Pointers III

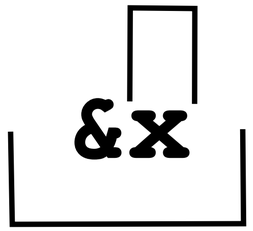
CS143: lecture 9

Konstantinos Ameranis, July 7

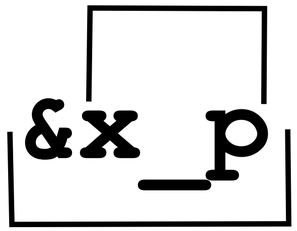
Pointers

Review

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

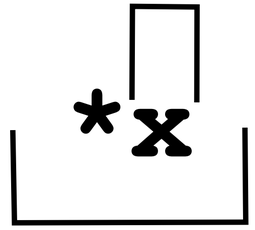



```
type : int *
value: 100
```

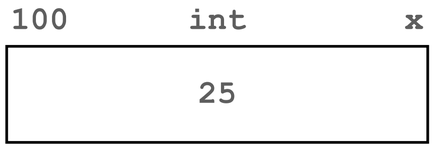


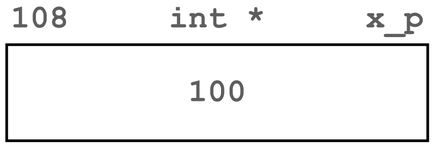

```
type : int **
value: 108
```

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

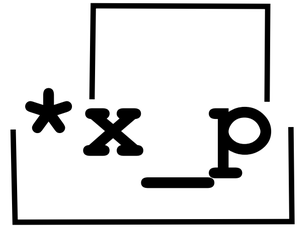



```
error
```






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

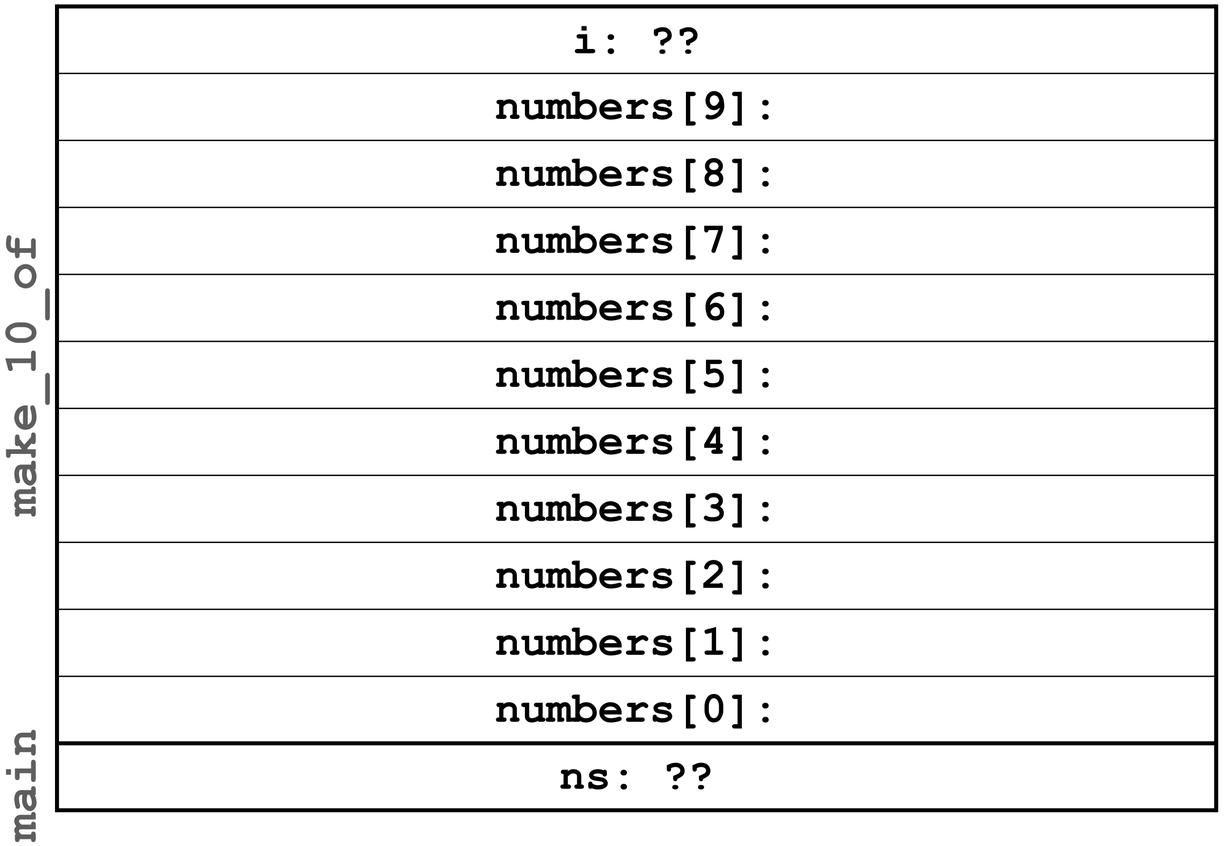



```
type : int
value: 25
```

Pointers

How about returning an array?

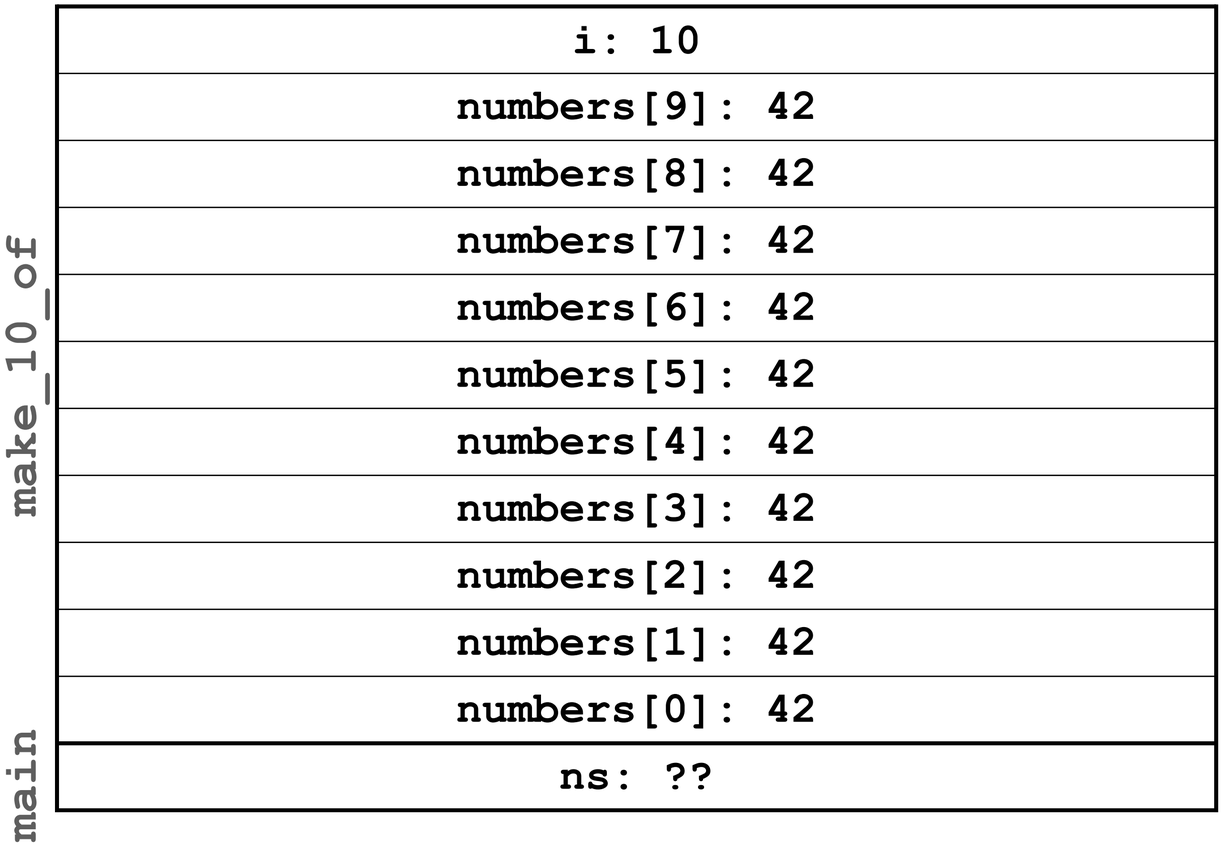
```
int *make_10_of(int n)
{
        int numbers[10];

        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?

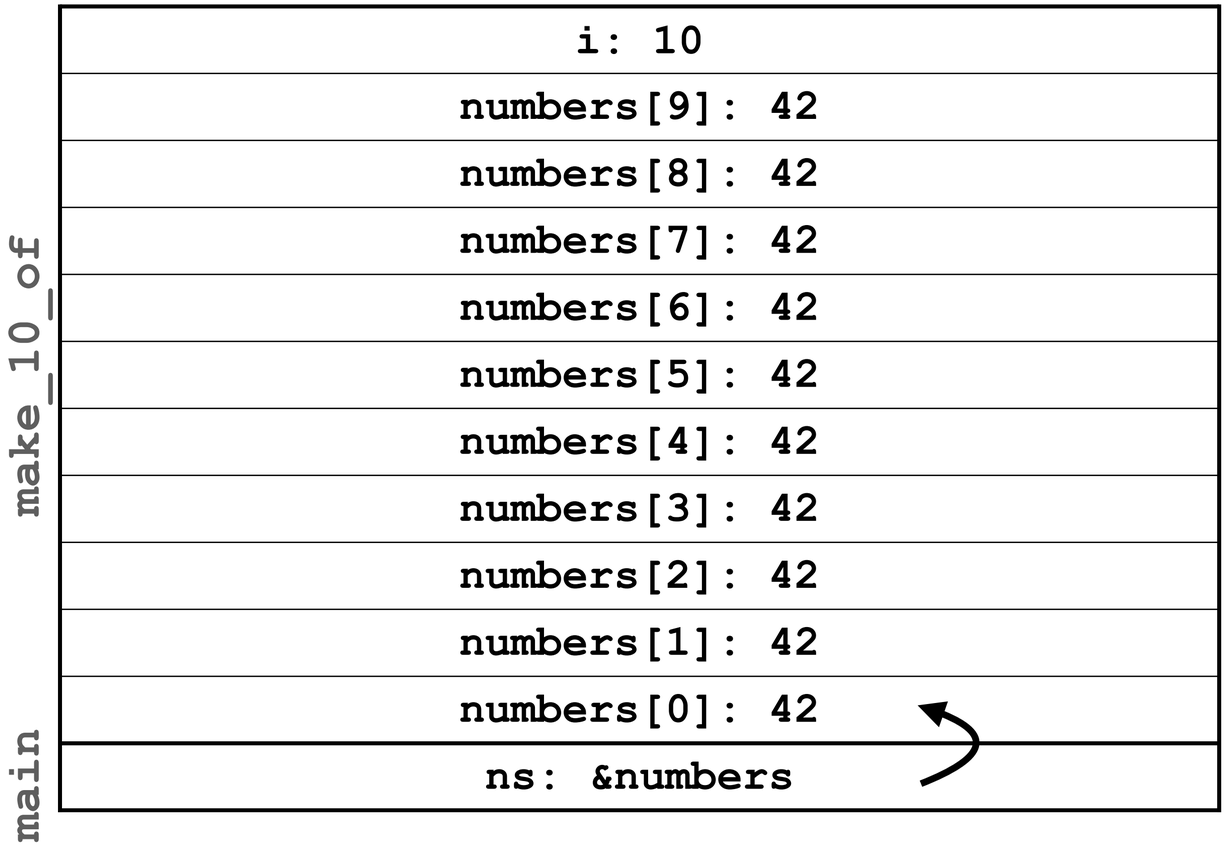
```
int *make_10_of(int n)
{
        int numbers[10];

        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?

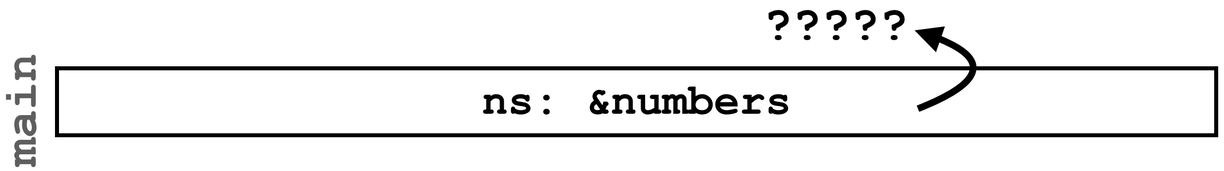
```
int *make_10_of(int n)
{
        int numbers[10];

        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;
}
```

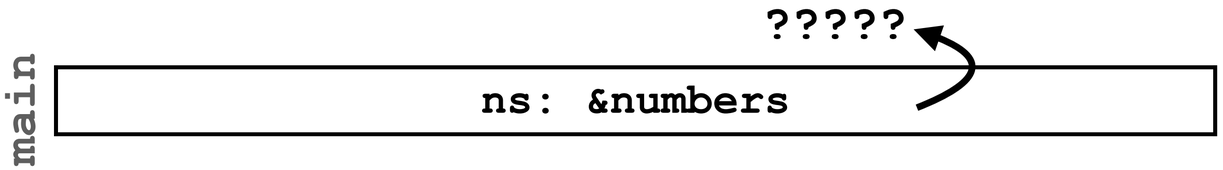


Pointers

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

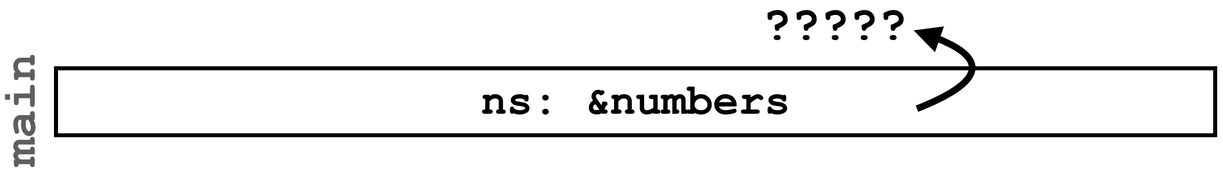


Pointers

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.

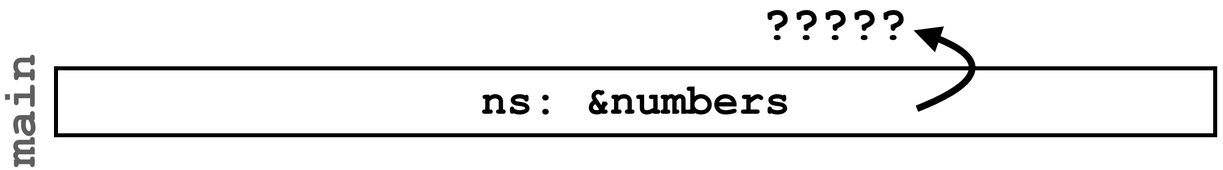


Pointers

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];
```

Pointers

How about variable-size arrays?

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

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

Pointers

How about variable-size arrays?

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

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

Pointers

How about variable-size arrays?

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

- `clang` wants to know how big a frame is.

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

Pointers

How about variable-size arrays?

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

- `clang` wants to know how big a frame is.
  - When calling the function, how much of stack does it need?

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

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?

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)

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

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
  - 🤷‍♂️

Pointers

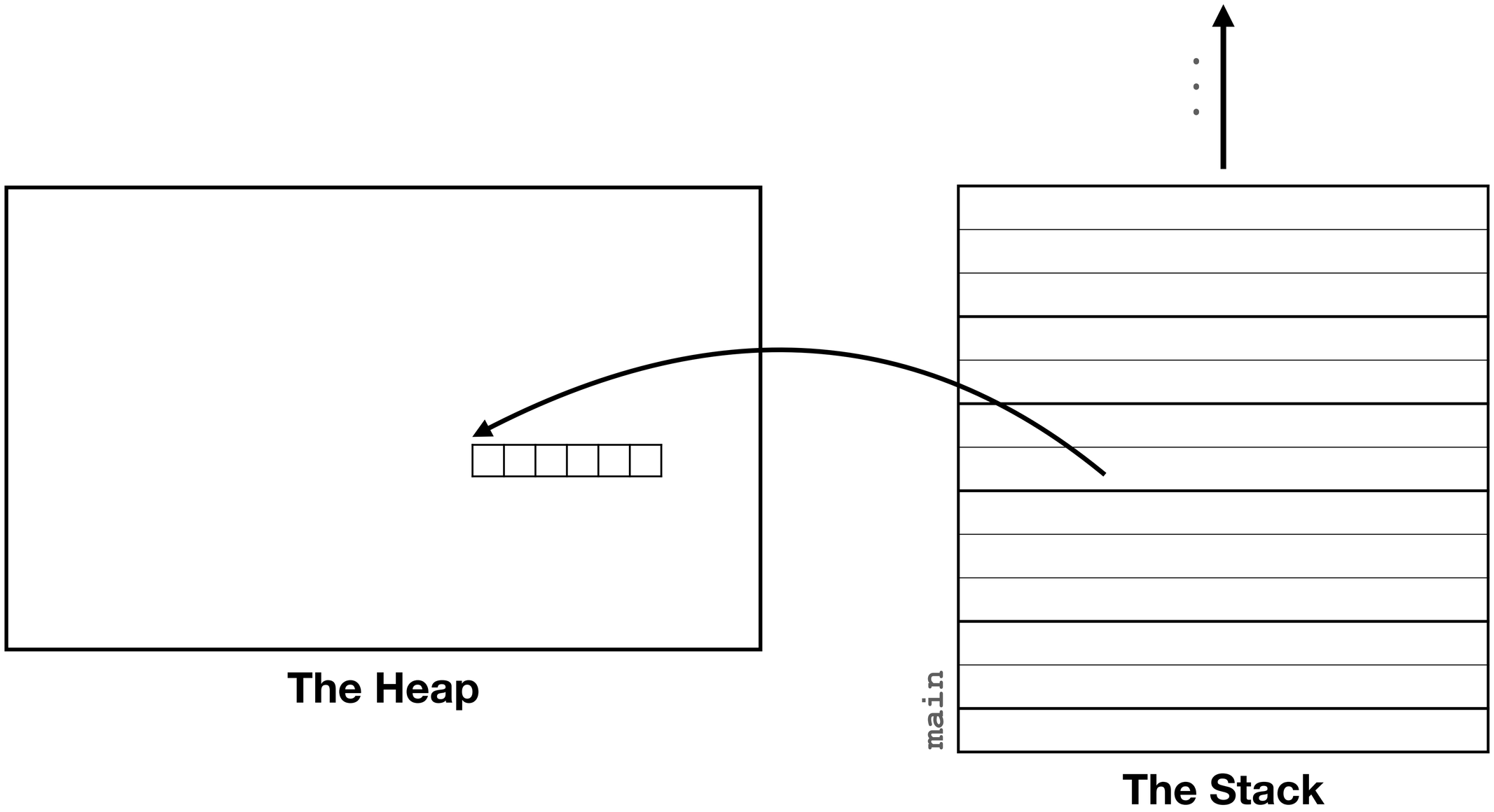
How about variable-size arrays?

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

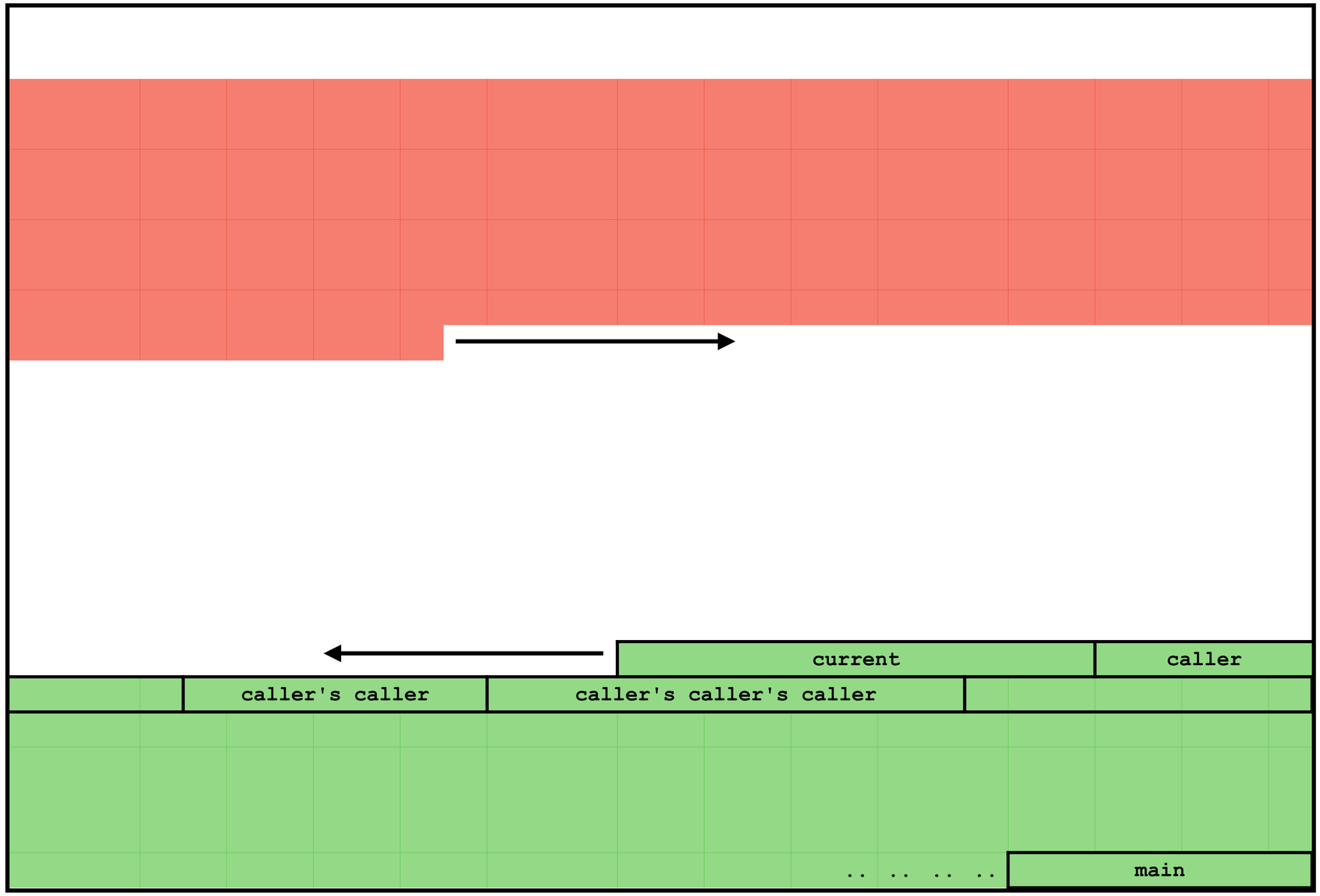
```
int N_LETTERS = 26;
int freq[N_LETTERS];
```

- `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`

The Heap

`malloc` and `free`

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

...

free(numbers);
```

