What have we learned?

struct

- Defining a struct (interface)

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

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

- Declaring/initializing an instance (variable)

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

```
    struct fish carp = {.teeth=56};
```

What have we learned?

struct

- Assignment

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

- Accessing the fields of a struct

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

- Struct pointers

```
struct turtle *t;
(*t).age++;
```

```
t->age ++;
```

Struct as the return value of a function

```
typedef struct {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} Part;
```

A struct may contain an **array** as a field

The entire array will be **copied** when passed to a function.

```
Part build_part(int number, const char *name, int on_hand)
{
    Part p;
    p.number = number;
    strcpy(p.name, name);
    p.on_hand = on_hand;
    return p;
}
```

Avoid copying BIG structs for efficiency

- Passing/returning a struct to/from a function requires making a copy of all members in the struct.
- Copying a big struct is inefficient, in terms of both memory and speed.
- To avoid this overhead, it's sometimes advisable to pass/return a pointer to a struct.
- Example: the FILE struct defined in `<stdio.h>` stores information about the state of an open file. Every function that opens a file returns a FILE pointer. A function that performs an operation on an open file requires a FILE pointer as an argument.

Arrays of structs

- An array structs capable of storing information about 100 parts

```
Part inventory[100];
```

- Access a part in the array

```
print_part(inventory[i]);
```

- Access a field of a struct in an array

```
inventory[i].number = 883;
strcpy(inventory[i].name,”Part1\0”);
```

Example: Maintaining a part database

- The program tracks parts stored in a warehouse.
- Information about the parts is stored in an array of structs.
- Contents of each structure:
  - Part number
  - Name
  - Quantity

	A	B	C	D
1	Part number	Name	Quantity	
2	1	Hard Drive	203	
3	2	Memory	510	
4	3	Monitor	120	
5	4	Keyboard	729	
6	5	Mouse	1200	
7				

Example: Maintaining a part database

- Operations supported by the program:
  - Add a new part, input its part number, name, and initial quantity on hand
  - Given a part number, print the name and the current quantity on hand
  - Given a part number, change the quantity on hand
  - Print a table showing all information in the database
  - Terminate program execution
- The codes `i` (insert), `s` (search), `u` (update), `p` (print), and `q` (quit) will be used to represent these operations.

Example: Maintaining a part database

```
#define NAME_LEN 25
#define MAX_PARTS 100

typedef struct part
{
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} PART;
PART inventory[MAX_PARTS];
     /* A global array to store the inventory */

void insert(void);
void search(void);     /* Functions for inventory operations */
void update(void);
void print(void);
```

Example: Maintaining a part database

```
Enter operation code: i
Enter part number: 528
Enter part name: Disk drive
Enter quantity on hand: 10

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 10
```

```
Enter operation code: s
Enter part number: 914
Part not found.

Enter operation code: u
Enter part number: 528
Enter change in quantity on hand: -2
```

```
Enter operation code: p
Part Number   Part Name              Quantity on Hand
    528       Disk drive                    8
    914       Printer cable                 5

Enter operation code: q
```

Example: Maintaining a part database

- outline of the main loop

```
while (1)
{
   prompt user to enter operation code;
   read code;
   switch (code) {
      case 'i': perform insert operation; break;
      case 's': perform search operation; break;
      case 'u': perform update operation; break;
      case 'p': perform print operation; break;
      case 'q': terminate program;
      default:  print error message;
   }
}
```

```c
int main(void)
{
    char code;
    while(1) {
        printf("Enter operation code: ");
        scanf(" %c", &code);
        while (getchar() != '\n');   /* skips to end of line */
        switch (code) {
            case 'i': insert();
                break;
            case 's': search();
                break;
            case 'u': update();
                break;
            case 'p': print();
                break;
            case 'q': return 0;
            default:  printf("Illegal code\n");
        }
        printf("\n");
    }
}
```

```c
void insert(void)
{
    int part_number;
    if (num_parts == MAX_PARTS) {
        printf("Database is full; can't add more parts.\n");
        return;
    }
    printf("Enter part number: ");
    scanf("%d", &part_number);
    if (find_part(part_number) >= 0) {
        printf("Part already exists.\n");
        return;
    }
    inventory[num_parts].number = part_number;
    printf("Enter part name: ");
    read_line(inventory[num_parts].name, NAME_LEN);
    printf("Enter quantity on hand: ");
    scanf("%d", &inventory[num_parts].on_hand);
    num_parts++;
}
```

```
int find_part(int number);
    /* To find a part by its part number,
     return the index in the array, or -1 if not found */
```

```
int find_part(int number)
{
    for (int i = 0; i < num_parts; i++)
        if (inventory[i].number == number)
            return i;
    return -1;   /* Part not found in the inventory */
}
```

```c
#include <ctype.h>

int read_line(char str[], int n)
{
    int ch, i = 0;
    while (isspace(ch = getchar()));
    while (ch != '\n' && ch != EOF)
    {
        if (i < n) str[i++] = ch;
        ch = getchar();
    }
    str[i] = '\0';
    return i;
}
```

```
void search(void)
{
    int i, number;

    printf("Enter part number: ");
    scanf("%d", &number);
    i = find_part(number);
    if (i >= 0)
    {
        printf("Part name: %s\n", inventory[i].name);
        printf("Quantity on hand: %d\n", inventory[i].on_hand);
    }
    else
        printf("Part not found.\n");
}
```

```
void update(void)
{
    int i, number, change;

    printf("Enter part number: ");
    scanf("%d", &number);
    i = find_part(number);
    if (i >= 0)
    {
        printf("Enter change in quantity on hand: ");
        scanf("%d", &change);
        inventory[i].on_hand += change;
    }
    else
        printf("Part not found.\n");
}
```

```
void print(void)
{
    int i;

    printf("Part Number   Part Name                      "
        "Quantity on Hand\n");
    for (i = 0; i < num_parts; i++)
        printf("%7d       %-25s%11d\n", inventory[i].number,
            inventory[i].name, inventory[i].on_hand);
}
```

