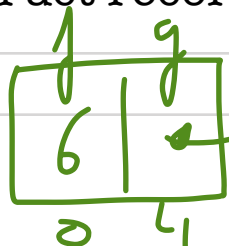


```
typedef int *array_t;
array_t a1;
a1 = malloc(10 * sizeof(int));
struct record_of_array_t {
    int f;
    array_t g;
};
```

$\text{struct } r_t \{ f_0_t f_0; \dots; f_{n-1_t} f_{n-1}; \};$
 $\text{struct } r_t *r;$

$$\text{size}(r_t) = \sum_{0 \leq i < n} \text{size}(f_{i_t})$$

```
struct record_of_array_t *r4;
r4 = malloc(sizeof(struct record_of_array_t));
r4->f = 6;
r4->g = a1;
f(r4);
```



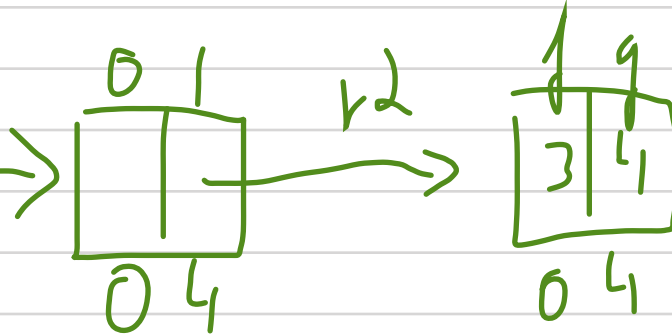
$$\text{address}(r \rightarrow f_i) = \text{address}(r) + \sum_{0 \leq j < i} \text{size}(f_{j_t})$$

Array of Records

```
typedef struct record_t * *array_of_record_references_t;  
array_of_record_references_t a2;
```

```
a2 = malloc(2 * sizeof(struct record_t *));  
a2[1] = r2;
```

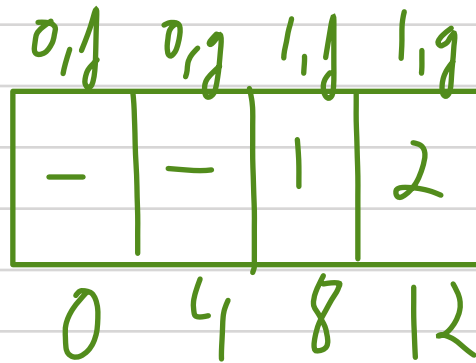
• required



```
typedef struct record_t *array_of_records_t;  
array_of_records_t a3;
```

• optional

```
a3 = malloc(2 * sizeof(struct record_t));  
a3[1] = r1;
```



copies r1 into a3[1]

Symbol Table ~> save in object file (partially)

```
int i;
```

```
struct object_t;
```

```
typedef int *array_t;
```

```
struct type_t {
```

```
    int form;
```

```
    struct object_t *fields;
```

```
    struct type_t *base;
```

```
};
```

```
array_t a1;
```

```
array_t a2;
```

```
struct record_t {
```

```
    int f;
```

```
    int g;
```

```
}
```

```
typedef char *string_t;
```

```
struct object_t {
```

```
    string_t name;
```

```
    int class;
```

```
    struct type_t *type;
```

```
    struct object_t *next;
```

```
};
```

```
struct record_t *r1;
```

length
of
arrays

• lists vs. trees: linear ok because
symbol tables are "small"

