Chapter 42: Typedef

The `typedef` mechanism allows the creation of aliases for other types. It does not create new types. People often use `typedef` to improve the portability of code, to give aliases to structure or union types, or to create aliases for function (or function pointer) types.

In the C standard, `typedef` is classified as a 'storage class' for convenience; it occurs syntactically where storage classes such as `static` or **`extern`** could appear.

Section 42.1: Typedef for Structures and Unions

You can give alias names to a `struct`:

```
typedef struct Person {
    char name[32];
    int age;
} Person;

Person person;
```

Compared to the traditional way of declaring structs, programmers wouldn't need to have `struct` every time they declare an instance of that struct.

Note that the name `Person` (as opposed to `struct Person`) is not defined until the final semicolon. Thus for linked lists and tree structures which need to contain a pointer to the same structure type, you must use either:

```
typedef struct Person {
    char name[32];
    int age;
    struct Person *next;
} Person;
```

or:

```
typedef struct Person Person;

struct Person {
    char name[32];
    int age;
    Person *next;
};
```

The use of a `typedef` for a `union` type is very similar.

```
typedef union Float Float;

union Float
{
    float f;
    char  b[sizeof(float)];
};
```

A structure similar to this can be used to analyze the bytes that make up a `float` value.

Section 42.2: Typedef for Function Pointers

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.

Section 42.3: Simple Uses of Typedef

For giving short names to a data type

Instead of:

```
long long int foo;
struct mystructure object;
```

one can use

```
/* write once */
typedef long long ll;
typedef struct mystructure mystruct;

/* use whenever needed */
ll foo;
mystruct object;
```

This reduces the amount of typing needed if the type is used many times in the program.

Improving portability

The attributes of data types vary across different architectures. For example, an `int` may be a 2-byte type in one implementation and an 4-byte type in another. Suppose a program needs to use a 4-byte type to run correctly.

In one implementation, let the size of `int` be 2 bytes and that of `long` be 4 bytes. In another, let the size of `int` be 4 bytes and that of `long` be 8 bytes. If the program is written using the second implementation,

```
/* program expecting a 4 byte integer */
int foo; /* need to hold 4 bytes to work */
/* some code involving many more ints */
```

For the program to run in the first implementation, all the `int` declarations will have to be changed to `long`.

```
/* program now needs long */
long foo; /*need to hold 4 bytes to work */
/* some code involving many more longs - lot to be changed */
```

To avoid this, one can use `typedef`

```
/* program expecting a 4 byte integer */
typedef int myint; /* need to declare once - only one line to modify if needed */
myint foo; /* need to hold 4 bytes to work */
/* some code involving many more myints */
```

Then, only the `typedef` statement would need to be changed each time, instead of examining the whole program.

`Version ≥ C99`

The **`<stdint.h>`** header and the related **`<inttypes.h>`** header define standard type names (using `typedef`) for integers of various sizes, and these names are often the best choice in modern code that needs fixed size integers. For example, `uint8_t` is an unsigned 8-bit integer type; `int64_t` is a signed 64-bit integer type. The type `uintptr_t` is an unsigned integer type big enough to hold any pointer to object. These types are theoretically optional — but it is rare for them not to be available. There are variants like `uint_least16_t` (the smallest unsigned integer type with at least 16 bits) and `int_fast32_t` (the fastest signed integer type with at least 32 bits). Also, `intmax_t` and `uintmax_t` are the largest integer types supported by the implementation. These types are mandatory.

To specify a usage or to improve readability

If a set of data has a particular purpose, one can use `typedef` to give it a meaningful name. Moreover, if the property of the data changes such that the base type must change, only the `typedef` statement would have to be changed, instead of examining the whole program.