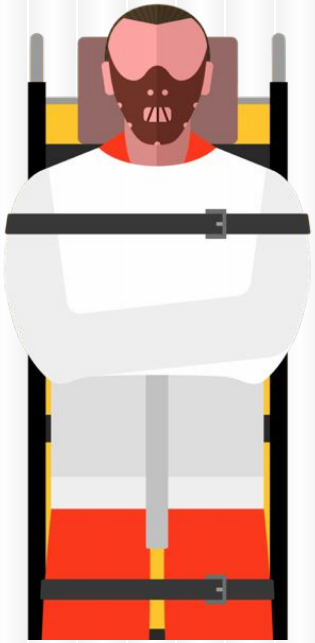
Structs, Unions, and Enums

CS580U - Fall 2017

Problem

- Define a Person in code:
  - char person_name[] = “Hannibal Lecter”;
    int person_age = 57;
    char person_hobbies[3][50] = {
    “Long Talks”,
    “Problem Solving”,
    “Foodie”};
- How do we organize this information?
  - This is the fundamental problem data structures tries to solve

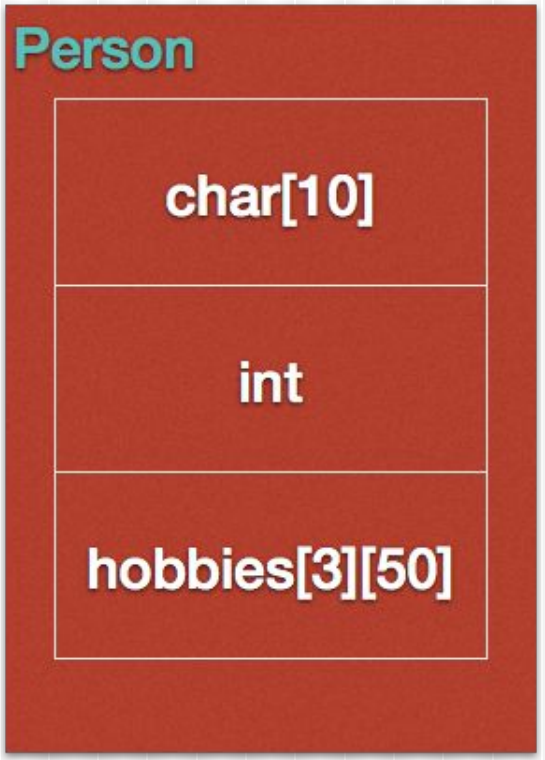


Structs

- User defined type
  - treated just like int, double, etc
  - Complex Type
- Compound data structure
  - Two ways to think about it
    - *A container for stuff*
    - *A multi-type array*

Declaring Structs

- syntax
  - struct <Name>{
    <data>
    };
- Example
  - struct Person{
    char name[10];
    int age;
    char hobbies[3][50];
    };



Initializing Structs

- Multiple ways to initialize a struct
  - struct Person hannibal;
    hannibal.name = “Hannibal”;
    hannibal.age = 57;
    hannibal.hobbies = hobby_list;
  - struct Person hannibal = {
    .age = 57,
    .name = “Hannibal”,
    .hobbies = hobby_list
    };
    - struct Person hannibal = {“Hannibal”, 57, hobby_list};
      - *this form should be considered deprecated*

Uninitialized Structs

- When you define a struct, if you do not give it any values, it is contains junk values
  - Once upon a time this was useful, less so today
- You can zero out a struct on definition by assigning a blank struct
  - struct Person hannibal = {}
    - *This makes sure that all values are zeroed out*

Inline Initialization

- Can initialize a struct inline (as needed)
  - struct {
    int i;
    char c;
    } my_struct; //declaration
  - struct {
    int i;
    char c;
    } my_struct = {6, ‘a’}; //definition

