


typedef versus #define in C



typedef : typedef is used to give data type a new name, for example



```
// C program to demonstrate typedef
#include <stdio.h>


// After this line BYTE can be used
// in place of unsigned char
typedef unsigned char BYTE;

int main()
{
    BYTE b1, b2;
    b1 = 'c';
    printf("%c ", b1);
    return 0;
}
```



c



#define : is a C directive which is used to #define alias.



```
// C program to demonstrate #define
#include <stdio.h>

// After this line HYD is replaced by
// "Hyderabad"
#define HYD "Hyderabad"


int main()
{
    printf("%s ", HYD);
    return 0;
}
```



Hyderabad

typedef is different from #define among the following aspects

- typedef is limited to giving symbolic names to types only, where as #define can be used to define alias for values as well, e.g., you can define 1 as ONE, 3.14 as PI, etc.
- typedef interpretation is performed by the compiler where #define statements are performed by preprocessor.
- #define should not be terminated with semicolon, but typedef should be terminated with semicolon.
- #define will just copy-paste the definition values at the point of use, while typedef is actual definition of a new type.
- typedef follows the scope rule which means if a new type is defined in a scope (inside a function), then the new type name will only be visible till the scope is there. In case of #define, when preprocessor encounters #define, it replaces all the occurrences, after that (No scope rule is followed).



```
// C program to demonstrate importance
// of typedef over #define for data types
#include <stdio.h>
typedef char* ptr;
#define PTR char*
int main()
{
    ptr a, b, c;
    PTR x, y, z;
    printf("sizeof a:%u\n", sizeof(a) );
    printf("sizeof b:%u\n", sizeof(b) );
    printf("sizeof c:%u\n", sizeof(c) );
    printf("sizeof x:%u\n", sizeof(x) );
    printf("sizeof y:%u\n", sizeof(y) );
    printf("sizeof z:%u\n", sizeof(z) );
    return 0;
}
```

```
sizeof a:8
sizeof b:8
sizeof c:8
sizeof x:8
sizeof y:1
sizeof z:1
```

From the output of the above program size of "a" which is a pointer is 8 (on a machine where pointers are stored using 8 bytes). In the above program, when the compiler comes to

```
typedef char* ptr;  
ptr a, b, c;
```

the statement effectively becomes

```
char *a, *b, *c;
```

This declares a, b, c as char*.

In contrast, #define works like this:

```
#define PTR char*  
PTR x, y, z;
```

the statement effectively becomes

```
char *x, y, z;
```

This makes x, y and z different, as, x is pointer-to-a char, whereas, y and z are char variables. When we declare macros with pointers while defining if we declare more than one identifier then the actual definition is given to the first identifier and for the rest non-pointer definition is given. In the above case x will be declared as char*, so its size is the size of a pointer, whereas, y and z will be declared as char so, their size will be 1 byte.