User-defined types

Lecture 03.03

Outline

- struct
- struct pointers
- union
- enum

Programming fish store

```
/* Print out the catalog entry */
void catalog (char *name, char *species, int teeth, int age)
{
      printf ("%s is a %s with %i teeth. He is %i\n",
                                    name, species, teeth, age);
}


/* Print the label for the tank */
void label (char *name, char *species)
{
      printf ("Name:%s\nSpecies:%s\n,
                                    name, species);
}
```

Passing around multiple separate pieces of data

- We always need to think about the order of parameters
  - *What if we pass the values in a wrong order?*
- If we add more data about the fish, or remove the data:
  - *We need to update the code with more parameters*
- If we add 10 more pieces of data :
  - *We will have 10 more function arguments*

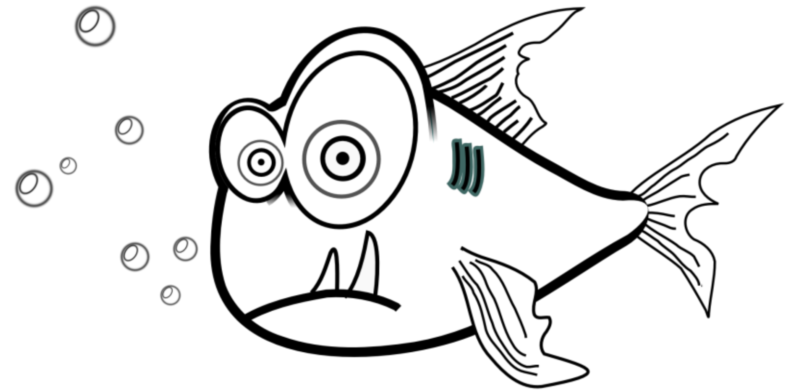
- We need to group the data and pass it as a single thing

Would an array work?

Create your own structured data type with a *struct*

Definition of a new data type

```
struct fish {
    const char *name;
    const char *species;
    int teeth;
    int age;
};
```



```
struct fish snappy = {"Snappy", "Piranha", 69, 4};
```

Variable of a new type

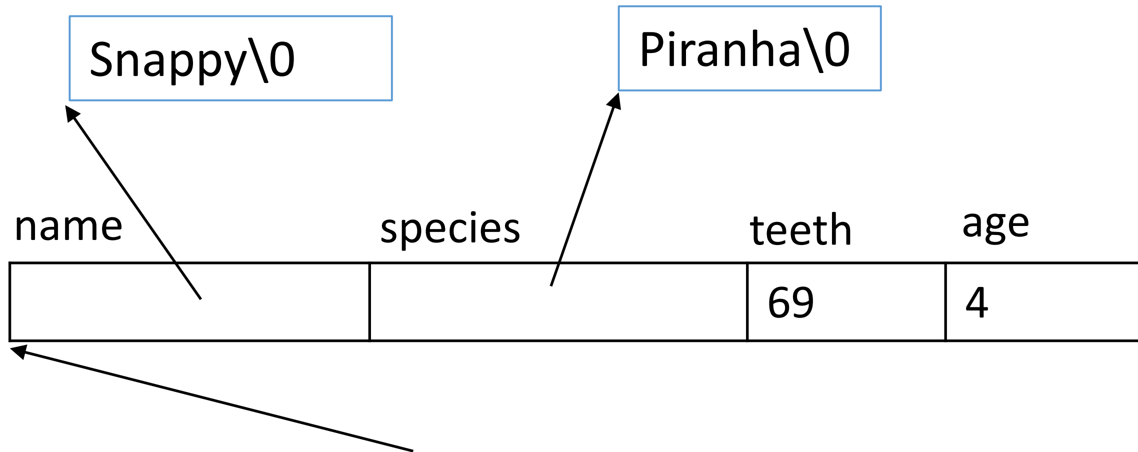
Passing struct as a parameter

```
int main() {
    catalog("Snappy", "Piranha", 69, 4);
    label("Snappy", "Piranha");
    return 0;
}
```

```
int main() {
    catalog (snappy);
    label (snappy);
    return 0;
}
```

Wrapping parameters in a struct makes your code more stable

Struct in memory: pointer fields



struct fish snappy = {"Snappy", "Piranha", 69, 4};

What segment of memory does char * name points to?
Can we update fish name?

```
struct fish {
    const char *name;
    const char *species;
    int teeth;
    int age;
};
```

Struct in memory: array fields

name	species	teeth	age
Snappy\0	Piranha\0	69	4

```
struct fish snappy = {"Snappy", "Piranha", 69, 4};
```

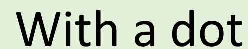
```
struct fish {
    char name[10];
    char species[10];
    int teeth;
    int age;
};
```

Can we update fish name now?