Structs in Memory

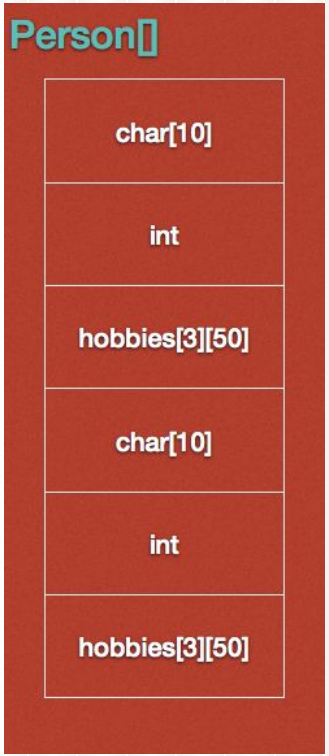
- A struct is a single data type, regardless of how many elements make it up
- Structs are basically a hash

people[0]

- .name
- .age
- .hobbies

people[1]

- .name
- .age
- .hobbies



Structs as Parameters

- Must use the ‘struct’ keyword
  - int escape(struct Person var);
  - the keyword is part of the type
    - *the type is ‘struct Person’*
- You can create a struct on the fly, a.k.a. a Compound Literal
  - When passing a compound literal, must typecast
    - *escape((struct Person){“Hannibal”, 57, hobby_list});*

Direct Definition of Struct Arrays

- Direct Definition:
  - struct Person people[2] = {
    {“Hannibal”, 57, hannibal_hobbies},
    {“Forrest”, 45, forrest_hobbies}
    };

-or-

- struct Person people[2] = {“Hannibal”, 57, hannibal_hobbies,“Forrest”, 45, forrest_hobbies};
  - Why does this work?

sizeof Struct

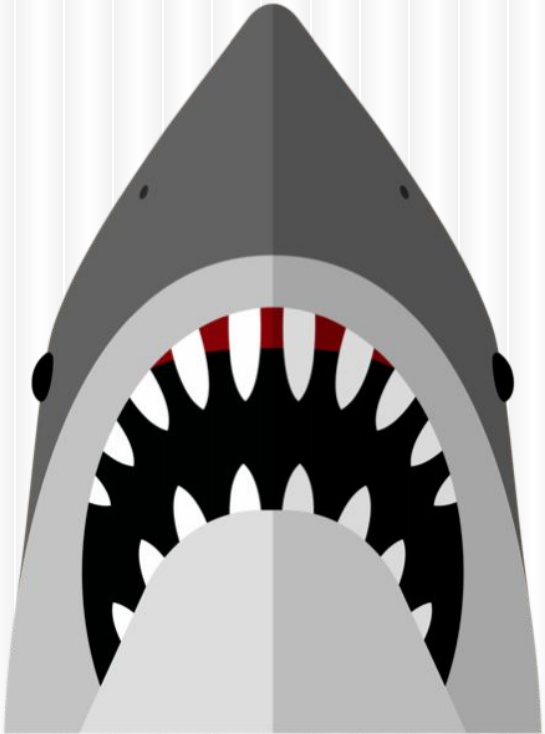
- You cannot count on the size of a struct
  - struct Stuff{
    int i;
    char c;
    }
    - *What size should this be?*
- Word Alignment
  - This is a way to simplify hardware
  - Pads out to the next CPU word according to the largest internal data type

