**Pointers in C Programming with examples**

A **pointer** is a variable that stores the address of another variable. Unlike other variables that hold values of a certain type, pointer holds the address of a variable. For example, an integer variable holds (or you can say stores) an integer value, however an integer pointer holds the address of a integer variable. In this guide, we will discuss pointers in [C programming](https://beginnersbook.com/2014/01/c-tutorial-for-beginners-with-examples/) with the help of examples.

Before we discuss about **pointers in C**, lets take a simple example to understand what do we mean by the address of a variable.

**A simple example to understand how to access the address of a variable without pointers?**

In this program, we have a variable num of int type. The value of num is 10 and this value must be stored somewhere in the memory, right? A memory space is allocated for each variable that holds the value of that variable, this memory space has an address. For example we live in a house and our house has an address, which helps other people to find our house. The same way the value of the variable is stored in a memory address, which helps the C program to find that value when it is needed.

So let’s say the address assigned to variable num is 0x7fff5694dc58, which means whatever value we would be assigning to num should be stored at the location: 0x7fff5694dc58. See the diagram below.

#include <stdio.h>

int main()

{

int num = 10;

printf("Value of variable num is: %d", num);

/\* To print the address of a variable we use %p

\* format specifier and ampersand (&) sign just

\* before the variable name like &num.

\*/

printf("\nAddress of variable num is: %p", &num);

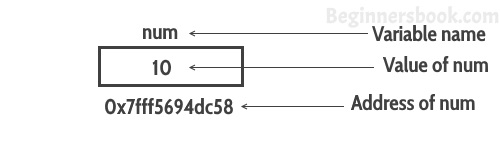
return 0;

}

**Output:**

Value of variable num is: 10

Address of variable num is: 0x7fff5694dc58



**A Simple Example of Pointers in C**

This program shows how a pointer is declared and used. There are several other things that we can do with pointers, we have discussed them later in this guide. For now, we just need to know how to link a pointer to the address of a variable.

**Important point to note is**: The data type of pointer and the variable must match, an int pointer can hold the address of int variable, similarly a pointer declared with float data type can hold the address of a float variable. In the example below, the pointer and the variable both are of int type.

#include <stdio.h>

int main()

{

//Variable declaration

int num = 10;

//Pointer declaration

int \*p;

//Assigning address of num to the pointer p

p = &num;

printf("Address of variable num is: %p", p);

return 0;

}

Output:

Address of variable num is: 0x7fff5694dc58

**C Pointers – Operators that are used with Pointers**

Lets discuss the operators & and \* that are used with Pointers in C.

**“Address of”(&) Operator**

We have already seen in the first example that we can display the address of a variable using ampersand sign. I have used &num to access the address of variable num. The **& operator**is also known as “**Address of**” Operator.

printf("Address of var is: %p", &num);

**Point to note:** %p is a format specifier which is used for displaying the address in hex format.  
Now that you know how to get the address of a variable but **how to store that address in some other variable?** That’s where pointers comes into picture. As mentioned in the beginning of this guide, pointers in C programming are used for holding the address of another variables.

**Pointer is just like another variable, the main difference is that it stores address of another variable rather than a value.**

**“Value at Address”(\*) Operator**

The \* Operator is also known as **Value at address** operator.

**How to declare a pointer?**

int \*p1 /\*Pointer to an integer variable\*/

double \*p2 /\*Pointer to a variable of data type double\*/

char \*p3 /\*Pointer to a character variable\*/

float \*p4 /\*pointer to a float variable\*/

The above are the few examples of pointer declarations. **If you need a pointer to store the address of integer variable then the data type of the pointer should be int**. Same case is with the other data types.

By using \* operator we can access the value of a variable through a pointer.  
For example:

double a = 10;

double \*p;

p = &a;

\*p would give us the value of the variable a. The following statement would display 10 as output.

printf("%d", \*p);

Similarly if we assign a value to \*pointer like this:

\*p = 200;

It would change the value of variable a. The statement above will change the value of a from 10 to 200.

**Example of Pointer demonstrating the use of & and \***

#include <stdio.h>

int main()

{

/\* Pointer of integer type, this can hold the

\* address of a integer type variable.

\*/

int \*p;

int var = 10;

/\* Assigning the address of variable var to the pointer

\* p. The p can hold the address of var because var is

\* an integer type variable.

\*/

p= &var;

printf("Value of variable var is: %d", var);

printf("\nValue of variable var is: %d", \*p);

printf("\nAddress of variable var is: %p", &var);

printf("\nAddress of variable var is: %p", p);

printf("\nAddress of pointer p is: %p", &p);

return 0;

}

Output:

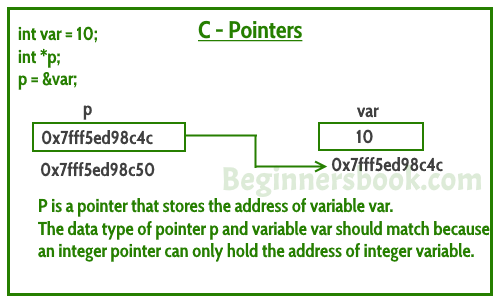
Value of variable var is: 10

Value of variable var is: 10

Address of variable var is: 0x7fff5ed98c4c

Address of variable var is: 0x7fff5ed98c4c

Address of pointer p is: 0x7fff5ed98c50



Lets take few more examples to understand it better –  
Lets say we have a char variable ch and a pointer ptr that holds the address of ch.

char ch='a';

char \*ptr;

**Read the value of ch**

printf("Value of ch: %c", ch);

or

printf("Value of ch: %c", \*ptr);

**Change the value of ch**

ch = 'b';

or

\*ptr = 'b';

The above code would replace the value ‘a’ with ‘b’.