Reading struct fields

```
struct fish f = {"Snappy", "Piranha", 69, 4};

void label (struct fish f)
{
    printf ("Name:%s\nSpecies:%s\n,
                                    f.name, f.species);
}
```

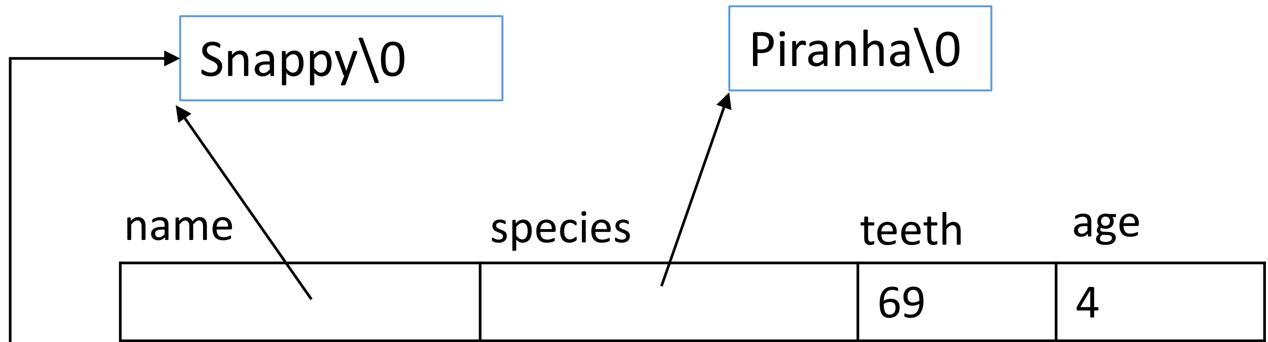


C: assignment copies data

- In languages like Java, if you assign an object to a variable, it doesn’t copy the object, it copies a reference

- In C, all assignments copy data
- If you want to copy a reference to a piece of data, you should assign a pointer

Assigning structs: by copy




```
struct fish snappy = {"Snappy", "Piranha", 69, 4};
struct fish guppy = snappy;
```

Copy of a pointer

Copy of a number

```
struct fish {
        const char *name;
        const char *species;
        int teeth;
        int age;
};
```

Summary:
Differences between array and struct

- Like an array, struct groups a number of pieces of data together
- An array variable is just an address of the first element of the array, while struct variable is a name for a variable itself, it has its own address
- In array you can access elements by index, in struct you can only access fields by name
- Struct is fixed length (no dynamic allocation)
- Struct may store data of different types

Nested structs

```
struct preferences {
    char *food;
    float exercise_hours;
};

struct fish {
    const char *name;
    const char *species;
    int teeth;
    int age;
    struct preferences care;
};
```

Reading nested structs

```
struct fish snappy = {"Snappy", "Piranha", 69, 4, {"Meat", 7.5}};
```

```
printf ("Snappy likes to eat %s", snappy.care.food);

printf("Snappy likes to exercise for %f hours",
                                        snappy.care.exercise_hours);
```

Give your new type a proper name using typedef

```
struct peter_parker {
...
};
struct peter_parker p;

typedef struct peter_parker{
    ...
} spider_man;

spider_man t;
```

We have to use word struct here

Name

Alias

Use as normal data type

Anonymous stucts

```
typedef struct {
        ...
} spider_man;
```

Memory alignment in structs

- In general, struct fields get placed next to each other in memory
- Sometimes computer adds small gaps between the fields, because it likes data to fit inside word boundaries
- That happens because during program execution complete words are read from the memory address: If a field was split across more than one word, the CPU would have to read several locations and somehow stitch the value together

Size of struct: Aligning to the closest int (32 bit): 1/2

```
typedef struct n6 {
   short n2;
   int n4;
}N6;


int main (int argc, char **argv){
        N6 number;
        printf("Size of <number> is %lu\n", sizeof (N6));
}
```

8

Aligning to the closest int: 2/2

```
typedef struct n5 {
    char n1;
    int n4;
}N5;

int main (int argc, char **argv){
        N5 number;
        printf("Size of <number> is %lu\n", sizeof (N5));
}
8
```

Structs as function parameters: 1/2

- Define a new data type: turtle

```
typedef struct {
    const char *name;
    const char *species;
    int age;
} turtle;
```

- Function argument is of type turtle

```
void happy_birthday (turtle t) {
    t.age = t.age + 1;
    printf("Happy Birthday %s! You are now %i years old!\n",
                                        t.name, t.age);
}
```

Structs as function parameters: 2/2

- Function argument is of type turtle

```
void happy_birthday (turtle t) {
    t.age = t.age + 1;
    printf("Happy Birthday %s! You are now %i years old!\n",
                                    t.name, t.age);
}
```

What is printed here?

