Write a function that prints an integer.

Prototype: void print_number(int n);
You can only use _putchar function to print
You are not allowed to use long
You are not allowed to use arrays or pointers
You are not allowed to hard-code special values
julien@ubuntu:~/0x06$ cat 101-main.c

```
#include "main.h"

/**
* main - check the code
*
* Return: Always 0.
*/
int main(void)
{
print_number(98);
_putchar('\n');
print_number(402);
_putchar('\n');
print_number(1024);
_putchar('\n');
print_number(0);
_putchar('\n');
print_number(-98);
_putchar('\n');
return (0);
}
```

julien@ubuntu:~/0x06$ gcc -Wall -pedantic -Werror -Wextra -std=gnu89 _putchar.c 101-main.c 101-print_number.c -o 101-print_numbers
julien@ubuntu:~/0x06$ ./101-print_numbers

```
98
402
1024
0
-98
```

julien@ubuntu:~/0x06$

my solution

```
void print_number(int n)
{

unsigned int number;
```

```
if (n < 0)
{
_putchar('-');

n *= -1;
}

number = n;

if (number / 10)
{
print_number(number / 10);
}

_putchar(number % 10 + '0');
}
```

break down the solution

Certainly, let's break down the provided solution for the `print_number` function.

1. The function starts by defining an unsigned integer variable named `number`.

```
unsigned int number;
```

Unsigned integers can represent only non-negative numbers. We'll use this to help with arithmetic operations since negative integers can complicate calculations.

2. The function then checks if the given integer `n` is negative.

```
if (n < 0)
{
        _putchar('-');
        n *= -1;
}
```

If `n` is negative:

- It prints a minus sign (`-`) using the `_putchar` function.
- It then multiplies `n` by `-1` to convert it into a positive value. This makes it easier to process the digits of `n`.

3. The positive value of `n` is then assigned to `number`.

```
number = n;
```

This step is necessary since the function modifies the value of `n` in the previous step if it's negative. By assigning the positive value of `n` to `number`, we are now working with an unsigned integer.

4. Next, the function checks if `number` divided by 10 is not zero.

```
if (number / 10)
{
        print_number(number / 10);
}
```

This step essentially checks if there are more digits to the left in the number. If `number / 10` is not zero, it means there are more digits, and we recursively call `print_number` with `number / 10`. This breaks down the number into its individual digits, starting from the leftmost digit.

For example, with the number 123:

- First call: 123 / 10 = 12 (We have more digits, so we recursively call with 12)
- Second call: 12 / 10 = 1 (We still have more digits, so we recursively call with 1)
- Third call: 1 / 10 = 0 (No more digits, move to the next step)

5. Finally, the function prints the rightmost digit of `number`.

```
_putchar(number % 10 + '0');
```

`number % 10` fetches the rightmost digit of `number`. By adding `'0'`, we are converting the numeric digit into its corresponding ASCII character. This is crucial for printing.

For our example number 123:

- In the third call: 1 % 10 = 1 -> We print '1'
- In the second call: 12 % 10 = 2 -> We print '2'
- In the first call: 123 % 10 = 3 -> We print '3'

Thus, 123 is printed as expected.

In summary, the function converts negative numbers to positive (while printing a minus sign), then it recursively breaks down the number to print each digit from left to right.

i don't understand

Of course, I'll attempt a more straightforward explanation.

The main idea behind the solution is to print a number digit-by-digit starting from its leftmost digit. If you think about how we humans say a number out loud, it's from the leftmost side to the rightmost side. For example, for the number 321, we first say "three" then "two" and finally "one". This solution emulates that process.

Let's break it down step-by-step:

Step 1: Check if the number is negative

```
if (n < 0)
{
        _putchar('-');
```

```
        n *= -1;
}
```

If the number `n` is less than 0, it means it's negative. We print a negative sign and then convert `n` to its positive counterpart to make things simpler moving forward.

