Unit 8.1

Pointer

Address in C

- Before you get into the concept of pointers, let's first get familiar with address in C.
- If you have a variable var in your program, &var will give you its address in the memory, where & is commonly called the reference operator.
- You must have seen this notation while using scanf() function. It was used in the function to store the user inputted value in the address of var.

```
scanf("%d", &var);
```

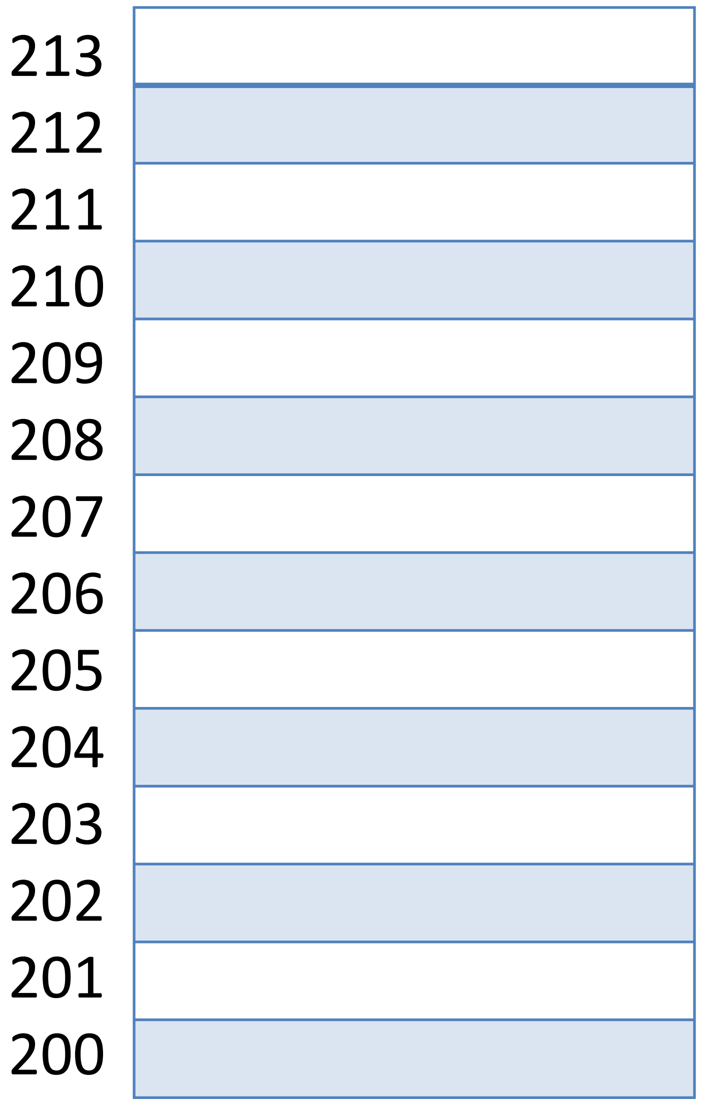
Example

```
#include <stdio.h>
int main()
{
int var = 5;
printf("Value: %d\n", var);
printf("Address: %u", &var); //Notice, the ampersand(&) before var.
return 0;
}
```

Output

```
Value: 5
Address: 2686778
```

- **Note:** You may obtain different value of address while using this code.
- In above source code, value 5 is stored in the memory location 2686778. var is just the name given to that location.



Memory Hierarchy

Pointer variables

- In C, you can create a special variable that stores the address (rather than the value). This variable is called pointer variable or simply a pointer.
- **How to create a pointer variable?**
- data_type* pointer_variable_name;
  - int* p;
- Above statement defines, p as pointer variable of type int.

Advantages of Pointer

1. pointers are more efficient in handling arrays & data tables.
2. Pointers can be used to return multiple values from a function via function arguments.
3. pointers permit references to functions & there by facilitating passing of functions as arguments to other functions .
4. The use of pointers arrays to character strings results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures. Such as structures, linked lists, queues, stacks & trees.

