Spark Session: ft_printf

updated: 27/01/2021

Project description:

> Recode printf

Topics

1. Variadic Arguments
2. Function Pointers

Variadic Arguments

1. Variadic functions add flexibility to your code by allowing an unknown number of arguments. (30 mins)
    - What would its prototype look like? (5 mins)
    - Identify the 4 macros that allow you to access these arguments. (25 mins)
        - What are the argument types? For example, what exactly is the 2nd argument to va_start?
        - What are default argument promotions?
2. Let's practice accessing and carrying out operations on a variable argument list! (30 mins)
    - Write a variadic function that:
        - has a prototype of `function(const int n, ...)`
          n being the number of arguments in the list,
        - returns the **sum** of the integers in that list.
    - Write the accompanying main to test your function.
      Example test: does `yourfunction(3, 40, 5, -3)` return `42`?

Break (5 mins)

Function Pointers

1. Just as we can have pointers to data (char *, int *), we can have pointers to functions. (45 mins)
    - How do we declare a pointer to a function? Pay attention to bracket placement! (10 mins)
        - Let's break down the syntax. What does each part of the declaration mean?
        - Is there a difference between `void (*fn)` and `void *fn`?
    - What's happening when we assign the function pointer to a function?
      What information does the function pointer hold? (5 mins)
    - Like normal pointers, we can also have an array of function pointers.
      What is their syntax? (10 mins)
    - When can function pointers come in handy? (10 mins)
    - What is a typedef and how can it be used with function pointers? (10 mins)

2. Let's practice using a function pointer! (30 mins)
    - Write a function that: (10 mins)
        - takes an integer **n** as argument,
        - prints "Hello" **n** times,
        - returns nothing.
    - Now write an accompanying main that: (20 mins)
        - declares a pointer to a function that takes an int and returns nothing,
        - initialises that pointer to the Hello function you just wrote,
        - prints "Hello" 3 times **using the function pointer**.

Break (5 mins)

3. Now let's try doing something cooler with an array of function pointers. (20 mins)

    - Here's some code to get you started:

```
enum    e_op
{
    PLUS = 0, MINUS
};

void    operation_add(int a, int b)
{
    printf("%d + %d = %d\n", a, b, a + b);
}

void    operation_minus(int a, int b)
{
    printf("%d - %d = %d\n", a, b, a - b);
}
```

    - Write a main that:

        - declares an array of 2 function pointers, taking 2 ints and returning nothing,
        - assigns the first array element to `operation_add` and the second element to `operation_minus` ,
        - calls each function at least once through the array.
          Hint: enums can make indexing easier.

Bonus

1. Here's some code to get you started again:

```
typedef void    (*printfunct)(va_list list);

void    print_char(va_list list)
{
    printf("%c\n", va_arg(list, int));
}

void    print_string(va_list list)
{
    printf("%s\n", va_arg(list, char *));
}

void    print_digit(va_list list)
{
    printf("%d\n", va_arg(list, int));
}
```

Write a **variadic function** that:

- has a prototype of `function(char *str, char *filler, ...)`,
- has an **array of function pointers** assigned to the 3 `print_` functions above,
- for every valid option in `str`, calls the corresponding function from the function pointer array,
    - valid options: `'c'` should trigger `print_char`, `'d'` triggers `print_digit`, and `'s'` triggers `print_string`
    - invalid options: print the `filler` string and then continue onto the next character in `str`

2. Write the accompanying main. Test it with the following input: `yourprintfunc("csdcx", "REJECTED", 'k', "hello", 42, 'f')`.

3. *Bonus bonus*: how can you avoid using a bunch of if-else statements in this exercise?