

# How to declare a pointer to a function?

Well, we assume that you know what does it mean by pointer in C. So how do we create a pointer to an integer in C?

Huh..it is pretty simple..

```
int * ptrInteger; /*We have put a * operator between int
                  and ptrInteger to create a pointer.*/
```

Here ptrInteger is a pointer to integer. If you understand this, then logically we should not have any problem in declaring a pointer to a function 😊

So let us first see ..how do we declare a function? For example,

```
int foo(int);
```

Here foo is a function that returns int and takes one argument of int type. So as a logical guy will think, by putting a \* operator between int and foo(int) should create a pointer to a function i.e.

```
int * foo(int);
```

But Oops..C operator precedence also plays role here ..so in this case, operator () will take priority over operator \*. And the above declaration will mean – a function foo with one argument of int type and return value of int \* i.e. integer pointer. So it did something that we didn't want to do. 😞


So as a next logical step, we have to bind operator \* with foo somehow. And for this, we would change the default precedence of C operators using () operator.

```
int (*foo)(int);
```

That's it. Here \* operator is with foo which is a function name. And it did the same that we wanted to do.

Why do we need an extra bracket around function pointers like fun\_ptr in above example?

If we remove bracket, then the expression "void (\*fun\_ptr)(int)" becomes "void \*fun\_ptr(int)" which is declaration of a function that returns void pointer.



```

#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    /* The above line is equivalent of following two
    void (*fun_ptr)(int);
    fun_ptr = &fun;
    */

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);

    return 0;
}


```

Output:

```
Value of a is 10
```

**Following are some interesting facts about function pointers.**

- 1) Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.
- 2) Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
- 3) A function's name can also be used to get functions' address. For example, in the below program, we have removed address operator '&' in assignment. We have also changed function call by removing \*, the program still works.



```

#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

```

```

int main()
{
    void (*fun_ptr)(int) = fun; // & removed

    fun_ptr(10); // * removed

    return 0;
}

```

Output:

```
Value of a is 10
```

4) Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.

5) Function pointer can be used in place of switch case. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks.

```

#include <stdio.h>
void add(int a, int b)
{
    printf("Addition is %d\n", a+b);
}
void subtract(int a, int b)
{
    printf("Subtraction is %d\n", a-b);
}
void multiply(int a, int b)
{
    printf("Multiplication is %d\n", a*b);
}

```

```

int main()
{
    // fun_ptr_arr is an array of function pointers
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int ch, a = 15, b = 10;

    printf("Enter Choice: 0 for add, 1 for subtract and 2 "
           "for multiply\n");
    scanf("%d", &ch);

    if (ch > 2) return 0;

    (*fun_ptr_arr[ch])(a, b);

    return 0;
}

```

```
Enter Choice: 0 for add, 1 for subtract and 2 for multiply
2
Multiplication is 150
```

6) Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.

For example, consider the following C program where wrapper() receives a void fun() as parameter and calls the passed function.

```
// A simple C program to show function pointers as parameter
#include <stdio.h>

// Two simple functions
void fun1() { printf("Fun1\n"); }
void fun2() { printf("Fun2\n"); }

// A function that receives a simple function
// as parameter and calls the function
void wrapper(void (*fun)())
{
    fun();
}

int main()
{
    wrapper(fun1);
    wrapper(fun2);
    return 0;
}
```

This point in particular is very useful in C. In C, we can use function pointers to avoid code redundancy. For example a simple `qsort()` function can be used to sort arrays in ascending order or descending or by any other order in case of array of structures. Not only this, with function pointers and void pointers, it is possible to use `qsort` for any data type.

```
// An example for qsort and comparator
#include <stdio.h>
#include <stdlib.h>

// A sample comparator function that is used
// for sorting an integer array in ascending order.
// To sort any array for any other data type and/or
// criteria, all we need to do is write more compare
// functions. And we can use the same qsort()
int compare (const void * a, const void * b)
{
    return ( *(int*)a - *(int*)b );
}
```

```

int main ()
{
    int arr[] = {10, 5, 15, 12, 90, 80};
    int n = sizeof(arr)/sizeof(arr[0]), i;

    qsort (arr, n, sizeof(int), compare);

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

```

Output:

```
5 10 12 15 80 90
```

Similar to qsort(), we can write our own functions that can be used for any data type and can do different tasks without code redundancy. Below is an example search function that can be used for any data type. In fact we can use this search function to find close elements (below a threshold) by writing a customized compare function.

```

#include <stdio.h>
#include <stdbool.h>

// A compare function that is used for searching an integer
// array
bool compare (const void * a, const void * b)
{
    return ( *(int*)a == *(int*)b );
}

// General purpose search() function that can be used
// for searching an element *x in an array arr[] of
// arr_size. Note that void pointers are used so that
// the function can be called by passing a pointer of
// any type. ele_size is size of an array element
int search(void *arr, int arr_size, int ele_size, void *x,
          bool compare (const void * , const void *))
{
    // Since char takes one byte, we can use char pointer
    // for any type/ To get pointer arithmetic correct,
    // we need to multiply index with size of an array
    // element ele_size
    char *ptr = (char *)arr;

    int i;
    for (i=0; i<arr_size; i++)
        if (compare(ptr + i*ele_size, x))
            return i;

    // If element not found
    return -1;
}

```

```
int main()
{
    int arr[] = {2, 5, 7, 90, 70};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 7;
    printf ("Returned index is %d ", search(arr, n,
                                             sizeof(int), &x, compare));
    return 0;
}
```

Output:

Returned index is 2

The above search function can be used for any data type by writing a separate customized compare().

---