C main header:

- `int main(int argc, char** argv)`
  - `argc` is the argument count
  - `argv` is argument values
    - An array of strings
- In java:
  - `public static void main(String[] args)`

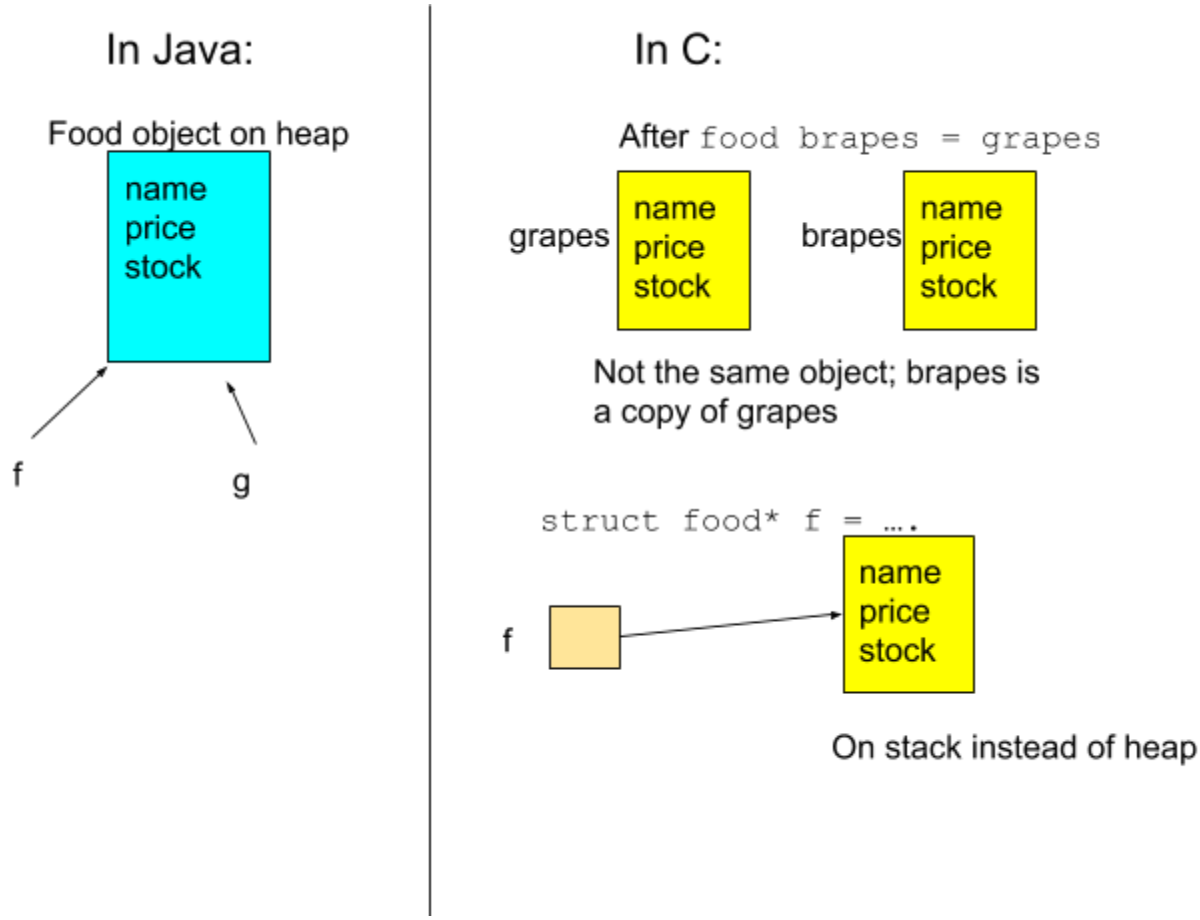
Command line arguments:

- `argv[0]` is always the name of the user who is running the executable
  - `$ ./myprogram one two three`
- Split on white space, each item becomes an element in the `argv` array
  - `argv[0] = ./myprogram`
  - `argv[1] = one`
- Arguments really start at `argv[1]`
- Command line file name comes in as a string, it's up to the program to fopen it
- `argv += 1`
  - Moves the pointer, now you are at `argv[1]`

