Pointer II

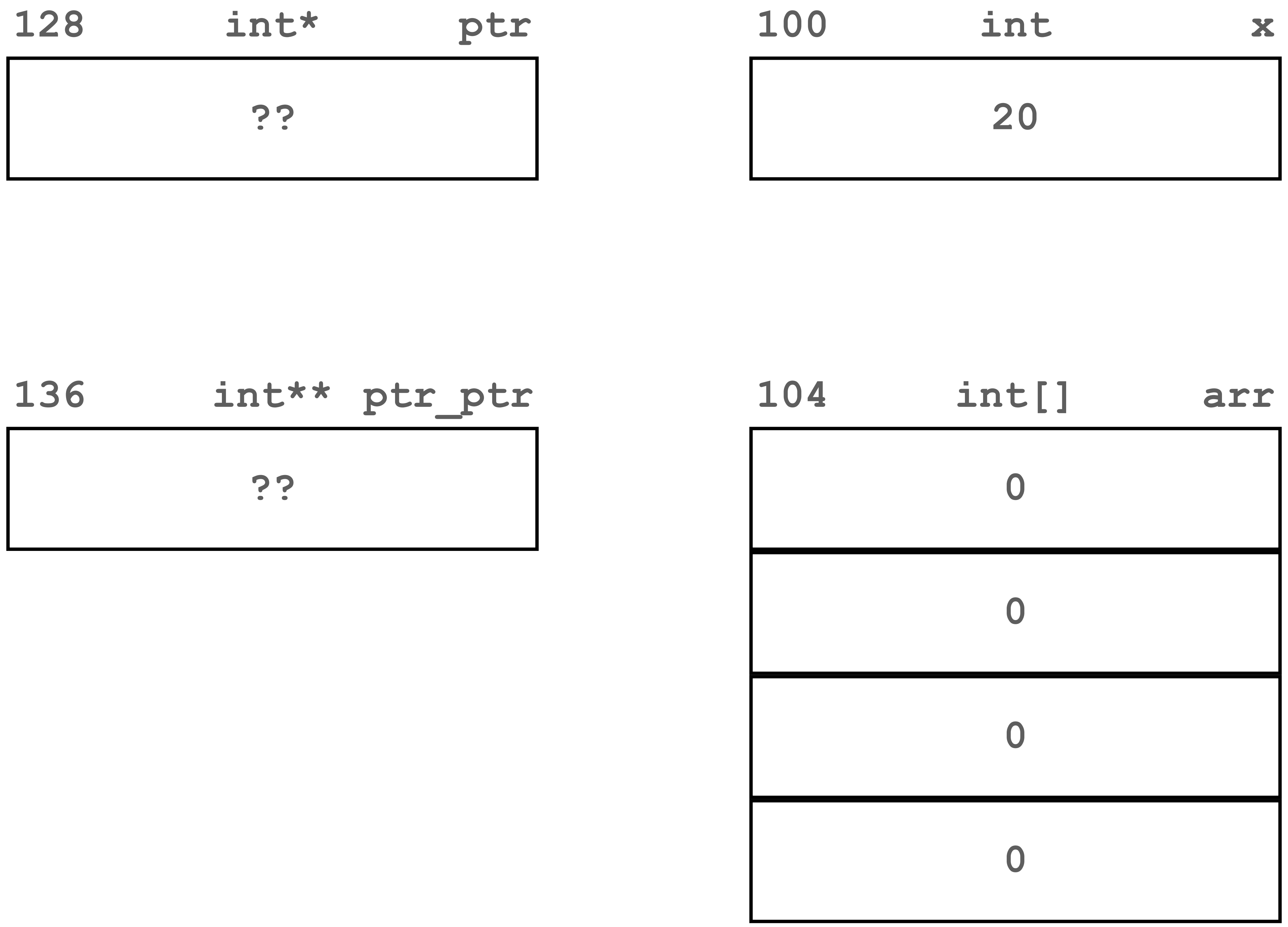
CS143: lecture 5

Byron Zhong, June 22

Pointers

Review

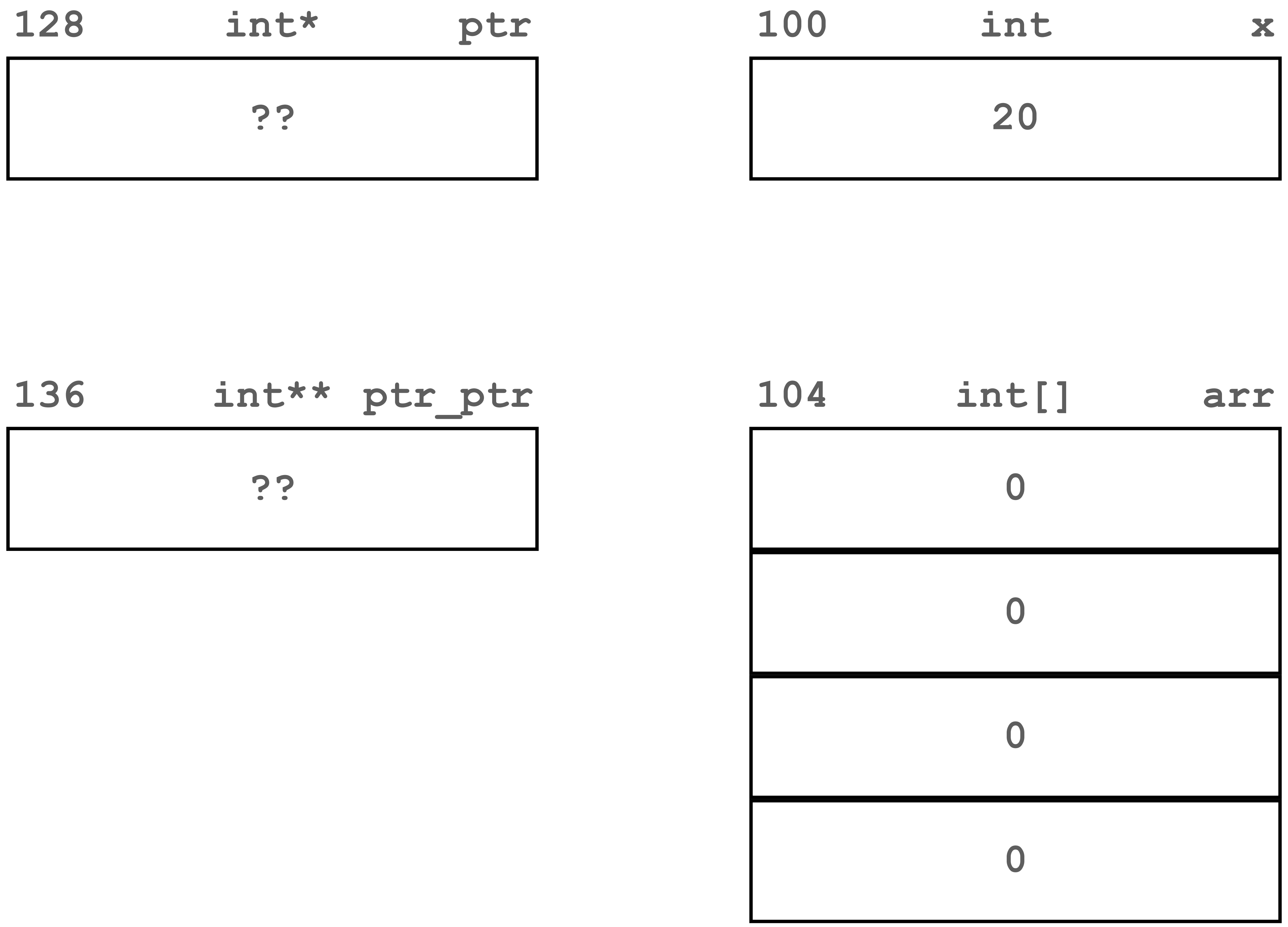
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;
```



Pointers

Review

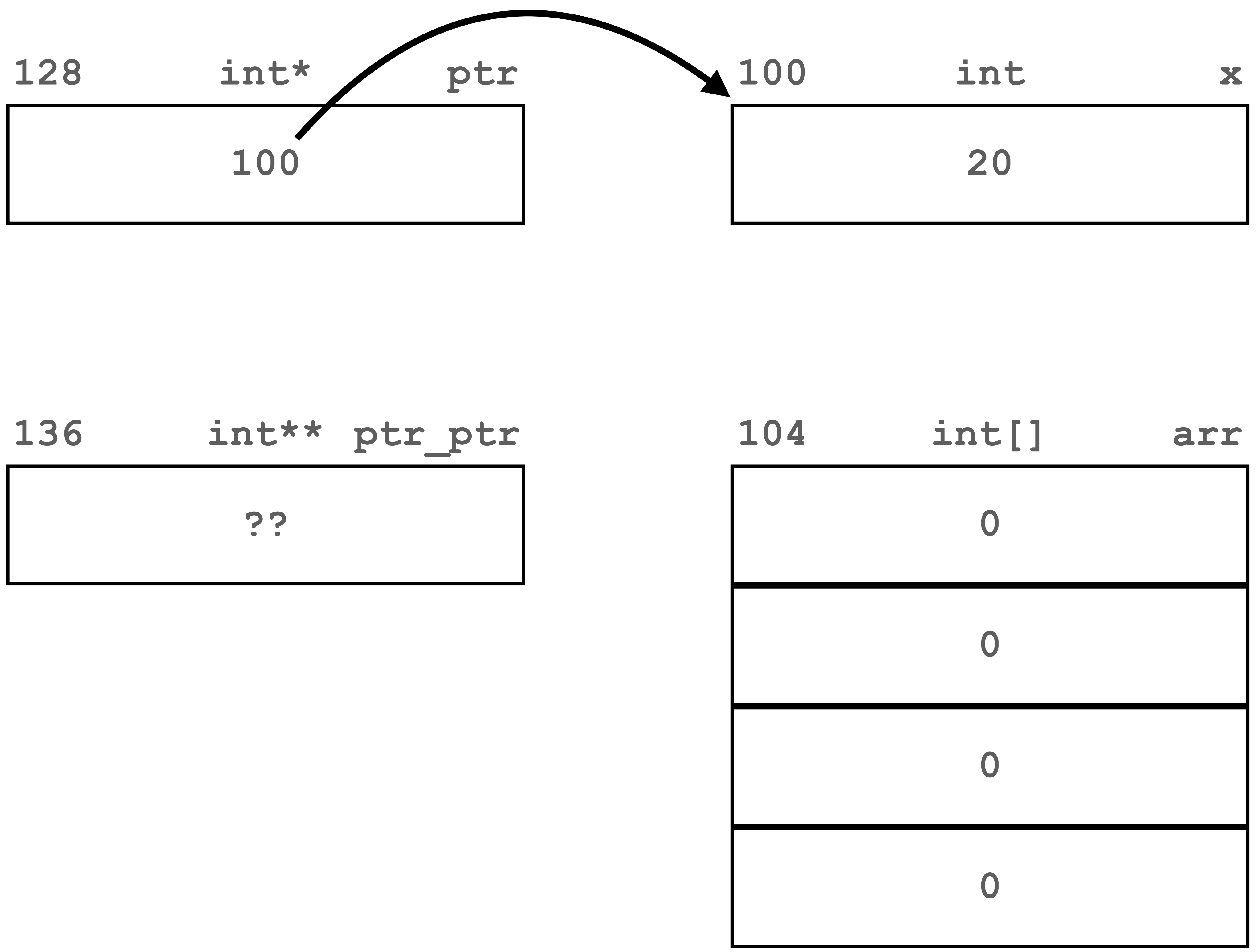
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
```



Pointers

Review

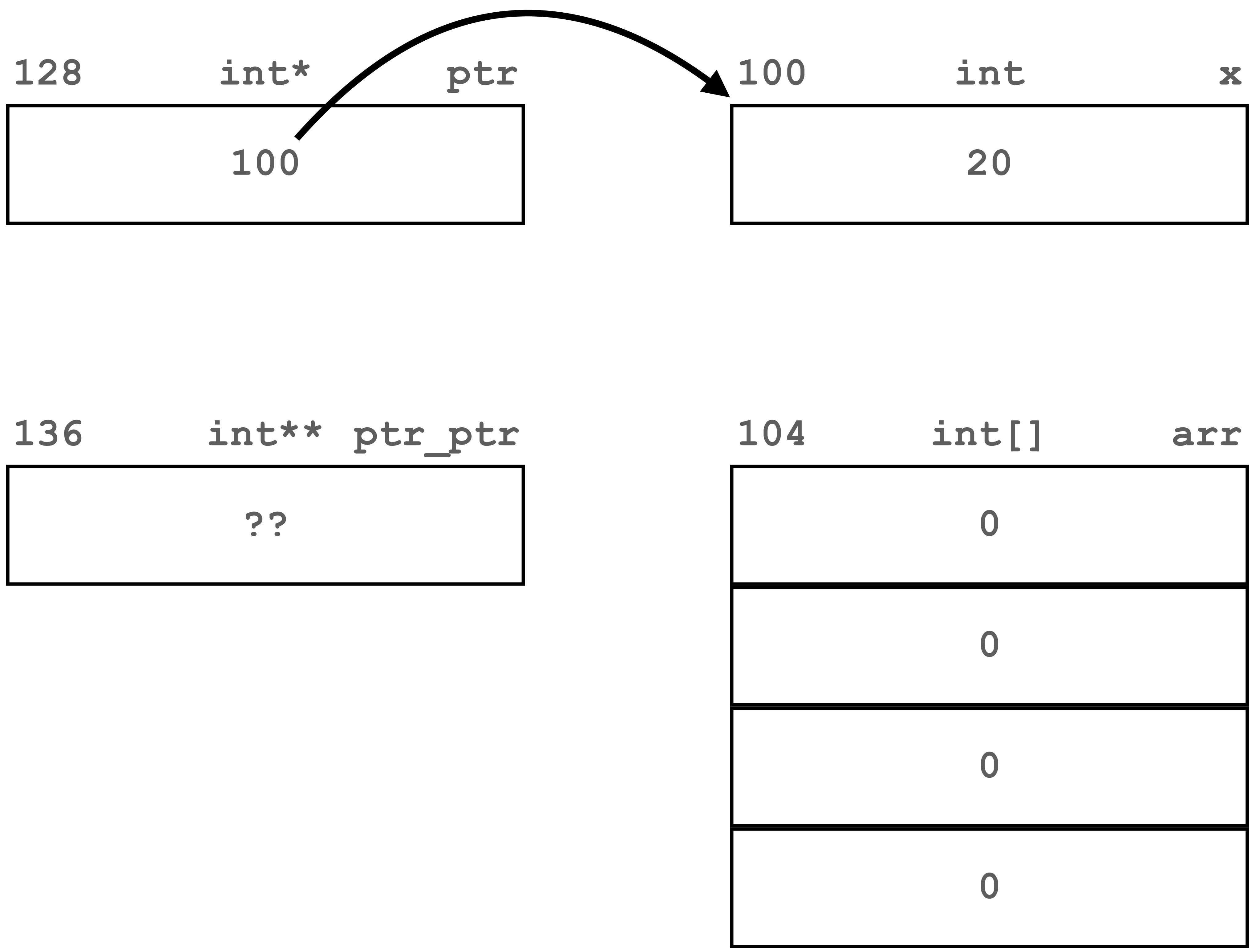
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
```



Pointers

Review

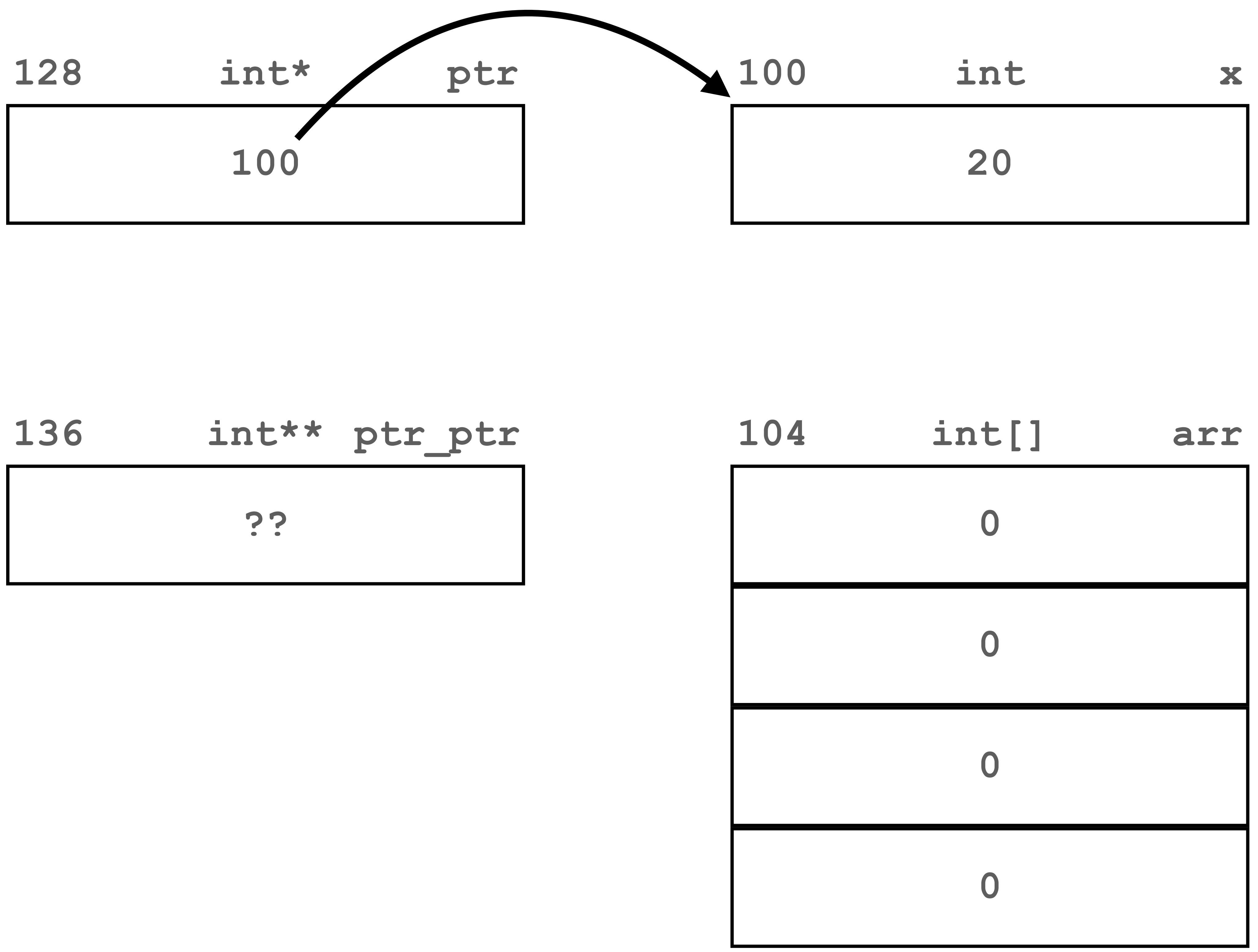
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);        <-- 20
```



Pointers

Review

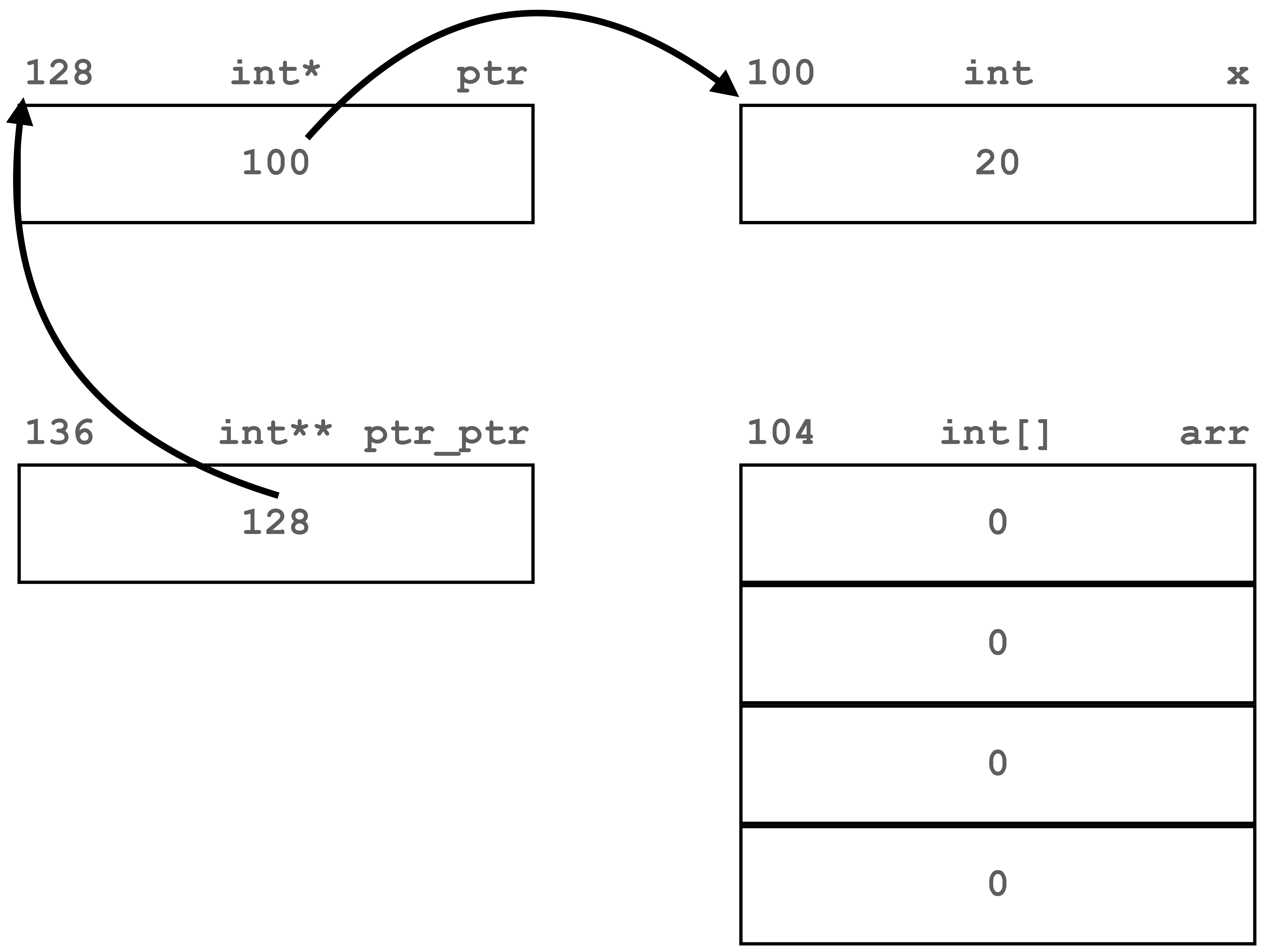
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
```



Pointers

Review

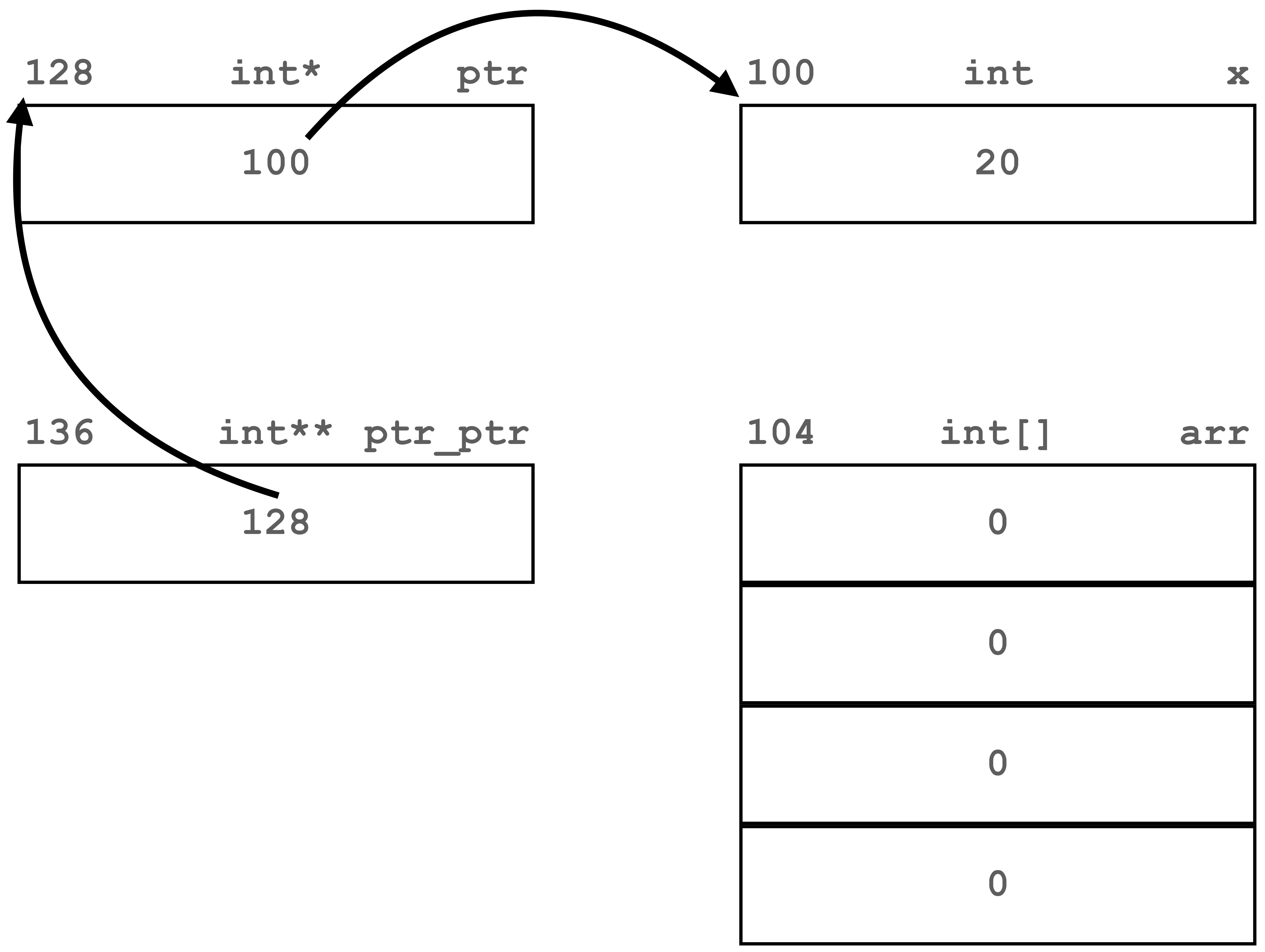
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
```