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.

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`

The Heap

`malloc` and `free`

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

...

free(numbers);

*numbers = 123;
```

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.

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.

The Heap

`malloc` and `free`

```c
#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

Heap

The Heap

Stack vs Heap

Stack

- Acquire memory:

Heap

- Acquire memory:

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables

Heap

- Acquire memory:

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables

Heap

- Acquire memory:
  - `ptr = malloc(n)`

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables
  - size: compiler calculates *before* running (static)

Heap

- Acquire memory:
  - `ptr = malloc(n)`

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables
  - size: compiler calculates *before* running (static)

Heap

- Acquire memory:
  - `ptr = malloc(n)`
  - size: you provide *during* running (dynamic)

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables
  - size: compiler calculates *before* running (static)
- Release memory:

Heap

- Acquire memory:
  - `ptr = malloc(n)`
  - size: you provide *during* running (dynamic)
- Release memory:

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables
  - size: compiler calculates *before* running (static)
- Release memory:
  - do nothing

Heap

- Acquire memory:
  - `ptr = malloc(n)`
  - size: you provide *during* running (dynamic)
- Release memory:

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables
  - size: compiler calculates *before* running (static)
- Release memory:
  - do nothing

Heap

- Acquire memory:
  - `ptr = malloc(n)`
  - size: you provide *during* running (dynamic)
- Release memory:
  - `free(ptr)`

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables
  - size: compiler calculates *before* running (static)
- Release memory:
  - do nothing

Heap

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

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*

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?

Pointers

How about returning an array?

```
int *make_10_of(int n)
{
```

Pointers

How about returning an array?

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

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));
```

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;
}
```

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()
{
```

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);
```

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 */
```

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);
```

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;
}
```

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;
}
```

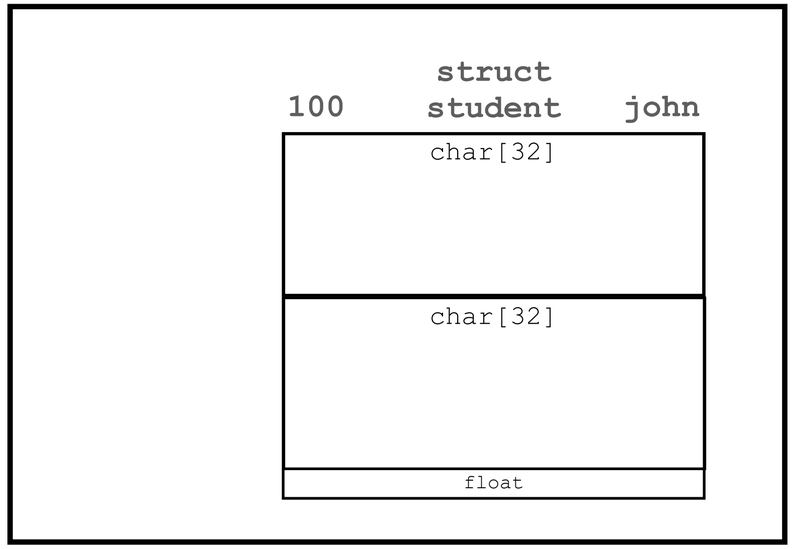
Forgetting to free here causes memory leak.

Pointer Hazards

Be aware of shallow copy

```c
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

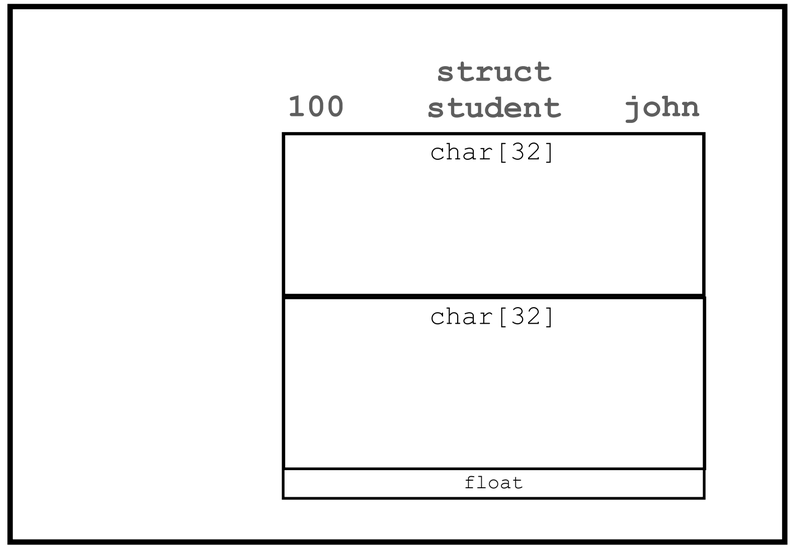


Pointer Hazards

Be aware of shallow copy

```c
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

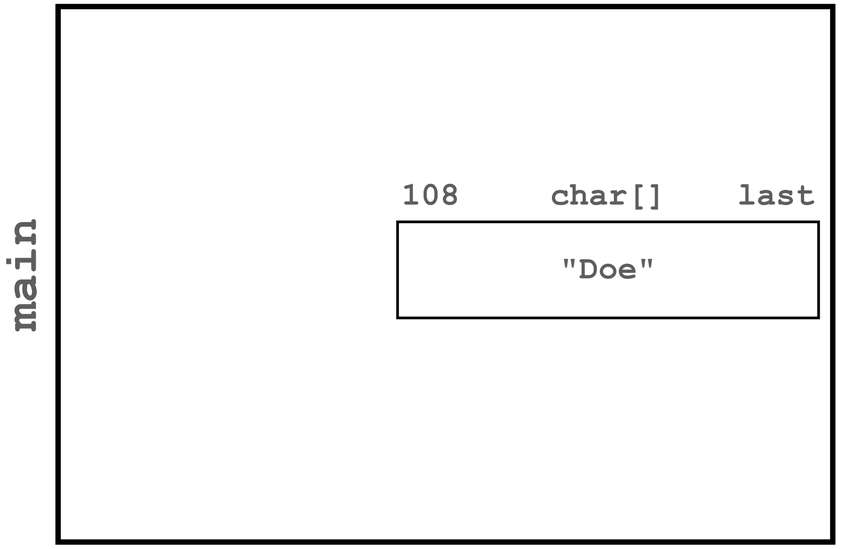
Be aware of shallow copy

```
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

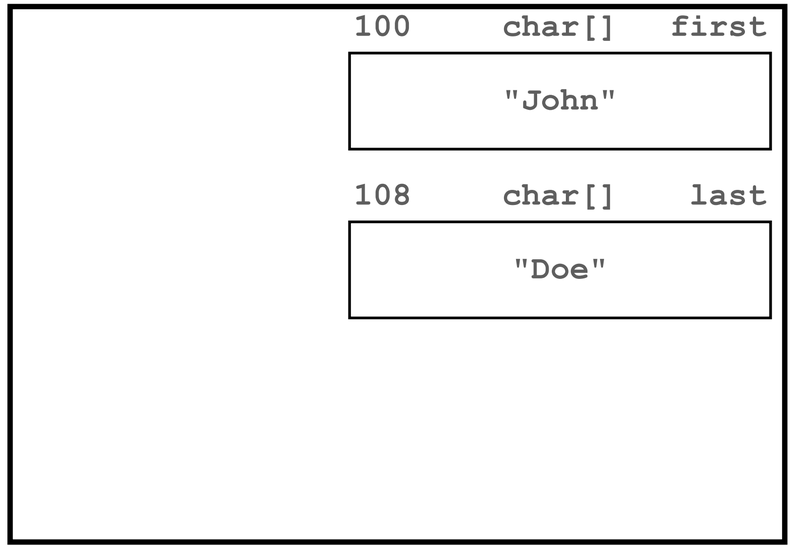
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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

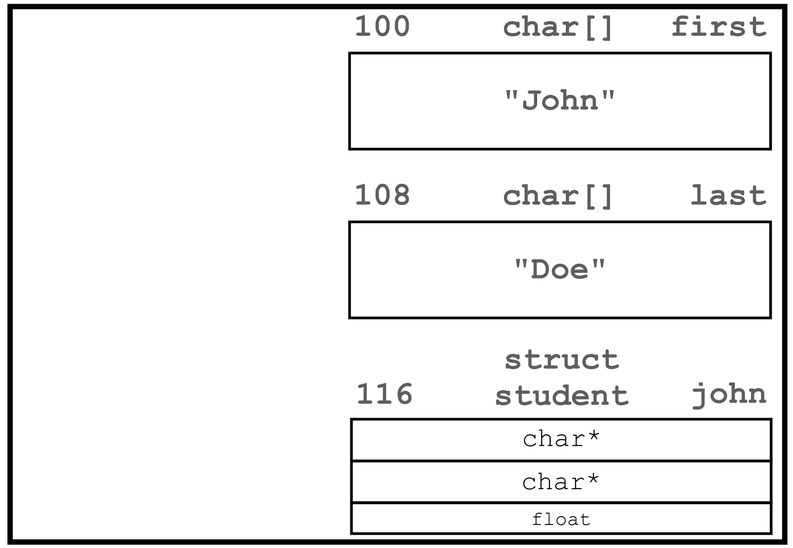
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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

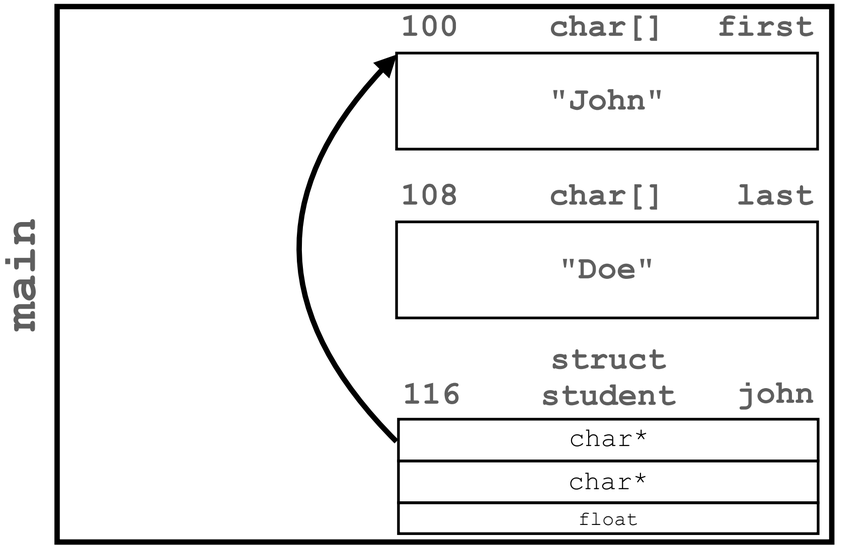
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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

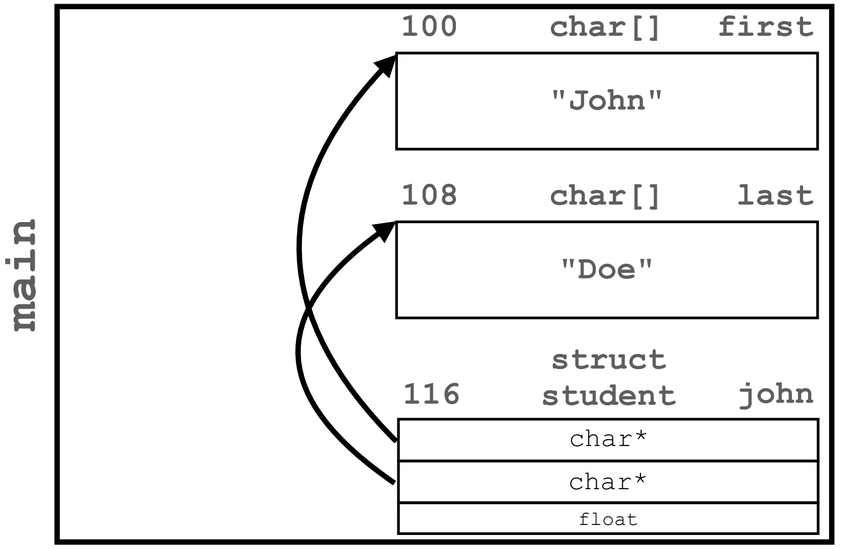
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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

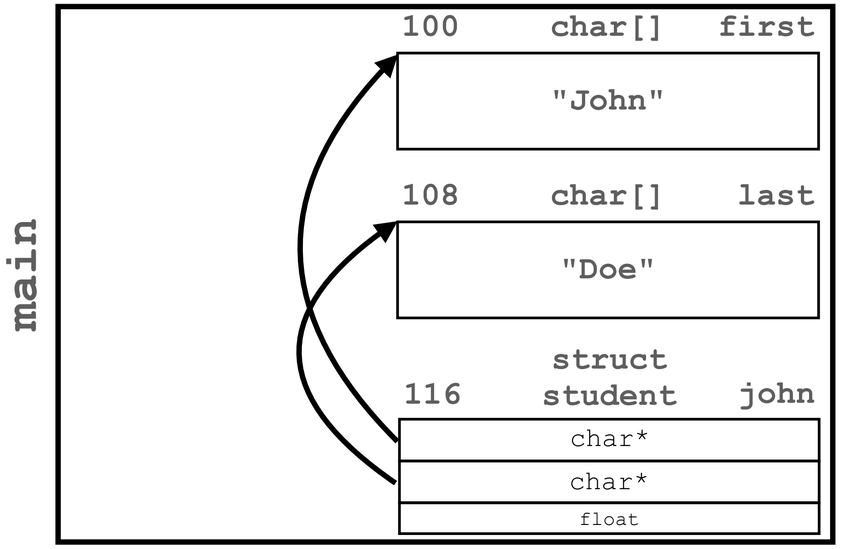
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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

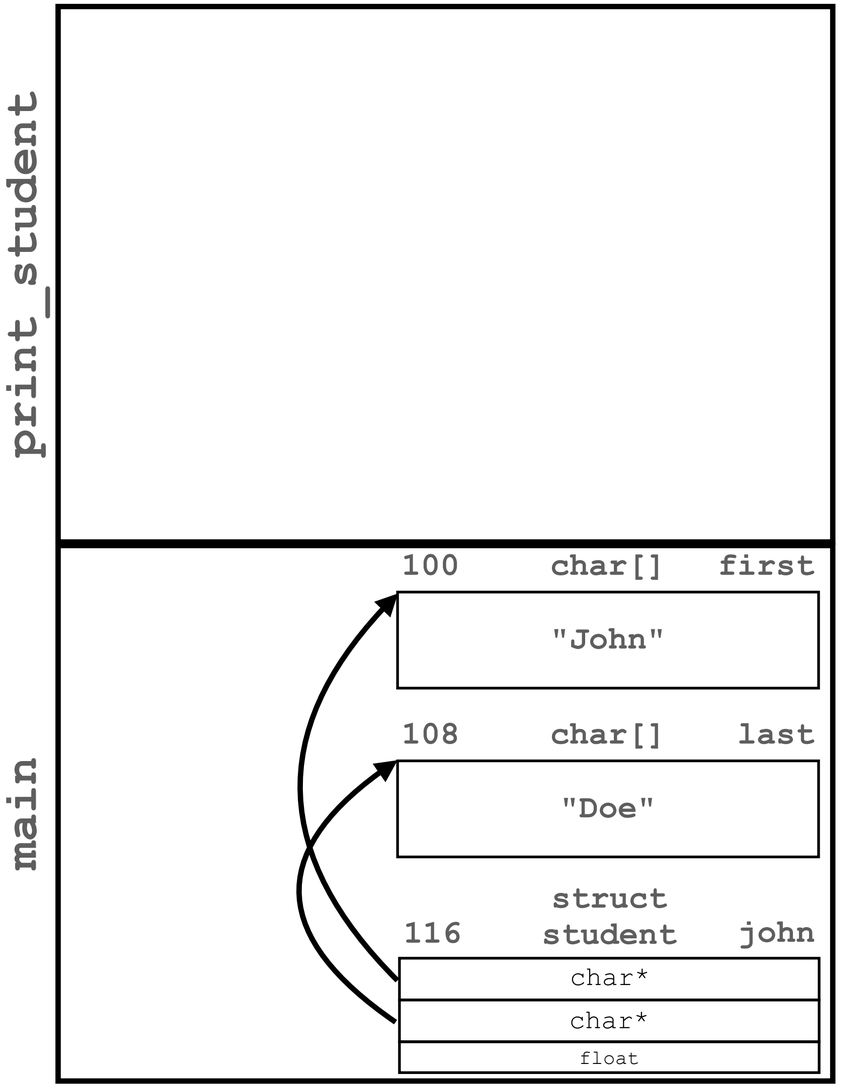
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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;
}
```