Disadvantages of Pointer

1. Pointers are a little complex to understand.
2. Pointers can lead to various errors such as segmentation faults or can access a memory location which is not required at all. (A segmentation fault occurs when your program attempts to access an area of memory that it is not allowed to access)
3. If an incorrect value is provided to a pointer, it may cause memory corruption.
4. Pointers are also responsible for memory leakage.
5. Pointers are comparatively slower than that of the variables.

Reference operator (&) and Dereference operator (*)

- As discussed, & is called reference operator. It gives you the address of a variable.
- Likewise, there is another operator that gets you the value from the address, it is called a dereference operator *.
- Below example clearly demonstrates the use of pointers, reference operator and dereference operator.
- **Note:** The * sign when declaring a pointer is not a dereference operator. It is just a similar notation that creates a pointer.

Example

```
#include<stdio.h>
void main()
{
	int b=100,*p;
	p=&b;
	printf("\n%d",b);
	printf("\n%u",&b);

	printf("\n%u",p);
	printf("\n%d",*p);
}
```

Example: How Pointer Works?

```
#include <stdio.h>
int main()
{
int* pc, c;
c = 22;
printf("Address of c: %u\n", &c);
printf("Value of c: %d\n\n", c);
pc = &c;
printf("Address of pointer pc: %u\n", pc);
printf("Content of pointer pc: %d\n\n", *pc);
c = 11;
printf("Address of pointer pc: %u\n", pc);
printf("Content of pointer pc: %d\n\n", *pc);
*pc = 2;
printf("Address of c: %u\n", &c);
printf("Value of c: %d\n\n", c);
return 0;
}
```

- **Output**

```
Address of c: 2686784
Value of c: 22
Address of pointer pc: 2686784
Content of pointer pc: 22
Address of pointer pc: 2686784
Content of pointer pc: 11
Address of c: 2686784
Value of c: 2
```

Common mistakes when working with pointers

Suppose, you want pointer pc to point to the address of c. Then,

```
int c, *pc;

// Wrong! pc is address whereas, c is not an address.
pc = c;

// Wrong! *pc is the value pointed by address whereas, &c is an
address.
*pc = &c;
```

```
// Correct! pc is an address and, &c is also an address.
pc = &c;

// Correct! *pc is the value pointed by address and,
// c is also a value (not address).
*pc = c;
```

Add two numbers using pointer

```
#include<stdio.h>
void main()
{
      int a,b,sum;
      int *pa,*pb;
      printf("enter two numbers");
      scanf("%d%d",&a,&b);
      pa=&a;
      pb=&b;
      sum=*pa+*pb;
      printf("the sum is %d",sum);
}
```

Swaping pointer variable

```
#include<stdio.h>
void main()
{
	int a,b,sum;
	int *pa,*pb,*temp;
	printf("enter two numbers");
	scanf("%d%d",&a,&b);
	pa=&a;
	pb=&b;

	printf("\nbefore swapping");
	printf("\n1st number is %d",*pa);
	printf("\n2nd number is %d",*pb);
```

```
    temp=pa;
    pa=pb;
    pb=temp;
    printf("\nafter swapping");
    printf("\n1st number is %d",*pa);
    printf("\n2nd number is %d",*pb);

}
```

Passing pointer to functions

- A pointer can be passed to a function as an arguments
- Passing a pointer means passing address of a variable instead of value of the variable
- As address is passed in this case, this mechanism is also known as call by reference or call by address
- When pointer is passed to a function, while function calling, the formal argument of the function must be compatible with the passing pointer i.e. if integer pointer is being passed, the formal argument in function must be pointer of the type integer and so on.

