Function pointers

Lecture 09.01

Outline

- Function pointers
- Arrays of function pointers
- Passing function pointers as parameters
- Sorting with *qsort*

We have met function pointers before

struct sigaction contains following members:

- void (***sa_handler**)(int) Pointer to a signal-catching function
- sigset_t sa_mask Set of signals to be blocked during execution of the signal handling function
- int sa_flags Special flags

Object-oriented programming in C: 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*

```
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?

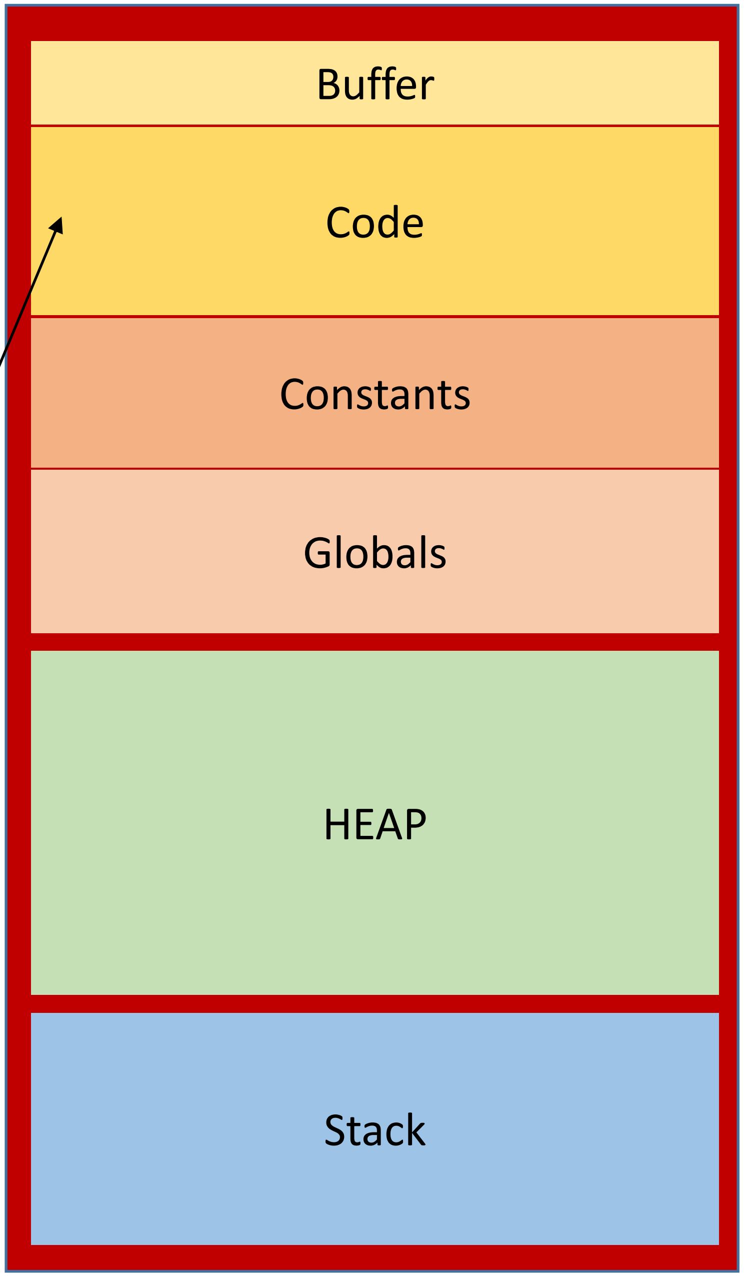
- Yes, with *function pointers*

- 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



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 does **NOT** 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 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);
swim_fp = swim;
swim_fp(4);
```

This will create a <u>variable</u> called swim_fp that can store the address of the swim() function.

This is just like calling swim(4)

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

We can use a variable of type *function pointer:*

- Assign different values to this variable: different functions **with the same signature**
- Add it to the array of function pointers
- 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

An **array** of function pointers

- The trick is to create an array of function pointers that matches different fly types
- If we had an *array* of possible fly behaviors 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 behaviors

Storing function pointers as “methods” of a struct

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

Code in ducks_object.c

Assign corresponding ‘method’ to the ‘object’ when it is created

```
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

Problem: sorting things in C (typed language)

- 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

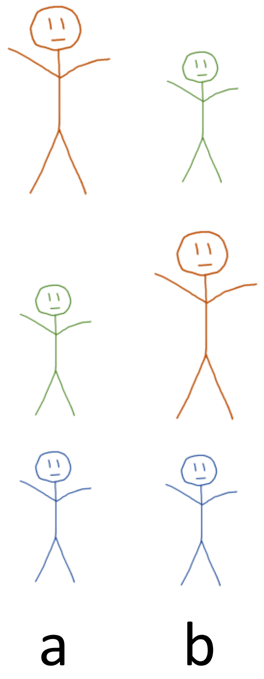
- C Standard Library function ***qsort***:

