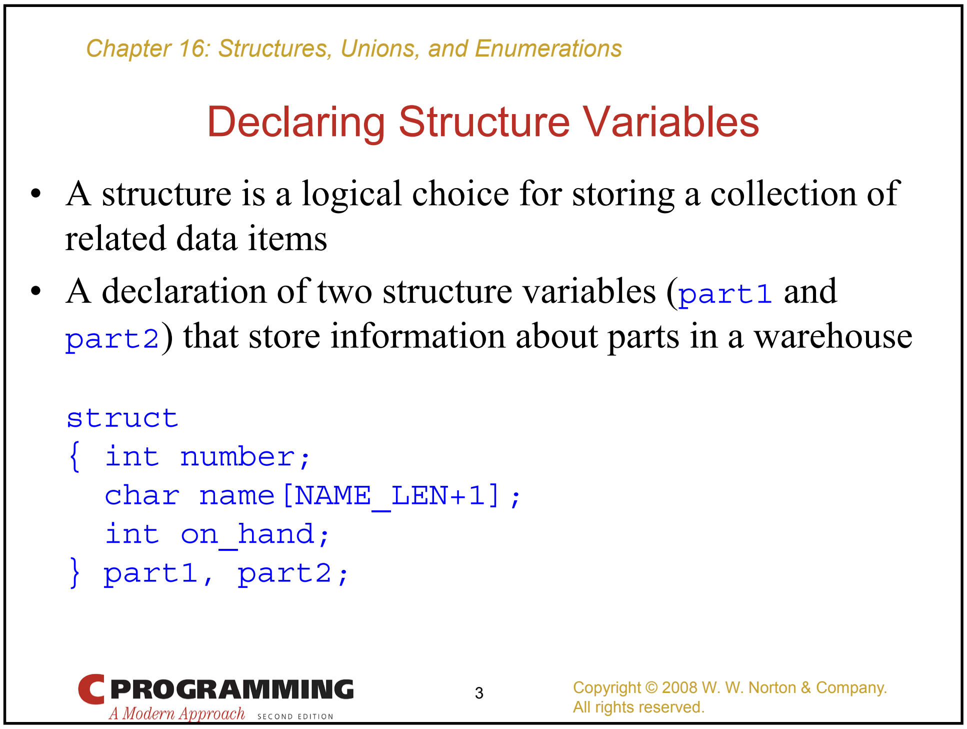
Chapter 16

Structures, Unions, and Enumerations

Structure Variables

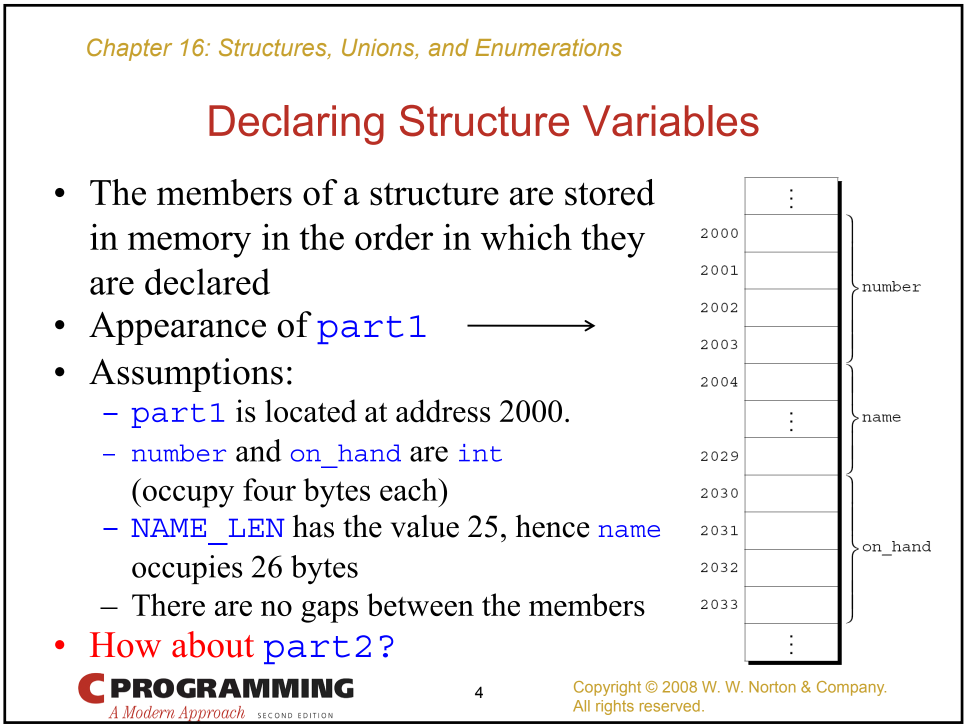
- The properties of a ***structure*** are different from those of an array
  - The elements of a structure (its ***members***) are not required to have the same type
  - The members of a structure have names; to select a particular member, we specify its name
- In some languages, *structures* are called ***records*****,** and *members* are known as ***fields***



Declaring Structure Variables

- A structure is a logical choice for storing a collection of related data items
- A declaration of two structure variables (`part1` and `part2`) that store information about parts in a warehouse

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

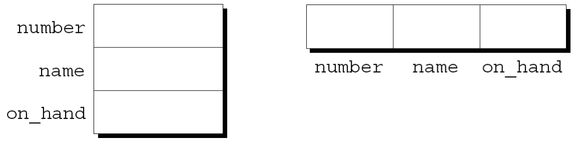


Declaring Structure Variables

- The members of a structure are stored in memory in the order in which they are declared
- Appearance of `part1` →
- Assumptions:
  - `part1` is located at address 2000.
  - `number` and `on_hand` are `int` (occupy four bytes each)
  - `NAME_LEN` has the value 25, hence `name` occupies 26 bytes
  - There are no gaps between the members
- How about `part2`?

Declaring Structure Variables

- Abstract representations of a structure



- Member values will go in the boxes later

Declaring Structure Variables

- Each structure represents a new scope
- Any names declared in that scope will not conflict with other names in a program. *For example*, the following declarations can appear in the same program

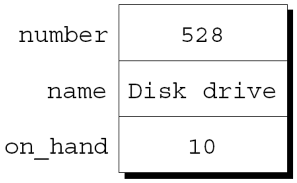
```
struct
{ int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;

struct
{ char name[NAME_LEN+1];
  int number;
  char sex;
} employee1, employee2;

int number, on_hand;
```

