**Pointers**

&var - returns address

\*var - returns value

#include <stdio.h>  
int main()  
{  
 int \*p, q;  
 q = 199;  
 p = &q;  
 printf("%d %d", \*p, q);  
}

C

OUTPUT: 199 (address of q)

199 is just the value of q. \*p points to q. The value of p is the address of q.

#include <stdio.h>  
int main()  
{  
 int q=5;  
 int \*fp;  
 fp = &q;  
 \*fp = 100; *//value of q changed to 100 printf("%d", q);*}

C

OUTPUT: 100

#include <stdio.h>  
int main()  
{  
 int q, \*p;  
 long long lq, \*lp;  
 q = 5, lq = 7;  
 p = &q, lp = &lq;  
 p++, lp++;  
 *//increases value of p and lp by 1 (4 bytes and 8 bytes respectively)*}

C

|  |  |  |
| --- | --- | --- |
|  | Address | value |
| q | 100 | 5 |
| \*p | 104 (int takes 4 bytes) | 100 (changed to 104) |
| lq | 106 | 7 |
| \*lp | 114 (long long takes 8 bytes) | 106 (changed to 114) |

\*p and \*lp aren’t pointing to anything anymore

#include <stdio.h>  
int main()  
{  
 int arr[] = {1, 2, 9, 4, 7, 6, 3};  
 int \*p = &arr[0];  
 printf("%d", \*p);  
  
 p++;  
 printf("%d", \*p);  
  
 printf("%d", p[3]); *//print the value that is at p+3* int n=1000;  
 printf("%d", p[6]); *//goes to next memory since array ends at p[5]*}

C

OUTPUT: 1 2 7 1000

Pointers can give access to memory that was not intended to be accessed. This is a security risk, but only the C language still uses pointers.

#include <stdio.h>  
int func(int arr[100]);  
int main()  
{  
 int arr[100];  
 func(arr[100]);  
}

C

Array is erased from memory after function finishes. Using a pointer will prevent this.

#include <stdio.h>  
int func(int \*arr[100]);  
int main()  
{  
 int arr[100];  
 func(arr[100]);  
}

C

int main(int argc, char argr[])  
{  
 for (int i=0; i<argc; i++) puts(argr[i]);  
}  
*//If input a and b are given, argc = 3 and argr[3] = {a, b}.*

C

OUTPUT: a

b

Arguments can be given to the file itself using command prompt. argc stores the number of arguments and argr stores those arguments.