- We call function with a variable of type turtle

```
int main() {
    turtle myrtle = {"Myrtle", "Leatherback sea turtle", 99};
    happy_birthday (myrtle);
    printf("%s's age is now %i\n", myrtle.name, myrtle.age);
    return 0;
}
```

And what is printed here?

What happened with myrtle?

Why it never gets older?

The code is cloning the turtle

```
void happy_birthday (turtle t) {
        t.age = t.age + 1;
        …
}
```

- The myrtle struct is copied to the parameter t
- Myrtle is the turtle that we are passing to the function
- Parameters are passed to functions by value: i.e. when you call a function, the values you pass into it are *assigned* to the parameters
- As if you had written:

```
turtle t = myrtle;
```

So what do we do if we want pass a struct to a function that needs to update it?

We need a pointer to the struct

- When you pass a variable to scanf(), you pass a pointer:
```
scanf("%d", &number);
```

- The same with structs. If you want a function to update a struct variable, you pass the address of the struct:

```
void happy_birthday (turtle *t) {
    ...
}

happy_birthday(&myrtle);
```

Changing age of myrtle by dereferencing the pointer

```
void happy_birthday (turtle *t) {
    (*t).age = (*t).age + 1;
    printf ("Happy Birthday %s! You are now %i years old!\n",
                                    (*t).name, (*t).age);
}
```

Dereferencing struct pointers

- Make sure that *t is always wrapped in parentheses:

(*t).age ≠ *t.age

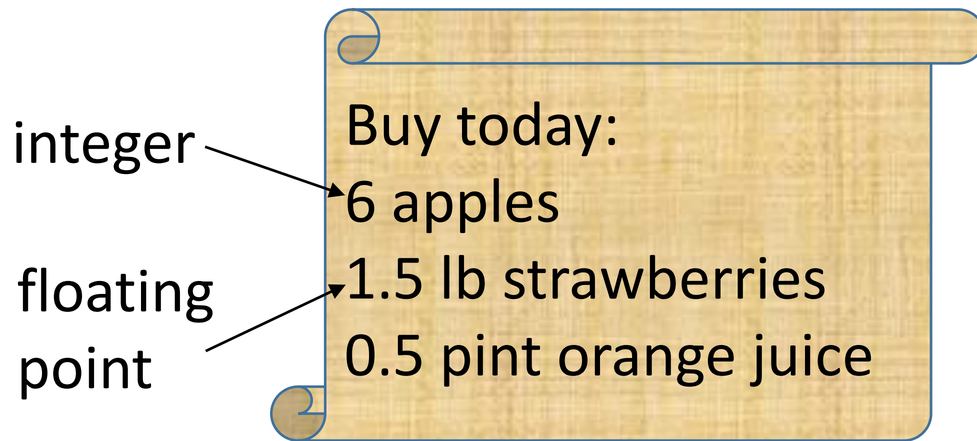
*t.age = *(t.age), because dot has precedence over *

*(t.age) = "the contents of the memory at address t.age." But t.age isn't a memory location! It's like dereferencing an *int* variable

(*t).age = t->age

- The -> notation saves on parentheses and makes the code more readable

Sometimes the same type of thing needs different types of data



How do we model quantity?

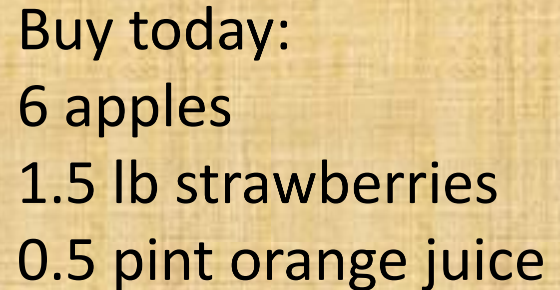
Sometimes the same type of thing needs different types of data

- Quantity might be a count, a weight, or a volume
- Create several fields with a struct and use the corresponding field when needed:

```
typedef struct {
    ...
    short count;
    float volume;
    ...
} fruit;
```

What is the problem with this model?

```
typedef struct {
    ...
    short count;
    float volume;
    ...
} fruit;
```



- It will waste memory space
- Someone might set more than one value
- There’s nothing called *quantity*

A union lets you reuse memory space

```
typedef struct {
    short count;
    float weight;
    float volume;
} quantity;
```

Memory space of the largest field (4 bytes)

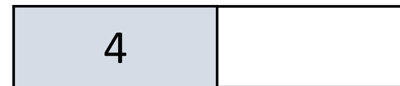
```
typedef union {
    short count;
    float weight;
    float volume;
} quantity;
```

union

- A union will use the space for just one of the fields in its definition
- Whether you set the count, weight, or volume field, the data will go into the same space in memory

```
typedef union {
    short count;
    float weight;
    float volume;
} quantity;

quantity q;
```



