What have we learned?

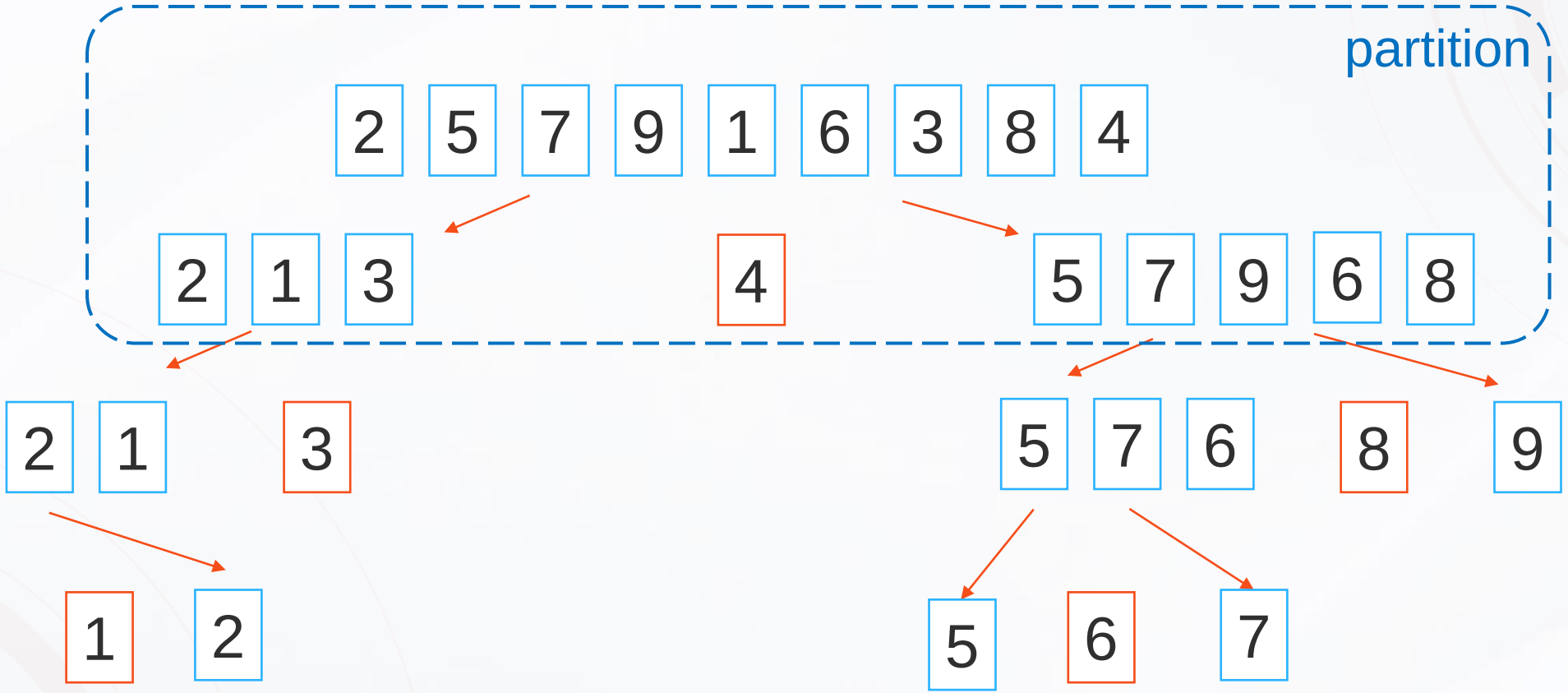
Functions in C can be recursive: *it may call itself.*

```
int fact(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * fact(n - 1);
}
```

