APS105
Winter 2012

Jonathan Deber
jdeber -at- cs -dot- toronto -dot- edu

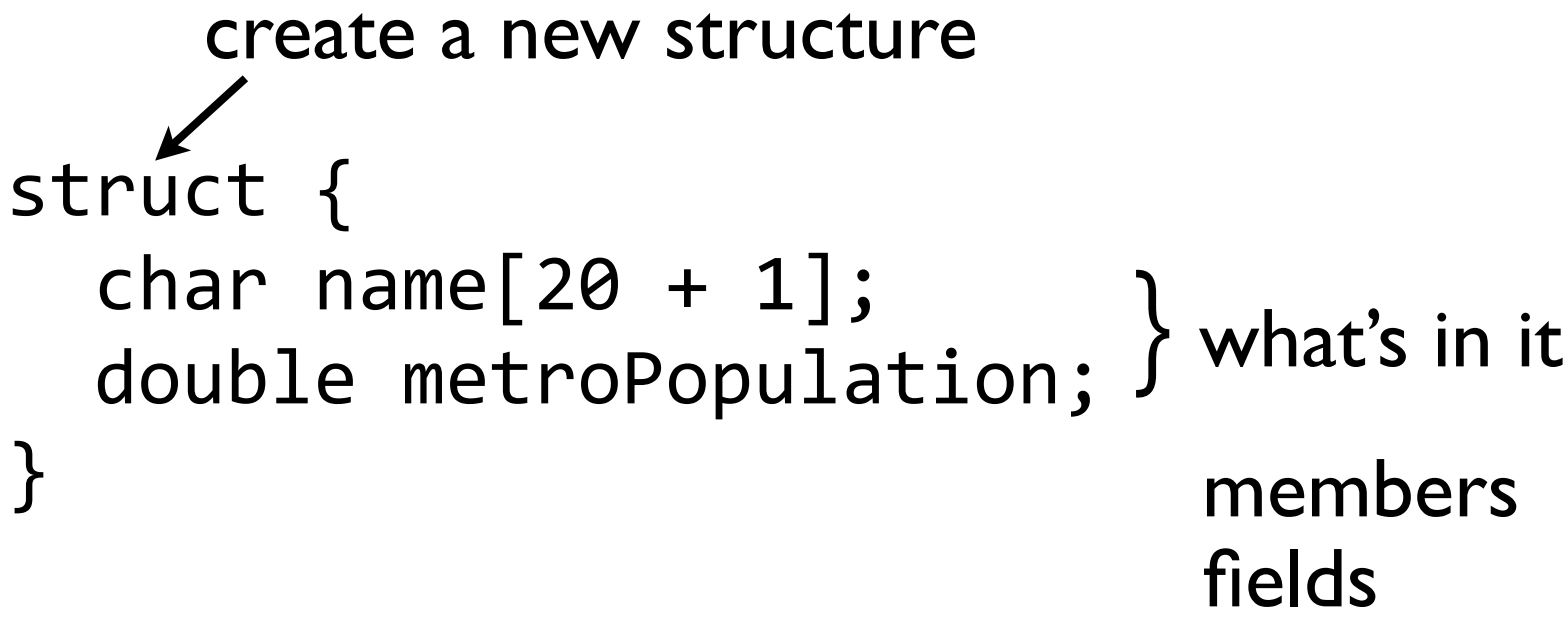
Lecture 28
March 26, 2012

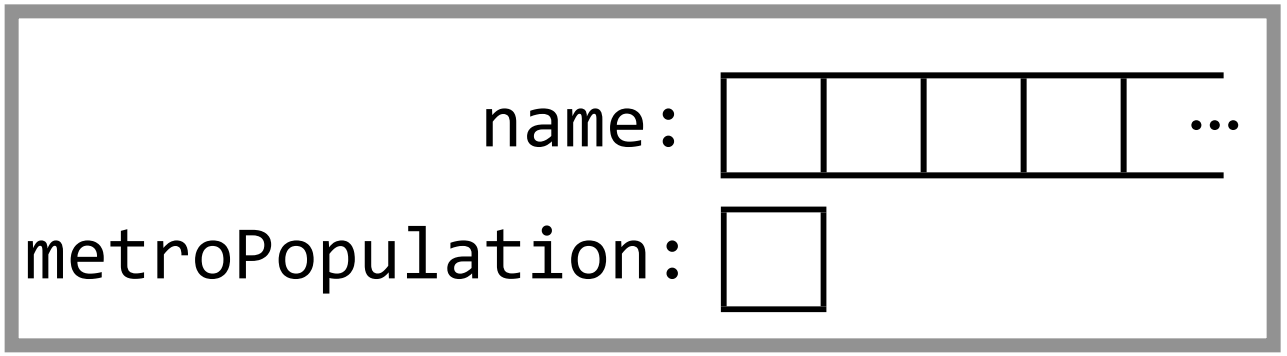
Today

- `struct`
- Data Structures

Structures

- Way of grouping variables together into a single unit
- Acts as a template to create instances of that grouping





This code is useless on its own

```
struct {
  char name[20 + 1];
  double metroPopulation;
}
```

```
  ??? toronto;
```

```
int
```

```
int i;
```

```
struct city {
  char name[20 + 1];
  double metroPopulation;
};
```
← Watch this ;

```
struct city toronto = {"Toronto", 6.324};
```

```
int i;                    Type of i is int
int *p;                   Type of p is int *
struct city toronto;      Type of toronto is struct city
city vancouver;           Type of vancouver is city
```

Error

typedef

- A way of defining a new type
- More accurately, a new name for an existing type

```
typedef existing-type new-type;

typedef double Dollar;

Dollar price = 1.99;

Dollar calculateTax(Dollar d)
{
  ...
}
```

```
typedef struct city {
   char name[20 + 1];
   double metroPopulation;
} City;
```

create a `struct` named `city`

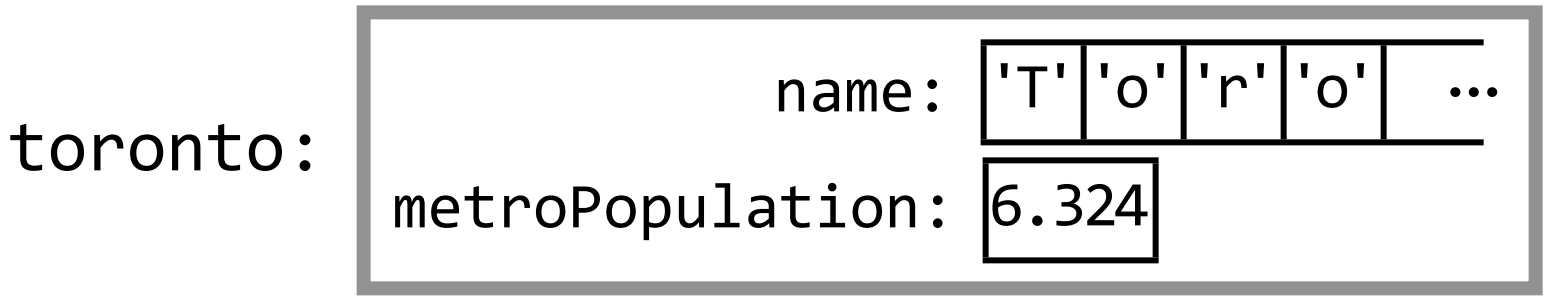
```
typedef struct city { ... } City;
```

create a `typedef` for `struct city` called `City`

```
City toronto = {"Toronto", 6.324};
```

Type of `toronto` is `City`

Dot Notation



toronto.name
toronto.metroPopulation } lvalues

```
City toronto;
toronto.metroPopulation = 6.324;
toronto.name = "Toronto";
strncpy(toronto.name, "Toronto",
            sizeof(toronto.name));
printf("The pop of %s is %g million\n",
          toronto.name, toronto.metroPopulation);
```

Error

```
int a[3];
a[0]
```

```
The pop of Toronto is 6.324 million
```

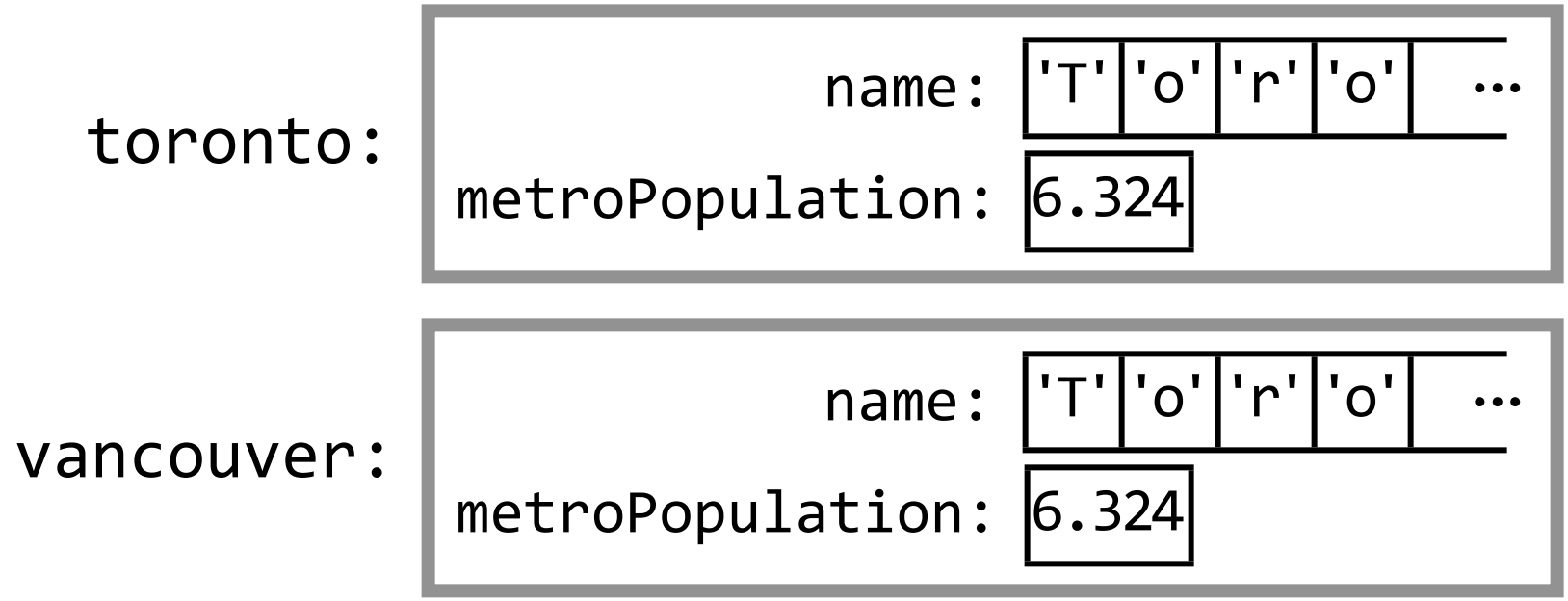
Definition Location

- `structs` can be defined most places in your program

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

typedef struct city {
  char name[20 + 1];
  double metroPopulation;
} City;

int main(void)
{
  ...
}
```

structs and Operators

- Assignment operator works (everything is copied)




```
City toronto = {"Toronto", 6.324};

City vancouver = toronto;
```

`structs` and Operators

- Relational operators do not work
- Arithmetic operators do not work
- Equality operators do not work

```
City toronto = {"Toronto", 6.324};
City vancouver = toronto;
```

```
toronto > vancouver
```
Error

```
toronto + vancouver
```
Error

```
toronto == vancouver
```
Error

Comparison

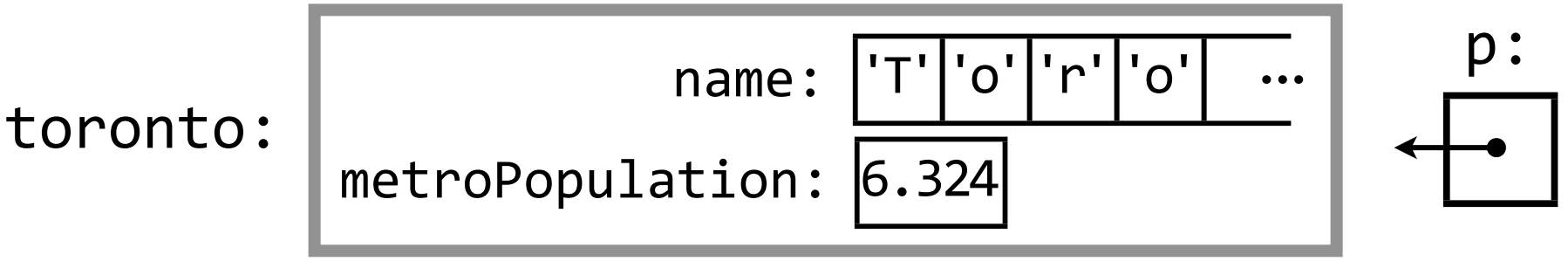
- If you need to compare `structs`, you can write your own comparison function similar to `strcmp()`

```
int compareCity(City c1, City c2)
{
  return c1.metroPopulation - c2.metroPopulation;
}


if (compareCity(toronto, vancouver) > 0)
{
  ...
}
```

Pointers and structs

- You can get a pointer to a `struct`

```
City toronto = {"Toronto", 6.324};
City *p = &toronto;
```



`*p.name` Error

