Pointers in C

By: Atiya Jokhio

Difference between pointers and arrays

- Assigning any address to an array variable is not allowed while address can be assigned to a pointer variable using address operator.
- A pointer is a place in memory that keeps address of another place inside while an array is a single, pre allocated chunk of contiguous elements (all of the same type), fixed in size and location.
- Pointer is dynamic in nature. The memory allocation can be resized or freed later while the arrays are static in nature. Once memory is allocated, it cannot be resized or freed dynamically.

Passing Array Elements to a Function by reference

```c
#include <stdio.h>
int main()
{
    int array[]= {10,20,30,40,50};
    display(&array[0],5);
}

display (int *j, int n)
{
    int i;
    for(i=0; i<n; i++)
    {

        printf("\nelement= %d", *j);
        j++; // increment pointer to point to next element
    }}
```

Passing Array Elements to a Function by reference

```c
#include <stdio.h>
#include <string.h>

void Sentence(char *arr)
{
    int i;
    int n = strlen(arr);

    printf("n = %d\n", n);

    for (i=0; i<n; i++)
      printf("%c", arr[i]);
}

// Driver program
int main()
{
    char arr[] = "Fast is National University";
    Sentence(arr);
    return 0;
}
```

Pointer to Pointer (Double Pointer)

- We already know that a pointer points to a location in memory and thus used to store the address of variables. So, when we define a pointer to pointer.
- The first pointer is used to store the address of the variable. And the second pointer is used to store the address of the first pointer. That is why they are also known as double pointers.

Pointer to Pointer (Double Pointer)

Pointer to pointer declaration:

Syntax:

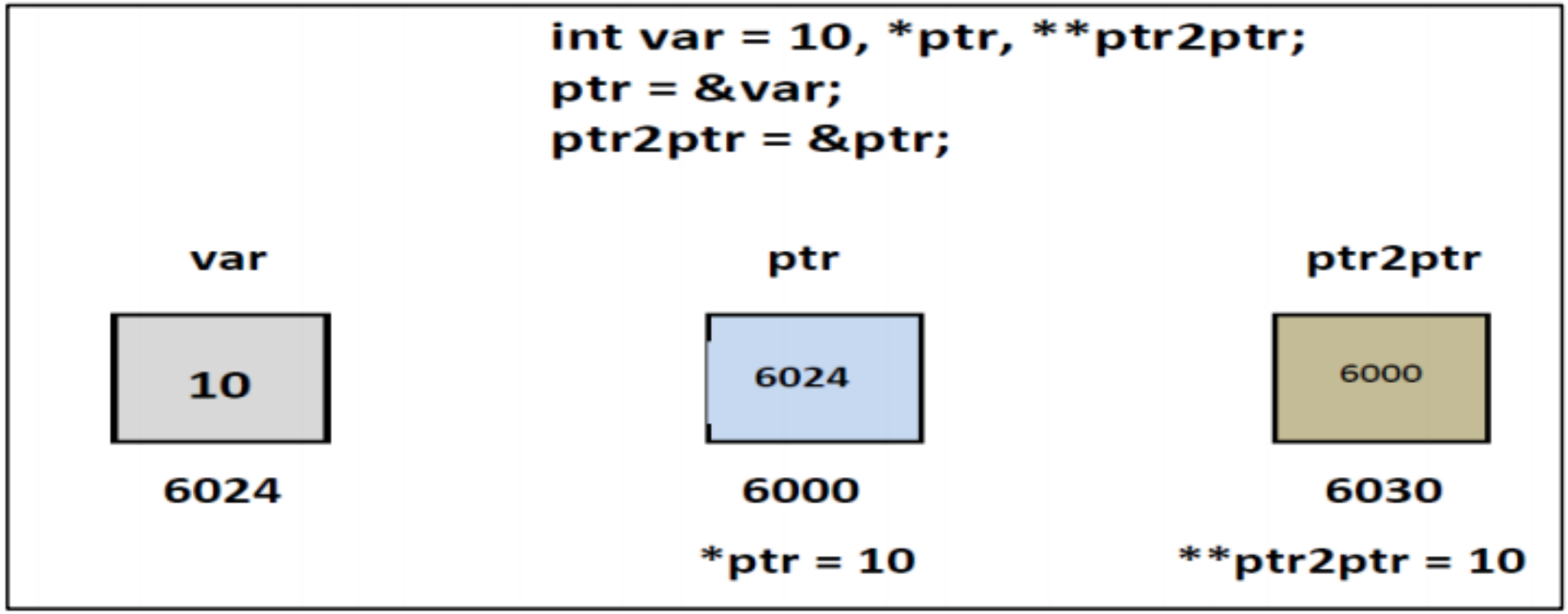
type ** variable ;

Example:

int **ptr2ptr;

Pointer to Pointer (Double Pointer)

INTERPRETATION:
The value of the pointer variable ptr2ptr is a memory address of another pointer.