Pointers

Review

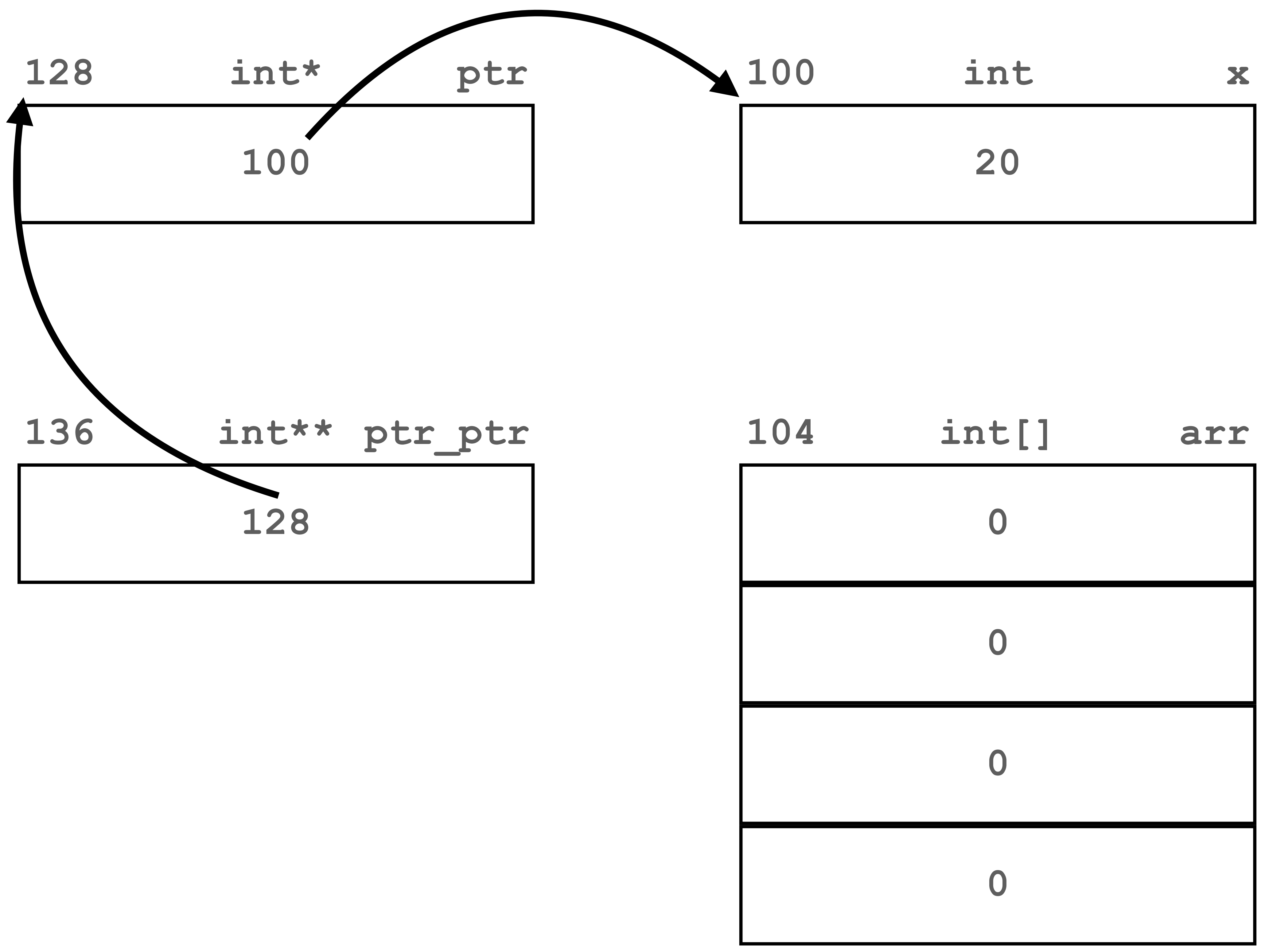
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);  <-- 100
```



Pointers

Review

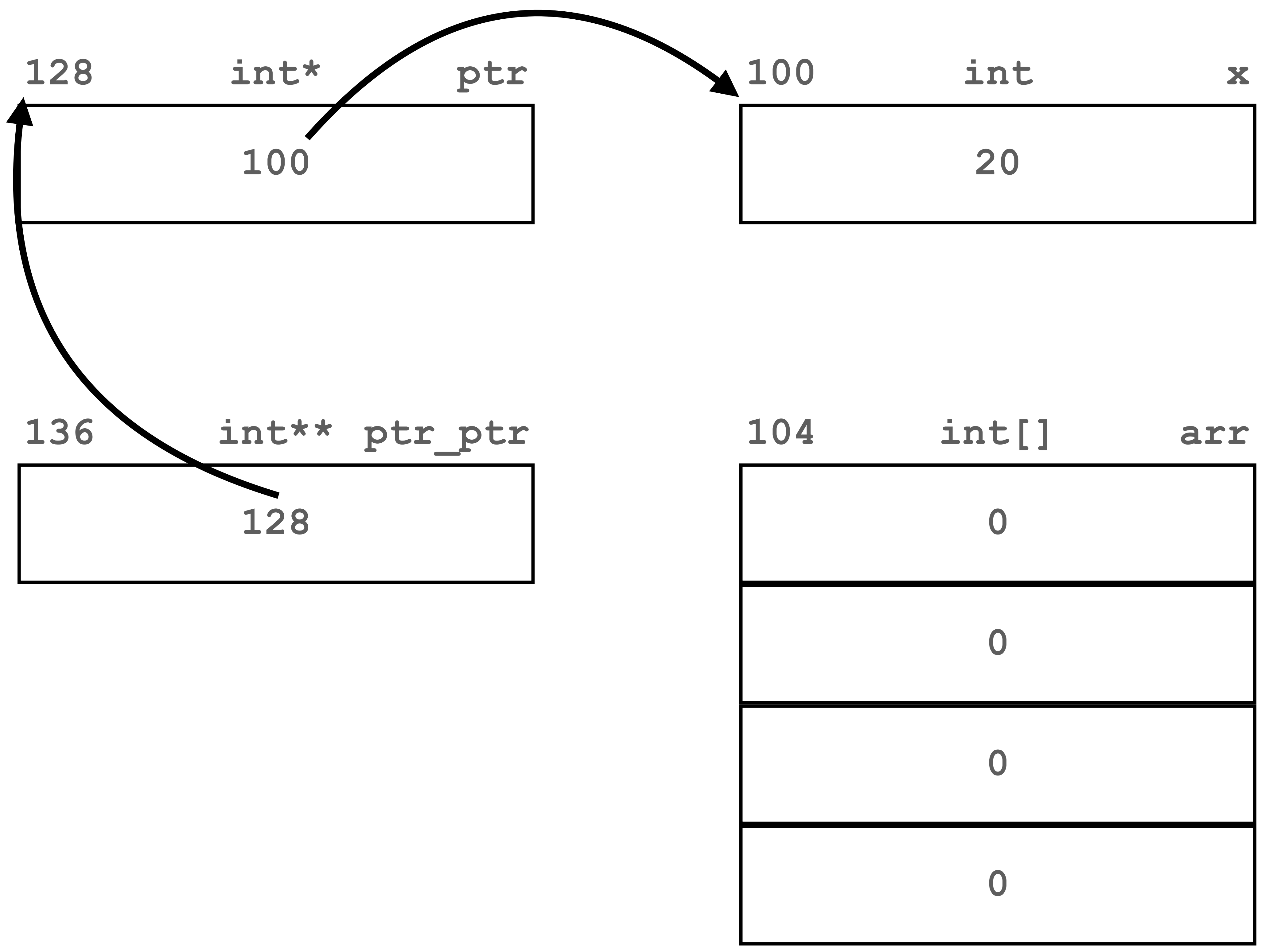
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr); <-- 20
```



Pointers

Review

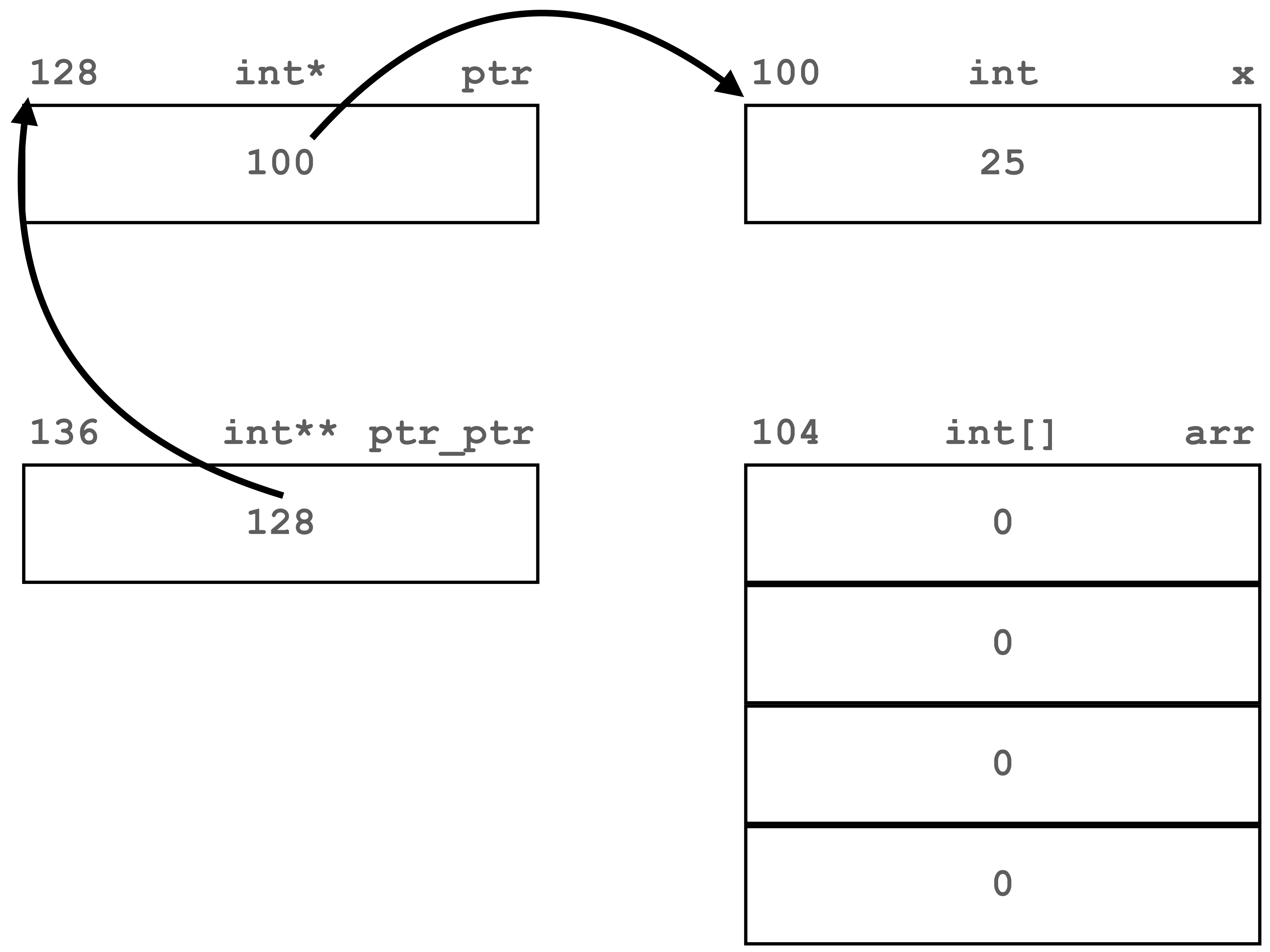
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
```



Pointers

Review

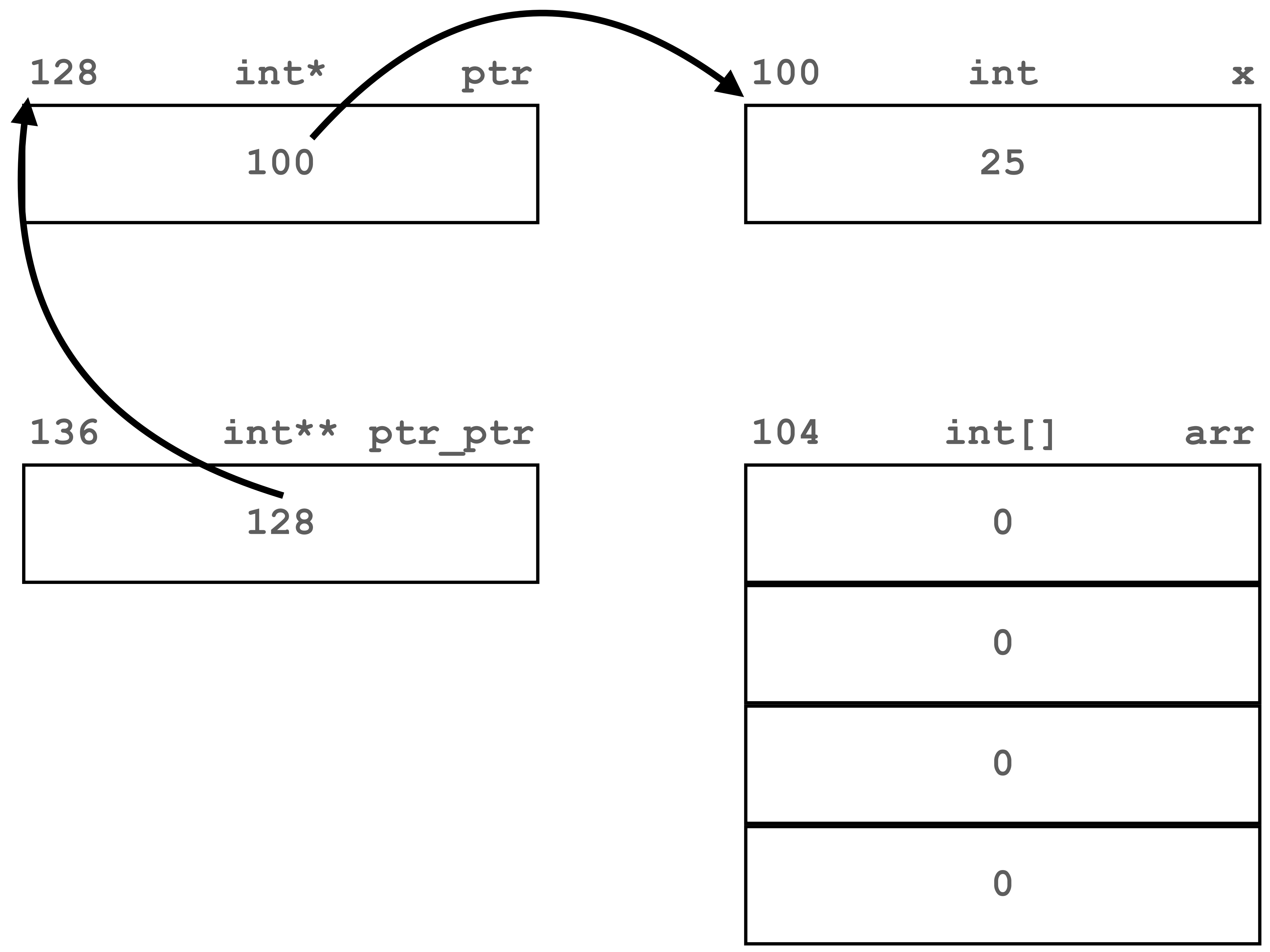
```c
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
```



Pointers

Review

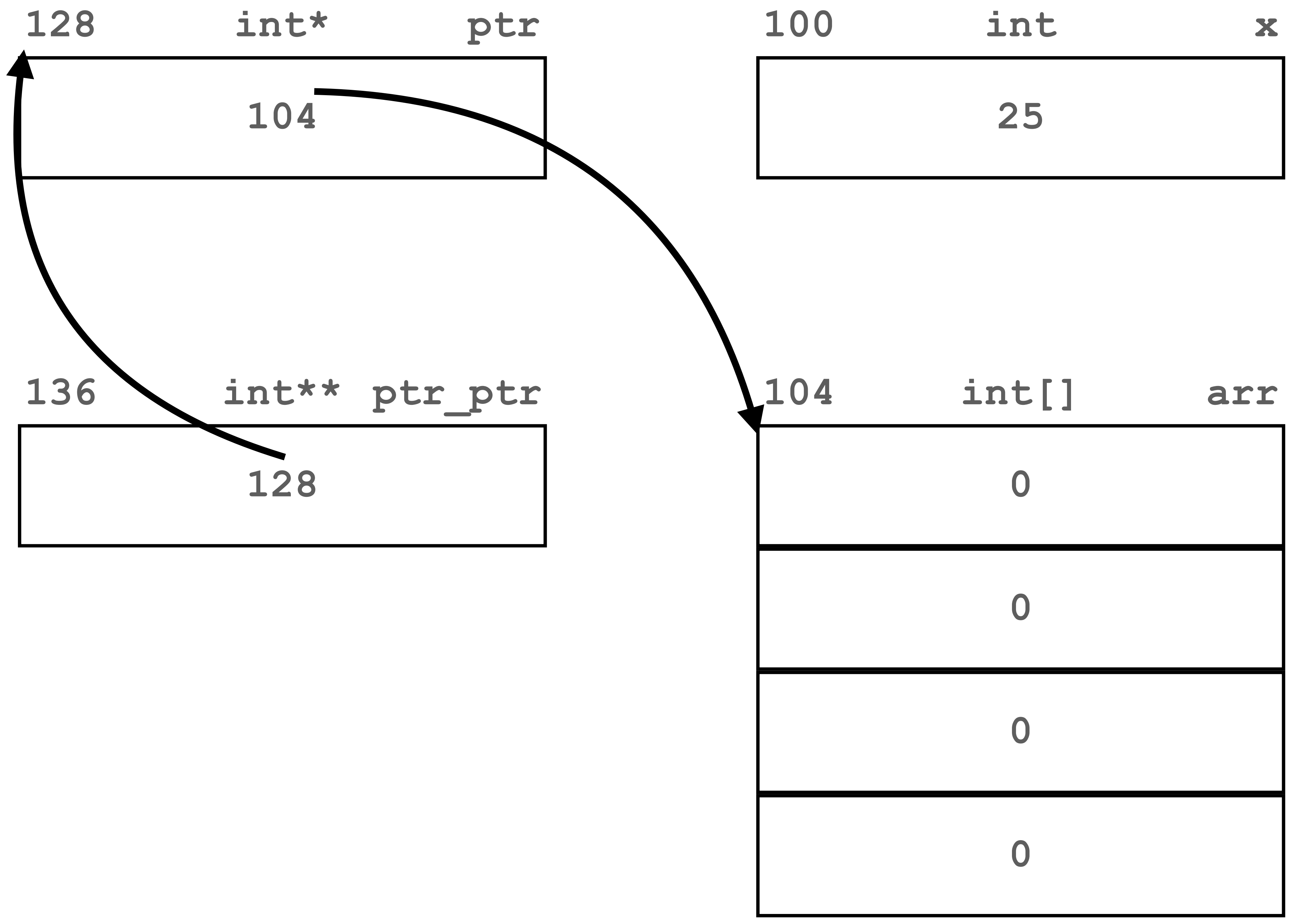
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
```



Pointers

Review

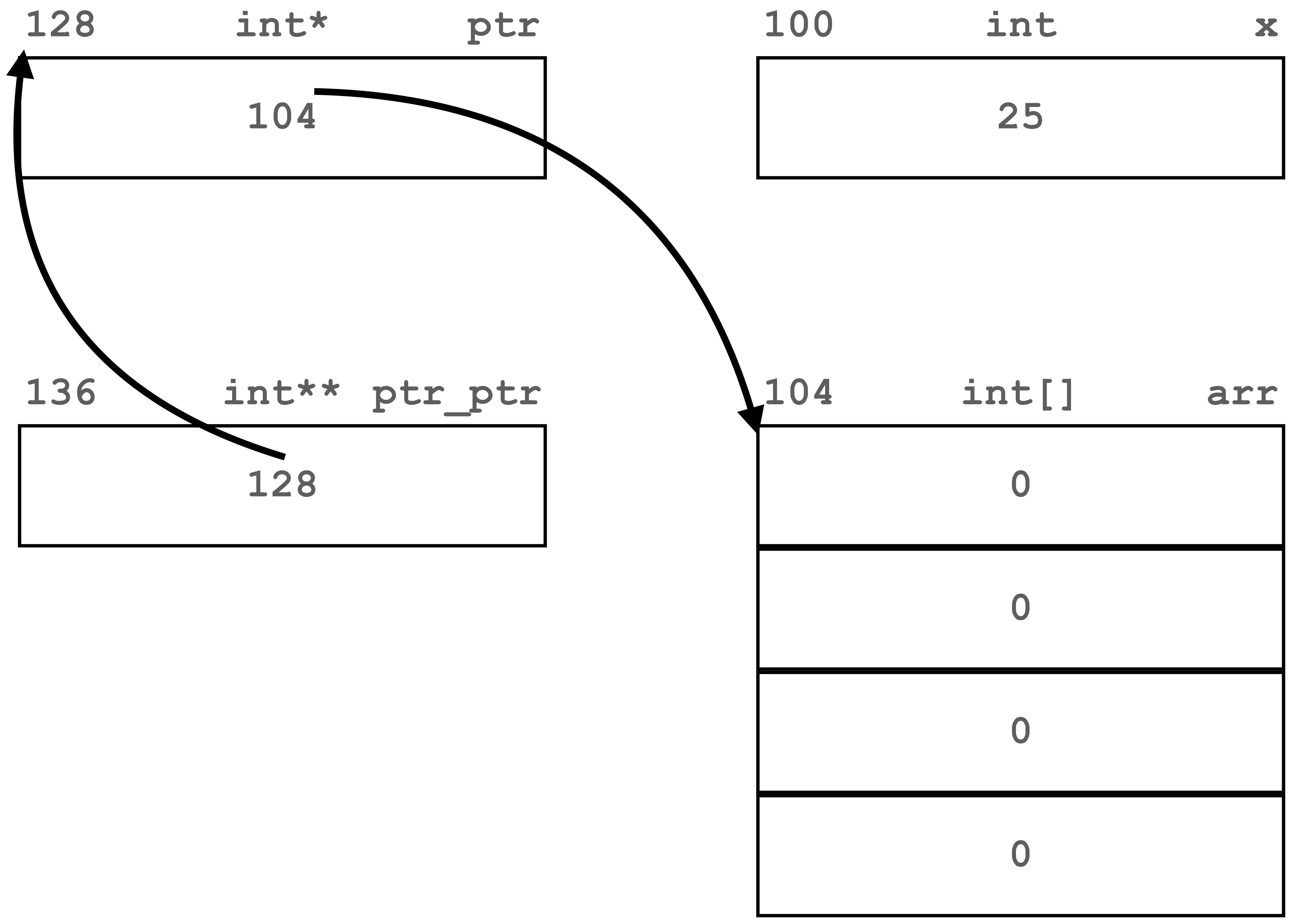
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
```



Pointers

Review

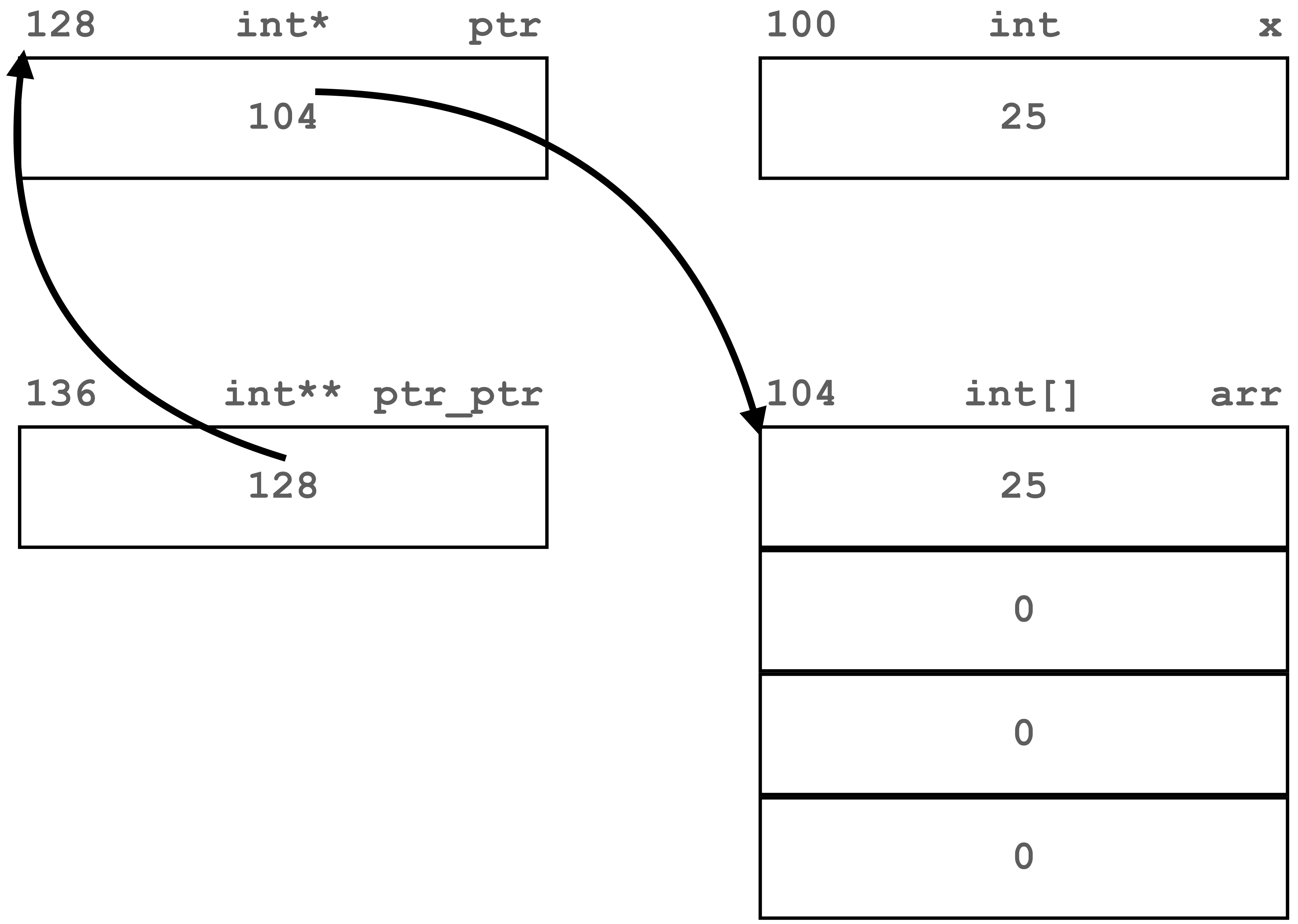
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;
```