```
error: request for member ‘name’ in
 something not a structure or union
```

`*p.name` means `*(p.name)`

`(*p).name`

Too Much Typing

- Dennis Ritchie thought this was too much typing for a very common operation
- So he added a shortcut, the right arrow operator
- Only valid when used on a pointer to a `struct`

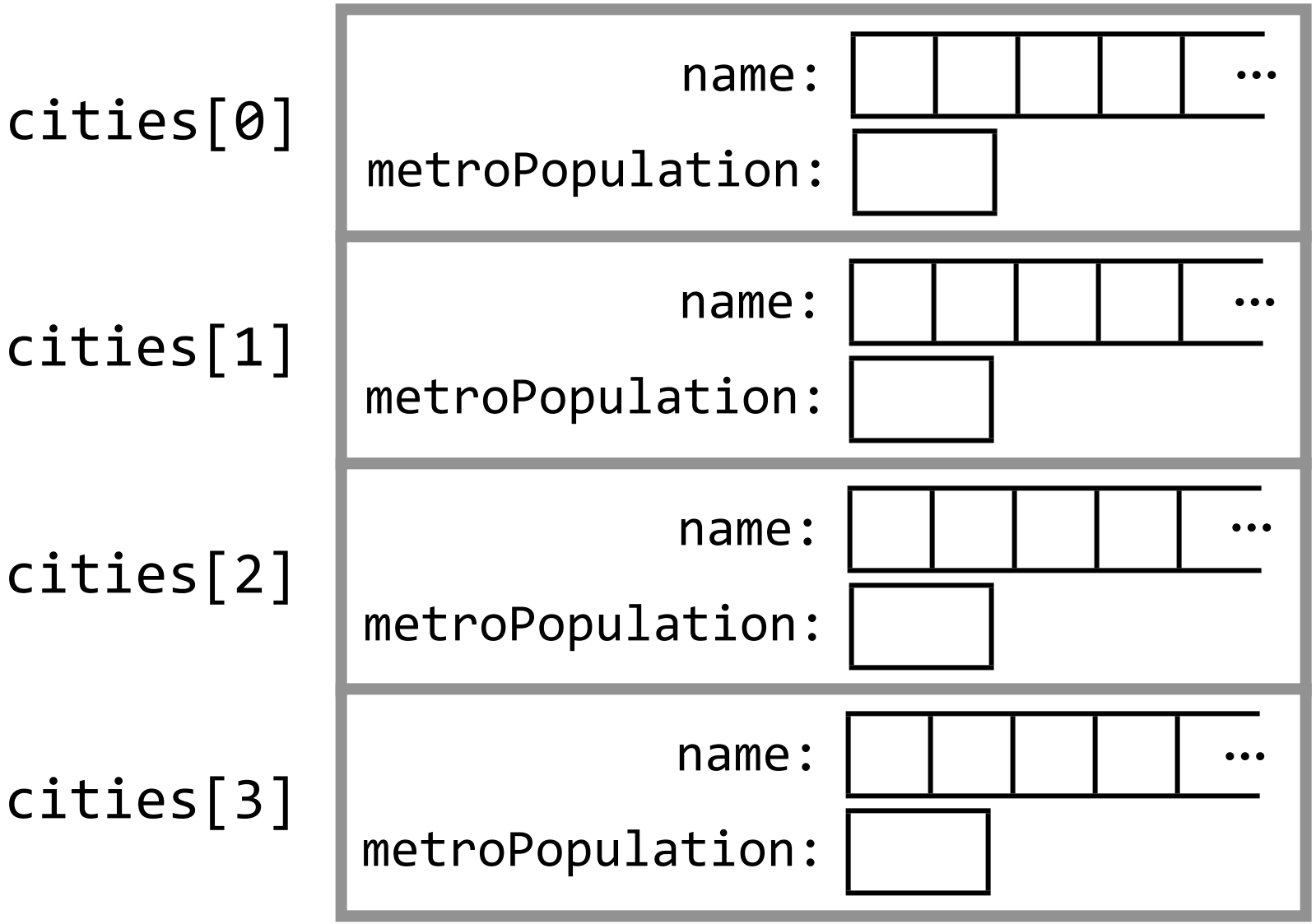
```
(*p).name
```

```
p->name
```

Saves two whole characters!

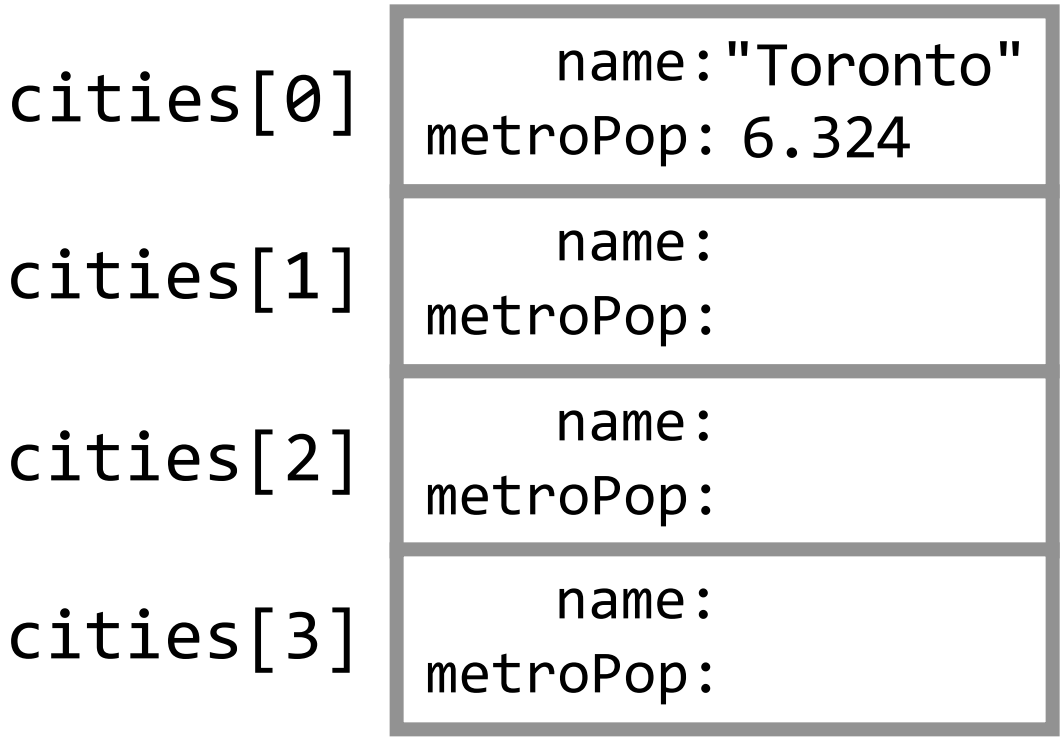
Arrays of `structs`

```
City cities[4];
```



Arrays of structs

```
City cities[4];
```




```
cities[0].metroPopulation = 6.324;
strncpy(cities[0].name, "Toronto",
            sizeof(cities[0].name));
```

Initializers

```
City cities[] = { {"Halifax",   0.283},
                  {"Montreal",  3.764},
                  {"Toronto",   6.324},
                  {"Vancouver", 2.254} };


for (int i = 0; i < 4; i++)
{
  printf("The pop of %s is %g million\n",
         cities[i].name, cities[i].metroPopulation);
}
```

```
The pop of Halifax is 0.283 million
The pop of Montreal is 3.764 million
The pop of Toronto is 6.324 million
The pop of Vancouver is 2.254 million
```

sizeof() structs

`sizeof(City)` approx the sum of the sizes of each member (very implementation specific)

```
City cities[] = { {"Halifax",   0.283},
                  {"Montreal",  3.764},
                  {"Toronto",   6.324},
                  {"Vancouver", 2.254} };
```

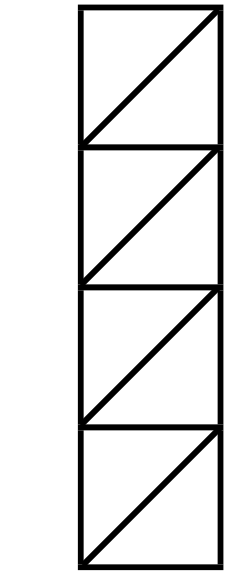
`sizeof(cities)` number of elements * `sizeof(City)`

`sizeof(cities) / sizeof(City)` number of elements

Tip: `sizeof(cities) / sizeof(cities[0])`

Arrays of Pointers

```
#define N 4

City toronto = {"Toronto", 6.324};
City vancouver = {"Vancouver", 2.254};
City *cities2[N];
for (int i = 0; i < N; i++)
{
  cities2[i] = NULL;
}
```



cities2

toronto:

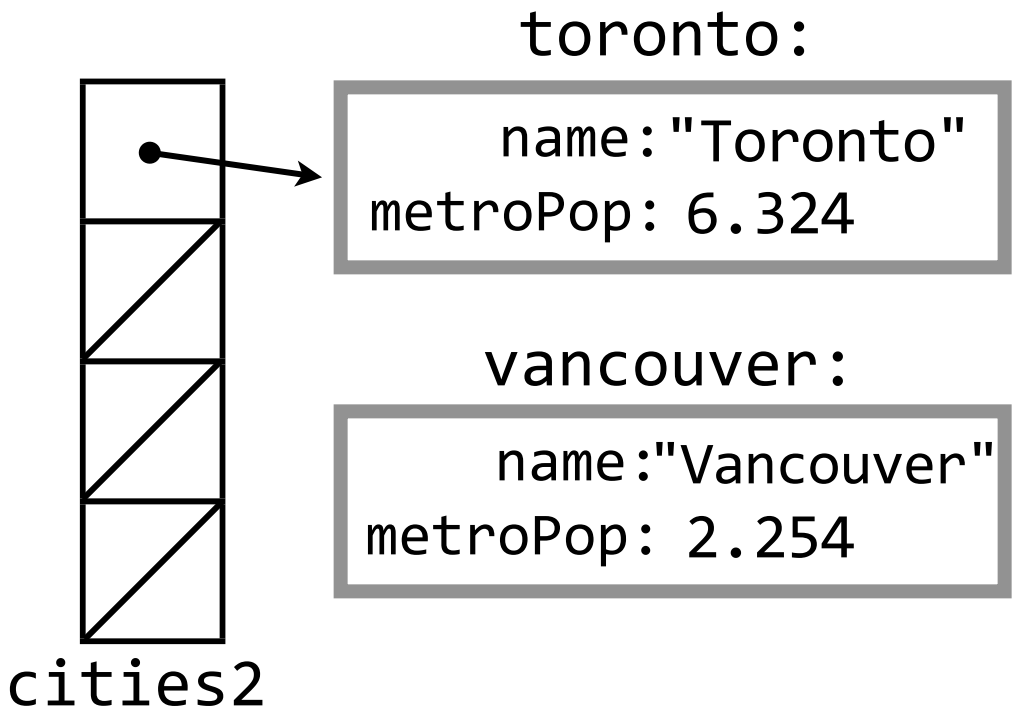
```
    name:"Toronto"
metroPop: 6.324
```

vancouver:

```
    name:"Vancouver"
metroPop: 2.254
```

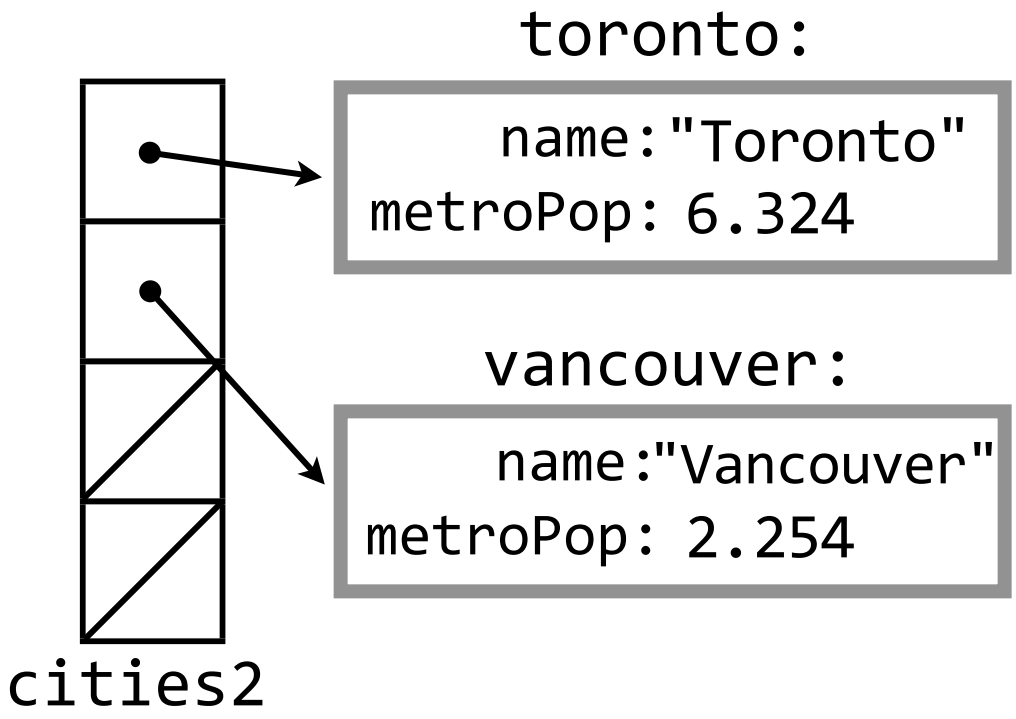
Arrays of Pointers

```
#define N 4

City toronto = {"Toronto", 6.324};
City vancouver = {"Vancouver", 2.254};
City *cities2[N];
for (int i = 0; i < N; i++)
{
  cities2[i] = NULL;
}
cities2[0] = &toronto;
```

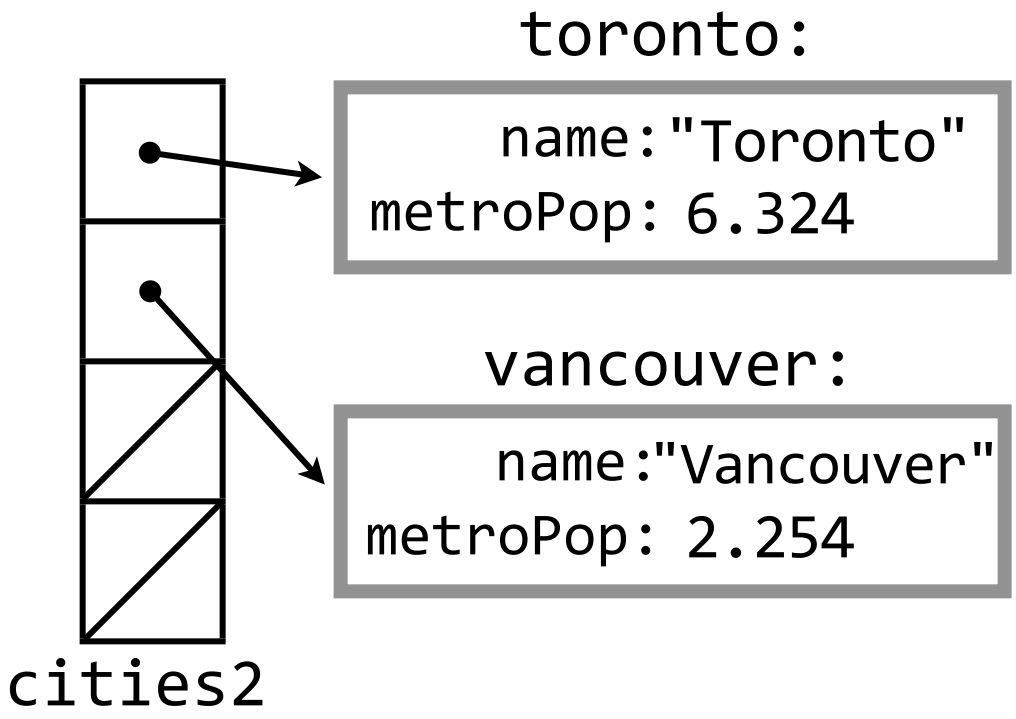


Arrays of Pointers

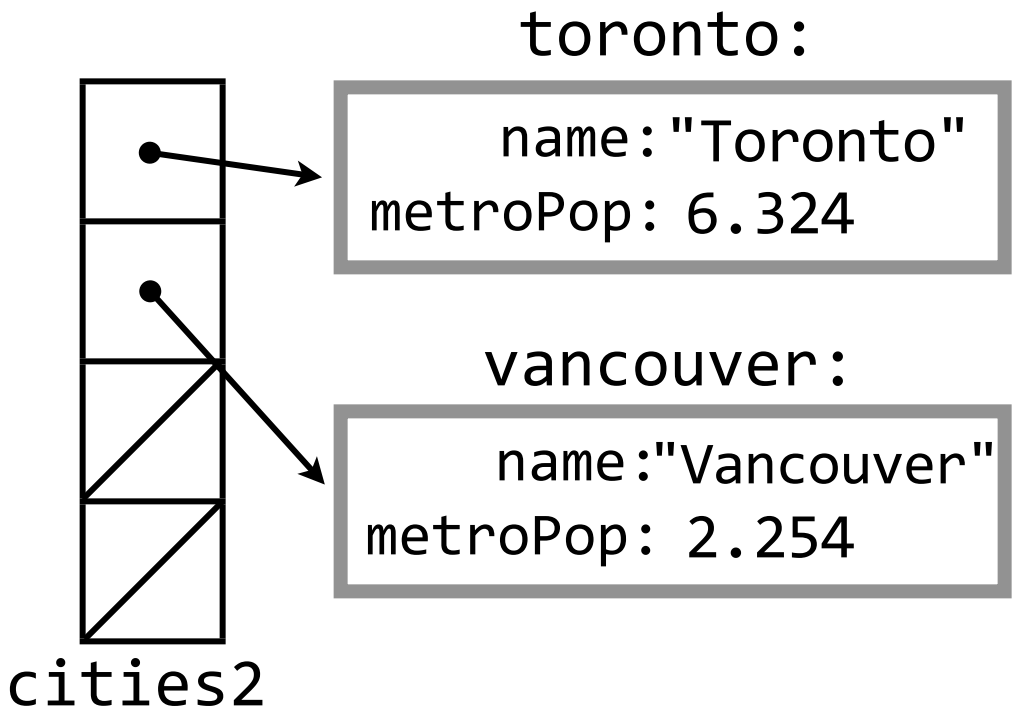
```
#define N 4