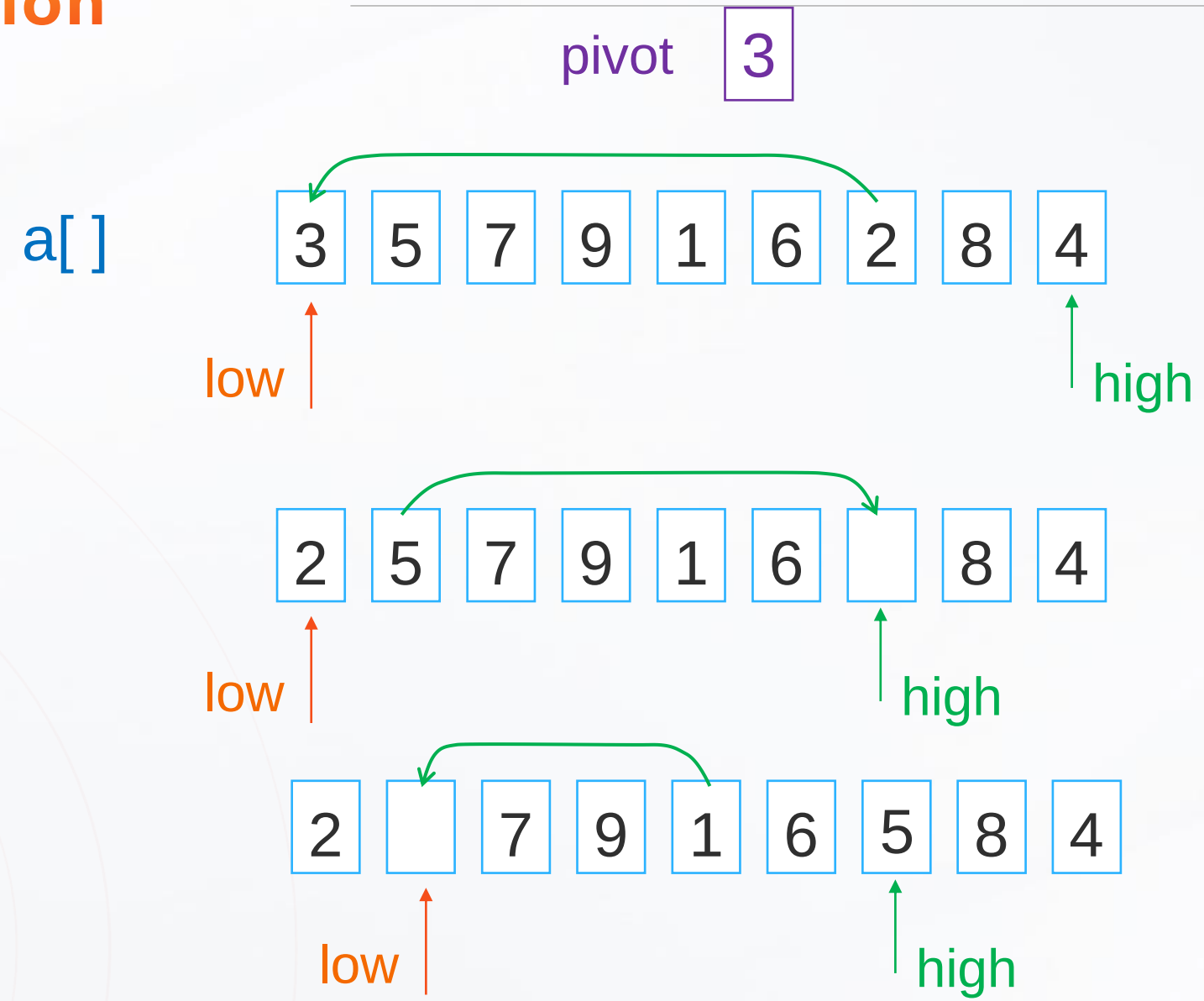
A termination condition is always needed!

Factorial: n! = 1*2*…*n = n*(n-1)!

