Chapter 11: Structure Data

11.1 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

• **Structure**: C++ construct that allows multiple variables to be grouped together

• General Format:

```c++
struct <structName>
{
    type1 field1;
    type2 field2;
    ...
};
```
Example **struct** Declaration

```plaintext
struct Student
{
    int studentID;
    string name;
    short yearInSchool;
    double gpa;
};
```

**struct** Declaration Notes

- Must have `;` after closing `}
- **struct** names commonly begin with uppercase letter
- Multiple fields of same type can be in comma-separated list:
  ```plaintext
  string name, address;
  ```
Defining Variables

- `struct` declaration does not allocate memory or create variables
- To define variables, use structure tag as type name:
  ```
  Student bill;
  ```

![Student bill structure](image)

11.3

Accessing Structure Members
Accessing Structure Members

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

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

• Member variables can be used in any manner appropriate for their data type
```cpp
26       cin.ignore();  // To skip the remaining '\n' character
27       getline(cin, employee.name);
28
29       // Get the hours worked by the employee.
30       cout << "How many hours did the employee work? ";
31       cin >> employee.hours;
32
33       // Get the employee's hourly pay rate.
34       cout << "What is the employee's hourly payRate? ";
35       cin >> employee.payRate;
36
37       // Calculate the employee's gross pay.
38       employee.grossPay = employee.hours * employee.payRate;
39
40       // Display the employee data.
41       cout << "Here is the employee's payroll data:
"
42       cout << "Name: " << employee.name << endl;
43       cout << "Number: " << employee.empNumber << endl;
44       cout << "Hours worked: " << employee.hours << endl;
45       cout << "Hourly payRate: " << employee.payRate << endl;
46       cout << "Gross Pay: $" << employee.grossPay << endl;
47       return 0;
48   }
```

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

- Instead, must compare on a field basis:
  ```c++
  if (bill.studentID ==
      william.studentID) ...
  ```
11.4

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

Excerpts From Program 11-3

```c
8 struct EmployeePay
9 {
10   string name; // Employee name
11   int empNum; // Employee number
12   double payRate; // Hourly pay rate
13   double hours; // Hours worked
14   double grossPay; // Gross pay
15 }
```

```c
19   EmployeePay employee1 = {"Betty Ross", 141, 18.75};
20   EmployeePay employee2 = {"Jill Sandburg", 142, 17.50};
```
Arrays of Structures

- Structures can be defined in arrays
- Can be used in place of parallel arrays
  ```c++
  const int NUM_STUDENTS = 20;
  Student stuList[NUM_STUDENTS];
  ```
- Individual structures accessible using subscript notation
- Fields within structures accessible using dot notation:
  ```c++
  cout << stuList[5].studentID;
  ```
Program 11-4

```
// This program uses an array of structures.
#include <iostream>
#include <iomanip>
using namespace std;

struct PayInfo
{
    int hours;    // Hours worked
    double payRate; // Hourly pay rate
};

int main()
{
    const int NUM_WORKERS = 3; // Number of workers
    PayInfo workers[NUM_WORKERS]; // Array of structures
    int index; // Loop counter

    // Get employee pay data.
    cout << "Enter the hours worked by " << NUM_WORKERS
         << " employees and their hourly rates.\n";
    for (index = 0; index < NUM_WORKERS; index++)
    {
        // Get the hours worked by an employee.
        cout << "Hours worked by employee " << (index + 1) << ": " << endl;
        cin >> workers[index].hours;
        // Get the employee's hourly pay rate.
        cout << "Hourly pay rate for employee " << (index + 1) << ": " << endl;
        cin >> workers[index].payRate;
    }
    // Display each employee's gross pay.
    cout << "Here is the gross pay for each employee:\n";
    cout << fixed << showpoint << setprecision(2);
    for (index = 0; index < NUM_WORKERS; index++)
    {
        double gross;
        gross = workers[index].hours * workers[index].payRate;
        cout << "Employee " << (index + 1) << " $ " << gross << endl;
    }
    return 0;
```

11.6

Nested Structures
Nested Structures

A structure can contain another structure as a member:

```c
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:

```c
Student s;
s.pData.name = "Joanne";
s.pData.city = "Tulsa";
```
11.7

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
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
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;
}
```

11.8

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 11.7

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
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 }
Pointers to Structures

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
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;
From Program 11-8

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

11.11

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

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

- Example:

```cpp
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.

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.

Enumerated Data Types

• 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.

```java
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...

```cpp
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:

```cpp
Day workDay = THURSDAY; //OKAY
```

• workDay = 3; // Error!

• Instead, you must cast the integer:

```cpp
workDay = (Day)3;
workDay = static_cast<Day>(3);
```
Assigning an Enumerator to an `int` Variable

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

```c
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."

```c
if (FRIDAY > MONDAY)
{
    cout << "Friday is greater " << "than Monday.\n";
}
```
Program 11-12 (Continued)

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 58573.98
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:

```cpp
enum { MONDAY, TUESDAY,
       WEDNESDAY, THURSDAY,
       FRIDAY };
```
Using Math Operators with \texttt{enum} Variables

- You can run into problems when trying to perform math operations with \texttt{enum} variables. For example:

  ```cpp
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 \texttt{day1 + 1} results in the integer value 2, and you cannot store an \texttt{int} in an \texttt{enum} variable.

  ```cpp
  // This will work.
  day2 = static_cast<Day>(day1 + 1);
  // MONDAY+TUESDAY+WEDNESDAY;
  // Day2 = day1 + (Day)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;
```

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

```cpp
Day workDay;  // Define a Day variable
// ERROR!!! This code will NOT work.
for (workDay = MONDAY; workDay <= FRIDAY; workDay++)
{//workDay = workDay +1;
  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
```
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:

```cpp
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:

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

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

```java
enum Color { RED, ORANGE, YELLOW = 9, GREEN, BLUE};
```

**RED** is 0

**ORANGE** is 1

**YELLOW** is 9

**GREEN** is 10

**BLUE** is 11