Example

```
#include<stdio.h>
void conversion(char *);
void main()
{
	char ch;
	printf("enter a character");
	scanf("%c",&ch);
	conversion(&ch);
	printf("\n the corresponding character is %c",ch);
}
```

```
void conversion(char *c)
{
	if(*c>=97 && *c<=122){
		*c=*c-32;
	}
	else{
		*c=*c+32;
	}
}
```

Call by value

- The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- By default, C programming uses *call by value* to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

```
#include <stdio.h>
/* function prototype */
void swap(int x, int y);
void main () {

   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );
```

```
 /* calling a function to swap the values */
   swap(a, b);
 printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
}
/* function definition to swap the values */
void swap(int x, int y) {
   int temp;
   temp = x; //save the value of x
   x = y;    // put y into x
   y = temp; //put temp into y
}
```

Call by reference

- The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.
- To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

```
#include <stdio.h>
/* function prototype */
void swap(int *x, int *y);

void main () {
   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );
```

```
 /* calling a function to swap the values.
     * &a indicates pointer to a ie. address of variable a and
     * &b indicates pointer to b ie. address of variable b.   */
   swap(&a, &b);
   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
}
void swap(int *x, int *y) {
   int temp;
   temp = *x;    /* save the value at address x */
   *x = *y;      /* put y into x */
   *y = temp;    /* put temp into y */
}
```

Return Pointer from a function

```
#include<stdio.h>
#include<conio.h>
int *check();
void main()
{

	int *p;
	p=check();
	printf("%d",*p);

}
```

```
int *check()
{
      int *ptr,a;
      printf("enter a number");
      scanf("%d",&a);
      ptr=&a;
      return ptr;

}
```

Returning pointer from function

```
#include<stdio.h>
#include<conio.h>
int *makeDouble(int[]);
void main()
{
	int x[]={10,20,30,40,50};
	int *p,i;
	printf("the elements of original array are:\n");
	for(i=0;i<5;i++)
		printf("\n%d",x[i]);
	p=makeDouble(x);
	printf("the elements of array after making double are:\n");
	for(i=0;i<5;i++)
		printf("\n%d",*(p+i));
}
```

```
int *makeDouble(int a[5])
{
    int *ptr,i;
    ptr=a;
    for(i=0;i<5;i++)
    {
        a[i]=2*a[i];
    }
    return ptr;

}
```

C Pointers and Arrays

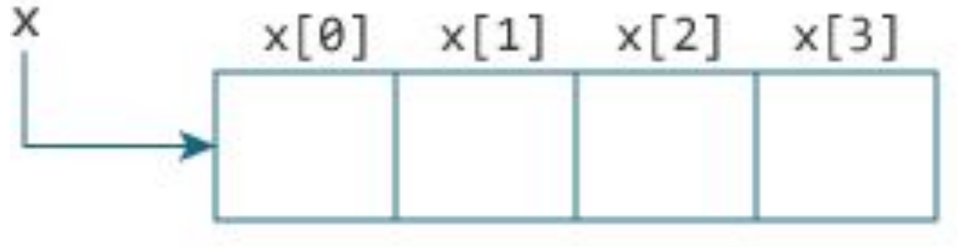
```
#include <stdio.h>
int main()
{
int x[4];
int i;
for(i = 0; i < 4; ++i)
{
printf("&x[%d] = %u\n", i, &x[i]);
}
printf("Address of array x: %u", x);
return 0;
}
```

Output:

```
&x[0] = 1450734448
&x[1] = 1450734452
&x[2] = 1450734456
&x[3] = 1450734460
Address of array x: 1450734448
```

Relation between Arrays and Pointers

- Consider an array:
- int x[4];



- From the above example, it's clear that x and &x[0] both contains the same address. Hence, &x[0] is equivalent to x.
- And, x[0] is equivalent to *x.
- Similarly,
- &x[1] is equivalent to x+1 and x[1] is equivalent to *(x+1).
- &x[2] is equivalent to x+2 and x[2] is equivalent to *(x+2).
- Basically, &x[i] is equivalent to x+i and x[i] is equivalent to *(x+i).

Example

```
#include <stdio.h>
int main()
{
int i, x[6], sum = 0;
printf("Enter 6 numbers: ");
for(i = 0; i < 6; ++i)
{
scanf("%d", x+i);
sum += *(x+i);
}
printf("Sum = %d", sum);
return 0;
}
```

Output:

```
Enter 6 numbers: 2
3
4
4
12
4
Sum = 29
```

- In most contexts, array names "decays" to pointers. In simple words, array names are converted to pointers. That's the reason why you can use pointer with the same name as array to manipulate elements of the array. However, you should remember that **pointers and arrays are not same**.

