```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

Here, we've declared a structure tag named `part` (making it possible to use `part` later to declare more variables) as well as variables named `part1` and `part2`.

All structures declared to have type `struct part` are compatible with one another:

```
struct part part1 = {528, "Disk drive", 10};
struct part part2;

part2 = part1;   /* legal; both parts have the same type */
```

Defining a Structure Type

As an alternative to declaring a structure tag, we can use `typedef` to define a genuine type name. For example, we could define a type named `Part` in the following way:

```
typedef struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} Part;
```

Note that the name of the type, `Part`, must come at the end, not after the word `struct`.

We can use `Part` in the same way as the built-in types. For example, we might use it to declare variables:

```
Part part1, part2;
```

Since `Part` is a `typedef` name, we're not allowed to write `struct Part`. All `Part` variables, regardless of where they're declared, are compatible.

When it comes time to name a structure, we can usually choose either to declare a structure tag or to use `typedef`. However, as we'll see later, declaring a structure tag is mandatory when the structure is to be used in a linked list. I'll use structure tags rather than `typedef` names in most of my examples.

Q&A

linked lists ➤ *17.5*

Structures as Arguments and Return Values

Functions may have structures as arguments and return values. Let's look at two examples. Our first function, when given a `part` structure as its argument, prints the structure's members:

```
void print_part(struct part p)
{
  printf("Part number: %d\n", p.number);
```