Pointer Hazards

Be aware of shallow copy

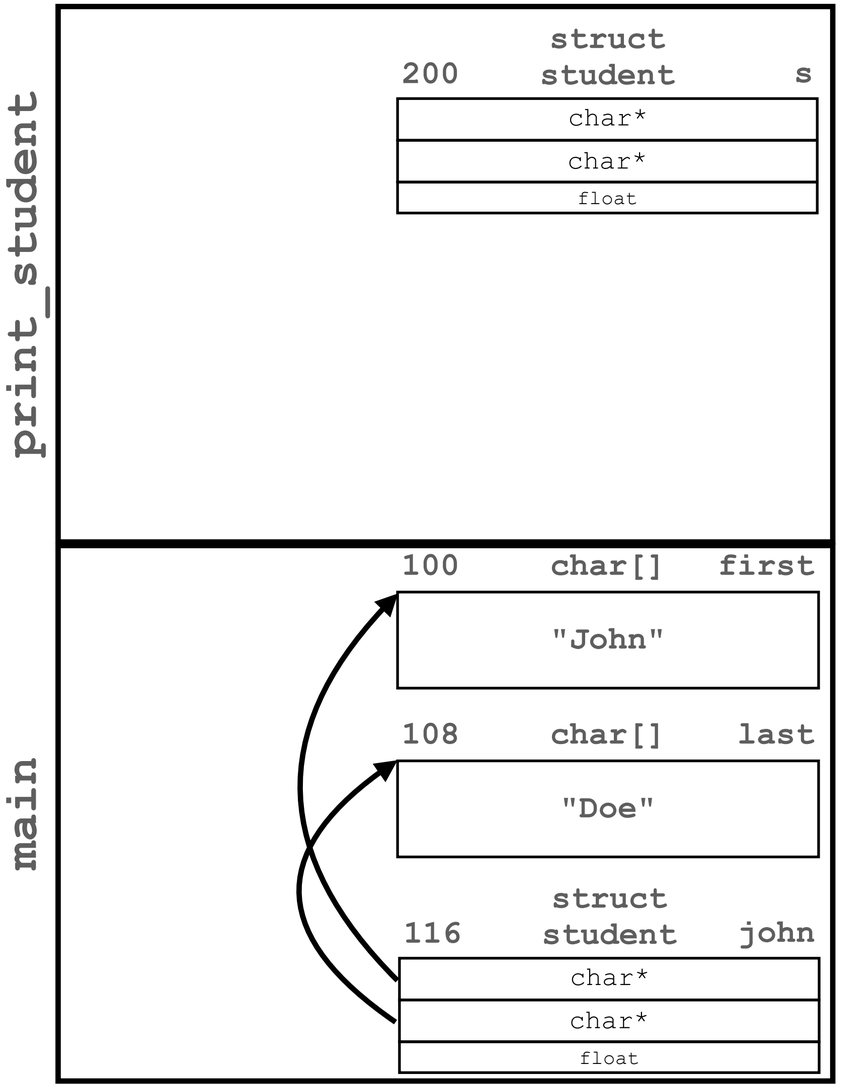
```c
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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;
}
```



Pointer Hazards

Be aware of shallow copy

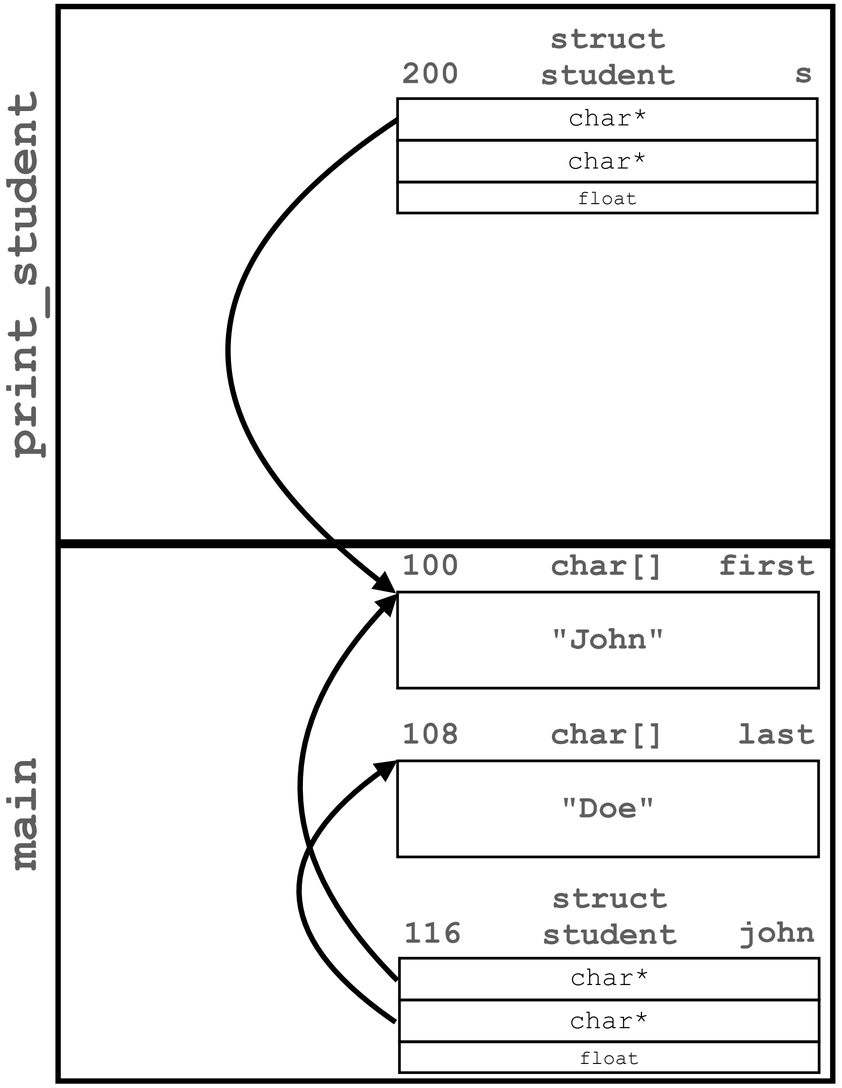
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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;
}
```



Pointer Hazards

Be aware of shallow copy

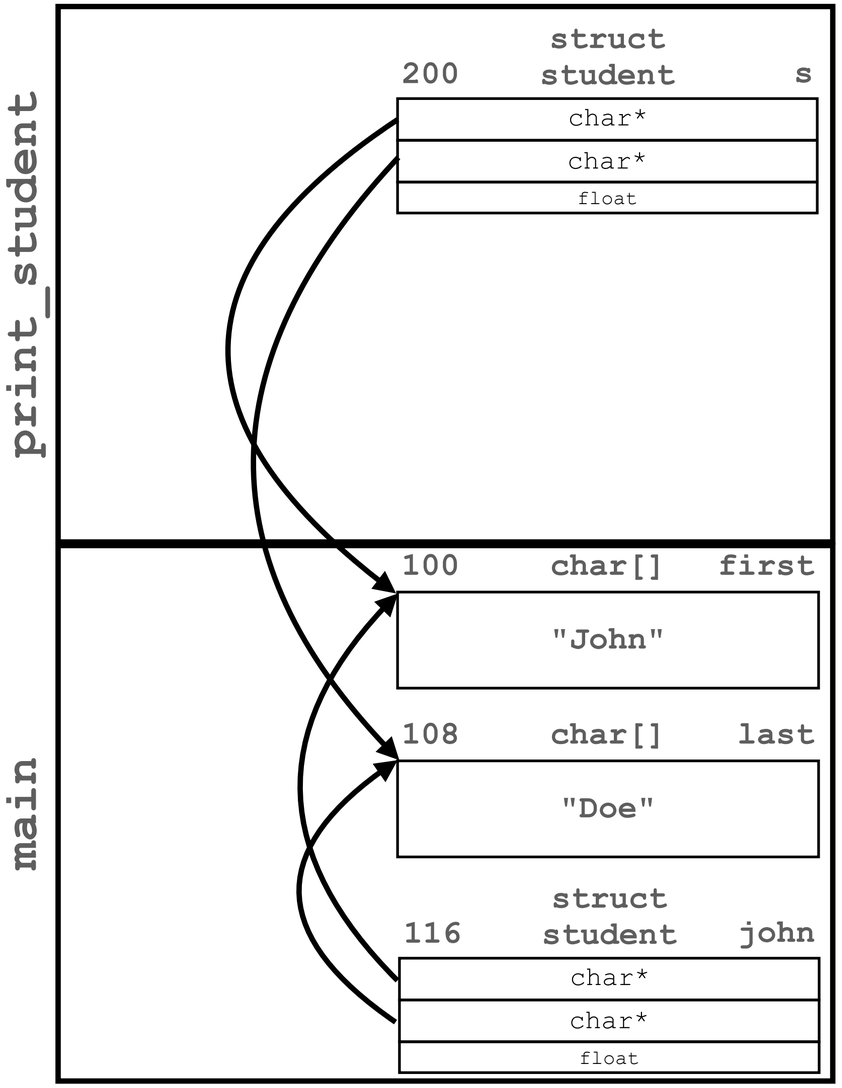
```c
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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;
}
```



Pointer Hazards

Be aware of shallow copy

```
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;
}
```

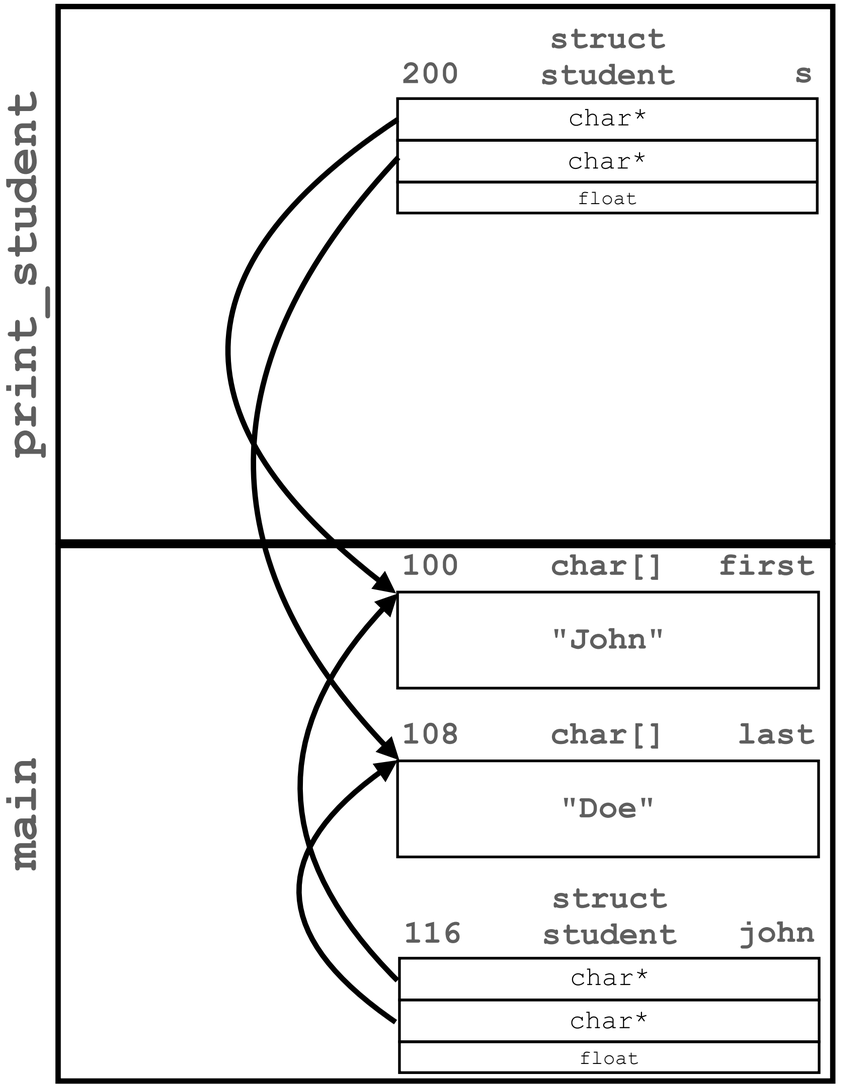


Pointer Hazards

Be aware of shallow copy

```
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

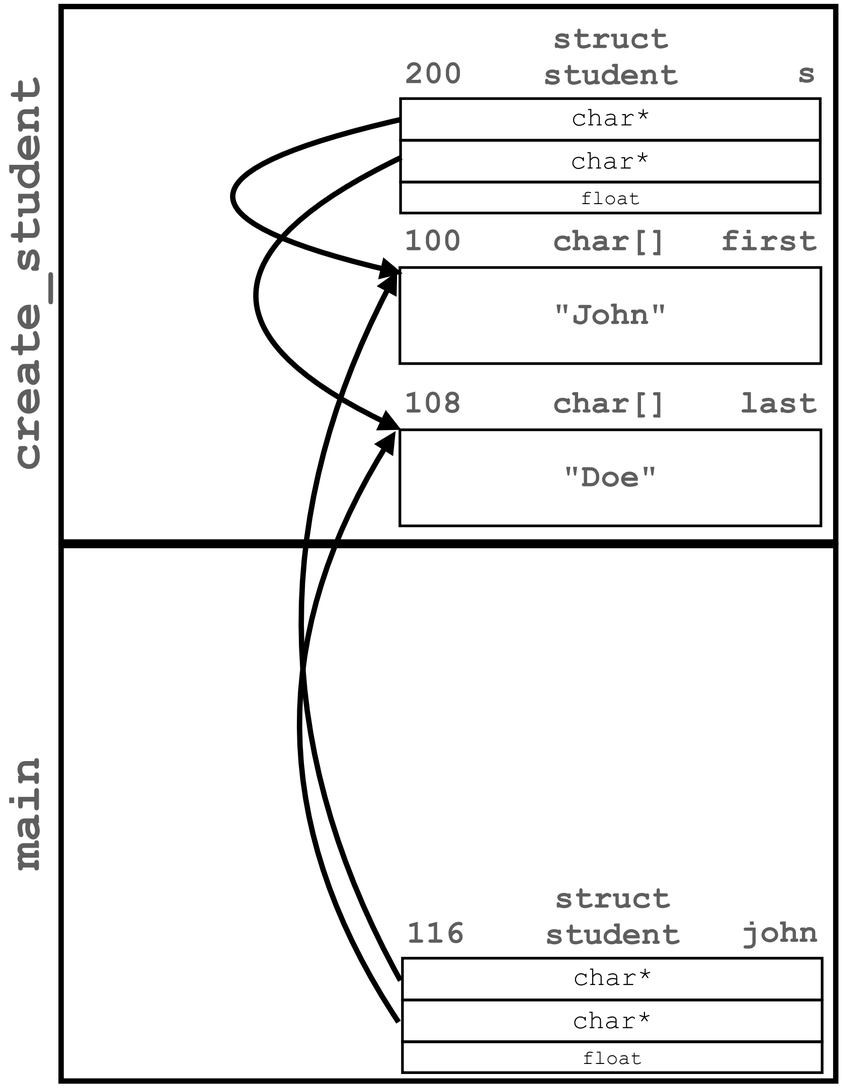
Be aware of dangling pointer (again)

```
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)

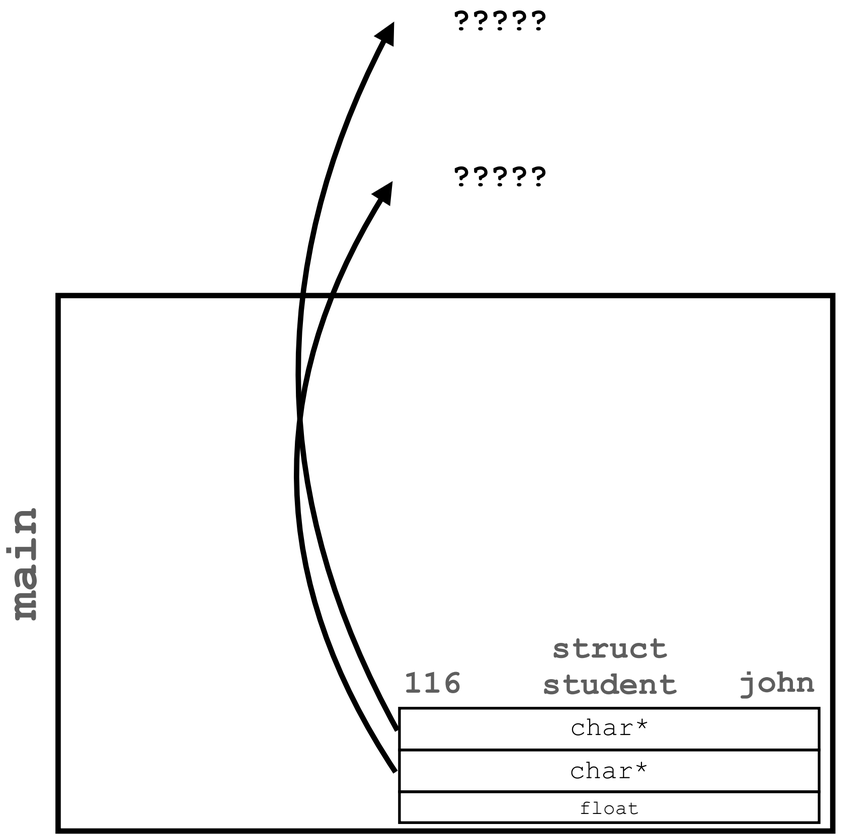
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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)

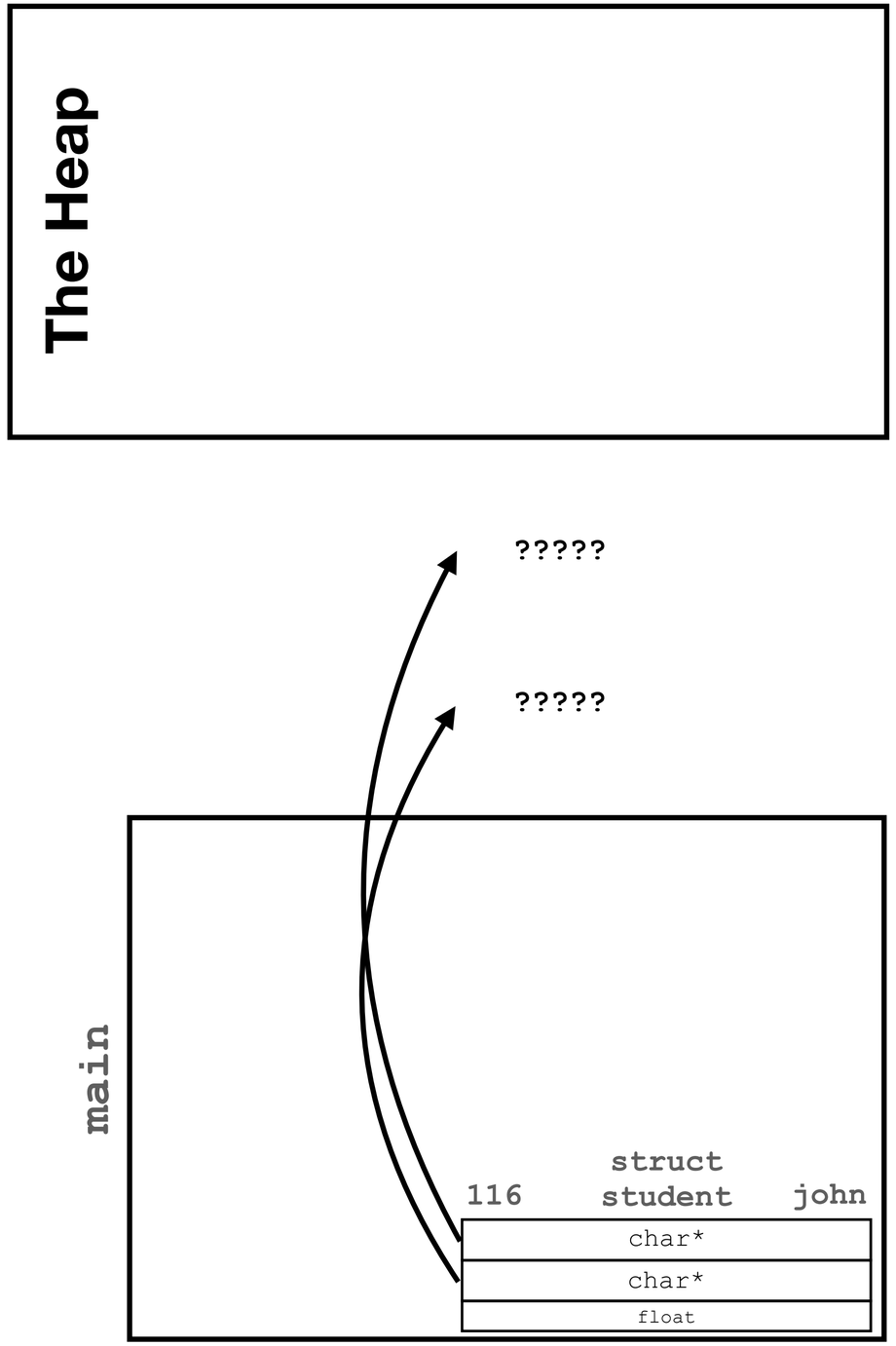
```c
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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)


```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 dangling pointer (again)

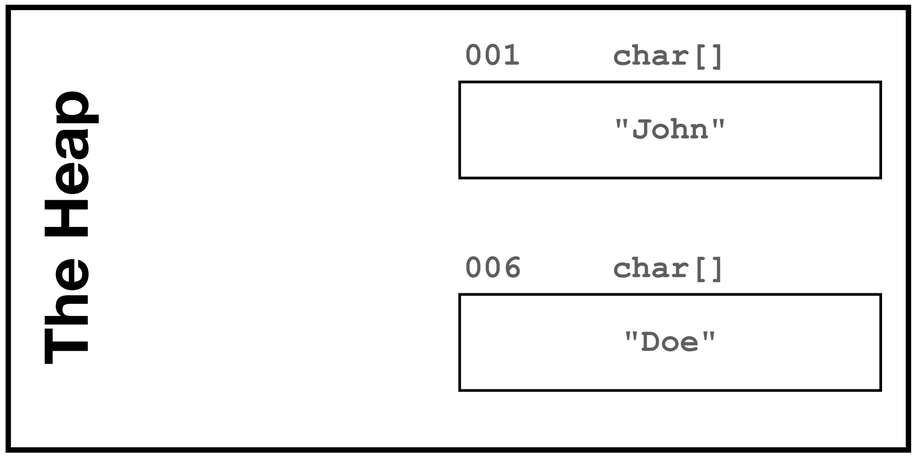

