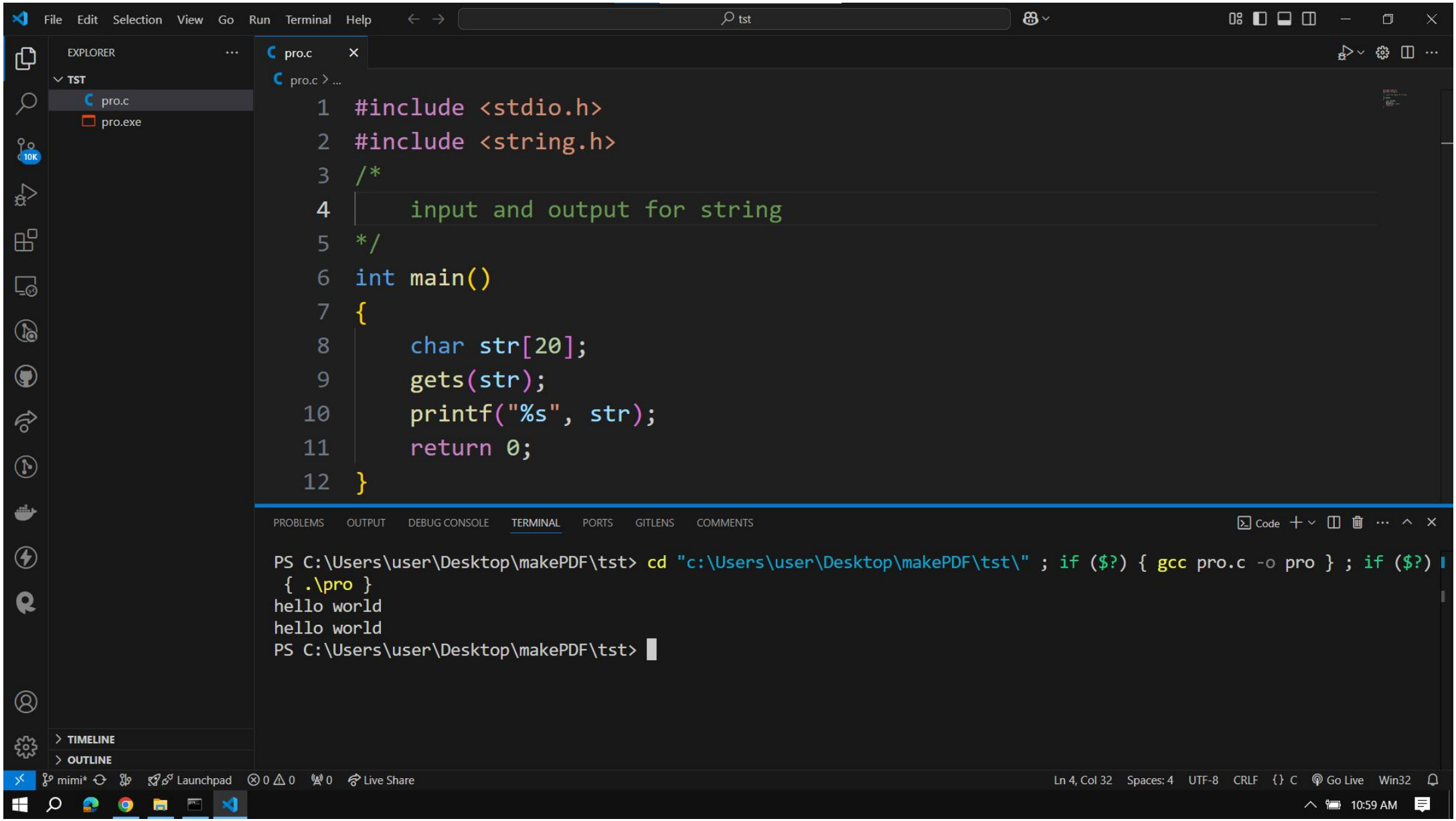