```
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 one of three different values:



Positive number

Negative number

Zero

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 Comparator function: compare_scores()

- 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;
```

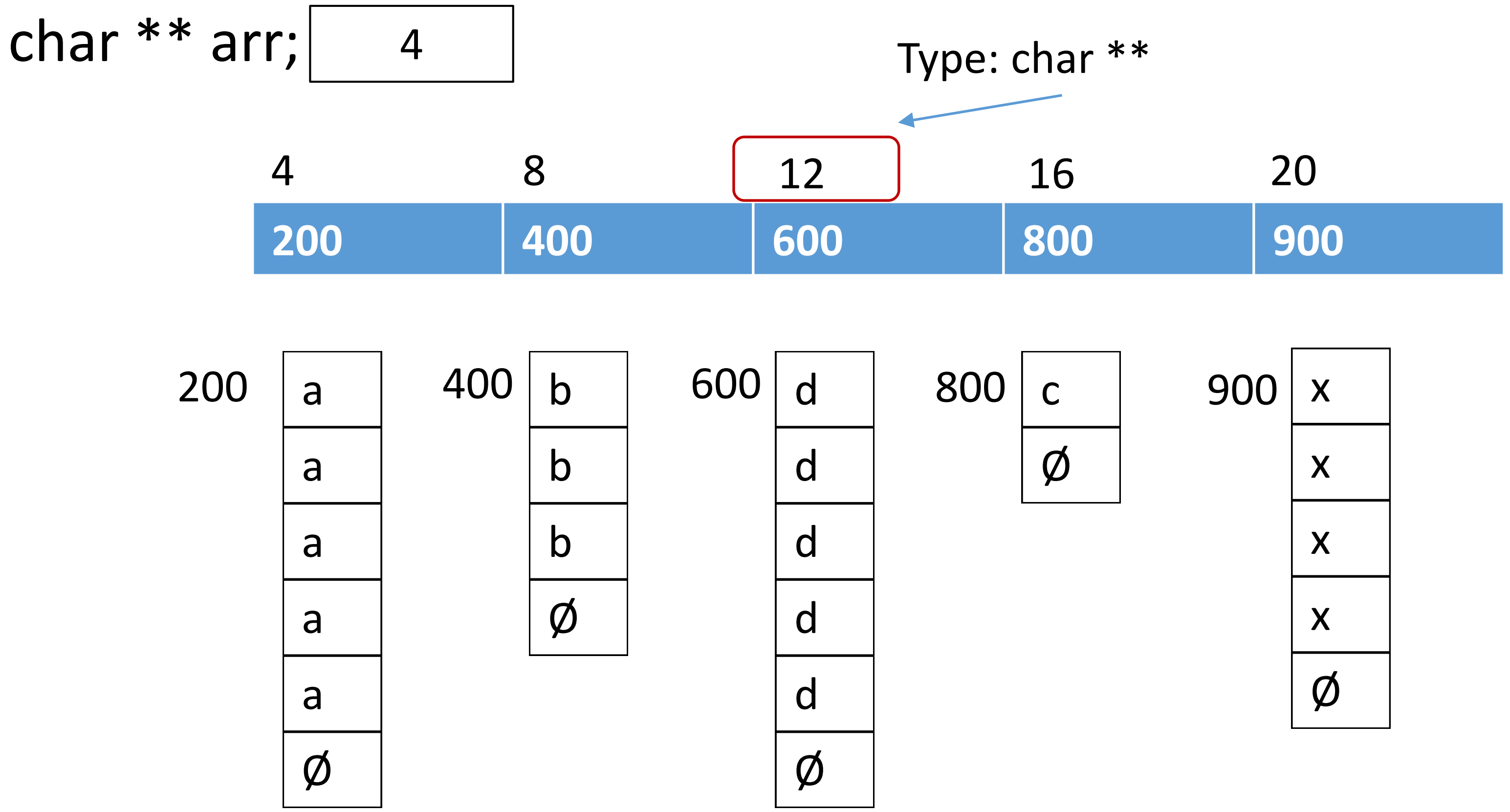
- And this is how you ask qsort() to sort the array:

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

Comparator example 2/3: Rectangles

```
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;
}
```

Recap: Array of strings: char **



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);
}
```