Pointer to Pointer (Double Pointer)

Example:

```c
#include <stdio.h>
int main() {
    int  a = 10;
    int  *p1;        //this can store the address of variable a
    int  **p2;
    /*
        this can store the address of pointer variable p1 only.
        It cannot store the address of variable 'a'
    */
    p1 = &a;
    p2 = &p1;
    printf("Address of a = %u\n", &a);
    printf("Address of p1 = %u\n", &p1);
    printf("Address of p2 = %u\n\n", &p2);
    // below print statement will give the address of 'a'
    printf("Value at the address stored by p2 = %u\n", *p2);
    printf("Value at the address stored by p1 = %d\n", *p1);
    printf("Value of **p2 = %d\n", **p2); //read this *(*p2)
    return 0;
}
```

2D- Array using Pointers

- To access a two dimensional array using pointer, let us recall basics from one dimensional array. Since it is just an array of one dimensional array.

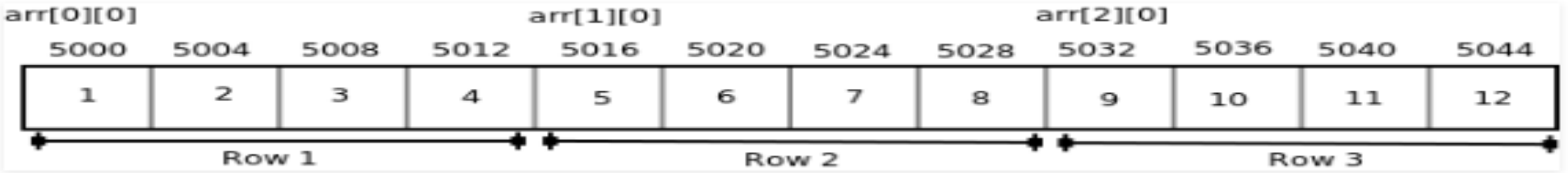
- Suppose I have a pointer array_ptr pointing at base address of one dimensional array. To access nth element of array using pointer we use *(array_ptr+n) (where array_ptr points to 0^{th} element of array, n is the nth element to access and nth element starts from 0).

2D- Array using Pointers

Let us take a two dimensional array *arr[3][4]*:
Int arr[3][4] = { {1, 2, 3 , 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };

	Col 1	Col 2	Col 3	Col 4
Row 1	1	2	3	4
Row 2	5	6	7	8
Row 3	9	10	11	12

The following figure shows how the above 2-D array will be stored in memory



2D- Array using Pointers

- Now we know two dimensional array is array of one dimensional array. Hence let us see how to access a two dimensional array through pointer.
- Let us suppose a two-dimensional array

Int matrix[3][3];