it's bigger on the inside

Classwork: padding

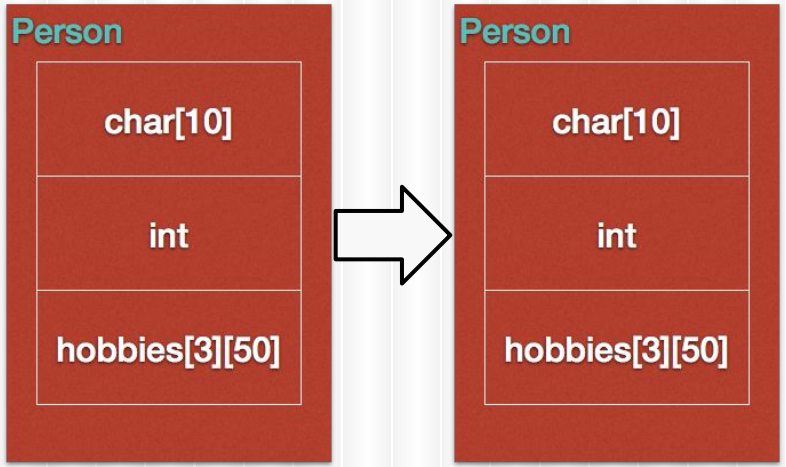
Structs-ception

- Structs can contain other structs
- Example:
  - *struct Food{*
    struct Person people[2];
    };
- Structs cannot contain themselves
  - Why not?
    - *Infinite structs-ception*
  - Workaround? Structs can contain pointers to themselves
    - Why does this work?



Struct Assignment

- Can assign structs to other structs
- Performs shallow copy
- Example:
  - struct Person forrest = hannibal;
- Copies all values over, why?
  - Structs are a single unit of data, like any other data type
  - variable names are just keys into the ‘hash’



Typedef

- What is the declaration if I need an unsigned long integer value
  - unsigned long int num;
- Can I shorten that?
  - Use typedef
- Allows you to give a additional name for an existing type.
  - new names are just an alias for some other type
- Improves clarity of the program with additional benefits

Using Typedef

- Syntax
  - typedef <type> <alias>;
- To shorten a type name
  - typedef unsigned long int uint64;
- To shorten user types
  - typedef enum months month_t;

Dropping the struct Keyword

- Let's get rid of the 'struct' keyword
  - structs are just user defined types, right?
  - use typedef to create an alias
  - Example:
    - *typedef struct {*
      int i;
      char c;
      } MyType;

  - usage: MyType var;

Struct-ception 2: Typedef

Example:

```
typedef struct mytype{
        int i;
        struct mytype * var;
        char c;
} MyType;
//Have to refer to the struct the old way isnide the typedef
```

Freeing Structs

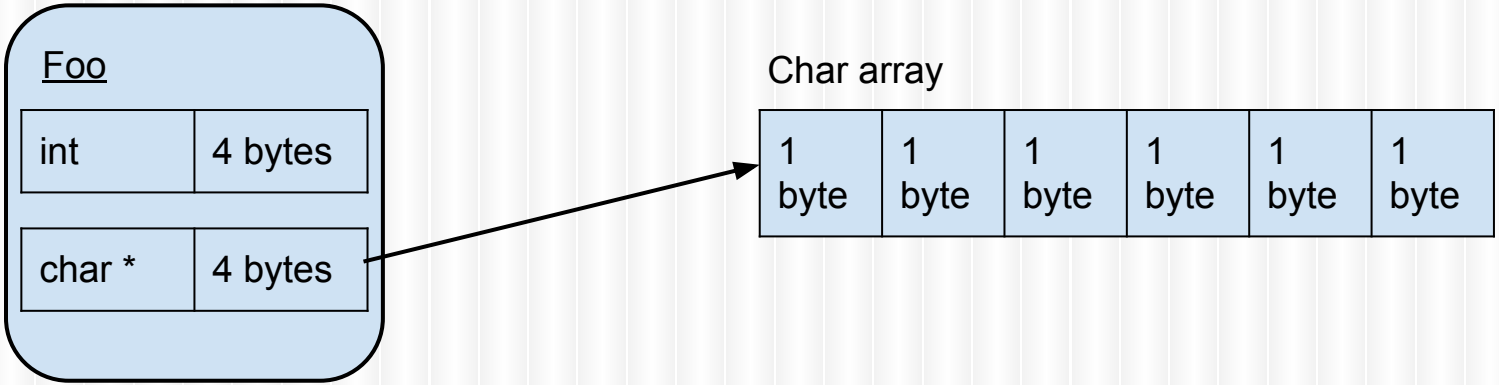
Given: *int = 4 bytes, char = 1 byte, * = 4 bytes*