```c
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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;
}
```

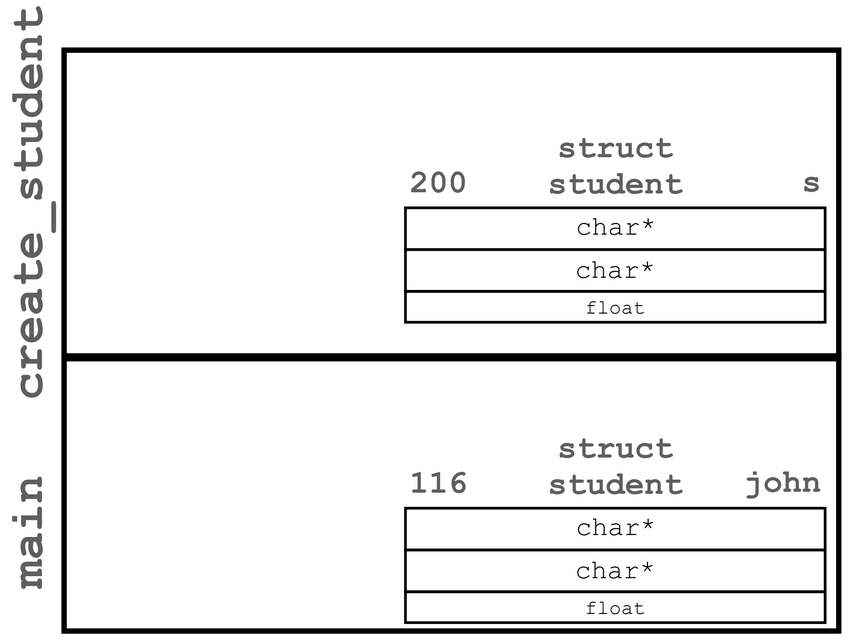
strdup in <string.h> duplicates the string on the heap. i.e. strdup calls malloc, and then copy the string to the malloc'ed space.
It's a very handy function.

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;
}
```



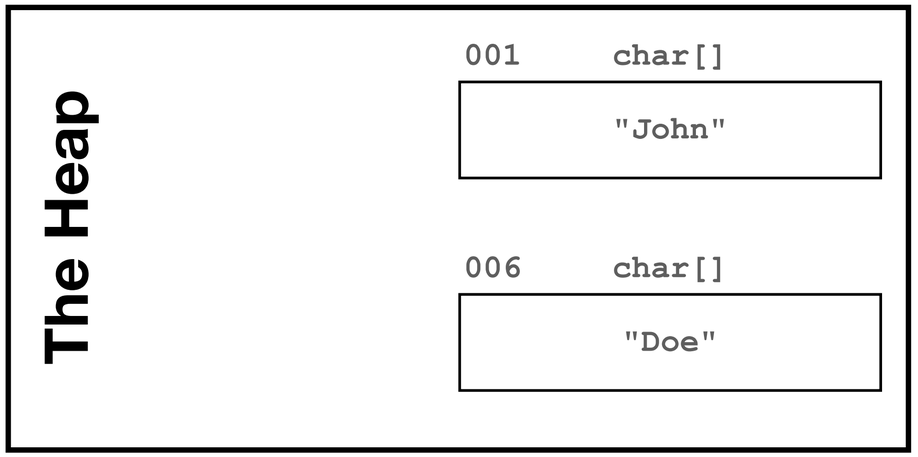


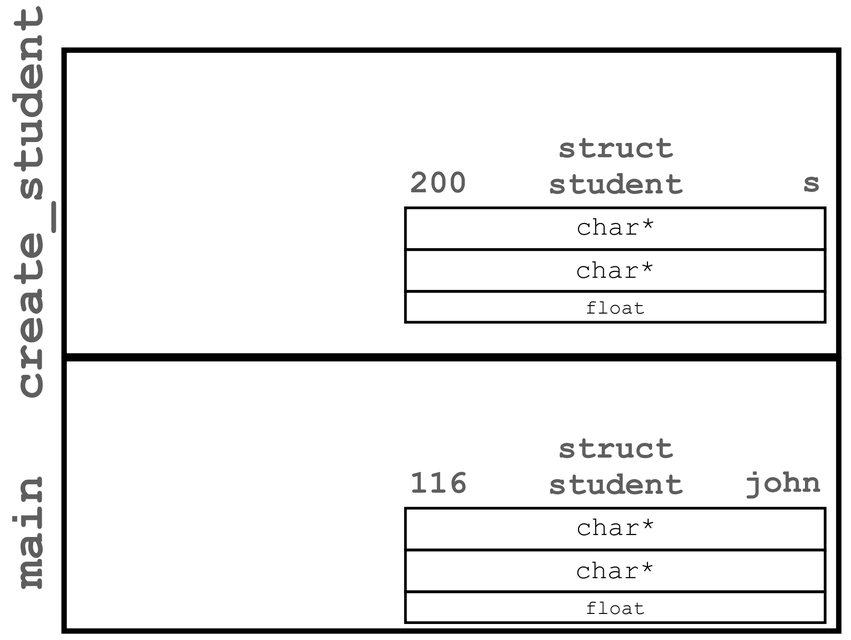
Pointer Hazards

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.

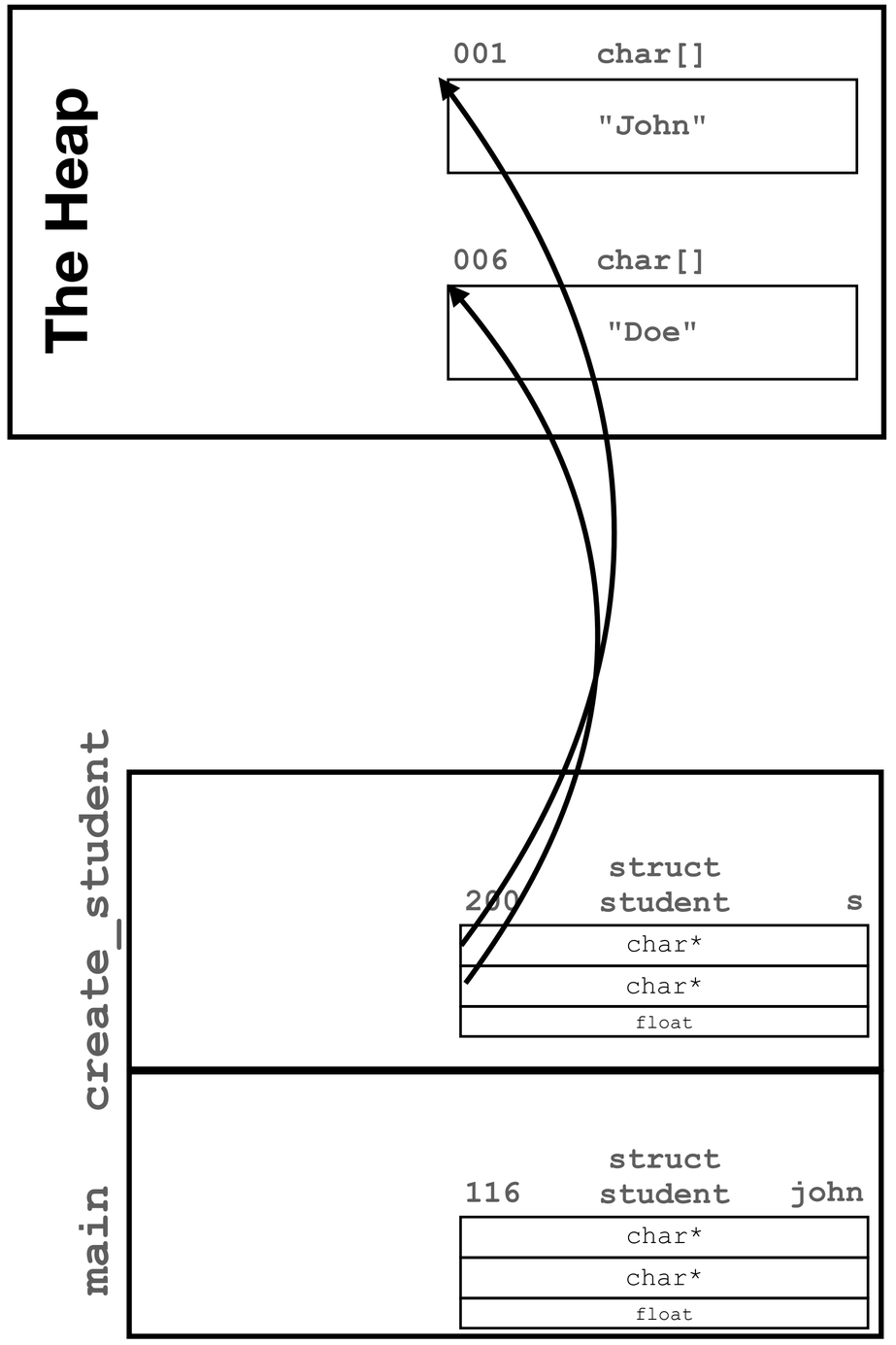




Pointer Hazards

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;
}
```

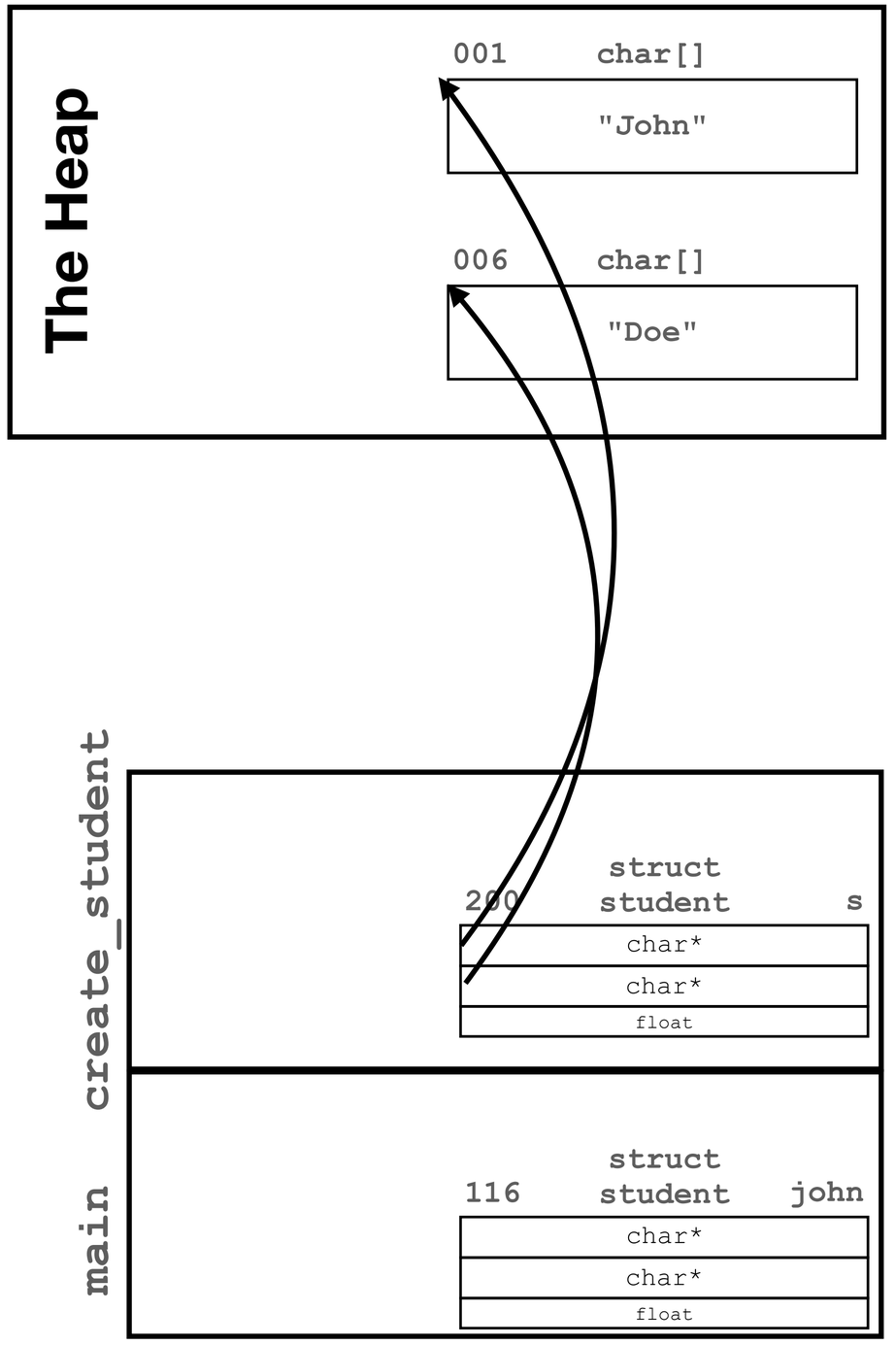


Pointer Hazards

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;
}
```

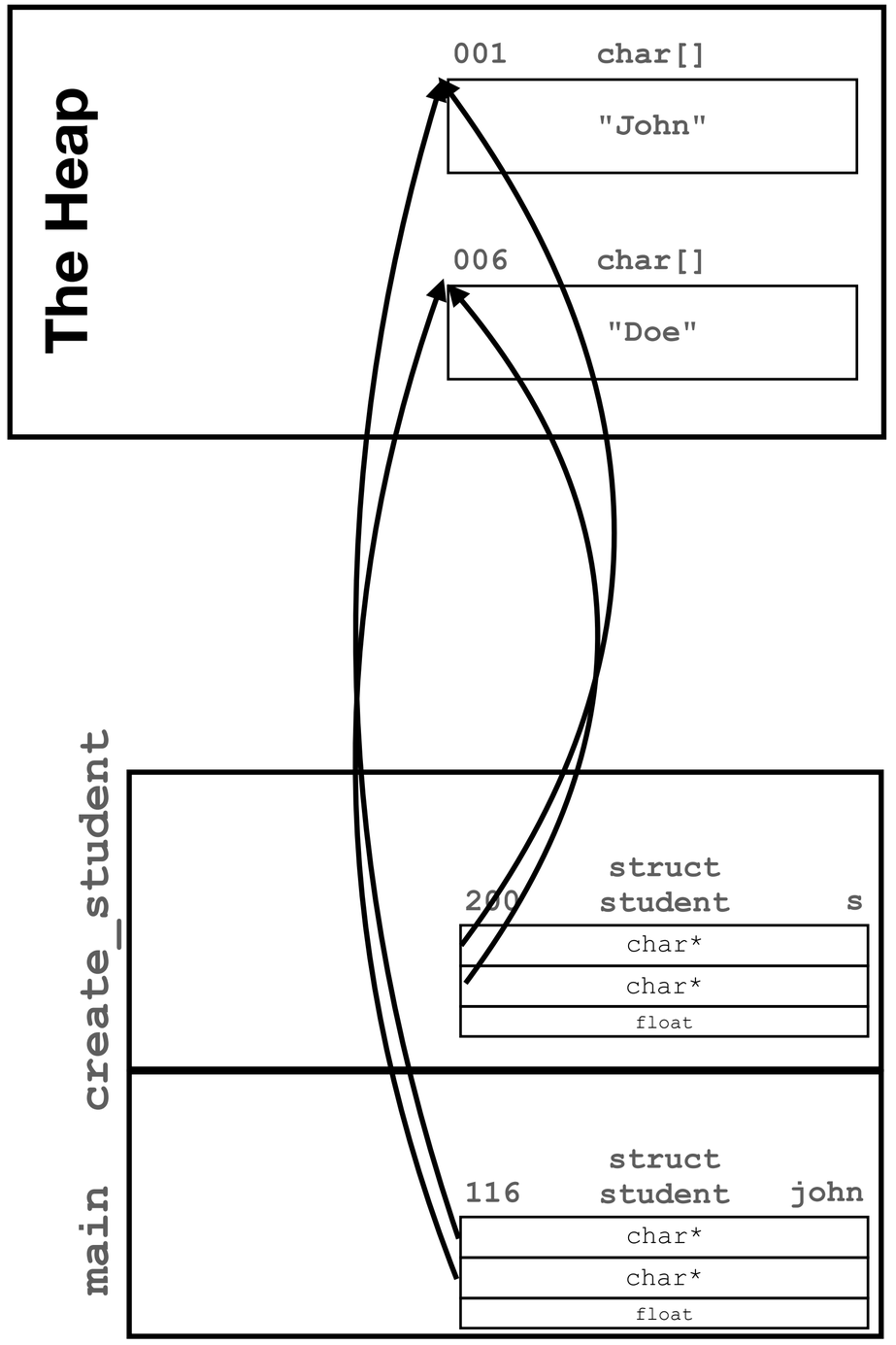
strdup in <string.h> duplicates the string on the heap. i.e. strdup calls malloc, and then copy the string to the malloc'ed space.
It's a very handy function.