Structs

- Class's ancestor
  - A C struct is like a class without most of the features
  - You can put data in it and that's it
  - `struct food {`
    - `char name[10]`
    - `double price;`
    - `int stock;`
  - `};`
  - You have to put the semicolon after the close brace
- To make a variable out of it:
  - `struct food grapes;`
- In Java:
  - Use `new` keyword to get a new object
  - `food f = new food;`
  - `new` always allocates it on the heap
  - `f` is a pointer, every object is a pointer in Java
    - Which is why it can be set to `null`
    - `food g = f`
      - `g` is also a pointer that points to the same object on the heap
  - `new is a reference type`
- In C:
  - `grapes` is not a pointer inside the variable
  - `food brapes = grapes;`
    - Makes a copy of `grapes`
  - You can have reference types in C

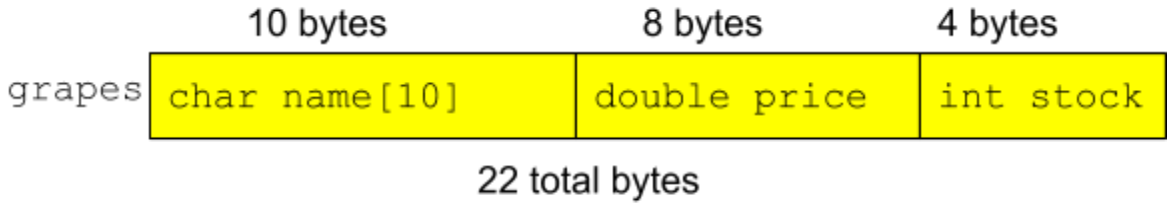
- `struct food* f = …`
- In the stack instead of the heap



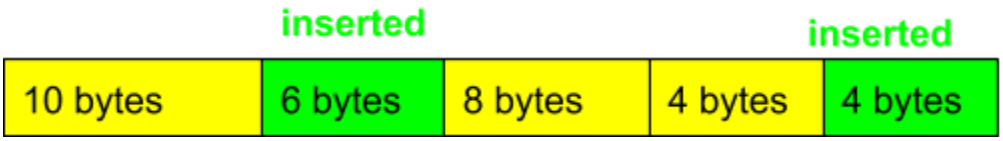
-

Memory representation

- All the struct fields are allocated inside the struct variable
- `struct food grapes;`



  -
  - 22 total bytes
  - But if you have `printf("sizeof(struct food) = %d\n", (int)sizeof(struct food));`
    - It will print that 32 is the size
    - Due to alignment and padding
- Alignment and padding
  - The C compiler is free to position your fields in memory any way

        - It will keep them in the same order
    - Typically it will align the fields using padding
    - `offsetof()` from `<studded.h>`
    - 

    - It inserted padding here (in green) to ensure the double field was aligned to 8 bytes
    - 4 byte values must appear at multiples of 4, doubles at multiples of 8, etc.
        - It's faster to access in single chunks
    - Padding at the end is to ensure the next struct is also aligned
        - Ensure the struct is a multiple of the maximum alignment of any field

Typedef struct

- `typedef struct food {`
    - `char name[10];`
    - `double price;`
    - `int stock;`
- `} food;`
    - Now you can write `food grapes;` to declare variables

Typedefs

- Way of making a type alias
    - Aliases: names that refer to the same thing
- A more convenient name for a type
- Syntax of a typedef is a variable declaration but with typedef in front
    - `typedef int x;`
        - Making a type named X
        - X is an alias for int
        - Can use X instead of int
        - `X x;`
            - Same as writing `int x;`
- `struct food {`
    - …
    - `} food, grapes, egg;`
        - This declares grapes and egg as variables of type food

Initializers

- `food grapes = {"grapes", 3.99, 20};`
- `food produce[] = {`
    - `{"grapes", 3.99, 20},`
    - `{"bananas", 0.89, 300},`
    - `("cucumbers", 1.49, 50}`
- `};`

