08: STRUCTURED DATA

Programming Technique I

(SCSJ1013)

Abstract Data Types

Abstract Data Types

A data type that specifies:
- values that can be stored
- operations that can be done on the values

User of an abstract data type does not need to know the implementation of the data type, e.g., how the data is stored

ADTs are created by programmers

Abstraction and Data Types

Abstraction: a definition that captures general characteristics without details.

- ◆ Ex: An abstract triangle is a 3-sided polygon. A specific triangle may be scalene, isosceles, or equilateral

Data Type: defines the values that can be stored in a variable and the operations that can be performed on it

Combining Data into Structures

Introduction to Structures

Structure: C++ construct that allows **multiple variables** to be **grouped together**.
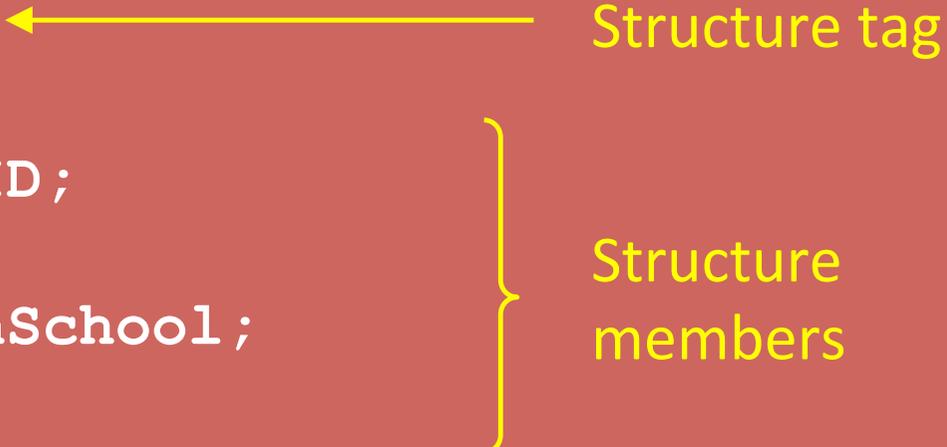
General format:

```
struct <structName>
{
  type1 field1;
  type2 field2;
  . . .
};
```

Example: struct Declaration

```
struct Student                    ← Structure tag
{
    int studentID;
    string name;                    Structure
    short yearInSchool;             members
    double gpa;
};
```

struct Declaration Notes

⚜ Must have **;** after closing **}**.

⚜ struct names commonly begin with **<u>uppercase letter</u>**.

⚜ Multiple fields of same type can be in comma-separated list:
`string name, address;`

⚜ struct declaration **<u>does not</u>** allocate memory or create variables.

⚜ To define variables, use structure tag as type name:
`Student bill;`

Defining Variables

🏵 `struct` declaration does not allocate memory or create variables

🏵 To define variables, use structure tag as type name:

`Student bill;`

```
bill

 studentID  [        ]

 name  [          ]

 yearInSchool [     ]

 gpa  [          ]
```

Accessing Structure Members

Accessing Structure Members

❀ Use the dot (`.`) operator to refer to members of `struct` variables:

```
cin >> stu1.studentID;
  getline(cin, stu1.name);
stu1.gpa = 3.75;
```

❀ General Format: Member variables can be used in any manner appropriate for their data type

Program 11-1

```cpp
1   // This program demonstrates the use of structures.
2   #include <iostream>
3   #include <iomanip>
4   using namespace std;
5
6   const int SIZE = 25; // Array size
7
8   struct PayRoll
9   {
10      int empNumber;      // Employee number
11      char name[SIZE];    // Employee's name
12      double hours;       // Hours worked
13      double payRate;     // Hourly payRate
14      double grossPay;    // Gross Pay
15  };
16
```

```cpp
17   int main()
18   {
19       PayRoll employee; // employee is a PayRoll structure.
20
21       // Get the employee's number.
22       cout << "Enter the employee's number: ";
23       cin >> employee.empNumber;
24
25       // Get the employee's name.
26       cout << "Enter the employee's name: ";
27       cin.ignore();   // To skip the remaining '\n' character
28       cin.getline(employee.name, SIZE);
29
30       // Get the hours worked by the employee.
31       cout << "How many hours did the employee work? ";
32       cin >> employee.hours;
33
34       // Get the employee's hourly pay rate.
35       cout << "What is the employee's hourly payRate? ";
36       cin >> employee.payRate;
37
38       // Calculate the employee's gross pay.
39       employee.grossPay = employee.hours * employee.payRate;
40
```

