Chapter 11: Structured 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

11.2 Combining Data into Structures

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

```c
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:
  ```
  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;
  ```

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;
  ```
- Member variables can be used in any manner appropriate for their data type

Program 11-1

```c
1 // This program demonstrates the use of structures.
2 #include <iostream>
3 #include <string>
4 #include <iomanip>
5 using namespace std;
6
7 struct Payroll
8 {
9    int empNumber; // Employee number
10    string name; // Employee's name
11    double hours; // Hours worked
12    double payRate; // Hourly pay rate
13    double grossPay; // Gross pay
14    
15};
16
17 struct employee; // employee is a Payroll structure.
18
19 // get the employee's number.
20 cout << "Enter the employee's number: ";
21 cin >> employee.empNumber;
22
23 // get the employee's name.
24 cout << "Enter the employee's name: ";
25```
To display the contents of a struct variable, must display each field separately, using the dot operator:

```
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

- A 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

```cpp
typedef struct EmployeePay
{
  string name; // Employee name
  int empNum; // Employee number
  double payRate; // Hourly pay rate
  double hours; // Hours worked
  double grossPay; // Gross pay
} EmployeePay;

EmployeePay employee1 = ("Beaty Ross", 141, 18.75);
EmployeePay employee2 = ("Till Sandburg", 142, 17.50);
```

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:
  ```cpp
cout << stuList[5].studentID;
```

Program 11-4

```cpp
int main()
{
  Student worklist[NUM_WORKERS]; // Array of structures

  for (int i = 0; i < NUM_WORKERS; i++)
    // Input worker information
    cin >> worklist[i].studentID;

  // Display employee pay data.
  for (int i = 0; i < NUM_WORKERS; i++)
    cout << " Employee Pay Data:
          " << worklist[i].name << " 
          " << worklist[i].studentID << "
          " " Hours Worked: " << worklist[i].hours << " Hours Pay Rate: " << worklist[i].payRate << " Gross Pay: " << worklist[i].grossPay;
  return 0;
}
```
11.6 Nested Structures

A structure can contain another structure as a member:

```cpp
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";
```

11.7 Structures as Function Arguments

- May pass members of `struct` variables to functions:
  ```cpp```
  `computeGPA(stu.gpa);`
  ```cpp```
- May pass entire `struct` variables to functions:
  ```cpp```
  `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

**11.8**

**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

```
Student getStudentData()
{
  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;
}
```
11.9 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;
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:

```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.

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

```cpp
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;
  ```

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.

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

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.”

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

Program 11-12 (Continued)

```cpp
// Calculate the total sales,
for (index = MONDAY; index <= FRIDAY; index++)
    total += sales[index];
// Display the total,
cout << "The total sales are $" << total << endl;
return 0;
```

**Program 11-12 Output with Example Input Showed In Table**

Enter the sales for day 0: 1525.00 [enter]
Enter the sales for day 1: 1806.00 [enter]
Enter the sales for day 2: 1975.63 [enter]
Enter the sales for day 3: 1678.23 [enter]
Enter the sales for day 4: 1496.32 [enter]
The total sales are $8553.00

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 enum Variables

- You can run into problems when trying to perform math operations with 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 `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:

```cpp
// 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
day enum { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
const int NUM_DAYS = 5;
double sales[NUM_DAYS];
sales[MONDAY] = 1525.8;
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.

```cpp
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];
}
```

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
// This program demonstrates an enumerated data type.
#include <iostream>
#include <string>
using namespace std;

int main()
{
    const int NUM_DAYS = 5; // The number of days
    double sales[NUM_DAYS]; // To hold sales for each day
    double total = 0.0; // accumulator
    day workDay; // Loop counter
```

Program 15.13

```cpp
// Not the same for each day.
int workDay = MONDAY; workDay = FRIDAY;
for (workDay = MONDAY; workDay <= FRIDAY; workDay++)
{
    cout << "Enter the sales for day " << workDay << ": ";
    cin >> sales[workDay];
}
```
Enumerators Must Be Unique Within the same Scope

● Enumerators must be unique within the same scope. (Unless strongly typed)
● 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.

Using Strongly Typed `enums` in C++ 11

● In C++ 11, you can use a new type of enum, known as a strongly typed enum
● Allows you to have multiple enumerators in the same scope with the same name

```cpp
class Presidents { MCKINLEY, ROOSEVELT, TAFT };
class VicePresidents { ROOSEVELT, FAIRBANKS, SHERMAN };
```

● Prefix the enumerator with the name of the enum, followed by the :: operator:

```cpp
Presidents prez = Presidents::ROOSEVELT;
VicePresidents vp = VicePresidents::ROOSEVELT;
```

● Use a cast operator to retrieve integer value:

```cpp
int x = static_cast<int>(Presidents::ROOSEVELT);
```

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:

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

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