Pointer Hazards

Be aware of dangling pointer (again)

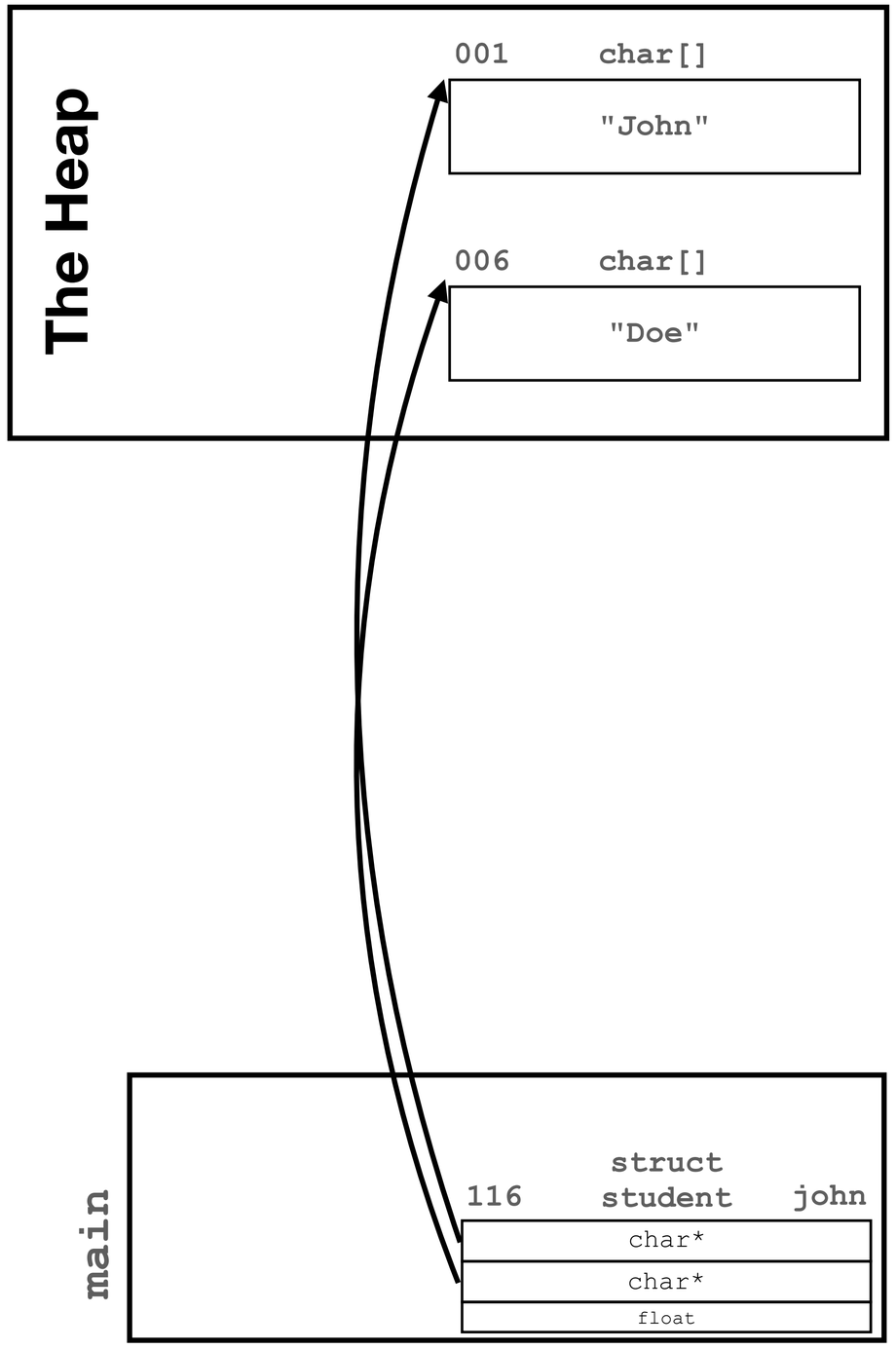
```c
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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;
}
```



Pointer Hazards

Be aware of dangling pointer (again)

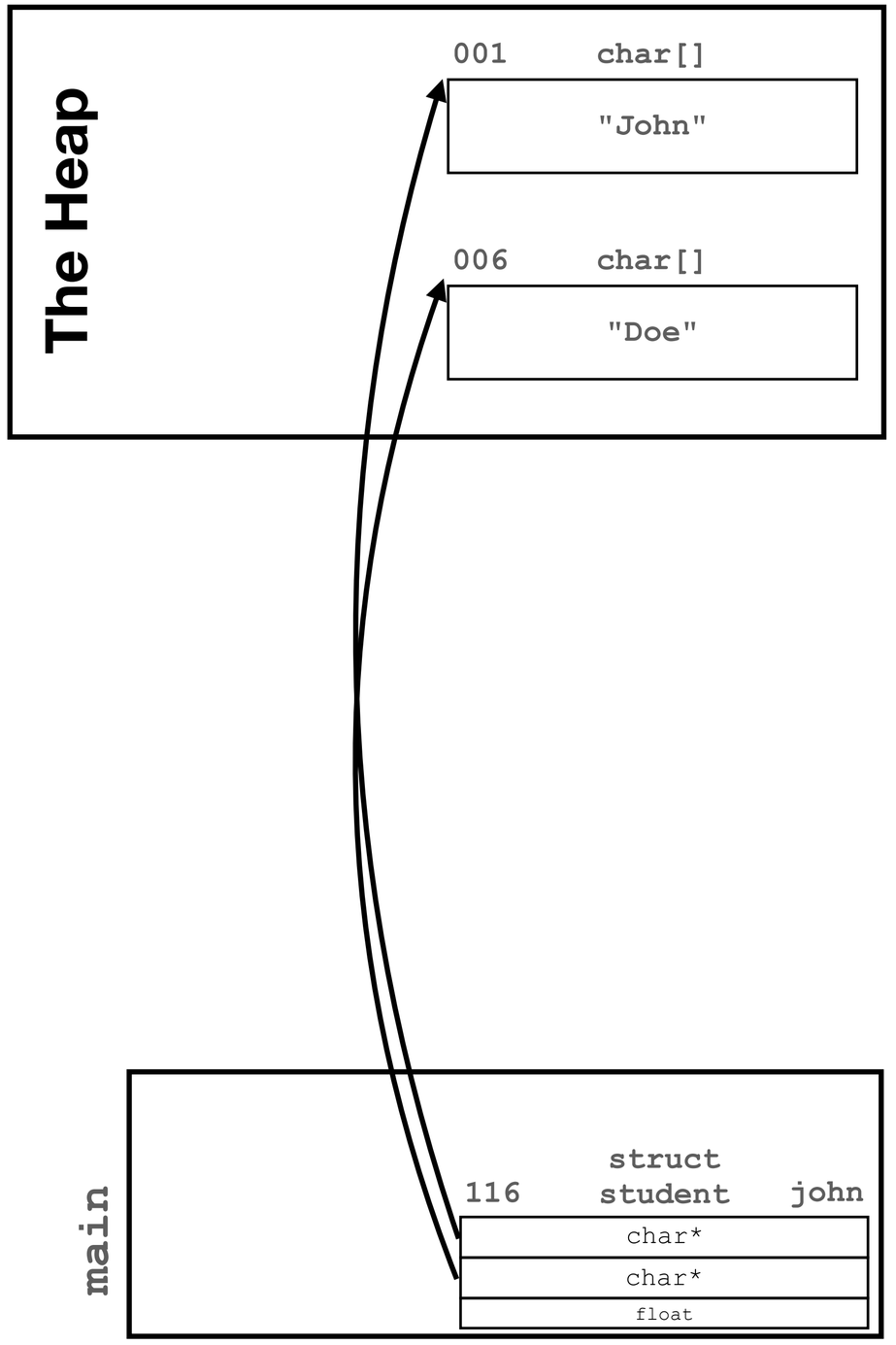
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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 memory leak

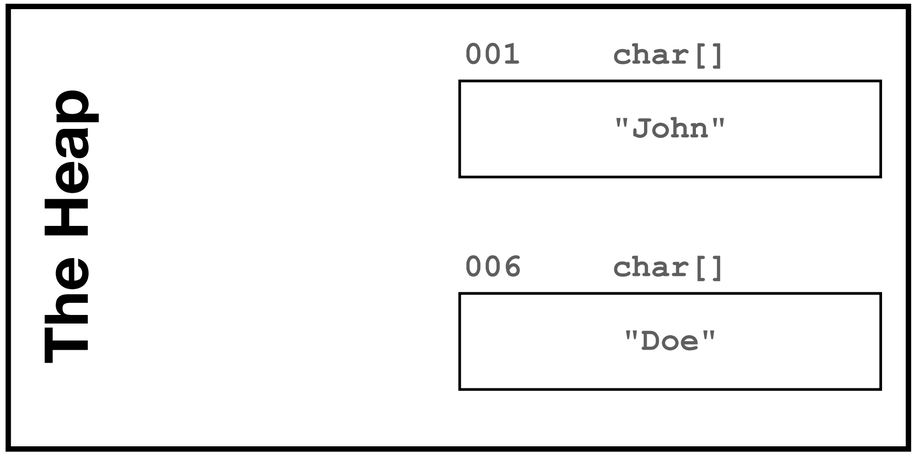
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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

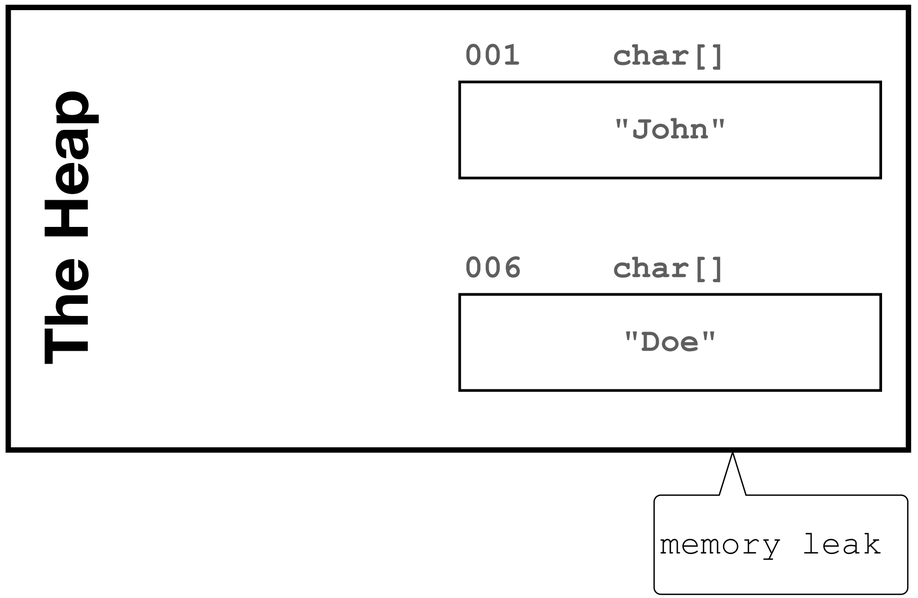
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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

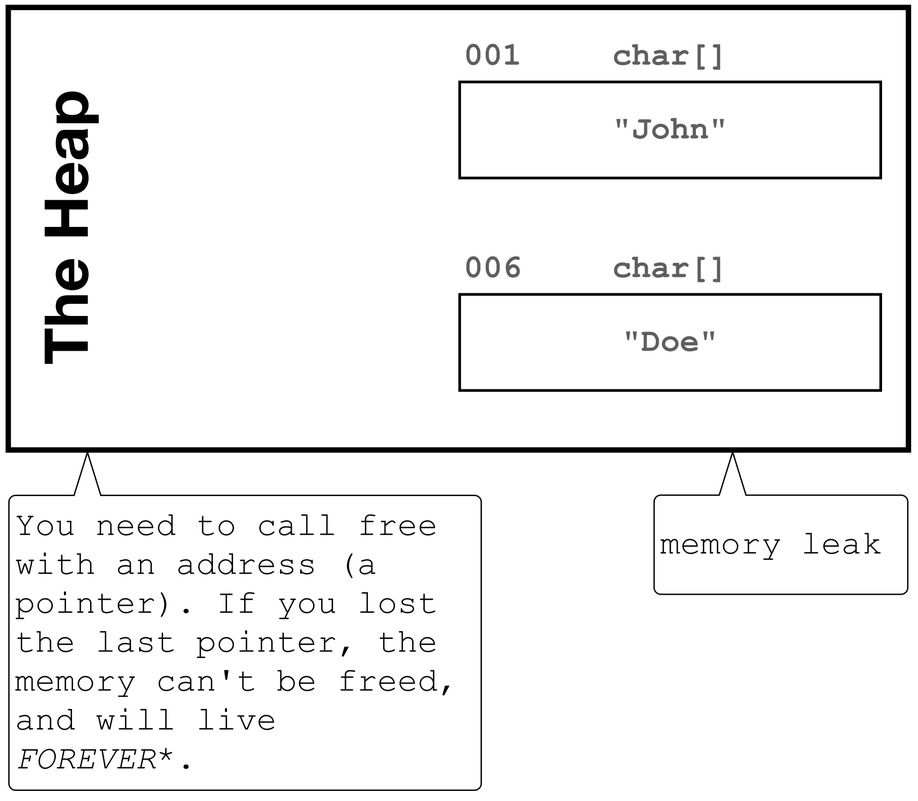
```c
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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

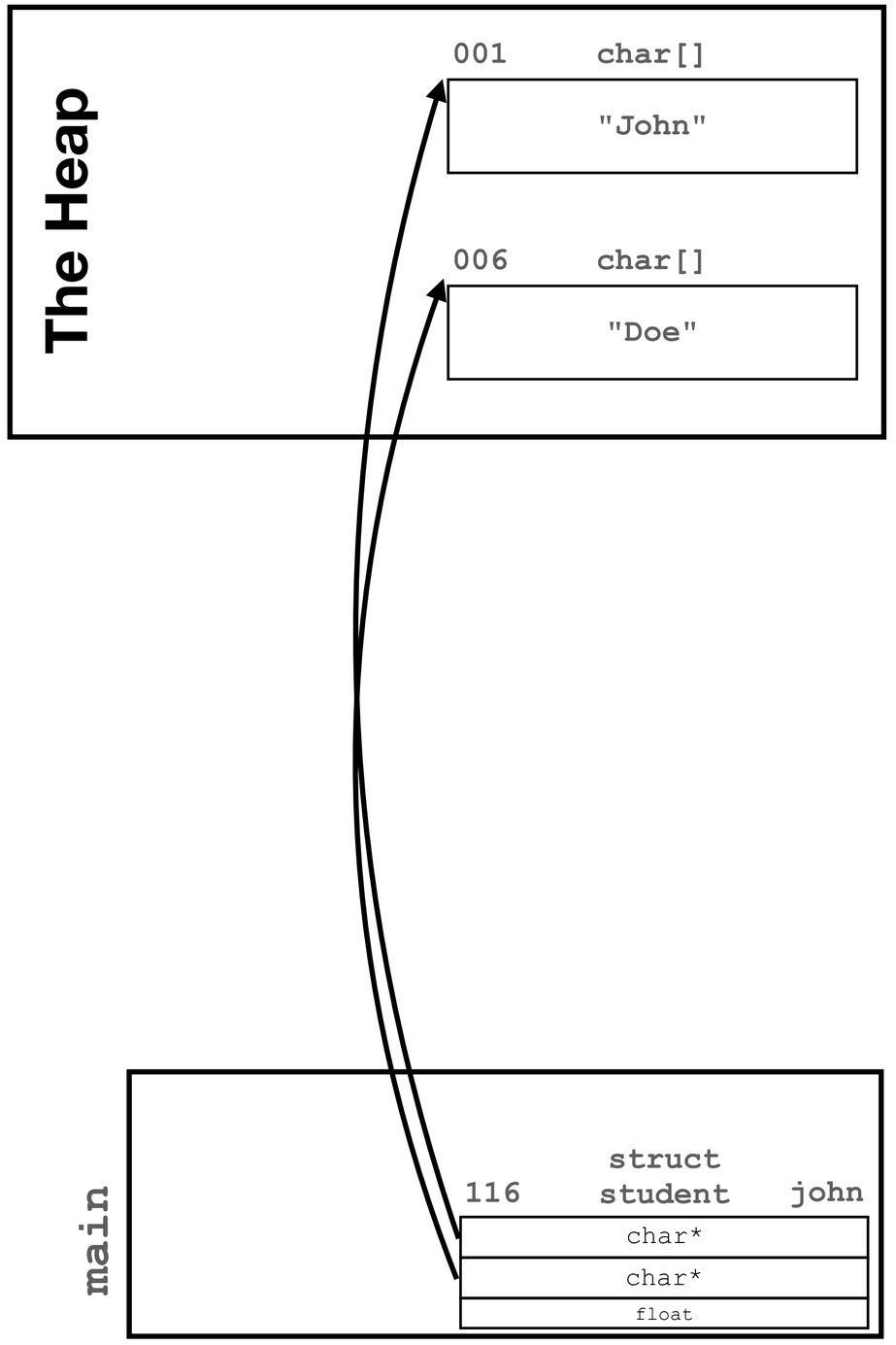
```c
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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();