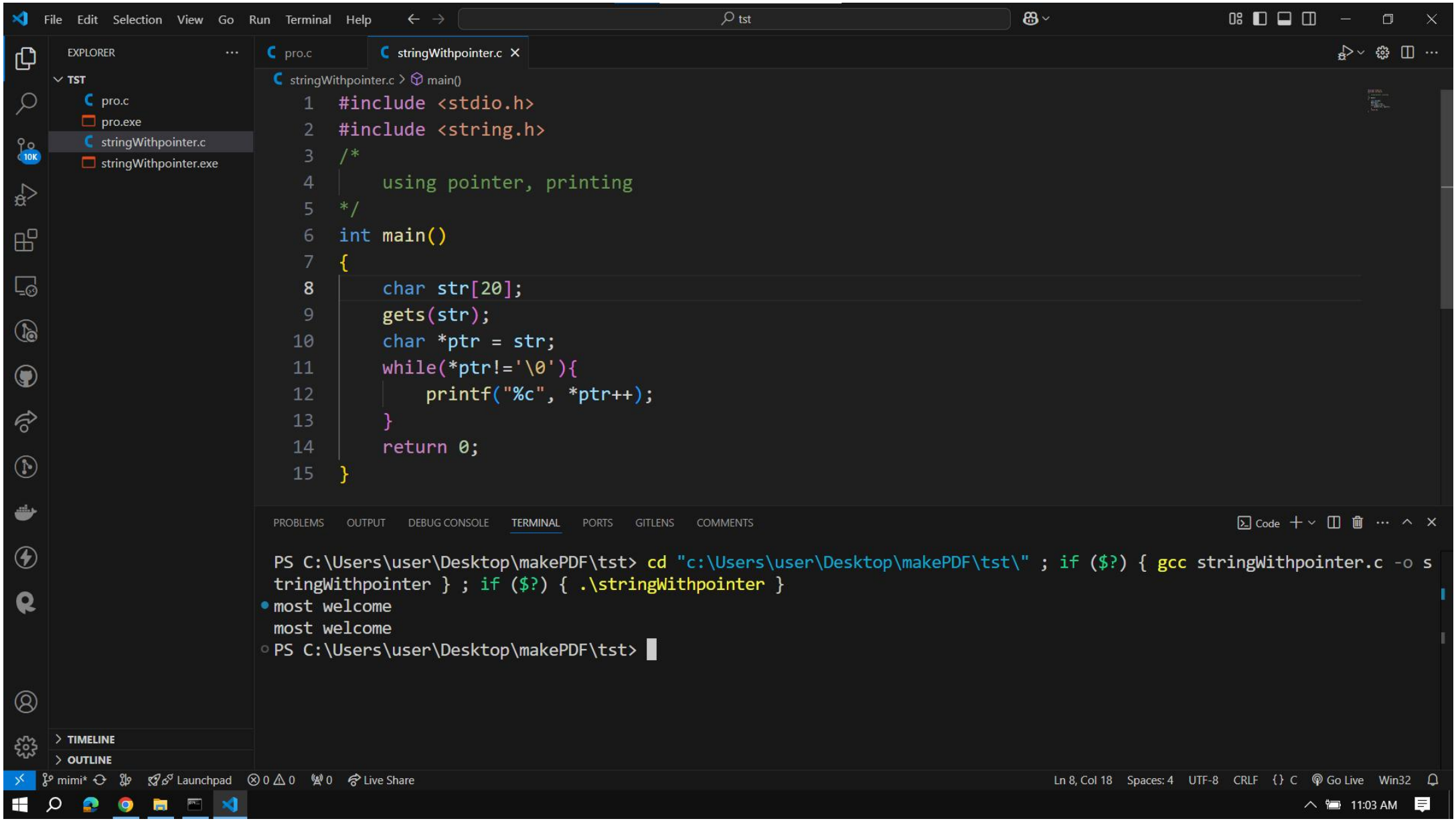
# Strings in C