Program 11-1 *(continued)*

```
41      // Display the employee data.
42      cout << "Here is the employee's payroll data:\n";
43      cout << "name: " << employee.name << endl;
44      cout << "Number: " << employee.empNumber << endl;
45      cout << "hours worked: " << employee.hours << endl;
46      cout << "Hourly payRate: " << employee.payRate << endl;
47      cout << fixed << showpoint << setprecision(2);
48      cout << "Gross Pay: $" << employee.grossPay << endl;
49      return 0;
50  }
```

Program Output with Example Input Shown in Bold

```
Enter the employee's number: 489 [Enter]
Enter the employee's name: Jill Smith [Enter]
How many hours did the employee work? 40 [Enter]
What is the employee's hourly pay rate? 20 [Enter]
Here is the employee's payroll data:
Name: Jill Smith
Number: 489
Hours worked: 40
Hourly pay rate: 20
Gross pay: $800.00
```

Displaying a `struct` Variable

✿ To display the contents of a `struct` variable, must display each field separately, using the dot operator:

```
cout << bill; // won't work
cout << bill.studentID << endl;
cout << bill.name << endl;
cout << bill.yearInSchool;
cout << " " << bill.gpa;
```

Comparing `struct` Variables

Cannot compare struct variables directly:

```
if (bill == william)
// won't work
```

Instead, must compare on a field basis:

```
if (bill.studentID ==
       william.studentID) ...
```

Initializing a Structure

Initializing a Structure

❀ struct variable can be initialized when defined:

    `Student s = {11465, "Joan", 2, 3.75};`

❀ Can also be initialized member-by-member after definition:

    `s.name = "Joan";`

    `s.gpa = 3.75;`

More on Initializing a Structure

❀ May initialize only some members:
```
Student bill = {14579};
```

❀ Cannot skip over members:
```
Student s = {1234, "John", ,
             2.83}; // illegal
```

❀ Cannot initialize in the structure declaration, since this does not allocate memory

```
 9   struct EmployeePay
10   {
11      char name[LENGTH];      // Employee name
12      int empNum;             // Employee number
13      double payRate;         // Hourly pay rate
14      double hours;           // Hours worked
15      double grossPay;        // Gross pay
16   };


20      EmployeePay employee1 = {"Betty Ross", 141, 18.75};
21      EmployeePay employee2 = {"Jill Sandburg", 142, 17.50};
```

Array of Structures

Arrays of Structures

- Structures can be defined in arrays

- Can be used in place of parallel arrays
```
const int NUM_STUDENTS = 20;
Student stuList[NUM_STUDENTS];
```

- Individual structures accessible using subscript notation

- Fields within structures accessible using dot notation:
```
cout << stuList[5].studentID;
```

Program 11-5

```cpp
1    // This program uses an array of structures.
2    #include <iostream>
3    #include <iomanip>
4    using namespace std;
5
6    struct PayInfo
7    {
8       int hours;          // Hours Worked
9       double payRate;     // Hourly Pay Rate
10   };
11
12   int main()
13   {
14      const int NUM_WORKERS = 3;      // Number of workers
15      PayInfo workers[NUM_WORKERS];   // Array of structures
16      int index;                      // Loop counter
17
```