Initializing Structure Variables

- A structure declaration may include an initializer

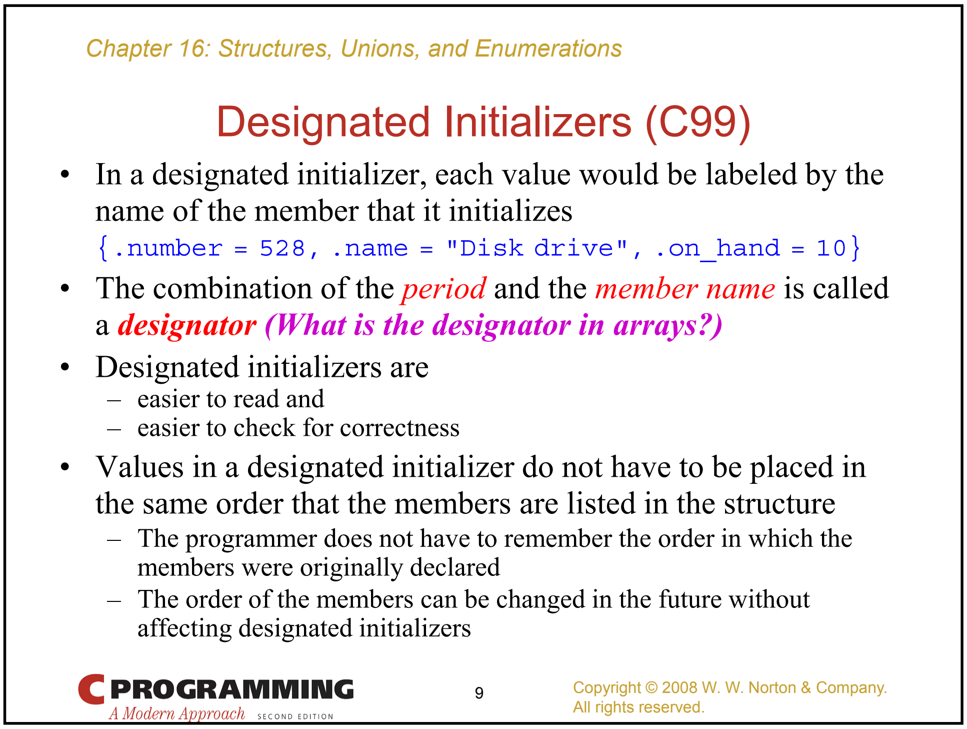
```
struct
{ int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1 = {528, "Disk drive", 10},
  part2 = {914, "Printer cable", 5};
```

- Appearance of `part1` after initialization



Initializing Structure Variables

- Structure initializers follow rules similar to those for array initializers
  - Expressions used in a structure initializer *must be* constant
  - An initializer *can have fewer members* than the structure it is initializing
  - Any “*leftover*” members are given `0` as their initial value

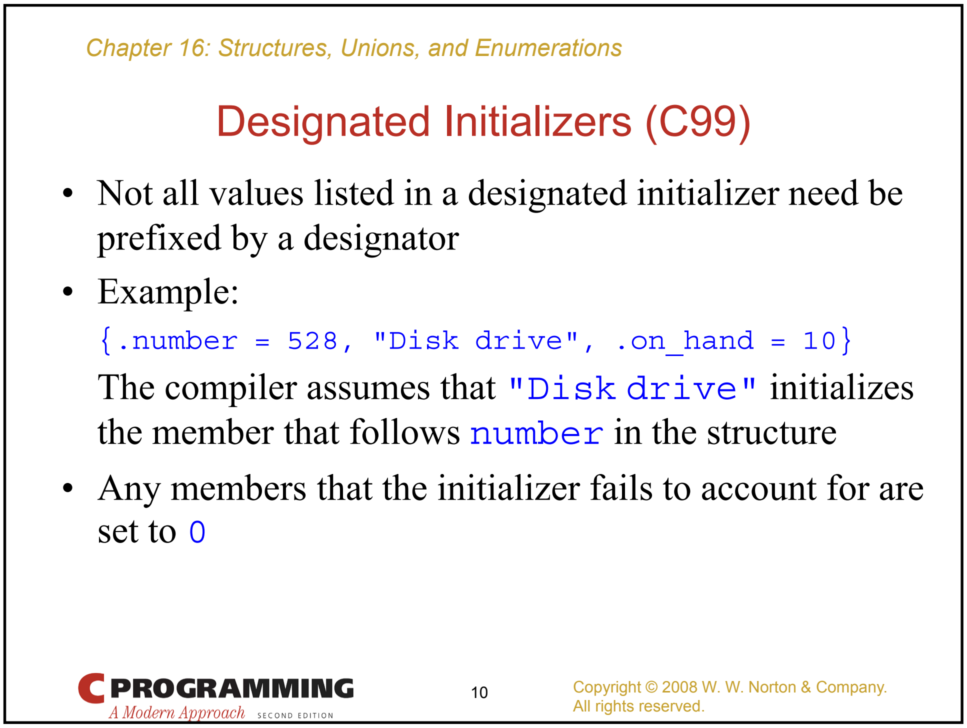


Designated Initializers (C99)

- In a designated initializer, each value would be labeled by the name of the member that it initializes

```
{.number = 528, .name = "Disk drive", .on_hand = 10}
```

- The combination of the *period* and the *member name* is called a ***designator (What is the designator in arrays?)***
- Designated initializers are
  - easier to read and
  - easier to check for correctness
- Values in a designated initializer do not have to be placed in the same order that the members are listed in the structure
  - The programmer does not have to remember the order in which the members were originally declared
  - The order of the members can be changed in the future without affecting designated initializers



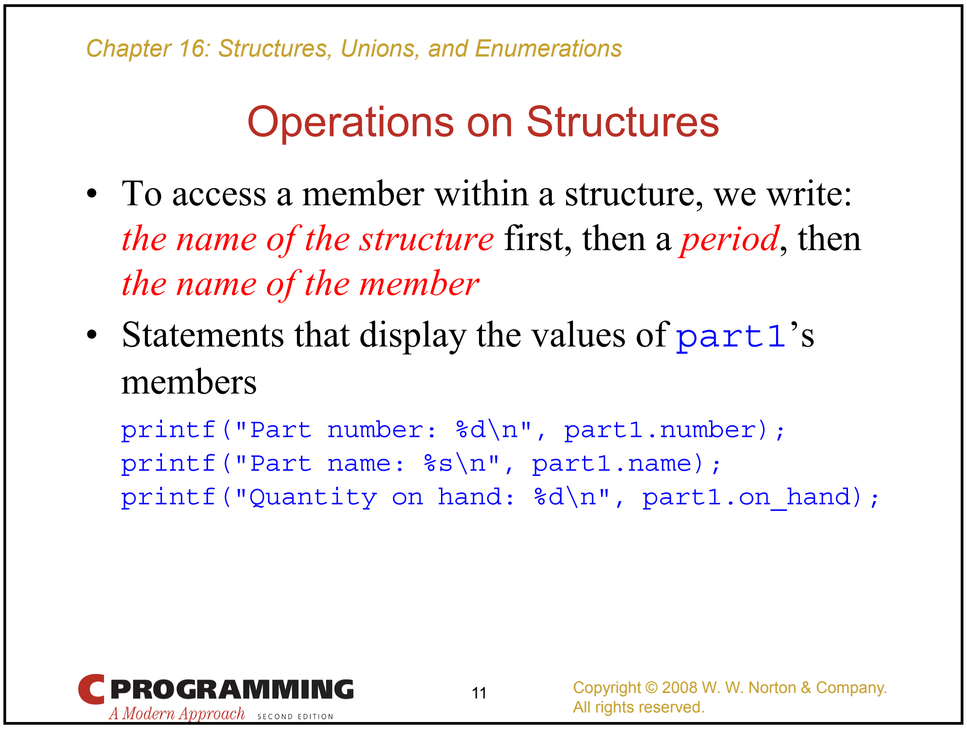
Designated Initializers (C99)