Step 2: Recursive Call

```
if (number / 10)
{
        print_number(number / 10);
}
```

We're trying to print the number from the leftmost digit. But how do we get there? By dividing the number by 10.

Here's why:

- For the number 5 (just one digit), 5/10 is 0.
- For the number 54, 54/10 is 5.
- For the number 543, 543/10 is 54.

Notice how dividing by 10 removes the last digit? That's what we're aiming for. We keep on dividing by 10 (and recursively calling the function) until we're left with the leftmost digit.

Step 3: Print the rightmost digit

```
_putchar(number % 10 + '0');
```

After we're done with step 2, we're left with the leftmost digit, and now, we want to print the current last digit. We do this using the modulus operator (`%`). The modulus operator gives the remainder after division.

For example:

- 5 % 10 is 5.
- 54 % 10 is 4.
- 543 % 10 is 3.

See? `% 10` gives the last digit of any number. We print that digit.

Example:

Consider the number 543.

1. The function sees that 543 is not negative, so it skips the negative check.
2. It divides 543 by 10 to get 54 and calls itself with 54.
3. Now, it divides 54 by 10 to get 5 and calls itself with 5.
4. This time 5/10 is 0, so it doesn't call itself again. It moves to the print statement.
5. It prints the digit 5 using 5 % 10.
6. The call with 54 now prints the digit 4 using 54 % 10.
7. The original call with 543 prints the digit 3 using 543 % 10.

The result is "543".

I hope this makes it clearer! Let me know if you have further questions or if there's any other way I can help.

why after print 5 it print 4 how I don't understand

Certainly! Let's dissect the process more thoroughly using the recursion concept.

Recursion can be thought of as a function calling itself, often with modified arguments, to break down a problem into smaller chunks. It's like a series of nested calls, and each nested call must complete before the previous one can continue.

Given the number `543`, let's see how the function behaves:

1. You call `print_number(543)`.
2. Inside this call:
    3. It checks if `543` is negative. It isn't, so nothing happens here.
    4. Then it checks `if (number / 10)`, which is `if (543 / 10)`. Since `543 / 10 = 54` (in integer division, fractions are discarded), it's true. So, it goes inside the if condition and calls itself with `number / 10` which is `54`.

       So now, `print_number(54)` is called **before** we reach the `_putchar` command for `print_number(543)`.
5. Inside the `print_number(54)` call:
    6. Again, it checks if `54` is negative. It isn't.
    7. It then checks `if (54 / 10)`. Since `54 / 10 = 5`, it's true. So, it calls itself with `5`.

       Thus, `print_number(5)` is called **before** we reach the `_putchar` command for `print_number(54)`.
8. Now, within `print_number(5)`:
    9. It's not negative.
    10. When it checks `if (5 / 10)`, it finds it's false (because `5 / 10 = 0.5` and fractions are discarded in integer division).
    11. Therefore, it proceeds to `_putchar(number % 10 + '0')`, which translates to `_putchar(5 % 10 + '0')` or `_putchar(5 + '0')`. This will print the number `5`.

       Having finished the tasks inside `print_number(5)`, this call is now complete and it returns to where it left off in the `print_number(54)` call.
12. Picking up from where we left in `print_number(54)` (after its recursive call to `print_number(5)`):
    13. It now executes `_putchar(number % 10 + '0')`, which is `_putchar(54 % 10 + '0')`, printing the digit `4`.

       Now, the tasks inside `print_number(54)` are complete, and it returns to the initial `print_number(543)` call.
14. Resuming `print_number(543)` (after its recursive call to `print_number(54)`):

15. It executes `_putchar(number % 10 + '0')`, which is `_putchar(543 % 10 + '0')`, printing the last digit `3`.

So, the order of printing is `5`, then `4`, and finally `3`, giving us the number `543`.

