From "The C Programming Language" 2nd ed.
B.W. Kernighan & D.M. Ritchie

6.7 Typedef

C provides a facility called `typedef` for creating new data type names. For example, the declaration

```
typedef int Length;
```

makes the name `Length` a synonym for `int`. The type `Length` can be used in declarations, casts, etc., in exactly the same ways that the type `int` can be:

```
Length  len, maxlen;
Length  *lengths[];
```

Similarly, the declaration

```
typedef char *String;
```

makes `String` a synonym for `char *` or character pointer, which may then be used in declarations and casts:

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

Notice that the type being declared in a `typedef` appears in the position of a variable name, not right after the word `typedef`. Syntactically, `typedef` is like the storage classes `extern`, `static`, etc. We have used capitalized names for `typedefs`, to make them stand out.

As a more complicated example, we could make `typedefs` for the tree nodes shown earlier in this chapter:

```
typedef struct tnode *Treeptr;

typedef struct tnode {    /* the tree node: */
    char *word;               /* points to the text */
    int count;                /* number of occurrences */
    Treeptr left;             /* left child */
    Treeptr right;            /* right child */
} Treenode;
```

This creates two new type keywords called `Treenode` (a structure) and `Treeptr` (a pointer to the structure). Then the routine `talloc` could become

```
Treeptr talloc(void)
{
    return (Treeptr) malloc(sizeof(Treenode));
}
```

It must be emphasized that a `typedef` declaration does not create a new type in any sense; it merely adds a new name for some existing type. Nor are there any new semantics: variables declared this way have exactly the same properties as variables whose declarations are spelled out explicitly. In effect, `typedef` is like `#define`, except that since it is interpreted by the compiler, it

can cope
preprocesso

typ

creates th
returning

PFI

in the sort
Beside
typedef
If typed
typedef
to use ty
set of ch
size_t a
The s
program-
declared

6.8 Un

A *uni*
types an
ments. U
area of s
program.
As a
suppose
value of
yet it is
amount
the purp
several t

u

}

The
specific
assigned
the type

can cope with textual substitutions that are beyond the capabilities of the preprocessor. For example,

```
typedef int (*PFI)(char *, char *);
```

creates the type `PFI`, for "pointer to function (of two `char *` arguments) returning `int`," which can be used in contexts like

```
PFI strcmp, numcmp;
```

in the sort program of Chapter 5.

Besides purely aesthetic issues, there are two main reasons for using `typedef`s. The first is to parameterize a program against portability problems. If `typedef`s are used for data types that may be machine-dependent, only the `typedef`s need change when the program is moved. One common situation is to use `typedef` names for various integer quantities, then make an appropriate set of choices of `short`, `int`, and `long` for each host machine. Types like `size_t` and `ptrdiff_t` from the standard library are examples.

The second purpose of `typedef`s is to provide better documentation for a program—a type called `Treeptr` may be easier to understand than one declared only as a pointer to a complicated structure.

6.8 Unions

A *union* is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements. Unions provide a way to manipulate different kinds of data in a single area of storage, without embedding any machine-dependent information in the program. They are analogous to variant records in Pascal.

As an example such as might be found in a compiler symbol table manager, suppose that a constant may be an `int`, a `float`, or a character pointer. The value of a particular constant must be stored in a variable of the proper type, yet it is most convenient for table management if the value occupies the same amount of storage and is stored in the same place regardless of its type. This is the purpose of a union—a single variable that can legitimately hold any one of several types. The syntax is based on structures:

```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

The variable u will be large enough to hold the largest of the three types; the specific size is implementation-dependent. Any one of these types may be assigned to u and then used in expressions, so long as the usage is consistent: the type retrieved must be the type most recently stored. It is the programmer's

responsibility to keep track of which type is currently stored in a union; the results are implementation-dependent if something is stored as one type and extracted as another.

Syntactically, members of a union are accessed as

union-name . member

or

union-pointer -> member

just as for structures. If the variable `utype` is used to keep track of the current type stored in `u`, then one might see code such as

```
if (utype == INT)
    printf("%d\n", u.ival);
else if (utype == FLOAT)
    printf("%f\n", u.fval);
else if (utype == STRING)
    printf("%s\n", u.sval);
else
    printf("bad type %d in utype\n", utype);
```

Unions may occur within structures and arrays, and vice versa. The notation for accessing a member of a union in a structure (or vice versa) is identical to that for nested structures. For example, in the structure array defined by