Pointers

Review

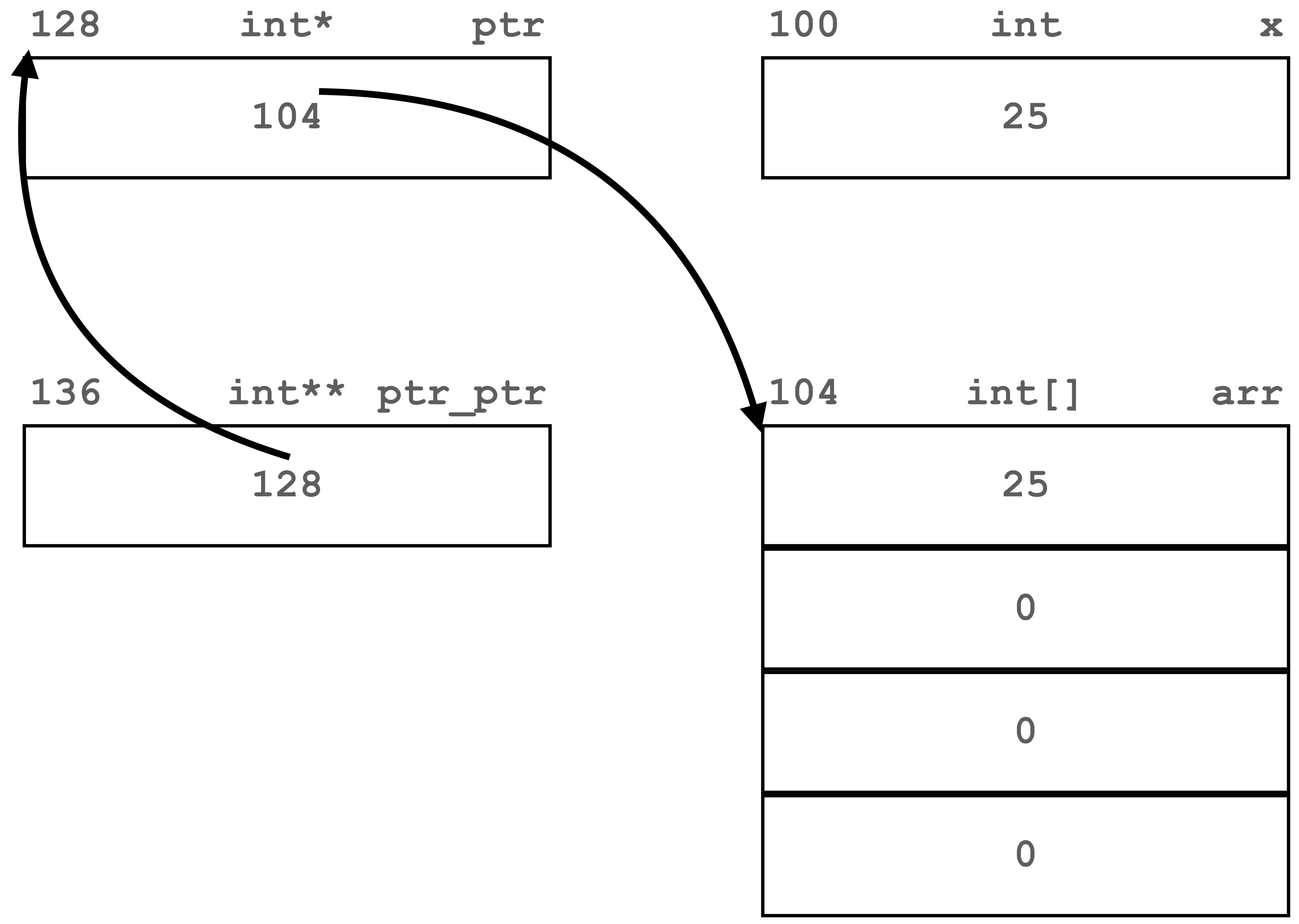
```c
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;
```



Pointers

Review

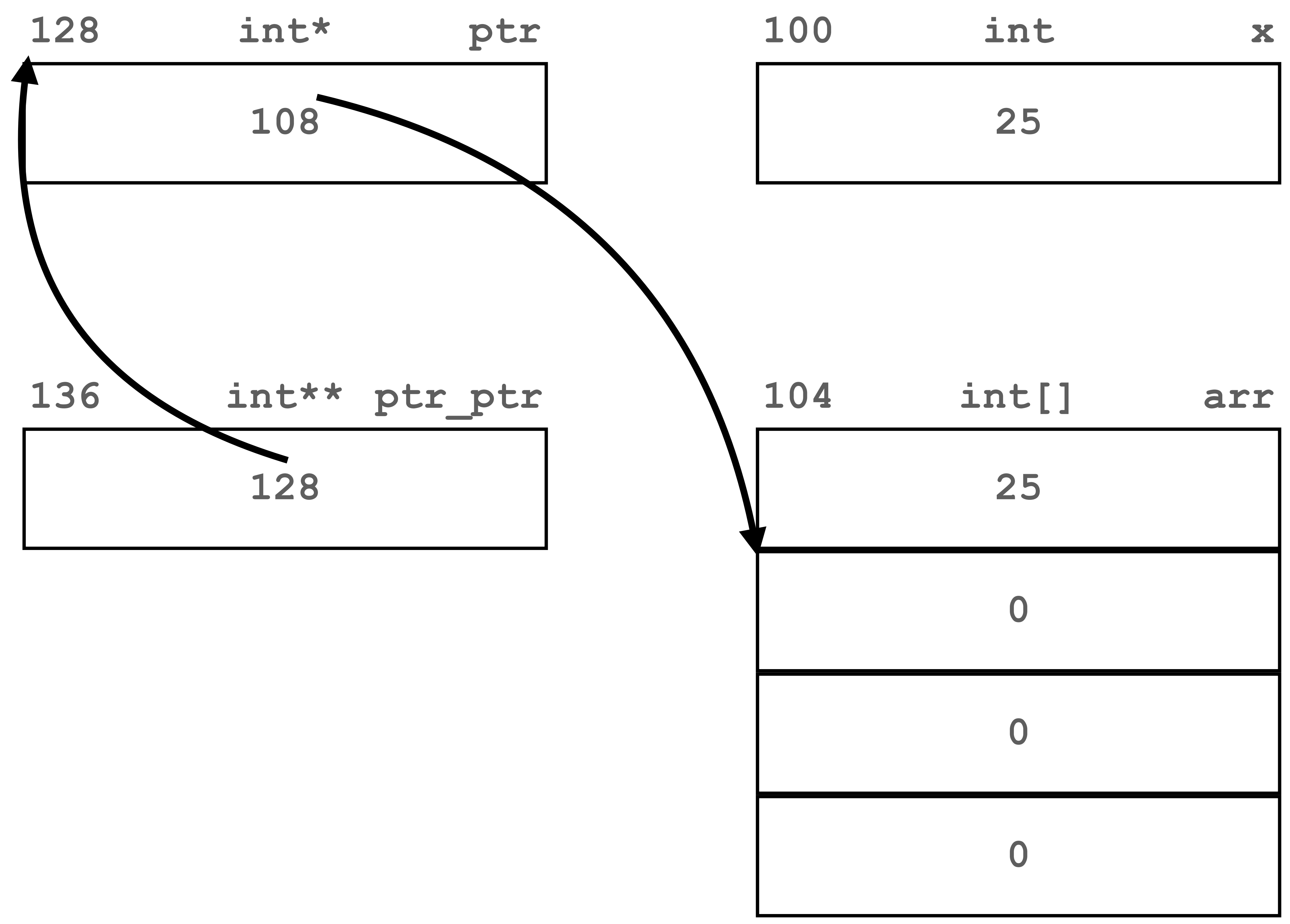
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;

*ptr_ptr = &arr[1];
```



Pointers

Review

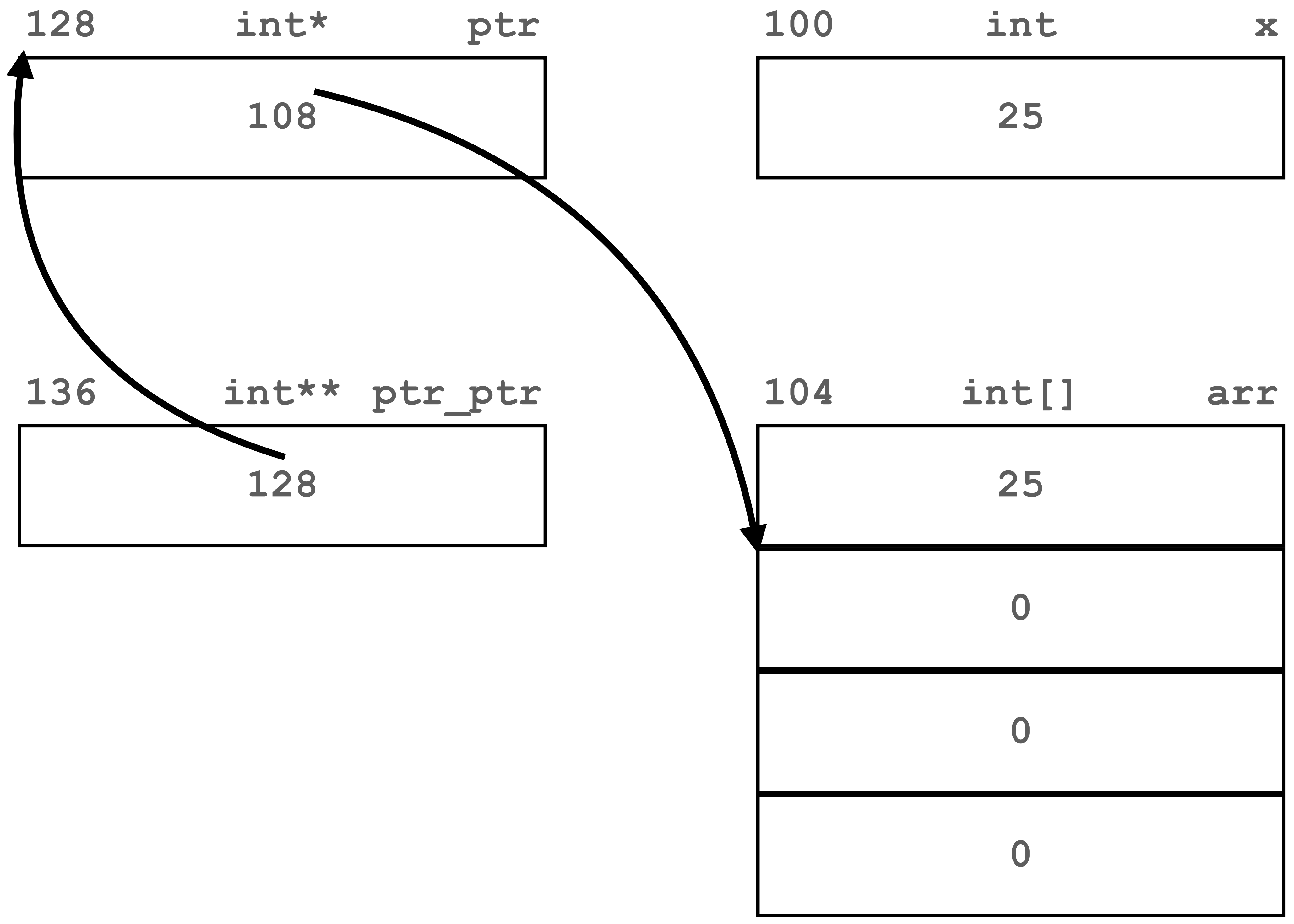
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;

*ptr_ptr = &arr[1];
```



Pointers

Review

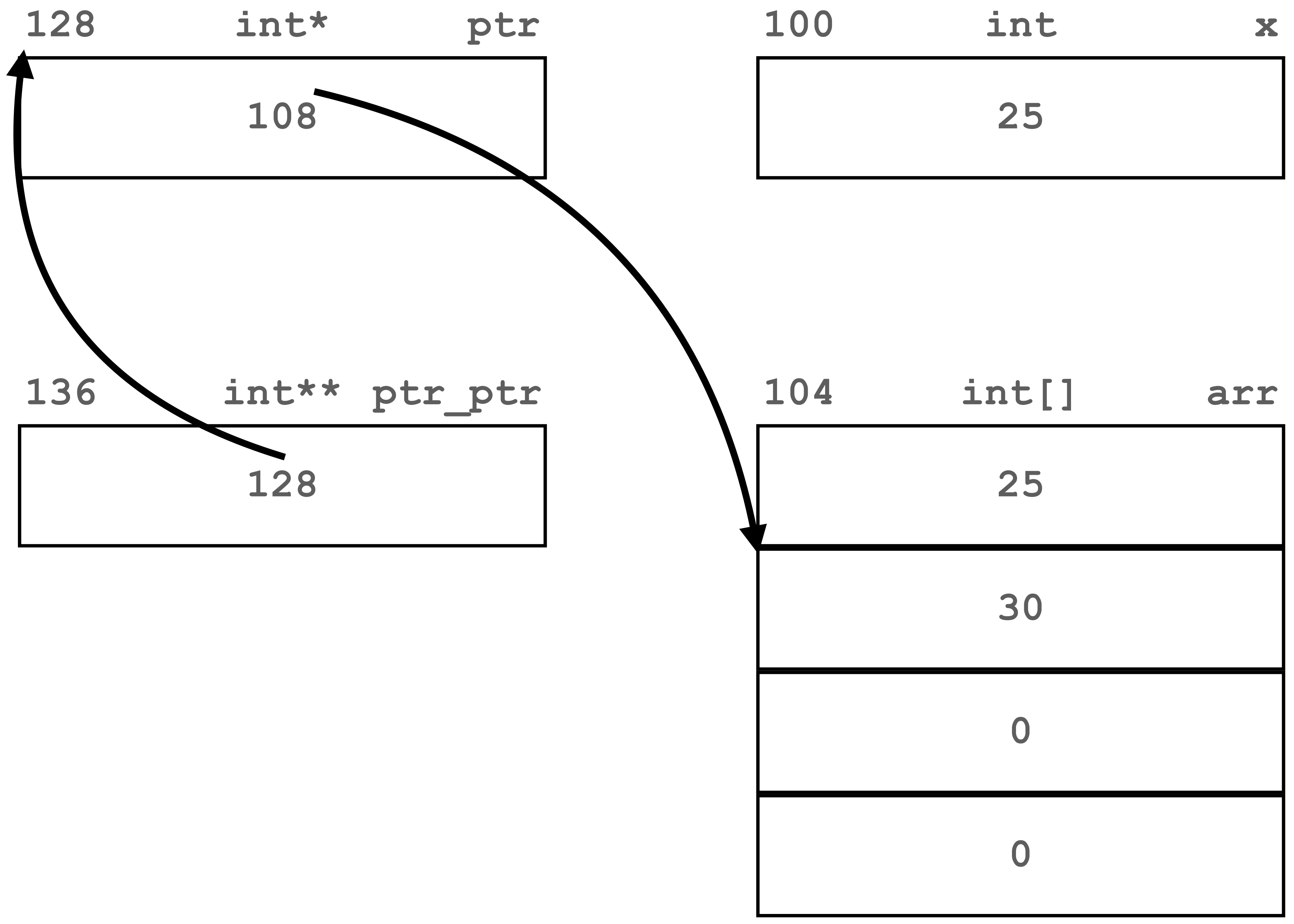
```c
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;

*ptr_ptr = &arr[1];
*ptr = 30;
```



Pointers

Review

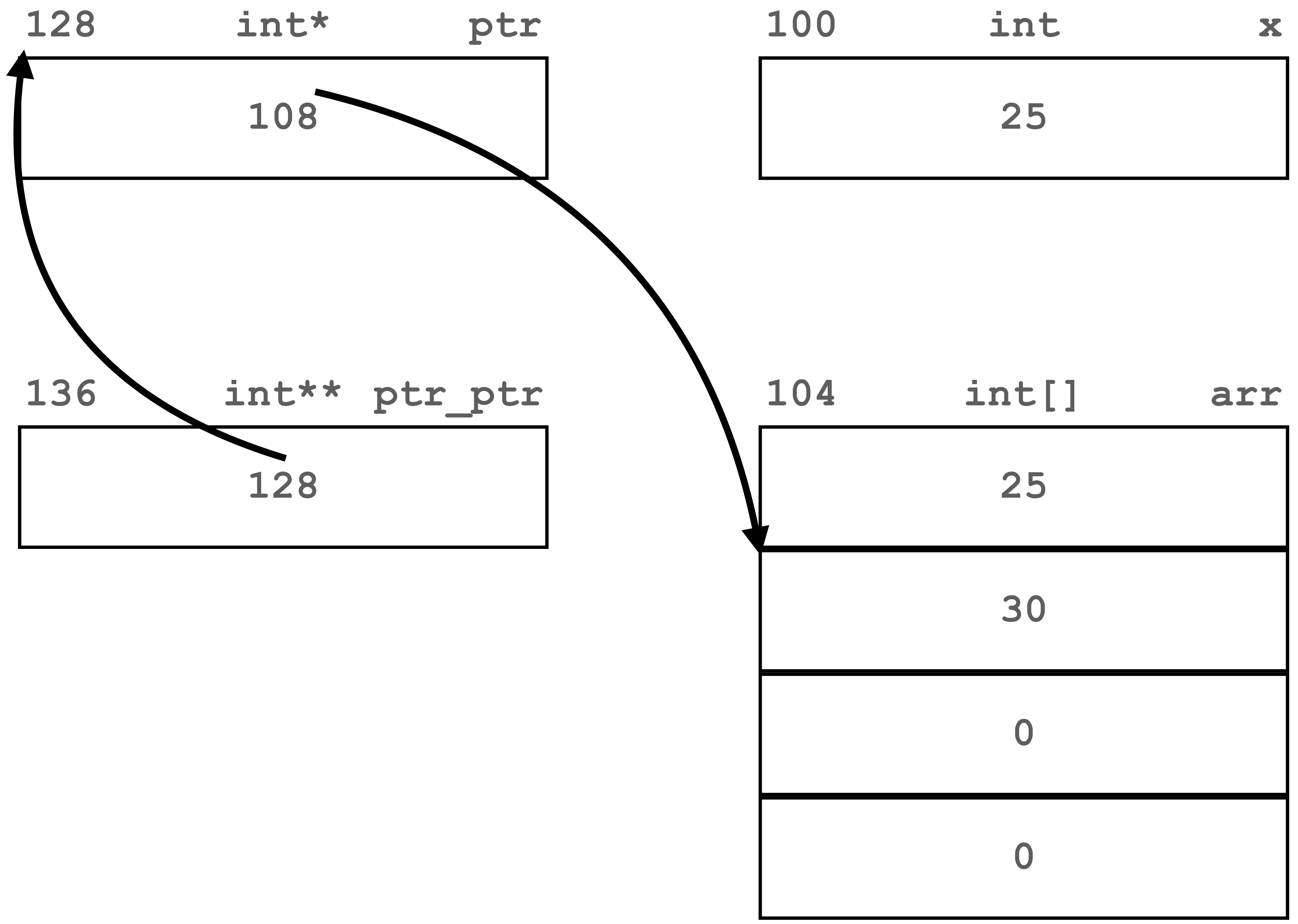
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;

*ptr_ptr = &arr[1];
*ptr = 30;
```



Pointers

Review

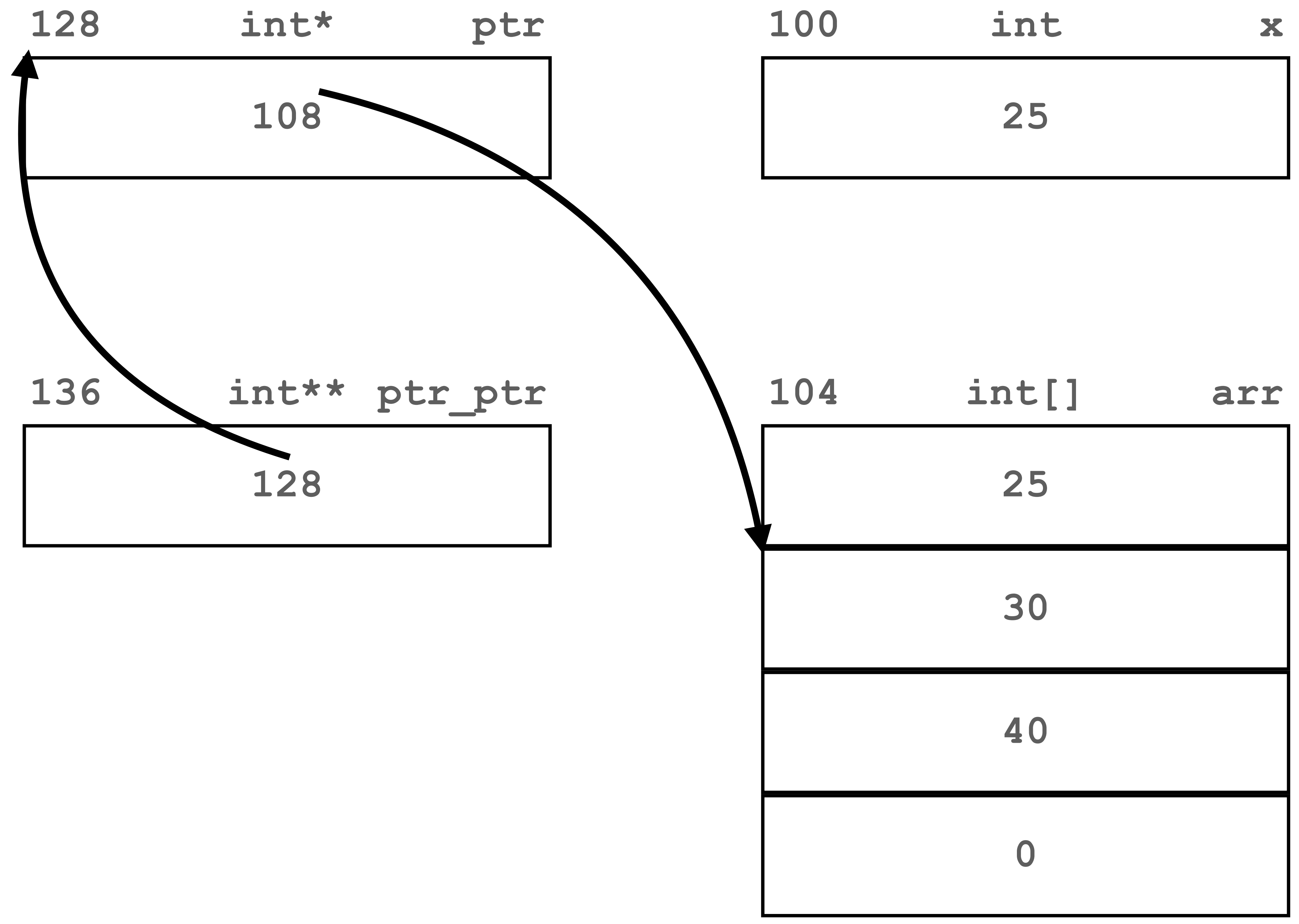
```c
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;

*ptr_ptr = &arr[1];
*ptr = 30;
ptr[1] = 40;
```



Pointers

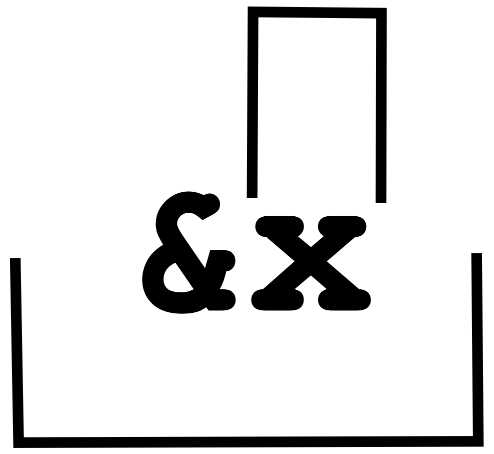
Review

```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;