City toronto = {"Toronto", 6.324};
City vancouver = {"Vancouver", 2.254};
City *cities2[N];
for (int i = 0; i < N; i++)
{
  cities2[i] = NULL;
}

cities2[0] = &toronto;
cities2[1] = &vancouver;
```



```
for (int i = 0; i < N; i++)
{
  printf("%s\n", (*cities2[i]).name);
}
```

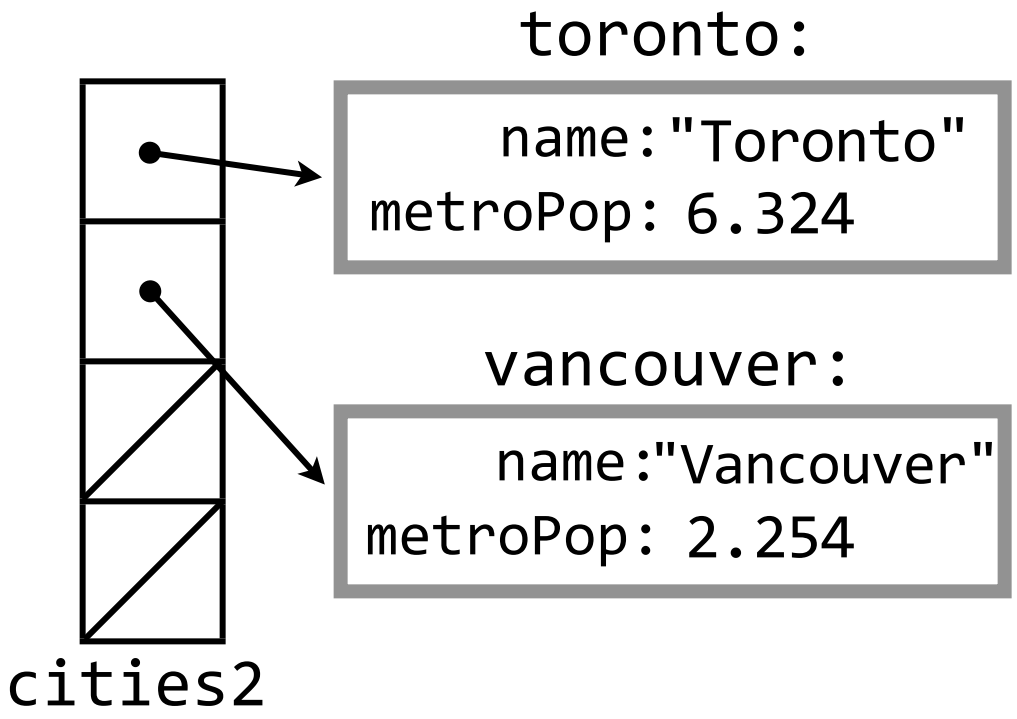


```
for (int i = 0; i < N; i++)
{
  printf("%s\n", cities2[i]->name);
}
```

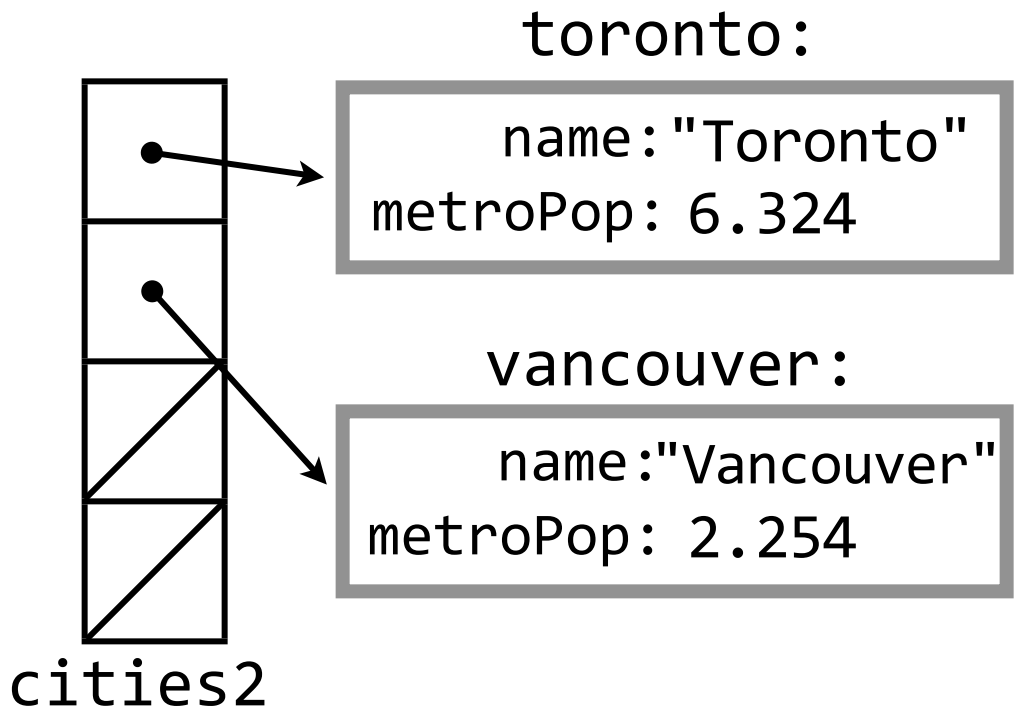


```
for (int i = 0; i < N; i++)
{
  printf("%s\n", cities2[i]->name);
}
```

cities2[i].name Error



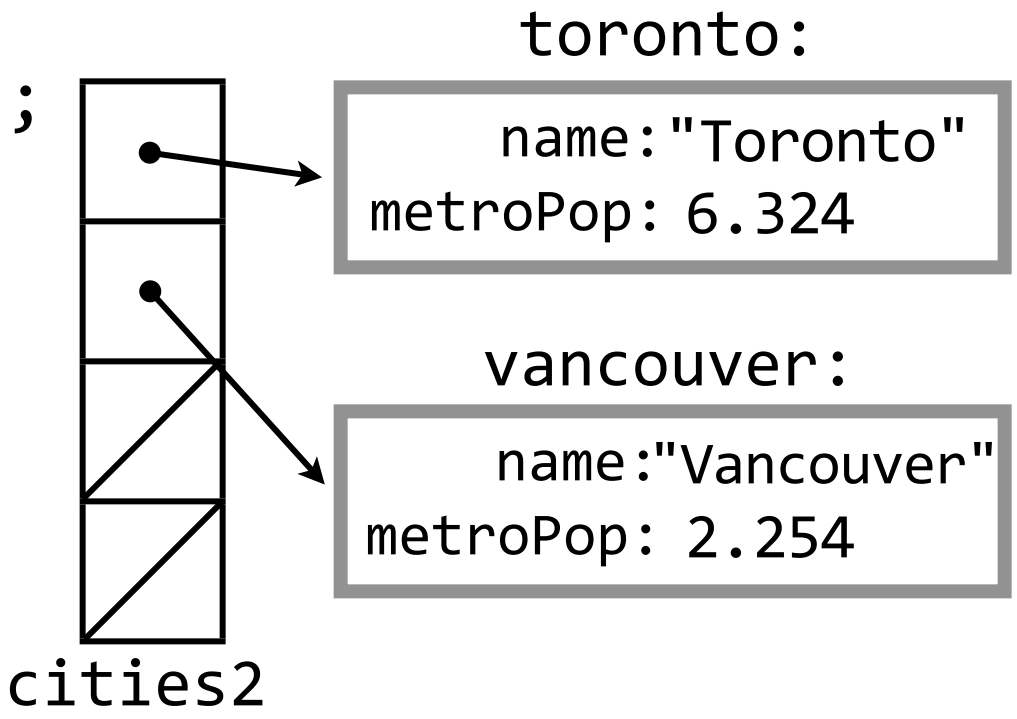
```
for (int i = 0; i < N; i++)
{
  printf("%s\n", cities2[i]->name);
}
```




```
Toronto
Vancouver
Segmentation Fault
```

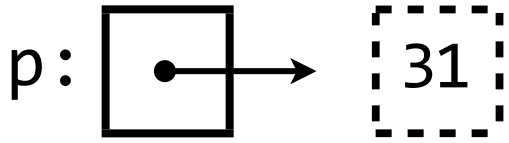
```
for (int i = 0; i < N; i++)
{
  if (cities2[i] != NULL)
  {
    printf("%s\n", cities2[i]->name);
  }
  else
  {
    printf("[empty]\n");
  }
}
```

```
Toronto
Vancouver
[empty]
[empty]
```

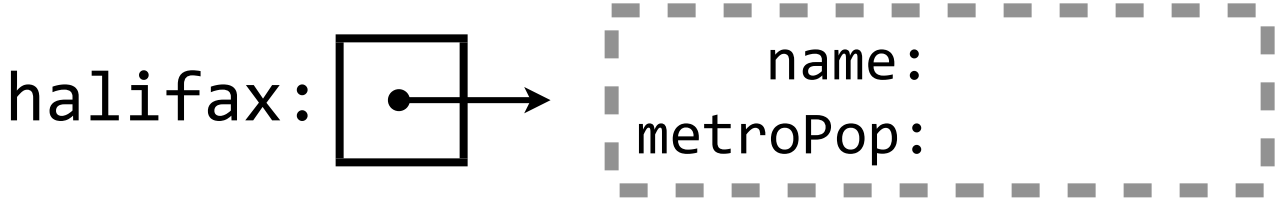


malloc()ing structs

```
int *p = malloc(sizeof(int));
*p = 31;
```

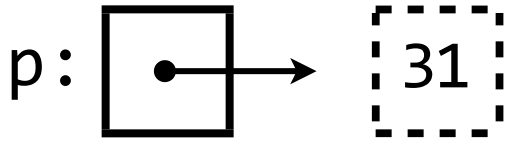



```
City *halifax = malloc(sizeof(City));
```

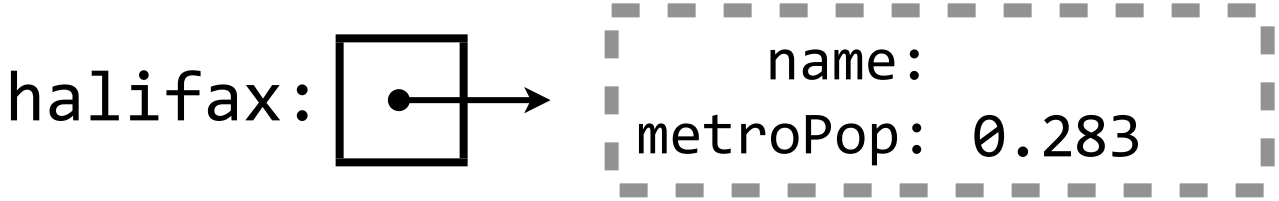


malloc()ing structs

```
int *p = malloc(sizeof(int));
*p = 31;
```

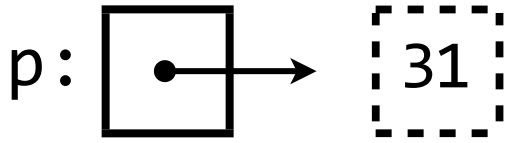



```
City *halifax = malloc(sizeof(City));
halifax->metroPopulation = 0.283;
```

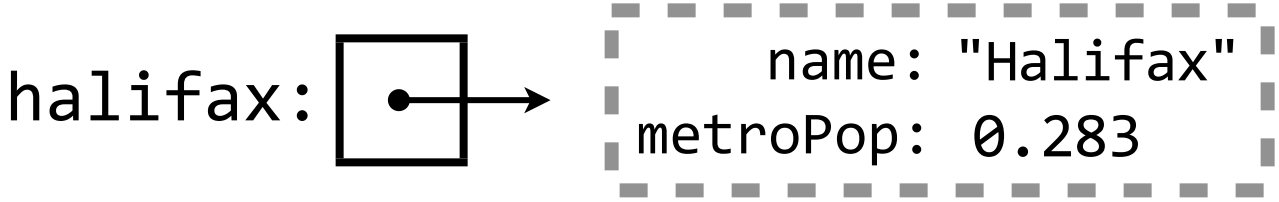


malloc()ing structs

```
int *p = malloc(sizeof(int));
*p = 31;
```

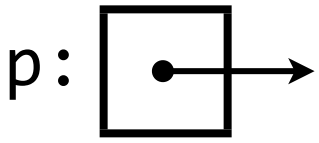