- struct foo{
  int x;
  char * str;
  };
  …
  foo * f = malloc(sizeof(foo));
  f.str = malloc(sizeof(char) * 6);
  …
  free(f);

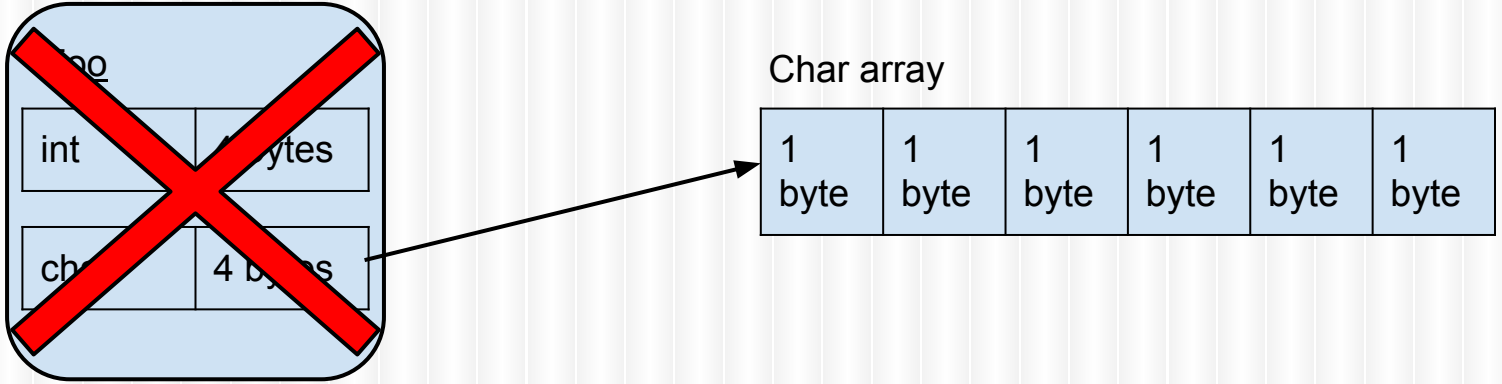
What is the size of foo?

What get deallocated?