        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;
}
```

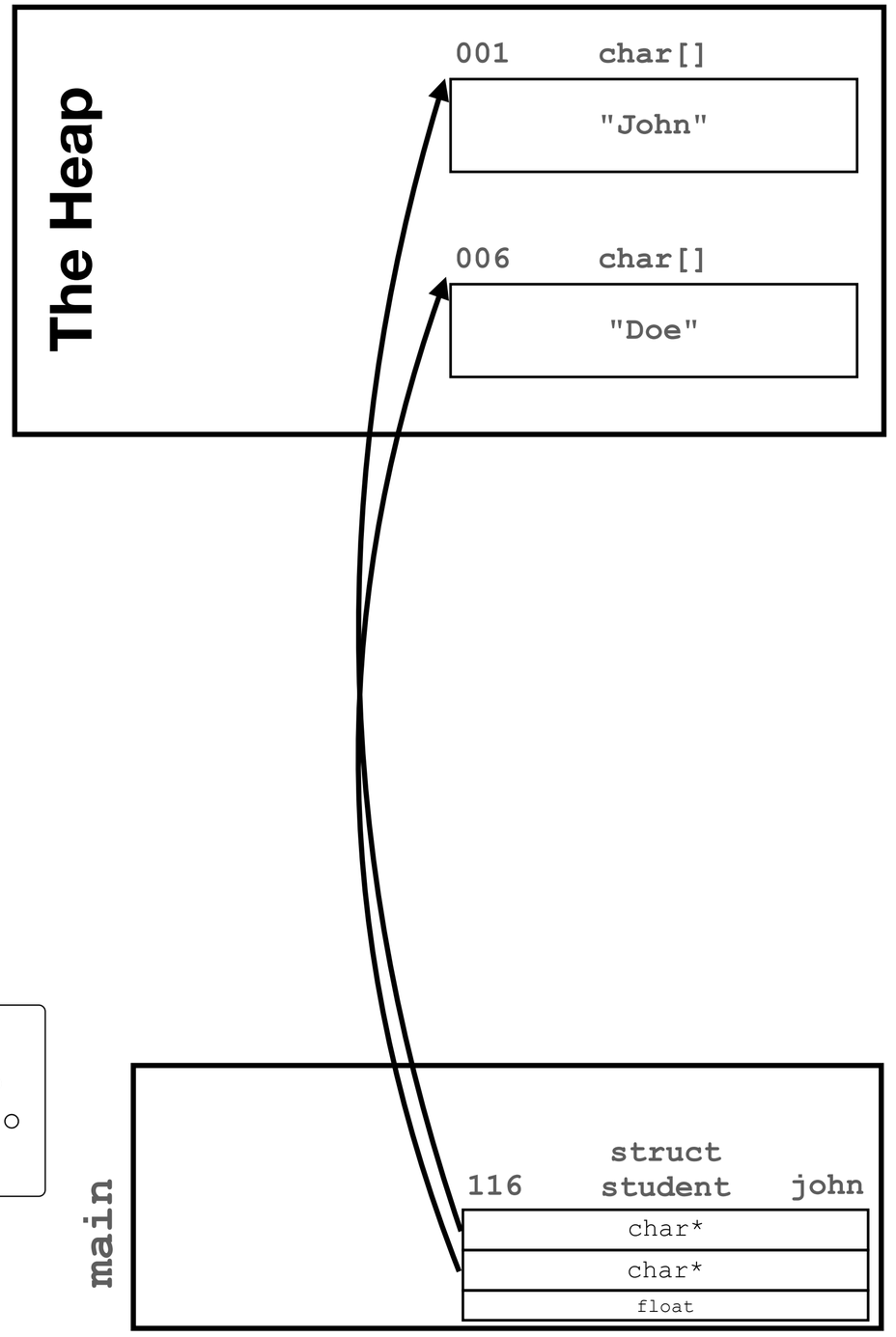


Pointer Hazards

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;
}
```

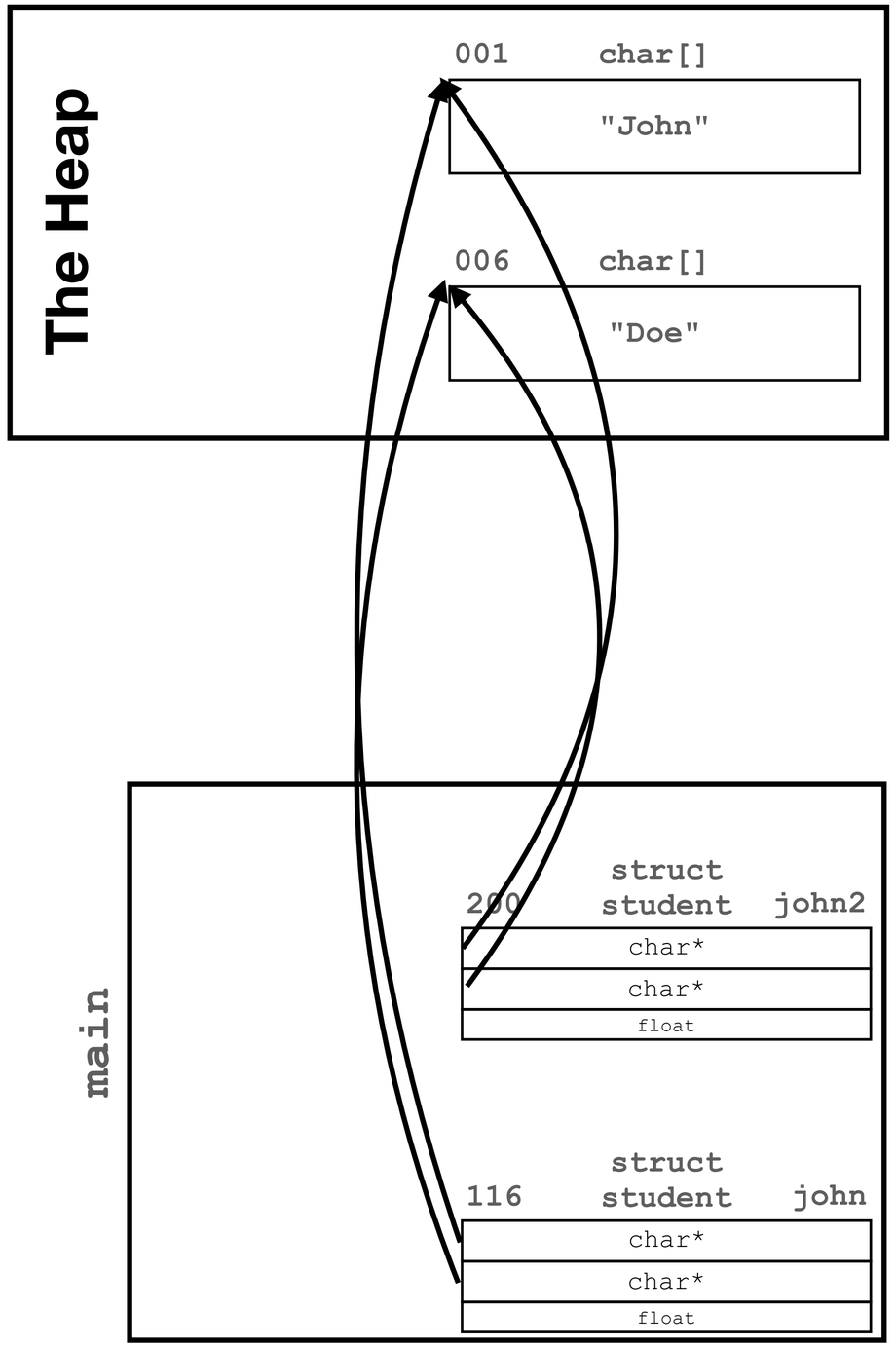
remember to call free when the last pointer to the heap data is about to go out of scope



Pointer Hazards

Be aware of double-free corruption

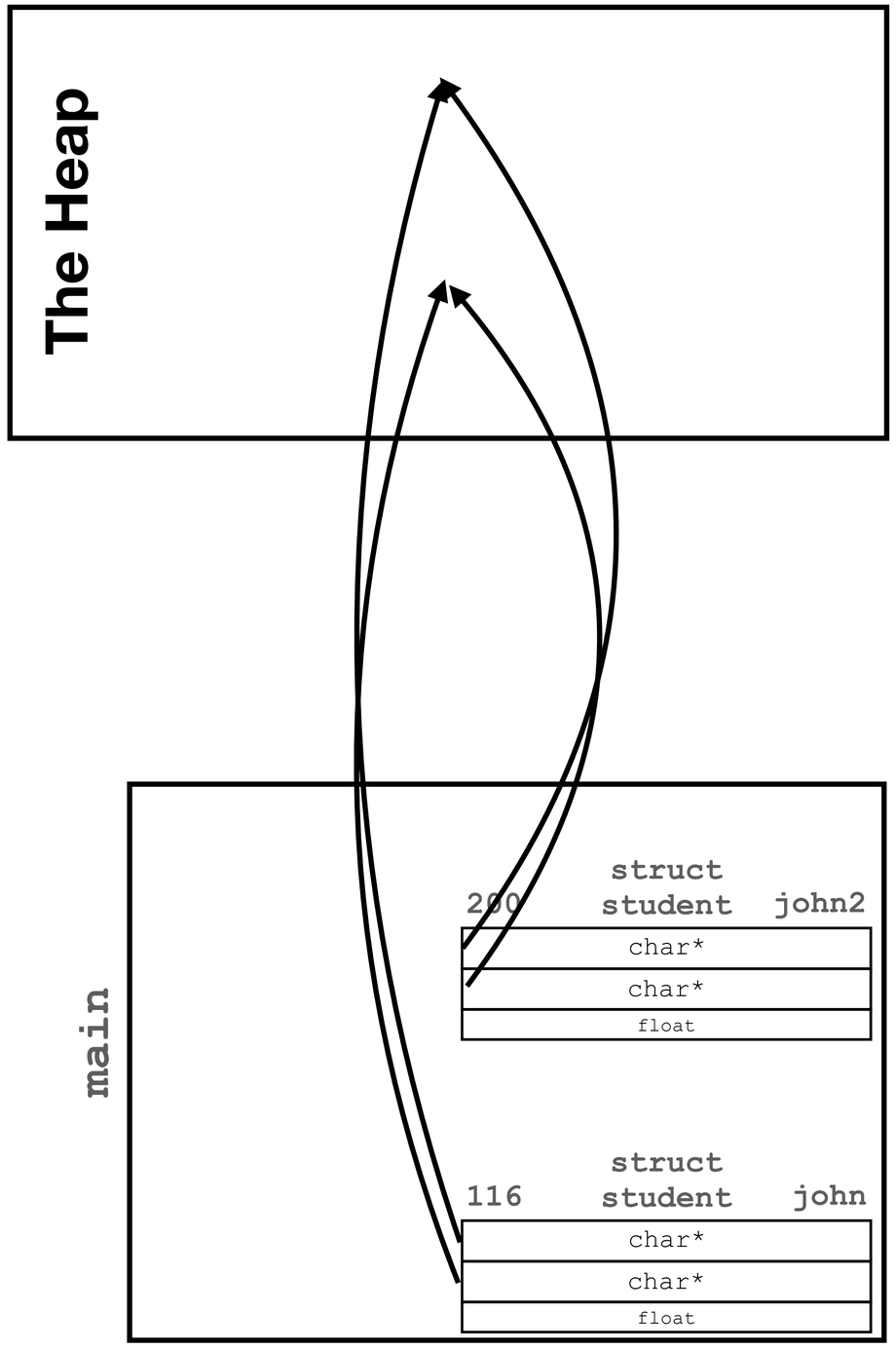
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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

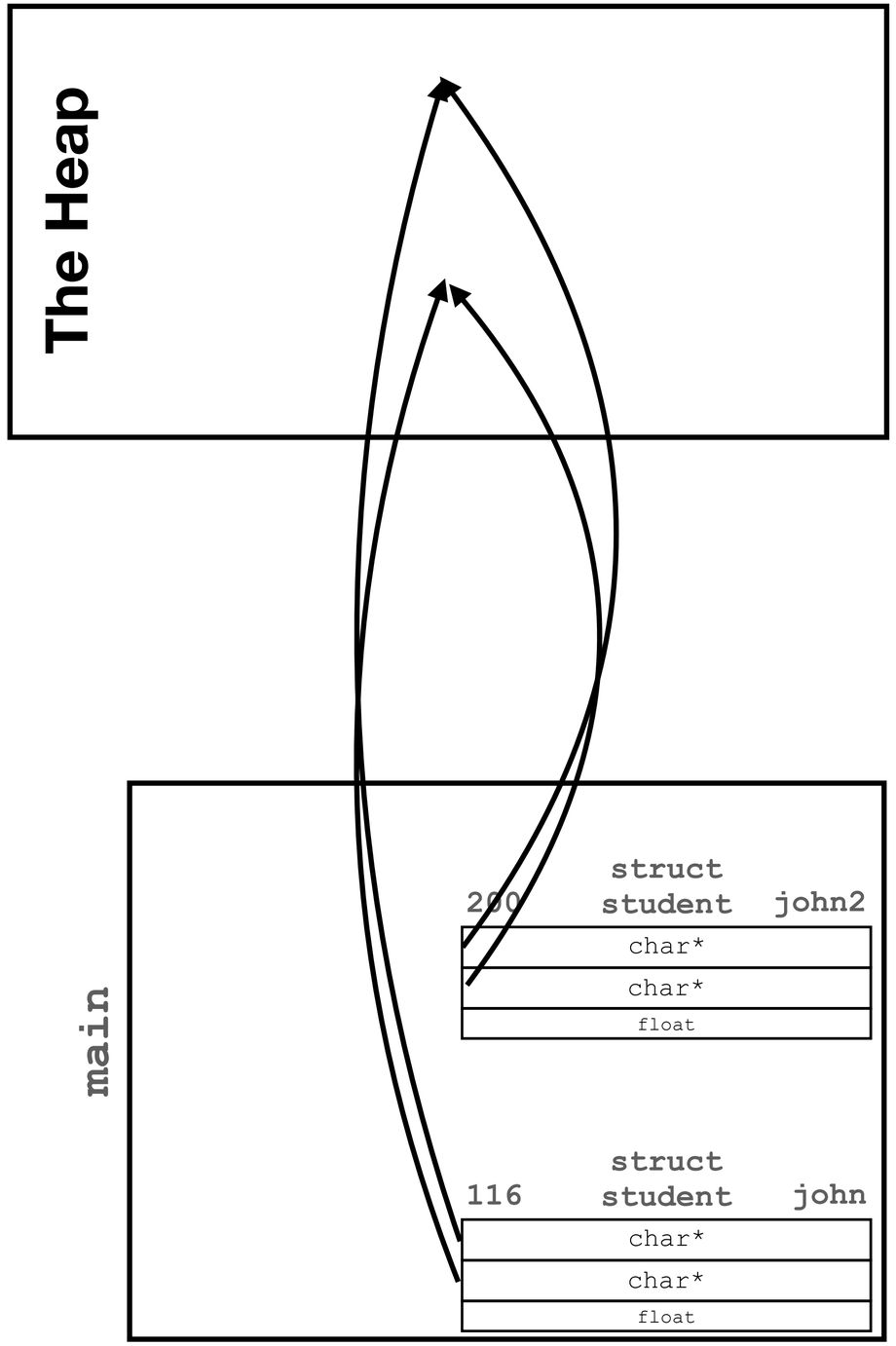
```
struct student {
        char *first_name;
        char *last_name;
        float gpa;
};

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;
}
```

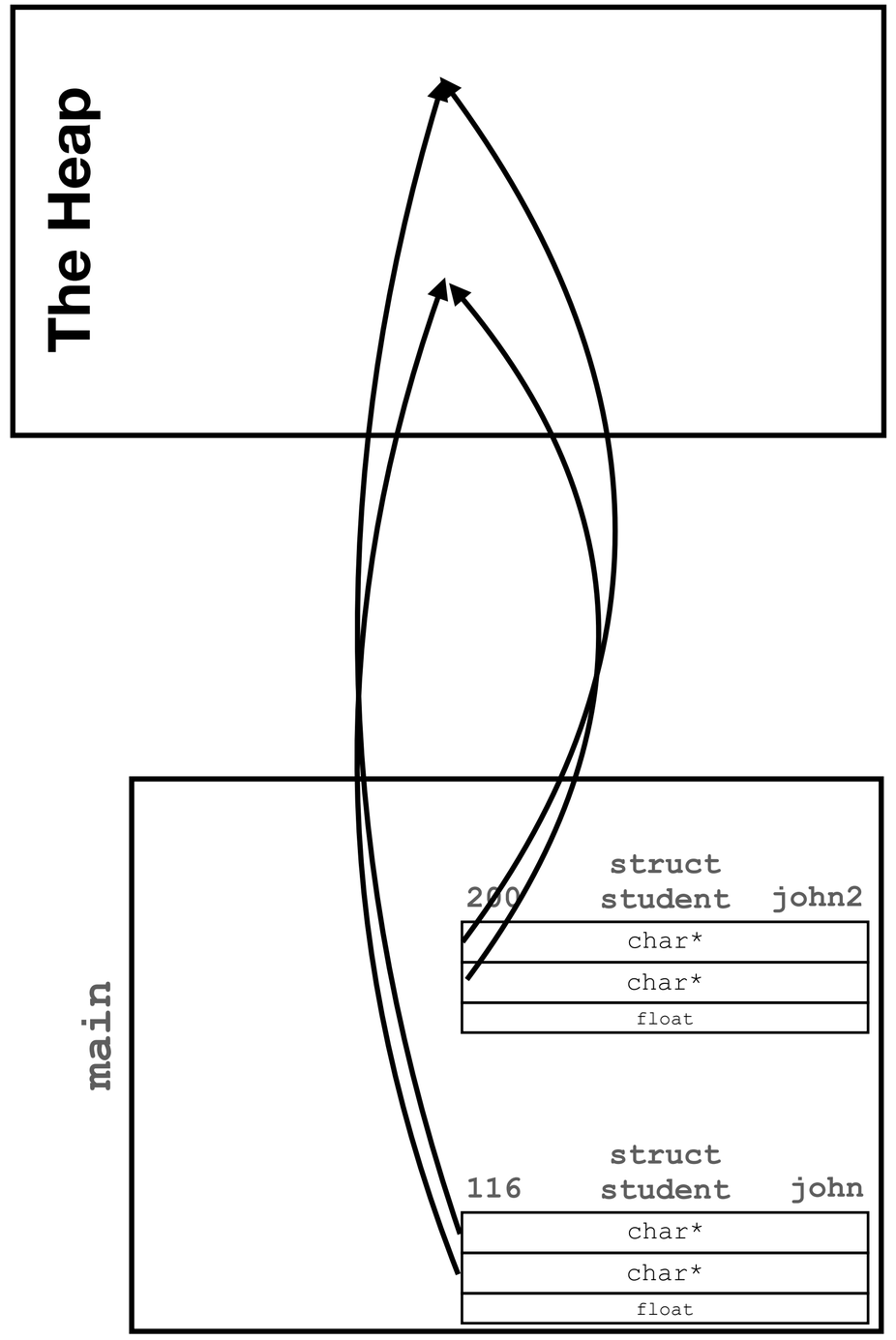


Pointer Hazards

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`

Review

Review

- Pointers: Syntax and meanings

Review

- Pointers: Syntax and meanings
- Stack: Frames, pass-by-value, pass-by-reference

Review

- Pointers: Syntax and meanings
- Stack: Frames, pass-by-value, pass-by-reference
- Heap: `malloc` and `free`

Review

- Pointers: Syntax and meanings
- Stack: Frames, pass-by-value, pass-by-reference
- Heap: `malloc` and `free`

The Stack

The Heap

Review

- Pointers: Syntax and meanings
- Stack: Frames, pass-by-value, pass-by-reference
- Heap: `malloc` and `free`
- Building structures in the heap via pointers

The Stack

The Heap

Review

- Pointers: Syntax and meanings
- Stack: Frames, pass-by-value, pass-by-reference
- Heap: `malloc` and `free`
- Building structures in the heap via pointers
  - Data structures!

The Stack

The Heap

Review

- Pointers: Syntax and meanings
- Stack: Frames, pass-by-value, pass-by-reference
- Heap: `malloc` and `free`
- Building structures in the heap via pointers
  - Data structures!
  - This week and next week

The Stack

The Heap

List

What is a list?

What is a list?

- An ordered collection of elements is a list: [🍏, 🍎, 🍐, 🍊, 🍋]

What is a list?

- An ordered collection of elements is a list: [🍏, 🍎, 🍐, 🍊, 🍋]
  - Homogeneous list: all elements have the same type

What is a list?

- An ordered collection of elements is a list: [🍏, 🍎, 🍐, 🍊, 🍋]
  - Homogeneous list: all elements have the same type
  - Heterogeneous list: elements may have different type

What is a list?