*ptr_ptr = &arr[1];
*ptr = 30;
ptr[1] = 40;
```

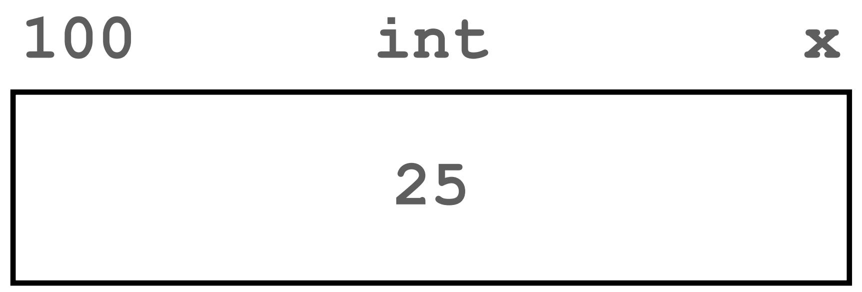


Pointers
Review

type : int
value: 25

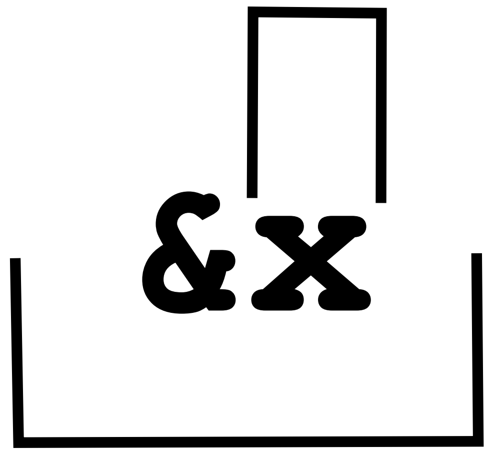


type : int *
value: 100

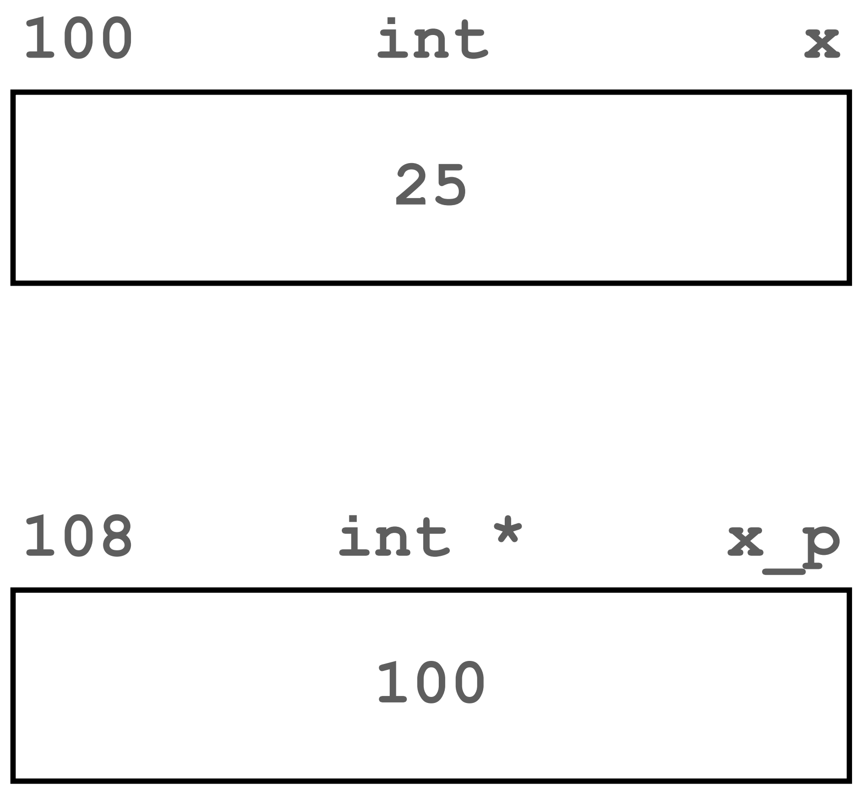


Pointers
Review

```
type : int
value: 25
```




```
type : int *
value: 100
```

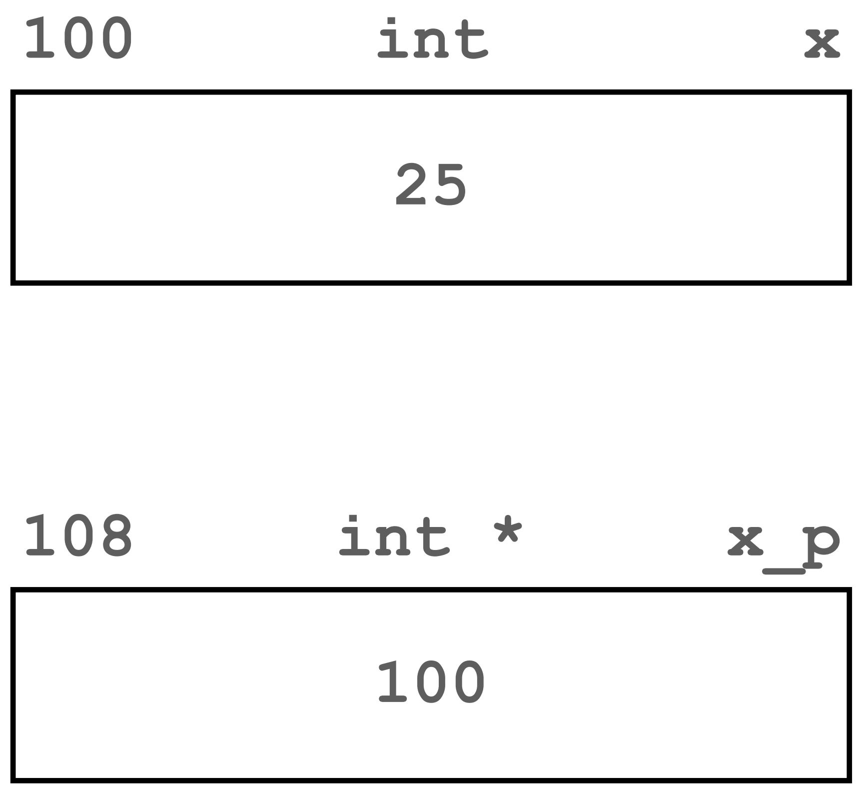


Pointers
Review

```
type : int
value: 25
```

&x

```
type : int *
value: 100
```




```
type : int *
value: 100
```

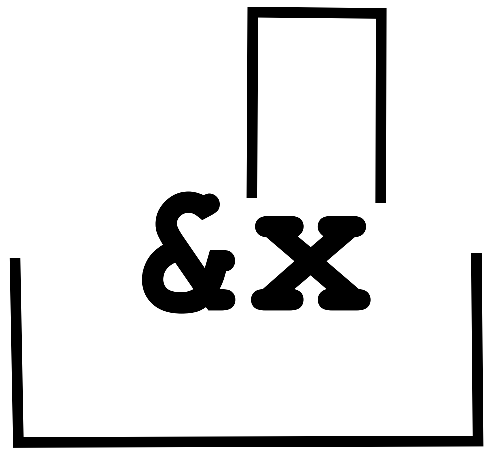
***x_p**

```
type : int
value: 25
```

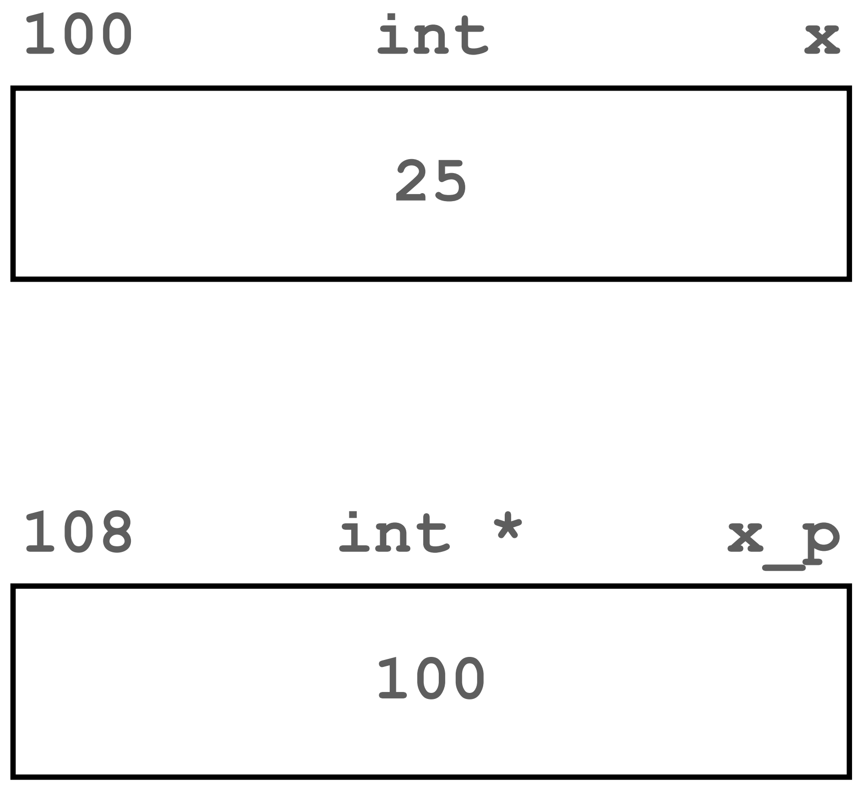
Pointers

Review

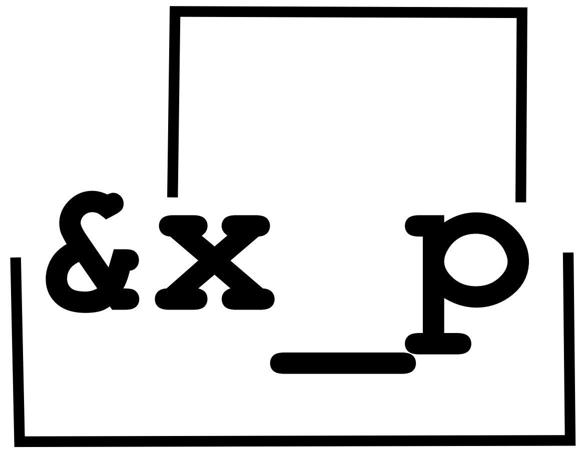
type : int
value: 25



type : int *
value: 100

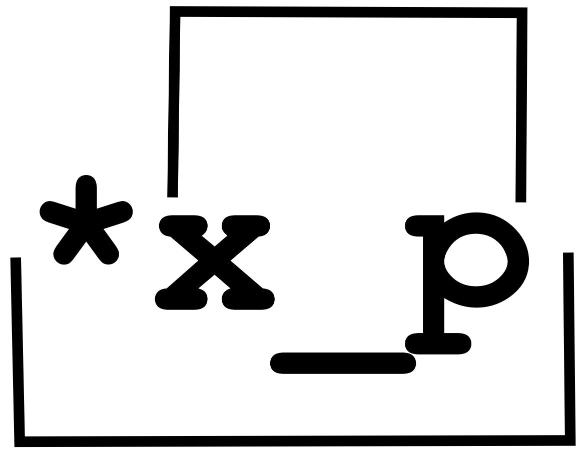


type : int *
value: 100



type : int **
value: 108

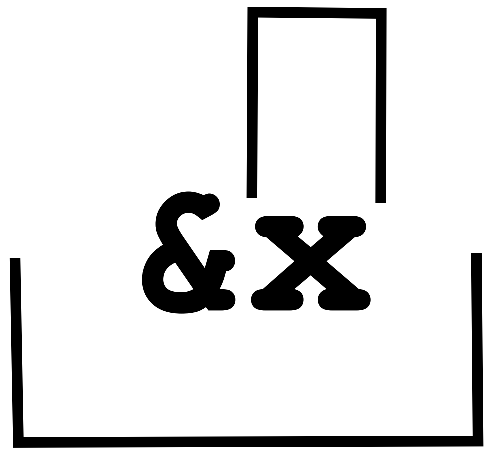
type : int *
value: 100



type : int
value: 25

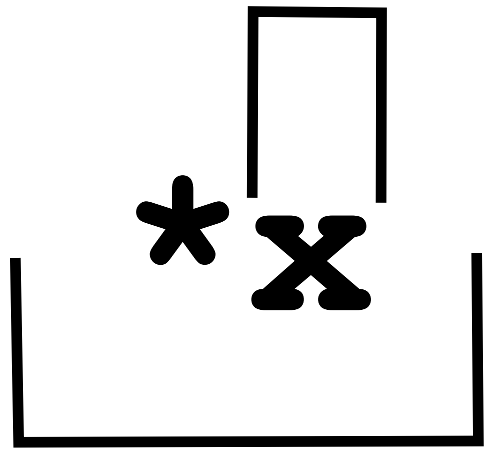
Pointers
Review

type : int
value: 25

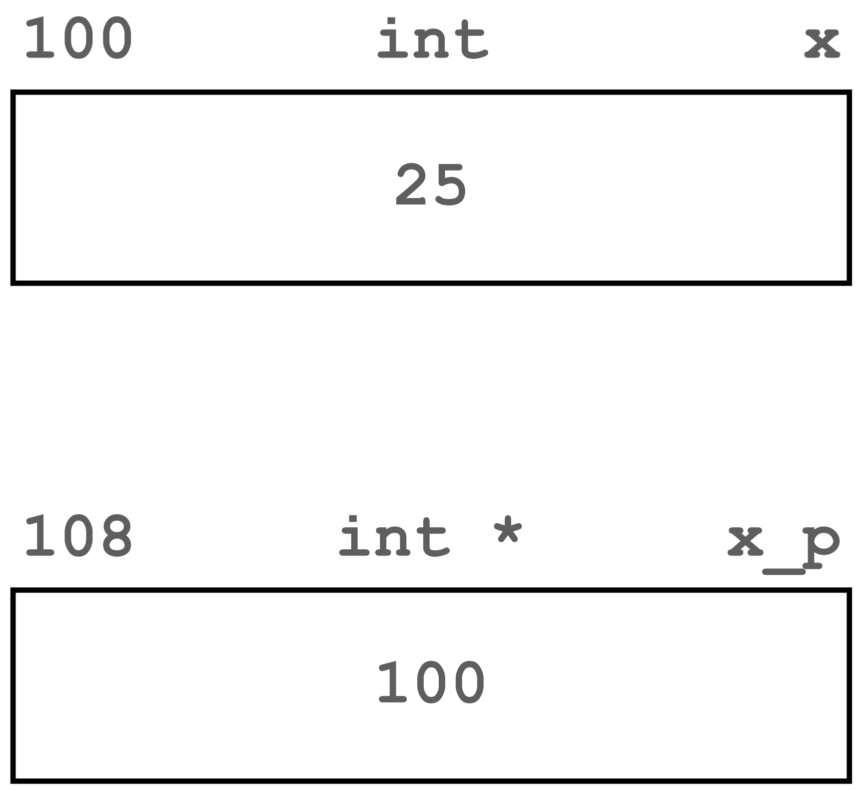


type : int *
value: 100

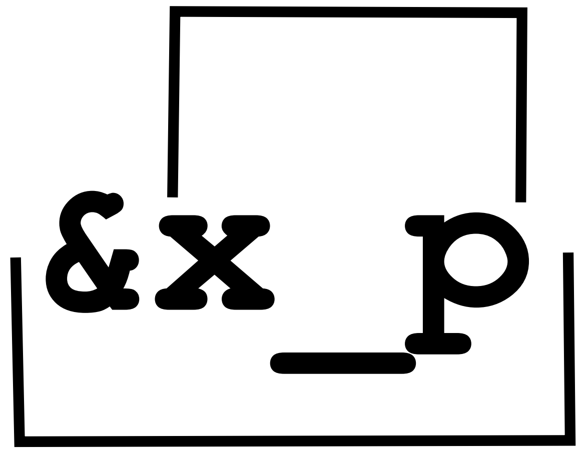
type : int
value: 25



error

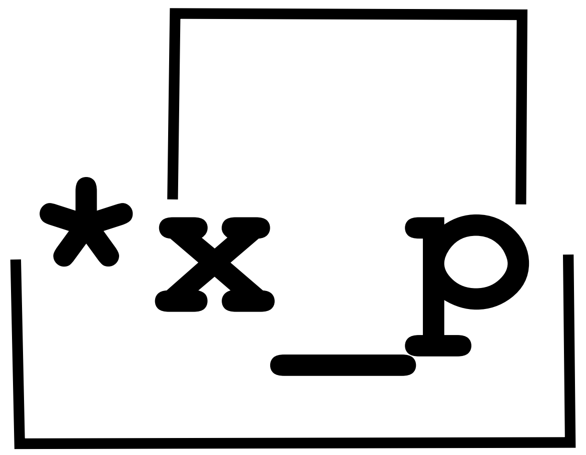


type : int *
value: 100



type : int **
value: 108

type : int *
value: 100

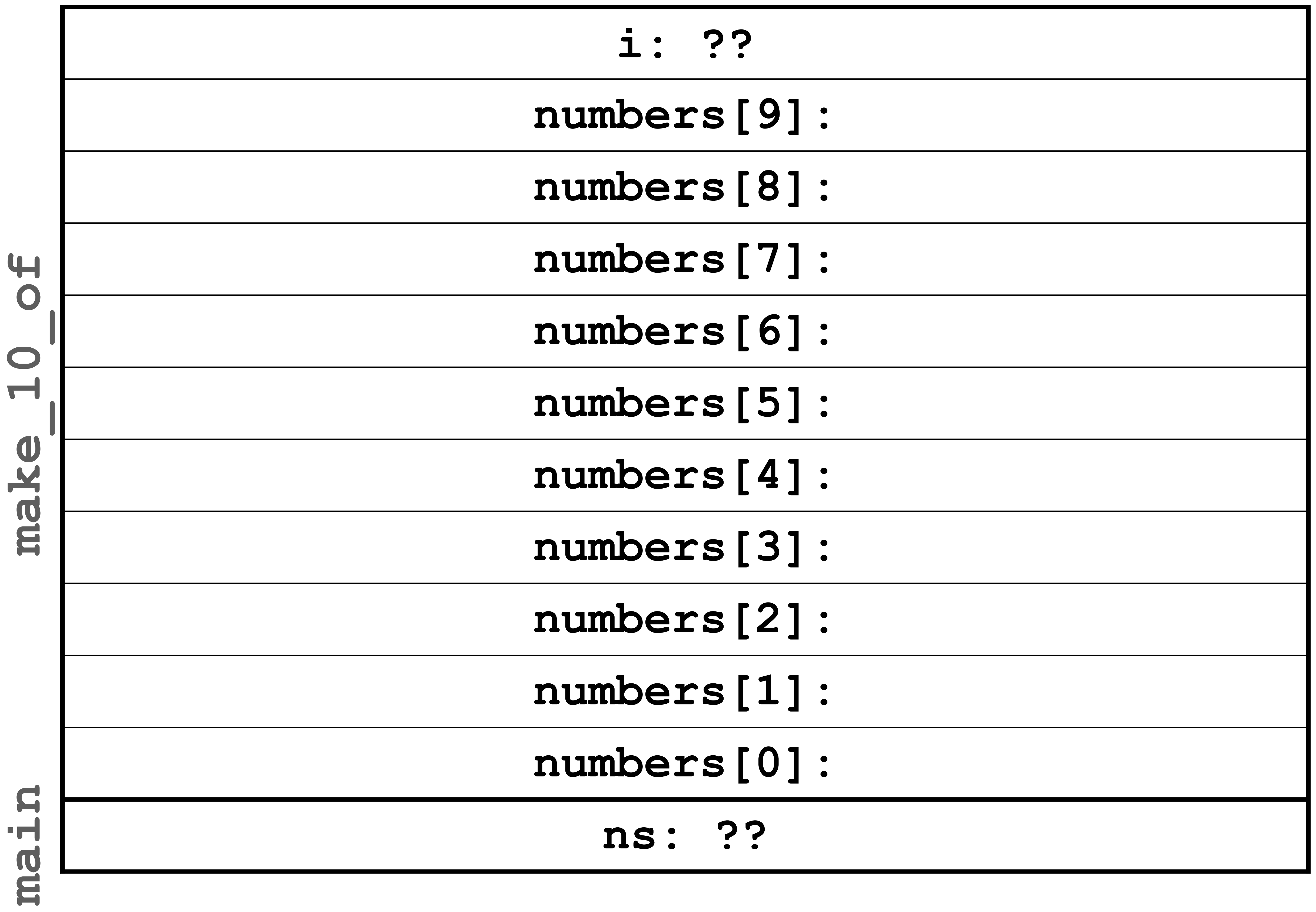


type : int
value: 25

Pointers

How about returning an array?

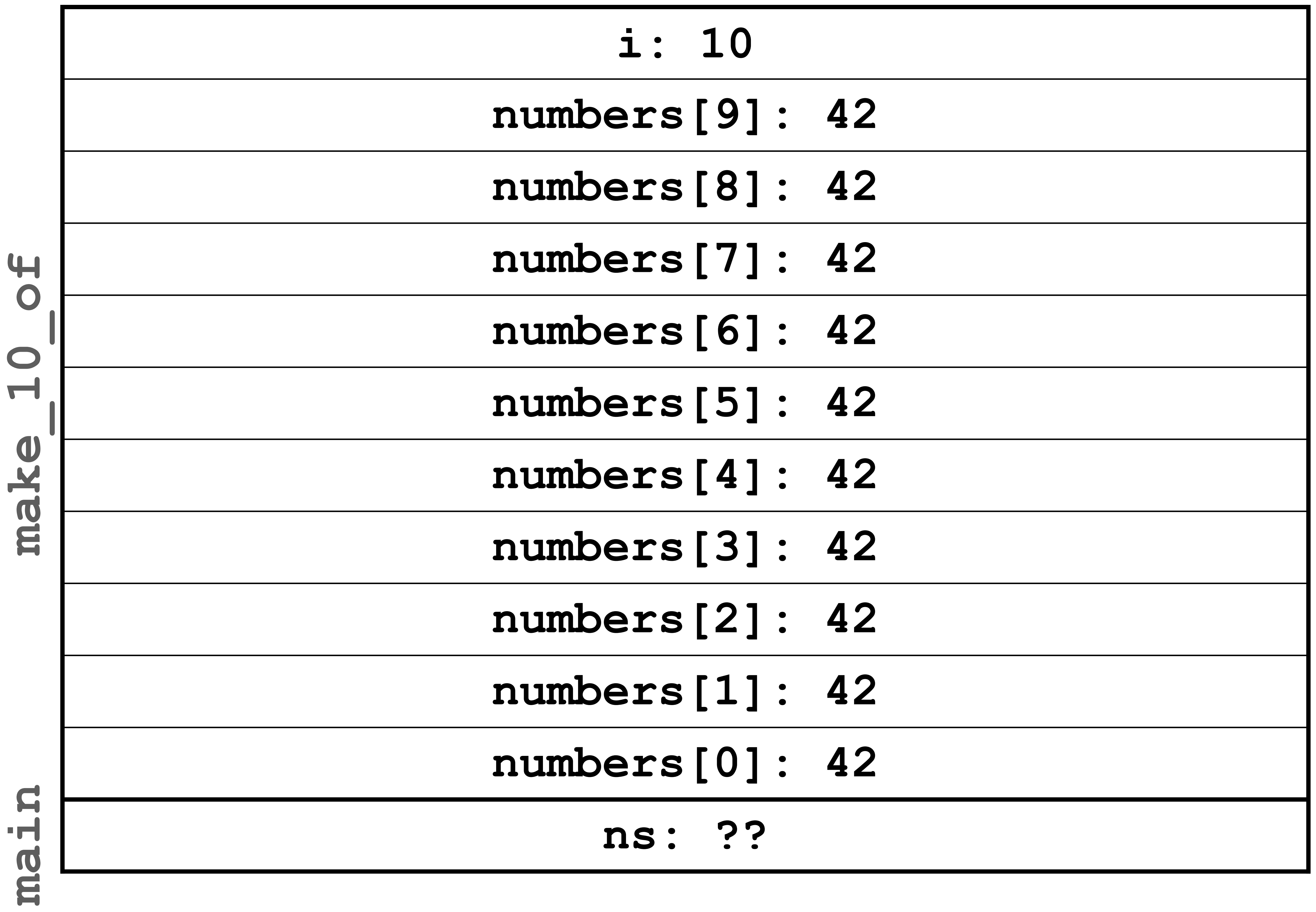
```c
int *make_10_of(int n)
{
        int numbers[10];

        for (int i = 0; i < 10; i++) {
                numbers[i] = n;
        }

        return numbers;
}


int main()
{
        int *ns = make_10_of(42);
        ...
        return 0;
}
```



Pointers

How about returning an array?

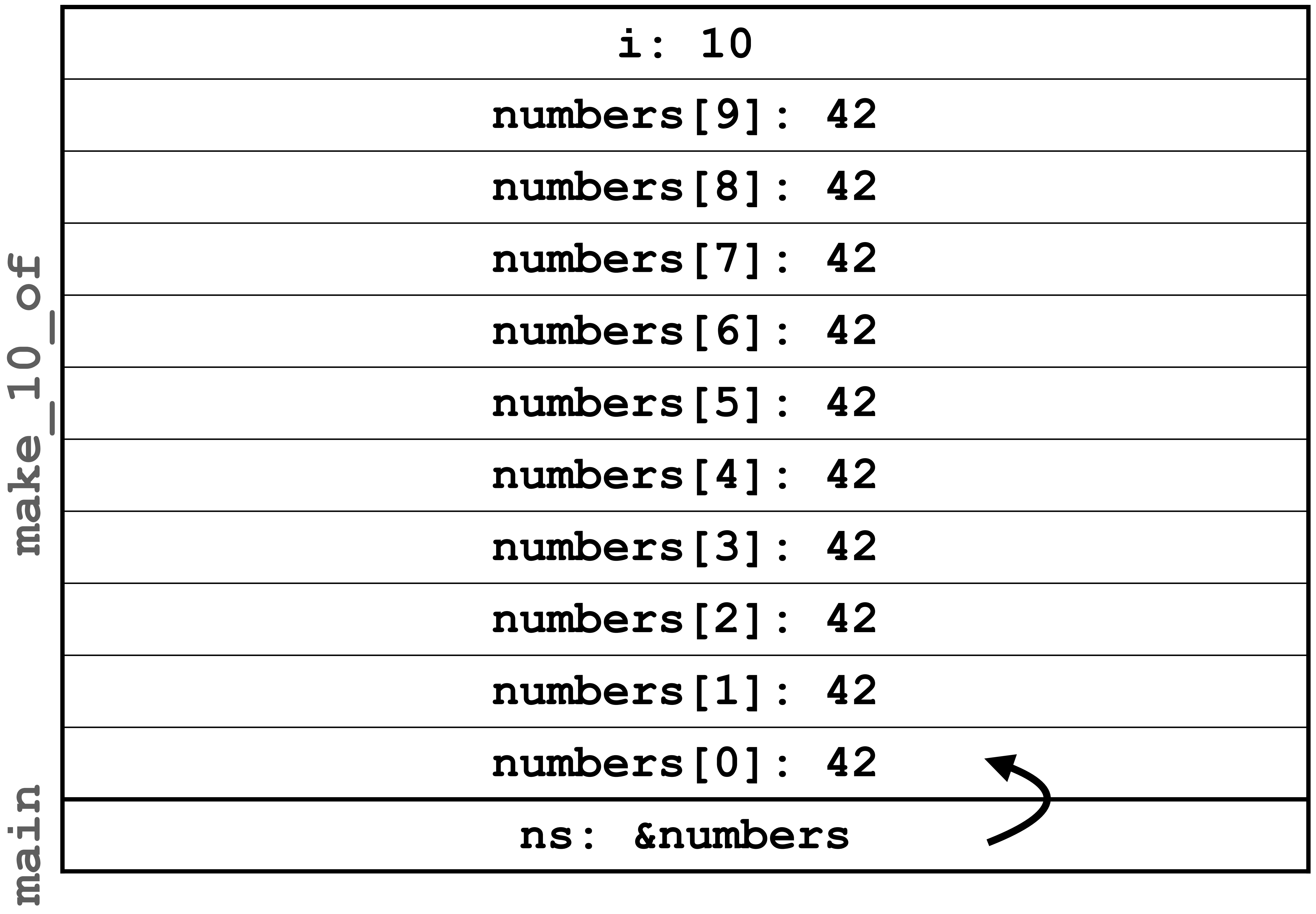
```
int *make_10_of(int n)
{
        int numbers[10];

        for (int i = 0; i < 10; i++) {
                numbers[i] = n;
        }

        return numbers;
}


int main()
{
        int *ns = make_10_of(42);
        ...
        return 0;
}
```



Pointers

How about returning an array?

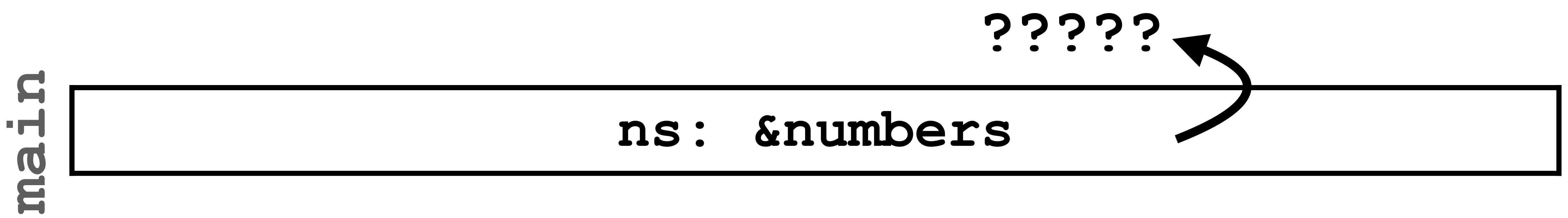
```
int *make_10_of(int n)
{
        int numbers[10];

        for (int i = 0; i < 10; i++) {
                numbers[i] = n;
        }

        return numbers;
}


int main()
{
        int *ns = make_10_of(42);
        ...
        return 0;
}
```



Pointers

How about returning an array?