Last Updated : 10 Jan, 2025

A String in C programming is a sequence of characters terminated with a null character '\0'. The C String is stored as an array of characters. The difference between a character array and a C string is that the string in C is terminated with a unique character '\0'.


File Edit Selection View Go Run Terminal Help
tst
EXPLORER
TST
pro.c
pro.exe
pro.c
pro.c > ...
#include <stdio.h>
#include <string.h>
/*
input and output for string
*/
int main()
{
char str[20];
gets(str);
printf("%s", str);
return 0;
}
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS COMMENTS
Code
PS C:\Users\user\Desktop\makePDF\tst> cd "c:\Users\user\Desktop\makePDF\tst\" ; if ($?) { gcc pro.c -o pro } ; if ($?)
{ .\pro }
hello world
hello world
PS C:\Users\user\Desktop\makePDF\tst>
TIMELINE
OUTLINE
mimi*
Launchpad
0 0
0
Live Share
Ln 4, Col 32
Spaces: 4
UTF-8
CRLF
{} C
Go Live
Win32
10:59 AM


File Edit Selection View Go Run Terminal Help
tst
EXPLORER
TST
pro.c
pro.exe
stringWithpointer.c
stringWithpointer.exe
pro.c
stringWithpointer.c
stringWithpointer.c > main()
#include <stdio.h>
#include <string.h>
/*
using pointer, printing
*/
int main()
{
char str[20];
gets(str);
char *ptr = str;
while(*ptr!='\0'){
printf("%c", *ptr++);
}
return 0;
}
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS COMMENTS
Code
PS C:\Users\user\Desktop\makePDF\tst> cd "c:\Users\user\Desktop\makePDF\tst\" ; if ($?) { gcc stringWithpointer.c -o stringWithpointer } ; if ($?) { .\stringWithpointer }
most welcome
most welcome
PS C:\Users\user\Desktop\makePDF\tst>
TIMELINE
OUTLINE
mimi* Launchpad 0 0 0 Live Share
Ln 8, Col 18 Spaces: 4 UTF-8 CRLF {} C Go Live Win32
11:03 AM

### 3. Assigning Character by Character with Size

We can also assign a string character by character. But we should remember to set the end character as '\0' which is a null character.

```
char str[14] = { 'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};
```

### 4. Assigning Character by Character without Size

We can assign character by character without size with the NULL character at the end. The size of the string is determined by the compiler automatically.

```
char str[] = { 'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};
```

# C Pointers

Last Updated : 23 Dec, 2024

A **pointer** is a variable that stores the **memory address** of another variable. Instead of holding a direct value, it holds the address where the value is stored in memory. There are **2 important operators** that we will use in pointers concepts i.e.

- **Dereferencing operator**(*) used to declare pointer variable and access the value stored in the address.
- **Address operator(&)** used to returns the address of a variable or to access the address of a variable to a pointer.

## Important Points:

- **%p format specifier** is used to print the address stored in pointer variables.
- **Printing a pointer with %d format specifier** may result in a warning or undefined behaviour because the size of a pointer (usually 4 or 8 bytes) may not match that of an integer.
- The memory address format will always be in **hexadecimal format**(starting with 0x).
- C does not use the term "**reference**" explicitly (**unlike C++**), "**referencing**" in C usually refers to obtaining the address of a variable using the **address operator (&)**.
- Pointers are essential for **dynamic memory allocatio**n, providing control over memory usage with functions like **malloc**, **calloc**, and **free.**

### A. Pointer Declaration

To declare a pointer, we use the **(*) dereference operator** before its name. In pointer declaration, we only declare the pointer but do not initialize it.

### B. Pointer Initialization

Pointer initialization is the process where we assign some initial value to the pointer variable. We use the **(&) addressof operator** to get the memory address of a variable and then store it in the pointer variable.

> ***Note:*** *We can also declare and initialize the pointer in a single step. This is called* ***pointer definition.***

### C. Pointer Dereferencing

Dereferencing a pointer is the process of accessing the value stored in the memory address specified in the pointer. We use the same **(*) dereferencing operator** that we used in the pointer declaration.

> ***Note:*** *It is recommended that the pointers should always be initialized to some value before starting using it. Otherwise, it may lead to number of errors.*

## Types of Pointers in C

Pointers in C can be classified into many different types depending on the data it is pointing to:

**1. Integer Pointers**

As the name suggests, these are the pointers that point to the integer values. These pointers are pronounced as **Pointer to Integer.** Similarly, a pointer can point to any primitive data type and is named accordingly.

Similarly, a pointer can point to any primitive data type. It can point also point to derived data types such as arrays and user-defined data types such as structures.

**2. Array Pointer**

Pointers and Array are closely related to each other. Even the array name is the pointer to its first element. They are also known as Pointer to Arrays.

**3. Structure Pointer**

The pointer pointing to the structure type is called Structure Pointer or Pointer to Structure. It can be declared in the same way as we declare the other primitive data types.