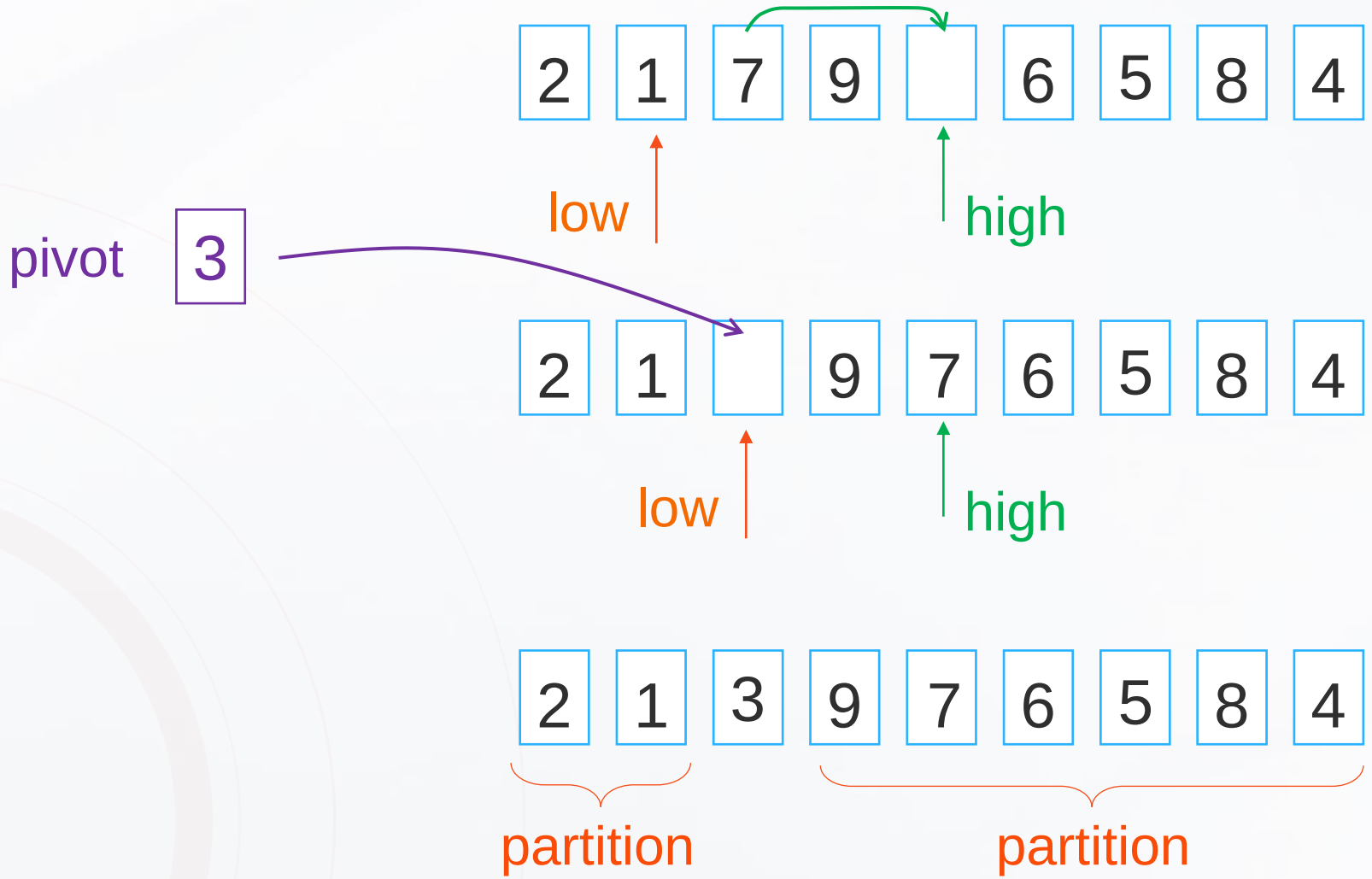
Quicksort



Partition


pivot
3
a[ ]
3 5 7 9 1 6 2 8 4
low
high
2 5 7 9 1 6 8 4
low
high
2 7 9 1 6 5 8 4
low
high

Partition (continued)



```c
void partition(int n, int a[n])
{
    int low = 0, high = n-1, pivot = a[0];
    while (low<high)
    {
        /* Search for a smaller element from the right */
        while (a[high]>pivot && low<high) high--;
        if (low < high) /* Found a smaller element at high */
            a[low++] = a[high];   /* Move the smaller element */

        /* Search for a larger element from the left */
        while (a[low]<pivot && low<high) low++;
        if (low < high) /* Found a larger element at low */
            a[high--] = a[low];   /* Move the larger element */
    }
    a[high] = pivot;     /* Copy the pivot element back */
    if (high>1) partition(high, a);
    if (high<n-2) partition(n-high-1, a+high+1);
}
```

```c
#include <stdio.h>
#define N 10

void partition(int n, int a[n]);
int main(void)
{
    int a[N], i;

    printf("Enter %d numbers to be sorted: ", N);
    for (i = 0; i < N; i++)
        scanf("%d", &a[i]);
    partition(N, a);
    printf("In sorted order: ");
    for (i = 0; i < N; i++)
        printf("%d ", a[i]);
    printf("\n");
    return 0;
}
```

Recursion

Advantages	Disadvantages
Simplifies complex problems.	Memory usage.
Saves time and space.	May cause stack overflow.
Increases code readability.	Difficulty in understanding and debugging.
Efficient data processing.	Slower execution.
Facilitates problem solving.	Limited applicability.