```
City *halifax = malloc(sizeof(City));
halifax->metroPopulation = 0.283;
strncpy(halifax->name, "Halifax", sizeof(halifax->name));
```

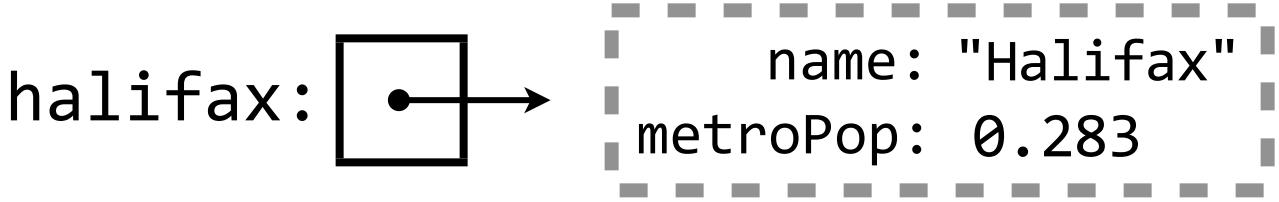


free()ing structs

```
int *p = malloc(sizeof(int));
*p = 31;
free(p);
```

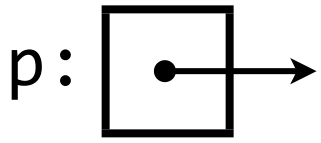



```
City *halifax = malloc(sizeof(City));
halifax->metroPopulation = 0.283;
strncpy(halifax->name, "Halifax", sizeof(halifax->name));
```

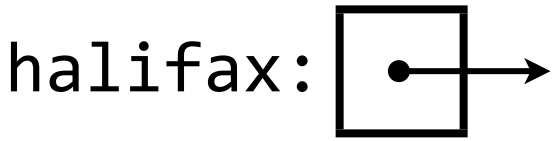


free()ing structs

```
int *p = malloc(sizeof(int));
*p = 31;
free(p);
```




```
City *halifax = malloc(sizeof(City));
halifax->metroPopulation = 0.283;
strncpy(halifax->name, "Halifax", sizeof(halifax->name));
free(halifax);
```



Pointer fields

```
typedef struct name
{
  char *first;
  char *last;
} Name;

Name name1 = {"Jonathan", "Deber"};
printf("%s %s \n", name1.first, name1.last);
```

```
Jonathan Deber
```

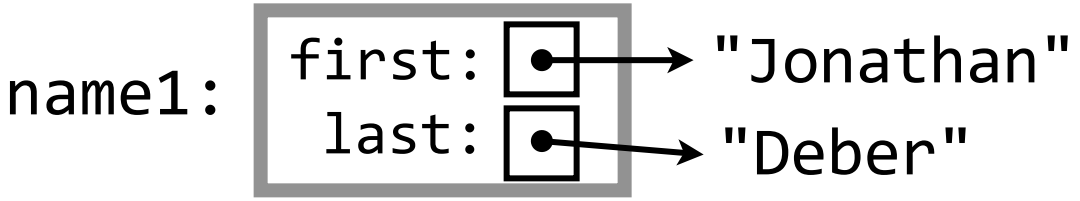
```
Name name2;
strncpy(name2.first, "Jonathan", sizeof(name2.first));
strncpy(name2.last, "Deber", sizeof(name2.last));
printf("%s %s \n", name2.first, name2.last);
```

```
Segmentation Fault
```

Wrong

Pointer fields

```
typedef struct name
{
  char *first;
  char *last;
} Name;
```




```
Name name1 = {"Jonathan", "Deber"};
printf("%s %s \n", name1.first, name1.last);
```

```
Jonathan Deber
```

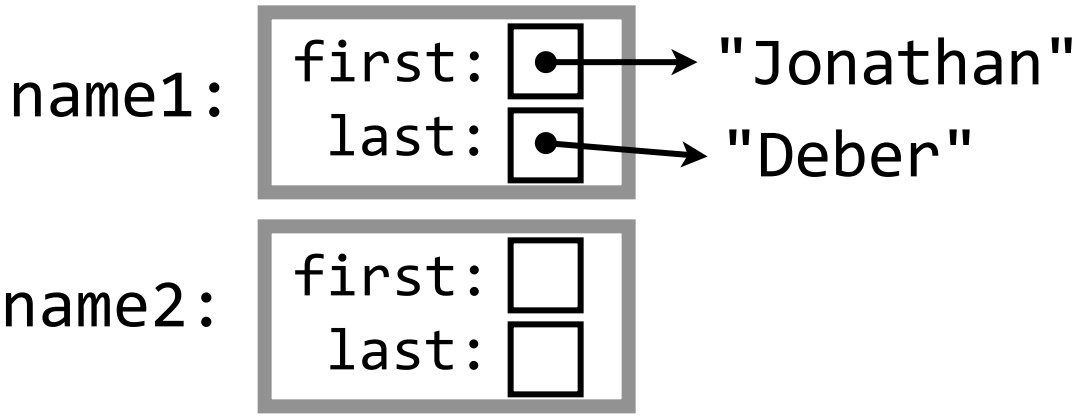
```
Name name2;
strncpy(name2.first, "Jonathan", sizeof(name2.first));
strncpy(name2.last, "Deber", sizeof(name2.last));
printf("%s %s \n", name2.first, name2.last);
```

```
Segmentation Fault
```

Wrong

Pointer fields

```
typedef struct name
{
  char *first;
  char *last;
} Name;
```




```
Name name1 = {"Jonathan", "Deber"};
printf("%s %s \n", name1.first, name1.last);
```

```
Jonathan Deber
```

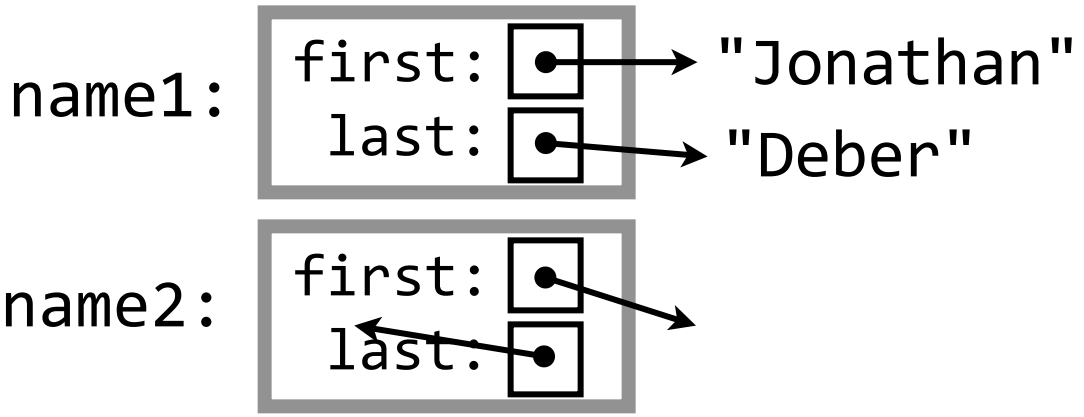
```
Name name2;
strncpy(name2.first, "Jonathan", sizeof(name2.first));
strncpy(name2.last, "Deber", sizeof(name2.last));
printf("%s %s \n", name2.first, name2.last);
```

```
Segmentation Fault
```

Wrong

Pointer fields

```
typedef struct name
{
  char *first;
  char *last;
} Name;
```




```
Name name1 = {"Jonathan", "Deber"};
printf("%s %s \n", name1.first, name1.last);
```

Jonathan Deber

```
Name name2;
strncpy(name2.first, "Jonathan", sizeof(name2.first));
strncpy(name2.last, "Deber", sizeof(name2.last));
printf("%s %s \n", name2.first, name2.last);
```

Segmentation Fault

Wrong

```
Name name2;
strncpy(name2.first, "Jonathan", sizeof(name2.first));
strncpy(name2.last, "Deber", sizeof(name2.last));
```

Wrong

```
char *first;
strncpy(first, "Jonathan", sizeof(first));
```

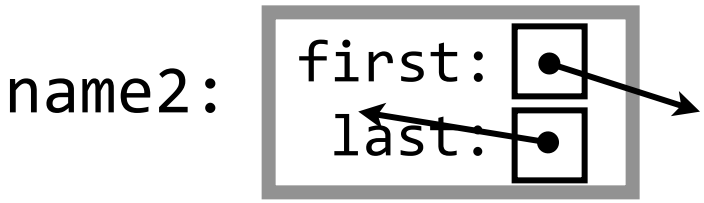
Wrong

```
int *p;
*p = 41;
```

Wrong

malloc()ing fields

```
Name name2;



strncpy(name2.first, "Jonathan", NAME_LEN + 1);
strncpy(name2.last, "Deber", NAME_LEN + 1);

printf("%s %s \n", name2.first, name2.last);
```



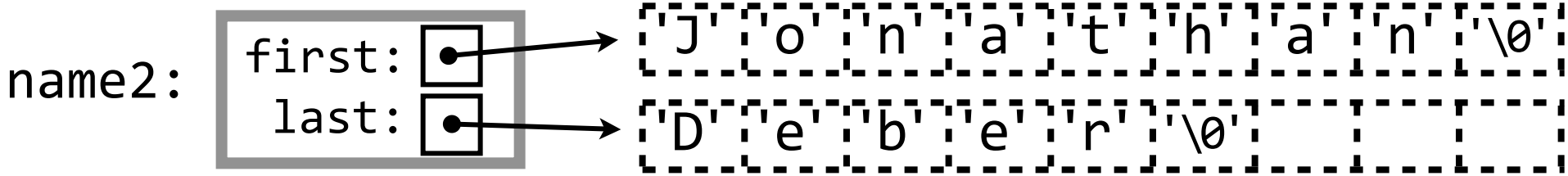
malloc()ing fields

```
Name name2;

name2.first = malloc( (NAME_LEN + 1) * sizeof(char) );
name2.last  = malloc( (NAME_LEN + 1) * sizeof(char) );


strncpy(name2.first, "Jonathan", NAME_LEN + 1);
strncpy(name2.last, "Deber", NAME_LEN + 1);


printf("%s %s \n", name2.first, name2.last);
```

malloc()ing fields

```
Name name2;

name2.first = malloc( (NAME_LEN + 1) * sizeof(char) );
name2.last  = malloc( (NAME_LEN + 1) * sizeof(char) );


strncpy(name2.first, "Jonathan", NAME_LEN + 1);
strncpy(name2.last, "Deber", NAME_LEN + 1);


printf("%s %s \n", name2.first, name2.last);
```

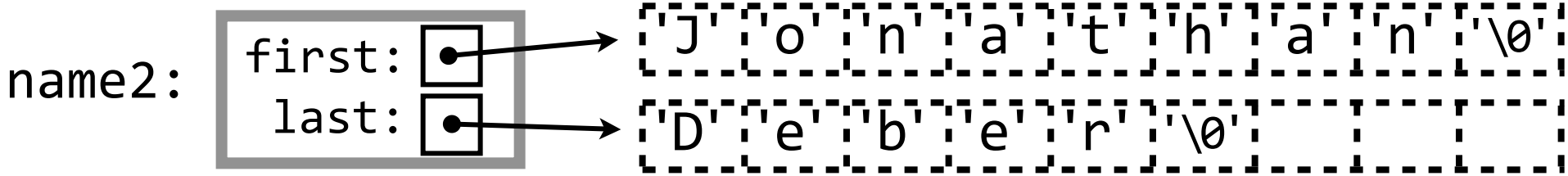
```
Jonathan Deber
```



free()ing fields

```
Name name2;

name2.first = malloc( (NAME_LEN + 1) * sizeof(char) );
name2.last  = malloc( (NAME_LEN + 1) * sizeof(char) );


strncpy(name2.first, "Jonathan", NAME_LEN + 1);
strncpy(name2.last, "Deber", NAME_LEN + 1);


printf("%s %s \n", name2.first, name2.last);

free(name2.first);
free(name2.last);
```

```
Jonathan Deber
```



free()ing fields

```
Name name2;

name2.first = malloc( (NAME_LEN + 1) * sizeof(char) );
name2.last  = malloc( (NAME_LEN + 1) * sizeof(char) );


strncpy(name2.first, "Jonathan", NAME_LEN + 1);
strncpy(name2.last, "Deber", NAME_LEN + 1);


printf("%s %s \n", name2.first, name2.last);

free(name2.first);
free(name2.last);
```

```
Jonathan Deber
```

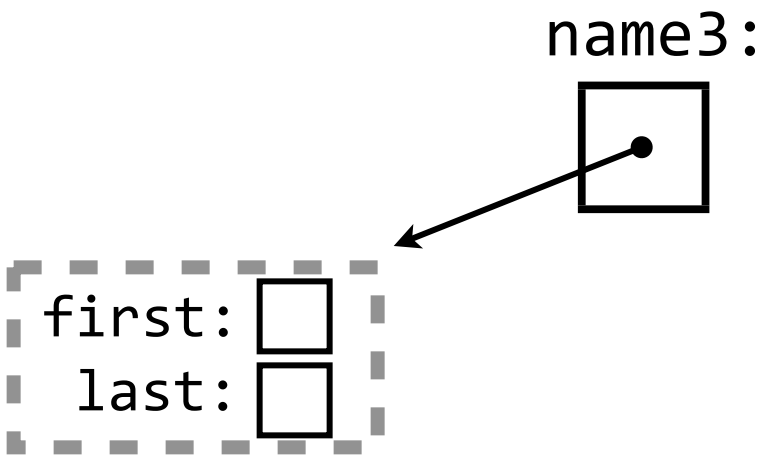
free()ing fields