**4. Function Pointers**

Function pointers point to the functions. They are different from the rest of the pointers in the sense that instead of pointing to the data, they point to the code.

**5. Double Pointers**

In C language, we can define a pointer that stores the memory address of another pointer. Such pointers are called double-pointers or pointers-to-pointer. Instead of pointing to a data value, they point to another pointer.

In C, we can create multi-level pointers with any number of levels such as – ***ptr3, ****ptr4, ******ptr5 and so on.

**6. NULL Pointer**

The Null Pointers are those pointers that do not point to any memory location. They can be created by assigning a NULL value to the pointer. A pointer of any type can be assigned the NULL value.

**7. Void Pointer**

The Void pointers in C are the pointers of type void. It means that they do not have any associated data type. They are also called **generic pointers** as they can point to any type and can be typecasted to any type.

**8. Wild Pointers**

The Wild Pointers are pointers that have not been initialized with something yet. These types of C-pointers can cause problems in our programs and can eventually cause them to crash. If values are updated using wild pointers, they could cause data abort or data corruption.

**9. Constant Pointers**

In constant pointers, the memory address stored inside the pointer is constant and cannot be modified once it is defined. It will always point to the same memory address.

**10. Pointer to Constant**

The pointers pointing to a constant value that cannot be modified are called pointers to a constant. Here we can only access the data pointed by the pointer, but cannot modify it. Although, we can change the address stored in the pointer to constant.

## Uses of Pointers in C

The C pointer is a very powerful tool that is widely used in C programming to perform various useful operations. It is used to achieve the following functionalities in C:

- Pass Arguments by Pointers
- Accessing Array Elements
- Return Multiple Values from Function
- Dynamic Memory Allocation
- Implementing Data Structures
- In System-Level Programming where memory addresses are useful.
- To use in Control Tables.

## Advantages of Pointers

Following are the major advantages of pointers in C:

- Pointers are used for dynamic memory allocation and deallocation.
- An Array or a structure can be accessed efficiently with pointers
- Pointers are useful for accessing memory locations.
- Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
- Pointers reduce the length of the program and its execution time as well.

## Disadvantages of Pointers

Pointers are vulnerable to errors and have following disadvantages:

- Memory corruption can occur if an incorrect value is provided to pointers.
- Pointers are a little bit complex to understand.
- Pointers are majorly responsible for memory leaks in C.
- Pointers are comparatively slower than variables in C.
- Uninitialized pointers might cause a segmentation fault.

