# Function pointers

Lecture 04.03

# Outline

- Function pointers

- Arrays of function pointers

- Passing function pointers as parameters

- Sorting with *qsort*

# Duck simulation

- Imagine that we are writing a simulation for a game about ducks in a lake

- Each duck behaves differently, it has its own quack and fly behavior

# Defining ducks

```
typedef enum {
    FLY_WINGS, NO_FLY
} FlyType;


typedef enum {
        QUACK, SILENCE
} QuackType;


typedef struct duck{
    char * name;
    FlyType fly_type;
    QuackType quack_type;
} Duck;
```

What is an advantage of using *enum* here?

# Implementing different fly and quack behaviors

```
void quack () {
    puts ("Quack quack");
}


void mute_quack () {
    puts ("<<Silence>>");
}


void fly_wings () {
    puts ("I am flying with the wings!");
}
```

# Adding some ducks

```
Duck ducks [2];
ducks[0].name = "Mallard";
ducks[0].fly_type = FLY_WINGS;
ducks[0].quack_type = QUACK;

ducks[1].name = "Domestic";
ducks[1].fly_type = NO_FLY;
ducks[1].quack_type = QUACK;

for (i=0; i< 2; i++){
    simulate_duck (ducks[i]);
 }
```

# Implementing *simulate_duck*

```c
void simulate_duck (Duck * duck){
    printf ("\nI am a %s Duck\n", duck->name);
    switch (duck->fly_type) {
        case FLY_WINGS:
            fly_wings ();
            break;
        default:
            no_fly();
    }

    switch (duck->quack_type) {
        …
    }
    …
}
```

Code in ducks.c

# We want to add new types of fly and quack

- The rubber duck: squeaks instead quacking

- The wooden duck: cannot fly

- The space duck: flies with the rocket

- …

# We need to change the code

- The code becomes too long
  - We need to introduce new types into each enum
  - We need to update switch statements with every new case


- Is there a better way of doing it in C?


- Yes, with *function pointers*

- Every function name is a pointer to the function: refers to the piece of code in memory

- Function names are also pointer variables

- When you create function

    *swim (int speed)*,

  you are also creating a

  pointer variable called *swim*

  that contains the *address* of the

  function

| Buffer |
| Code |
| Constants |
| Globals |
| HEAP |
| Stack |
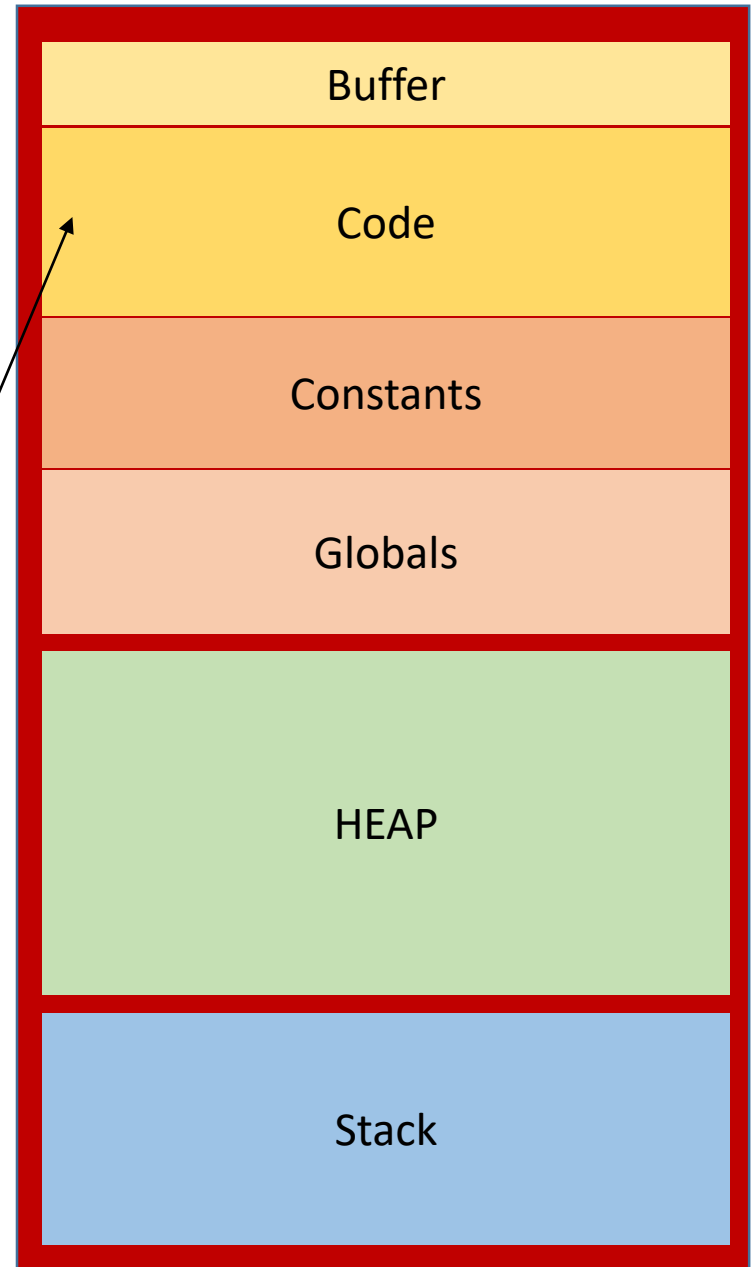
# How to declare a function pointer