Arrays of Structures- Example

Program 11-5 *(continued)*

```
18      // Get employee pay data.
19      cout << "Enter the hours worked by " << NUM_WORKERS
20          << " employees and their hourly rates.\n";
21
22      for (index = 0; index < NUM_WORKERS; index++)
23      {
24          // Get the hours worked by an employee.
25          cout << "Hours worked by employee #" << (index + 1);
26          cout << ": ";
27          cin >> workers[index].hours;
28
29          // Get the employee's hourly pay rate.
30          cout << "Hourly pay rate for employee #";
31          cout << (index + 1) << ": ";
32          cin >> workers[index].payRate;
33          cout << endl;
34      }
35
36      // Display each employee's gross pay.
37      cout << "Here is the gross pay for each employee:\n";
38      cout << fixed << showpoint << setprecision(2);
39      for (index = 0; index < NUM_WORKERS; index++)
40      {
41          double gross;
42          gross = workers[index].hours * workers[index].payRate;
43          cout << "Employee #" << (index + 1);
44          cout << ": $" << gross << endl;
45      }
46      return 0;
47  }
```

Arrays of Structures- Example

Program Output with Example Input Shown in Bold

Enter the hours worked by 3 employees and their hourly rates.
Hours worked by employee #1: **10 [Enter]**
Hourly pay rate for employee #1: **9.75 [Enter]**

Hours worked by employee #2: **20 [Enter]**
Hourly pay rate for employee #2: **10.00 [Enter]**

Hours worked by employee #3: **40 [Enter]**
Hourly pay rate for employee #3: **20.00 [Enter]**

Here is the gross pay for each employee:
Employee #1: $97.50
Employee #2: $200.00
Employee #3: $800.00

Nested Structures

Nested Structures

A structure can contain another structure as a member:

```
struct PersonInfo
{    string name,
           address,
           city;
};
struct Student
{int studentID;
     PersonInfo pData;
     short yearInSchool;
     double gpa;
};
```

Members of Nested Structures

Use the dot operator multiple times to refer to fields of nested structures:

```
Student s;
s.pData.name = "Joanne";
s.pData.city = "Tulsa";
```

Structures as Function Arguments

Structures as Function Arguments

- May pass members of struct variables to functions:

  ```
  computeGPA(stu.gpa);
  ```

- May pass entire `struct` variables to functions:

  ```
  showData(stu);
  ```

- Can use reference parameter if function needs to modify contents of structure variable

```cpp
 7   const int DESC_SIZE = 50;          // Array size
 8
 9   struct InventoryItem
10   {
11      int partNum;                     // Part number
12      char description[DESC_SIZE];     // Item description
13      int onHand;                      // Units on hand
14      double price;                    // Unit price
15   };


62   void showItem(InventoryItem p)
63   {
64      cout << fixed << showpoint << setprecision(2);
65      cout << "Part Number: " << p.partNum << endl;
66      cout << "Description: " << p.description << endl;
67      cout << "Units On Hand: " << p.onHand << endl;
68      cout << "Price: $" << p.price << endl;
69   }
```

Structures as Function Arguments - Notes

❀ Using value parameter for structure can slow down a program, waste space

❀ Using a reference parameter will speed up program, but function may change data in structure

❀ Using a `const` reference parameter allows read-only access to reference parameter, does not waste space, speed up program

Revised `showItem` Function

```cpp
void showItem(const InventoryItem &p)
{
    cout << fixed << showpoint << setprecision(2);
    cout << "Part Number: " << p.partNum << endl;
    cout << "Description: " << p.description << endl;
    cout << "Units On Hand: " << p.onHand << endl;
    cout << "Price: $" << p.price << endl;
}
```

Returning a Structure from a Function

Returning a Structure from a Function

Function can return a `struct`:

```
Student getStudentData();   // prototype
stu1 = getStudentData();    // call
```

Function must define a local structure
- for internal use
- for use with `return` statement

Returning a Structure from a Function - Example

```
Student getStudentData()

{     Student tempStu;
      cin >> tempStu.studentID;
      getline(cin, tempStu.pData.name);
      getline(cin, tempStu.pData.address);
    getline(cin, tempStu.pData.city);
    cin >> tempStu.yearInSchool;
    cin >> tempStu.gpa;
    return tempStu;
}
```