Free Secondary Types

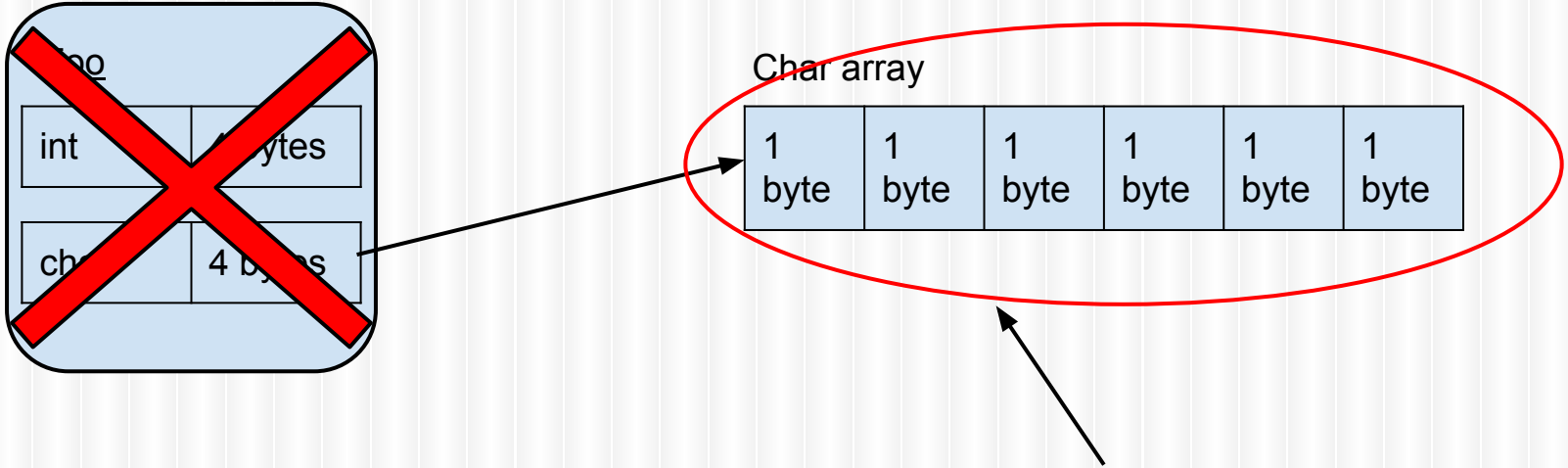


Free Secondary Types



…
free(f);
...

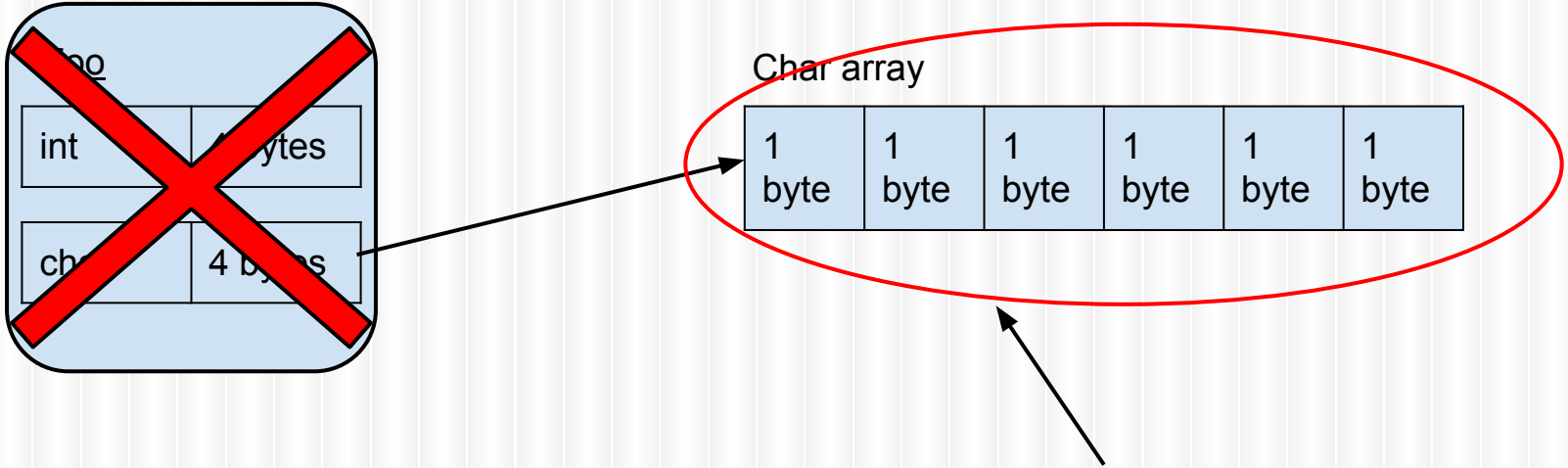
Free Secondary Types



…
free(f);
...

This still exists in memory. How can we access it?