Sometimes the same thing needs different types…

You may create a struct

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

But it's kind a waste!

- Takes up more memory
- Someone might set more than one value
- There's nothing called "quantity"

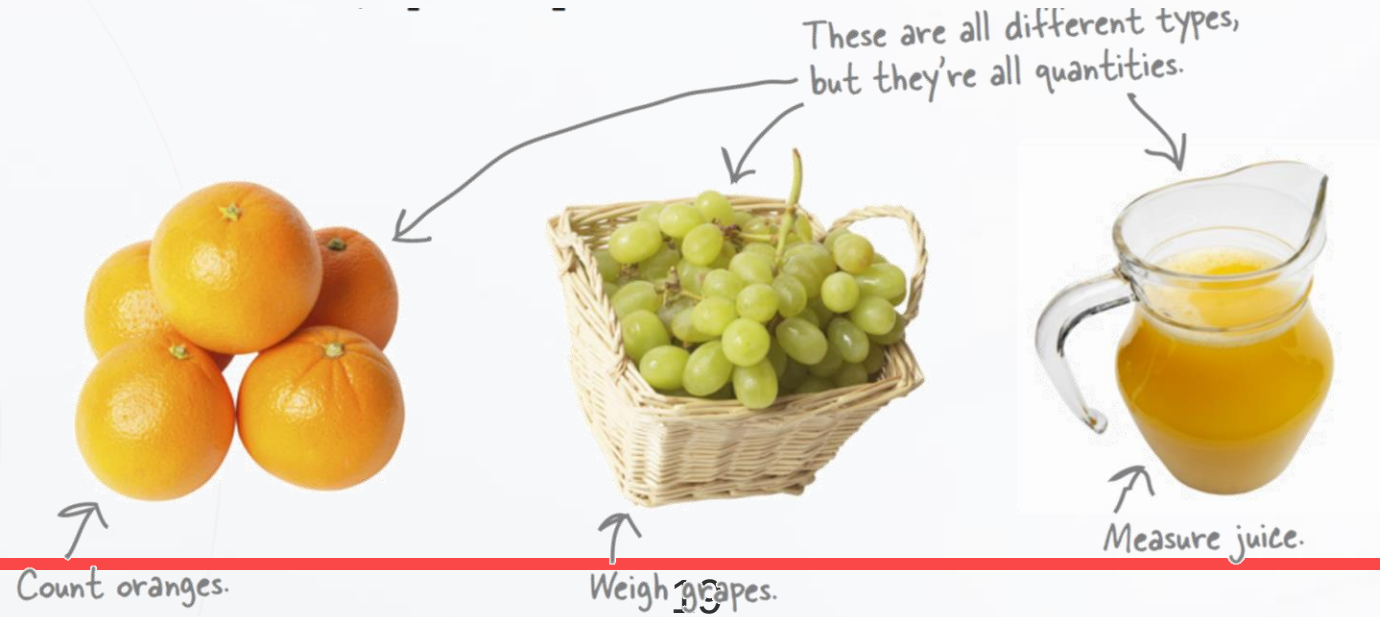
A union lets you reuse memory

short count	float weight	float volume

quantity (short or float)

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

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



Using a union

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

- Declaration & initialization

```
quantity q = {4};     /* default to the first field */
```

Designated initializer

Can also be used to initialize structs

```
quantity q = {.weight=1.5};
```

- Accessing one field

You are responsible for what's stored!

```
q.volume = 3.7;
```

Only one piece of data is stored/valid!

Unions used together with structs

```
typedef union {
    float lemon;
    int lime_pieces;
} lemon_lime;

typedef struct {
    float tequila;
    float cointreau;
    lemon_lime citrus;
} margarita;
```

```
margarita m = {2.0, 1.0, {0.5}};
```

```
margarita m = {2.0, 1.0, {.lime_pieces=1}};
```

```
margarita m = {2.0, 1.0, .citrus.lemon=2};
```