- it's easy to declare pointers in C:

- If a data type is *int*, you declare a pointer with *int \**

- Unfortunately, C doesn't have a *function* data type, so you **can't** declare a function pointer with anything like *function \**

int *a;          *This declares a pointer*

function *f;     *…but this won't declare a*
                 *function pointer*

# Function datatype includes many different types

- C doesn't have a function data type because there's not just one type of function
- When you create a function, you vary a lot of things (the return type or the list of parameters)
- That combination of things is what defines the type of the function:

int swim (int speed) {

...

}


char** album_names(char *artist, int year) {

...

}

# Declaring variable of type function pointer

int (*swim_fp)(int); ← *This will create a variable called swim_fp that can store the address of the swim() function.*

swim_fp = swim;

swim_fp(4);

*This is just like calling swim(4).*

char** (*names_fp)(char*,int);

names_fp = album_names;

char** results = names_fp("Elton John", 1972);

# Now we can use a function pointer to

- Assign values to it: different functions with the same signature

- Add it to arrays

- Pass it as parameters to other functions

# Back to ducks

```
void simulate_duck (Duck * duck){
    void (*fly_fp)(void);
    switch (duck->fly_type) {
        case FLY_WINGS:
            fly_fp = fly_wings;
            break;
        case ROCKET:
            fly_fp = fly_rocket;
            break;
        default:
            fly_fp = no_fly;
    }
    fly_fp();
}
```

But how does it help to shorten the code?

So far we have the same number of cases, and we need to add new cases each time we extend a set of fly types

Code in ducks_fp.c

# Create an array of function pointers

- The trick is to create an array of function pointers that match the different fly types

- If we had an array of possible fly behaviours we could use is like this:

fly_behaviors[] = {fly_wings, fly_rocket, no_fly};

fly_behaviors [1];


- Instead, for array of function pointers we use:

void (*fly_behaviors[])() = {fly_wings, fly_rocket, no_fly};

fly_behaviors [1] ();

# Now we can call the function at the corresponding array index

```
void (*fly_behaviors[])() = {fly_wings, fly_rocket, no_fly};
void simulate_duck (Duck * duck){
    printf ("\nI am a %s Duck\n", duck->name);
    fly_behaviors [duck->fly_type] ();
    quack_behaviors [duck->quack_type] ();
}
```

Remember that each enum value is actually an int and it starts from 0?

One line of code replaces all the cases and we do not need to change this code to add new behaviours

# Storing function pointers as "methods" of a struct

```c
typedef struct duck{
    char * name;
    void (*fly) (void);
    void (*quack) (void);
} Duck;
```

Code in ducks_object.c

# Set function pointer to point to the corresponding "method"

```
Duck ducks [4];
ducks[0].name = "Mallard";
ducks[0].fly = fly_wings;
ducks[0].quack = quack;

ducks[1].name = "Domestic";
ducks[1].fly = no_fly;
ducks[1].quack = quack;

ducks[2].name = "Rubber";
ducks[2].fly = no_fly;
ducks[2].quack = squeak;

ducks[3].name = "Wooden";
ducks[3].fly = no_fly;
ducks[3].quack = mute_quack;
```

# And simulate object-oriented programming in C ☺

```
void simulate_duck (Duck * duck){

    duck->fly();

    duck->quack();
}
```

*fly* is a function pointer that points to one of the following real functions:
fly_wings
no_fly
fly_rocket

# Recognizing function pointers

| Return type | (* | Pointer variable | )( | Param types | ) |

char**          (*names_fp)          (char*, int);

This is the name of the variable you're declaring

# Exercise: sorting things

- Lots of programs need to sort data.

- If the data is not a set of numbers, which have their own natural order – how do you sort them?

- Imagine you have a set of people. How would you put them in order? By height? By intelligence? By hotness?

- How could we write a general sort function which will sort any type of data?