Program 11-8

```cpp
1   // This program uses a function to return a structure. This
2   // is a modification of Program 11-2.
3   #include <iostream>
4   #include <iomanip>
5   #include <cmath>   // For the pow function
6   using namespace std;
7
8   // Constant for Pi.
9   const double PI = 3.14159;
10
11  // Structure declaration
12  struct Circle
13  {
14      double radius;       // A circle's radius
15      double diameter;     // A circle's diameter
16      double area;         // A circle's area
17  };
18
19  // Function prototype
20  Circle getInfo();
21
22  int main()
23  {
24      Circle c;            // Define a structure variable
```

Program 11-8   (continued)

```
25
26      // Get data about the circle.
27      c = getInfo();
28
29      // Calculate the circle's area.
30      c.area = PI * pow(c.radius, 2.0);
31
32      // Display the circle data.
33      cout << "The radius and area of the circle are:\n";
34      cout << fixed << setprecision(2);
35      cout << "Radius: " << c.radius << endl;
36      cout << "Area: " << c.area << endl;
37      return 0;
38  }
39
```

```
40   //*********************************************************
41   // Definition of function getInfo. This function uses a local   *
42   // variable, tempCircle, which is a circle structure. The user  *
43   // enters the diameter of the circle, which is stored in         *
44   // tempCircle.diameter. The function then calculates the radius *
45   // which is stored in tempCircle.radius. tempCircle is then      *
46   // returned from the function.                                   *
47   //*********************************************************
48
49   Circle getInfo()
50   {
51      Circle tempCircle;   // Temporary structure variable
52
53      // Store circle data in the temporary variable.
54      cout << "Enter the diameter of a circle: ";
55      cin >> tempCircle.diameter;
56      tempCircle.radius = tempCircle.diameter / 2.0;
57
58      // Return the temporary variable.
59      return tempCircle;
60   }
```

Program Output with Example Input Shown in Bold

```
Enter the diameter of a circle: 10 [Enter]
The radius and area of the circle are:
Radius: 5.00
Area: 78.54
```

Pointer to Structure

Pointers to Structures

A structure variable has an address

Pointers to structures are variables that can hold the address of a structure:

```
Student *stuPtr;
```

Can use & operator to assign address:

```
stuPtr = & stu1;
```

Structure pointer can be a function parameter

Accessing Structure Members via Pointer Variables

❀ Must use () to dereference pointer variable, not field within structure:

```
cout << (*stuPtr).studentID;
```

❀ Can use structure pointer operator to eliminate () and use clearer notation:

```
cout << stuPtr->studentID;
```

```cpp
43   void getData(Student *s)
44   {
45       // Get the student name.
46       cout << "Student name: ";
47       cin.getline(s->name, NAME_LENGTH);
48
49       // Get the student ID number.
50       cout << "Student ID Number: ";
51       cin >> s->idNum;
52
53       // Get the credit hours enrolled.
54       cout << "Credit Hours Enrolled: ";
55       cin >> s->creditHours;
56
57       // Get the GPA.
58       cout << "Current GPA: ";
59       cin >> s->gpa;
60   }
```

Unions

Unions

- Similar to a `struct`, but
  - all members share a single memory location, and
  - only one member of the union can be used at a time

- Declared using `union`, otherwise the same as `struct`

- Variables defined as for `struct` variables

Anonymous Union

- A `union` without a union tag:

  `union { ... };`

- Must use `static` if declared outside of a function

- Allocates memory at declaration time

- Can refer to members directly without dot operator

- Uses only one memory location, saves space

Enumerated Data Types

Enumerated Data Types

An enumerated data type is a programmer-defined data type. It consists of values known as *enumerators*, which represent integer constants.

Example:
```
enum Day { MONDAY, TUESDAY,
           WEDNESDAY, THURSDAY,
           FRIDAY };
```

The identifiers `MONDAY, TUESDAY, WEDNESDAY, THURSDAY,` and `FRIDAY,` which are listed inside the braces, are enumerators. They represent the values that belong to the `Day` data type.

Enumerated Data Types

```
enum Day { MONDAY, TUESDAY,
           WEDNESDAY,THURSDAY,
           FRIDAY };
```

Note that the enumerators are not strings, so they aren't enclosed in quotes.