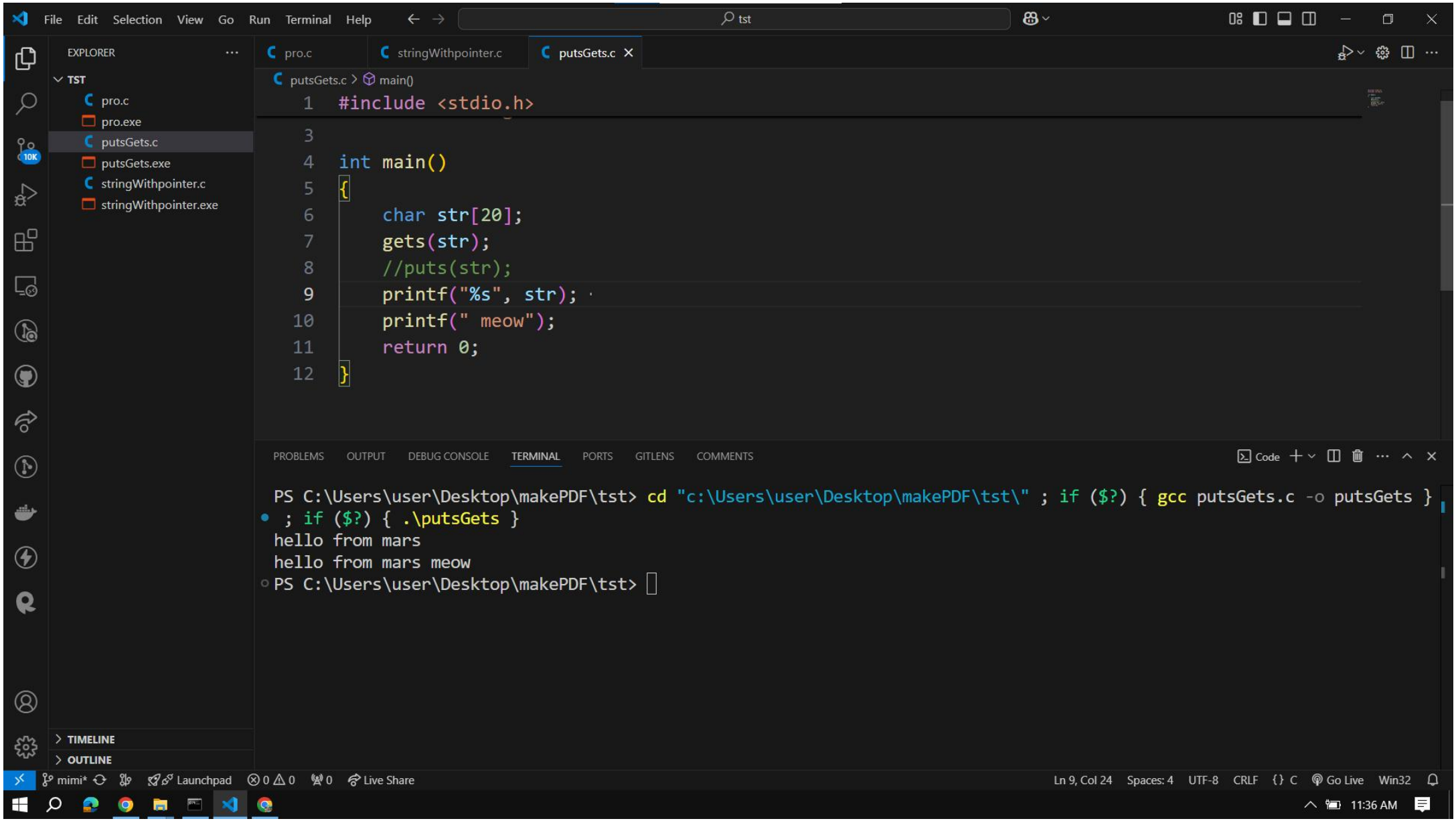
| *Pointer* | *Array* |
| --- | --- |
| *A pointer is a derived data type that can store the address of other variables.* | *An array is a homogeneous collection of items of any type such as int, char, etc.* |
| *Pointers are allocated at run time.* | *Arrays are allocated at runtime.* |
| *The pointer is a single variable.* | *An array is a collection of variables of the same type.* |
| *Dynamic in Nature* | *Static in Nature.* |

