7. Structures and classes

Topic: Structure types and classes

- Data abstraction: defining new types
- typedef and enum define types
- A structure type describes compound data items
- We may pass structures to functions
- An object models a thing in the real world
- A class is a structure type with operations
- Encapsulation hides data
- Constructors initialize class instances
- Application classes speed development

Simple data abstraction

A programmer may define new data types.

Simplest examples:

- We may define a type identifier as a synonym for another type:
  ```
  typedef float distances;
  distances miles_to_boston = 20.0;
  ```
- We may define an integer type identifier and list its possible values by name:
  ```
  enum seasons {Spr, Sum, Fall, Wint};
  seasons this_season = Wint;
  ```

Structures

A structure is a compound data item whose components may be of any types chosen by the programmer

- Example:
  ```
  struct locations {
  int x, y;
  }
  ```
- Usage:
  ```
  locations loc;
  loc.x = 5;
  loc.y = 10;
  ```

Structure types

- Programmer may define a structure type identifier and use it to declare instances (structure variables)
- Member data items exist in memory only when we declare instances of a structure type.
- We must use instance name, dot, and member name to refer to a member:
  ```
  cout << loc.x;
  ```
- Declare a structure type by listing its name and declaring its members:
  ```
  struct locations { int x, y; };
  ```

Structures associate values

- A structure implements the database concept of record or tuple
- Example:
  ```
  enum teams { RedSox, Yankees, Orioles }; 
  struct games {
   teams home_team, visitor_team;
   int home_score, visitor_score;
  };
  ```
- The example associates elements of a set of teams with each other and with elements of the set of integers
- Instance:
  ```
  games g = {RedSox, Orioles, 3, 2};
  ```

Using a structure type

```
#include <iostream.h>

struct employees {
 char name[40];
 int hours;
};

void main() {
 employees emp = ("Dale", 35);
 cout << emp.name << " worked " << emp.hours << " hours." << endl;
}
```

Output: Dale worked 35 hours.
7. Structures and classes

Passing a structure to a function

```c
struct employees { char name[40]; int hours; };  
void employee_display(employees e);  
void main() {  
    employees emp1 = {"Dale",49}, emp2 = {"Lin",35};  
    employee_display(emp1);  
    employee_display(emp2);  
}  
void employee_display(employees e) {  
    cout << e.name << " worked "  
        << e.hours << " hrs.\n";  
}  
Output: Dale worked 49 hours.  
    Lin worked 35 hours.  
• Function employee_display implements an operation on  
    any instance of the structure type employees.
```

Classes and objects

Class: a compound type defined by data attributes and operations

```c
struct employees {  
    char name[40];  
    int hours;  
    void display();  
};  
Object: an instance of a class
employees emp1,emp2;
```

Using a class

```c
struct employees {  
    char name[40];  
    int hours;  
    void input();  
    void display();  
};  
void main() {  
    employees emp;  
    emp.input();  
    emp.display();  
}  
void employees::input() {  
    cout << "Enter name, hrs: ";  
    cin >> name >> hours;  
}  
void employees::display() {  
    cout << name << " worked "  
    << hours << " hrs.\n";  
}  
Sample I/O:  
Enter name, hrs: Musa 40  
Musa worked 40 hours.
```

Member functions

• A call to a member function must name an  
    object, using dot notation.  
    employees emp;  
    emp.input();  
• Member functions of an object’s class have  
    access to that object’s members.  
    void input() {  
        cin >> name >> hours;  
    }  
• Member function header includes class name  
    and scope resolution operator (::).  
    void employees::input()
```

Object-oriented design

• Any concept is a candidate for a class: persons, things, places, transactions
• Relationships among classes include  
    - containment (an address object is part  
        of a customer object)  
    - inheritance (scrollers and dialogs are two kinds of views)  
• A class implements an abstraction; it may be instantiated by one or more objects

UML class diagrams

```
Class name
  e

Attributes
  D
  y

Methods
  )
)```
Encapsulation hides members

- The keyword `class` may replace `struct`.
- With `class`, members are hidden unless declared `public`.

```cpp
class employees{
    public:
        void main()
        
    void input();
    void display();
    
    private:
        char name[40];
        int hours;
};
```

Access functions assign and retrieve member data

```cpp
int employees::get_hours() const
{
    return hours;
}
private

void employees::set_name(char nm[]) {
    if (strlen(nm) < 40)
        strcpy(name, nm);
}
```

Accessors and mutators

- An accessor is a member function that does not modify data members
- It is wise to enforce this by putting `const` at end of header
- A mutator is a member function that does modify member data
- In the `employees` class, `get_hours` is an accessor; `set_name` is a mutator

```cpp
interface and implementation

- Interface: public member function declarations
- Implementation: private members and definitions of member functions
- A programmer using a class needs to know only its interface.

Constructors initialize members

```cpp
class employees {

    public:
        employees();
        employees(char nm[], int hrs);
    void input();
    void display();

    private:
        char name[40];
        int hours;
};
```

Constructors

- Execute automatically when an object is declared
- Take name of class:
  ```cpp
  employees::employees()
  ```
- Have no return value or type
- May take parameters:
  ```cpp
  employees emp("Dass", 43);
  ```
- May be overloaded:
  ```cpp
  employees();
  employees(char nm[], int hrs);
  ```
### Application classes
- Used in Windows and Java programming
- A user-interface library defines a general-purpose application class
- Application programmer defines a class that inherits from library class, extends its features
- Application programmer may focus on special purpose of application rather than on user-interface details

### Why program with classes?
- We may model the state and behavior of what we work with: e.g., persons, events, collections, displayed objects
- Classes let us model the interactions found in the environment we work with
- Class libraries may be reused conveniently
- Bookseller examples: books, customers, transactions (challenge: define classes for them)