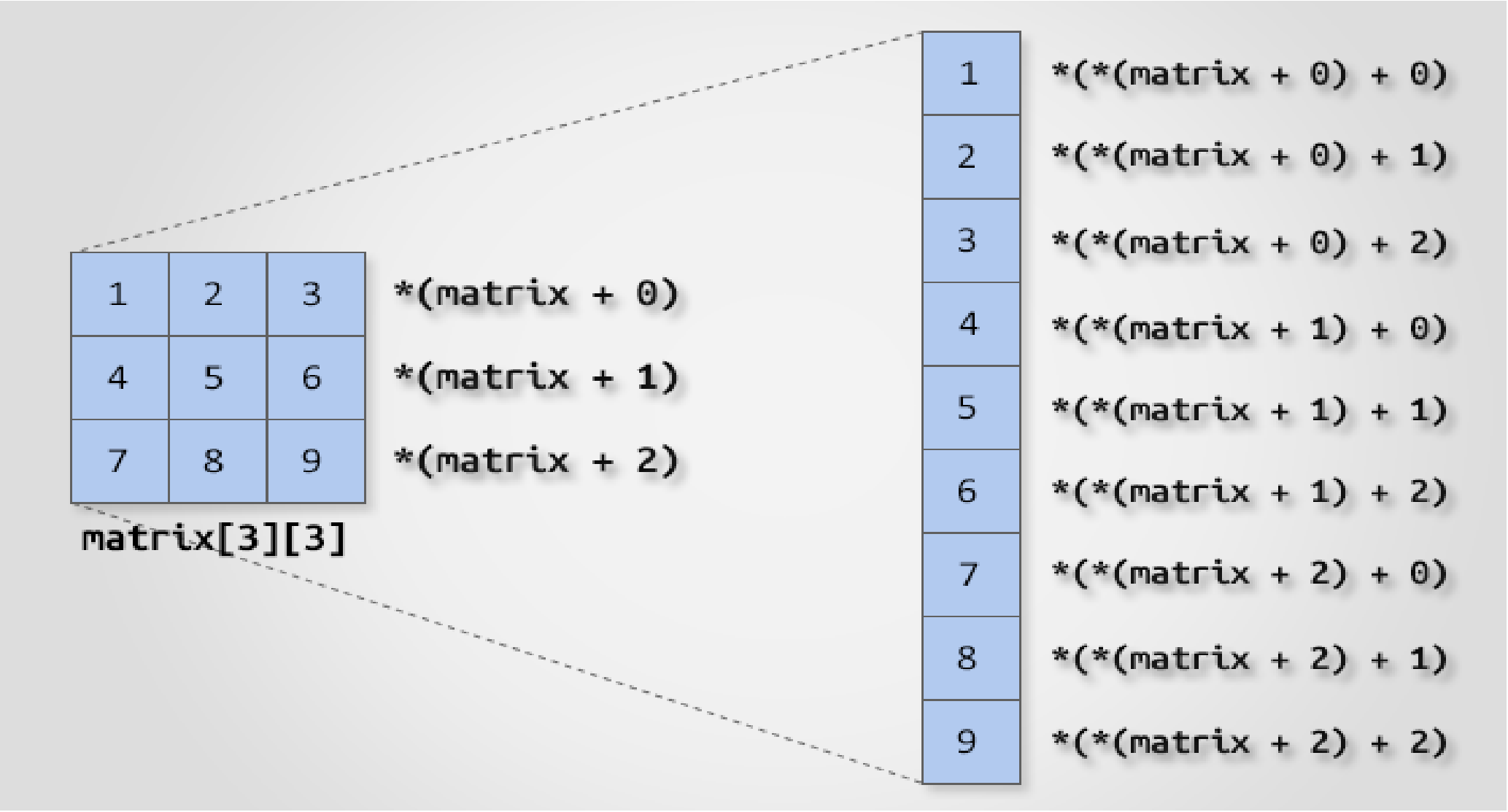
For the above array,

```
matrix                  =>    Points to base address of two-dimensional array.
                              Since array decays to pointer.

*(matrix)               =>    Points to first row of two-dimensional array.
*(matrix + 0)           =>    Points to first row of two-dimensional array.
*(matrix + 1)           =>    Points to second row of two-dimensional array.

**matrix                =>    Points to matrix[0][0]
*(*(matrix + 0))        =>    Points to matrix[0][0]
*(*(matrix + 0) + 0)    =>    Points to matrix[0][0]
*(*matrix + 1)          =>    Points to matrix[0][1]
*(*(matrix + 0) + 1)    =>    Points to matrix[0][1]
*(*(matrix + 2) + 2)    =>    Points to matrix[2][2]
```

2D- Array using Pointers



2D- Array using Pointers

Example:

```
#include <stdio.h>
int main( )
{
int s[4][2] = {
{ 1, 2 },
{ 3, 4 },
{ 5, 6 },
{ 7, 8 }
} ;
int i, j ;
for ( i = 0 ; i <= 3 ; i++ )
{
printf ( "\n" ) ;
//printf("Adress of i %u", *(s+i));
//printf ( "\n" ) ;
for ( j = 0 ; j <= 1 ; j++ )
{
//printf ( "Adress of j %d\n", ( *( s + i ) + j ) ) ;
printf ( "%d ", *( *( s + i ) + j ) ) ;
}}
}
```

Pointers to structures

- We have already learned that a pointer is a variable which points to the address of another variable of any data type like int, char, float etc.
- Similarly, we can have a pointer to structures, where a pointer variable can point to the address of a structure variable.
- Here is how we can declare a pointer to a structure variable.

```
struct name {
member1;
member2; . . };

 int main()
{
struct name *ptr, Michal;
}
```

Pointers to structures

- Here is how we can declare a pointer to a structure variable.

```c
#include <stdio.h>

struct Book
{
    char name[10];
    int price;
}

int main()
{
    struct Book a;          //Single structure variable
    struct Book* ptr;       //Pointer of Structure type
    ptr = &a;

    struct Book b[10];      //Array of structure variables
    struct Book* p;         //Pointer of Structure type
    p = &b;

    return 0;
}
```