**Can you guess the output of following C program?**

#include <stdio.h>

int main()

{

int var =10;

int \*p;

p= &var;

printf ( "Address of var is: %p", &var);

printf ( "\nAddress of var is: %p", p);

printf ( "\nValue of var is: %d", var);

printf ( "\nValue of var is: %d", \*p);

printf ( "\nValue of var is: %d", \*( &var));

/\* Note I have used %p for p's value as it represents an address\*/

printf( "\nValue of pointer p is: %p", p);

printf ( "\nAddress of pointer p is: %p", &p);

return 0;

}

**Output:**

Address of var is: 0x7fff5d027c58

Address of var is: 0x7fff5d027c58

Value of var is: 10

Value of var is: 10

Value of var is: 10

Value of pointer p is: 0x7fff5d027c58

Address of pointer p is: 0x7fff5d027c50

**More Topics on Pointers**

1) [**Pointer to Pointer**](https://beginnersbook.com/2014/01/c-pointer-to-pointer/) – A pointer can point to another pointer (which means it can store the address of another pointer), such pointers are known as double pointer OR pointer to pointer.

2) [**Passing pointers to function**](https://beginnersbook.com/2014/01/c-passing-pointers-to-functions/) – Pointers can also be passed as an argument to a function, using this feature a function can be called by reference as well as an array can be passed to a function while calling.

3) [**Function pointers**](https://beginnersbook.com/2014/01/c-function-pointers/) – A function pointer is just like another pointer, it is used for storing the address of a function. Function pointer can also be used for calling a function in C program.

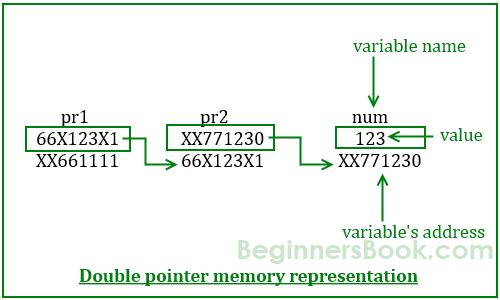
# C – Pointer to Pointer (Double Pointer) with example

We already know that a pointer holds the address of another variable of same type. When a pointer holds the address of another pointer then such type of pointer is known as **pointer-to-pointer** or **double pointer**. In this guide, we will learn what is a double pointer, how to declare them and how to use them in C programming. To understand this concept, you should know the [basics of pointers](https://beginnersbook.com/2014/01/c-pointers/).

## How to declare a Pointer to Pointer (Double Pointer) in C?

int \*\*pr;

Here pr is a double pointer. There must be two \*’s in the declaration of double pointer.

Let’s understand the concept of double pointers with the help of a diagram:  


As per the diagram, pr2 is a normal pointer that holds the address of an integer variable num. There is another pointer pr1 in the diagram that holds the address of another pointer pr2, the pointer pr1 here is a pointer-to-pointer (or double pointer).

**Values from above diagram:**

Variable num has address: XX771230

Address of Pointer pr1 is: XX661111

Address of Pointer pr2 is: 66X123X1

## Example of double Pointer

Lets write a C program based on the diagram that we have seen above.

#include <stdio.h>

int main()

{

int num=123;

//A normal pointer pr2

int \*pr2;

//This pointer pr2 is a double pointer

int \*\*pr1;

/\* Assigning the address of variable num to the

\* pointer pr2

\*/

pr2 = #

/\* Assigning the address of pointer pr2 to the

\* pointer-to-pointer pr1

\*/

pr1 = &pr2;

/\* Possible ways to find value of variable num\*/

printf("\n Value of num is: %d", num);

printf("\n Value of num using pr2 is: %d", \*pr2);

printf("\n Value of num using pr1 is: %d", \*\*pr1);

/\*Possible ways to find address of num\*/

printf("\n Address of num is: %p", &num);

printf("\n Address of num using pr2 is: %p", pr2);

printf("\n Address of num using pr1 is: %p", \*pr1);

/\*Find value of pointer\*/

printf("\n Value of Pointer pr2 is: %p", pr2);

printf("\n Value of Pointer pr2 using pr1 is: %p", \*pr1);

/\*Ways to find address of pointer\*/

printf("\n Address of Pointer pr2 is:%p",&pr2);

printf("\n Address of Pointer pr2 using pr1 is:%p",pr1);

/\*Double pointer value and address\*/

printf("\n Value of Pointer pr1 is:%p",pr1);

printf("\n Address of Pointer pr1 is:%p",&pr1);

return 0;

}

**Output:**

Value of num is: 123

Value of num using pr2 is: 123

Value of num using pr1 is: 123

Address of num is: XX771230

Address of num using pr2 is: XX771230

Address of num using pr1 is: XX771230

Value of Pointer pr2 is: XX771230

Value of Pointer pr2 using pr1 is: XX771230

Address of Pointer pr2 is: 66X123X1

Address of Pointer pr2 using pr1 is: 66X123X1

Value of Pointer pr1 is:  66X123X1

Address of Pointer pr1 is: XX661111

There are some confusions regarding the output of this program, when you run this program you would see the address similar to this: 0x7fff54da7c58. The reason I have given the address in different format is because I want you to relate this program with the diagram above. I have used the exact address values in the above diagram so that it would be easy for you to relate the output of this program with the above diagram.

You can also understand the program logic with these simple equations:

num == \*pr2 == \*\*pr1

&num == pr2 == \*pr1

&pr2 == pr1

# Passing pointer to a function in C with example

## Example: Passing Pointer to a Function in C Programming

In this example, we are passing a pointer to a function. When we pass a pointer as an argument instead of a variable then the address of the variable is passed instead of the value. So any change made by the function using the pointer is permanently made at the address of passed variable. This technique is known as call by reference in C.