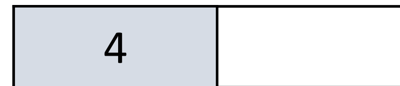
q.count = 4;

q.volume = 0.5;

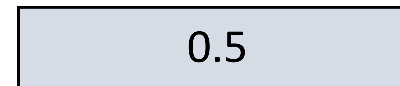
- Remember: with union there will only ever be **one** piece of data stored
- The union gives a way of creating a variable that supports several different data types
- You can interpret the same sequence of bits in a different way

```
typedef union {
        short count;
        float weight;
        float volume;
} quantity;

quantity q;
```



q.count = 4;



q.volume = 0.5;

unions are often used with structs

```
typedef union {
    short count;
    float weight;
    float volume;
} quantity;

typedef struct {
    char *name;
    char *country;
    quantity amount;
} fruit_order;
```

Using union with struct: example

```
fruit_order apples;
apples.name = “apples”;
apples.country = “Canada”;
apples.amount.count = 12;

fruit_order blueberries;
blueberries.name = “blueberries”;
blueberries.country = “Mexico”;
blueberries.amount.volume = 4.2;
```

Interpreting the value

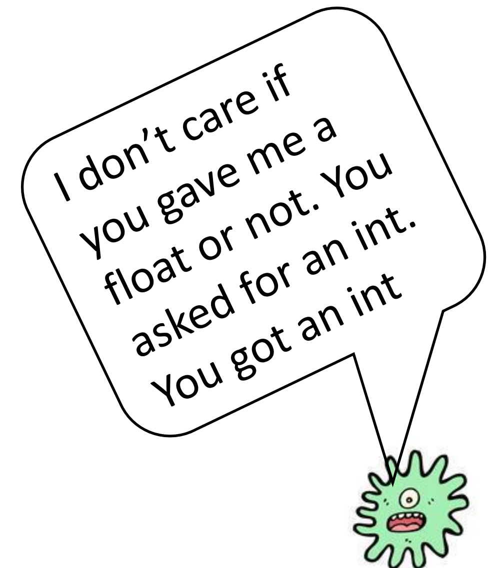
- We can store many possible values in a union, but no way of knowing what type it is once it's stored
- The compiler does not keep track of which fields are set and how to interpret them
- There's nothing to prevent us from setting one field and reading another

- Is that a problem? Sometimes it can be a BIG PROBLEM

That’s a lot of cupcakes!

```
typedef union {
	float weight;
	int count;
} cupcake;

int main() {
	cupcake order;
	order.weight = 2;
	printf("Cupcakes quantity: %i\n", order.count);
	return 0;
}
```




```
gcc badunion.c -o badunion && ./badunion
Cupcakes quantity: 1073741824
```

We need to keep track of the value types we’ve stored in a union **by ourselves**

One possible trick is to create an *enum*

enum

- Sometimes you want to store something from a predefined list of symbols:

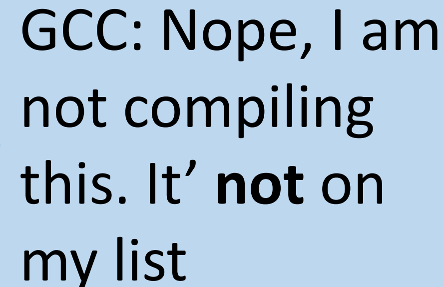
```
enum day_of_week {MONDAY, TUESDAY, WEDNESDAY,…};
enum colors {RED, GREEN, BLUE};
```

enums make your code more readable and prevent invalid values

```
enum colors {RED, GREEN, BLUE};
enum colors favorite = BLUE;
```

- Under the covers the enum will just store one of consecutive numbers
- In C code we can just refer to the symbols

favorite = *GLUE*;



Using enum to keep track of union values

```
typedef enum {
    COUNT, POUNDS, PINTS
} unit_of_measure;

typedef union {
    short count;
    float weight;
    float volume;
} quantity;
```

```
typedef struct {
    const char *name;
    const char *country;
    quantity amount;
    unit_of_measure units;
} fruit_order;
```

Example of union with enum

```
printf("This order contains ");
if ( order.units == PINTS)
    printf("%2.2f pints of %s\n", order.amount.volume ,
    order.name);
else if ( order.units == COUNT)
    printf("%i %s\n", order.amount.count , order.name);
…
```

```
This order contains 144 apples
This order contains 17.60 lbs of strawberries
This order contains 10.50 pints of orange juice
```

Recursive struct: island tours

```
typedef struct island {
    char *name;
    struct island * next;
} island;
```