Doubt 03 - String scanset

```c
#include<stdio.h>
int main(){

    //input using scanset
    char str1[10];
    printf("Enter vowel letter : ");
    scanf("%[aeiou]",str1);
    printf("%s\n",str1);

    char str2[10];
    printf("Enter consonant letter : ");
    scanf(" %[^aeiou]",str2);
    printf("%s\n",str2);
    return 0;

}
```

```
$ ./03-readWriteFunctions
Enter vowel letter : aiopranav
aio
Enter consonant letter : pr

$ ./03-readWriteFunctions
Enter vowel letter : aiolkj
```

```
aio
Enter consonant letter : pranav
lkj
pr

$ ./03-readWriteFunctions
Enter vowel letter : aiopkpka
aio
Enter consonant letter : pkpk
```

CODE EXPLANATION

1. **First Input Block**
   - `scanf("%[aeiou]", str1)` reads only vowel letters from the input. The `%[aeiou]` scanset tells `scanf` to read and store all the characters that belong to the set `[aeiou]` until it encounters a character that is not a vowel.
   - Example input: `aiopranav`
   - The program will only read the vowels (`aio`), and it stops at `p`, a consonant.
2. **Second Input Block**
   - `scanf(" %[^aeiou]", str2)` reads everything except vowels. The `[^aeiou]` means "read all characters except the ones in the set `[aeiou]`". The leading space in `" %[^aeiou]"` helps to ignore any leading whitespace.
   - Input in first scanf was `aiopranav`, which have read only vowels that are `aio`, the remaining input after reading vowels was `pranav`, the second input will read `pr` (as it stops at the vowel `a`).

OUTPUT FOR VARIOUS INPUTS

1. **First Output
   - Input: `aiopranav`

- First `scanf` : reads vowels `aio`
- Second `scanf` : reads consonants `pr`
- Output ```

```
aio
pr
```

2. **Second Output**
    - Input: `aiolkj pranav`
    - First `scanf` : reads vowels `aio`
    - Second `scanf` : reads consonants `lkj` (from first input) and then reads `pr` (from `pranav`).
    - Output

```
aio
lkj
pr
```

3. **Third Output**
    - Input: `aiopkpka`
    - First `scanf` : reads vowels `aio` .
    - Second `scanf` : reads consonants `pkpk` .
    - Output

```
aio
pkpk
```

SOLUTION

To avoid the behavior where the second `scanf` reads part of the leftover input from the first `scanf`, you can add logic to discard the remaining characters after the first `scanf`. There are a few ways to do this :

Solution 1: Use `getchar()` to consume the remaining characters

After reading the vowels, you can use a loop to consume all leftover characters in the input buffer until a newline (`\n`) or some other terminator. This ensures that the second input starts fresh.

Here's an updated version of your code :

```
#include<stdio.h>
int main(){

    //input using scanset
    char str1[10];
    printf("Enter vowel letter : ");
    scanf("%[aeiou]",str1);
    printf("%s\n",str1);

    // Consume the leftover characters in the input buffer
    int c;
    while ((c = getchar()) != '\n' && c != EOF);  //Clears the buffer until newline

    char str2[10];
    printf("Enter consonant letter : ");
    scanf(" %[^aeiou]",str2);
    printf("%s\n",str2);
```

```
    return 0;
}
```

How it works:

- `while ((c = getchar()) != '\n' && c != EOF);` : This loop reads and discards characters from the input buffer until it encounters a newline (`\n`) or reaches the end of the file (EOF). This prevents leftover input from affecting the next `scanf` .

Solution 2: Use `fflush(stdin)` (non-standard)

Some compilers allow the use of `fflush(stdin)` to clear the input buffer, although it is non-standard and not recommended for portable code. The above method with `getchar()` is more widely supported.

Solution 3: Read the entire input as a string

Alternatively, you can read the entire input as a single string, process the vowels and consonants separately, and avoid dealing with buffer issues altogether.

#DOUBT **- why used `int c`**

The variable `int c` is used to temporarily store the characters being read by the `getchar()` function.

Why use `int` instead of `char` ?

1. **Return Type of `getchar()` :**
    - The `getchar()` function returns an `int` , not a `char` , because it needs to return all possible `char` values (including special ones) as well as a special value to indicate the end of input (`EOF`).

- In C, `EOF` (end-of-file) is typically defined as `-1`, which doesn't fit into a `char` type, as `char` is usually an 8-bit type and can't represent negative numbers like `EOF`.

2. **Ensuring Proper Comparison**:
   - By storing the result of `getchar()` in an `int`, you can safely check whether it equals `EOF` without truncating the value. Using `char` could cause incorrect comparisons or data loss if the character returned has the value of `EOF`.

Example of the line:

```
int c; while ((c = getchar()) != '\n' && c != EOF);
```

This:

- Reads one character at a time using `getchar()`, which returns an `int` representing the character or `EOF`.
- Stores the result in `c`.
- Stops reading when it encounters a newline (`\n`) or end-of-file (`EOF`).

So, using `int c` is essential to correctly handle the `EOF` condition.

#DOUBT **- but remaining input buffer contains characters then how can it be stored in `int c`**

Let's clarify why `int c` can store characters even though it's an `int`.

How `getchar()` Works with `int`:

- **Characters as Integers**: In C, characters are simply small integers that represent ASCII values. For example, the character `'A'` is represented by the integer value `65`, `'a'` by `97`, and so on. The type `char` is just a smaller version of `int` (typically 8 bits), but all characters are still internally stored as integer values.