Try this same program without pointer, you would find that the bonus amount will not reflect in the salary, this is because the change made by the function would be done to the local variables of the function. When we use pointers, the value is changed at the address of variable

#include <stdio.h>

void salaryhike(int \*var, int b)

{

\*var = \*var+b;

}

int main()

{

int salary=0, bonus=0;

printf("Enter the employee current salary:");

scanf("%d", &salary);

printf("Enter bonus:");

scanf("%d", &bonus);

salaryhike(&salary, bonus);

printf("Final salary: %d", salary);

return 0;

}

**Output:**

Enter the employee current salary:10000

Enter bonus:2000

Final salary: 12000

## Example 2: Swapping two numbers using Pointers

This is one of the most popular example that shows how to swap numbers using call by reference.

Try this program without pointers, you would see that the numbers are not swapped. The reason is same that we have seen above in the first example.

#include <stdio.h>

void swapnum(int \*num1, int \*num2)

{

int tempnum;

tempnum = \*num1;

\*num1 = \*num2;

\*num2 = tempnum;

}

int main( )

{

int v1 = 11, v2 = 77 ;

printf("Before swapping:");

printf("\nValue of v1 is: %d", v1);

printf("\nValue of v2 is: %d", v2);

/\*calling swap function\*/

swapnum( &v1, &v2 );

printf("\nAfter swapping:");

printf("\nValue of v1 is: %d", v1);

printf("\nValue of v2 is: %d", v2);

}

**Output:**

Before swapping:

Value of v1 is: 11

Value of v2 is: 77

After swapping:

Value of v1 is: 77

Value of v2 is: 11

# C – Function Pointer with examples

In [C programming language](https://beginnersbook.com/2014/01/c-tutorial-for-beginners-with-examples/), we can have a concept of Pointer to a function known as **function pointer in C**. In this tutorial, we will learn how to declare a function pointer and how to call a function using this pointer. To understand this concept, you should have the basic knowledge of [Functions](https://beginnersbook.com/2014/01/c-functions-examples/) and [Pointers in C](https://beginnersbook.com/2014/01/c-pointers/).

## How to declare a function pointer?

function\_return\_type(\*Pointer\_name)(function argument list)

For example:

double  (\*p2f)(double, char)

Here double is a return type of function, p2f is name of the function pointer and (double, char) is an argument list of this function. Which means the first argument of this function is of double type and the second argument is char type.

Lets understand this with the help of an example: Here we have a function sum that calculates the sum of two numbers and returns the sum. We have created a pointer f2p that points to this function, we are invoking the function using this function pointer f2p.

int sum (int num1, int num2)

{

return num1+num2;

}

int main()

{

/\* The following two lines can also be written in a single

\* statement like this: void (\*fun\_ptr)(int) = &fun;

\*/

int (\*f2p) (int, int);

f2p = sum;

//Calling function using function pointer

int op1 = f2p(10, 13);

//Calling function in normal way using function name

int op2 = sum(10, 13);

printf("Output1: Call using function pointer: %d",op1);

printf("\nOutput2: Call using function name: %d", op2);

return 0;

}

**Output:**

Output1: Call using function pointer: 23

Output2: Call using function name: 23

**Some points regarding function pointer:**  
1. As mentioned in the comments, you can declare a function pointer and assign a function to it in a single statement like this:

void (\*fun\_ptr)(int) = &fun;

2. You can even remove the ampersand from this statement because a function name alone represents the function address. This means the above statement can also be written like this:

void (\*fun\_ptr)(int) = fun;

# Pointer and Array in C programming with example

## A simple example to print the address of array elements

#include <stdio.h>

int main( )

{

int val[7] = { 11, 22, 33, 44, 55, 66, 77 } ;

/\* for loop to print value and address of each element of array\*/

for ( int i = 0 ; i < 7 ; i++ )

{

/\* The correct way of displaying the address would be using %p format

\* specifier like this:

\* printf("val[%d]: value is %d and address is %p\n", i, val[i], &val[i]);

\* Just to demonstrate that the array elements are stored in contiguous

\* locations, I m displaying the addresses in integer

\*/

printf("val[%d]: value is %d and address is %d\n", i, val[i], &val[i]);

}

return 0;

}

**Output:**

val[0]: value is 11 and address is 1423453232

val[1]: value is 22 and address is 1423453236

val[2]: value is 33 and address is 1423453240

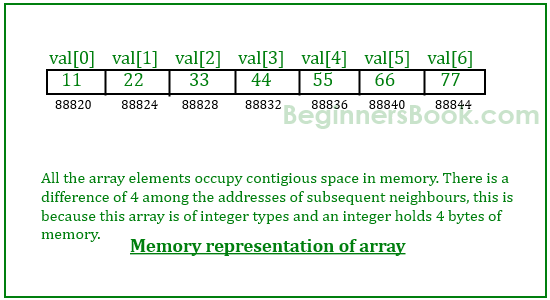
val[3]: value is 44 and address is 1423453244

val[4]: value is 55 and address is 1423453248

val[5]: value is 66 and address is 1423453252

val[6]: value is 77 and address is 1423453256

Note that there is a difference of 4 bytes between each element because that’s the size of an integer. Which means all the elements are stored in consecutive contiguous memory locations in the memory.(See the diagram below)



In the above example I have used &val[i] to get the address of ith element of the array. We can also use a pointer variable instead of using the ampersand (&) to get the address.

## Example – Array and Pointer Example in C

#include <stdio.h>

int main( )

{

/\*Pointer variable\*/

int \*p;

/\*Array declaration\*/

int val[7] = { 11, 22, 33, 44, 55, 66, 77 } ;

/\* Assigning the address of val[0] the pointer

\* You can also write like this:

\* p = var;

\* because array name represents the address of the first element

\*/

p = &val[0];

for ( int i = 0 ; i<7 ; i++ )

{

printf("val[%d]: value is %d and address is %p\n", i, \*p, p);

/\* Incrementing the pointer so that it points to next element

\* on every increment.