```
Name name2;

name2.first = malloc( (NAME_LEN + 1) * sizeof(char) );
name2.last  = malloc( (NAME_LEN + 1) * sizeof(char) );


strncpy(name2.first, "Jonathan", NAME_LEN + 1);
strncpy(name2.last, "Deber", NAME_LEN + 1);


printf("%s %s \n", name2.first, name2.last);

free(name2.first);
free(name2.last);
```

```
Jonathan Deber
```

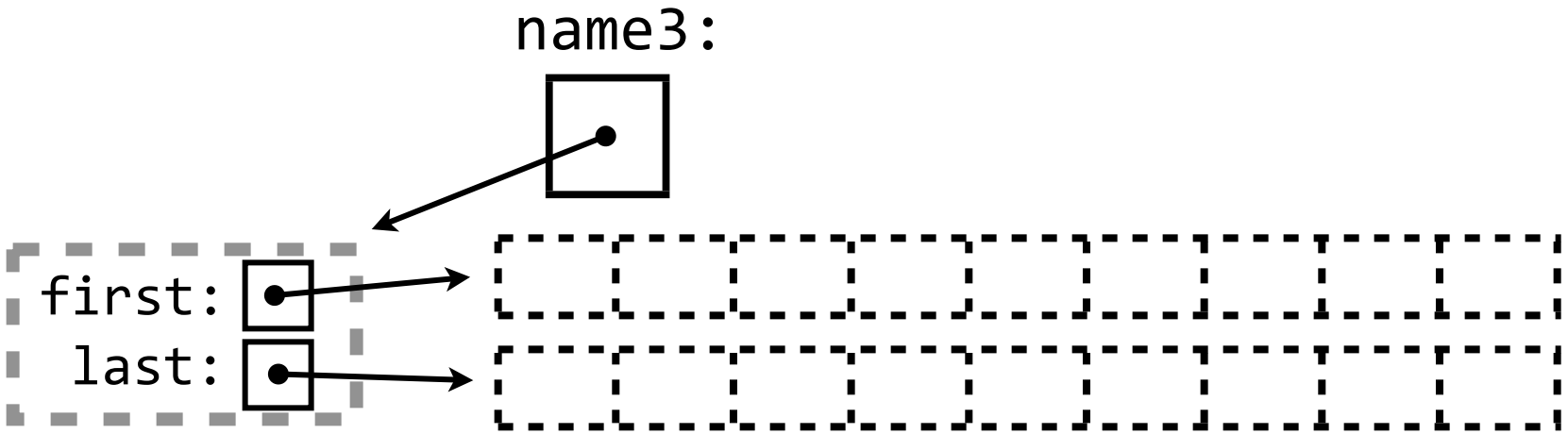
malloc() and malloc()

```
Name *name3 = malloc(sizeof(Name));
```



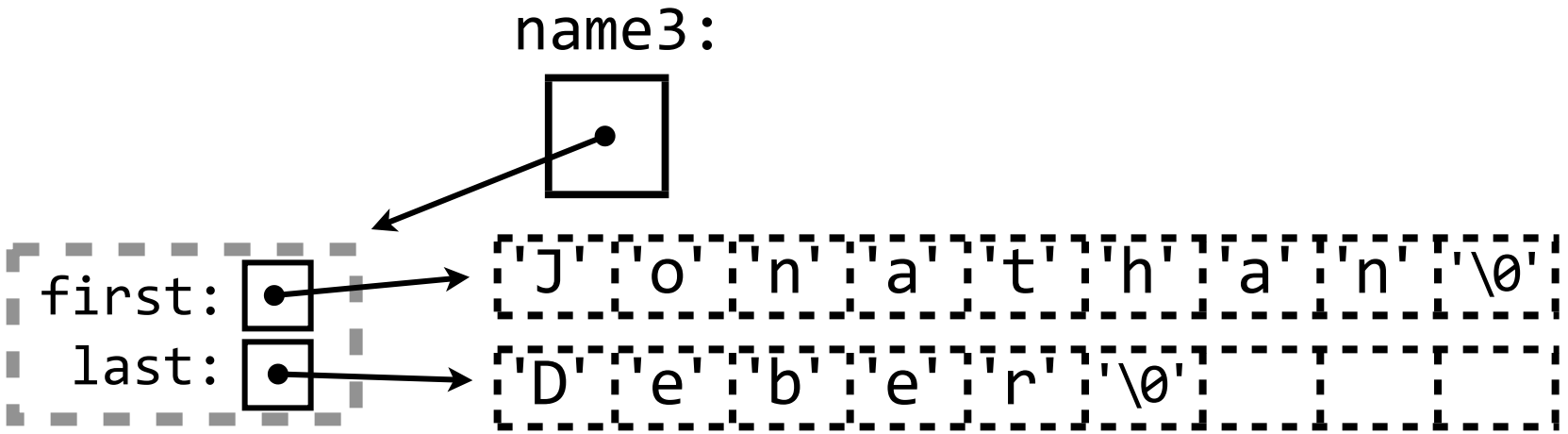
malloc() and malloc()

```
Name *name3 = malloc(sizeof(Name));

name3->first = malloc((NAME_LEN + 1) * sizeof(char));
name3->last  = malloc((NAME_LEN + 1) * sizeof(char));
```



malloc() and malloc()

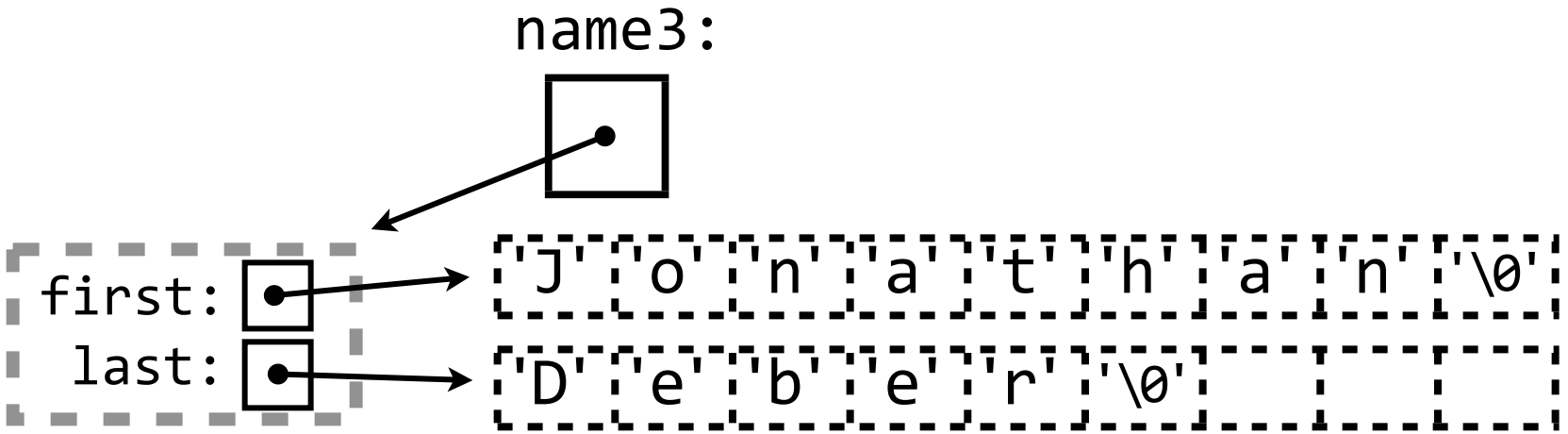
```
Name *name3 = malloc(sizeof(Name));

name3->first = malloc((NAME_LEN + 1) * sizeof(char));
name3->last  = malloc((NAME_LEN + 1) * sizeof(char));

strncpy(name3->first, "Jonathan", NAME_LEN + 1);
strncpy(name3->last, "Deber", NAME_LEN + 1);
```



malloc() and malloc()

```
Name *name3 = malloc(sizeof(Name));

name3->first = malloc((NAME_LEN + 1) * sizeof(char));
name3->last  = malloc((NAME_LEN + 1) * sizeof(char));

strncpy(name3->first, "Jonathan", NAME_LEN + 1);
strncpy(name3->last, "Deber", NAME_LEN + 1);

printf("%s %s \n", name3->first, name3->last);
```

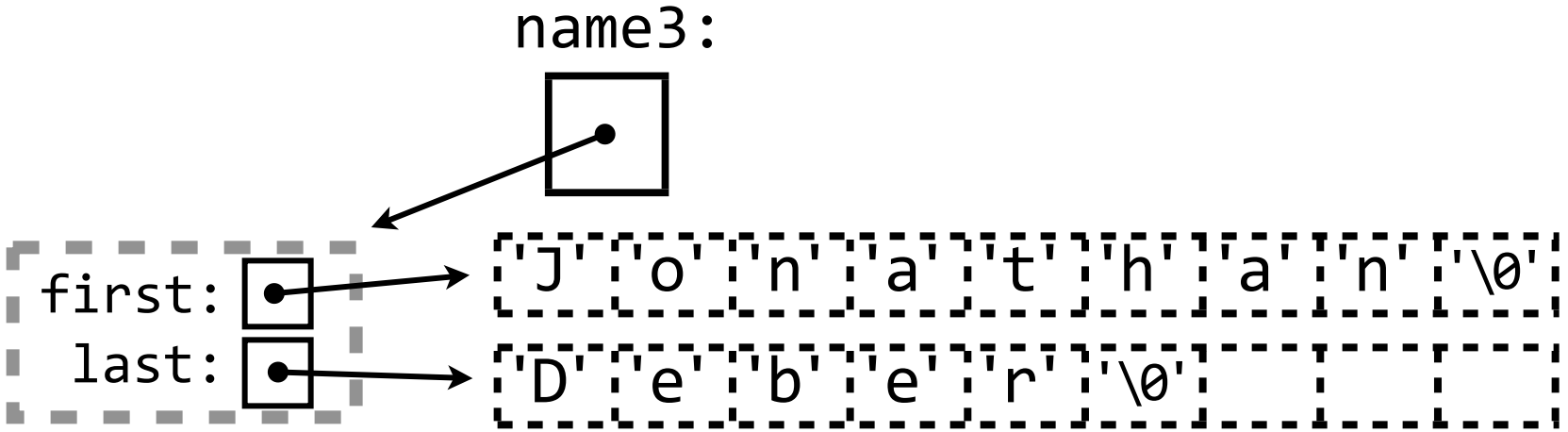
```
Jonathan Deber
```



malloc() and malloc()

```
Name *name3 = malloc(sizeof(Name));

name3->first = malloc((NAME_LEN + 1) * sizeof(char));
name3->last  = malloc((NAME_LEN + 1) * sizeof(char));

strncpy(name3->first, "Jonathan", NAME_LEN + 1);
strncpy(name3->last, "Deber", NAME_LEN + 1);

printf("%s %s \n", name3->first, name3->last);

free(name3->first);
free(name3->last);
```

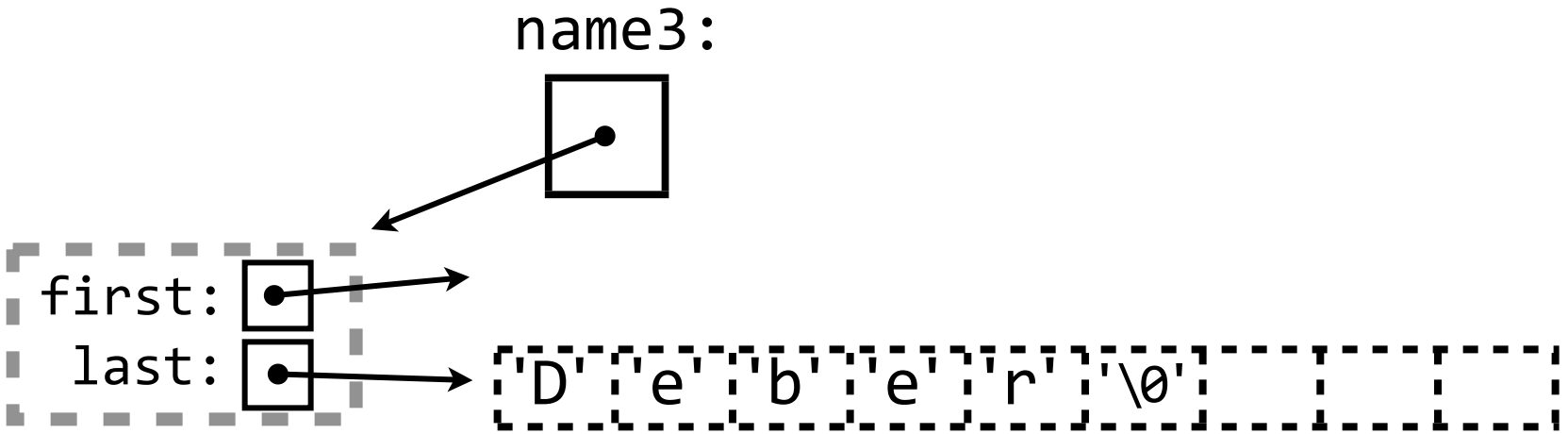
```
Jonathan Deber
```



malloc() and malloc()

```
Name *name3 = malloc(sizeof(Name));

name3->first = malloc((NAME_LEN + 1) * sizeof(char));
name3->last  = malloc((NAME_LEN + 1) * sizeof(char));

strncpy(name3->first, "Jonathan", NAME_LEN + 1);
strncpy(name3->last, "Deber", NAME_LEN + 1);

printf("%s %s \n", name3->first, name3->last);

free(name3->first);
free(name3->last);
```

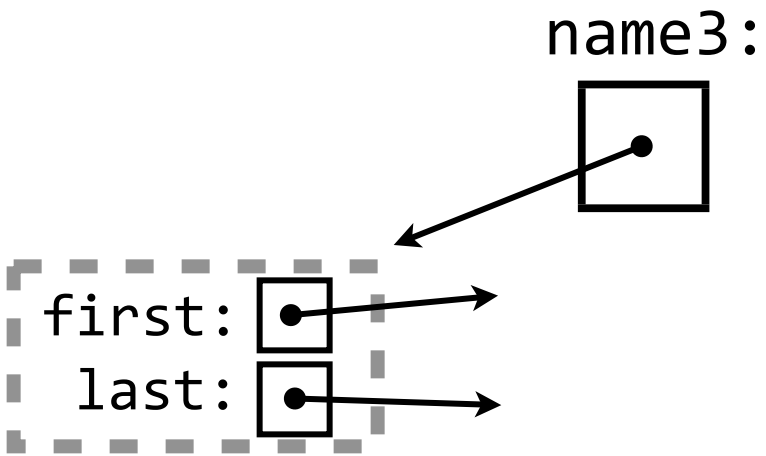
```
Jonathan Deber
```



malloc() and malloc()

```
Name *name3 = malloc(sizeof(Name));

name3->first = malloc((NAME_LEN + 1) * sizeof(char));
name3->last  = malloc((NAME_LEN + 1) * sizeof(char));

strncpy(name3->first, "Jonathan", NAME_LEN + 1);
strncpy(name3->last, "Deber", NAME_LEN + 1);

printf("%s %s \n", name3->first, name3->last);

free(name3->first);
free(name3->last);
```

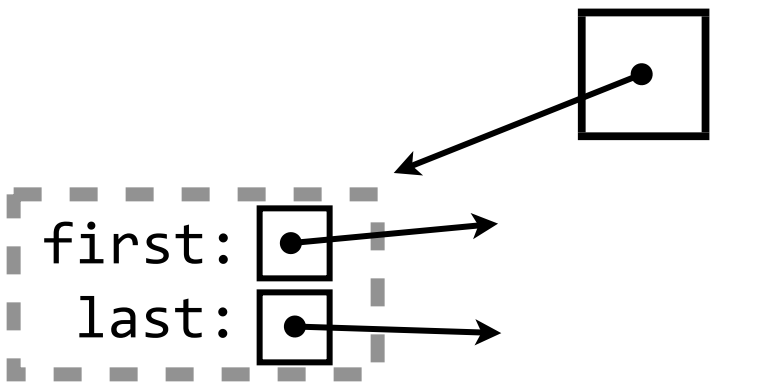
```
Jonathan Deber
```



malloc() and malloc()