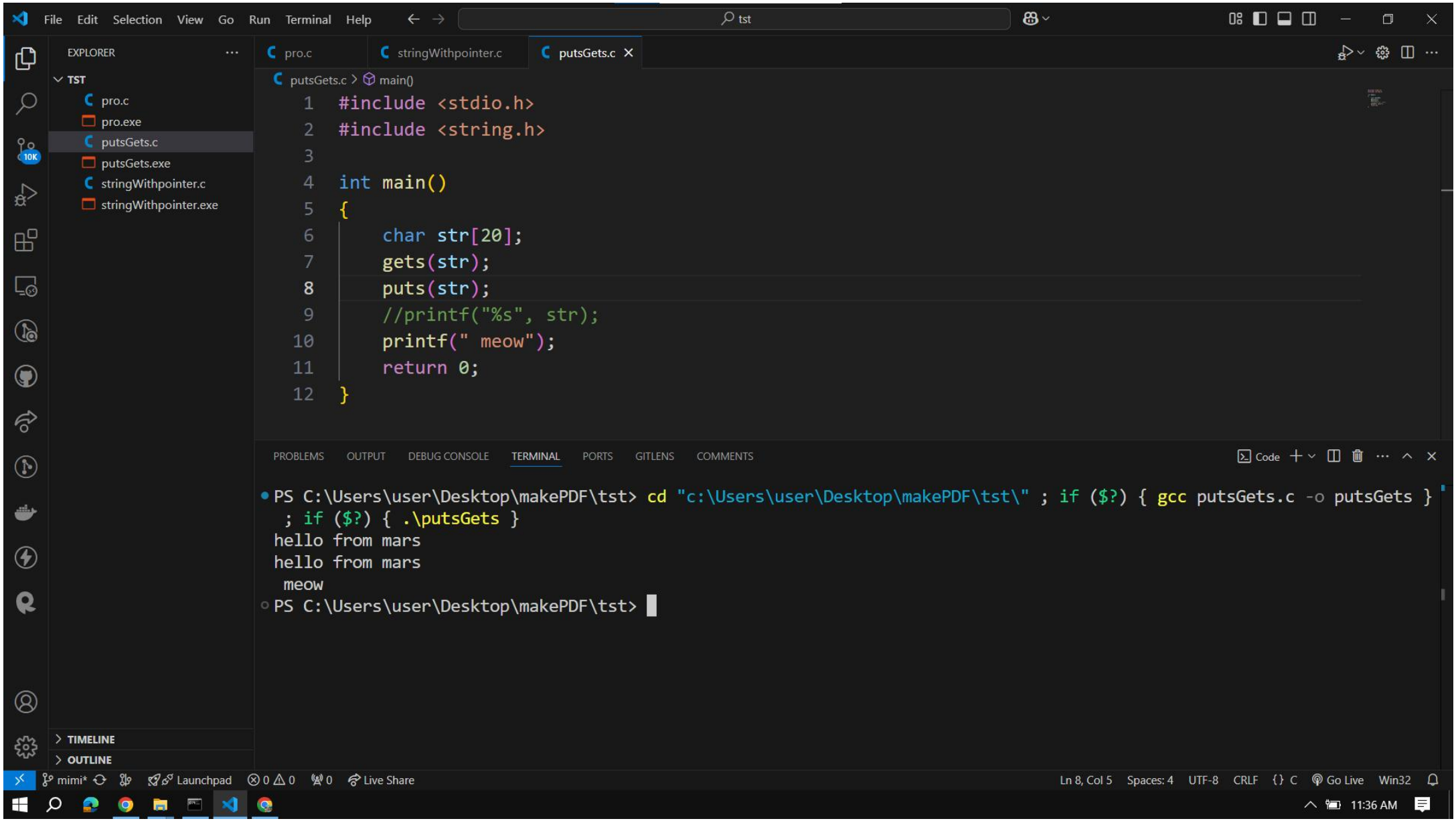
| S.No. | malloc() | calloc() |
|---|---|---|
| 1. | malloc() is a function that creates one block of memory of a fixed size. | calloc() is a function that assigns a specified number of blocks of memory to a single variable. |
| 2. | malloc() only takes one argument | calloc() takes two arguments. |
| 3. | malloc() is faster than calloc. | calloc() is slower than malloc() |
| 4. | malloc() has high time efficiency | calloc() has low time efficiency |
| 5. | malloc() is used to indicate memory allocation | calloc() is used to indicate contiguous memory allocation |
| 6. | Syntax : void* malloc(size_t size); | Syntax : void* calloc(size_t num, size_t size); |
| 7. | malloc() does not initialize the memory to zero | calloc() initializes the memory to zero |
| 8. | malloc() does not add any extra memory overhead | calloc() adds some extra memory overhead |

# What is Memory Leak? How can we avoid?

Last Updated : 21 Jan, 2025

In C, **memory leak** occurs when a program allocates memory dynamically (using functions like malloc(), calloc(), or realloc()) but fails to deallocate it using free() when the memory is no longer needed. Eventually, in the worst case, too much of the available memory may become allocated, all or part of the system or device stops working correctly, the application fails, or the system slows down vastly.


File Edit Selection View Go Run Terminal Help
tst
EXPLORER
TST
pro.c
pro.exe
putsGets.c
putsGets.exe
stringWithpointer.c
stringWithpointer.exe
pro.c
stringWithpointer.c
putsGets.c
putsGets.c > main()
1 #include <stdio.h>
3
4 int main()
5 {
6 char str[20];
7 gets(str);
8 //puts(str);
9 printf("%s", str);
10 printf(" meow");
11 return 0;
12 }
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS COMMENTS
Code
PS C:\Users\user\Desktop\makePDF\tst> cd "c:\Users\user\Desktop\makePDF\tst\" ; if ($?) { gcc putsGets.c -o putsGets } ; if ($?) { .\putsGets }
hello from mars
hello from mars meow
PS C:\Users\user\Desktop\makePDF\tst>
TIMELINE
OUTLINE
mimi*
Launchpad
0 0
0
Live Share
Ln 9, Col 24 Spaces: 4 UTF-8 CRLF { } C Go Live Win32
11:36 AM