\*/

p++;

}

return 0;

}

Output:

val[0]: value is 11 and address is 0x7fff51472c30

val[1]: value is 22 and address is 0x7fff51472c34

val[2]: value is 33 and address is 0x7fff51472c38

val[3]: value is 44 and address is 0x7fff51472c3c

val[4]: value is 55 and address is 0x7fff51472c40

val[5]: value is 66 and address is 0x7fff51472c44

val[6]: value is 77 and address is 0x7fff51472c48

**Points to Note:**  
1) While using pointers with array, the data type of the pointer must match with the data type of the array.  
2) You can also use array name to initialize the pointer like this:

p = var;

because the array name alone is equivalent to the base address of the array.

val==&val[0];

3) In the loop the increment operation(p++) is performed on the pointer variable to get the next location (next element’s location), this arithmetic is same for all types of arrays (for all data types double, char, int etc.) even though the bytes consumed by each data type is different.

**Pointer logic**  
You must have understood the logic in above code so now its time to play with few pointer arithmetic and expressions.

if p = &val[0] which means

\*p ==val[0]

(p+1) == &val[2]  & \*(p+1) == val[2]

(p+2) == &val[3]  & \*(p+2) == val[3]

(p+n) == &val[n+1) & \*(p+n) == val[n+1]

Using this logic we can rewrite our code in a better way like this:

#include <stdio.h>

int main( )

{

int \*p;

int val[7] = { 11, 22, 33, 44, 55, 66, 77 } ;

p = val;

for ( int i = 0 ; i<7 ; i++ )

{

printf("val[%d]: value is %d and address is %p\n", i, \*(p+i), (p+i));

}

return 0;

}

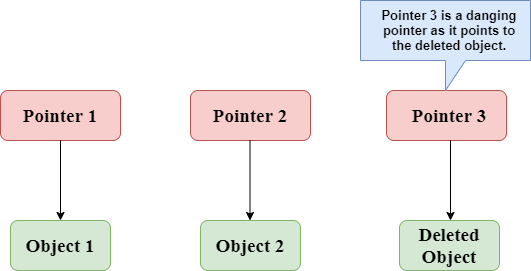
We don’t need the p++ statement in this program.

Dangling Pointers in C

The most common bugs related to pointers and memory management is dangling/wild pointers. Sometimes the programmer fails to initialize the pointer with a valid address, then this type of initialized pointer is known as a dangling pointer in C.

Dangling pointer occurs at the time of the object destruction when the object is deleted or de-allocated from memory without modifying the value of the pointer. In this case, the pointer is pointing to the memory, which is de-allocated. The dangling pointer can point to the memory, which contains either the program code or the code of the operating system. If we assign the value to this pointer, then it overwrites the value of the program code or operating system instructions; in such cases, the program will show the undesirable result or may even crash. If the memory is re-allocated to some other process, then we dereference the dangling pointer will cause the segmentation faults.

**Let's observe the following examples.**



In the above figure, we can observe that the **Pointer 3** is a dangling pointer. **Pointer 1** and **Pointer 2** are the pointers that point to the allocated objects, i.e., Object 1 and Object 2, respectively. **Pointer 3** is a dangling pointer as it points to the de-allocated object.

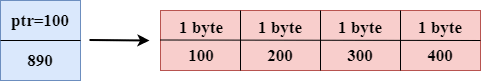
**Let's understand the dangling pointer through some C programs.**

**Using free() function to de-allocate the memory.**

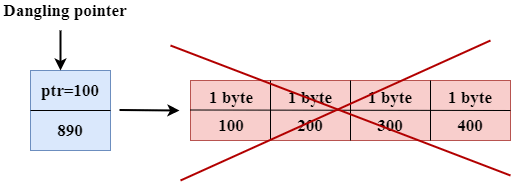
1. #include <stdio.h>
2. **int** main()
3. {
4. **int** \*ptr=(**int** \*)malloc(sizeof(**int**));
5. **int** a=560;
6. ptr=&a;
7. free(ptr);
8. **return** 0;
9. }

In the above code, we have created two variables, i.e., \*ptr and a where 'ptr' is a pointer and 'a' is a integer variable. The \*ptr is a pointer variable which is created with the help of **malloc()** function. As we know that malloc() function returns void, so we use int \* to convert void pointer into int pointer.

The statement **int \*ptr=(int \*)malloc(sizeof(int));** will allocate the memory with 4 bytes shown in the below image:



The statement **free(ptr)** de-allocates the memory as shown in the below image with a cross sign, and 'ptr' pointer becomes dangling as it is pointing to the de-allocated memory.



If we assign the NULL value to the 'ptr', then 'ptr' will not point to the deleted memory. Therefore, we can say that ptr is not a dangling pointer, as shown in the below image:



**Variable goes out of the scope**

When the variable goes out of the scope then the pointer pointing to the variable becomes a **dangling pointer.**

1. #include<stdio.h>
2. **int** main()
3. {
4. **char** \*str;
5. {
6. **char** a = ‘A’;
7. str = &a;
8. }
9. // a falls out of scope
10. // str is now a dangling pointer
11. printf("%s", \*str);
12. }

**In the above code, we did the following steps:**

* First, we declare the pointer variable named 'str'.
* In the inner scope, we declare a character variable. The str pointer contains the address of the variable 'a'.
* When the control comes out of the inner scope, 'a' variable will no longer be available, so str points to the de-allocated memory. It means that the str pointer becomes the dangling pointer.

**Function call**

Now, we will see how the pointer becomes dangling when we call the function.