- Not all values listed in a designated initializer need be prefixed by a designator
- Example:

```
{.number = 528, "Disk drive", .on_hand = 10}
```

  The compiler assumes that `"Disk drive"` initializes the member that follows `number` in the structure

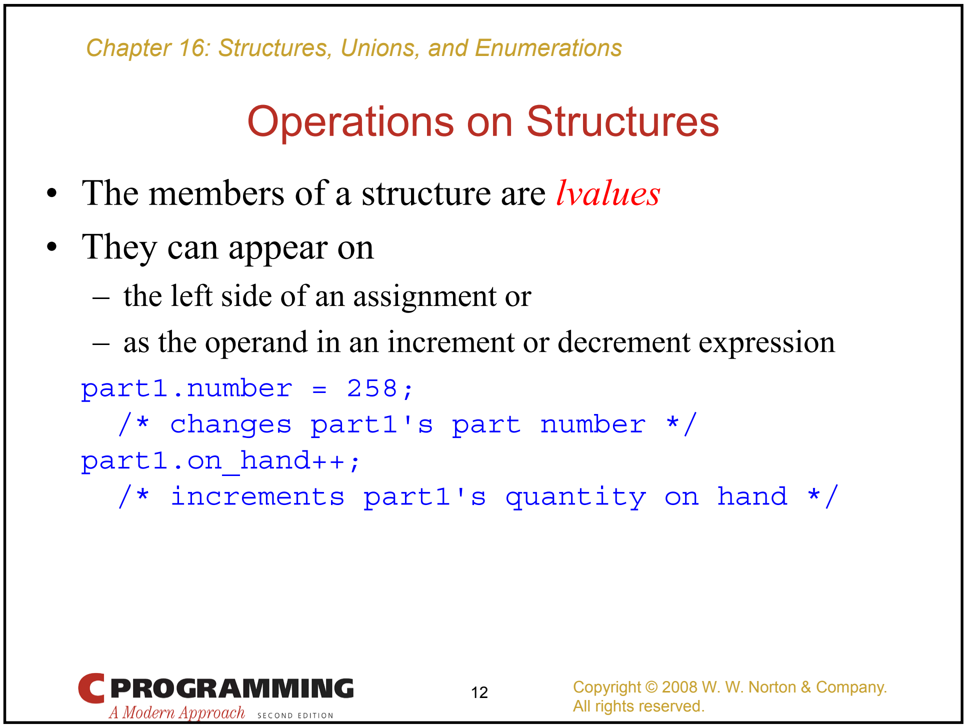
- Any members that the initializer fails to account for are set to `0`



Operations on Structures

- To access a member within a structure, we write: *the name of the structure* first, then a *period*, then *the name of the member*
- Statements that display the values of `part1`’s members

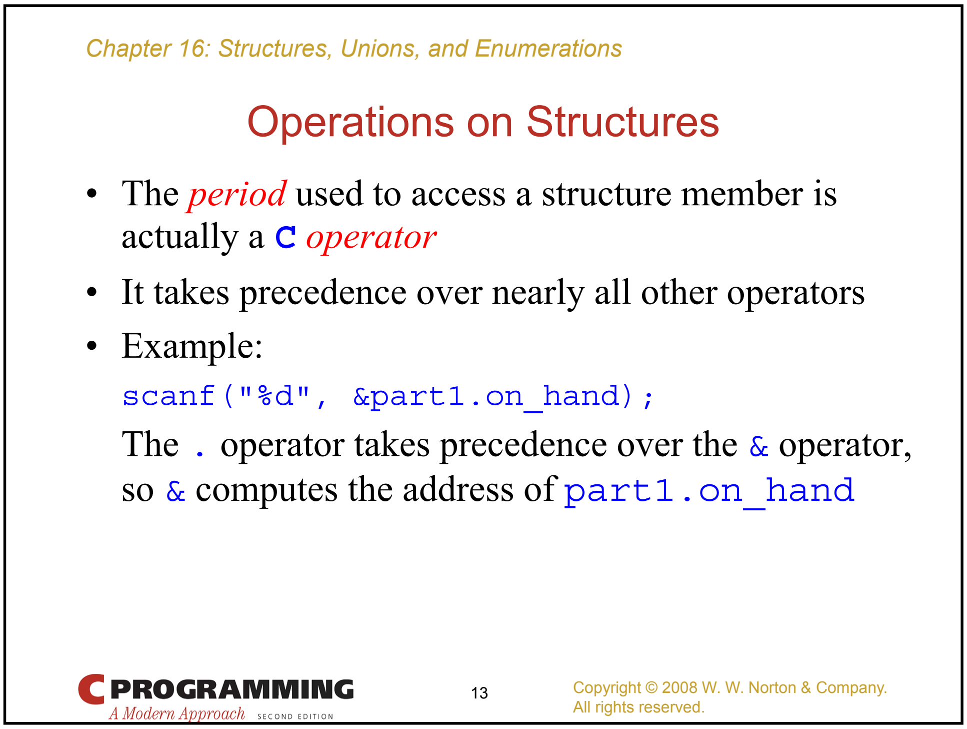
```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);
```



Operations on Structures

- The members of a structure are *lvalues*
- They can appear on
  - the left side of an assignment or
  - as the operand in an increment or decrement expression

```
part1.number = 258;
  /* changes part1's part number */
part1.on_hand++;
  /* increments part1's quantity on hand */
```

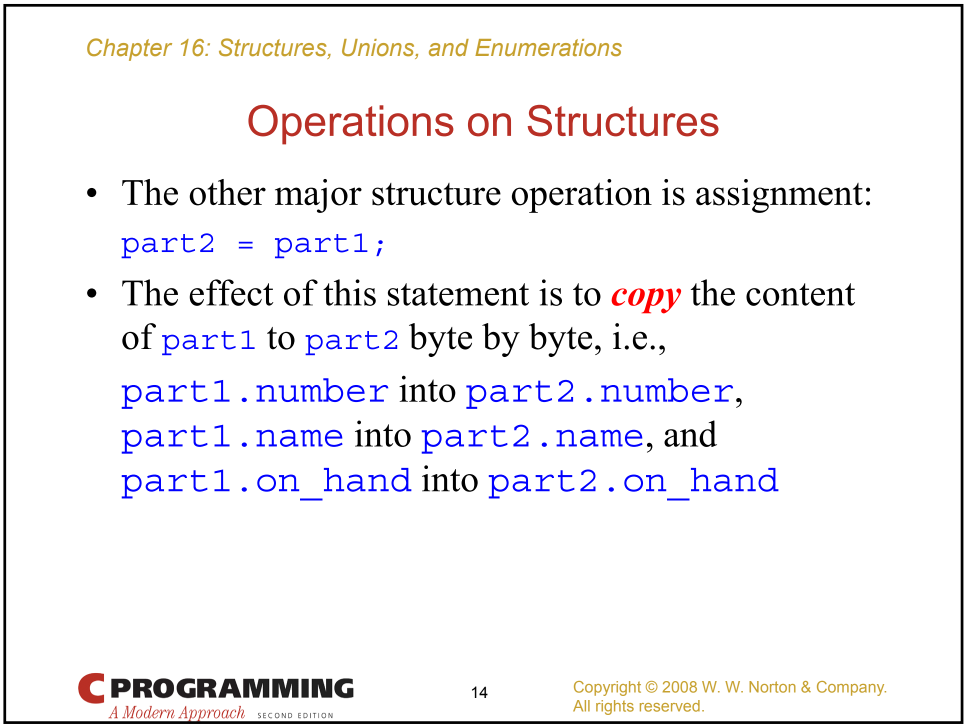


Operations on Structures

- The *period* used to access a structure member is actually a C *operator*
- It takes precedence over nearly all other operators
- Example:

```
scanf("%d", &part1.on_hand);
```