- An ordered collection of elements is a list: [🍏, 🍎, 🍐, 🍊, 🍋]
  - Homogeneous list: all elements have the same type
  - Heterogeneous list: elements may have different type
  - Ordered: 0th, 1st, 2nd, 3rd, ... (does not mean sorted)

What is a list?

- An ordered collection of elements is a list: [🍏, 🍎, 🍐, 🍊, 🍋]
  - Homogeneous list: all elements have the same type
  - Heterogeneous list: elements may have different type
  - Ordered: 0th, 1st, 2nd, 3rd, ... (does not mean sorted)
- Dynamic size: can grow/shrink as needed

What is a list?

- An ordered collection of elements is a list: [🍏, 🍎, 🍐, 🍊, 🍋]
  - Homogeneous list: all elements have the same type
  - Heterogeneous list: elements may have different type
  - Ordered: 0th, 1st, 2nd, 3rd, ... (does not mean sorted)
- Dynamic size: can grow/shrink as needed
- What operations does a list support?

List Operations

List Operations

- `append`

List Operations

- `append`
- `prepend`

List Operations

- `append`
- `prepend`
- `len`

List Operations

- `append`
- `prepend`
- `len`
- `insert_at`

List Operations

- `append`
- `prepend`
- `len`
- `insert_at`
- `remove_at`

List Operations

- `append`
- `prepend`
- `len`
- `insert_at`
- `remove_at`
- `is_empty?`

List Operations

- `append`
- `prepend`
- `len`
- `insert_at`
- `remove_at`
- `is_empty?`
- `at(i)`

Array as a List

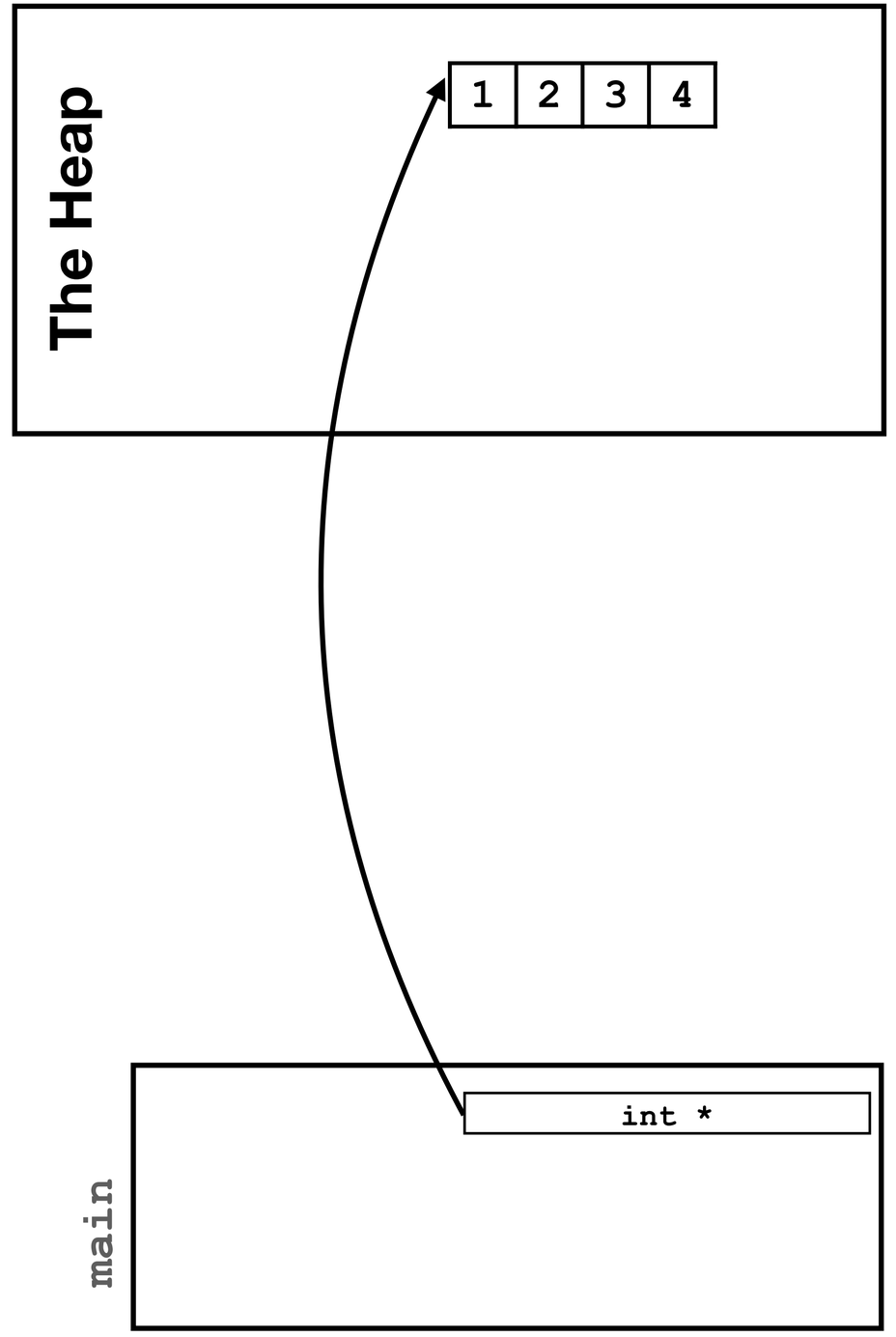
Array

- Can we just use an array to implement a list?
- Almost! Arrays have a fixed size, but ...
  - If it's on the stack, no luck here
  - If it's on the heap, we may have some ways around this

Array

Growing an array

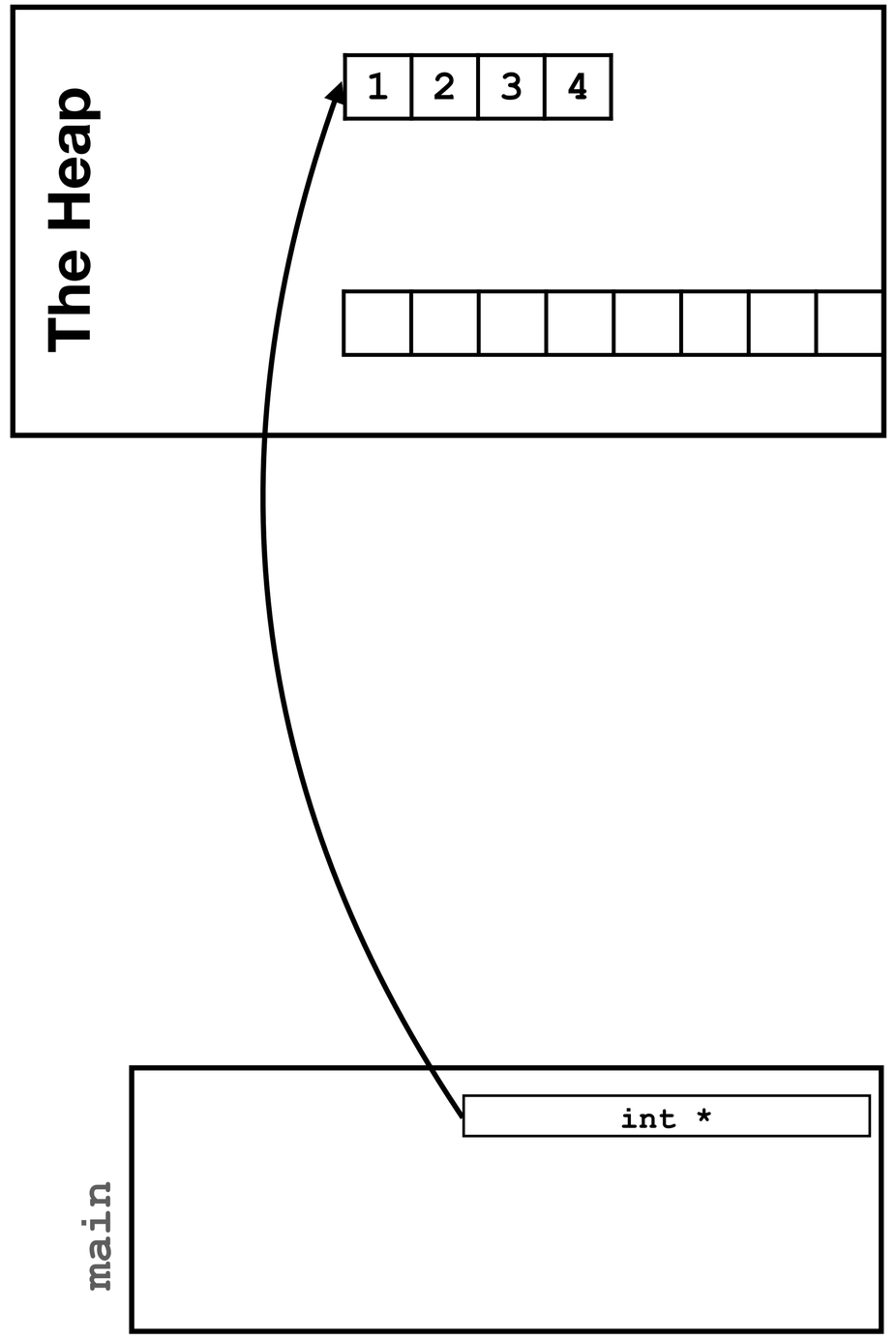
- Create a bigger array.



Array

Growing an array

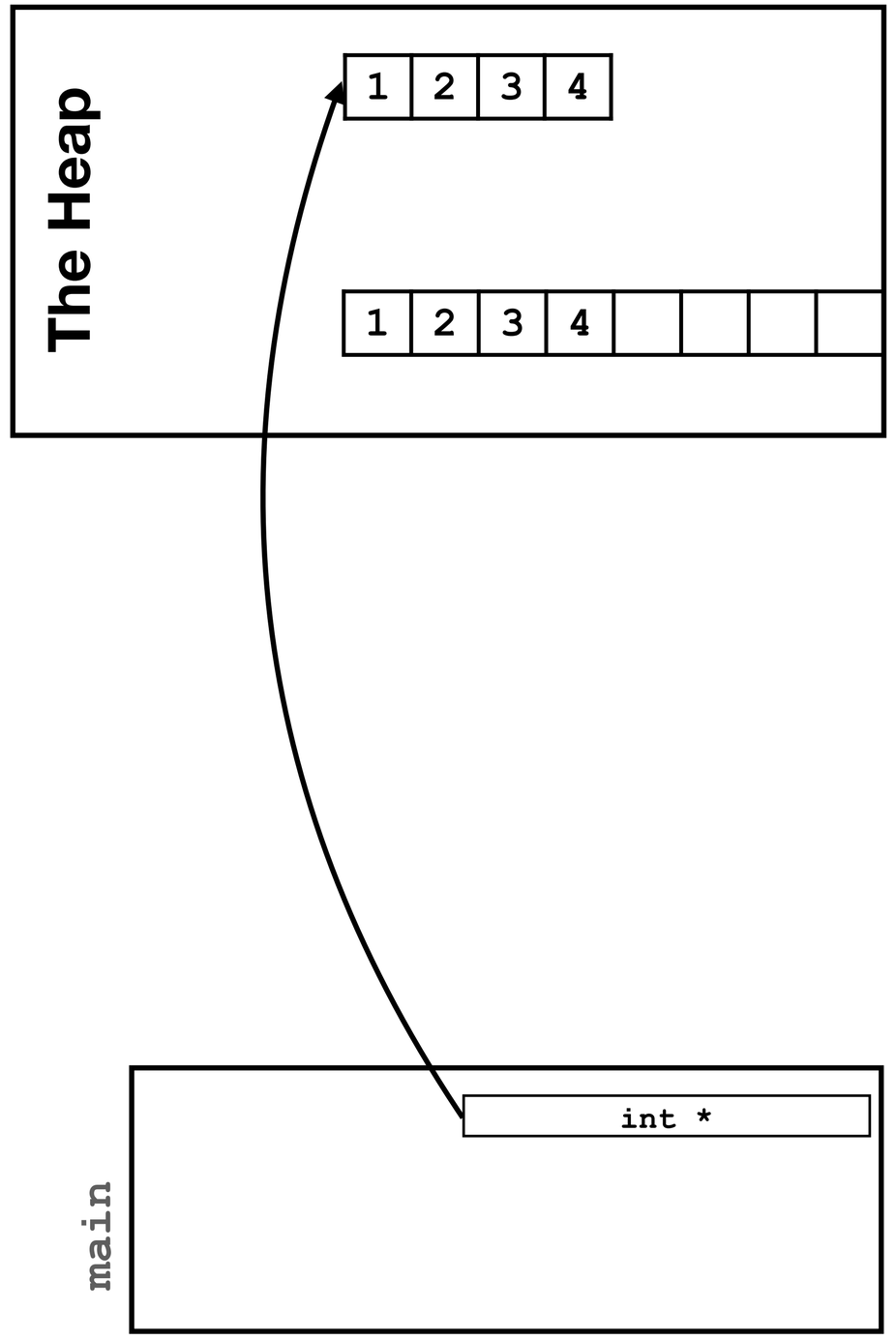
- Create a bigger array.



Array

Growing an array

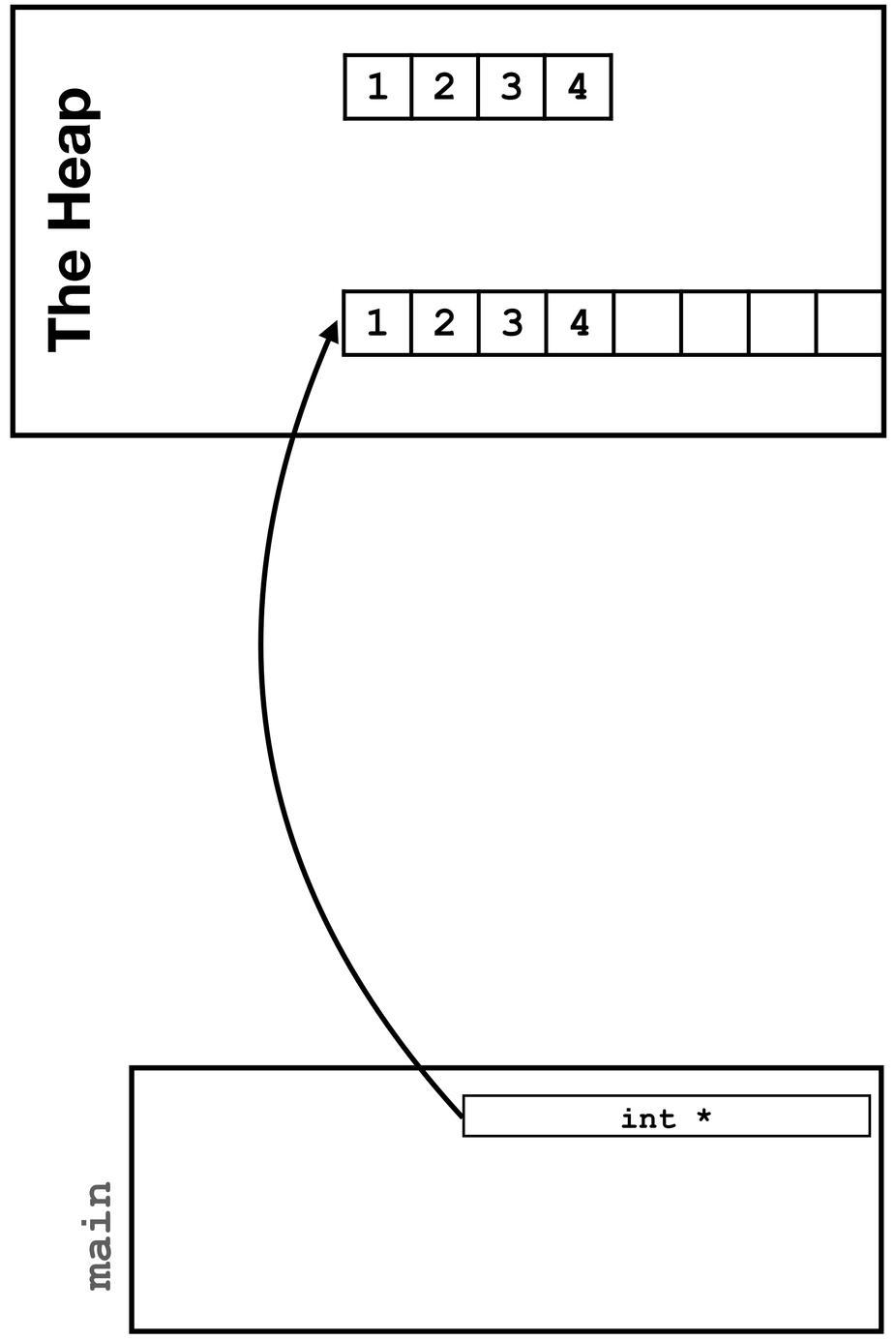
- Create a bigger array.
- Copy the elements into the new array



Array

Growing an array

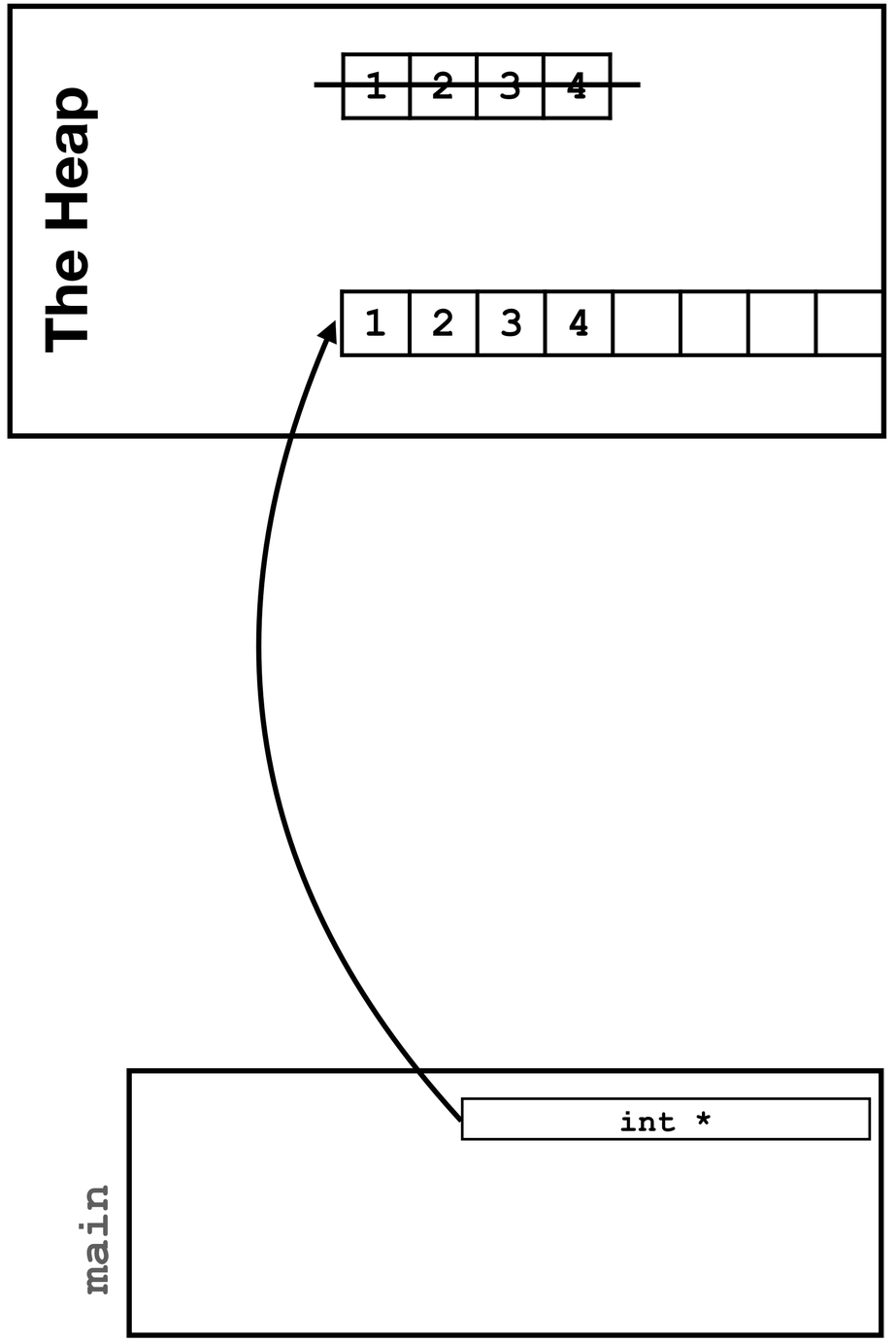
- Create a bigger array.
- Copy the elements into the new array
- Reassign the pointer to point to the new array