Example 2: Arrays and Pointers

```
#include <stdio.h>
int main()
{
int x[5] = {1, 2, 3, 4, 5};
int* ptr;
ptr = &x[2];
printf("*ptr = %d \n", *ptr);
printf("*ptr+1 = %d \n", *ptr+1);
printf("*ptr-1 = %d", *ptr-1);
return 0;
}
```

Output:

```
*ptr = 3
*ptr+1 = 4
*ptr-1 = 2
```

- In this example, &x[2] (address of the third element of array x) is assigned to the pointer ptr. Hence, 3 was displayed when we printed *ptr.
- And, printing *ptr+1 gives us the fourth element. Similarly, printing *ptr-1 gives us the second element.

Sorting in ascending order

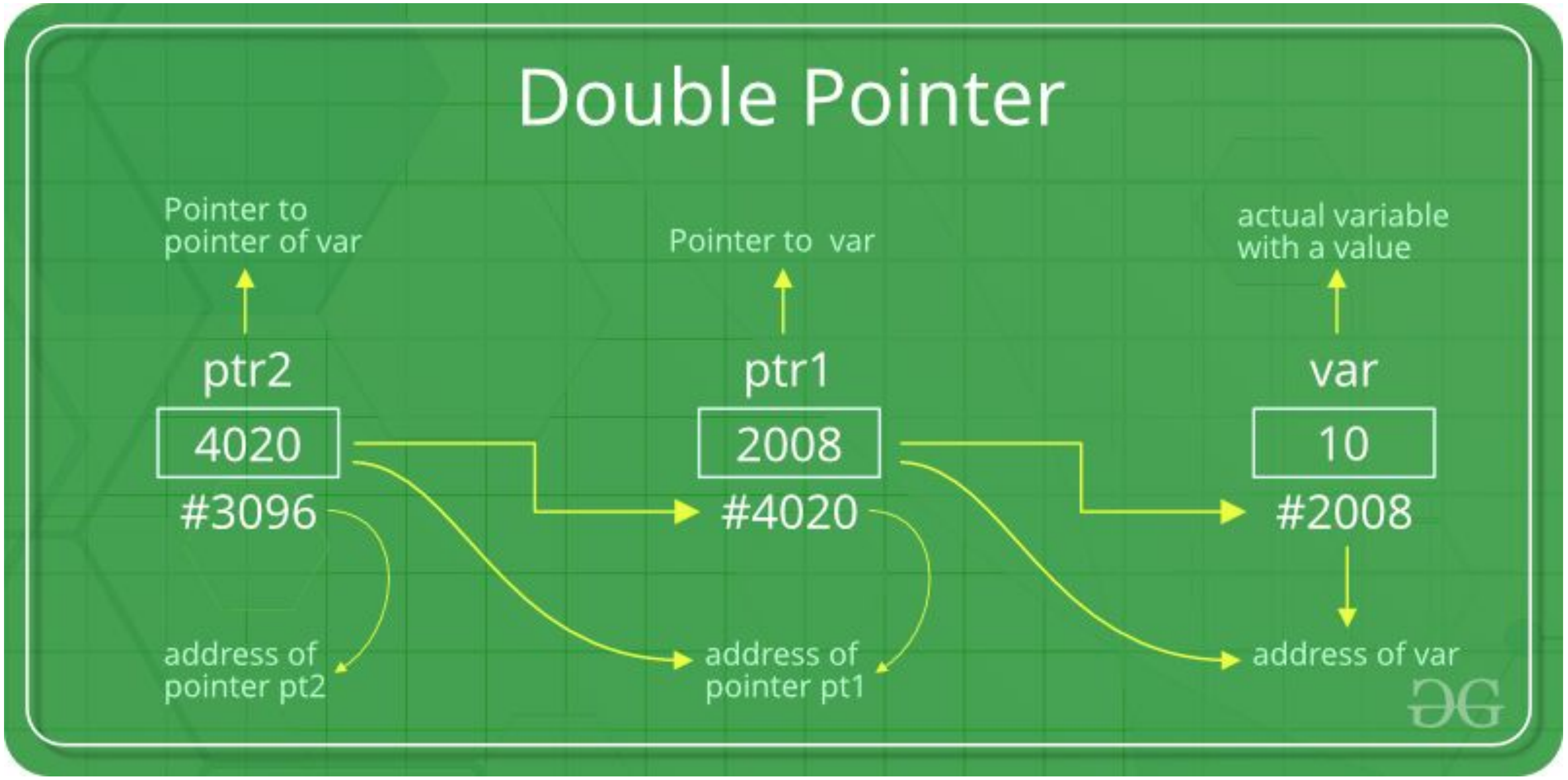
```
#include<stdio.h>
#include<conio.h>
void main()
{
    int x[5];
    int *p,i,j,temp=0;
    printf("enter 5 numbers");
    for(i=0;i<5;i++)
        scanf("%d",&x[i]);
    printf("\narray before sorting:\n");
    for(i=0;i<5;i++)
        printf("\n%d",x[i]);
```