  The `.` operator takes precedence over the `&` operator, so `&` computes the address of `part1.on_hand`



Operations on Structures

- The other major structure operation is assignment:

```
part2 = part1;
```

- The effect of this statement is to ***copy*** the content of `part1` to `part2` byte by byte, i.e.,

  `part1.number` into `part2.number`,
  `part1.name` into `part2.name`, and
  `part1.on_hand` into `part2.on_hand`

Operations on Structures

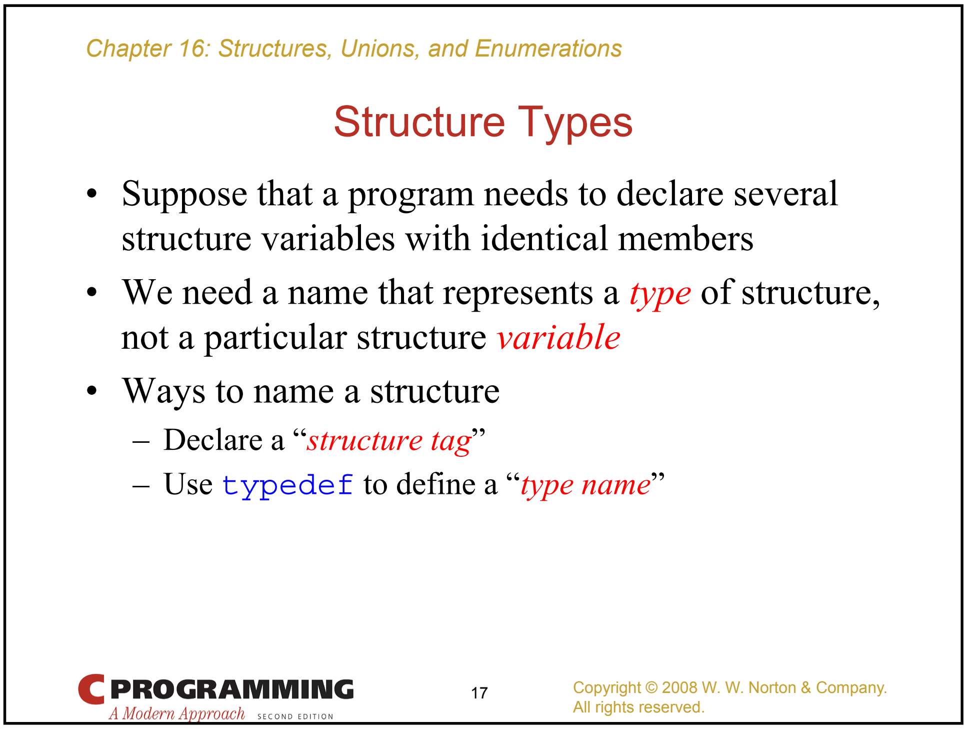
- *Arrays can not be copied using the = operator, but an array embedded within a structure is copied when the enclosing structure is copied*

- Some programmers exploit this property by creating "dummy" structures to enclose arrays that will be copied later

```
struct { int a[10]; } a1, a2;
a1 = a2;
  /* legal, since a1 and a2 are structures */
```

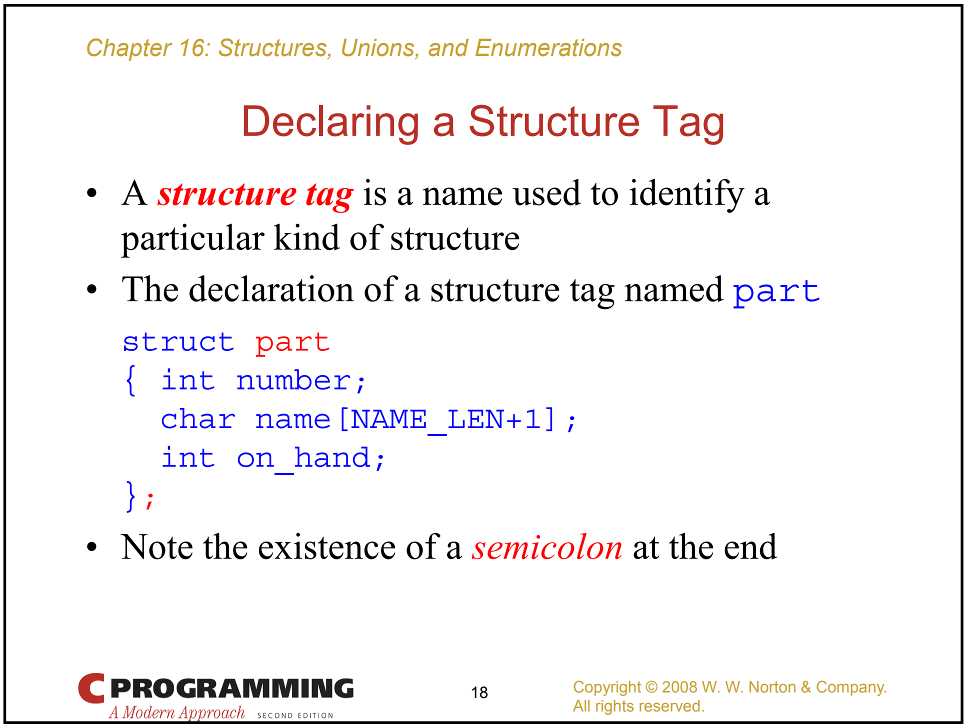
Operations on Structures

- The `=` operator can be used only with structures of ***compatible*** types
- Compatible structures can be
  - Structures declared at the same time (e.g., `part1` and `part2`)
  - Structures declared using the same "*structure tag*" or
  - Structures declared using the same "*type name*"
- Other than assignment, **`C`** provides *no* operations on entire structures
- In particular, the `==` and `!=` operators *can not* be used with structures



Structure Types

- Suppose that a program needs to declare several structure variables with identical members
- We need a name that represents a *type* of structure, not a particular structure *variable*
- Ways to name a structure
  - Declare a "*structure tag*"
  - Use `typedef` to define a "*type name*"



Declaring a Structure Tag

- A ***structure tag*** is a name used to identify a particular kind of structure
- The declaration of a structure tag named `part`

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

- Note the existence of a *semicolon* at the end

Declaring a Structure Tag

- The `part` tag can be used to declare variables

```
struct part part1, part2;
```

- We *can not* drop the word `struct`:

```
part part1, part2;    /*** WRONG ***/
```