They are **identifiers**.

Enumerated Data Types

⚜ Once you have created an enumerated data type in your program, you can define variables of that type. Example:

```
Day workDay;
```

⚜ This statement defines `workDay` as a variable of the `Day` type.

⚜ We may assign any of the enumerators `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, or `FRIDAY` to a variable of the `Day` type. Example:

```
workDay = WEDNESDAY;
```

Enumerated Data Types

❀ So, what is an *enumerator*?

❀ Think of it as an integer named constant

❀ Internally, the compiler assigns integer values to the enumerators, beginning at 0.

Enumerated Data Types

```
enum Day { MONDAY, TUESDAY,
           WEDNESDAY,THURSDAY,
           FRIDAY };
```

In memory…

```
MONDAY = 0
TUESDAY = 1
WEDNESDAY = 2
THURSDAY = 3
FRIDAY = 4
```

Enumerated Data Types

Using the `Day` declaration, the following code…

```
cout << MONDAY << " "
     << WEDNESDAY << " "
     << FRIDAY << endl;
```

will produce this output:

```
0 2 4
```

Assigning an integer to an enum Variable

🌸 You cannot directly assign an integer value to an `enum` variable. This will not work:
```
workDay = 3;  // Error!
```

🌸 Instead, you must cast the integer:
```
workDay = static_cast<Day>(3);
```

Assigning an Enumerator to an **`int`** Variable

❀ You CAN assign an enumerator to an `int` variable. For example:

```
int x;
x = THURSDAY;
```

❀ This code assigns 3 to `x`.

Comparing Enumerator Values

❀ Enumerator values can be compared using the relational operators. For example, using the `Day` data type the following code will display the message "Friday is greater than Monday."

```
if (FRIDAY > MONDAY)
{
    cout << "Friday is greater "
        << "than Monday.\n";
}
```

Program 11-12

```cpp
1   // This program demonstrates an enumerated data type.
2   #include <iostream>
3   #include <iomanip>
4   using namespace std;
5
6   enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
7
8   int main()
9   {
10      const int NUM_DAYS = 5;      // The number of days
11      double sales[NUM_DAYS];      // To hold sales for each day
12      double total = 0.0;          // Accumulator
13      int index;                   // Loop counter
14
15      // Get the sales for each day.
16      for (index = MONDAY; index <= FRIDAY; index++)
17      {
18          cout << "Enter the sales for day "
19               << index << ": ";
20          cin >> sales[index];
21      }
22
```

```
23        // Calculate the total sales.
24        for (index = MONDAY; index <= FRIDAY; index++)
25            total += sales[index];
26
27        // Display the total.
28        cout << "The total sales are $" << setprecision(2)
29            << fixed << total << endl;
30
31        return 0;
32  }
```

Program Output with Example Input Shown in Bold

```
Enter the sales for day 0: 1525.00 [Enter]
Enter the sales for day 1: 1896.50 [Enter]
Enter the sales for day 2: 1975.63 [Enter]
Enter the sales for day 3: 1678.33 [Enter]
Enter the sales for day 4: 1498.52 [Enter]
The total sales are $8573.98
```

Enumerated Data Types

Program 11-12 shows enumerators used to control a loop:

```cpp
// Get the sales for each day.
for (index = MONDAY; index <= FRIDAY;
index++)
{
    cout << "Enter the sales for day "
            << index << ": ";
      cin >> sales[index];
}
```

Anonymous Enumerated Types

✿ An *anonymous enumerated type* is simply one that does not have a name. For example, in Program 11-13 we could have declared the enumerated type as:

```
enum { MONDAY, TUESDAY,
       WEDNESDAY, THURSDAY,
       FRIDAY };
```

Using Math Operators with `enum` Variables

❀ You can run into problems when trying to perform math operations with `enum` variables. For example:

```
Day day1, day2;

// Define two Day variables.
day1 = TUESDAY;

// Assign TUESDAY to day1.
day2 = day1 + 1;

// ERROR! Will not work!
```