```
struct {
    char *name;
    int flags;
    int utype;
    union {
        int ival;
        float fval;
        char *sval;
    } u;
} symtab[NSYM];
```

the member `ival` is referred to as

```
symtab[i].u.ival
```

and the first character of the string `sval` by either of

```
*symtab[i].u.sval
symtab[i].u.sval[0]
```

In effect, a union is a structure in which all members have offset zero from the base, the structure is big enough to hold the "widest" member, and the alignment is appropriate for all of the types in the union. The same operations are permitted on unions as on structures: assignment to or copying as a unit, taking the address, and accessing a member.

A union may only be initialized with a value of the type of its first member;

thus the union

The storag
variable to be

6.9 Bit-fiel

When stor
objects into a
applications li
as interfaces t
of a word.

Imagine a
identifier in a
whether or no
so on. The n
flags in a sing

The usual
relevant bit pc

```
#defin
#defin
#defin
```

or

```
enum {
```

The numbers
of "bit-fiddlin
were describe

Certain idi

```
flags
```

turns on the E

```
flags
```

turns them off

```
if ((f
```

is true if both

Although
capability of
bitwise logical
within a singl
The syntax of
the symbol ta

thus the union u described above can only be initialized with an integer value.

The storage allocator in Chapter 8 shows how a union can be used to force a variable to be aligned on a particular kind of storage boundary.

6.9 Bit-fields

When storage space is at a premium, it may be necessary to pack several objects into a single machine word; one common use is a set of single-bit flags in applications like compiler symbol tables. Externally-imposed data formats, such as interfaces to hardware devices, also often require the ability to get at pieces of a word.

Imagine a fragment of a compiler that manipulates a symbol table. Each identifier in a program has certain information associated with it, for example, whether or not it is a keyword, whether or not it is external and/or static, and so on. The most compact way to encode such information is a set of one-bit flags in a single `char` or `int`.

The usual way this is done is to define a set of "masks" corresponding to the relevant bit positions, as in

```
#define KEYWORD  01
#define EXTERNAL 02
#define STATIC   04
```

or

```
enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

The numbers must be powers of two. Then accessing the bits becomes a matter of "bit-fiddling" with the shifting, masking, and complementing operators that were described in Chapter 2.

Certain idioms appear frequently:

```
flags |= EXTERNAL | STATIC;
```

turns on the `EXTERNAL` and `STATIC` bits in `flags`, while

```
flags &= ~(EXTERNAL | STATIC);
```

turns them off, and

```
if ((flags & (EXTERNAL | STATIC)) == 0) ...
```

is true if both bits are off.

Although these idioms are readily mastered, as an alternative C offers the capability of defining and accessing fields within a word directly rather than by bitwise logical operators. A *bit-field*, or *field* for short, is a set of adjacent bits within a single implementation-defined storage unit that we will call a "word." The syntax of field definition and access is based on structures. For example, the symbol table `#defines` above could be replaced by the definition of three

fields:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern  : 1;
    unsigned int is_static  : 1;
} flags;
```

This defines a variable called `flags` that contains three 1-bit fields. The number following the colon represents the field width in bits. The fields are declared `unsigned int` to ensure that they are unsigned quantities.

Individual fields are referenced in the same way as other structure members: `flags.is_keyword`, `flags.is_extern`, etc. Fields behave like small integers, and may participate in arithmetic expressions just like other integers. Thus the previous examples may be written more naturally as

```
flags.is_extern = flags.is_static = 1;
```

to turn the bits on;

```
flags.is_extern = flags.is_static = 0;
```

to turn them off; and

```
if (flags.is_extern == 0 && flags.is_static == 0)
    ...
```

to test them.

Almost everything about fields is implementation-dependent. Whether a field may overlap a word boundary is implementation-defined. Fields need not be named; unnamed fields (a colon and width only) are used for padding. The special width 0 may be used to force alignment at the next word boundary.

Fields are assigned left to right on some machines and right to left on others. This means that although fields are useful for maintaining internally-defined data structures, the question of which end comes first has to be carefully considered when picking apart externally-defined data; programs that depend on such things are not portable. Fields may be declared only as `int`s; for portability, specify `signed` or `unsigned` explicitly. They are not arrays, and they do not have addresses, so the & operator cannot be applied to them.