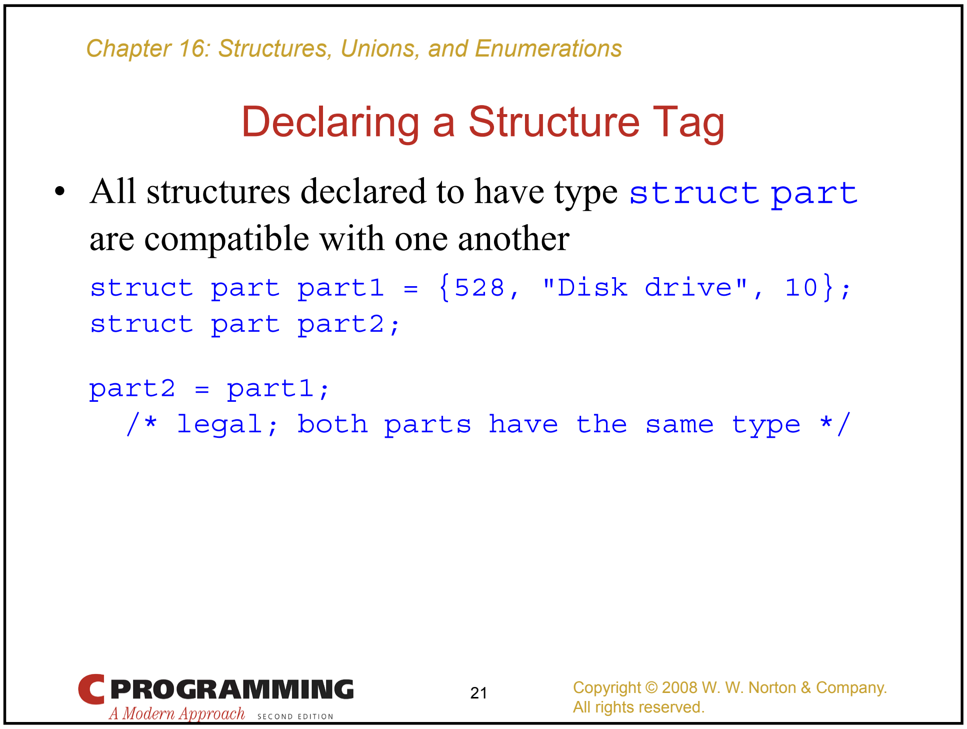
  - `part` is not a type name;
  - without the word `struct`, it is meaningless

- Since *structure tags* are not recognized unless preceded by the word `struct`, they do not conflict with other names used in a program

Declaring a Structure Tag

- The declaration of a *structure tag* can be combined with the declaration of structure *variables*

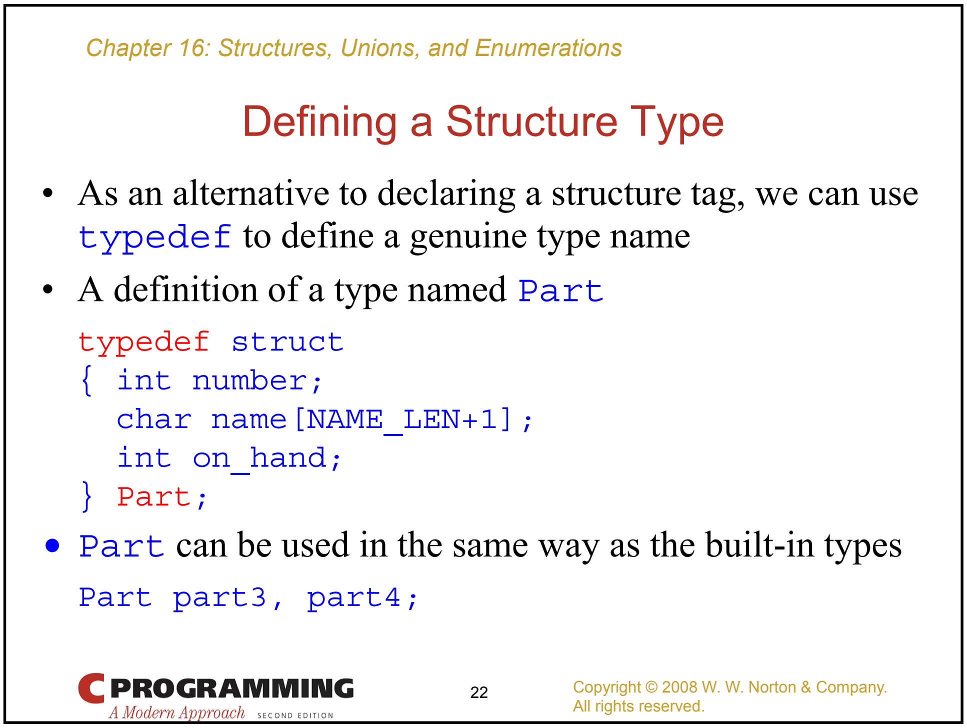
```
struct part
{ int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```



Declaring a Structure Tag

- 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 */
```



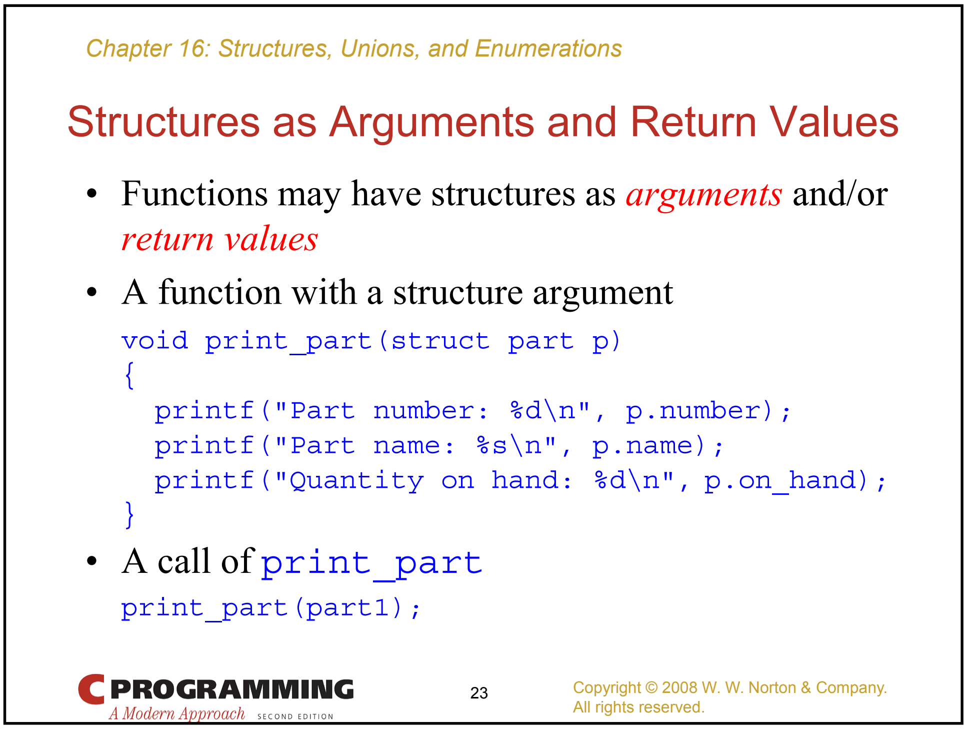
Defining a Structure Type

- As an alternative to declaring a structure tag, we can use `typedef` to define a genuine type name
- A definition of a type named `Part`