Pointers to structures

- **Accessing Structure Members with Pointer**

  To access members of structure using the structure variable, we used the dot . operator.
  But when we have a pointer of structure type, we use arrow -> to access structure members.
  Example:

```c
#include <stdio.h>
struct person
{
    int age;
    float weight;
};
int main(){
     struct person *personPtr, person1;
     personPtr = &person1;

     printf("Enter age: ");
     scanf("%d", &personPtr->age);

     printf("Enter weight: ");
     scanf("%f", &personPtr->weight);

     printf("Displaying:\n");
     printf("Age: %d\n", personPtr->age);
     printf("weight: %f", personPtr->weight);
     return 0;
}
```

Pointers to structures

- **Passing pointer structure to function**

```
# include <stdio.h>
struct book
{
char name[50] ;
char author[25] ;
int volno ;
} ;

main()
{
struct book b1 = { "Introduction to Computers", "Peter Norton", 100 } ;
display ( &b1 ) ;
}

display ( struct book *b )
{
 printf ( "\n%s \n%s \n%d", b->name, b->author, b->volno ) ;
}
```

Void Pointers

- In the c programming language, pointer to void is the concept of defining a pointer variable that is independent of data type.
- In C programming language, a void pointer is a pointer variable used to store the address of a variable of any datatype.
- That means single void pointer can be used to store the address of integer variable, float variable, character variable, double variable or any structure variable.
- We use the keyword **void** to create void pointer.

```
void  *ptr;
```

Void Pointers

```c
#include<stdio.h>

int main()
{
    int a ;
    float b ;
    char c ;

    void *ptr ;

    ptr = &a ;
    printf("Address of integer variable 'a' = %u\n", ptr) ;

    ptr = &b ;
    printf("Address of float variable 'b' = %u\n", ptr) ;

    ptr = &c ;
    printf("Address of character variable 'c' = %u\n", ptr) ;
    return 0;
}
```

Void Pointers (Type Cast)

```c
#include <stdio.h>
void function(void *data,int ptrsize)
{
    if(ptrsize == sizeof(char))
    {
        char *ptrchar;
        //Typecast data to a char pointer
        ptrchar = (char*)data;
        //Increase the char stored at *ptrchar by 1
        (*ptrchar)++;
        printf("*data points to a char\n\n");
    }
    else if(ptrsize == sizeof(int))
    {
        int *ptrint;
        //Typecast data to a int pointer
        ptrint = (int*)data;
        //Increase the int stored at *ptrchar by 1
        (*ptrint)++;
        printf("*data points to an int\n\n");
    }
}
```

```c
main_function()
{
    // Declare a character
    char c='x';

    // Declare an integer
    int i=10;

    //Call increase function using a char and int address respectively
    function(&c,sizeof(c));
    printf("The new value of c is:  %c \n\n" , c);
    function(&i,sizeof(i));
    printf("The new value of i is:  %d ", i);
}
//Driver program
int main()
{
    main_function();
}
```

String using pointers

```c
#include <stdio.h>
int main()
{
    char str[100];
    char *p;
    printf("Enter any string: ");
    gets(str);

    /* Assigning the base address str[0] to pointer
     * p. p = str is same as p = str[0]
     */
    p=str;
    printf("The input string is: ");
    //'\0' signifies end of the string
    while(*p!='\0')
        printf("%c",*p++);
    return 0;
}
```

Pointer to a whole array

we have a pointer *ptr* that points to the 0^{th} element of the array. Similarly, we can also declare a pointer that can point to whole array instead of only one element of the array. This pointer is useful when talking about multidimensional arrays.

Syntax:

```
data_type (*var_name)[size_of_array];
```

Example:

```
int (*ptr)[10];
```

Here *ptr* is pointer that can point to an array of 10 integers. Since subscript have higher precedence than indirection, it is necessary to enclose the indirection operator and pointer name inside parentheses. Here the type of ptr is ‘pointer to an array of 10 integers’.