The recursive structure makes it so that the function will always "go as deep as it can" (i.e., call itself with smaller and smaller values) before it starts printing anything. And when it does start printing, it does so from the smallest (leftmost) of those values and works its way back out.

print_number(5), this call is now complete and it returns to where it left off in the print_number(54) call. why it should return to 54

The behavior you're asking about is a fundamental aspect of how recursive functions work in programming.

When `print_number(54)` calls `print_number(5)`, the program hasn't finished executing everything inside `print_number(54)` yet. Specifically, it called `print_number(5)` before reaching the `_putchar` line. This is a key point to understand: `print_number(54)` is paused in the middle of its execution at the point it calls `print_number(5)`. It's waiting for `print_number(5)` to complete so it can continue.

Here's a simplified step-by-step of what's happening:

1. `print_number(543)` is called.
2. Inside `print_number(543)`, `print_number(54)` is called (but `print_number(543)` has not finished yet; it's paused).
3. Inside `print_number(54)`, `print_number(5)` is called (but `print_number(54)` has not finished yet; it's paused too).
4. `print_number(5)` runs and hits the `_putchar` line, printing "5". `print_number(5)` has no more lines of code to execute, so it's now complete.
5. Because `print_number(5)` is complete, control returns to where `print_number(54)` had paused. Now, `print_number(54)` can continue executing.
6. `print_number(54)` continues, hitting its `_putchar` line and printing "4". With no more lines of code to execute, `print_number(54)` is now complete.
7. Control returns to where `print_number(543)` had paused. It continues executing, hitting its `_putchar` line and printing "3". It then completes.

This "pausing" and "returning" behavior is a fundamental aspect of recursive functions and, more broadly, of how functions work in many programming languages. When Function A calls Function B, Function A pauses at the point of the call, and execution continues inside Function B. Once Function B finishes, control returns to the point in Function A where it had paused, and it continues from there. If Function B itself calls Function C, then Function B pauses, waits for Function C to complete, and then resumes, and so on. This creates a "stack" of function calls, each waiting for the next to complete, which is why the call structure in recursive functions (and function calls in general) is often referred to as a "call stack."

Good question! The concept behind this is the nature of recursion and how function calls work in programming, especially when a function calls itself.

Let's use a simpler analogy to understand this:

Imagine you're reading a book, and you come across a term you don't understand, so you consult a dictionary to look up that term. While reading the definition, you come across another term in the definition that you don't understand, so you look up that term too. This could continue for a few terms. But once you understand the last term you looked up, you'll go back to the previous term's definition, and then, once you're done with that, you'll return to your main book.

Your original book reading was "interrupted" by the first dictionary lookup, which was then "interrupted" by the second lookup, and so on. But once each "interruption" is resolved, you return to the task that was interrupted.

The process of `print_number` is similar:

1. When `print_number(543)` is called, before it can complete its job (print the digit `3`), it finds it needs to call `print_number(54)`. So, the task of `print_number(543)` is temporarily "interrupted" or "paused" so `print_number(54)` can run.
2. Now, inside `print_number(54)`, before it can complete its job (print the digit `4`), it finds it needs to call `print_number(5)`. Again, the task of `print_number(54)` is temporarily "interrupted" so `print_number(5)` can run.
3. `print_number(5)` has no further recursion, so it completes its job by printing the digit `5` and then it's done. Now, control returns to where we "paused" or "interrupted" in the `print_number(54)` call.
4. `print_number(54)` can now complete its job (print the digit `4`) since its "interruption" (`print_number(5)`) is finished. After printing `4`, it's done. Control then returns to the original `print_number(543)` call.
5. Finally, `print_number(543)` prints its remaining digit, `3`.

This is how function calls work in programming. When a function (A) calls another function (B), function A is paused until function B completes. Once B completes, control returns to function A to finish its task. If B were to call another function C, then B would pause until C completes, and so on. This chain of calls and returns is managed using a call stack in computer memory.

From ChatGPT, designed with FancyGPT