```
Name *name3 = malloc(sizeof(Name));

name3->first = malloc((NAME_LEN + 1) * sizeof(char));
name3->last  = malloc((NAME_LEN + 1) * sizeof(char));

strncpy(name3->first, "Jonathan", NAME_LEN + 1);
strncpy(name3->last, "Deber", NAME_LEN + 1);

printf("%s %s \n", name3->first, name3->last);

free(name3->first);
free(name3->last);
free(name3);
```

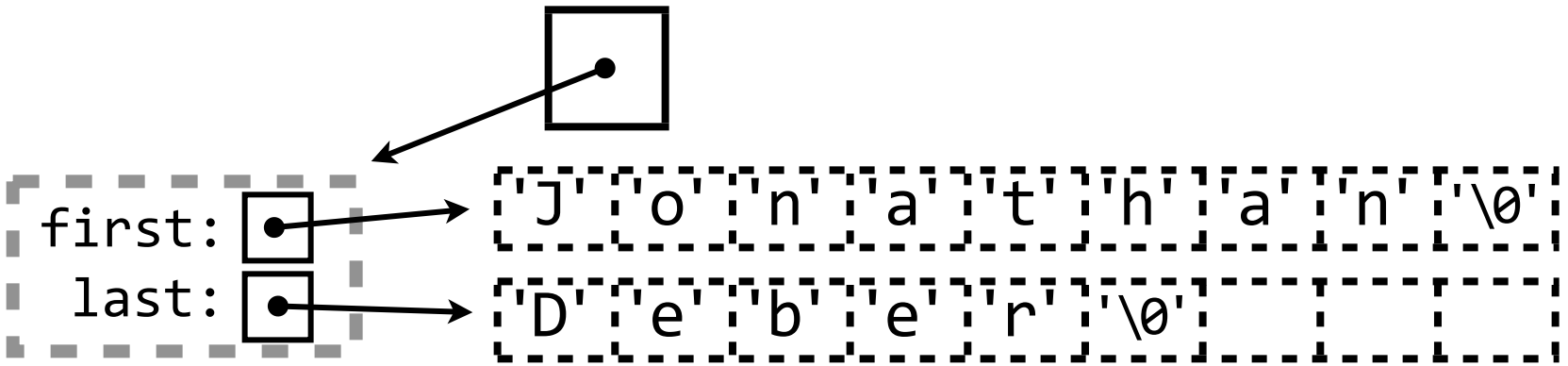
```
Jonathan Deber
```



malloc() and malloc()

```
Name *name3 = malloc(sizeof(Name));

name3->first = malloc((NAME_LEN + 1) * sizeof(char));
name3->last  = malloc((NAME_LEN + 1) * sizeof(char));

strncpy(name3->first, "Jonathan", NAME_LEN + 1);
strncpy(name3->last, "Deber", NAME_LEN + 1);

printf("%s %s \n", name3->first, name3->last);

free(name3);
free(name3->first);
free(name3->last);
```

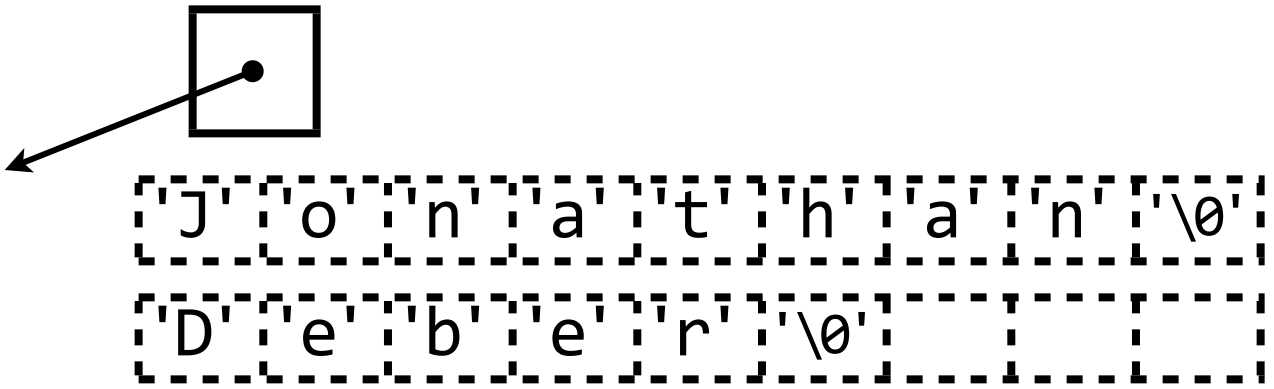
```
Jonathan Deber
```



malloc() and malloc()

```
Name *name3 = malloc(sizeof(Name));

name3->first = malloc((NAME_LEN + 1) * sizeof(char));
name3->last  = malloc((NAME_LEN + 1) * sizeof(char));

strncpy(name3->first, "Jonathan", NAME_LEN + 1);
strncpy(name3->last, "Deber", NAME_LEN + 1);

printf("%s %s \n", name3->first, name3->last);

free(name3);
free(name3->first);
free(name3->last);
```

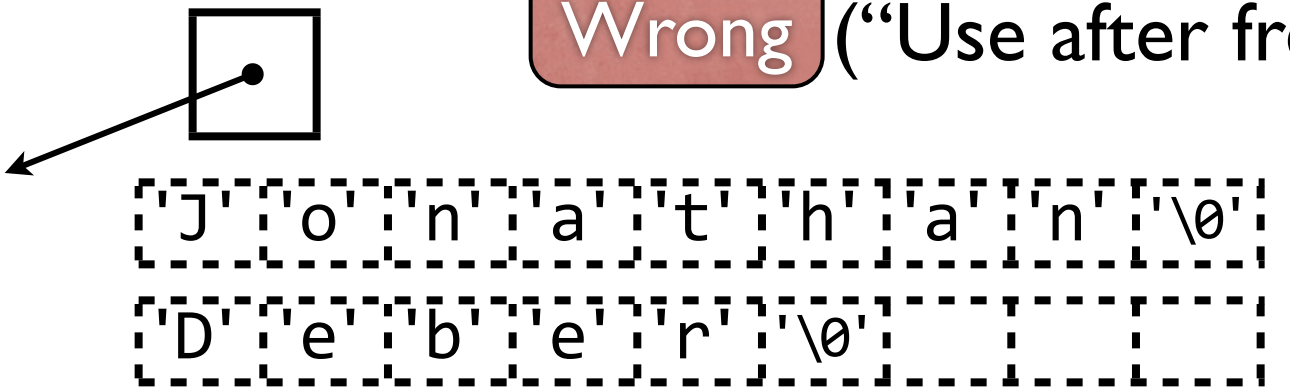
```
Jonathan Deber
```



malloc() and malloc()

```
Name *name3 = malloc(sizeof(Name));

name3->first = malloc((NAME_LEN + 1) * sizeof(char));
name3->last  = malloc((NAME_LEN + 1) * sizeof(char));

strncpy(name3->first, "Jonathan", NAME_LEN + 1);
strncpy(name3->last, "Deber", NAME_LEN + 1);

printf("%s %s \n", name3->first, name3->last);

free(name3);
free(name3->first);
free(name3->last);
```

```
Jonathan Deber
```



Wrong ("Use after free")

Create and Destroy

- Functions that help us create/destroy `structs`

```
Name *createName(const char *first, const char *last);
void destroyName(Name *name);

Name *name3 = malloc(sizeof(Name));
name3->first = malloc((NAME_LEN + 1) * sizeof(char));
name3->last  = malloc((NAME_LEN + 1) * sizeof(char));
...
free(name3->first);
free(name3->last);
free(name3);

Name *name4 = createName("Jonathan", "Deber");
...
destroyName(name4);
```

structs and Functions

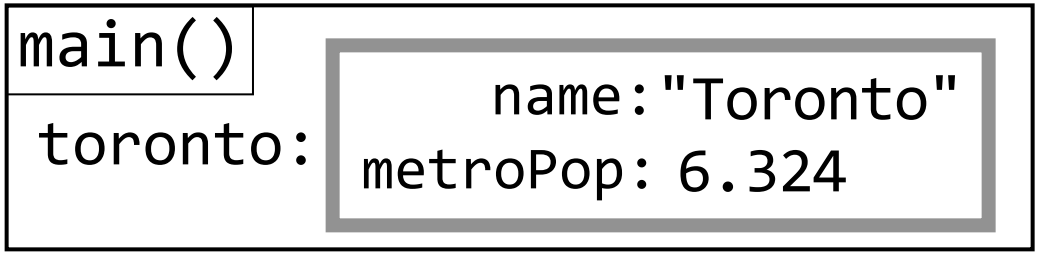
- You can pass structs in to functions
- You can return structs from functions

```
void printMaleGreeting(Name name)
{
    printf("Dear Mr. %s:\n", name.last);
}

Name me = {"Jonathan", "Deber"};
printMaleGreeting(me);
```

```
Dear Mr. Deber:
```

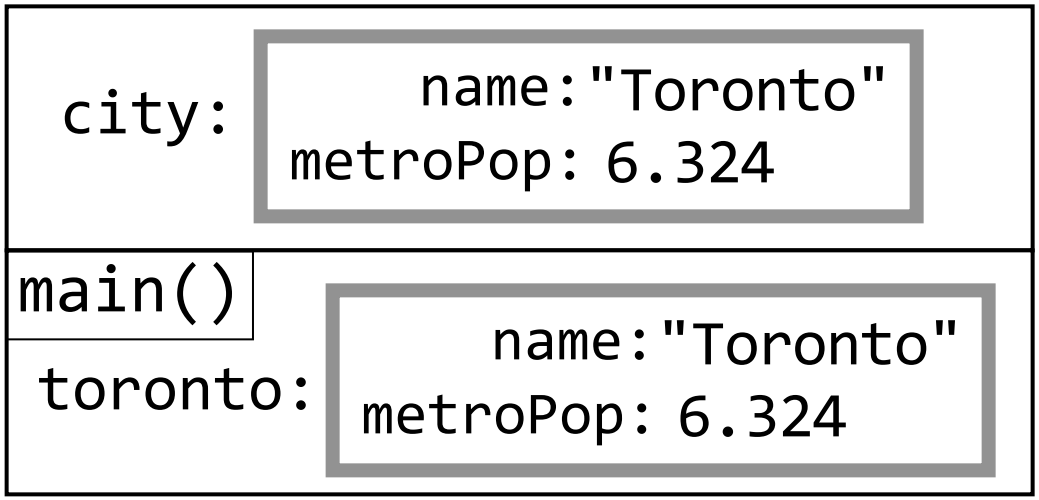
Pass by Value

```
void printCity(City city);
int main(void)
{
  City toronto = {"Toronto", 6.324};
  printCity(toronto);
  ...
```



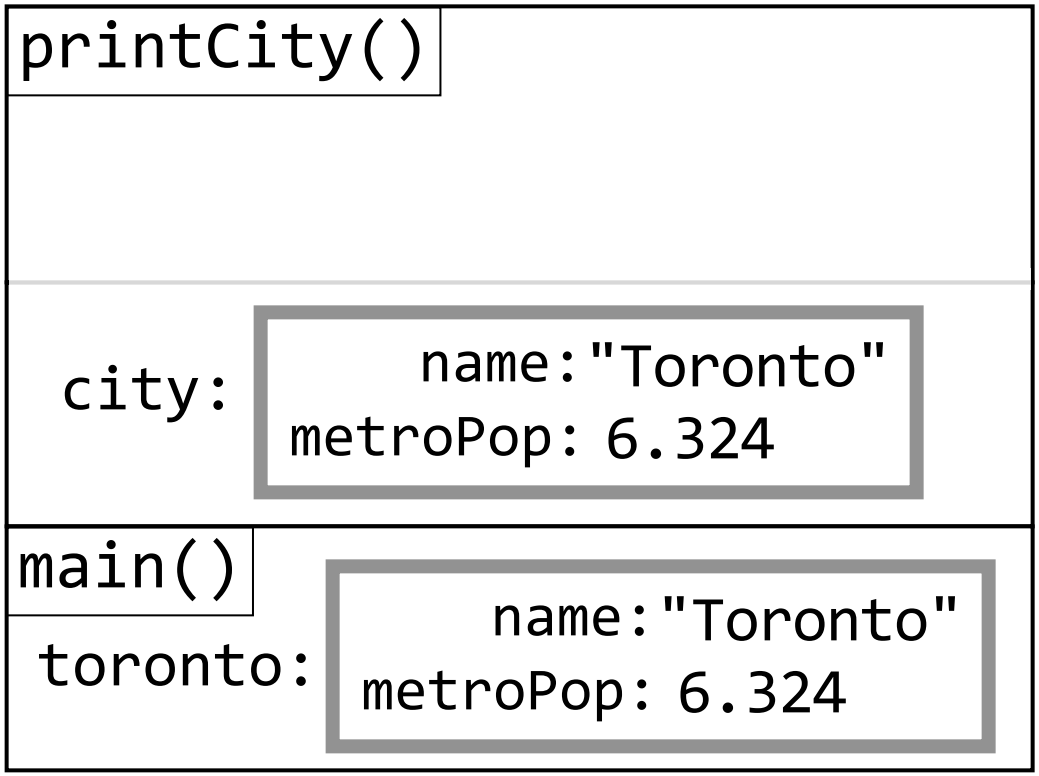
Pass by Value

