Records
C++ Structs

Chapter 14

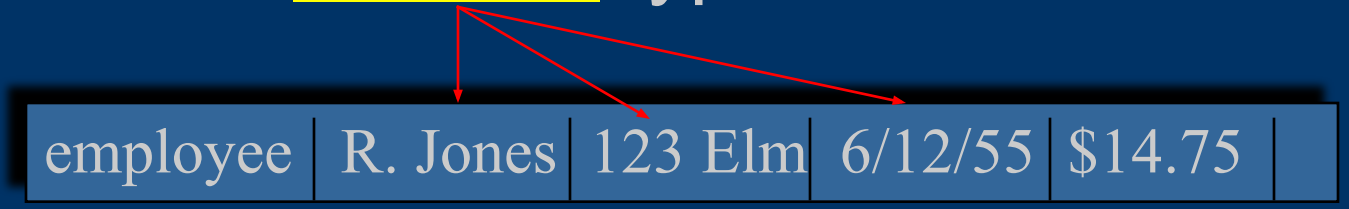
What to do with records?

- Declaring records
- Accessing records
- Accessing the field of a record
- What is a union?
- Can records be in arrays?

Records

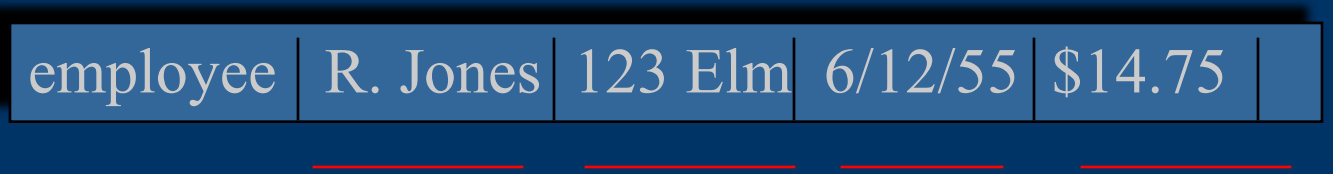
- Recall that elements of arrays must all be of the same type

- In some situations, we wish to group elements of different types



Records

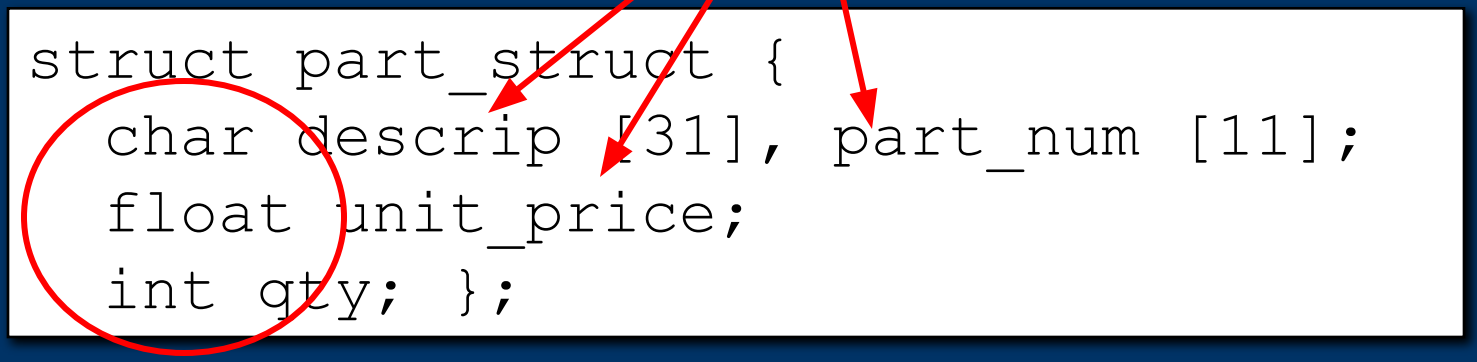
- RECORDS are used to group related components of different types
- Components of the record are called fields



- In C++
  - record called a struct (structure)
  - fields called members

Records

- C++ **struct**
  - structured data type
  - fixed number of components
  - elements accessed by name, not by index
  - components may be of different types



Declaring `struct` Variables

- Given

```
struct part_struct {
   char descrip [31], part_num [11];
   float unit_price;
   int qty; };
```

- Declare :

Use `struct` name as a type.

```
part_struct new_part, old_part;
```

Accessing Components

- Use the name of the record
  the name of the member
  separated by a dot .

```
old_part.qty = 5;
cout << new_part.descrip;
```

- The dot is called the member selector

Aggregate Operations with Structures

- Recall that arrays had none (except reference parameter)
- Structures DO have aggregate operators
  - assignment statement =
  - parameter (value or reference)
  - return a structure as a function type

Aggregate Operations with Structures

- Limitations on aggregate operations
  - no I/O

```
cout << old_part;
cin >> new_part;
```

  - no arithmetic operations

```
old_part = new_part + old_part;
```

  - no comparisons

```
if (old_part < new_part)
    cout << ...;
```

Aggregate Operations with Structures

- `struct` variables must be compared member-wise.
  - To compare the values of `student` and `newStudent`, you must compare them member-wise, as follows:

```
if(student.firstName == newStudent.firstName &&
   student.lastName == newStudent.lastName) ...
```

Input/Output

- There are no aggregate input/output operations on **`struct`**.
  - Data in a **`struct`** variable must be read one member at a time.
  - Contents of a **`struct`** must be written one member at a time.

`struct` Variables and Functions

- A `struct` variable can be passed as a parameter either by value or by reference.
- A function can return a value of the type `struct`
- Note example program fragment

Arrays of Records

- First declare a struct (such as `part_struct`)
- Then specify an array of that type

```
part_struct part_list [50];
```

- Access elements of the array, elements of the struct

How do we print all the descrip fields?

```
for (x = 0; x <50; x++)
  cout << part_list[x].descrip ;
```

Records with Arrays

- Example

```
const int arraySize = 1000;

struct listType
{
   int elements[arraySize];
       //array containing the list
   int listLength;
    //length of the list
}
```

See sample program

Hierarchical Records

- records where at least one of the components is, itself, a record
- Example:

```
struct inventory_struct {
  part_struct part;
  int qty_sold, re_order_qty;
  vendor_struct vendor;  };
```

Choosing Data Structures

- Strive to group logical elements of a structure together
  - calls for hierarchical structures
- Push details of entities down to lower levels of the structure
- Data Abstraction <=> separation of logical peoperties of a data type from its implementation

Testing and Debugging Hints

- Declaration of a **struct** type must end with a semicolon ;
- Be sure to specify the full member selector when referencing a component of a struct variable
  - don't leave out the struct name

Testing and Debugging

- When using an array in a struct, the index goes at the end

```
student_rec.scores[x]
```

- When using an array of struct, the index goes after the struct name

```
parts_list[x].qty
```

Testing and Debugging

- Process struct members separately … the only aggregate operations will be
- Assignment =
- Parameter passing

```
void do_it (part_struct
part);
```

- Function return

```
part_struct blanked_part ( );
```

Testing and Debugging

- Be careful using same member names in different struct types

```
struct parts {
  int qty;
  . . .
} ;
```

```
struct test_scores {
  int qty;
  . . .
} ;
```

- Compiler keeps them separate OK
- **Human** readers can easily confuse them