Pointer to a whole array

For Example:

```c
#include<stdio.h>
int main()
{
    // Pointer to an integer
    int *p;
    // Pointer to an array of 5 integers
    int (*ptr)[5];
    int arr[5];
    // Points to 0th element of the arr.
    p = arr;
    // Points to the whole array arr.
    ptr = &arr;

    printf("p = %u, ptr = %u\n", p, ptr);

    p++;
    ptr++;
    printf("p = %u, ptr = %u\n", p, ptr);
    return 0;
}
```

2D-Array with pointers

For Example:

```c
#include<stdio.h>
int main()
{
    int arr[3][4] = {
                        {1,2,3,4},
                        {5,6,7,8},
                        {9,10,11,12}
                    };
    int i, j;
    int (*p)[4];
    p = arr;
    for(i = 0; i < 3; i++)
    {
        printf("Address of %d th array %u \n",i , p + i);
        for(j = 0; j < 4; j++)
        {
            printf("arr[%d][%d]=%d\n", i, j, *( *(p + i) + j) );
        }
        printf("\n\n");
    }
}
```

Function pointers

- Function Pointers are like normal pointers but they have capability to point to a function.
- A function pointer is a pointer that refers to the address of a function.
- Some facts about function pointers are:
  - Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.
  - Unlike normal pointers, we do not allocate de-allocate memory using function pointers
- The syntax of declaring a function pointer is similar to the syntax of declaring a function.

  The only difference is that instead of using the function name, you use a pointer name inside the parentheses ().
- Syntax:

```
<return_type> (*<pointer_name>) (function_arguments);
```

Function pointers

- Let’s examine the function pointer syntax above in more detail:
- First, you specify the return type of the function pointer. It can be any valid type such as int, float, char, or void.
- Second, you put the name of function pointer inside parentheses.
- Third, you specify all parameters of the function with their corresponding types. Notice that the function pointer only can refer to a function with the same signature. It means all functions, which the function pointer refers to, must have the same return type and parameters.

Function pointers

The following example declares a function pointer referred to a function that accepts two integer parameters and returns an integer.

Example:

```
int add (int a, int b)
{
return a+b;
}
```

```
int main ()
{
        int (*ptr) (int, int);
}
```

Function pointers

- **Using function pointers**

Before using a function pointer, you need to assign it an address of a function

```
int main()
{
int (*ptr) (int, int);
ptr = &add;
}
```

```
int add (int a, int b)
{
return a+b;
}
```

```
int main ()
{
        int (*ptr) (int, int) = &add;
        int result = (*ptr) (10,20);
        printf("%d", result);
}
```

Function pointers

- **Using function pointers**
- There are some situations in which user has to decide which function has to be called at particular point in time.
- We can use them to replace switch/if-statements, and to realize late-binding.
- Late binding refers to deciding the proper function during runtime instead of compile time.
- Like normal pointers, we can have an array of function pointers.
- Lets say, we want to design a calculator which has the capability to perform addition, subtraction, multiplication and division.

Function pointers

```c
#include <stdio.h>

float sum(float a, float b) { return (a+b); }
float sub(float a, float b) { return (a-b); }
float mult(float a, float b) { return (a*b); }
float divi(float a, float b) { return (a/b); }

int main() {
    int choice;
    float a, b, result;
    printf("Enter your choice: 0 for sum, 1 for sub, 2 for mult, 3 for div:\n");
    scanf("%d", &choice);
    printf("Enter the two numbers:\n");
    scanf("%f %f", &a, &b);

    switch(choice)
    {
        case 0: result = sum(a, b); break;
        case 1: result = sub(a, b); break;
        case 2: result = mult(a, b); break;
        case 3: result = divi(a, b); break;
    }
```

Function pointers

By using an array of function pointers.

```
float sum(float a, float b){ return (a+b);}
float sub(float a, float b){ return (a-b);}
float mul(float a, float b){ return (a*b);}
float divi(float a, float b){ return (a/b);}

int main ()
{
    float (*ptr2func[]) (float, float)= {sum, sub,mul, divi};
    int choice;
    float a, b;
    printf("Enter your choice: 0 for sum, 1 for sub, 2 for mult, 3 for div:\n");
    scanf("%d", &choice);
    printf("Enter the two numbers: \n");
    scanf("%f %f", &a, &b);
    printf("%f", (*ptr2func[choice]) (a,b));
    return 0;
}
```

Function pointers

Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.
A simple prototype for a function which takes a *function parameter* (sometimes called a formal parameter), is something like this:

```
void myfunction(void (*f)(int));
```

This states that a parameter f will be a pointer (*f) to the function myFunction, which has a void return type and which takes just a single int parameter.

Function pointers

Example:

We can pass a function pointer as a function's calling argument.

```c
#include <stdio.h>
void print()
{
    printf("Hello World!");
}
void helloworld(void (*f)())
{
    f();
}
int main(void)
{
    helloworld(print);
    return (0);
}
```

```c
#include <stdio.h>
void printNumber(int nbr)
{
    printf("%d\n", nbr);
}
void myFunction(void (*f)(int))
{
    int i;
    for( i = 0; i < 5; i++)
    {
        (*f)(i);
    }
}
int main(void)
{
    myFunction(printNumber);
    return (0);
}
```