❀ The third statement will not work because the expression `day1 + 1` results in the integer value 2, and you cannot store an int in an `enum` variable

Using Math Operators with enum Variables

❁ You can fix this by using a cast to explicitly convert the result to `Day`, as shown here:

```
// This will work.
day2 = static_cast<Day>(day1 + 1);
```

Using an `enum` Variable to Step through an Array's Elements

❀ Because enumerators are stored in memory as integers, you can use them as array subscripts. For example:

```cpp
enum Day { MONDAY, TUESDAY, WEDNESDAY,
            THURSDAY, FRIDAY };
const int NUM_DAYS = 5;
double sales[NUM_DAYS];
sales[MONDAY] = 1525.0;
sales[TUESDAY] = 1896.5;
sales[WEDNESDAY] = 1975.63;
sales[THURSDAY] = 1678.33;
sales[FRIDAY] = 1498.52;
```

Using an **enum** Variable to Step through an Array's Elements

✿ Remember, though, you cannot use the ++ operator on an `enum` variable. So, the following loop will NOT work.

```
Day workDay;   // Define a Day variable
// ERROR!!! This code will NOT work.
for (workDay = MONDAY; workDay <= FRIDAY; workDay+
   +)⁊
{
    cout << "Enter the sales for day "
        << workDay << ": ";
    cin >> sales[workDay];
}
```

Using an **enum** Variable to Step through an Array's Elements

❀ You must rewrite the loop's update expression using a cast instead of ++:

```cpp
for (workDay = MONDAY; workDay <= FRIDAY;
    workDay = static_cast<Day>(workDay + 1))↲
{
    cout << "Enter the sales for day "
        << workDay << ": ";
    cin >> sales[workDay];
}
```

Program 11-13

```cpp
1    // This program demonstrates an enumerated data type.
2    #include <iostream>
3    #include <iomanip>
4    using namespace std;
5
6    enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
7
8    int main()
9    {
10       const int NUM_DAYS = 5;      // The number of days
11       double sales[NUM_DAYS];      // To hold sales for each day
12       double total = 0.0;          // Accumulator
13       Day workDay;                 // Loop counter
14
```

Program 11-13   *(continued)*

```
15      // Get the sales for each day.
16      for (workDay = MONDAY; workDay <= FRIDAY;
17                           workDay = static_cast<Day>(workDay + 1))
18      {
19         cout << "Enter the sales for day "
20              << workDay << ": ";
21         cin >> sales[workDay];
22      }
23
24      // Calculate the total sales.
25      for (workDay = MONDAY; workDay <= FRIDAY;
26                           workDay = static_cast<Day>(workDay + 1))
27         total += sales[workDay];
28
29      // Display the total.
30      cout << "The total sales are $" << setprecision(2)
31           << fixed << total << endl;
32
33      return 0;
34   }
```

Program Output with Example Input Shown in Bold

```
Enter the sales for day 0: 1525.00 [Enter]
Enter the sales for day 1: 1896.50 [Enter]
Enter the sales for day 2: 1975.63 [Enter]
Enter the sales for day 3: 1678.33 [Enter]
Enter the sales for day 4: 1498.52 [Enter]
The total sales are $8573.98
```

Enumerators Must Be Unique Within the same Scope

✿ Enumerators must be unique within the same scope. For example, an error will result if both of the following enumerated types are declared within the same scope:

```
enum Presidents { MCKINLEY, ROOSEVELT, TAFT };

enum VicePresidents { ROOSEVELT, FAIRBANKS,
                      SHERMAN };
```
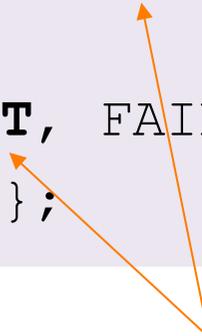
ROOSEVELT is declared twice.

Declaring the Type and Defining the Variables in One Statement

❁ You can declare an enumerated data type and define one or more variables of the type in the same statement. For example:

```
enum Car { PORSCHE, FERRARI, JAGUAR } sportsCar;
```

This code declares the `Car` data type and defines a variable named

```
sportsCar.
```