```
int *make_10_of(int n)
{
        int numbers[10];

        for (int i = 0; i < 10; i++) {
                numbers[i] = n;
        }

        return numbers;
}


int main()
{
        int *ns = make_10_of(42);
        ...
        return 0;
}
```

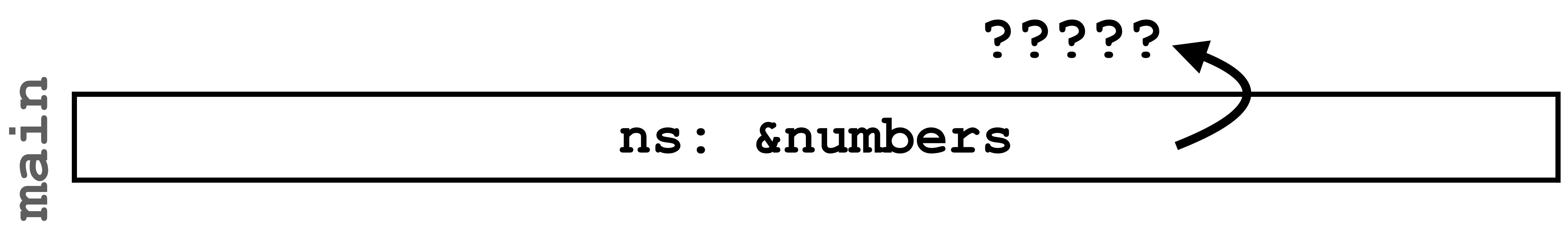


Pointers

How about returning an array?

```
int *make_10_of(int n)
{
        int numbers[10];

        for (int i = 0; i < 10; i++) {
                numbers[i] = n;
        }

        return numbers;
}


int main()
{
        int *ns = make_10_of(42);
        ...
        return 0;
}
```

- `numbers` is recycled after returning
- `ns` becomes a pointer to invalid/ non-existent/dead data.
- We call `ns` a dangling pointer



Pointers

How about variable-size arrays?

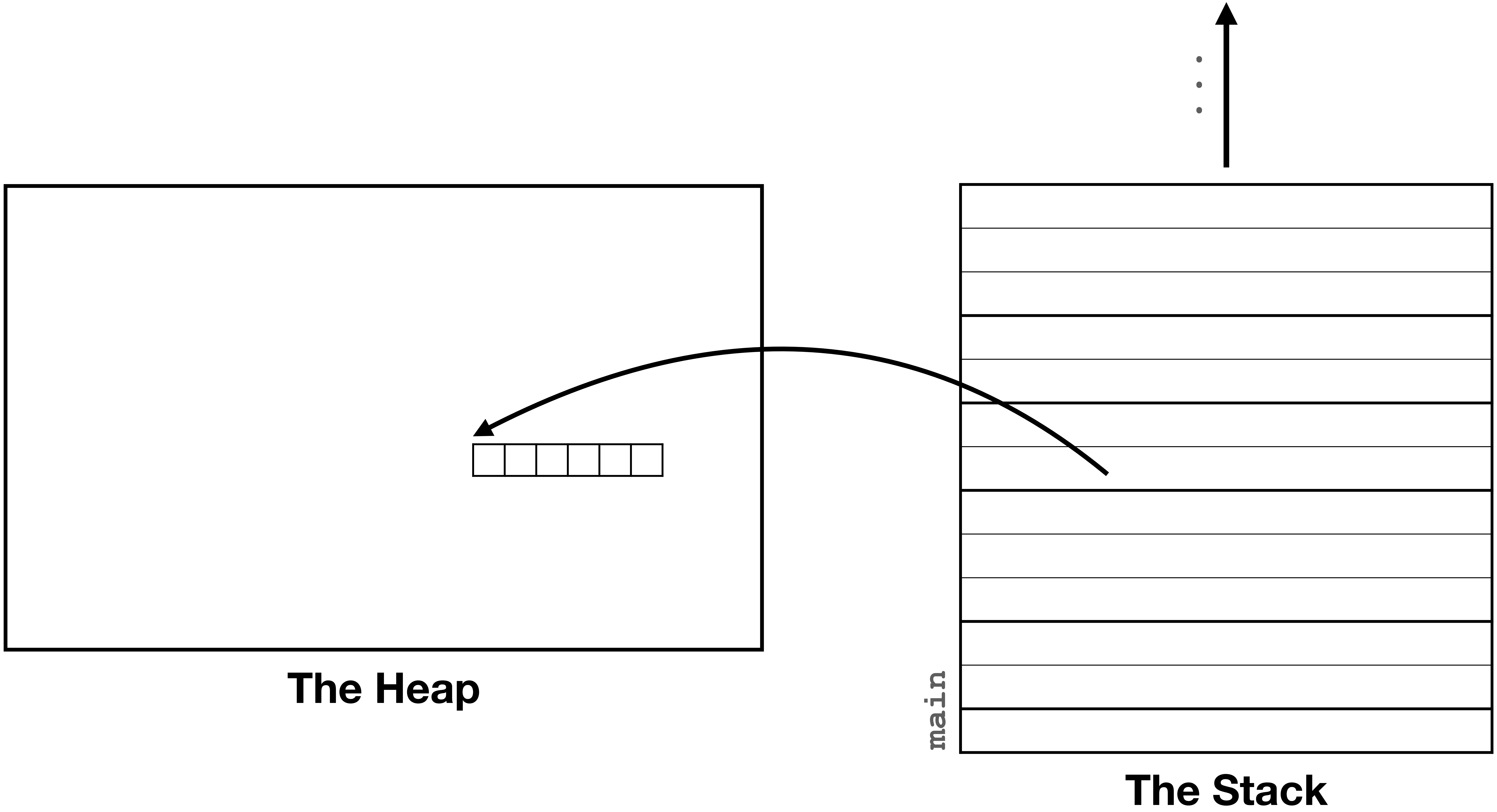
```
int freq[26];
```

```
int N_LETTERS = 26;
int freq[N_LETTERS];
```

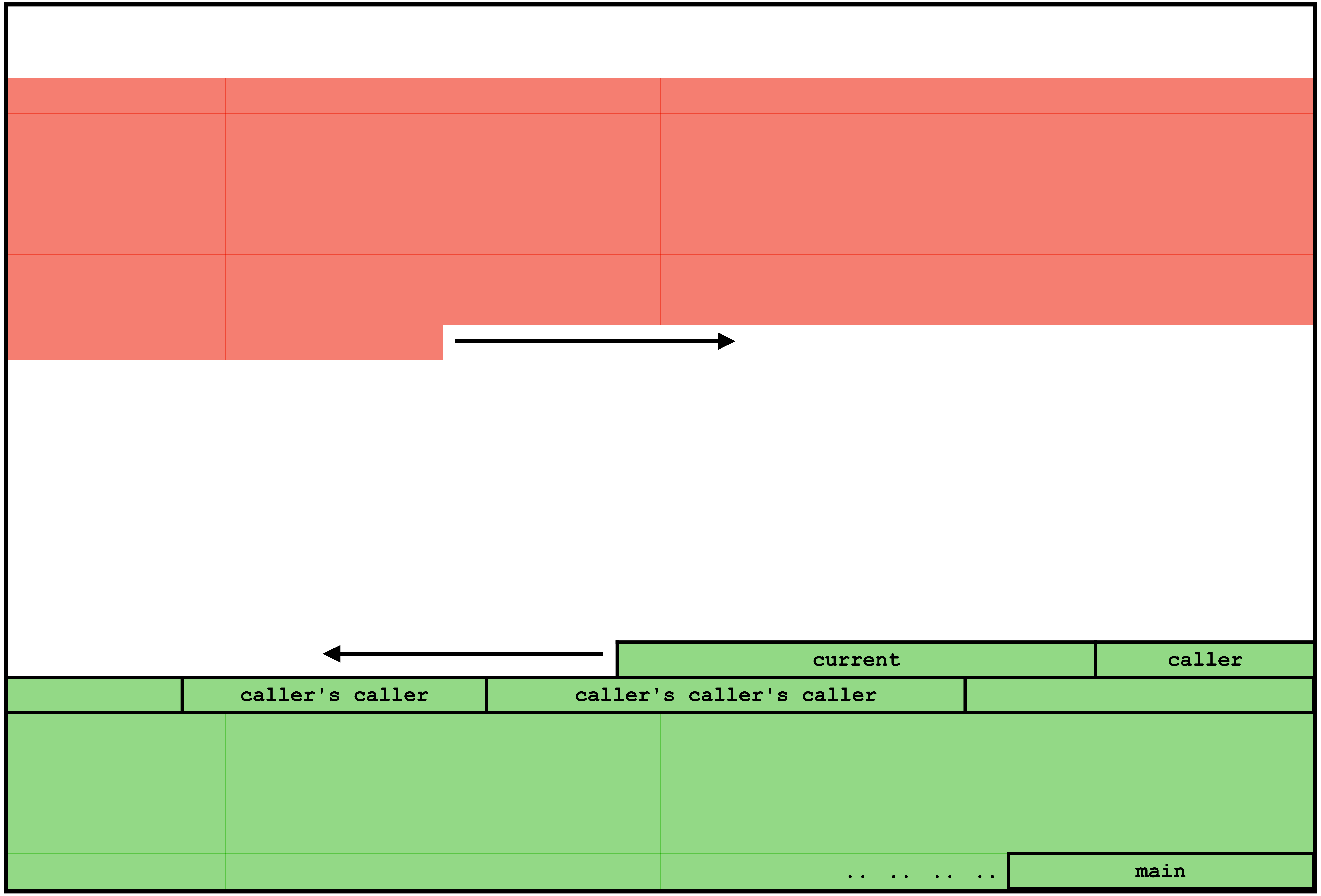
❌

- `clang` wants to know how big a frame is.
  - When calling the function, how much of stack does it need?
  - How much memory does `freq` use?
  - 26 * sizeof(int)
  - What if `N_LETTERS` is passed by user input
  - 🤷‍♂️

The Heap



The Heap


current
caller
caller's caller
caller's caller's caller
.. ..
main

The Heap

The Stack

- To request space on the stack, we *declare* a variable.
  - Compiler figures out the size according to the type.
- To release space on the stack, we do *nothing*.
  - Compiler recycles the space when we reach the end of the enclosing function.
- How about the heap?

The Heap

`malloc` and `free`

```
#include <stdlib.h>
int *numbers = malloc(104);

...

free(numbers);
```

`malloc(n)` requests `n` bytes from the heap, and it returns a pointer to the beginning of the space.

`free(ptr)` recycles the space pointed by `ptr`; malloc may now reuse the space for something else.

The Heap

`malloc` and `free`

```
#include <stdlib.h>
int *numbers = malloc(104);

...

free(numbers);

*numbers = 123;
```

`malloc(n)` requests `n` bytes from the heap, and it returns a pointer to the beginning of the space.

`free(ptr)` recycles the space pointed by `ptr`; malloc may now reuse the space for something else.

Big no-no! Because this address is probably used by something else, this *corrupts* the memory -- memory error.

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables
  - size: compiler calculates *before* running (static)
- Release memory:
  - do nothing
  - You can't forget to release

Heap

- Acquire memory:
  - `ptr = malloc(n)`
  - size: you provide *during* running (dynamic)
- Release memory:
  - `free(ptr)`
  - You can forget to release; *memory leak*

- Accessing released memory is bad; *memory error*

Pointers

How about returning an array?

DO NOT calculate the size by hand; use **sizeof**

```
int *make_10_of(int n)
{
        int *numbers = malloc(10 * sizeof(int));

        for (int i = 0; i < 10; i++) {
                numbers[i] = n;
        }

        return numbers;
}


int main()
{
        int *ns = make_10_of(42);
        /* do something with ns */
        free(ns);

        return 0;
}
```

Forgetting to free here causes memory leak.

Intermezzo: `struct`

Making your own types

- Data placed in memory can be: `char`, `short`, `int`, `long`, `float`, `double`
- What if you want to save something other than a number?
  - Student?
  - Course?
  - House?
  - ...
- You use numbers to represent them; you *digitize* them.
- In C, we can use *structures* to bundle data together.

Intermezzo: `struct`

Making your own types

- Data placed in memory can be: `char, short, int, long, float, double`
- What if you want to save something other than a number?
  - Student?
  - Course?
  - House?
  - ...
- You use numbers to represent them; you *digitize* them.
- In C, we can use *structures* to bundle data together.

Intermezzo: `struct`

Syntax

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};
```

Don't forget the ;

```
int main(void)
{
        struct student john;
```

Don't forget the word struct

```
        strcpy(john.first_name, "John");
        strcpy(john.last_name, "Doe");
        john.gpa = 3.0;
```

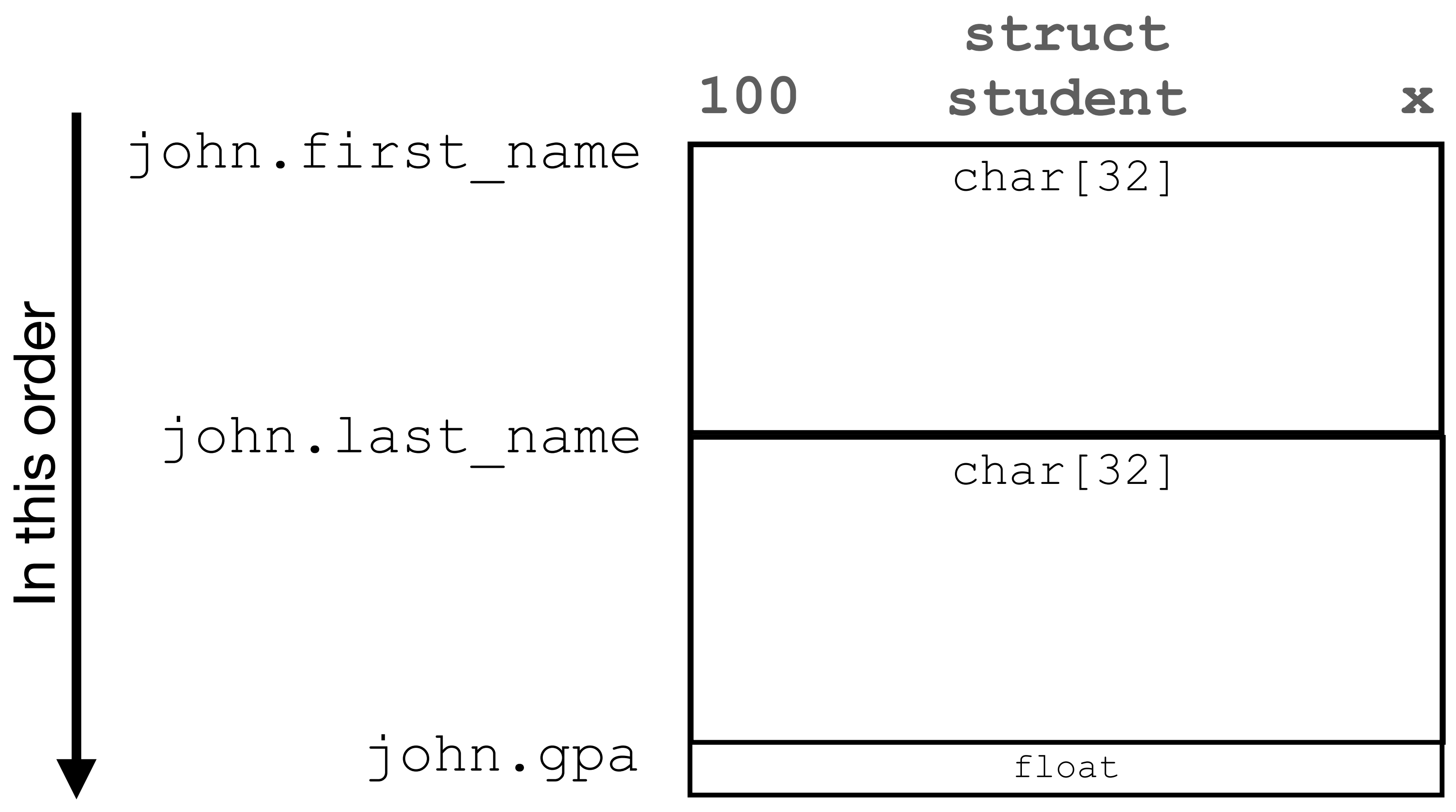
Assignment (=) doesn't work with arrays

```
        printf("%s %s: %.2f\n", john.first_name, john.last_name, john.gpa);

        return 0;
}
```

Intermezzo: `struct`

Syntax

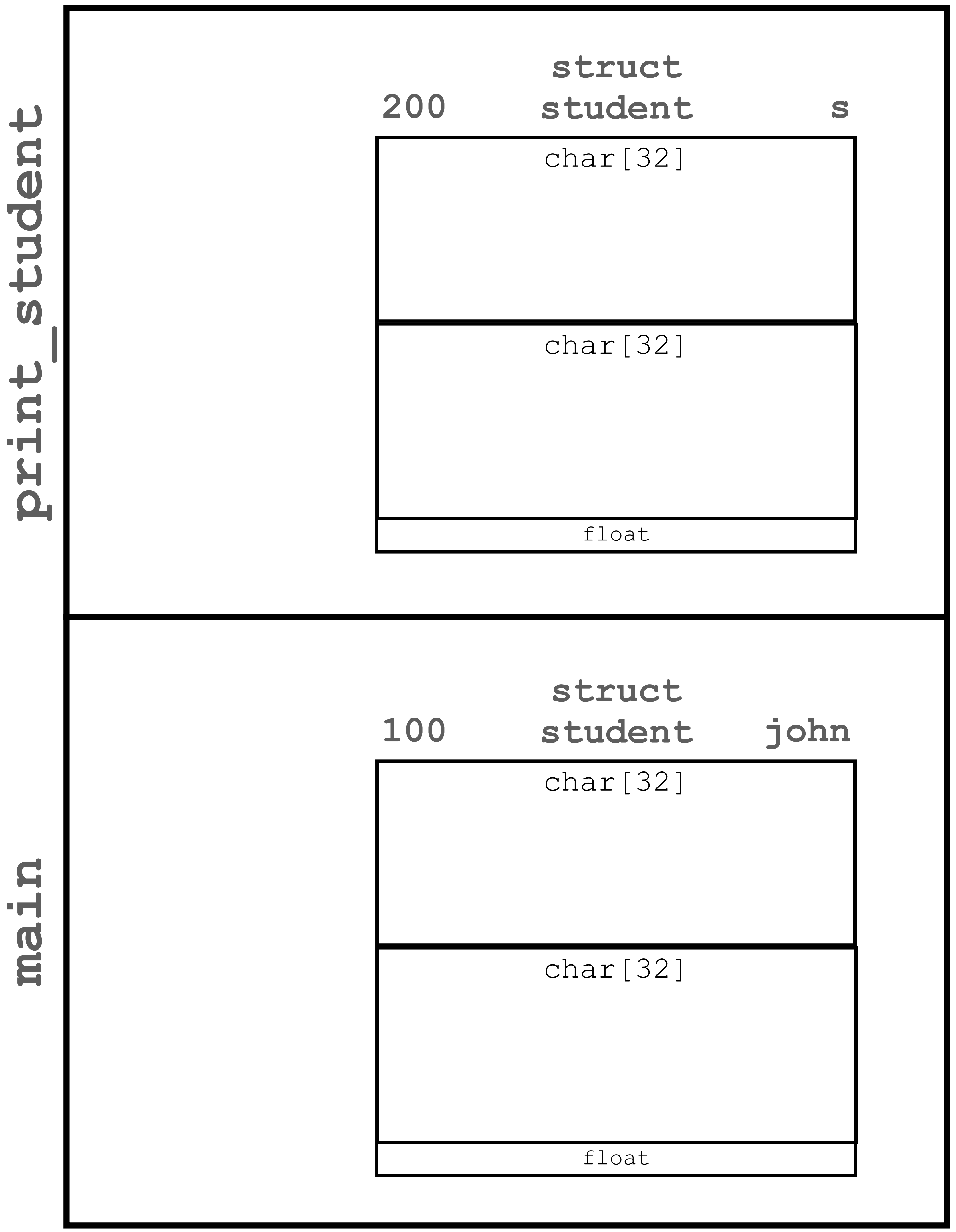
```c
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

int main(void)
{
        struct student john;

        strcpy(john.first_name, "John");
        strcpy(john.last_name, "Doe");
        john.gpa = 3.0;

        printf("%s %s: %.2f\n", john.first_name, john.last_name, john.gpa);

        return 0;
}
```



Intermezzo: struct

Structures are passed by value

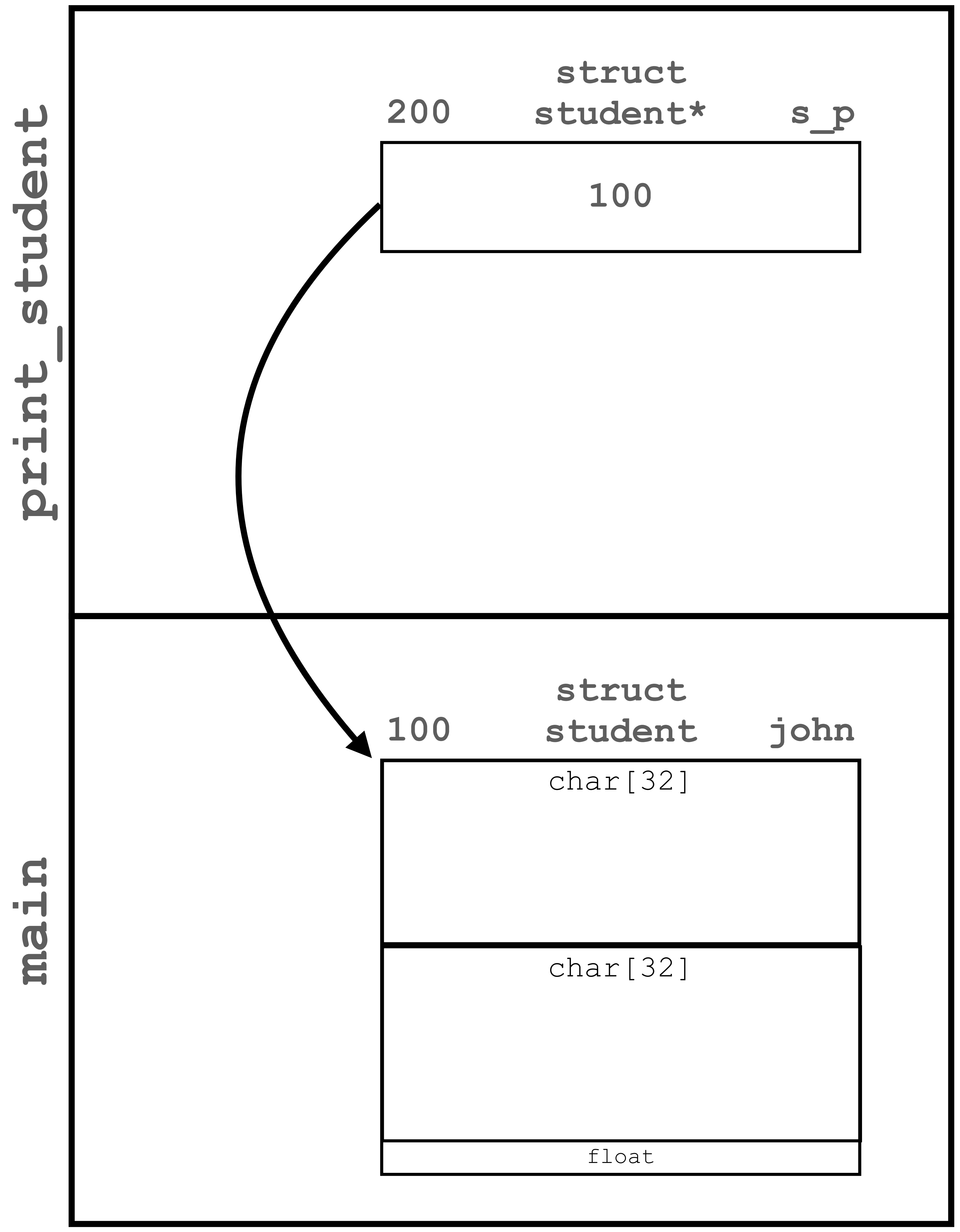
```c
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

void print_student(struct student s)
{
        printf("%s %s: %.2f\n", s.first_name, s.last_name, s.gpa);
}


int main(void)
{
        struct student john;
        ...
        print_student(john);

        return 0;
}
```