Replace recursion with loops if speed is your concern!

Lecture 7 struct, union & enum

Life is like a cup of tea...

A lot of data, used together

```
/* Print out the catalog entry */
void catalog(const char *name, const 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(const char *name, const char *species, int teeth, int age)
{
    printf("Name:%s\nSpecies:%s\n%i years old, %i teeth\n",
           name, species, teeth, age);
}
```

Create your own structured type

Keyword for struct definition

Name of the struct

Members of the struct

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

c.f. Arrays:

- Contains data of different types
- Has fixed length
- Elements have distinct names

Using struct

- Declaration & initialization

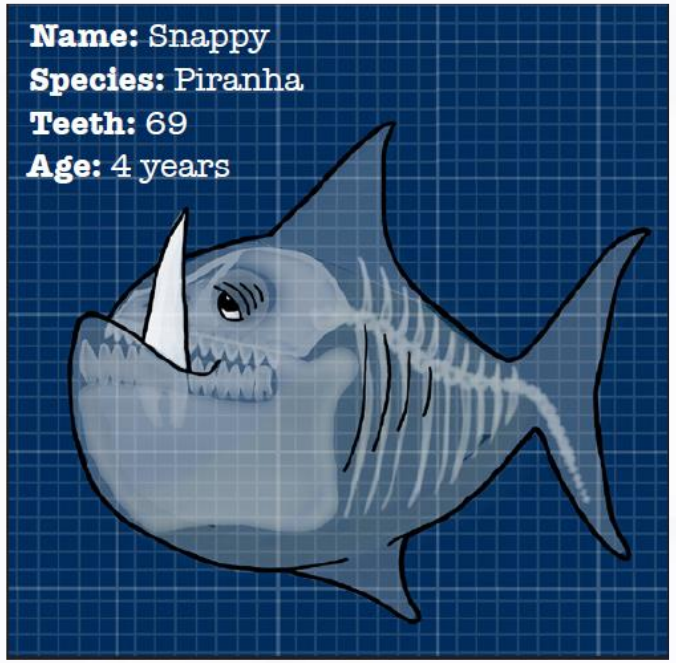
```
struct fish snappy = {"Snappy", "Piranha", 69, 4};
```

an instance of struct fish

- As function arguments

```
void catalog(struct fish f)
{
    ...
}
```

```
catalog(snappy);
```



Using struct

- Accessing a struct's fields with the . operator

```
printf("Name = %s\n", snappy.name);
```

- A struct may be extended easily with added fields

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

No need to update the functions that do not use the new fields.

FAQs about struct and array

- ➢Is a struct just an array?
- ✓No, but it's like an array. It groups different types of data together.
- ➢An array variable is a pointer to the array. Is a struct variable a pointer?
- ✓No, a struct is the name of the struct itself.
- ➢Can I use [ ] to access the fields of a struct?
- ✓No, you can only access the fields by name.
- ➢Are structs like classes in other languages?
- ✓They are similar, but it's not easy to add methods.

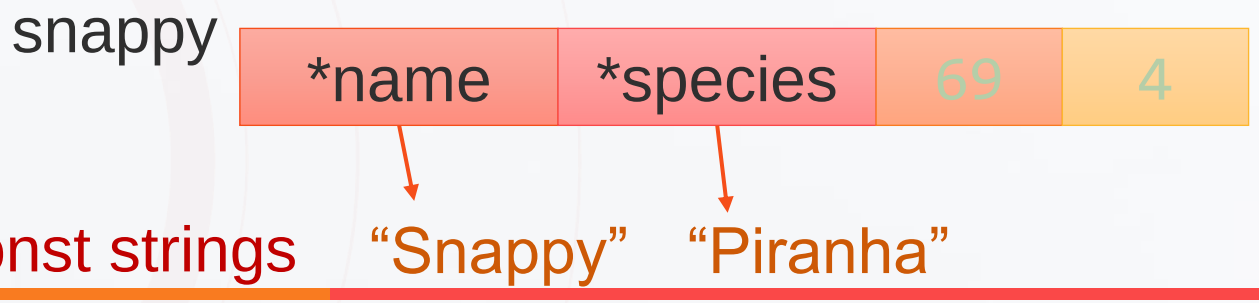
Structs in memory

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

No memory is allocated when defining a struct. Only a template for a new type of data is prescribed.