- `getchar()` reads these character values from the input buffer and returns them as `int`. This way, any character like `'a'` (which has ASCII value 97) can be returned and stored in an `int` without a problem.

Why `int` is Used and Not `char`:

- **Full Range of Characters + EOF**: The reason `int` is used instead of `char` in this context is to accommodate all possible character values **plus** the special `EOF` marker.
  - A `char` is typically 8 bits, so it can represent values from `-128` to `127` (in signed form) or `0` to `255` (in unsigned form).
  - `EOF` (End of File) is usually defined as `-1`, and it's outside the range of normal `char` values.
  - By using an `int`, you can represent all possible character values (0–255 in ASCII) **and** the special `EOF` value. This is why `getchar()` returns an `int`, and why we store the result in `int c`.

How It Handles Characters:

When `getchar()` reads a character from the input buffer, it's just reading the ASCII value of that character and storing it as an `int`. For example:

1. If you input the string `"abc"`, the input buffer will contain the characters `'a'`, `'b'`, `'c'`, and the newline `\n`.
2. `getchar()` reads `'a'`, which has the ASCII value `97`. It returns this value, and it's stored in `int c`.
3. The process continues for the next character in the buffer, `'b'` (ASCII `98`), and so on.

Even though `c` is an `int`, it’s perfectly fine because ASCII values for characters fit into an `int`. The benefit is that it can also store the special value `EOF` when the input ends.

Example to Clarify:

Here’s a breakdown of the process using your example:

```c
int c; while ((c = getchar()) != '\n' && c != EOF); // Reads and discards input characters
```

- Input: `"abc\n"`
  - First `getchar()` call: reads `'a'`, `c = 97`.
  - Second `getchar()` call: reads `'b'`, `c = 98`.
  - Third `getchar()` call: reads `'c'`, `c = 99`.
  - Fourth `getchar()` call: reads `'\n'`, which breaks the loop.

The characters `'a'`, `'b'`, and `'c'` are simply stored as their ASCII integer equivalents in `int c`, which works fine.

Summary:

- **Characters can be stored in `int`** because characters are just small integers (ASCII values).
- `int` is used to allow detection of the special `EOF` marker, which wouldn't fit in a `char`.
- Storing characters in an `int` does not cause any issues; it's a common practice in C when handling input, especially when working with `getchar()`.

#DOUBT **- char have range from -128 to 127 then why it cant store EOF (-1)**

Let's clarify why a `char` in C, which can represent negative values (like `-128` to `127` for a signed `char`), still can't reliably store `EOF` (which is typically `-1`).

Why Can't `char` Store `EOF` Reliably?

1. **Signed vs. Unsigned `char`:**
   - The C standard allows `char` to be either **signed** or **unsigned**, depending on the compiler or platform. This means:
     - **Signed `char`**: Values range from `-128` to `127` (8-bit signed).
     - **Unsigned `char`**: Values range from `0` to `255` (8-bit unsigned).

- **Problem**: On a system where `char` is **unsigned**, `EOF` (which is `-1`) cannot be stored, because an `unsigned char` can only store values from `0` to `255`. This leads to data truncation or misinterpretation.

2. **ASCII Characters Overlap with `EOF`:**
   - In ASCII, the range of valid character values is from `0` to `255` (extended ASCII). If you use a `char` to store a character, the value `255` (which represents a valid character) could **accidentally** be interpreted as `-1` (EOF) if you are using a signed `char` on certain platforms. This ambiguity makes it unreliable.
   - **Example**:
     - Suppose you're on a platform where `char` is signed, and you read a character with the value `255`. In a signed `char`, this would be interpreted as `-1`, which is `EOF`, even though the input was actually a valid character.
3. `int` **Provides a Clear Distinction:**
   - `int` is used by `getchar()` because it has a wider range than `char`. On most platforms, an `int` can store values much larger than `255` and can clearly distinguish between valid character values (which are in the range `0` to `255`) and `EOF` (which is typically `-1`).
   - This eliminates any ambiguity. By using an `int`, `EOF` is clearly `-1`, and all valid character values remain in the range `0` to `255`.

Key Points:

- `EOF = -1` is used to indicate the end of input, but it **must be distinct** from any valid character values.
- A `char` might be either signed or unsigned depending on the system. If it's unsigned, it can't hold negative values like `-1` (EOF). Even if it’s signed, there's a risk of ambiguity when valid character values overlap with `EOF`.
- Using `int` ensures that `EOF` is handled correctly because `int` can store both the full range of valid character values (`0` to `255`) and the special value `-1` for `EOF`.

Example of Potential Issue with `char`:

```
char c;
while ((c = getchar()) != EOF) { //Here 'char c' can’t handle 'EOF' properly
```

```
    // Process character
}
```

- If `char` is **unsigned**, `EOF` (which is `-1`) will never match, since `char` can only hold values `0` to `255`.
- Even if `char` is **signed**, if the character `255` is encountered, it might be misinterpreted as `-1` (EOF), causing the loop to terminate prematurely.

By using `int`, this problem is avoided :

```
int c;
while ((c = getchar()) != EOF) { // Now `EOF` can be detected properly
    // Process character
}
```

Summary:

- A `char` can't reliably store `EOF` because it might be **unsigned** (can't hold negative values) or, even if signed, it might interpret certain valid character values (like `255`) as `EOF`.
- `int` is used to handle input in functions like `getchar()` because it can store all valid character values **and** distinguish `EOF` unambiguously.

This is why using `int` for handling the return value of `getchar()` is essential for reliable behavior!