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

- `Part` can be used in the same way as the built-in types

```
Part part3, part4;
```



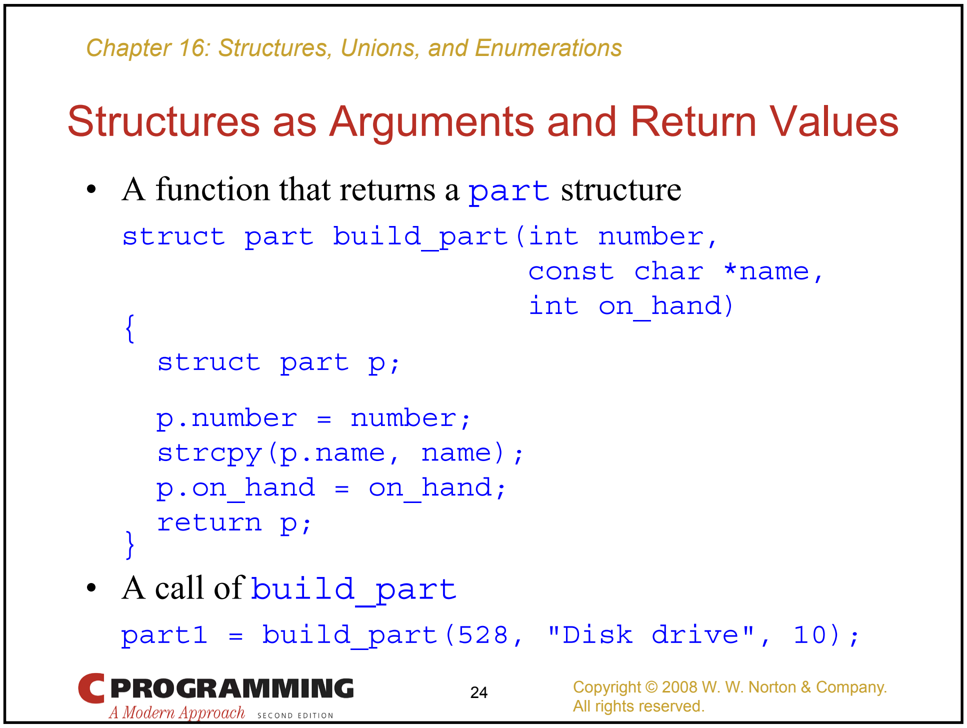
Structures as Arguments and Return Values

- Functions may have structures as *arguments* and/or *return values*
- A function with a structure argument

```
void print_part(struct part p)
{
  printf("Part number: %d\n", p.number);
  printf("Part name: %s\n", p.name);
  printf("Quantity on hand: %d\n", p.on_hand);
}
```

- A call of `print_part`

```
print_part(part1);
```



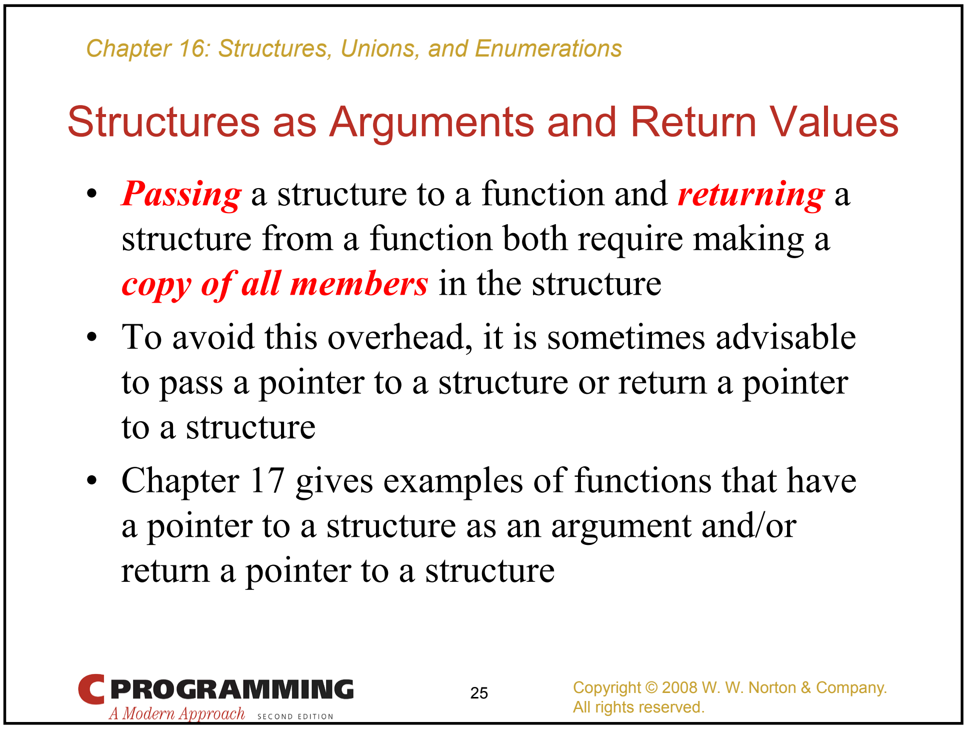
Structures as Arguments and Return Values

- A function that returns a `part` structure

```
struct part build_part(int number,
                       const char *name,
                       int on_hand)
{
  struct part p;

  p.number = number;
  strcpy(p.name, name);
  p.on_hand = on_hand;
  return p;
}
```

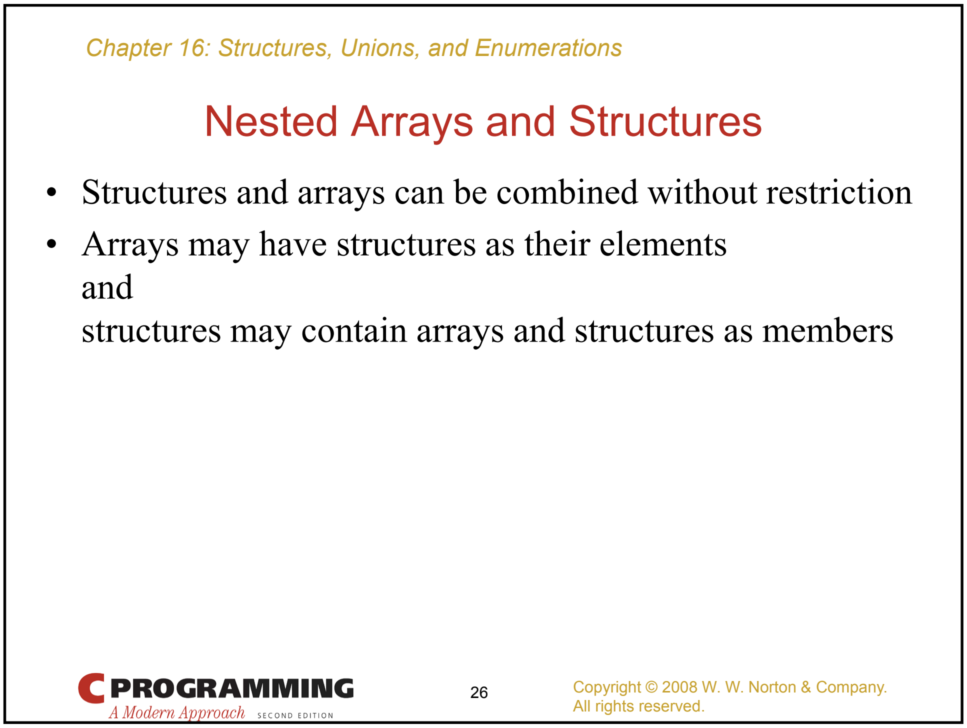
- A call of `build_part`

```
part1 = build_part(528, "Disk drive", 10);
```