```
void printCity(City city);
int main(void)
{
  City toronto = {"Toronto", 6.324};
  printCity(toronto);
  ...
```



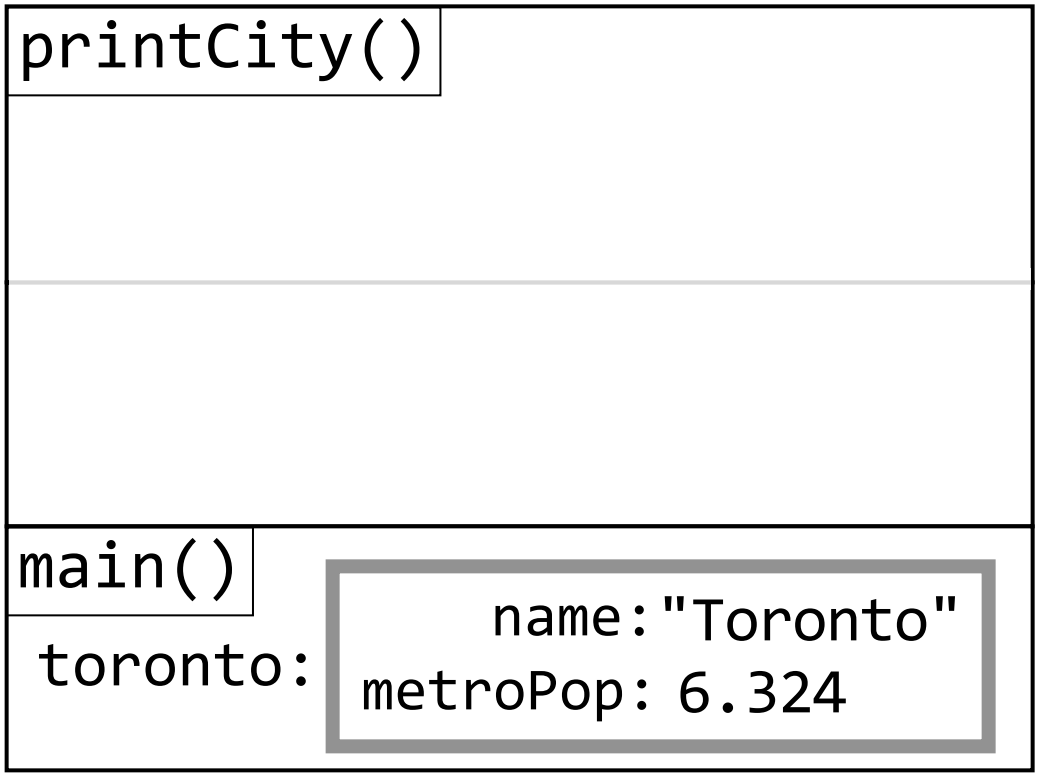
Pass by Value

```
void printCity(City city);
int main(void)
{
  City toronto = {"Toronto", 6.324};
  printCity(toronto);
  ...
```



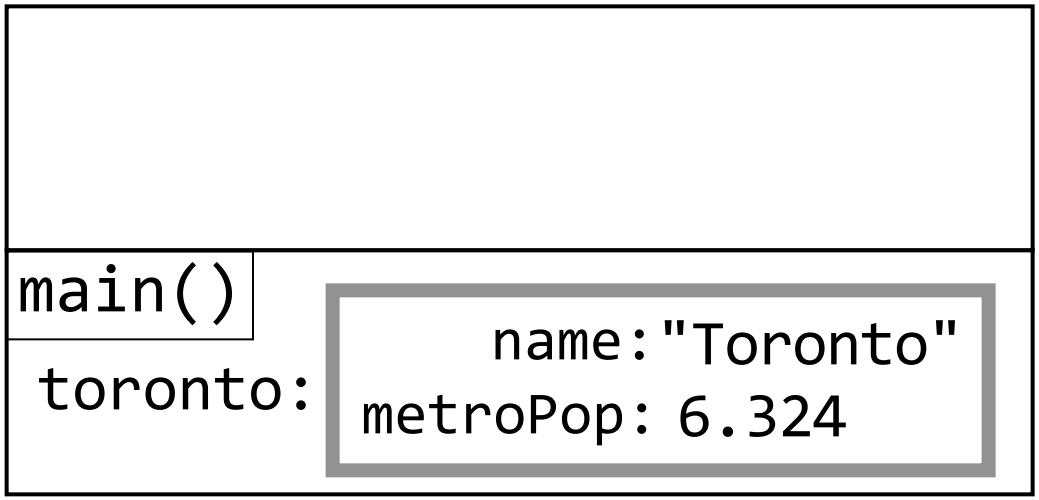
Pass by Value

```
void printCity(City city);
int main(void)
{
  City toronto = {"Toronto", 6.324};
  printCity(toronto);
  ...
```



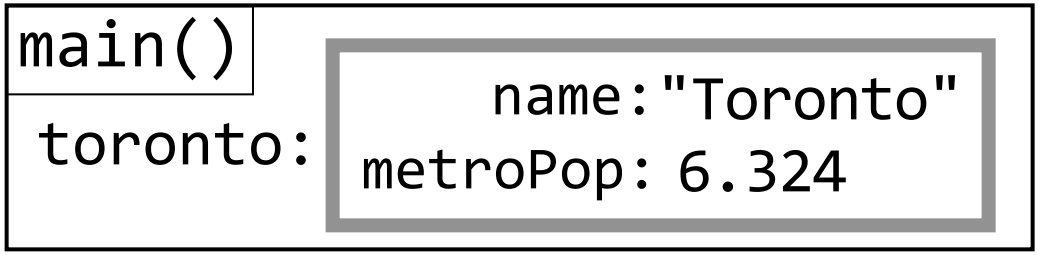
Pass by Value

```
void printCity(City city);
int main(void)
{
  City toronto = {"Toronto", 6.324};
  printCity(toronto);
  ...
```



Pass by Value

```
void printCity(City city);
int main(void)
{
  City toronto = {"Toronto", 6.324};
  printCity(toronto);
  ...
```



Pass by Value

```
typedef struct big
{
   int reallyBig[1000000000];
} Big;
```

We copy several GB every time we pass a `Big` to a function

Pointers and `structs`

- To save time/memory, we very frequently refer to structs via pointers
- Even when functions don't need to change them
  - Make sure to use `const`
- (That's why there's the `->` operator)

Data Structures

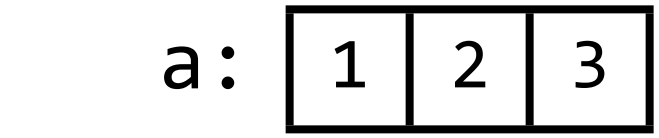
Data Structures

- How we represent and store data in our programs

- Individual variables
- Arrays
- Arrays of pointers
- `structs`

Arrays

```
int a[] = {1,2,3};
```



- Pro:
  - Simple
  - Easy (and fast) access to each element
- Con:
  - Only stores one type of data
  - Fixed size
  - Needs to be contiguous

Dynamically Allocated Arrays

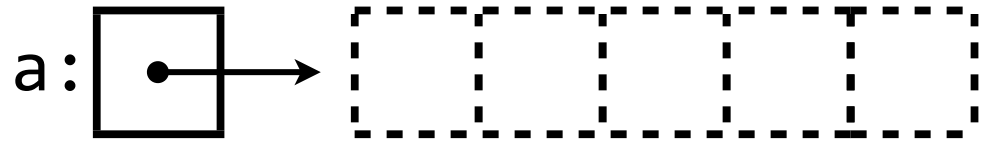
- Use `malloc()`
- Fixes the "fixed size" problem

```
int *a = malloc(N * sizeof(int));
```

Dynamically Allocated Arrays

- Use `malloc()`
- Fixes the "fixed size" problem

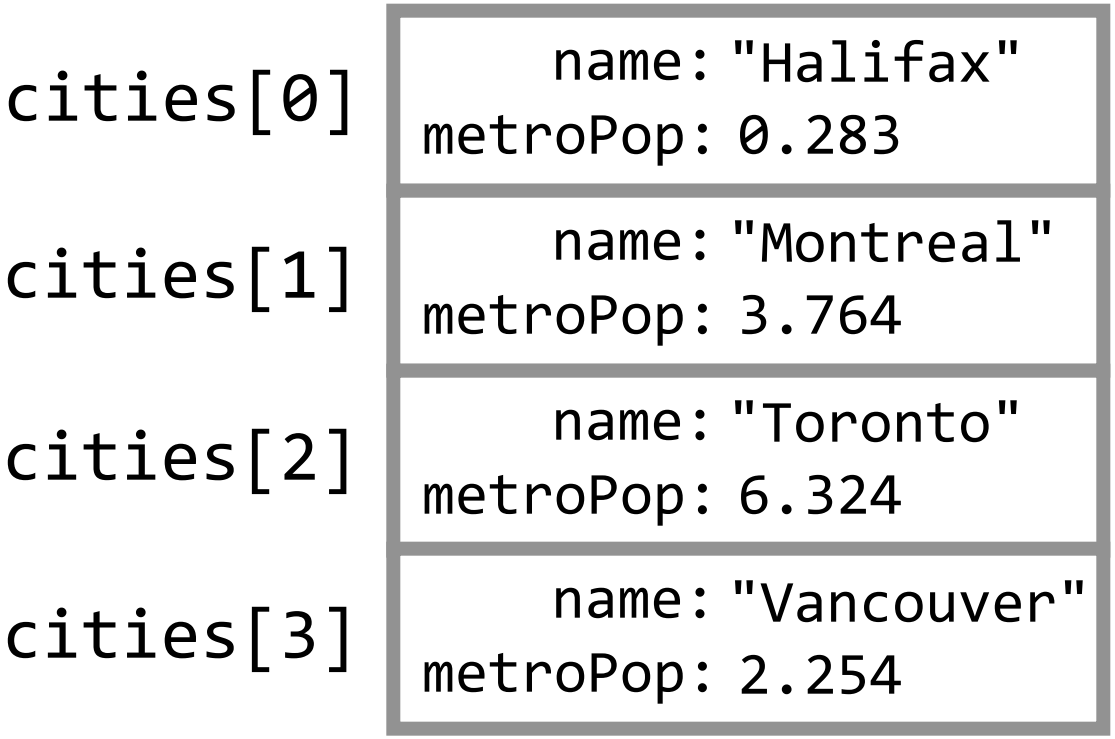
```
int *a = malloc(N * sizeof(int));

a = realloc(a, (N + 1) * sizeof(int));
```



Arrays of `structs`

- `structs` let us group multiple variables together
- Arrays let us group multiple `structs` together
- Fixes the "single type" problem

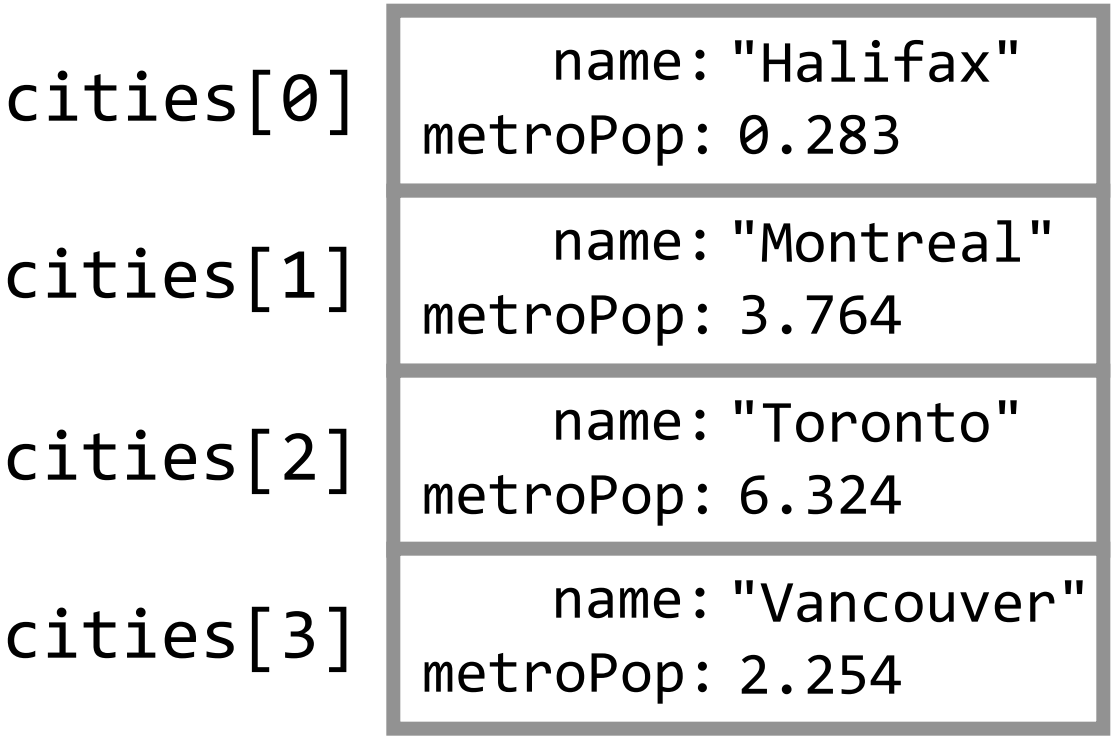
```
City cities[4];
```



Arrays of `structs`

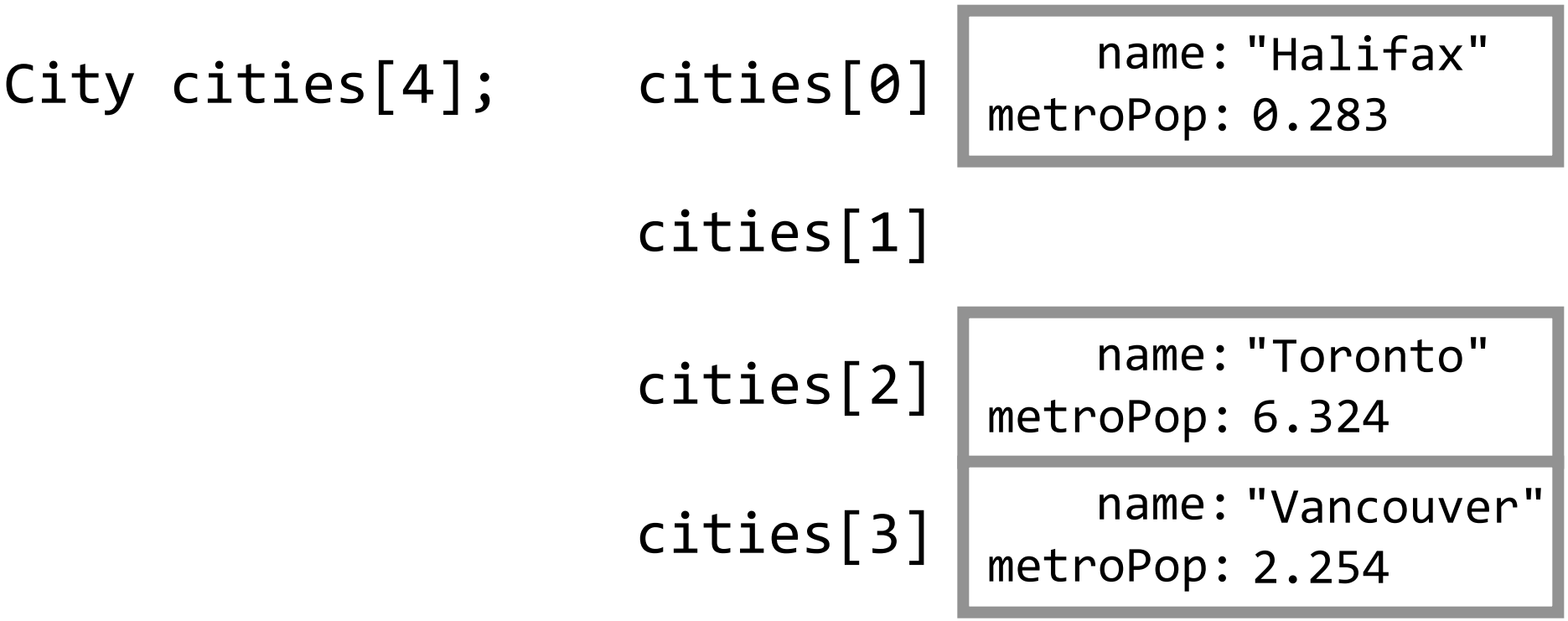
- Still need to be contiguous
- Hard to delete/move elements