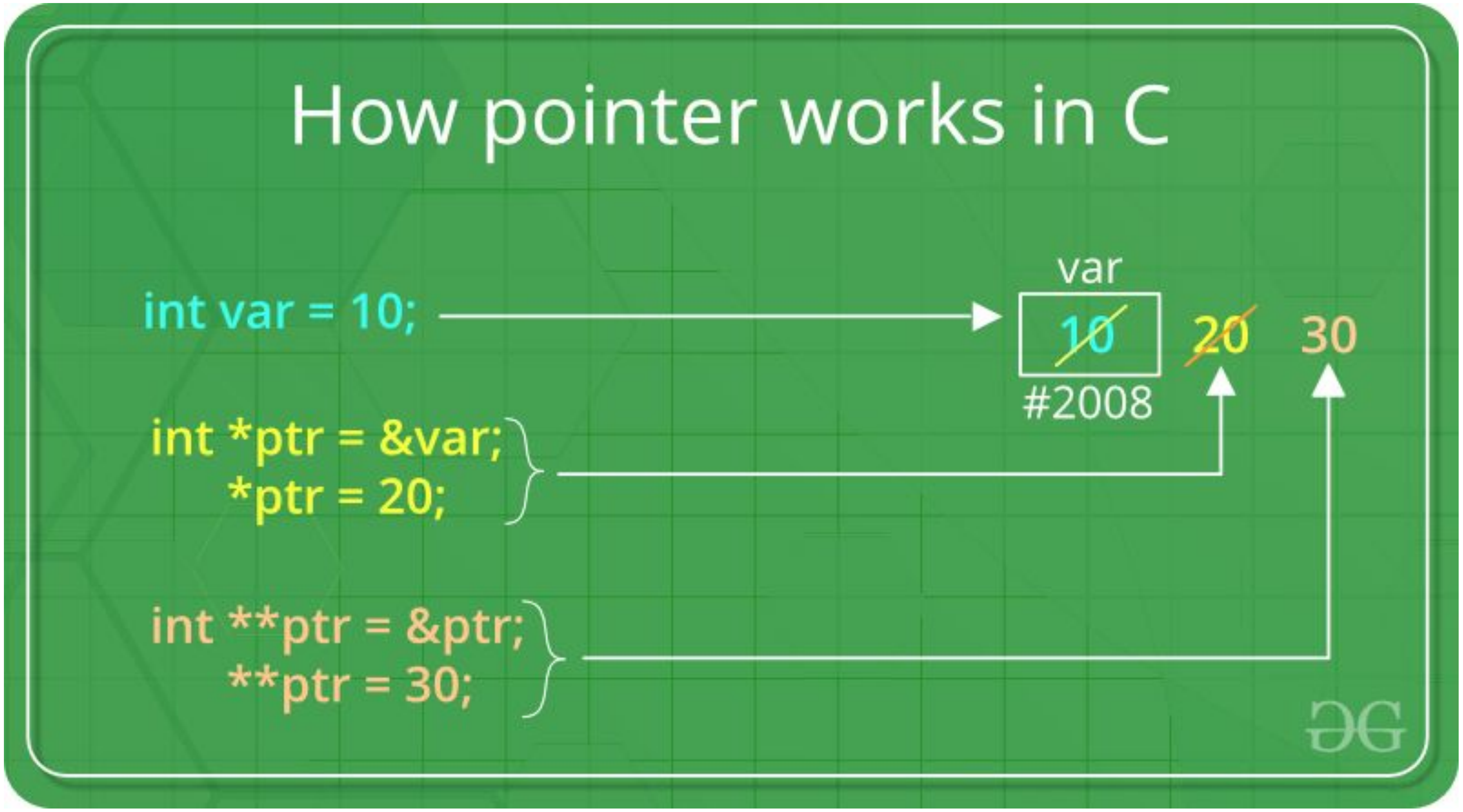
```
p=&x[0];
  for(i=0;i<5-1;i++)
  {
        for(j=i+1;j<5;j++)
        {
              if(*(p+i)>*(p+j))
              {
                    temp=*(p+i);
                    *(p+i)=*(p+j);
                    *(p+j)=temp;
              }
        }
  }
```

```
printf("\nthe array after sorting:\n");
      for(i=0;i<5;i++)
      {
            printf("\n%d",x[i]);
      }
}
```