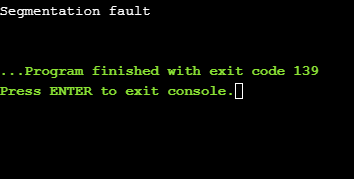
**Let's understand through an example.**

1. #include <stdio.h>
2. **int** \*fun(){
3. **int** y=10;
4. **return** &y;
5. }
6. **int** main()
7. {
8. **int** \*p=fun();
9. printf("%d", \*p);
10. **return** 0;
11. }

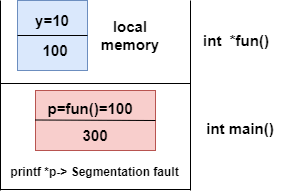
**In the above code, we did the following steps:**

* First, we create the **main()** function in which we have declared **'p'** pointer that contains the return value of the **fun()**.
* When the **fun()** is called, then the control moves to the context of the **int \*fun(),** the **fun()**returns the address of the 'y' variable.
* When control comes back to the context of the **main()** function, it means the variable **'y'** is no longer available. Therefore, we can say that the **'p'** pointer is a dangling pointer as it points to the de-allocated memory.

**Output**



**Let's represent the working of the above code diagrammatically.**

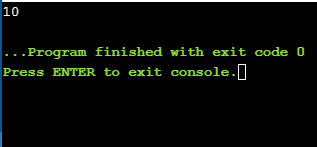


**Let's consider another example of a dangling pointer.**

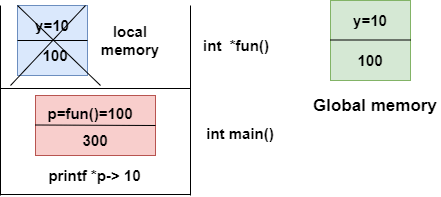
1. #include <stdio.h>
2. **int** \*fun()
3. {
4. **static** **int** y=10;
5. **return** &y;
6. }
7. **int** main()
8. {
9. **int** \*p=fun();
10. printf("%d", \*p);
11. **return** 0;
12. }

The above code is similar to the previous one but the only difference is that the variable 'y' is static. We know that static variable stores in the global memory.

**Output**



Now, we represent the working of the above code diagrammatically.



The above diagram shows the stack memory. First, **the fun()** function is called, then the control moves to the context of the **int \*fun().** As 'y' is a static variable, so it stores in the global memory; Its scope is available throughout the program. When the address value is returned, then the control comes back to the context of the **main().** The pointer 'p' contains the address of 'y', i.e., 100. When we print the value of '\*p', then it prints the value of 'y', i.e., 10. Therefore, we can say that the pointer 'p' is not a dangling pointer as it contains the address of the variable which is stored in the global memory.

**Avoiding Dangling Pointer Errors**

The dangling pointer errors can be avoided by initializing the pointer to the **NULL** value. If we assign the **NULL** value to the pointer, then the pointer will not point to the de-allocated memory. Assigning **NULL** value to the pointer means that the pointer is not pointing to any memory location.

sizeof() operator in C

The **sizeof()** operator is commonly used in C. It determines the size of the expression or the data type specified in the number of char-sized storage units. The **sizeof()** operator contains a single operand which can be either an expression or a data typecast where the cast is data type enclosed within parenthesis. The data type cannot only be primitive data types such as integer or floating data types, but it can also be pointer data types and compound data types such as unions and structs.

Need of sizeof() operator: Mainly, programs know the storage size of the primitive data types. Though the storage size of the data type is constant, it varies when implemented in different platforms. For example, we dynamically allocate the array space by using **sizeof()** operator:

1. **int** \*ptr=malloc(10\***sizeof**(**int**));

In the above example, we use the sizeof() operator, which is applied to the cast of type int. We use **malloc()** function to allocate the memory and returns the pointer which is pointing to this allocated memory. The memory space is equal to the number of bytes occupied by the int data type and multiplied by 10.

Note:  
The output can vary on different machines such as on 32-bit operating system will show different output, and the 64-bit operating system will show the different outputs of the same data types.

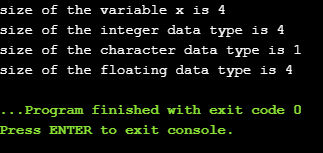
The **sizeof()** operator behaves differently according to the type of the operand.

* **Operand is a data type**
* **Operand is an expression**

When operand is a data type.

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** x=89;    // variable declaration.
5. printf("size of the variable x is %d", **sizeof**(x));  // Displaying the size of ?x? variable.
6. printf("\nsize of the integer data type is %d",**sizeof**(**int**)); //Displaying the size of integer data type.
7. printf("\nsize of the character data type is %d",**sizeof**(**char**)); //Displaying the size of character data type.
9. printf("\nsize of the floating data type is %d",**sizeof**(**float**)); //Displaying the size of floating data type.
10. **return** 0;
11. }

In the above code, we are printing the size of different data types such as int, char, float with the help of **sizeof()** operator.



When operand is an expression

1. #include <stdio.h>
2. **int** main()
3. {
4. **double** i=78.0; //variable initialization.
5. **float** j=6.78; //variable initialization.
6. printf("size of (i+j) expression is : %d",**sizeof**(i+j)); //Displaying the size of the expression (i+j).
7. **return** 0;
8. }

In the above code, we have created two variables 'i' and 'j' of type double and float respectively, and then we print the size of the expression by using **sizeof(i+j)** operator.

size of (i+j) expression is : 8

const Pointer in C

Constant Pointers

A constant pointer in C cannot change the address of the variable to which it is pointing, i.e., the address will remain constant. Therefore, we can say that if a constant pointer is pointing to some variable, then it cannot point to any other variable.