Intermezzo: struct

Structures are passed by value

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

void print_student(struct student *s_p)
{
        printf("%s %s: %.2f\n", (*s_p).first_name,
                                (*s_p).last_name,
                                (*s_p).gpa);
}

int main(void)
{
        struct student john;
        ...
        print_student(&john);

        return 0;
}
```



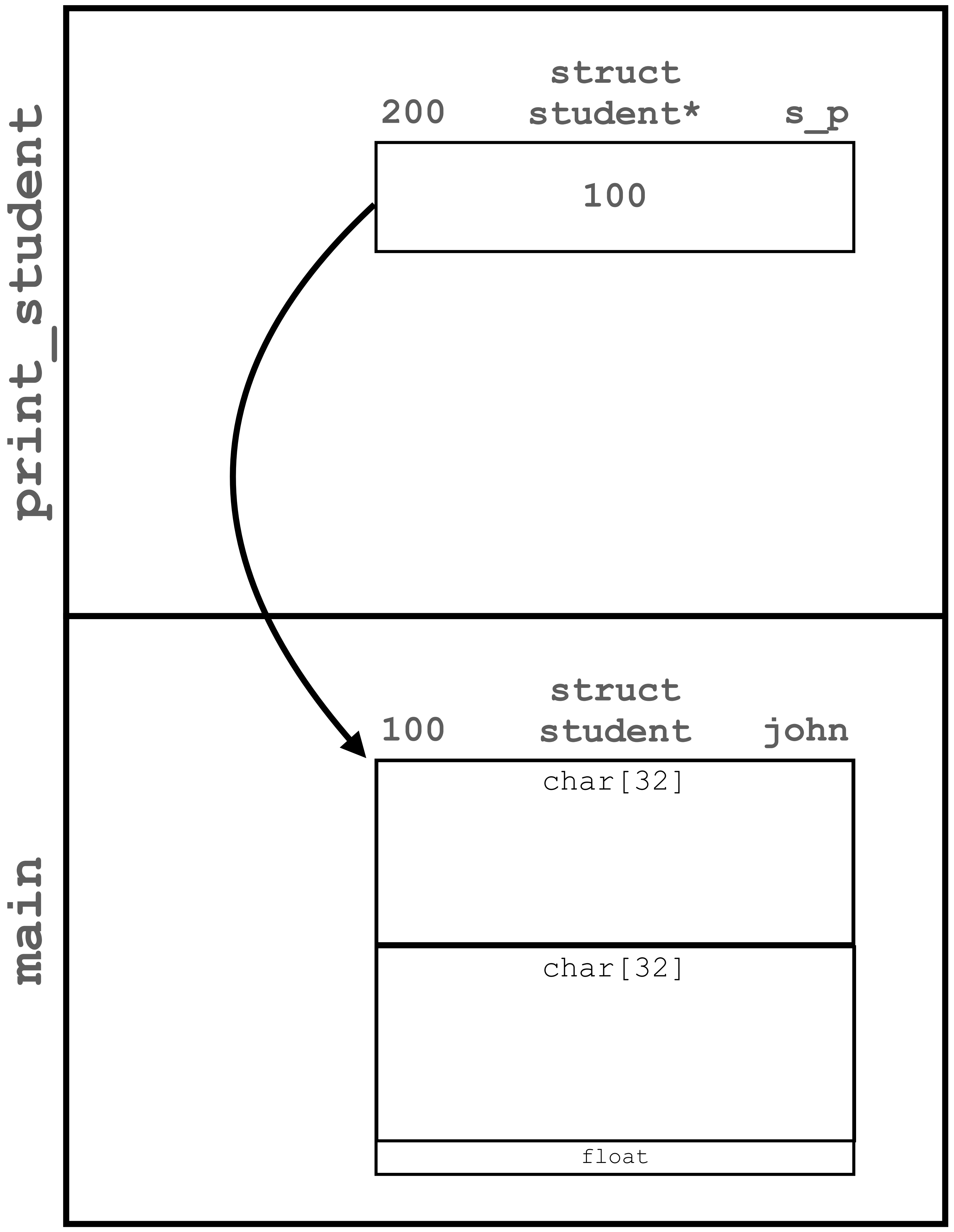
Intermezzo: `struct`

Structures are passed by value

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

void print_student(struct student *s_p)
{
        printf("%s %s: %.2f\n", s_p->first_name,
                                s_p->last_name,
                                s_p->gpa);
}

int main(void)
{
        struct student john;
        ...
        print_student(&john);

        return 0;
}
```

s_p->gpa is a shorthand for (*s_p).gpa

Btw, *s_p.gpa is read as *(s_p.gpa), which is an error

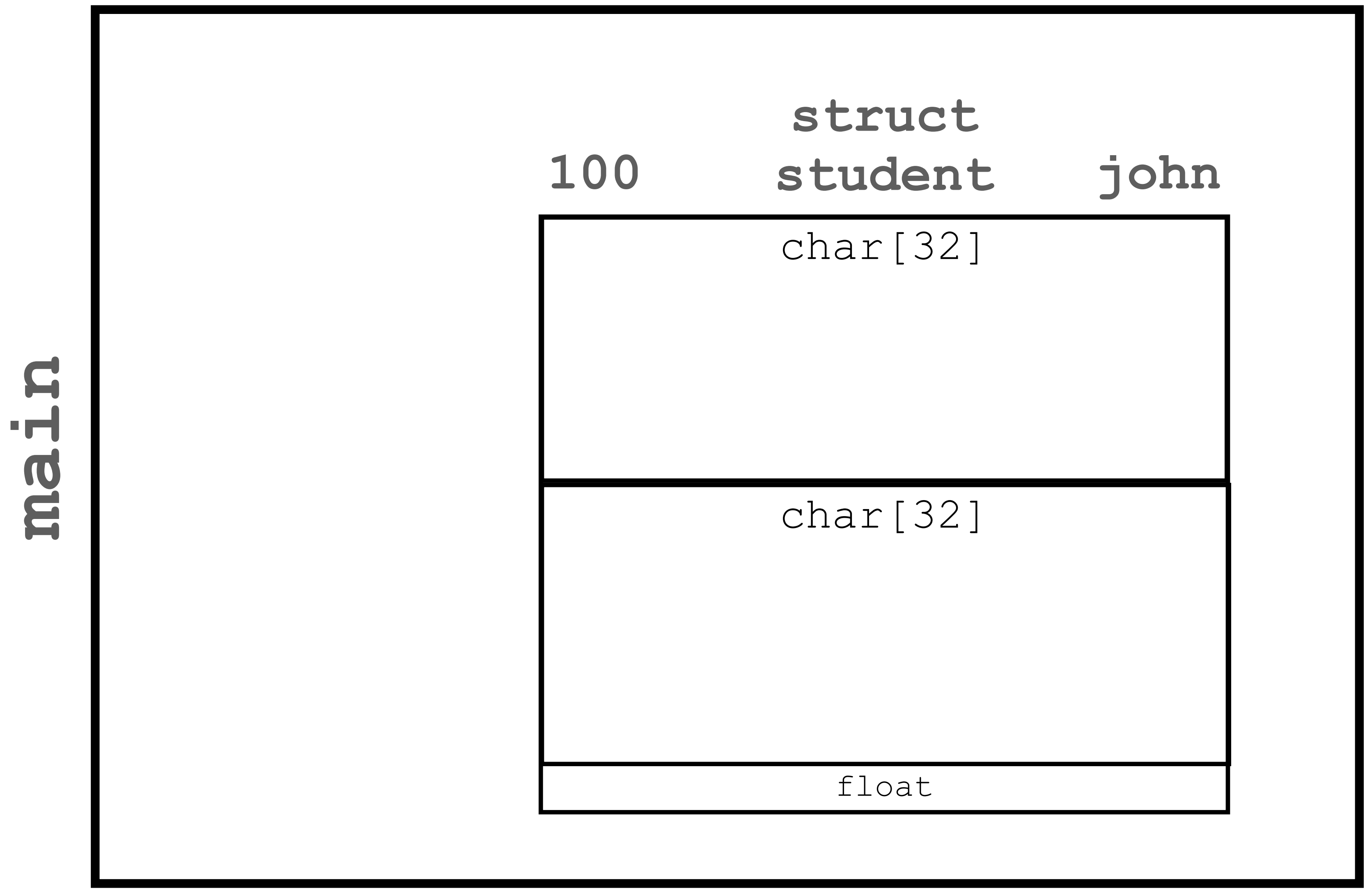


Pointer Hazards

Be aware of shallow copy

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

void print_student(struct student *s_p)
{
        printf("%s %s: %.2f\n", s_p->first_name,
                                s_p->last_name,
                                s_p->gpa);
}

int main(void)
{
        struct student john;
        ...
        print_student(&john);

        return 0;
}
```

Say, you want to remove the character limit on names

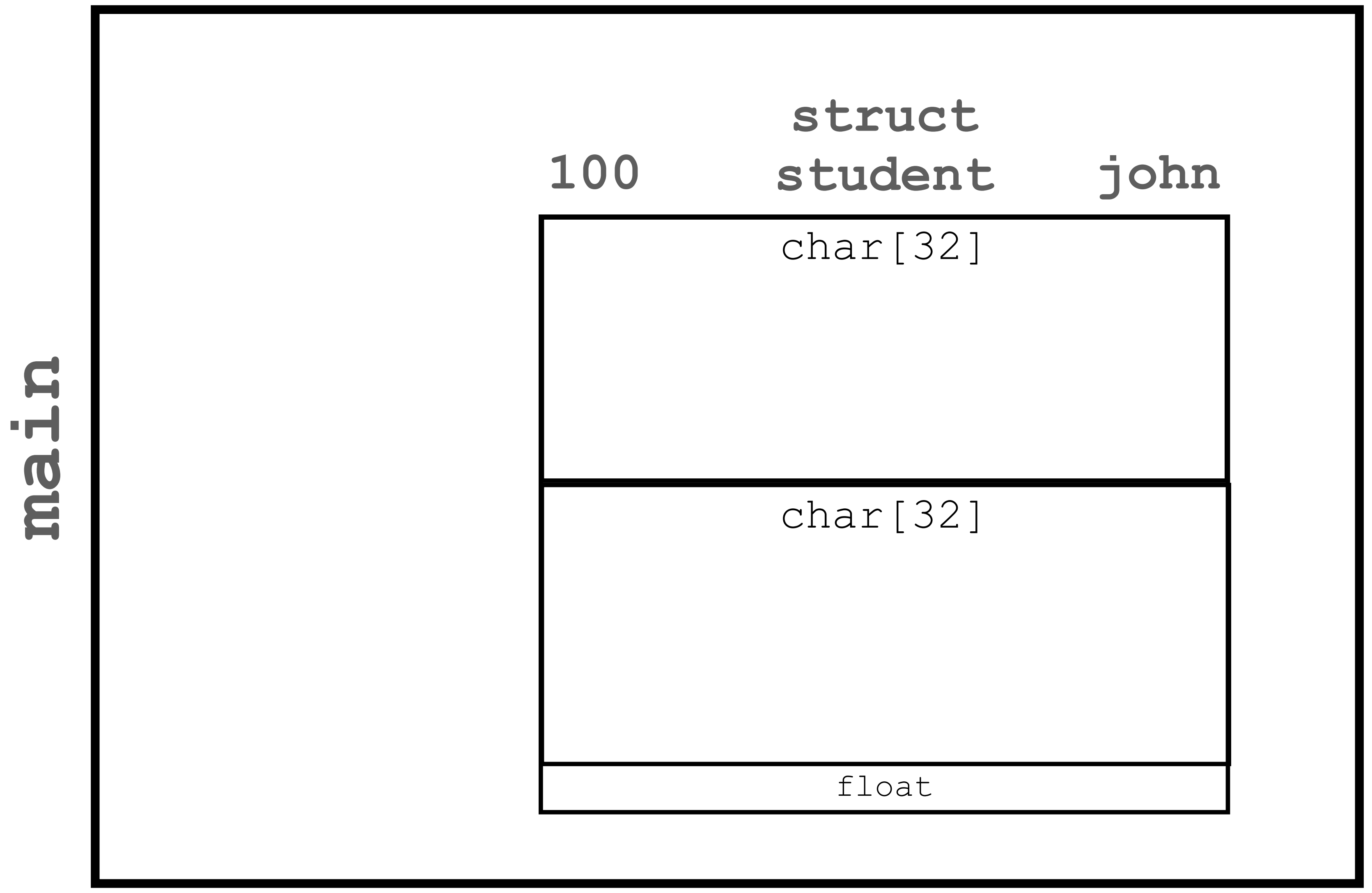


Pointer Hazards

Be aware of shallow copy

```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

void print_student(struct student *s_p)
{
        printf("%s %s: %.2f\n", s_p->first_name,
                                s_p->last_name,
                                s_p->gpa);
}

int main(void)
{
        struct student john;
        ...
        print_student(&john);

        return 0;
}
```

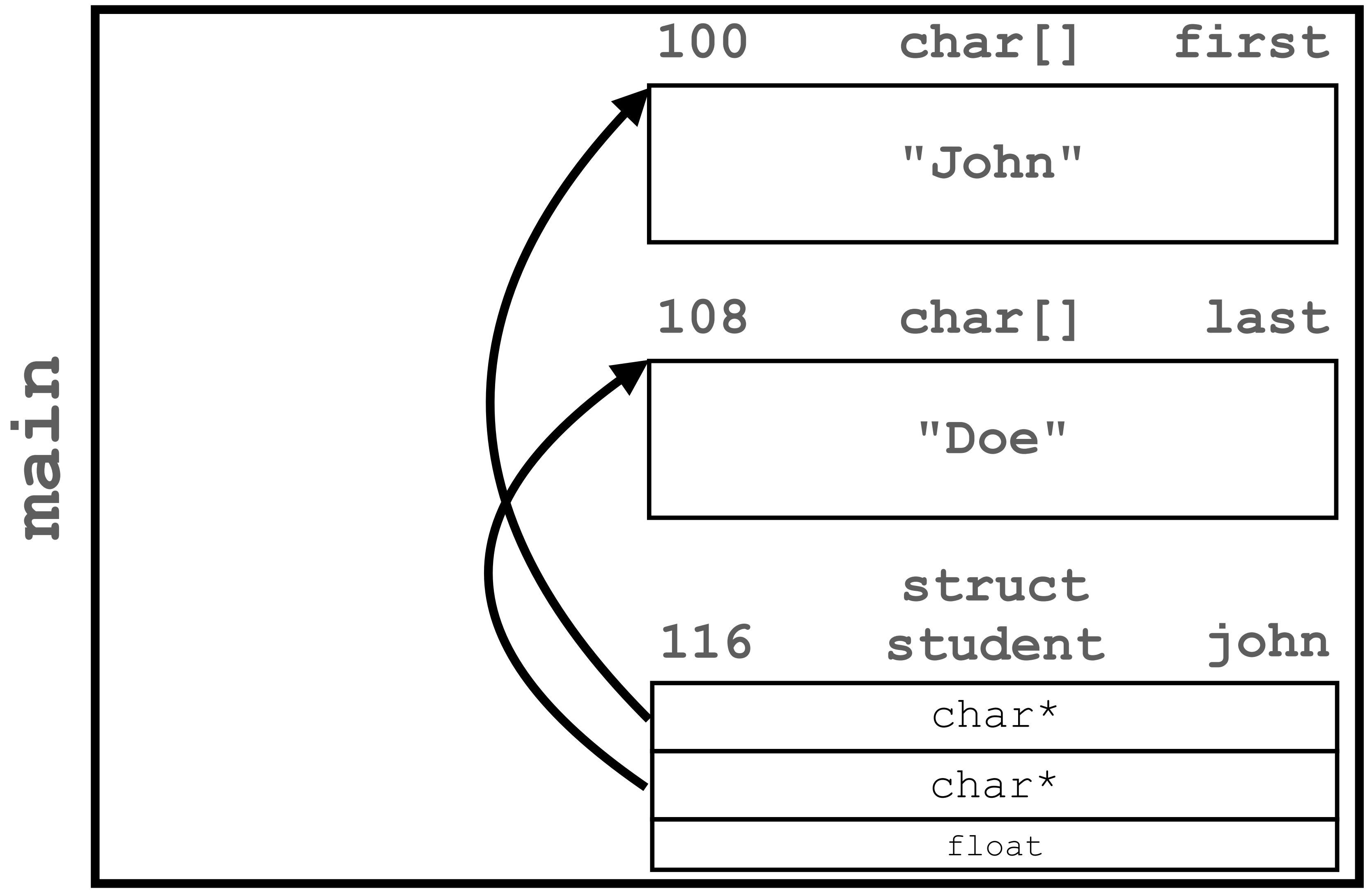
Say, you want to remove the character limit on names



Pointer Hazards

Be aware of shallow copy

```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

void print_student(struct student *s_p)
{
        ...
}

int main(void)
{
        char first_name[] = "John";
        char last_name[] = "Doe";

        struct student john;
        john.first_name = first_name;
        john.last_name = last_name;
        john.gpa = 3.0;

        return 0;
}
```

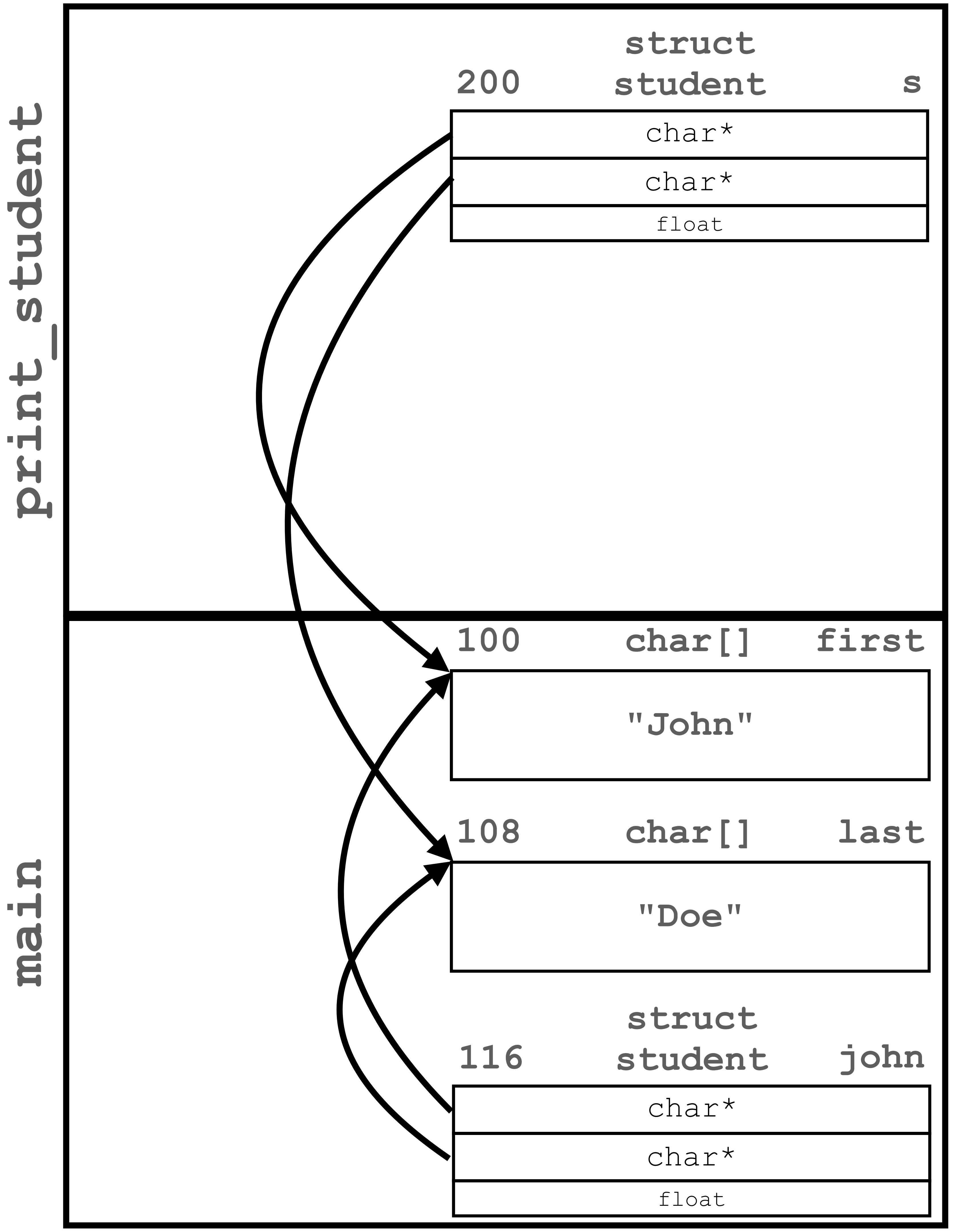


Pointer Hazards

Be aware of shallow copy

```c
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

void print_student(struct student s)
{
        ...
}

int main(void)
{
        char first_name[] = "John";
        char last_name[] = "Doe";

        struct student john;
        john.first_name = first_name;
        john.last_name = last_name;
        john.gpa = 3.0;

        return 0;
}
```

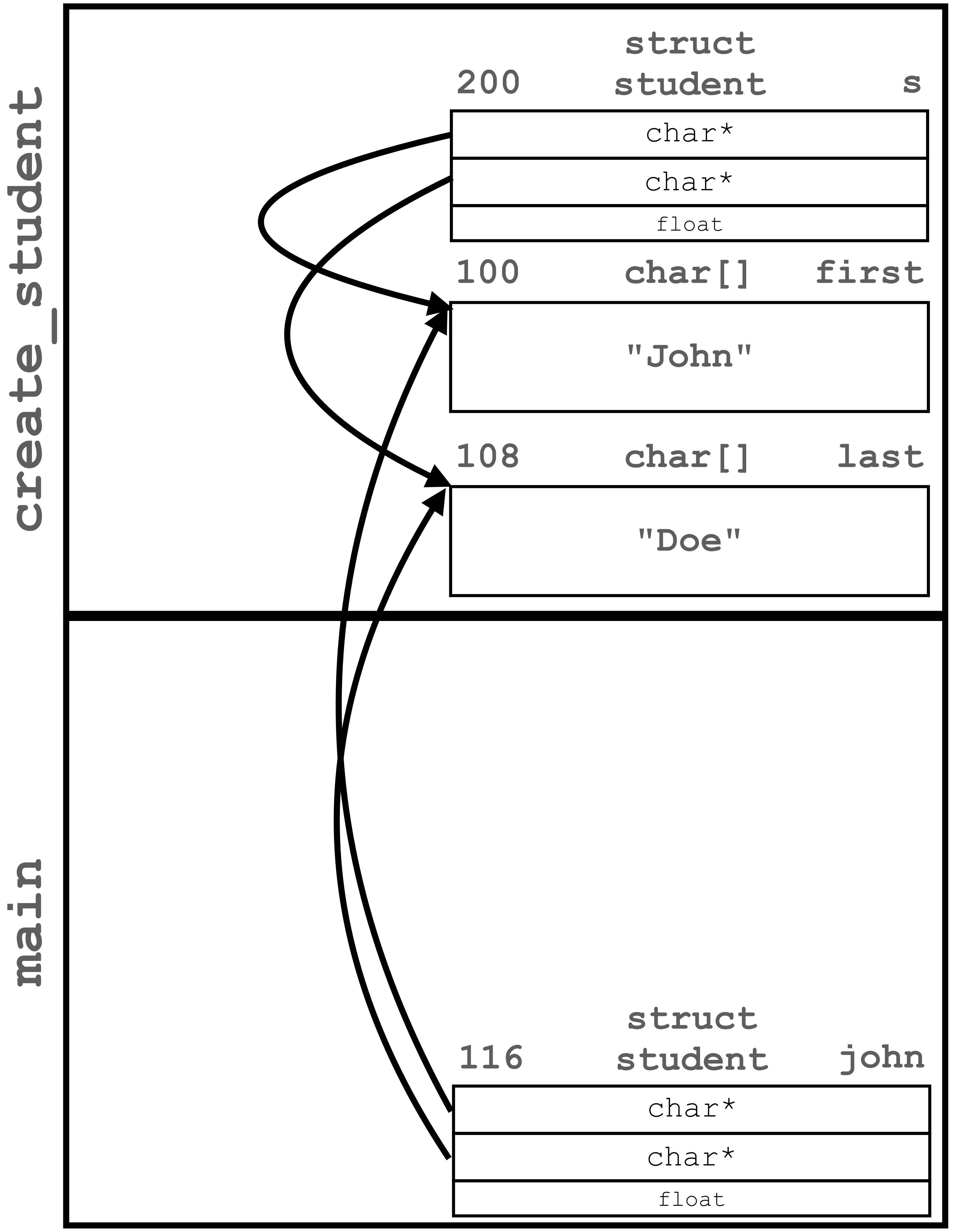
If you change s.first_name here, first_name and john.first_name are also changed in main.



Pointer Hazards

Be aware of dangling pointer (again)

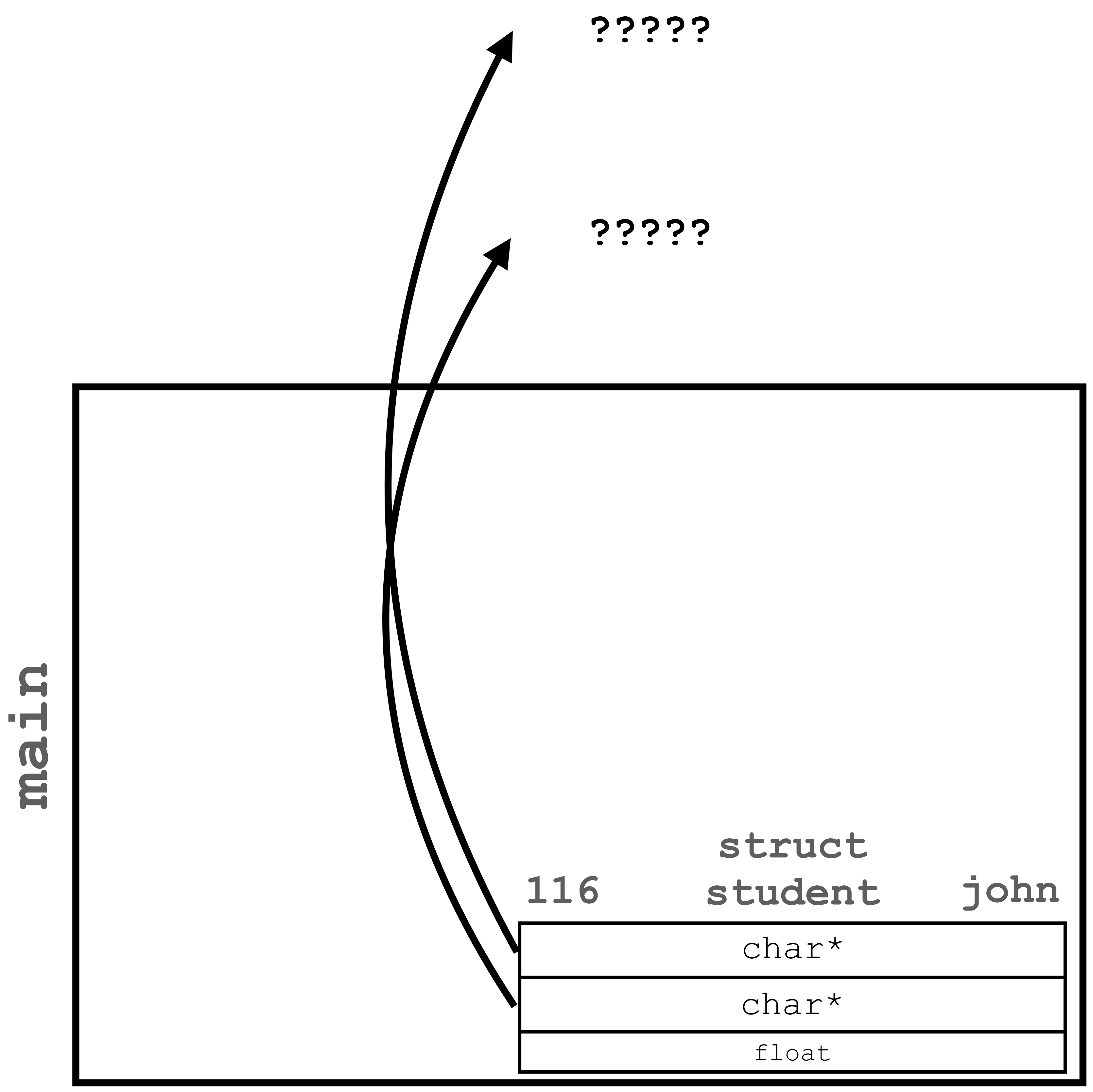
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

struct student create_student(void)
{
        char first_name[] = "John";
        char last_name[] = "Doe";

        struct student john;
        john.first_name = first_name;
        john.last_name = last_name;
        john.gpa = 3.0;
        return john;
}

int main(void)
{
        struct student john = create_student();

        return 0;
}
```



