4.5 Typedef: New names for types

- A `typedef` creates a **new name** for a type.
- **Portability**: `typedef`'d types like `size_t`, `ptrdiff_t` may be defined differently, depending on platform.
- **Mnemonic**: A type `TreePtr` may be easier to understand than one declared as a pointer to a complicated structure

Definition syntax:

```
1 typedef type name;
```

(in the most simple case)

- the keyword `typedef`,
- a type, and
- a name for the new type.

More exactly, the syntax is *almost the same* as for variable declarations, with only two differences:

- Presence of the keyword `typedef`.
- The name refers to a type, instead of a variable.

- A simple alias

```
1 typedef int t1;
2 t1 v1 = 23;                                          /* int v1 */
```

- You must specify the array size (or define a pointer instead)

```
1 typedef int *t2[8];
2 t2 v2 = { NULL, &v1 };                               /* int *v2[8] */
```

- Example with structures

```
1 typedef struct point { int x, y; } t3;
2 t3 v3 = { 21, 42 };                    /* struct point { int x, y; } v3; */
3 struct point v4 = { 0, 0 };
4
5 typedef struct { int x, y; } t5;                     /* without defining a tag */
6 t5 v5 = { 21, 42 };                          /* struct { int x, y; } v5; */
```

- Very bad style, reloaded

```
1 typedef int *t7, t8, * const t9;              /* int *v7, v8, * const v9; */
2 t7 v7 = &v1;
3 t8 v8 = 213;
4 t9 v9 = v7;
```

Note In the Linux kernel, definition of new types is not allowed!

- A `typedef` **hides details**: Was it an enum? A pointer? An array?
- An exception is pointers to functions (we will cover that soon), which look *very* complicated without `typedef`s.

```
1 $ checkpatch.pl typedef.c
2 WARNING: do not add new typedefs
3 #15: FILE: typedef.c:15:
4 +       typedef int t1;
```

We deviate from this regulation:

- That's why we have added `--ignore NEW_TYPEDEFS` to the command line arguments when running `checkpatch.pl`.

Fields, identifiers, tags

The names and tags you use in a C program, live in different **namespaces**.

- **Identifiers** of variables and types share one namespace.
  ⇒ You cannot name a variable like a type (*e.g.*, `int int;`)

  (There is an exception, but simply don't do it!)

- **Field names** are like variables, with a scope limited to that struct.

```
1 struct { int foo; } x;  x.foo = 3;
2 struct { char foo; } y;  y.foo = 'w';
3 double foo = 3.14;
```

- **Tags** have their own namespace, which is *shared* by unions, structs, and enumerations.

```
1 struct point { int x; int y; };
2 char point = '.';  /* valid */
3 enum point { infinity, closeby };  /* invalid redefinition of the tag point */
```

Tag names and identifiers are limited to the scope they are defined in.

What about these?

```
1 typedef struct foo { char foo; } foo;
2 foo v6 = { .foo = 'f' };
```

```
3 struct bar { char bar; } bar;
4 bar.bar = 'b';
```

```
5 typedef struct qux { char qux; } qux;
6 qux qux = { .qux = 'q' };
```

typedef vs. #define

- `typedef` is interpreted by the compiler, `#define` is removed by the preprocessor (*cf.* later), by **sourcecode modification** and thus *invisible* to the compiler.
- Macro typename can be extended with other specifiers

```
1 #define count int
2 unsigned count i;  /* works fine */
3
4 typedef int count;
5 unsigned count i;  /* illegal */
```

- `typedef`'d name provides the type for every declarator in a declaration

```
1 typedef int *int_ptr;
2 int_ptr chalk, cheese;
3
4 #define char_ptr char *
5 char_ptr Mercedes, BMW, VW;
```

4.6 Pointers to Functions

- A **function** itself *is not* a variable.
- But it is possible to define **pointers** to functions.
- Can be **assigned**, placed in **arrays**, **passed** to/**returned** from functions.

Syntax step-by-step examples of declarations:

- `int fun(char c, double x);` nothing new!
  - The expression `fun('Q', 3.14)` is of type `int`.
- `int *fun(char c, double x);` nothing new!
  - The expression `*fun('Q', 3.14)` is of type `int`.
  - Dereferencing `fun('Q', 3.14)` is of type `int`.
- `int (*fun)(char c, double x);`
  - The expression `(*fun)('Q', 3.14);` is of type `int`.
  - Dereferencing `fun`, and applying the result to `'Q',3.14`, is of type `int`.

  $\Rightarrow$ We have just dereferenced a function!

Example

```
1 size_t strlen(const char *s);    /* available with #include <string.h> */
2
3 size_t (*fp)(const char *);
4 fp = &strlen;
5 printf("result = %zu\n", (*fp)("hello world"));
```

- This can be abbreviated:

```
5 printf("result = %zu\n", fp("hello world"));
```

- Nicer with `typedef`

```
2 typedef size_t (*func)(const char *);
3 func fp = &strlen;
```

Note Function pointers are heavily used in the real world! *E.g.*,

- pass a comparing function to a queue datastructure;
- installing signal handlers (*cf.* OS lecture, and later in this course); and
- abstractions (syscall interface, subclasses, VFS, ...).

Question Can you read `int (*(*f)(void))[2]` ?