File Edit Selection View Go Run Terminal Help
tst
EXPLORER
TST
pro.c
pro.exe
putsGets.c
putsGets.exe
stringWithpointer.c
stringWithpointer.exe
pro.c
stringWithpointer.c
putsGets.c
putsGets.c > main()
#include <stdio.h>
#include <string.h>
int main()
{
char str[20];
gets(str);
puts(str);
//printf("%s", str);
printf(" meow");
return 0;
}
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS COMMENTS
Code
PS C:\Users\user\Desktop\makePDF\tst> cd "c:\Users\user\Desktop\makePDF\tst\" ; if ($?) { gcc putsGets.c -o putsGets } ; if ($?) { .\putsGets }
hello from mars
hello from mars
meow
PS C:\Users\user\Desktop\makePDF\tst>
TIMELINE
OUTLINE
mimi*
Launchpad
Live Share
Ln 8, Col 5
Spaces: 4
UTF-8
CRLF
C
Go Live
Win32
11:36 AM

# C Structures

Last Updated : 21 Jan, 2025

In C, a **structure** is a user-defined data type that can be used to group items of possibly different types into a single type. The **struct** keyword is used to define a structure. The items in the structure are called its **member** and they can be of any valid data type.

The typedef keyword is used to define an alias for the already existing datatype. In structures, we have to use the struct keyword along with the structure name to define the variables. Sometimes, this increases the length and complexity of the code. We can use the typedef to define some new shorter name for the structure.

# C Unions

Last Updated : 10 Jan, 2025

In C, **union** is a user-defined data type that can contain elements of the different data types just like structure. But unlike structures, all the members in the C union are stored in the same memory location. Due to this, only one member can store data at the given point in time.

| Parameter | Structure | Union |
| --- | --- | --- |
| **Definition** | A structure is a user-defined data type that groups different data types into a single entity. | A union is a user-defined data type that allows storing different data types at the same memory location. |
| **Keyword** | The keyword **struct** is used to define a structure | The keyword **union** is used to define a union |
| **Size** | The size is the sum of the sizes of all members, with padding if necessary. | The size is equal to the size of the largest member, with possible padding. |
| **Memory Allocation** | Each member within a structure is allocated unique storage area of location. | Memory allocated is shared by individual members of union. |
| **Data Overlap** | No data overlap as members are independent. | Full data overlap as members shares the same memory. |

# dot (.) Operator in C

Last Updated : 26 Dec, 2024

In C, the **dot (.) operator** is used to access members of user defined data types such as a structure or union. Also known as the direct member access operator, it allows us to access the data of the struct and union via the name of variables.

Let's take a look at an example:

```c
#include <stdio.h>

struct A {
    int x;
};

int main() {
    struct A a = {30};

    // Access structure members using the dot operator
    printf("%d", a.x);

    return 0;
}
```

# C Enums

An **enum** is a special type that represents a group of constants (unchangeable values).

| Aspect | stdio.h | stdlib.h |
| --- | --- | --- |
| **Purpose** | Focuses on input and output operations (reading and writing to console and files). | Provides functions for memory allocation, process control, random number generation, and conversion between data types. |
| **Functions Provided** | Includes functions like **printf(), scanf(), getchar(), and file I/O functions**. | Includes functions like **malloc(), free(), exit(), atoi(), and rand()**. |
| **Scope** | Primarily deals with handling input and output. | Deals with system-level utilities, memory management, and other miscellaneous operations. |
| **Usage** | stdio.h is used by almost every C program. | stdlib.h is only use when we need to allocate memory in our program. |

`<stdio.h>` searches in standard C library locations, whereas `"stdio.h"` searches in the current directory as well.

Ideally, you would use `<...>` for standard C libraries and `"..."` for libraries that you write and are present in the current directory.

```c
// C program to illustrate the use of #define directive
#include <stdio.h>

// Defining a macro for PI
#define PI 3.14159

int main()
{
    // Using the PI macro to calculate
    double radius = 8.0;
    double area = PI * radius * radius;

    // Displaying the calculated area
    printf("Area of the circle is: %f\n", area);

    return 0;
}
```