Syntax of Constant Pointer

1. <type of pointer> \***const** <name of pointer>;

**Declaration of a constant pointer is given below:**

1. **int** \***const** ptr;

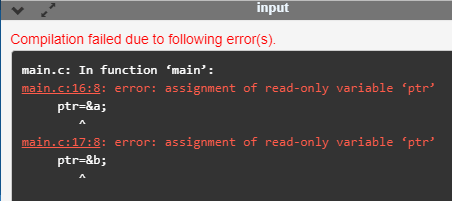
**Let's understand the constant pointer through an example.**

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a=1;
5. **int** b=2;
6. **int** \***const** ptr;
7. ptr=&a;
8. ptr=&b;
9. printf("Value of ptr is :%d",\*ptr);
10. **return** 0;
11. }

**In the above code:**

* We declare two variables, i.e., a and b with values 1 and 2, respectively.
* We declare a constant pointer.
* First, we assign the address of variable 'a' to the pointer 'ptr'.
* Then, we assign the address of variable 'b' to the pointer 'ptr'.
* Lastly, we try to print the value of the variable pointed by the 'ptr'.

**Output**



In the above output, we can observe that the above code produces the error "assignment of read-only variable 'ptr'". It means that the value of the variable 'ptr' which 'ptr' is holding cannot be changed. In the above code, we are changing the value of 'ptr' from &a to &b, which is not possible with constant pointers. Therefore, we can say that the constant pointer, which points to some variable, cannot point to another variable.

Pointer to Constant

A pointer to constant is a pointer through which the value of the variable that the pointer points cannot be changed. The address of these pointers can be changed, but the value of the variable that the pointer points cannot be changed.

Syntax of Pointer to Constant

1. **const** <type of pointer>\* <name of pointer>

**Declaration of a pointer to constant is given below:**

1. **const** **int**\* ptr;

**Let's understand through an example.**

* **First, we write the code where we are changing the value of a pointer**

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a=100;
5. **int** b=200;
6. **const** **int**\* ptr;
7. ptr=&a;
8. ptr=&b;
9. printf("Value of ptr is :%u",ptr);
10. **return** 0;
11. }

**In the above code:**

* We declare two variables, i.e., a and b with the values 100 and 200 respectively.
* We declare a pointer to constant.
* First, we assign the address of variable 'a' to the pointer 'ptr'.
* Then, we assign the address of variable 'b' to the pointer 'ptr'.
* Lastly, we try to print the value of 'ptr'.

**Output**

const Pointer in C

The above code runs successfully, and it shows the value of 'ptr' in the output.

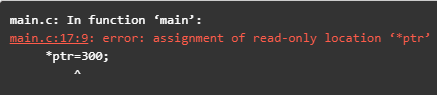
* Now, we write the code in which we are changing the value of the variable to which the pointer points.

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a=100;
5. **int** b=200;
6. **const** **int**\* ptr;
7. ptr=&b;
8. \*ptr=300;
9. printf("Value of ptr is :%d",\*ptr);
10. **return** 0;
11. }

**In the above code:**

* We declare two variables, i.e., 'a' and 'b' with the values 100 and 200 respectively.
* We declare a pointer to constant.
* We assign the address of the variable 'b' to the pointer 'ptr'.
* Then, we try to modify the value of the variable 'b' through the pointer 'ptr'.
* Lastly, we try to print the value of the variable which is pointed by the pointer 'ptr'.

**Output**



The above code shows the error "assignment of read-only location '\*ptr'". This error means that we cannot change the value of the variable to which the pointer is pointing.

Constant Pointer to a Constant

A constant pointer to a constant is a pointer, which is a combination of the above two pointers. It can neither change the address of the variable to which it is pointing nor it can change the value placed at this address.

Syntax

1. **const** <type of pointer>\* **const** <name of the pointer>;

**Declaration for a constant pointer to a constant is given below:**

1. **const** **int**\* **const** ptr;

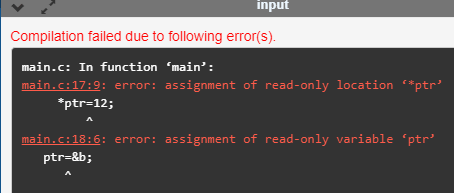
**Let's understand through an example.**

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a=10;
5. **int** b=90;
6. **const** **int**\* **const** ptr=&a;
7. \*ptr=12;
8. ptr=&b;
9. printf("Value of ptr is :%d",\*ptr);
10. **return** 0;
11. }

**In the above code:**

* We declare two variables, i.e., 'a' and 'b' with the values 10 and 90, respectively.
* We declare a constant pointer to a constant and then assign the address of 'a'.
* We try to change the value of the variable 'a' through the pointer 'ptr'.
* Then we try to assign the address of variable 'b' to the pointer 'ptr'.
* Lastly, we print the value of the variable, which is pointed by the pointer 'ptr'.

**Output**



The above code shows the error "assignment of read-only location '\*ptr'" and "assignment of read-only variable 'ptr'". Therefore, we conclude that the constant pointer to a constant can change neither address nor value, which is pointing by this pointer.

void pointer in C

Till now, we have studied that the address assigned to a pointer should be of the same type as specified in the pointer declaration. For example, if we declare the int pointer, then this int pointer cannot point to the float variable or some other type of variable, i.e., it can point to only int type variable. To overcome this problem, we use a pointer to void. A pointer to void means a generic pointer that can point to any data type. We can assign the address of any data type to the void pointer, and a void pointer can be assigned to any type of the pointer without performing any explicit typecasting.

Syntax of void pointer

1. **void** \*pointer name;

**Declaration of the void pointer is given below:**

1. **void** \*ptr;

In the above declaration, the void is the type of the pointer, and 'ptr' is the name of the pointer.

**Let us consider some examples:**

int i=9;         // integer variable initialization.

int \*p;         // integer pointer declaration.

float \*fp;         // floating pointer declaration.

void \*ptr;         // void pointer declaration.

p=fp;         // incorrect.

fp=&i;         // incorrect

ptr=p;         // correct

ptr=fp;         // correct

ptr=&i;         // correct

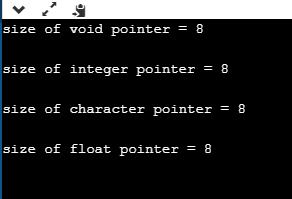
Size of the void pointer in C

The size of the void pointer in C is the same as the size of the pointer of character type. According to C perception, the representation of a pointer to void is the same as the pointer of character type. The size of the pointer will vary depending on the platform that you are using.