```
City cities[4];
```



Arrays of `structs`

- Still need to be contiguous
- Hard to delete/move elements



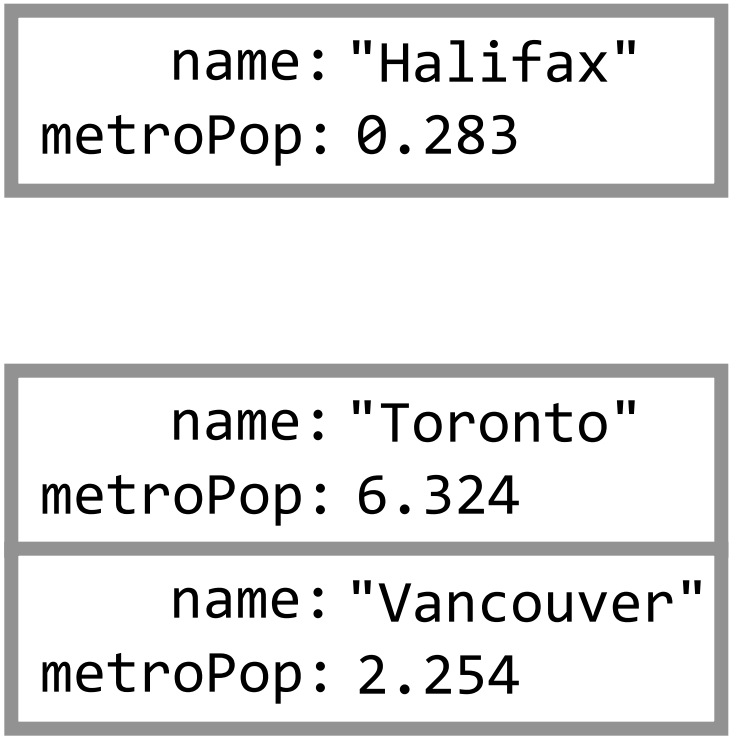
Arrays of `structs`

- Still need to be contiguous
- Hard to delete/move elements

```
City cities[4];
```

```
cities[1] = NULL;
```

Error



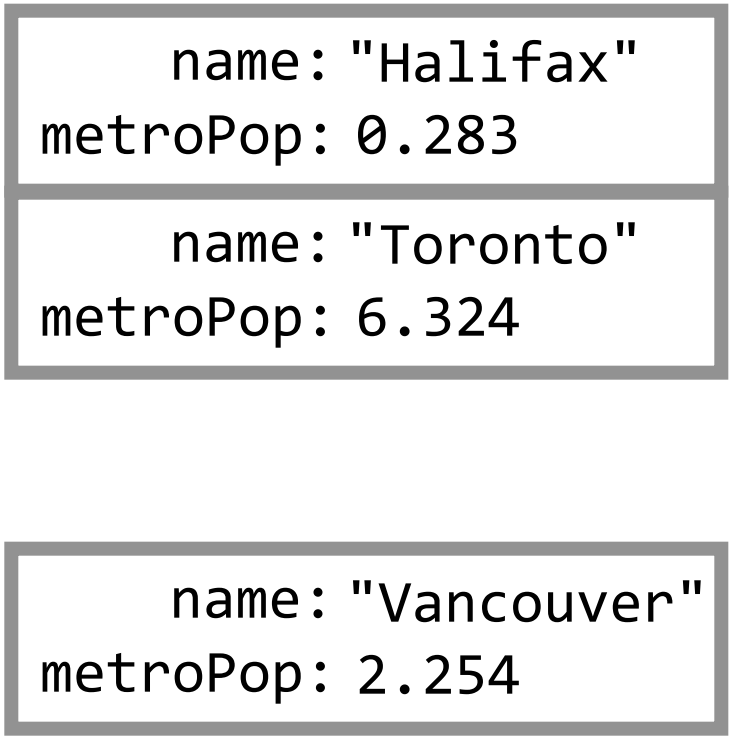
Arrays of `structs`

- Still need to be contiguous
- Hard to delete/move elements

```
City cities[4];
```

```
cities[1] = NULL;
```

Error



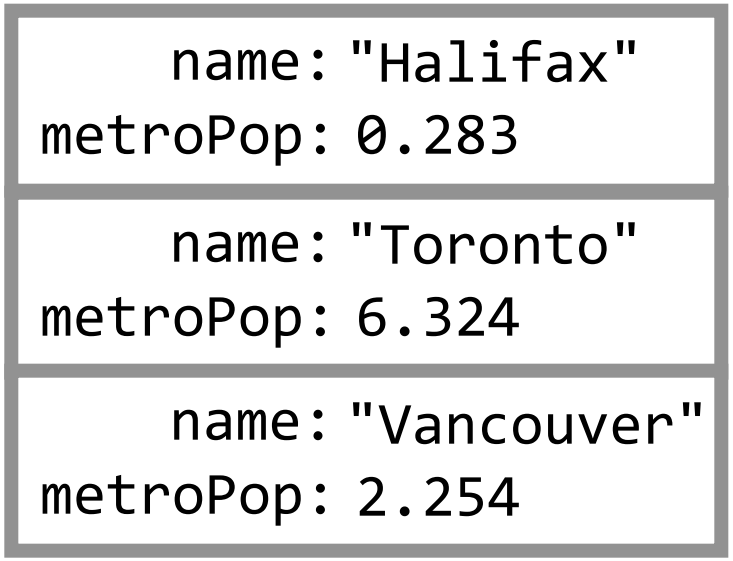
Arrays of `structs`

- Still need to be contiguous
- Hard to delete/move elements

```
City cities[4];
```

```
cities[1] = NULL;
```

Error



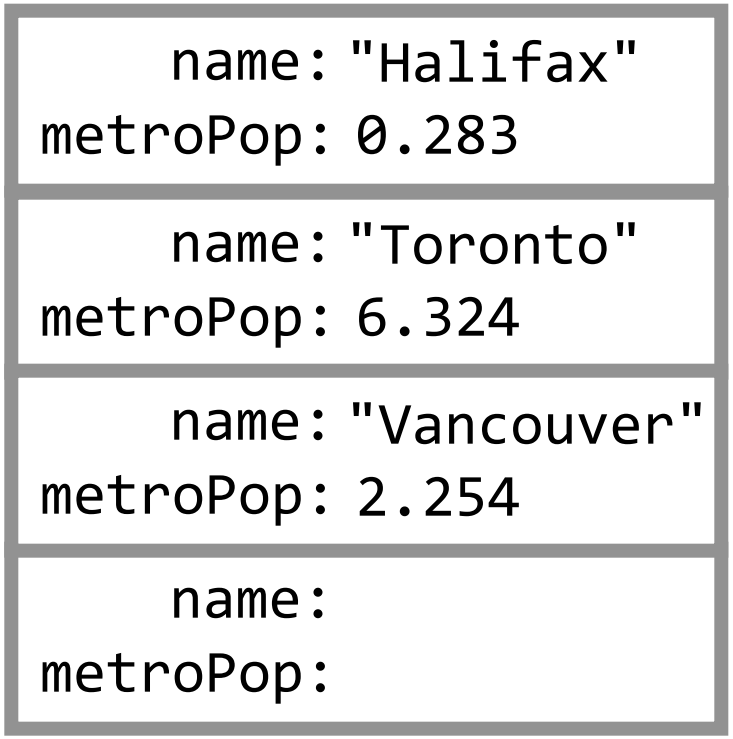
Arrays of `structs`

- Still need to be contiguous
- Hard to delete/move elements

```
City cities[4];
```

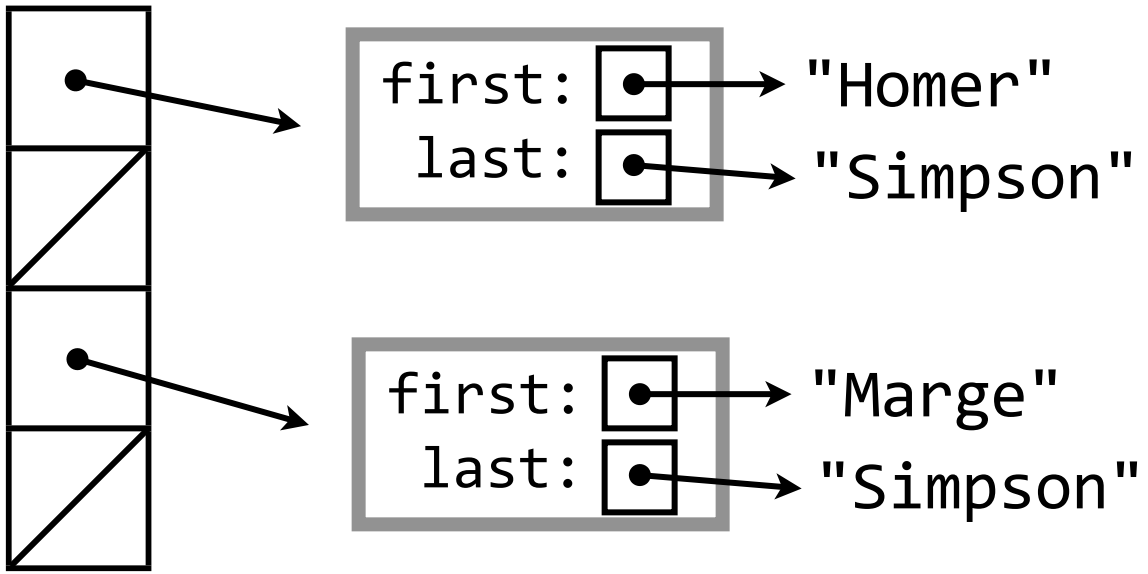
```
cities[1] = NULL;
```

Error



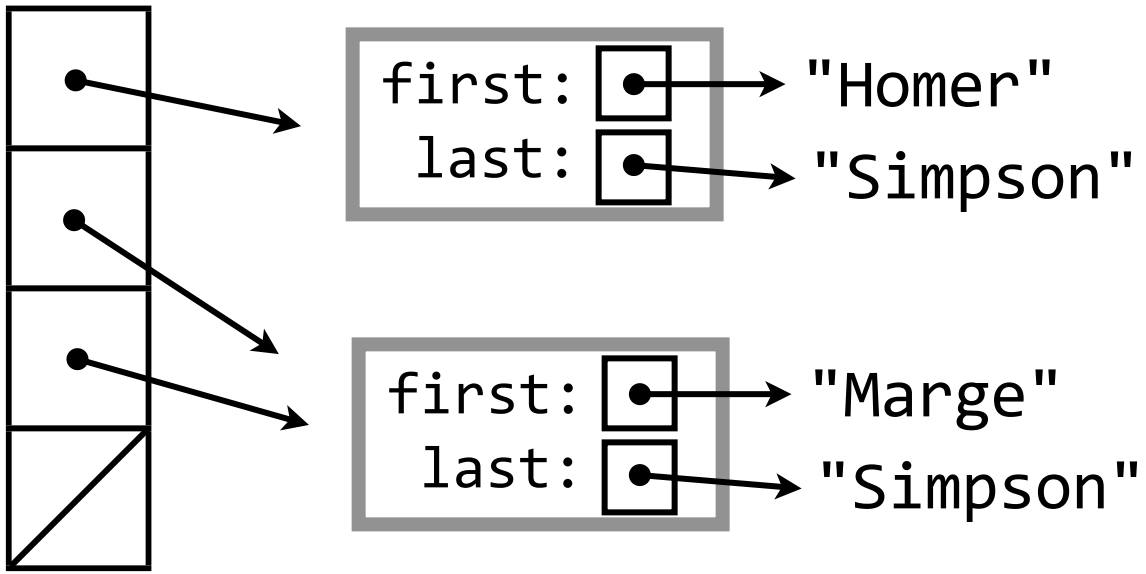
Arrays of pointers to `structs`

- Array elements are much smaller
- Array elements can be `NULL`



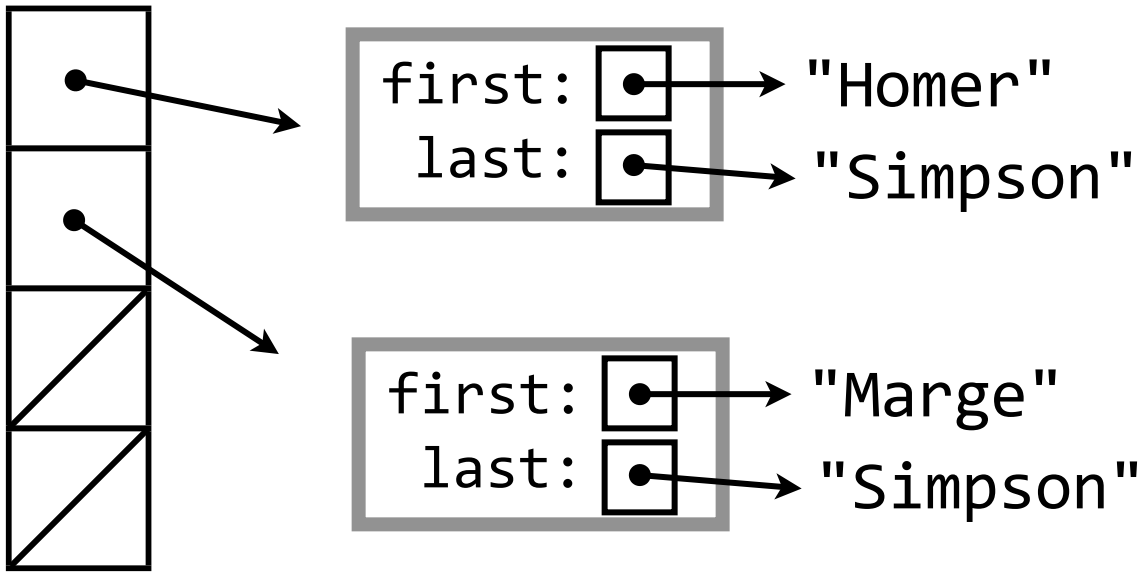
Arrays of pointers to `structs`

- Array elements are much smaller
- Array elements can be `NULL`



Arrays of pointers to `structs`

- Array elements are much smaller
- Array elements can be `NULL`



Shifting Can be Slow