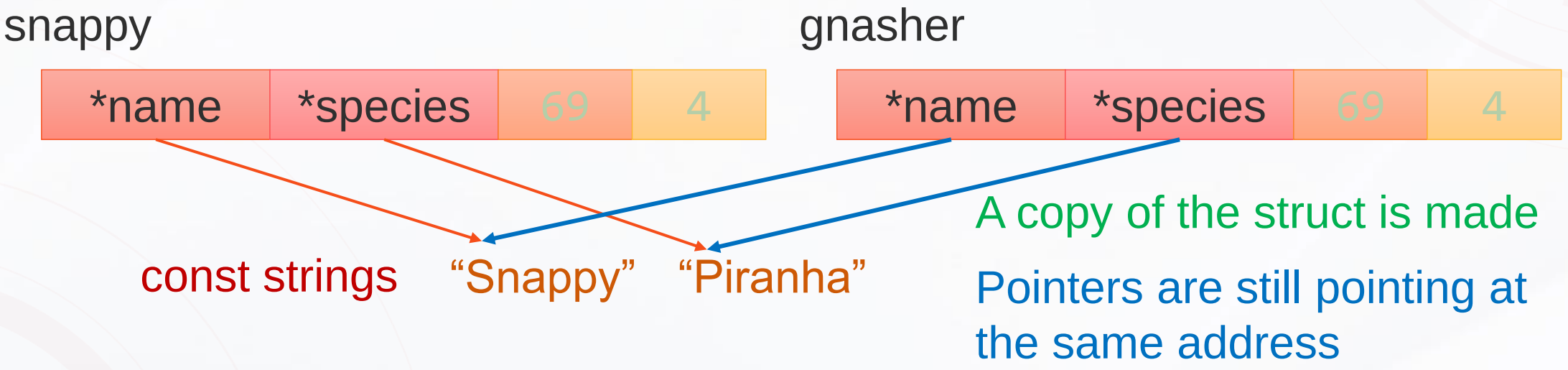
Memory is allocated when an instance of the struct is declared.

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



Assignment of struct

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




```
catalog(snappy);
```

A copy of the struct is made when pass to a function!

```c
#include <stdio.h>
struct turtle {
    const char *name;
    const char *species;
    int age;
};

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

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

The struct is copied before passing into a function!

You need a pointer to the struct

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

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

C Operator Precedence

Precedence	Operator	Description
1	++ -- () [] . ->	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer
2	! * & sizeof	Logical NOT Indirection (dereference) Address of … Size of …
3	* / %	Multiplication, division, and remainder
4	+ -	Addition and subtraction
5	< <= > >=	Relational operators < and ≤ respectively Relational operators > and ≥ respectively
6	== !=	Relational = and ≠ respectively
7	&&	Logical AND
8	||	Logical OR
9	= += -=	Simple assignment Assignment by sum and difference

The -> operator (arrow)

```
(*t).age
```

```
t->age
```

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

Nested structs

A struct is just like a new type, can we use it as a field of another struct?

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

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

Nested structs

- Declaration / initialization of nested structs

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

Initializer for the field *care*

- Accessing the fields of nested structs

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

Name your own type with typedef

```
struct cell_phone {
    int cell_no;
    const char *wallpaper;
    float minutes_of_charge;
};
```

```
struct cell_phone p = {5557879, "sinatra.png", 1.35};
```

C allows you to create an alias for any struct that you create by using **typedef**

```
typedef struct cell_phone
{
    int cell_no;
    const char *wallpaper;
    float minutes_of_charge;
} phone;
```

Name (alias) of the type defined

```
phone p = {5557879,
           "sinatra.png", 1.35};
```

typedef also works for other (complex) types