Pointer Hazards

Be aware of dangling pointer (again)

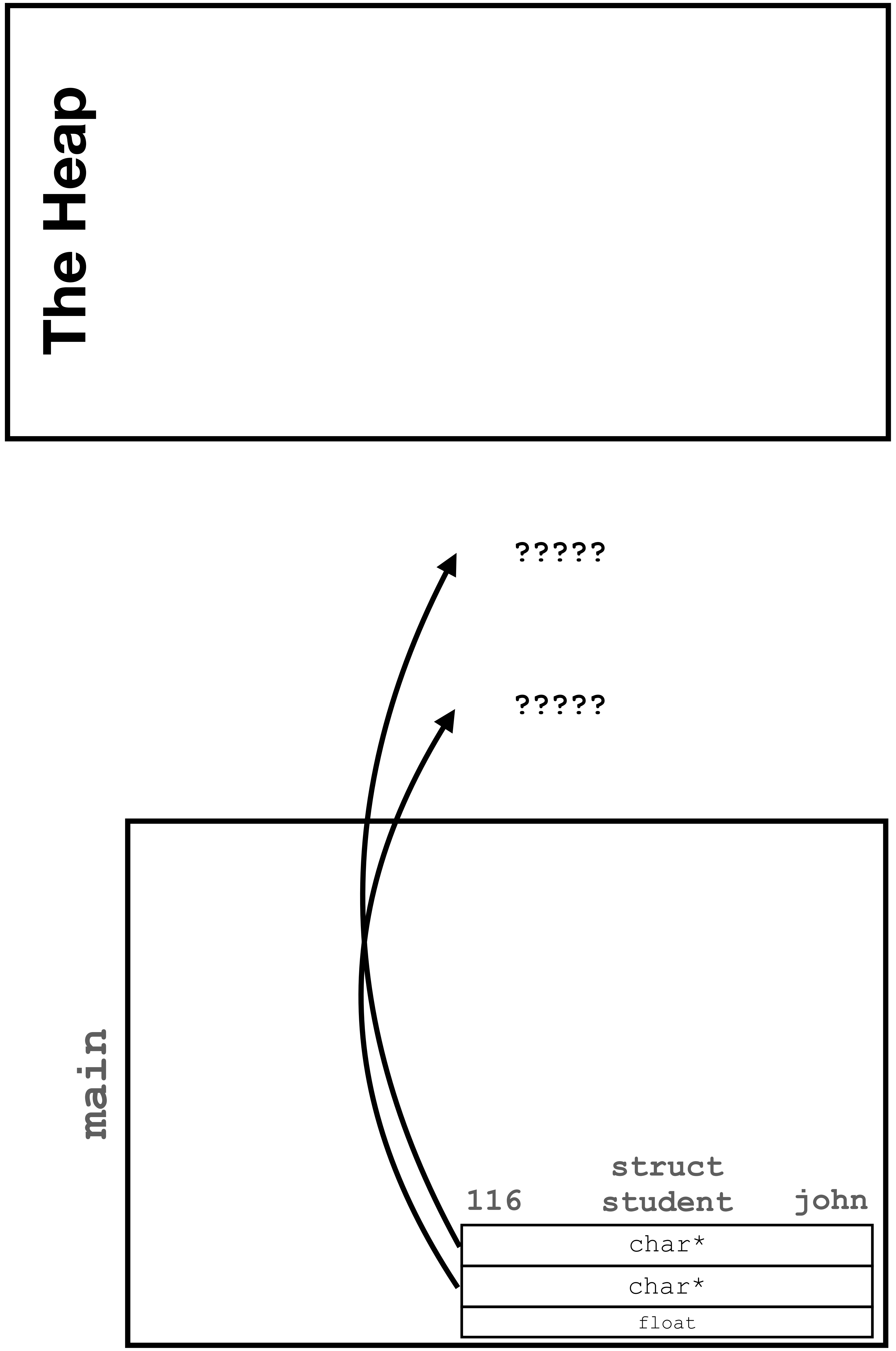
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

struct student create_student(void)
{
        char first_name[] = "John";
        char last_name[] = "Doe";

        struct student john;
        john.first_name = first_name;
        john.last_name = last_name;
        john.gpa = 3.0;
        return john;
}

int main(void)
{
        struct student john = create_student();

        return 0;
}
```



Pointer Hazards

Be aware of dangling pointer (again)

```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

struct student create_student(void)
{
        char first_name[] = "John";
        char last_name[] = "Doe";

        struct student john;
        john.first_name = first_name;
        john.last_name = last_name;
        john.gpa = 3.0;
        return john;
}

int main(void)
{
        struct student john = create_student();

        return 0;
}
```



Pointer Hazards

Be aware of dangling pointer (again)

The Heap

```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

struct student create_student(void)
{
        struct student john;
        john.first_name = strdup("John");
        john.last_name = strdup("Doe");
        john.gpa = 3.0;
        return john;
}

int main(void)
{
        struct student john = create_student();

        return 0;
}
```

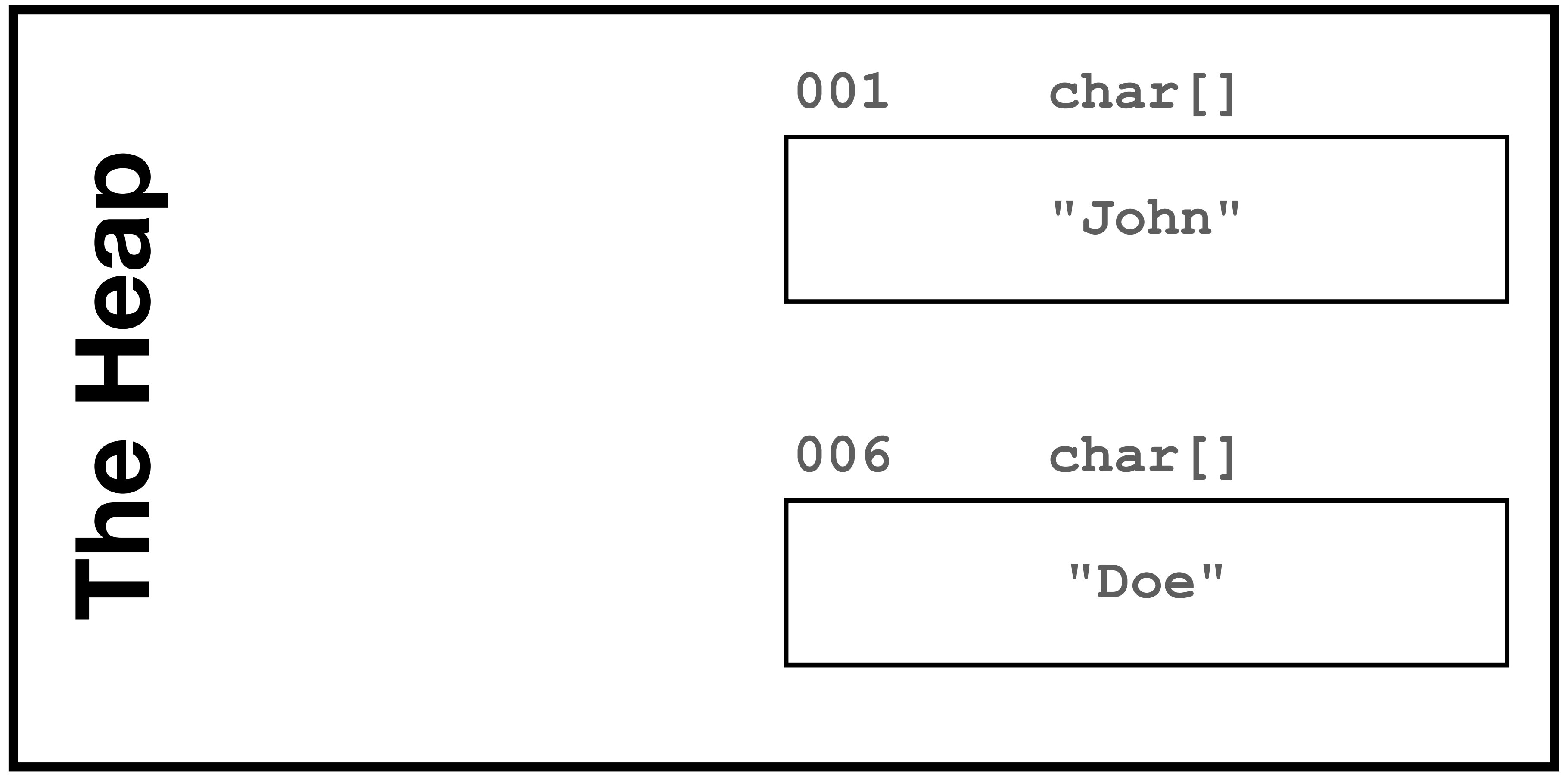
strdup in <string.h> duplicates the string on the heap. i.e. strdup calls malloc, and then copy the string to the malloc'ed space.
It's a very handy function.

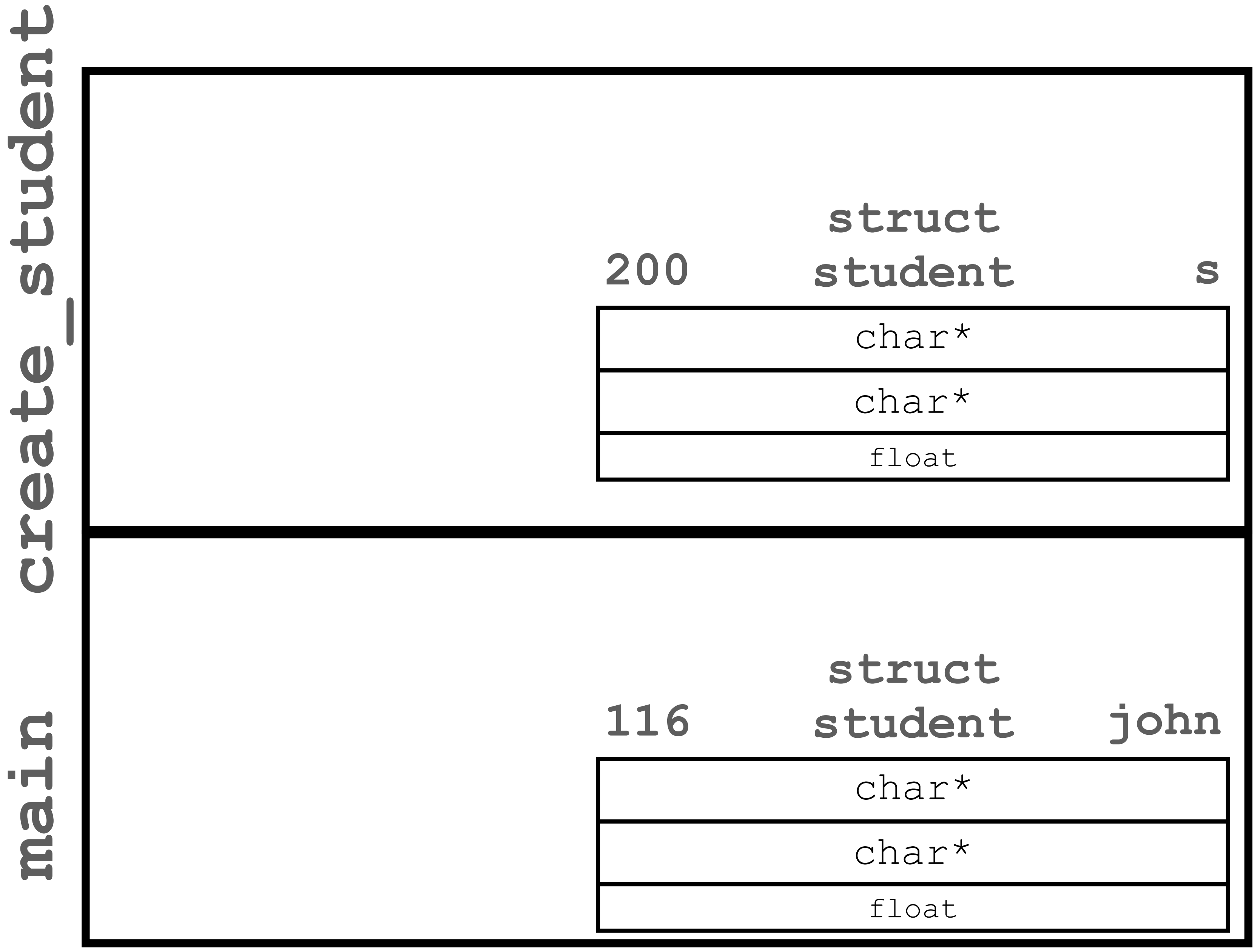
Pointer Hazards

Be aware of dangling pointer (again)

```c
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

struct student create_student(void)
{
        struct student john;
        john.first_name = strdup("John");
        john.last_name = strdup("Doe");
        john.gpa = 3.0;
        return john;
}

int main(void)
{
        struct student john = create_student();

        return 0;
}
```

strdup in <string.h> duplicates the string on the heap. i.e. strdup calls malloc, and then copy the string to the malloc'ed space.
It's a very handy function.



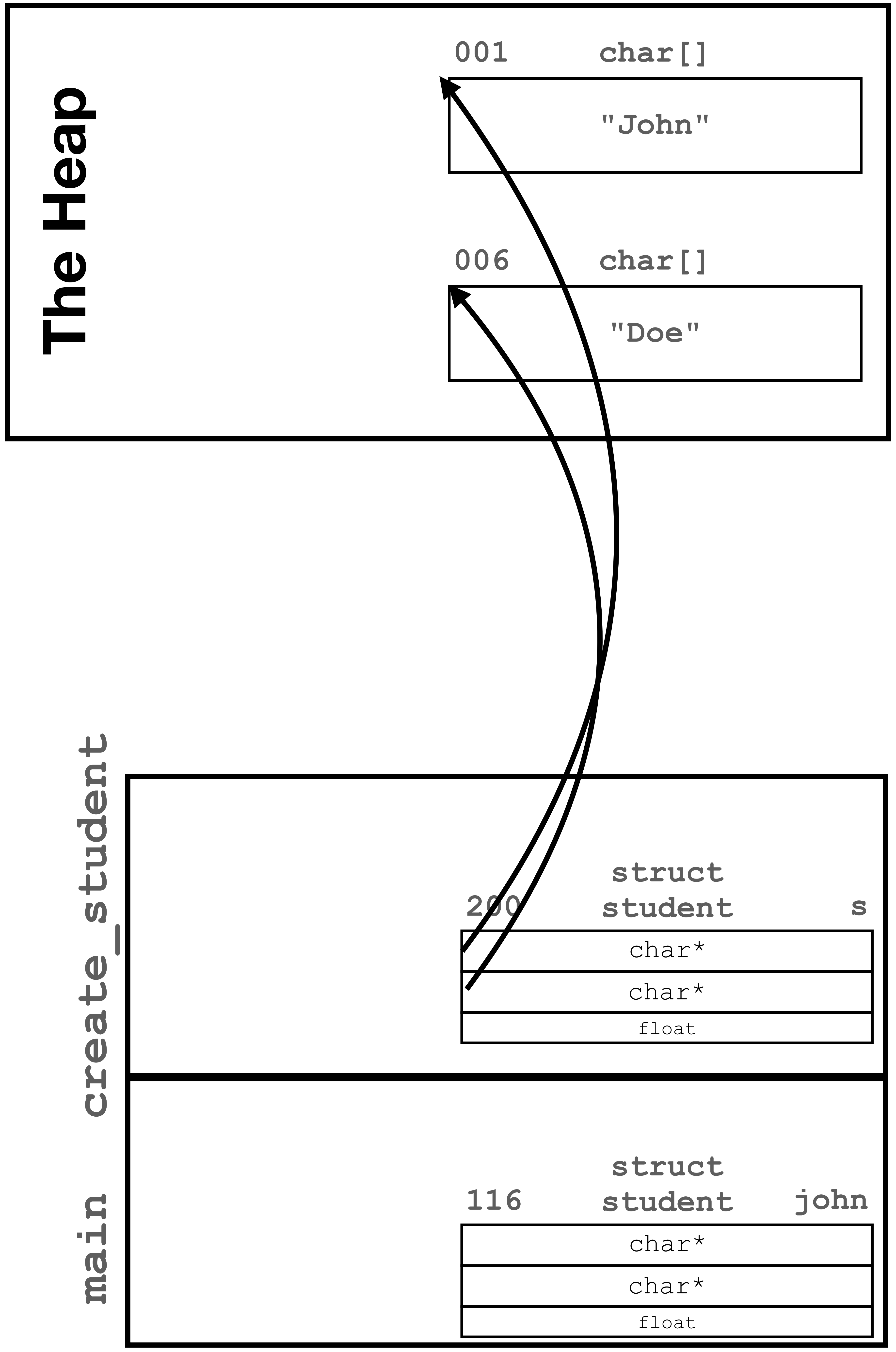


Pointer Hazards

Be aware of dangling pointer (again)

```c
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

struct student create_student(void)
{
        struct student john;
        john.first_name = strdup("John");
        john.last_name = strdup("Doe");
        john.gpa = 3.0;
        return john;
}

int main(void)
{
        struct student john = create_student();

        return 0;
}
```

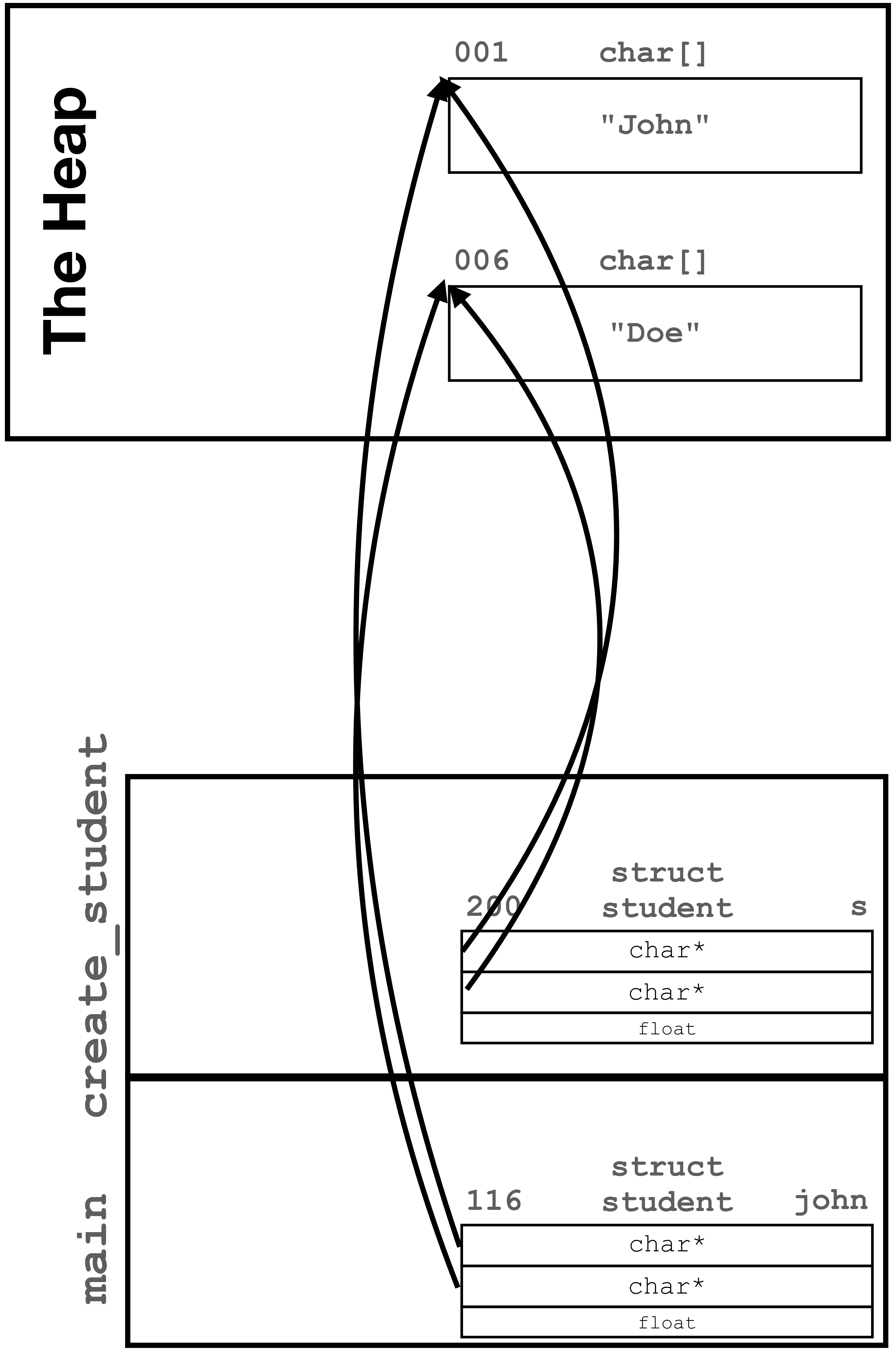
strdup in <string.h> duplicates the string on the heap. i.e. strdup calls malloc, and then copy the string to the malloc'ed space.
It's a very handy function.



Pointer Hazards

Be aware of dangling pointer (again)

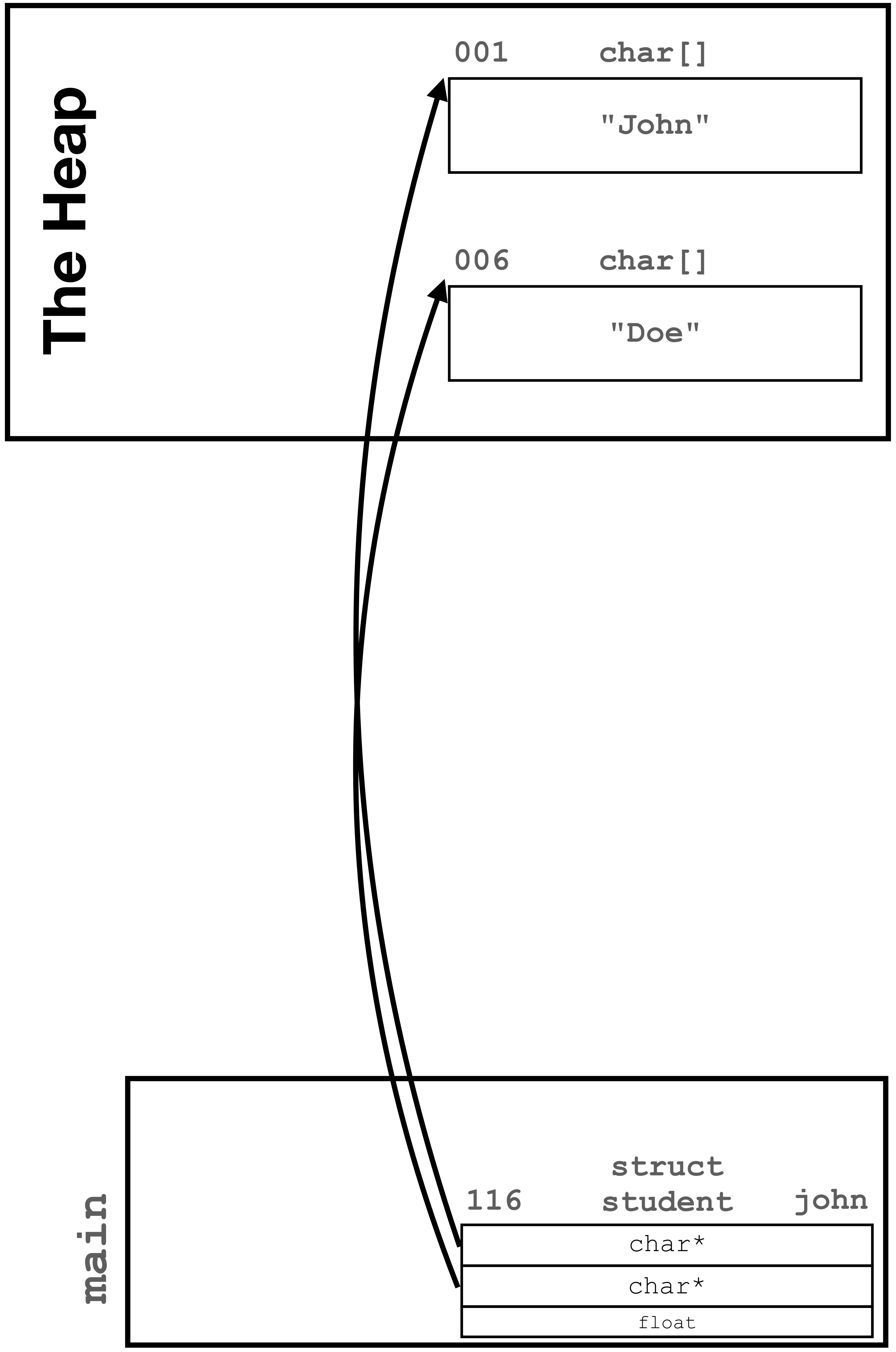
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

struct student create_student(void)
{
        struct student john;
        john.first_name = strdup("John");
        john.last_name = strdup("Doe");
        john.gpa = 3.0;
        return john;
}

int main(void)
{
        struct student john = create_student();

        return 0;
}
```