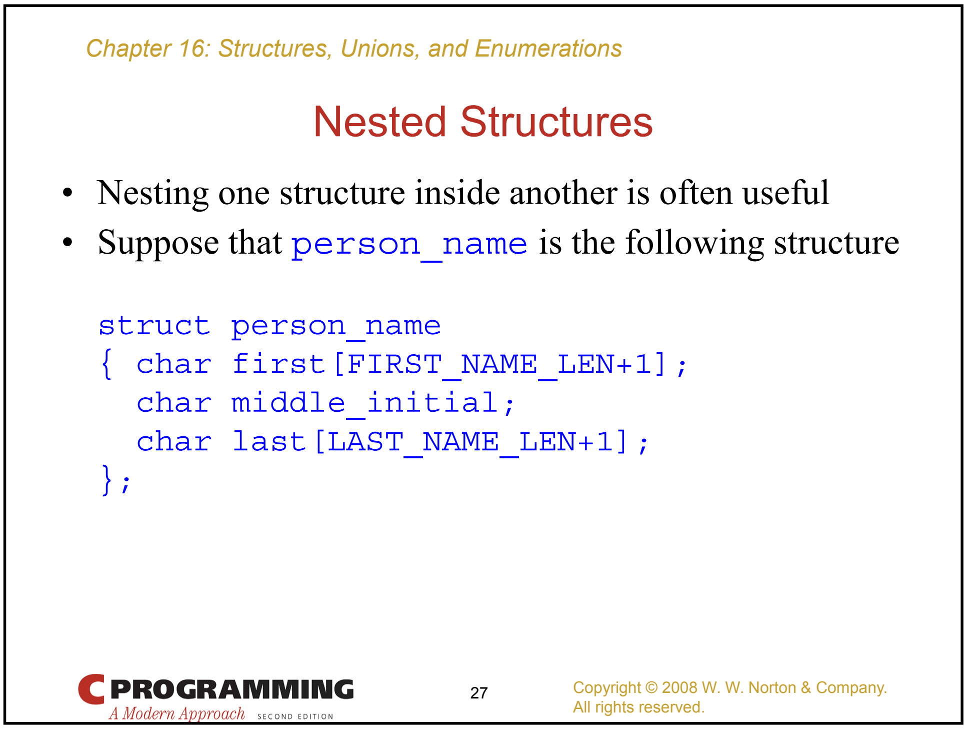
Structures as Arguments and Return Values

- ***Passing*** a structure to a function and ***returning*** a structure from a function both require making a ***copy of all members*** in the structure
- To avoid this overhead, it is sometimes advisable to pass a pointer to a structure or return a pointer to a structure
- Chapter 17 gives examples of functions that have a pointer to a structure as an argument and/or return a pointer to a structure



Nested Arrays and Structures

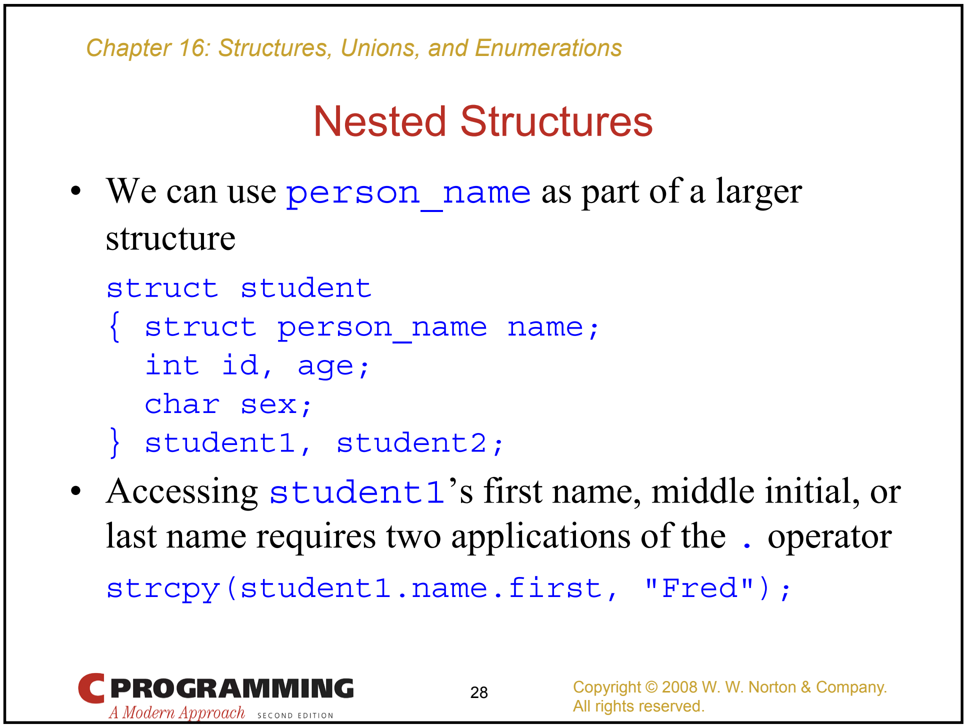
- Structures and arrays can be combined without restriction
- Arrays may have structures as their elements
  and
  structures may contain arrays and structures as members



Nested Structures

- Nesting one structure inside another is often useful
- Suppose that `person_name` is the following structure

```
struct person_name
{ char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
};
```



