

## Pointer Type in C

A Pointer Variable in C is a variable that stores the address of another variable, or the address of some other data structure, such as an array. To put it another way, a variable which stores the address of another variable is said to point to that variable.

THE ADDRESS-OF OPERATOR & RETURNS THE ADDRESS OF THE VARIABLE TO ITS RIGHT.  
FOR INSTANCE

```
#include <stdio.h>
int main (void) {
    int x = 6, y = 7;
    printf("x = %d, y = %d\n", x, y);
    printf("&x = %p, &y = %p\n", &x, &y);
    return 0;
}
```

ITS OUTPUT :

```
x = 6, y = 7
&x = 11betaac, &y = 11betaa8
```

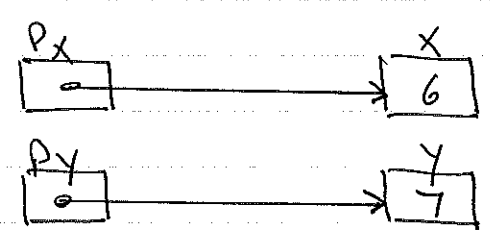
NOTE THAT THE FORMAT CONTROL CODE %P CAUSES printf TO PRINT A POINTER (I.E. AN ADDRESS) IN HEXIDECIMAL (BASE 16.)

IF YOU COMPILe AND RUN THE PRECEDING EXAMPLE THE ADDRESSES WILL MOST CERTAINLY BE DIFFERENT.

EACH DATA TYPE IN C HAS A CORRESPONDING POINTER TYPE CAPABLE OF STORING THE ADDRESS OF A VARIABLE OF THAT BASE TYPE. WE SPECIFY A POINTER VARIABLE BY PLACING A \* AFTER THE BASE TYPE NAME.

```
Ex int x=6, y=7;
int *px;
int *py;

px = &x;
py = &y;
```



NOW WE CAN SAY px POINTS TO x AND py POINTS TO y. THE VALUES STORED IN VARIABLE x AND y CAN BE ACCESSED THROUGH THE POINTERS px AND py USING THE INDIRECTION OPERATOR \*.

```
printf(" *px = %d \n", *px);
printf(" x = %d \n", x);
```

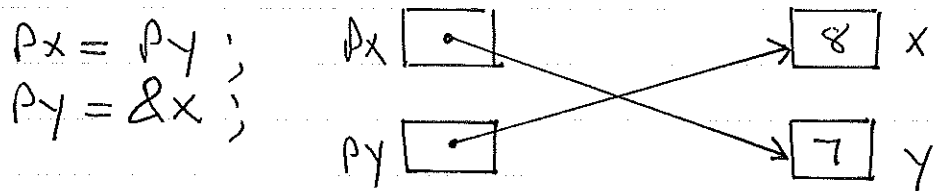
OUTPUT: \*px = 6
x = 6

WE CAN THINK OF INDIRECTION \* AS THE VALUE-AT OPERATOR, SINCE IT RETURNS THE VALUE STORED AT AN ADDRESS. THIS OPERATOR CAN ALSO APPEAR ON THE LEFT HAND SIDE OF AN ASSIGNMENT

```
*Px = 8
printf("x = %d\n", x)
```

OUTPUT: x = 8

POINTER VARIABLES CAN BE MADE TO POINT TO ANY ADDRESS STORING THE APPROPRIATE TYPE. CONTINUING THE PREVIOUS EXAMPLE



SO THAT

```
printf(" *Px = %d\n", *Px);
printf(" *Py = %d\n", *Py);
```

THE OUTPUT: \*Px = 7
              \*Py = 8

ALTHOUGH IT IS HELPFUL TO THINK OF \* IN THE DECLARATION `int *Px` AS MODIFYING THE DATA TYPE `int`, IN FACT IT MODIFIES `Px`.

THUS ONE MIGHT THINK

```
int * px, py;
```

DECLARED PY TO BE OF TYPE int\* (READ POINTER TO int), ACTUALLY PY IS OF TYPE int. INSTEAD WE MUST DO

```
int *px, *py;
```

THE COMPILER SEES THIS AS SAYING THAT VALUE AT PY WILL BE OF TYPE int, WHICH MEANS PY IS OF TYPE POINTER TO int.

POINTER VARIABLES CAN BE USED TO PASS FUNCTION ARGUMENTS BY REFERENCE, AS THE FOLLOWING EXAMPLE ILLUSTRATES.

```
void swap(int *px, int *py) {
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}

int main(void) {
    int x = 6, y = 7;
    printf("x = %d, y = %d\n", x, y);
    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

OUTPUT :

x = 6, y = 7

x = 7, y = 6

## ARRAYS

AN ARRAY NAME IN C IS ACTUALLY A POINTER TO THE FIRST ELEMENT IN THE ARRAY (i.e. THE ELEMENT WITH INDEX 0.)

THUS GIVEN THE DECLARATION

```
int x[] = {3, 1, 2, 7, 9};
```

THE EXPRESSIONS  $x == \&x[0]$  AND  $*x == x[0]$  WILL EVALUATE TO TRUE. ACTUALLY THE ARRAY NAME  $x$  ABOVE IS OF TYPE  $\text{const int}^*$ , WHICH MEANS THAT  $x$  CANNOT BE ASSIGNED TO POINT TO SOME OTHER  $\text{int}$ .

THUS ARRAYS ARE ALWAYS PASSED TO FUNCTIONS BY REFERENCE, AND ARE PASSED BY SIMPLY PASSING THE ARRAY NAME.

EX.

```
void swap (int * A, int i, int j) {
    int temp;
    temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
```

```

void arrayReverse (int x[], int n) {
    int i=0, j=n-1;
    while (i < j) {
        swap(x, i, j);
        i++;
        j--;
    }
}

```

```

int main (void) {
    int i, a[] = {1, 2, 3, 4, 5};

    for (i=0; i < 5; i++) printf("%d ", a[i]);
    printf("\n");
    arrayReverse(a, 5);
    for (i=0; i < 5; i++) printf("%d ", a[i]);
    printf("\n");
    return 0;
}

```

OUTPUT:

```

1 2 3 4 5
5 4 3 2 1

```

NOTE THAT THE HEADINGS

```

void swap (int *A, int i, int j)
AND
void swap (int A[], int i, int j)

```

ARE ENTIRELY EQUIVALENT.

NOTICE THAT ARRAYS ARE DECLARED DIFFERENTLY IN C THAN IN JAVA

```

int x[4];
int x[4] = {1, 2, 3, 4};
int x[] = {1, 2, 3, 4};

```

ALL OK. THE LAST TWO ARE EQUIVALENT. THE FOLLOWING DO NOT WORK IN C

```

int x[];
int[] x;
int x[n]; /* where n is a variable */

```

however

```

int *x;

```

is valid, BUT x DOES NOT yet POINT TO an array.