- Automatically detects size
- Just declare variables in the way they are ordered in the struct
- ```
  {
  ```
  - ```
    .stock = 20;
    ```
  - ```
    .price = 0.89;
    ```
  - ```
    .name = "grapes"
    ```
  - ```
    }
    ```
- Field access operator
  - ```
    produce[0].price = 2.49;
    ```
  - ```
    food *pgrapes = &produce[0];
    ```
    - & lets you create a pointer
  - ```
    pgrapes.price = 2.99;
    ```
    - Error
    - If you use a . operator on a pointer to a struct, it doesn't work
  - ```
    pgrapes->price = 2.99;
    ```
    - This works
    - Use `->`

Passing and returning structs

- What if you try to copy the whole struct?
  - ```
    typedef struct Big{
    ```
    - ```
      int arr[256];
      ```
    - ```
      } Big;
      ```
  - ```
    void funct(Big thing) {}
    ```
  - …
  - ```
    Big big;
    ```
  - ```
    funct(big);     //has to copy 1 KB of data
    ```
  - Inefficient and slow
- Passing structs by reference
  - Use pointers
  - ```
    void funct(Big* thing) {}
    ```
  - …
  - ```
    Big big;
    ```
  - ```
    funct(&big) // copies size of 1 pointer
    ```
- ```
  void apply_discount(food f, double discount) {
  ```
  - ```
    f->price * = …
    ```
  - Any type can be passed by reference
  - When the function modifies the struct, the changes show up in the struct

Data structures with structs

- Linked lists, trees, etc, have "nodes" that point to each other
- To make a pointer inside the struct to the same type:
  - ```
    typedef struct Node{
    ```
    ```

```
    int value;
    struct Node* next;  // have to say struct Node
    } Node;
```

Enums

- Enum is a way of defining (usually related) constants
  - Think of choices
- ```
  typedef enum{
    red,
    orange,
    yellow,
    green,
    blue,
    purple
    } Color;
  ```
- Enums are NOT a new type
  - Structs are a new type
- Enums ARE ints
  - ```
    typedef int color;
    color c = 17;  //ok
    int x = red; //ok
    ```
  - The values just start at 0 and count up
    - Red = 0
    - Orange = 1, etc
    - Can assign specific values as well
- Three ways to define constants in C:
  - Enums:
    - ```
      enum{
        red,
        green,
        blue,
        };
      ```
  - Const
    - ```
      const int red = 0;
      const int green =1;
      ```
  - #define
    - ```
      #define red 0
      #define green 1
      ```
  - Why use enums vs. #define?
    - Enums indicate intent
      - When you choose to represent something in a certain way, you are communicating to others what you mean
      - Enums says: there are THREE possibilities for color, here they are
      - #define says: these are convenient names for these integer values
      ```

        - Enums go well with switch
            - But you need to handle every enum case
        - Enums can only hold ints

Binary files and structs

- `sizeof()` is a compile-time operator
    - Tells you how many bytes something takes up
    - `char carr[10]`
        - `sizeof(carr)` will be 10
    - `int iarr[10]`
        - `sizeof(iarr)` will be 40
    - `char * p = carr;`
        - C doesn't know how big an array at a pointer is
            - Gives you the size of the pointer itself
- Reading/writing binary files
    - `fread` reads data and puts it into a buffer you provide (like `fgets`)
        - Just copying bytes
    - `fread(&thing, sizeof(thing), 1, f);`
        - `fread(address, size, 1, file);`
    - `fwrite` takes data out of a buffer you provide and writes it
        - `fwrite(&thing, sizeof(thing), 1, f);`
    - Can do with any type, arrays, structs, int,
        - `int x = 34;`
        - `fwrite(&x, sizeof(x), 1, f);`
    - Never read or write pointers, doesn't make sense
    - They just copy blobs of bytes directly between memory and the file
        - `food grapes;`
        - `fread(&grapes, sizeof(grapes), 1, f);`
            - Copies all of the bytes of the file directly into the struct variable
            - fwrite works the same way, but copies FROM memory into the file