Free Secondary Types

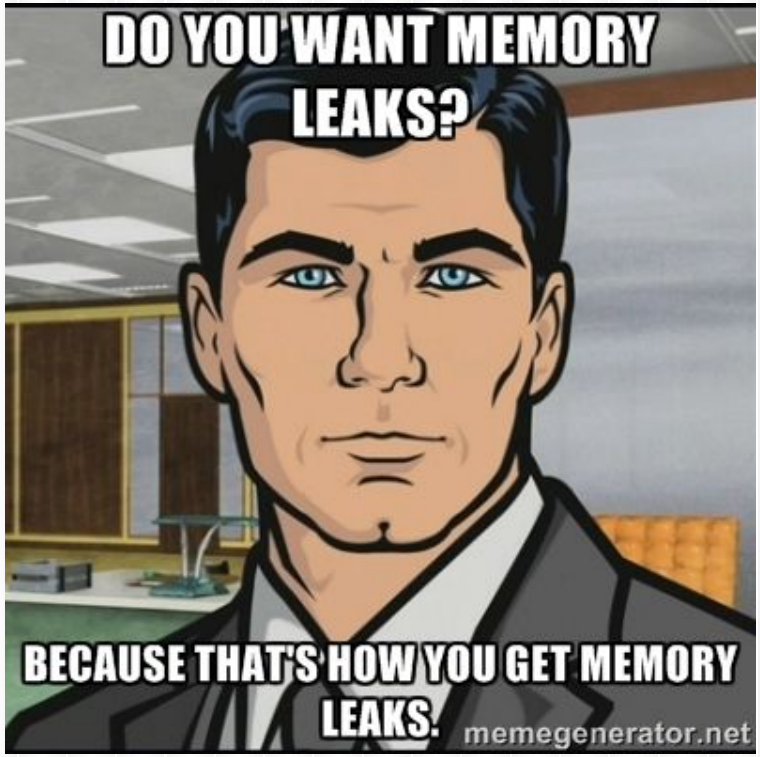


…
free(f);
...

We can’t. It is lost for the life of the program.

memory leaks

- A ***Memory Leak*** is when you free memory that contains the only address of dynamically allocated memory
- You must free the ‘deepest’ pointer to dynamically allocated memory first, then work your way ‘up’ the chain
- NULL pointers (if you nullify them properly) mean the memory has been freed
  - Remember, you can call free on *NULL* and it is a no-op (doesn’t do anything)



Structs as Classes

- Structs are not classes, they are compound data types.
  - However, we can easily modify how we use them to make them work more like classes.
- Every struct should have 3 accompanying functions:
  - new_<structname>
    - *Takes no paramaters*
    - *Returns a pointer to the initialized struct*
  - copy_<structname>
    - *Takes a pointer to another struct of the same type as a parameter*
    - *Returns a pointer to the copied struct*
  - free_<structname>
    - *Takes a pointer to a struct*
    - *Returns void*

Person Example

```
typedef struct Person{
	char * name;
	int age;
} Person;
Person * new_person(){
	Person * p = malloc(sizeof(Person));
	p->age = 0;
	p->name = NULL;
	return p;
}
void delete_person(Person * p){
		free(p->name);
		free(p);
}
Person * copy_person(Person * p){
		Person * copy = new_person();
		copy->name = malloc(strlen(p->name)+1); //Deep copy
		copy->age = p->age;
		strcpy(copy->name, p->name);
		return copy;
}
```

Function Pointers

- So far, we created some functions and a struct, but these aren't methods.
  - We need to have the functions belong to the struct
- We need function pointers
  - A function pointer is the address in memory of a function
  - A function's interface is also the functions type, and whenever we use an address in memory, we also need the type of the data stored at that address
    - *we declare function pointers with the following syntax:*
      - return (*name)(<parameters>)
  - We declares the function's 'type' when we declare the function pointer
    - *For example, if we wanted to get the address of a function: int foo(char *)*
      - int (*fp)(char *) = foo;

Struct Methods

- If we define function pointers within our struct for each function that ‘belongs’ to the struct, we can treat them as methods.
  - The exception to that is the new_ method. That becomes a bootstrapping problem and must stay outside the struct.
- We have now created a kind of namespacing, where we can use uniform names for copy, delete, and other functions belonging to the struct
  - We still have to use unique names for the functions, we just don’t have to remember them.

Person w/ Methods Example

```
typedef struct Person{
        char * name;
        int age;
        struct Person * (*copy)(struct Person *);
        void (*delete)(struct Person *);
} Person;

Person * new_person(){
        Person * p = malloc(sizeof(Person));
        p->age = 0;
        p->name = NULL;
        p->copy = copy_person;
        p->delete = delete_person;
}
```

Classwork: structs