**Let's look at the below example:**

1. #include <stdio.h>
2. **int** main()
3. {
4. **void** \*ptr = NULL; //void pointer
5. **int** \*p  = NULL;// integer pointer
6. **char** \*cp = NULL;//character pointer
7. **float** \*fp = NULL;//float pointer
8. //size of void pointer
9. printf("size of void pointer = %d\n\n",**sizeof**(ptr));
10. //size of integer pointer
11. printf("size of integer pointer = %d\n\n",**sizeof**(p));
12. //size of character pointer
13. printf("size of character pointer = %d\n\n",**sizeof**(cp));
14. //size of float pointer
15. printf("size of float pointer = %d\n\n",**sizeof**(fp));
16. **return** 0;
17. }

**Output**



Advantages of void pointer

**Following are the advantages of a void pointer:**

* The malloc() and calloc() function return the void pointer, so these functions can be used to allocate the memory of any data type.

1. #include <stdio.h>
2. #include<malloc.h>
3. **int** main()
4. {
5. **int** a=90;
7. **int** \*x = (**int**\*)malloc(**sizeof**(**int**)) ;
8. x=&a;
9. printf("Value which is pointed by x pointer : %d",\*x);
10. **return** 0;
11. }

**Output**

void pointer in C

* The void pointer in C can also be used to implement the generic functions in C.

**Some important points related to void pointer are:**

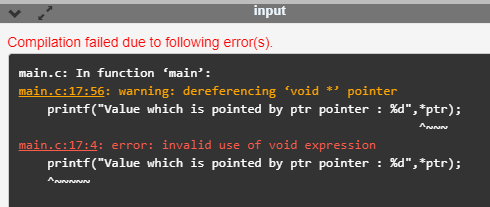
* **Dereferencing a void pointer in C**

The void pointer in C cannot be dereferenced directly. Let's see the below example.

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a=90;
5. **void** \*ptr;
6. ptr=&a;
7. printf("Value which is pointed by ptr pointer : %d",\*ptr);
8. **return** 0;
9. }

In the above code, \*ptr is a void pointer which is pointing to the integer variable 'a'. As we already know that the void pointer cannot be dereferenced, so the above code will give the compile-time error because we are printing the value of the variable pointed by the pointer 'ptr' directly.

**Output**



Now, we rewrite the above code to remove the error.

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a=90;
5. **void** \*ptr;
6. ptr=&a;
7. printf("Value which is pointed by ptr pointer : %d",\*(**int**\*)ptr);
8. **return** 0;
9. }

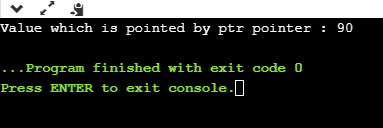
In the above code, we typecast the void pointer to the integer pointer by using the statement given below:

**(int\*)ptr;**

Then, we print the value of the variable which is pointed by the void pointer 'ptr' by using the statement given below:

**\*(int\*)ptr;**

**Output**



* **Arithmetic operation on void pointers**

We cannot apply the arithmetic operations on void pointers in C directly. We need to apply the proper typecasting so that we can perform the arithmetic operations on the void pointers.

**Let's see the below example:**

1. #include<stdio.h>
2. **int** main()
3. {
4. **float** a[4]={6.1,2.3,7.8,9.0};
5. **void** \*ptr;
6. ptr=a;
7. **for**(**int** i=0;i<4;i++)
8. {
9. printf("%f,",\*ptr);
10. ptr=ptr+1;         // Incorrect.
12. }}

The above code shows the compile-time error that "**invalid use of void expression**" as we cannot apply the arithmetic operations on void pointer directly, i.e., ptr=ptr+1.

**Let's rewrite the above code to remove the error.**

1. #include<stdio.h>
2. **int** main()
3. {
4. **float** a[4]={6.1,2.3,7.8,9.0};
5. **void** \*ptr;
6. ptr=a;
7. **for**(**int** i=0;i<4;i++)
8. {
9. printf("%f,",\*((**float**\*)ptr+i));
10. }}

The above code runs successfully as we applied the proper casting to the void pointer, i.e., (float\*)ptr and then we apply the arithmetic operation, i.e., \*((float\*)ptr+i).

**Output**

void pointer in C

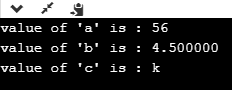
Why we use void pointers?

We use void pointers because of its reusability. Void pointers can store the object of any type, and we can retrieve the object of any type by using the indirection operator with proper typecasting.

**Let's understand through an example.**

1. #include<stdio.h>
2. **int** main()
3. {
4. **int** a=56; // initialization of a integer variable 'a'.
5. **float** b=4.5; // initialization of a float variable 'b'.
6. **char** c='k'; // initialization of a char variable 'c'.
7. **void** \*ptr; // declaration of void pointer.
8. // assigning the address of variable 'a'.
9. ptr=&a;
10. printf("value of 'a' is : %d",\*((**int**\*)ptr));
11. // assigning the address of variable 'b'.
12. ptr=&b;
13. printf("\nvalue of 'b' is : %f",\*((**float**\*)ptr));
14. // assigning the address of variable 'c'.
15. ptr=&c;
16. printf("\nvalue of 'c' is : %c",\*((**char**\*)ptr));
17. **return** 0;
18. }

**Output**



C dereference pointer

As we already know that **"what is a pointer"**, a pointer is a variable that stores the address of another variable. The dereference operator is also known as an indirection operator, which is represented by (\*). When indirection operator (\*) is used with the pointer variable, then it is known as **dereferencing a pointer.** When we dereference a pointer, then the value of the variable pointed by this pointer will be returned.