# Use function pointers to set the order

- C Standard Library qsort:

```
qsort (void *array,
       size_t length,
       size_t item_size,
       int (*compar)(const void *, const void *));
```
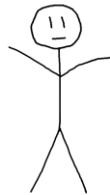
A pointer to a comparator function, which will be used to determine the order of 2 pieces of data

# comparator returns:

- The qsort() function compares pairs of values, and if they are in the wrong order, it will switch them

- The comparator function will tell qsort() which order a pair of elements should be in

- It does this by returning three different values:
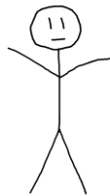
Positive number

Negative number

Zero

a     b

# Exercise: implement sorting of the following arrays

- Array of ints:

int scores[] = {543,323,32,554,11,3,112};

- Array of C strings:

char *names[] = {"Karen", "Mark", "Brett", "Molly"};

- Array of rectangles:

Rectangle rectangles [] = {{3,5}, {4,4},{1,18}};

# Comparator example 1/3: integers

- The first thing you need to do is get the integer values from the pointers:

int a = *(int*)score_a;

int b = *(int*)score_b;

- Then you need to return a positive, negative, or zero value, depending on whether a is greater than, less than, or equal to b:

return a - b;

- And this is how you ask qsort() to sort the array:

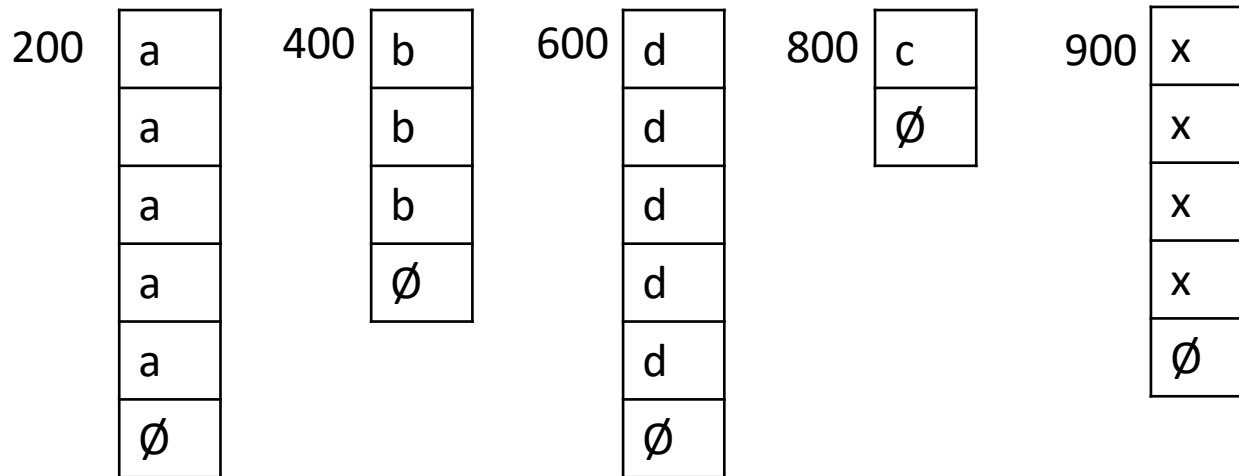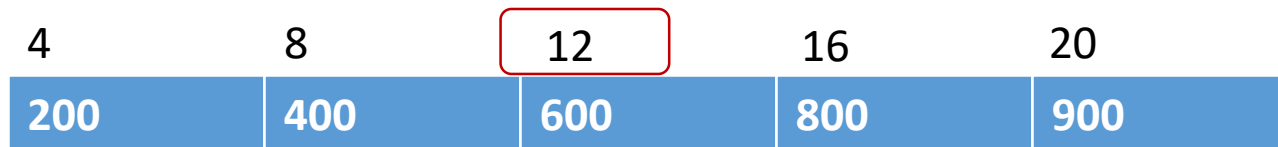qsort(scores, 7, sizeof(int), compare_scores);

# Comparator example 2/3: Rectangles

```c
int compare_rectangles(const void* a, const void* b){
    Rectangle ra = *(Rectangle*)a;
    Rectangle rb = *(Rectangle*)b;
    int area_a = (ra.width * ra.height);
    int area_b = (rb.width * rb.height);
    return area_a - area_b;
}
```

# Array of strings: char **

char ** arr;  `4`

Type: char **

| 4 | 8 | 12 | 16 | 20 |
|---|---|---|---|---|
| **200** | **400** | **600** | **800** | **900** |

200
| a |
|---|
| a |
| a |
| a |
| a |
| Ø |

400
| b |
|---|
| b |
| b |
| Ø |

600
| d |
|---|
| d |
| d |
| d |
| d |
| Ø |

800
| c |
|---|
| Ø |

900
| x |
|---|
| x |
| x |
| x |
| Ø |

# Comparator example 3/3: strings

```
int compare_strings (const void* a, const void* b) {
    char** aPP = (char**)a;
    char** bPP = (char**)b;
    char* aP = *aPP;
    char* bP = *bPP;
    return strcmp(aP, bP);
}
```