Pointer to Pointer

- We already know that a pointer points to a location in memory and thus used to store address of variables. So, when we define a pointer to pointer. The first pointer is used to store the address of second pointer. That is why they are also known as double pointers.
- **How to declare a pointer to pointer in C?**
  Declaring Pointer to Pointer is similar to declaring pointer in C. The difference is we have to place an additional '*' before the name of pointer.
  Syntax:
- int **ptr; // declaring double pointers


Double Pointer
Pointer to pointer of var
Pointer to var
actual variable with a value
ptr2
4020
#3096
ptr1
2008
#4020
var
10
#2008
address of pointer pt2
address of pointer pt1
address of var


How pointer works in C
int var = 10;
var
~~10~~
~~20~~
30
#2008
int *ptr = &var;
*ptr = 20;
int **ptr = &ptr;
**ptr = 30;

Example

```
#include<stdio.h>
int main()
{
   int var = 789;
   int *ptr2;
   int **ptr1;
   ptr2 = &var;
   ptr1 = &ptr2;

   printf("Value of var = %d\n", var );
   printf("Value of var using single pointer = %d\n", *ptr2 );
   printf("Value of var using double pointer = %d\n", **ptr1);
}
```

Pointer Arithmetic in C

- We can perform arithmetic operations on the pointers like addition, subtraction, etc.
- However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer.
- In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:
- Increment
- Decrement
- Addition
- Subtraction
- Comparison

Incrementing Pointer in C

- If we increment a pointer by 1, the pointer will start pointing to the immediate next location.
- This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.
- We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

- **Decrementing Pointer in C**
- Like increment, we can decrement a pointer variable.
- If we decrement a pointer, it will start pointing to the previous location.

- **C Pointer Addition**
- We can add a value to the pointer variable. The formula of adding value to pointer is given below:
  - p=p+3;   //adding 3 to pointer variable

- **C Pointer Subtraction**
- Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address.
- p=p-3; //subtracting 3 from pointer variable

Illegal arithmetic with pointers

- Address + Address = illegal
- Address * Address = illegal
- Address % Address = illegal
- Address / Address = illegal
- Address & Address = illegal
- Address ^ Address = illegal
- Address | Address = illegal
- ~Address = illegal

Types of Pointer

1. Null Pointer
2. Void Pointer
3. Wild Pointer
4. Dangling Pointer