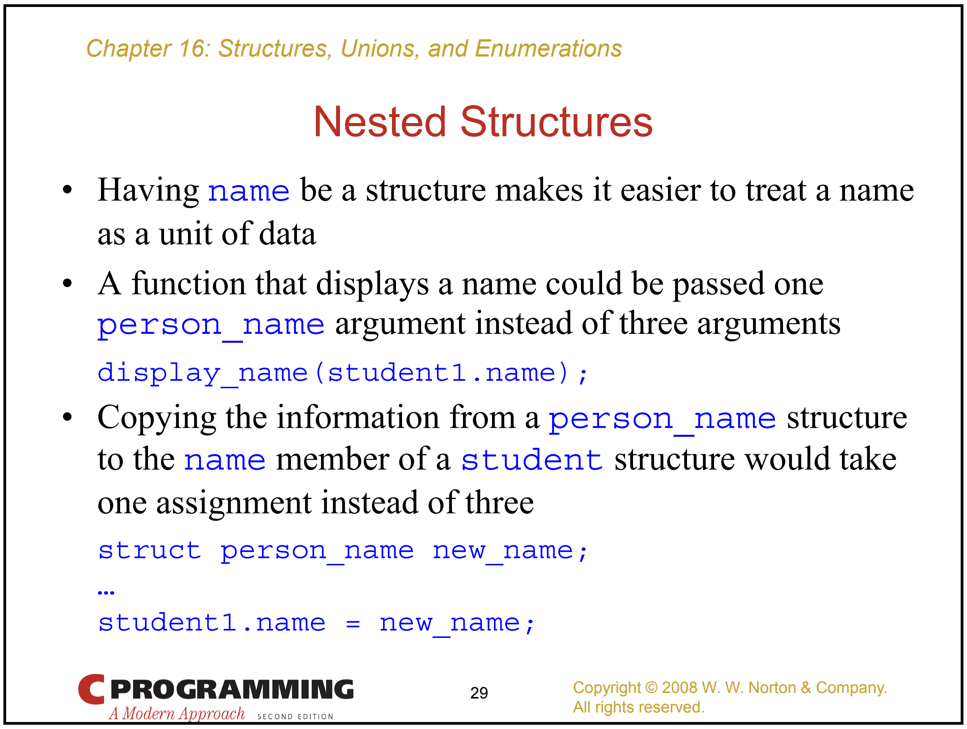
Nested Structures

- We can use `person_name` as part of a larger structure

```
struct student
{ struct person_name name;
  int id, age;
  char sex;
} student1, student2;
```

- Accessing `student1`'s first name, middle initial, or last name requires two applications of the `.` operator

```
strcpy(student1.name.first, "Fred");
```



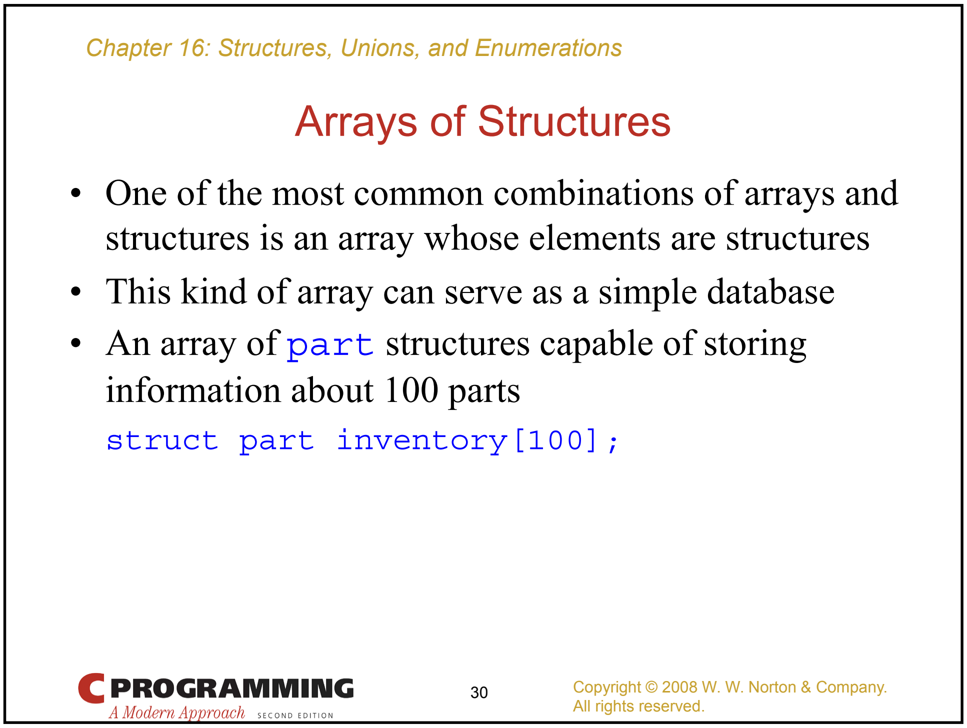
Nested Structures

- Having `name` be a structure makes it easier to treat a name as a unit of data
- A function that displays a name could be passed one `person_name` argument instead of three arguments

```
display_name(student1.name);
```

- Copying the information from a `person_name` structure to the `name` member of a `student` structure would take one assignment instead of three

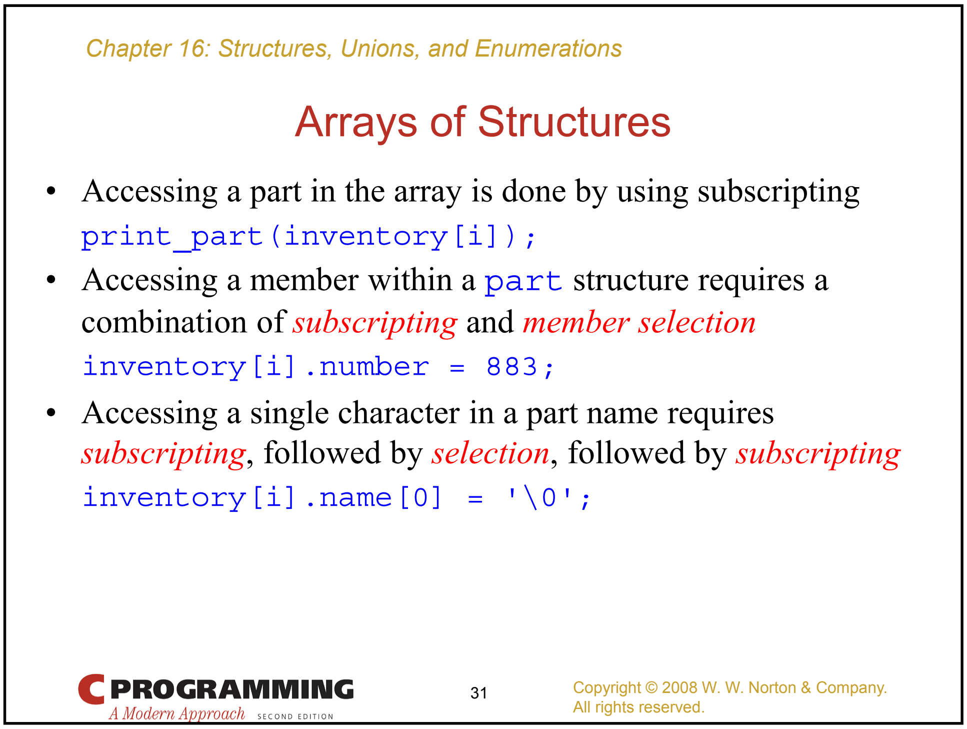
```
struct person_name new_name;
…
student1.name = new_name;
```



Arrays of Structures

- One of the most common combinations of arrays and structures is an array whose elements are structures
- This kind of array can serve as a simple database
- An array of `part` structures capable of storing information about 100 parts

```
struct part inventory[100];
```



Arrays of Structures

- Accessing a part in the array is done by using subscripting

  ```
  print_part(inventory[i]);
  ```

- Accessing a member within a `part` structure requires a combination of *subscripting* and *member selection*

  ```
  inventory[i].number = 883;
  ```

- Accessing a single character in a part name requires *subscripting*, followed by *selection*, followed by *subscripting*

  ```
  inventory[i].name[0] = '\0';
  ```