Why we use dereferencing pointer?

**Dereference a pointer is used because of the following reasons:**

* It can be used to access or manipulate the data stored at the memory location, which is pointed by the pointer.
* Any operation applied to the dereferenced pointer will directly affect the value of the variable that it points to.

**Let's observe the following steps to dereference a pointer.**

* First, we declare the integer variable to which the pointer points.

1. **int** x =9;

* Now, we declare the integer pointer variable.

1. **int** \*ptr;

* After the declaration of an integer pointer variable, we store the address of 'x' variable to the pointer variable 'ptr'.

1. ptr=&x;

* We can change the value of 'x' variable by dereferencing a pointer 'ptr' as given below:

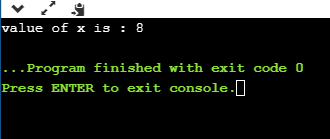
1. \*ptr =8;

The above line changes the value of 'x' variable from 9 to 8 because 'ptr' points to the 'x' location and dereferencing of 'ptr', i.e., \*ptr=8 will update the value of x.

**Let's combine all the above steps:**

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** x=9;
5. **int** \*ptr;
6. ptr=&x;
7. \*ptr=8;
8. printf("value of x is : %d", x);
9. **return** 0;}

**Output**



**Let's consider another example.**

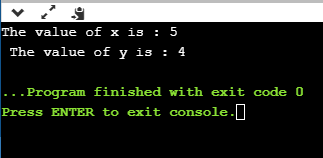
1. #include <stdio.h>
2. **int** main()
3. {
4. **int** x=4;
5. **int** y;
6. **int** \*ptr;
7. ptr=&x;
8. y=\*ptr;
9. \*ptr=5;
10. printf("The value of x is : %d",x);
11. printf("\n The value of y is : %d",y);
12. **return** 0;
13. }

**In the above code:**

* We declare two variables 'x' and 'y' where 'x' is holding a '4' value.
* We declare a pointer variable 'ptr'.
* After the declaration of a pointer variable, we assign the address of the 'x' variable to the pointer 'ptr'.
* As we know that the 'ptr' contains the address of 'x' variable, so '\*ptr' is the same as 'x'.
* We assign the value of 'x' to 'y' with the help of 'ptr' variable, i.e., y=\***ptr** instead of using the 'x' variable.

Note: According to us, if we change the value of 'x', then the value of 'y' will also get changed as the pointer 'ptr' holds the address of the 'x' variable. But this does not happen, as 'y' is storing the local copy of value '5'.

**Output**



**Let's consider another scenario.**

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a=90;
5. **int** \*ptr1,\*ptr2;
6. ptr1=&a;
7. ptr2=&a;
8. \*ptr1=7;
9. \*ptr2=6;
10. printf("The value of a is : %d",a);
11. **return** 0;
12. }

**In the above code:**

* First, we declare an 'a' variable.
* Then we declare two pointers, i.e., ptr1 and ptr2.
* Both the pointers contain the address of 'a' variable.
* We assign the '7' value to the \*ptr1 and '6' to the \*ptr2. The final value of 'a' would be '6'.

Note: If we have more than one pointer pointing to the same location, then the change made by one pointer will be the same as another pointer.

**Output**

C dereference pointer

Dynamic memory allocation in C

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

|  |  |
| --- | --- |
| **static memory allocation** | **dynamic memory allocation** |
| memory is allocated at compile time. | memory is allocated at run time. |
| memory can't be increased while executing program. | memory can be increased while executing program. |
| used in array. | used in linked list. |

Now let's have a quick look at the methods used for dynamic memory allocation.

|  |  |
| --- | --- |
| **malloc()** | allocates single block of requested memory. |
| **calloc()** | allocates multiple block of requested memory. |
| **realloc()** | reallocates the memory occupied by malloc() or calloc() functions. |
| **free()** | frees the dynamically allocated memory. |

malloc() function in C

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

1. ptr=(cast-type\*)malloc(byte-size)

Let's see the example of malloc() function.

1. #include<stdio.h>
2. #include<stdlib.h>
3. **int** main(){
4. **int** n,i,\*ptr,sum=0;
5. printf("Enter number of elements: ");
6. scanf("%d",&n);
7. ptr=(**int**\*)malloc(n\***sizeof**(**int**));  //memory allocated using malloc
8. **if**(ptr==NULL)
9. {
10. printf("Sorry! unable to allocate memory");
11. exit(0);
12. }
13. printf("Enter elements of array: ");
14. **for**(i=0;i<n;++i)
15. {
16. scanf("%d",ptr+i);
17. sum+=\*(ptr+i);
18. }
19. printf("Sum=%d",sum);
20. free(ptr);
21. **return** 0;
22. }

**Output**

Enter elements of array: 3

Enter elements of array: 10

10

10

Sum=30

calloc() function in C

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

1. ptr=(cast-type\*)calloc(number, byte-size)

Let's see the example of calloc() function.

1. #include<stdio.h>
2. #include<stdlib.h>
3. **int** main(){
4. **int** n,i,\*ptr,sum=0;
5. printf("Enter number of elements: ");
6. scanf("%d",&n);
7. ptr=(**int**\*)calloc(n,**sizeof**(**int**));  //memory allocated using calloc
8. **if**(ptr==NULL)
9. {
10. printf("Sorry! unable to allocate memory");
11. exit(0);
12. }
13. printf("Enter elements of array: ");
14. **for**(i=0;i<n;++i)
15. {
16. scanf("%d",ptr+i);
17. sum+=\*(ptr+i);
18. }
19. printf("Sum=%d",sum);
20. free(ptr);
21. **return** 0;
22. }

**Output**

Enter elements of array: 3

Enter elements of array: 10

10

10

Sum=30

realloc() function in C

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

1. ptr=realloc(ptr, **new**-size)

free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

1. free(ptr)