Array

Growing an array

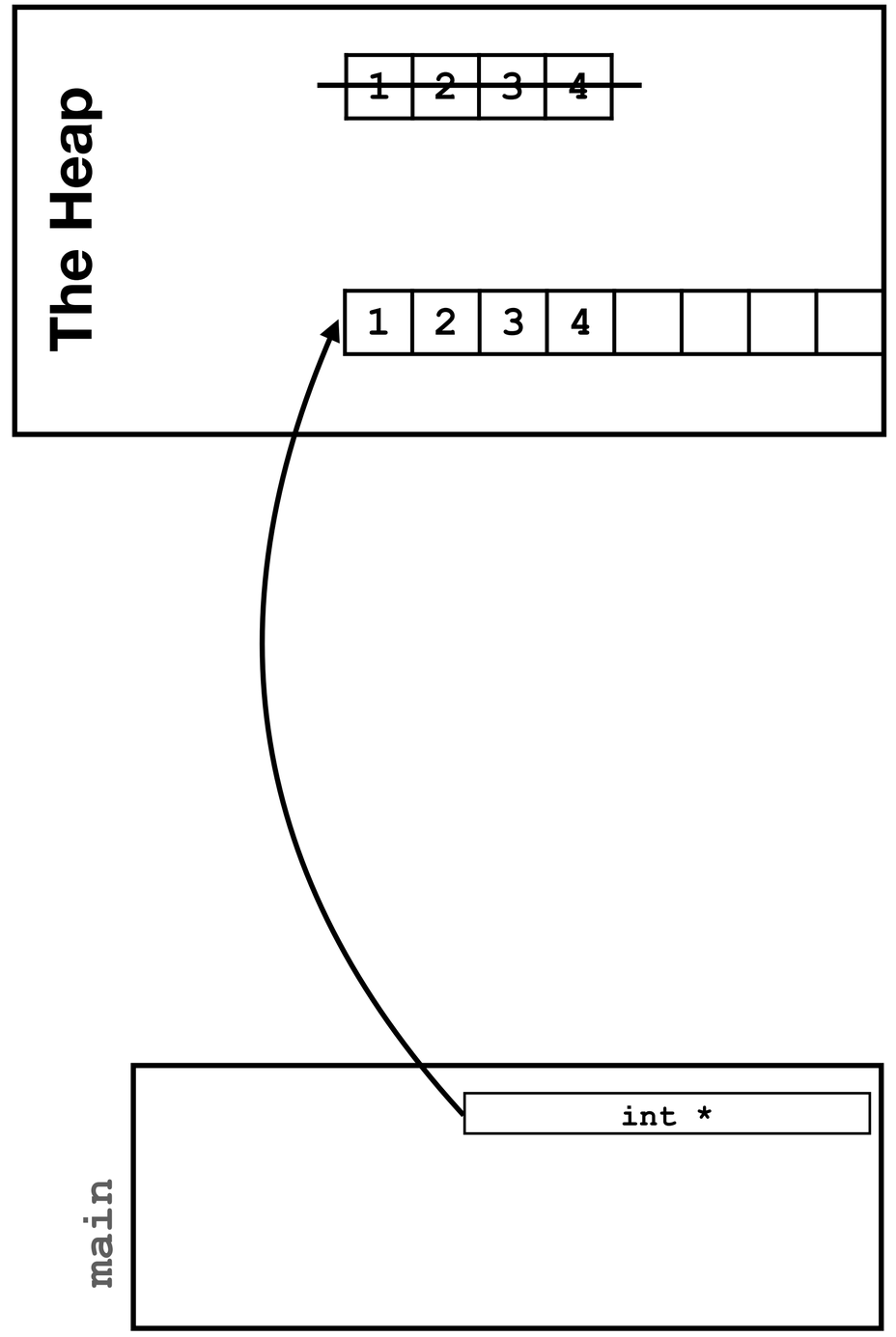
- Create a bigger array.
- Copy the elements into the new array
- Reassign the pointer to point to the new array
- Free the old array



Array

Growing an array

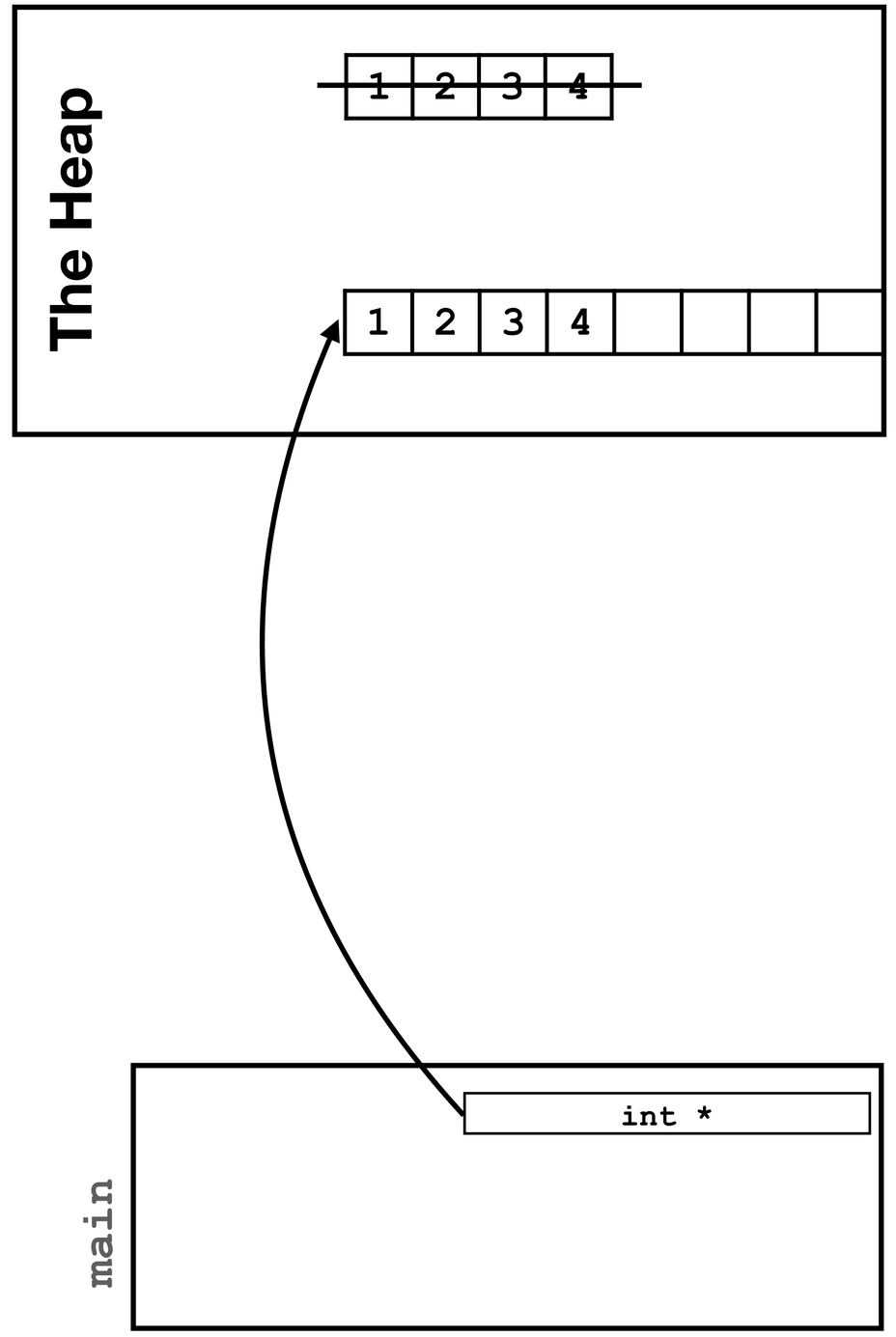
- Pointers serve as an indirection.
  - We aren't changing the size of the array; we are changing which array the pointers point to.
  - By changing the address of the pointer, it seems to the user that we have changed the size of the array.
- We create and delete memory however we want thanks to the heap.



Array

Abstractions

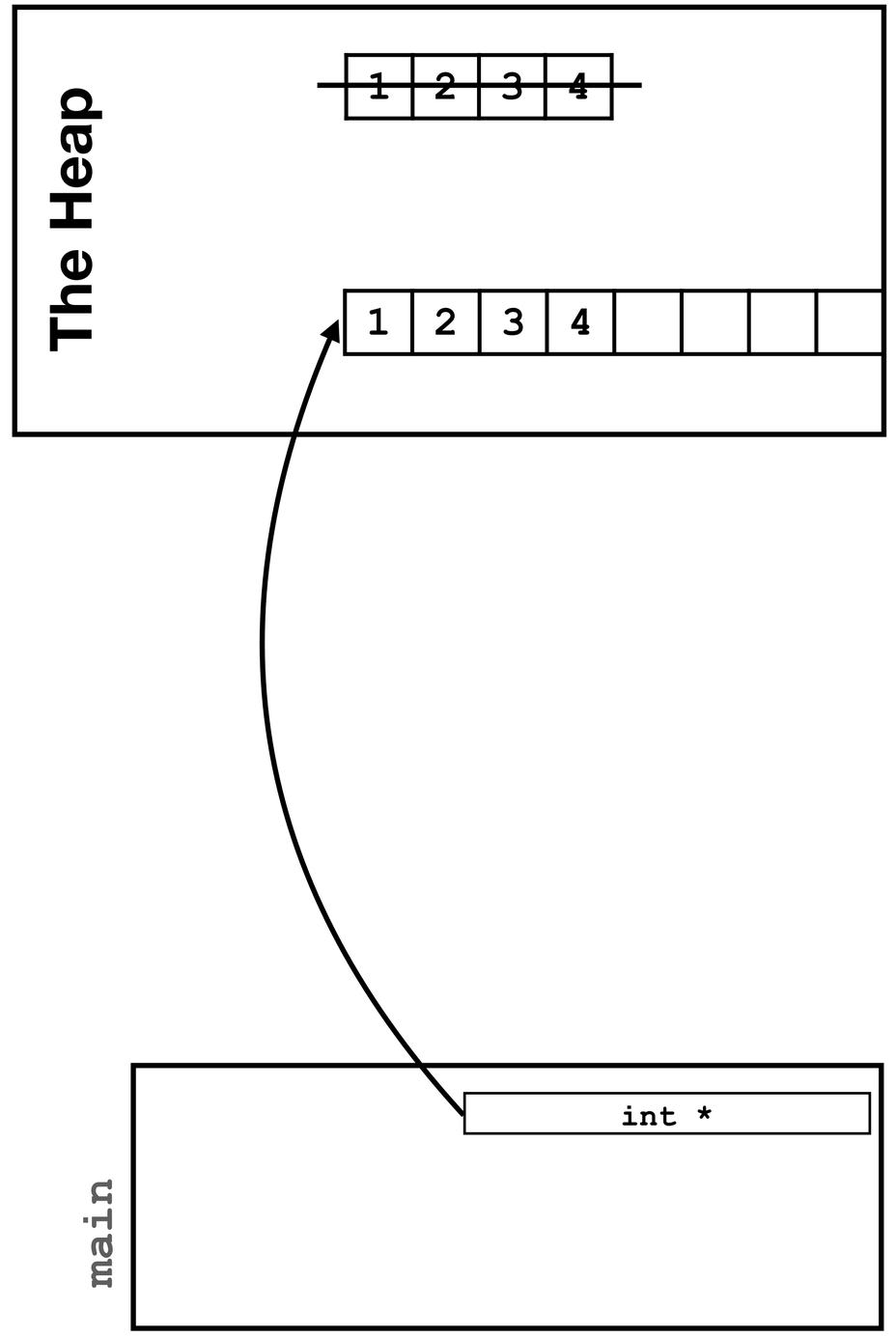
- Doing this repeatedly results in very messy code.
- Separation of concerns:
  - We wrap all of this up behind some interface
  - The user doesn't have to worry about any of that -- just adding and removing elements.



Array

Demo

- Let's write this together!



Array

Header File

Array

Header File

- When writing a library or module you need to write a **header** file that contains the function signatures and typ definitions needed

Array

Header File

- When writing a library or module you need to write a **header** file that contains the function signatures and typ definitions needed
- Python `import` = C `#include`

Array

Header File

- When writing a library or module you need to write a **header** file that contains the function signatures and typ definitions needed
- Python `import` = C `#include`
- `#include <stdio.h>` for standard library headers

Array

Header File

- When writing a library or module you need to write a **header** file that contains the function signatures and typ definitions needed
- Python `import` = C `#include`
- `#include <stdio.h>` for standard library headers
- `#include` "my_library.h" for project specific header filed

Array

Header File

- When writing a library or module you need to write a **header** file that contains the function signatures and typ definitions needed
- Python `import` = C `#include`
- `#include <stdio.h>` for standard library headers
- `#include` "my_library.h" for project specific header filed
- If a header file is included multiple times then functions will have multiple signatures

Array

ArrayList.h

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

typedef struct ArrayList_ {
    int *array;
    size_t length;
    size_t size;
} ArrayList;

ArrayList create(void);
void destroy(ArrayList array);
size_t length(const ArrayList array);
size_t size(const ArrayList array);
void append(ArrayList *array, const int elem);
void prepend(ArrayList *array, const int elem);
void insert_at(ArrayList *array, const int elem, const size_t i);
int remove_at(ArrayList *array, const size_t i);
int is_empty(const ArrayList array);
int at(const ArrayList array, const size_t i);
```

Array

ArrayList_example.c

```
#include <stdlib.h>

#include "ArrayList.h"
#include "ArrayList.h"

int main(void) {
    return EXIT_SUCCESS;
}
```

Array

ArrayList_example.c

```
clang -Wall -Wextra -pedantic -std=c11 -o ArrayList_example ArrayList.c ArrayList_example.c

In file included from ArrayList_example.c:4:
./ArrayList.h:23:16: error: redefinition of 'ArrayList_'
typedef struct ArrayList_ {
               ^
ArrayList_example.c:3:10: note: './ArrayList.h' included multiple times, additional include site here
#include "ArrayList.h"
         ^
ArrayList_example.c:4:10: note: './ArrayList.h' included multiple times, additional include site here
#include "ArrayList.h"
         ^
./ArrayList.h:23:16: note: unguarded header; consider using #ifdef guards or #pragma once
typedef struct ArrayList_ {
               ^
In file included from ArrayList_example.c:4:
./ArrayList.h:27:3: error: typedef redefinition with different types ('struct (unnamed struct at ./ArrayList.h:23:16)' vs 'struct
ArrayList_')
} ArrayList;
  ^
ArrayList_example.c:3:10: note: './ArrayList.h' included multiple times, additional include site here
#include "ArrayList.h"
         ^
ArrayList_example.c:4:10: note: './ArrayList.h' included multiple times, additional include site here
#include "ArrayList.h"
         ^
./ArrayList.h:27:3: note: unguarded header; consider using #ifdef guards or #pragma once
} ArrayList;
  ^
2 errors generated.
```

Array

ArrayList.h

`#pragma once` Preprocessor command "include this file only once"

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

typedef struct ArrayList_ {
    int *array;
    size_t length;
    size_t size;
} ArrayList;

ArrayList create(void);
void destroy(ArrayList array);
size_t length(const ArrayList array);
size_t size(const ArrayList array);
void append(ArrayList *array, const int elem);
void prepend(ArrayList *array, const int elem);
void insert_at(ArrayList *array, const int elem, const size_t i);
int remove_at(ArrayList *array, const size_t i);
int is_empty(const ArrayList array);
int at(const ArrayList array, const size_t i);
```

Array

ArrayList.c - create

Array

ArrayList.c - create

- When declaring a new `ArrayList`, `array`, `length` and `size` will be allocated in the stack

Array

ArrayList.c - create

- When declaring a new `ArrayList`, `array`, `length` and `size` will be allocated in the stack
- But `*array` will not be any valid address → need to `malloc`

Array

ArrayList.c - create

- When declaring a new `ArrayList`, `array`, `length` and `size` will be allocated in the stack
- But `*array` will not be any valid address → need to `malloc`

```
// Create a new ArrayList
ArrayList create(void) {
    ArrayList array;
    array.array = malloc(4 * sizeof(*array.array));
    if (!array.array) {
        perror("Unable to allocate memory");
        abort();
    }
    array.length = 0;
    array.size = 4;

    return array;
}
```

Array

ArrayList.c - destroy

Array

ArrayList.c - destroy

- For every `malloc`, there needs to be a `free`

Array

ArrayList.c - destroy

- For every `malloc`, there needs to be a `free`

```
// Destroy an ArrayList
void destroy(ArrayList array) {
    free(array.array);
}
```

Array

ArrayList.c - length and size

Array

ArrayList.c - length and size

- Users should not be directly interacting with stuct members

Array

ArrayList.c - length and size

- Users should not be directly interacting with stuct members
- Instead we use **accessors**

Array

ArrayList.c - length and size

- Users should not be directly interacting with stuct members
- Instead we use **accessors**

```
// Takes in an ArrayList and returns the number of elements
size_t length(const ArrayList array) {
    return array.length;
}

// Takes in an ArrayList and returns its maximum size
size_t size(const ArrayList array) {
    return array.size;
}
```

Array

ArrayList.c - insert_at

Array

ArrayList.c - insert_at

```
// Inserts elem at array[i] = v
// insert_at(&array, v, i)
void insert_at(ArrayList *array, const int elem, const size_t i) {
    // If position is greater than length, show error
    if (i > array->length) {
        fprintf(stderr, "Attempting to insert at position %lu of an ArrayList of length %lu", i, array->length);
        abort();
    }

    // Do I need to reallocate
    if (array->length == array->size) {
        array->array = realloc(array->array, 2 * array->size * sizeof(*array->array));
        if (!array->array) {
            perror("Unable to allocate memory");
            abort();
        }
        array->size = 2 * array->size;
    }

    // Copy over all following elements
    for (size_t j = array->length; j > i; j--) {
        array->array[j] = array->array[j - 1];
    }

    // Assign element
    array->array[i] = elem;
    array->length++;
}
```

Array

ArrayList.c - append and prepend

Array

ArrayList.c - append and prepend

- We can use `insert_at` to implement `append` and `prepend`

Array

ArrayList.c - append and prepend

- We can use `insert_at` to implement `append` and `prepend`

```
void append(ArrayList *array, const int elem) {
    insert_at(array, elem, array->length);
}

void prepend(ArrayList *array, const int elem) {
    insert_at(array, elem, 0);
}
```