Pointer Hazards

Be aware of dangling pointer (again)

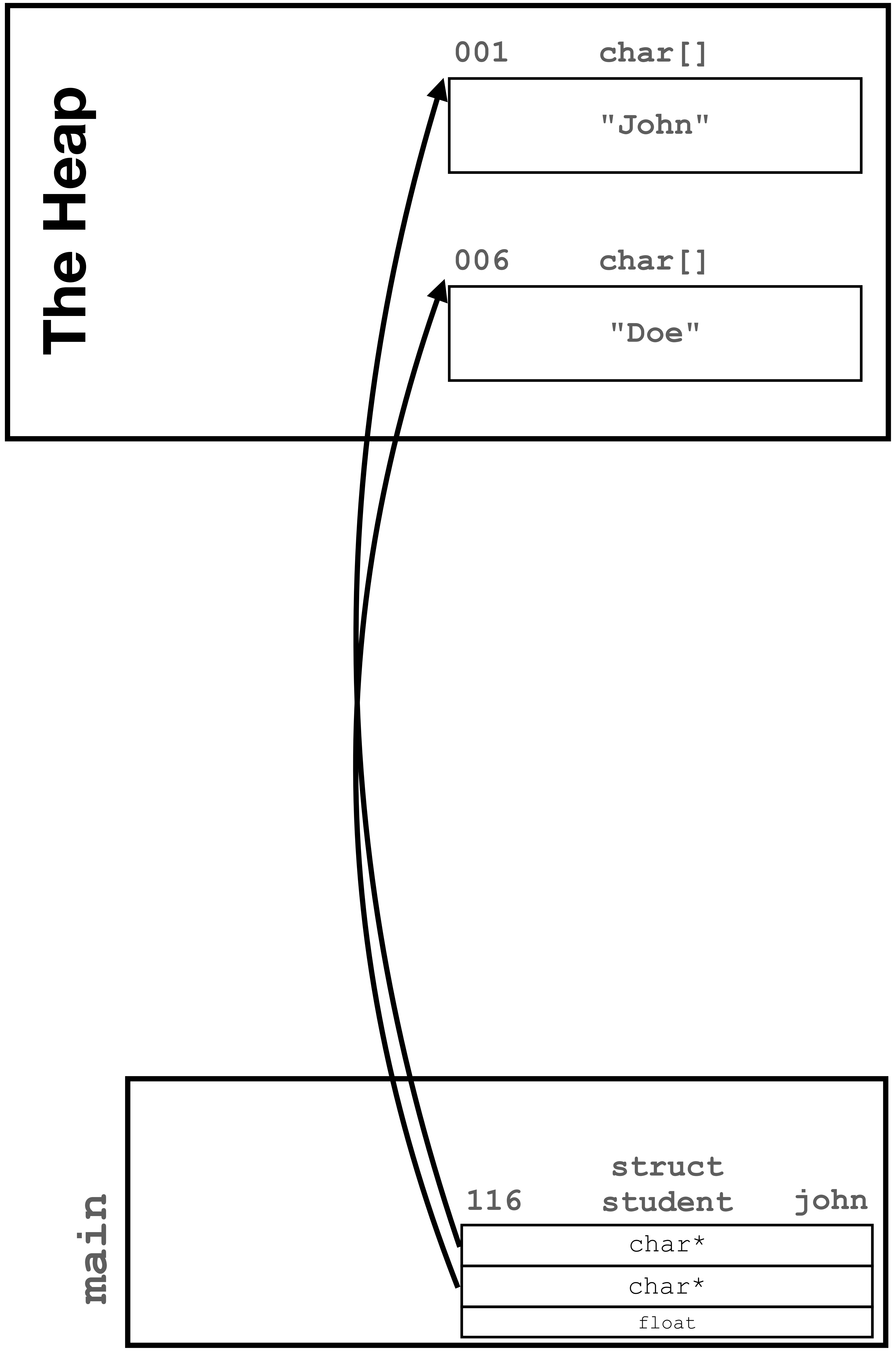
```c
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

struct student create_student(void)
{
        struct student john;
        john.first_name = strdup("John");
        john.last_name = strdup("Doe");
        john.gpa = 3.0;
        return john;
}

int main(void)
{
        struct student john = create_student();

        return 0;
}
```



Pointer Hazards

Be aware of memory leak

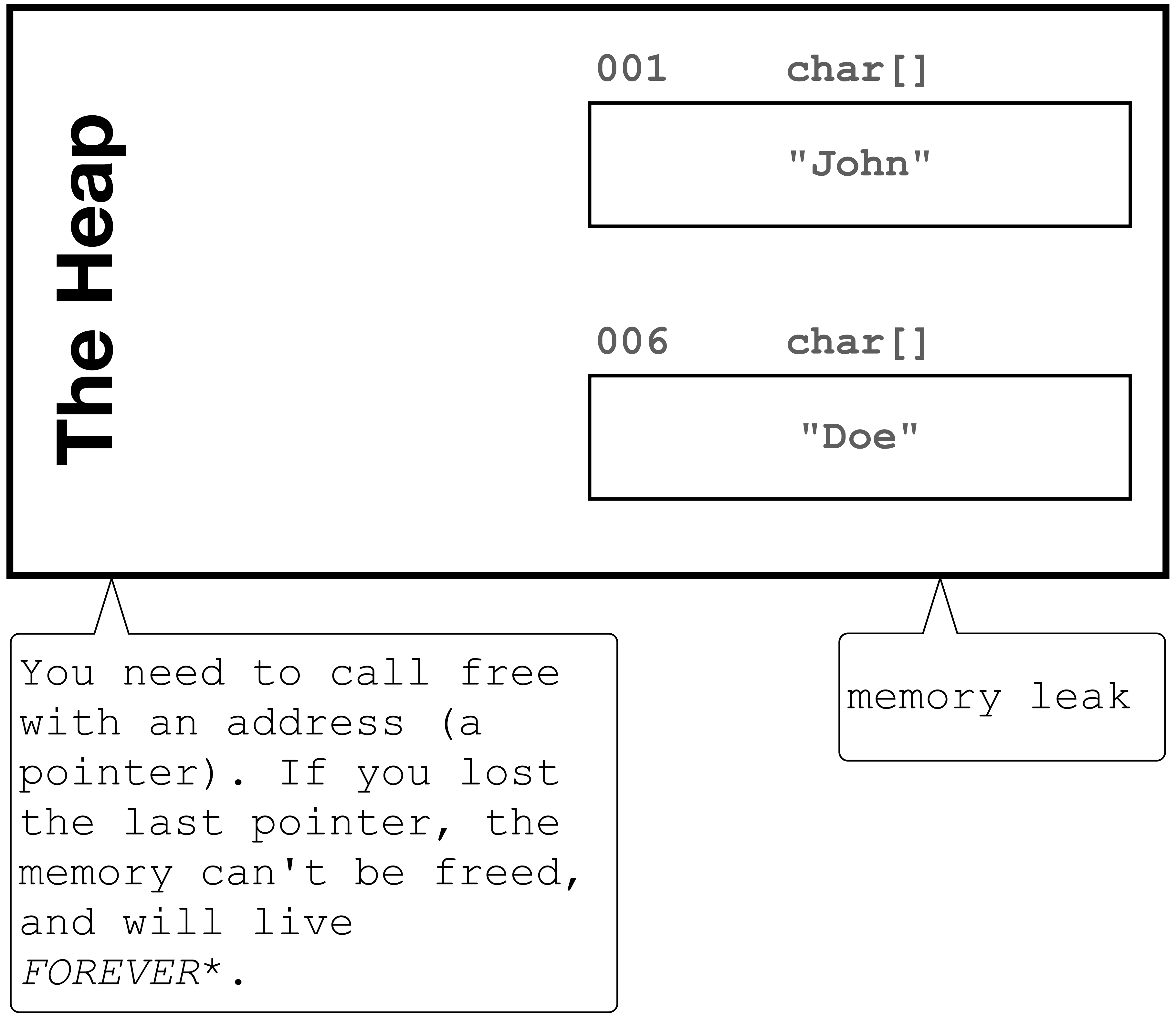
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

struct student create_student(void)
{
        struct student john;
        john.first_name = strdup("John");
        john.last_name = strdup("Doe");
        john.gpa = 3.0;
        return john;
}

int main(void)
{
        struct student john = create_student();

        return 0;
}
```



Pointer Hazards

Be aware of memory leak

```c
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

struct student create_student(void)
{
        struct student john;
        john.first_name = strdup("John");
        john.last_name = strdup("Doe");
        john.gpa = 3.0;
       return john;
}

int main(void)
{
        struct student john = create_student();

        return 0;
}
```

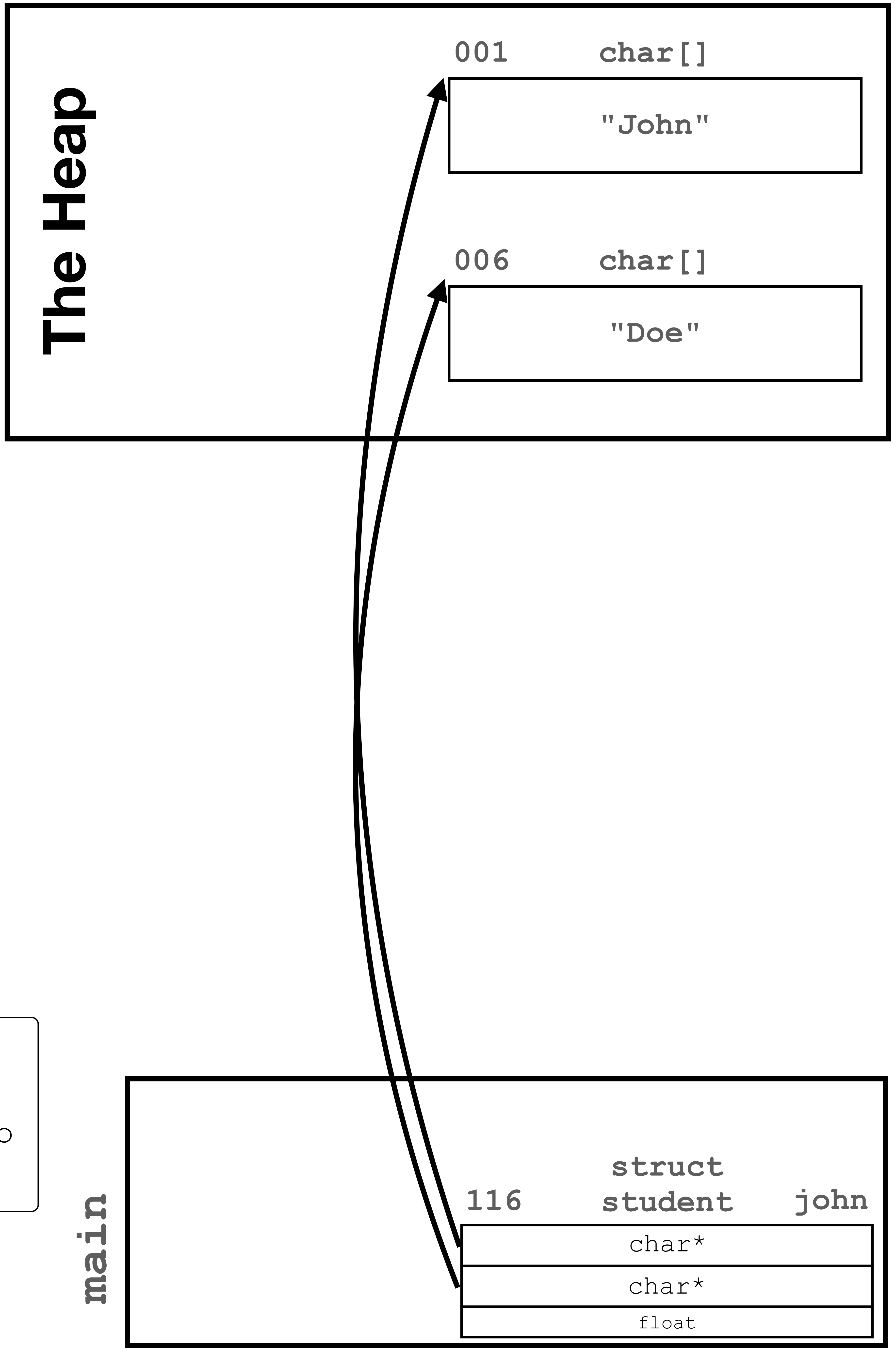


Pointer Hazards

Be aware of memory leak

```c
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

struct student create_student(void)
{
        struct student john;
        john.first_name = strdup("John");
        john.last_name = strdup("Doe");
        john.gpa = 3.0;
        return john;
}

int main(void)
{
        struct student john = create_student();
        free(john.first_name);
        free(john.last_name);
        return 0;
}
```

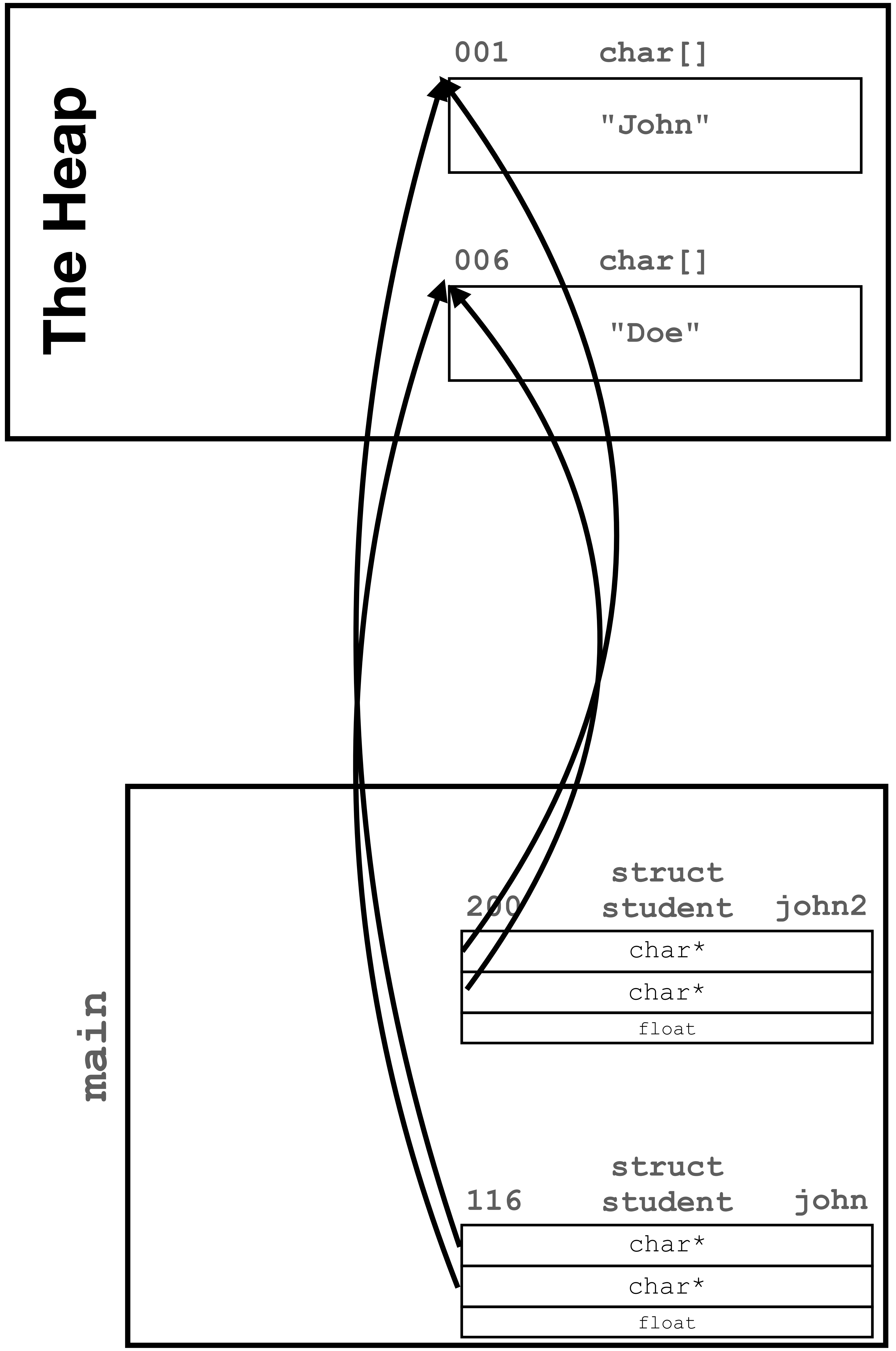
remember to call free when the last pointer to the heap data is about to go out of scope



Pointer Hazards

Be aware of double-free corruption

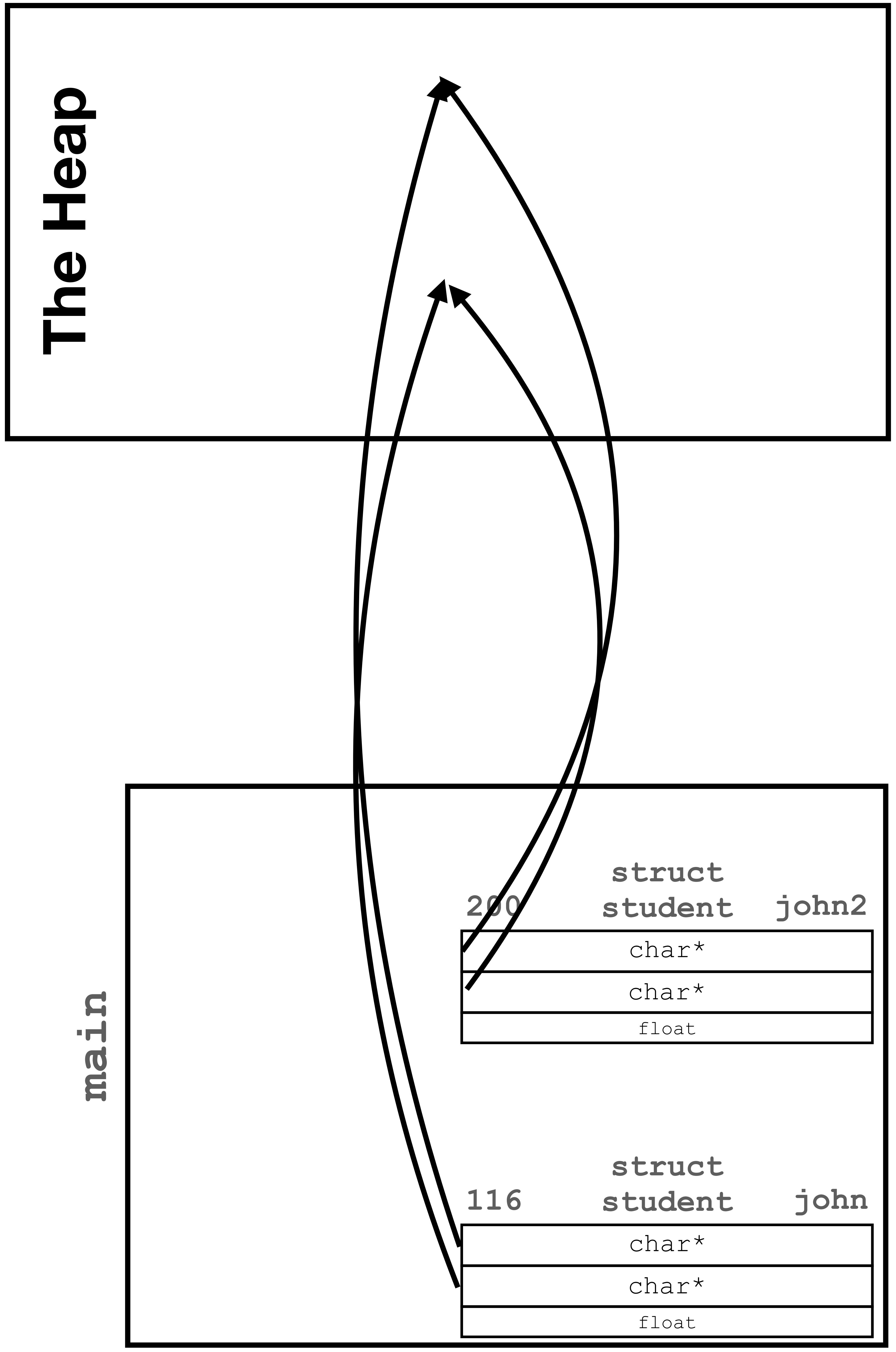
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

struct student create_student(void)
{
        ...
}

int main(void)
{
        struct student john = create_student();
        struct student john2 = john;
        return 0;
}
```



Pointer Hazards

Be aware of double-free corruption

```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

struct student create_student(void)
{
        ...
}

int main(void)
{
        struct student john = create_student();
        struct student john2 = john;
        free(john.first_name);
        free(john.last_name);

        return 0;
}
```

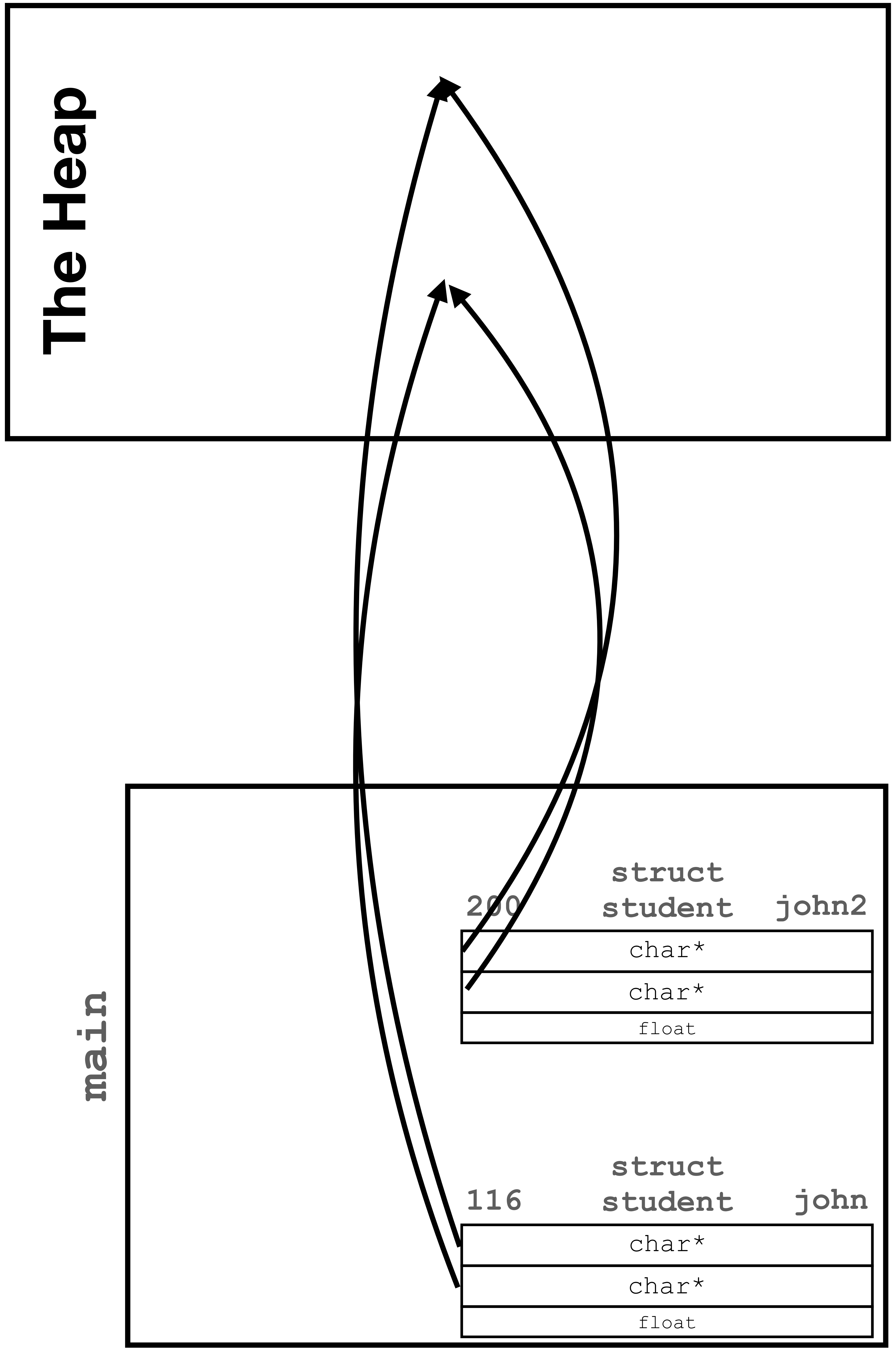


Pointer Hazards

Be aware of double-free corruption

```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

struct student create_student(void)
{
        ...
}

int main(void)
{
        struct student john = create_student();
        struct student john2 = john;
        free(john.first_name);
        free(john.last_name);
        free(john2.first_name);
        free(john2.last_name);
        return 0;
}
```

Oh, no



Pointer Hazards

Defense

- One `malloc`, one `free` (per pointer *values*), and do not use the pointer after `free`
- When you see a pointer, ask:
  - Where does it point to?
  - If on the stack, what is the function that will destroy it?
  - If on the heap, who is in charge of freeing it?
    - When is the last time this data is needed?
    - Who else might have this pointer?
- When in doubt, `valgrind`