# Typedef for Function Pointers

## Example #

We can use `typedef` to simplify the usage of function pointers. Imagine we have some functions, all having the same signature, that use their argument to print out something in different ways:

```
#include<stdio.h>

void print_to_n(int n)
{
    for (int i = 1; i <= n; ++i)
        printf("%d\n", i);
}

void print_n(int n)
{
    printf("%d\n, n);
}
```

Now we can use a `typedef` to create a named function pointer type called printer:

```
typedef void (*printer_t)(int);
```

This creates a type, named `printer_t` for a pointer to a function that takes a single `int` argument and returns nothing, which matches the signature of the functions we have above. To use it we create a variable of the created type and assign it a pointer to one of the functions in question:

```
printer_t p = &print_to_n;
void (*p)(int) = &print_to_n; // This would be required without the type
```

Then to call the function pointed to by the function pointer variable:

```
p(5);           // Prints 1 2 3 4 5 on separate lines
(*p)(5);        // So does this
```

Thus the `typedef` allows a simpler syntax when dealing with function pointers. This becomes more apparent when function pointers are used in more complex situations, such as arguments to functions.

If you are using a function that takes a function pointer as a parameter without a function pointer type defined the function definition would be,

```
void foo (void (*printer)(int), int y){
    //code
    printer(y);
    //code
}
```

However, with the `typedef` it is:

```
void foo (printer_t printer, int y){
    //code
    printer(y);
    //code
}
```

Likewise functions can return function pointers and again, the use of a `typedef` can make the syntax simpler when doing so.

A classic example is the `signal` function from `<signal.h>`. The declaration for it (from the C standard) is:

```
void (*signal(int sig, void (*func)(int)))(int);
```

That's a function that takes two arguments — an `int` and a pointer to a function which takes an `int` as an argument and returns nothing — and which returns a pointer to function like its second argument.

If we defined a type `SigCatcher` as an alias for the pointer to function type:

```
typedef void (*SigCatcher)(int);
```

then we could declare `signal()` using:

```
SigCatcher signal(int sig, SigCatcher func);
```

On the whole, this is easier to understand (even though the C standard did not elect to define a type to do the job). The `signal` function takes two arguments, an `int` and a `SigCatcher`, and it returns a `SigCatcher` — where a `SigCatcher` is a pointer to a function that takes an `int` argument and returns nothing.

Although using `typedef` names for pointer to function types makes life easier, it can also lead to confusion for others who will maintain your code later on, so use with caution and proper documentation. See also Function Pointers.