

Chapter 11:

Structured Data



Addison-Wesley
is an imprint of

PEARSON

Copyright © 2012 Pearson Education, Inc.

11.1



Abstract Data Types (ADT's)

Abstract Data Types

- ❖ A **data type** that specifies
 - values that can be stored
 - **attributes**
 - operations that can be done on the values
 - **behaviors**
- ❖ 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

```
struct <structName> //new type name
{
    type1 field1;
    type2 field2;
    . . .
};
```

Example **struct** Declaration

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

struct Declaration Notes

- ❖ Must have ; after closing }
- ❖ struct names commonly begin with an 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;
```

bill

studentID	<input type="text"/>
name	<input type="text"/>
yearInSchool	<input type="text"/>
gpa	<input type="text"/>

11.3



Accessing Structure Members

Accessing Structure Members

- ❖ Use the **dot (.) operator** (member access operator) to refer to a member of a **struct** variable:

```
Student stu1;
```

```
...
```

```
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

```
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 payRate
13     double grossPay;  // Gross pay
14 };
15
16 int main()
17 {
18     PayRoll employee; // employee is a PayRoll structure.
19
20     // Get the employee's number.
21     cout << "Enter the employee's number: ";
22     cin >> employee.empNumber;
23
24     // Get the employee's name.
25     cout << "Enter the employee's name: ";
```

```
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:\n";
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 << fixed << showpoint << setprecision(2);
47     cout << "Gross Pay: $" << employee.grossPay << endl;
48     return 0;
49 }
```

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

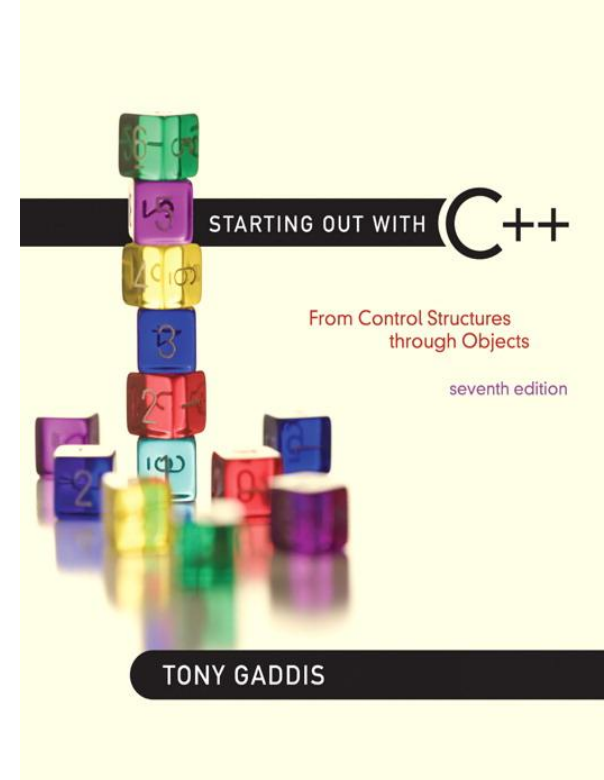
- ❖ You cannot compare **struct** variables directly:

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

- ❖ Instead, must compare on a field basis:

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


11.4



Initializing a Structure

Initializing a Structure

- ❖ a **struct** variable can be initialized when it is defined (similar to an array)

```
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

- ❖ You may initialize only some members (partial initialization)

```
Student bill = {14579};
```

- ❖ You cannot skip over members:

```
Student s = {1234, "John", , 2.83}; //illegal
```

- ❖ You cannot initialize variables in the structure declaration, since no memory is allocated.

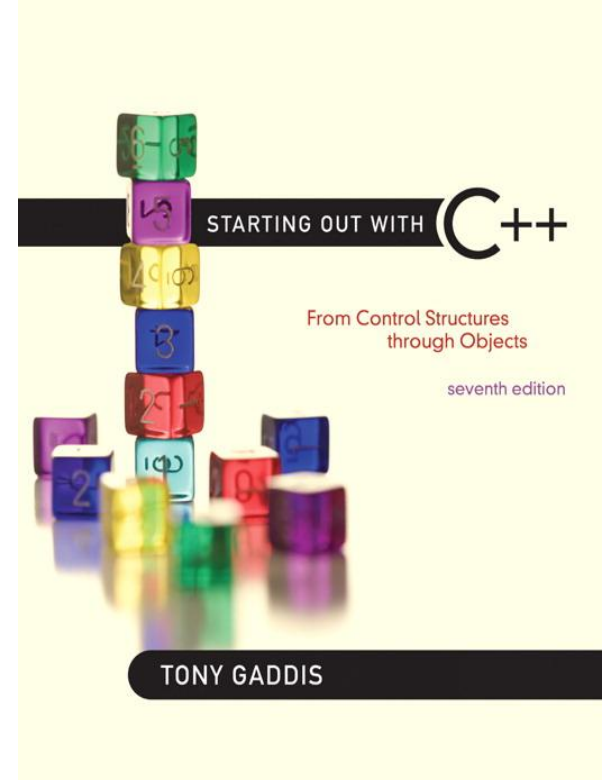
Excerpts From Program 11-3

```
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 };

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

11.5

Array of Structures



Array of Structures

- ❖ Structures can be defined in arrays
- ❖ An array of structures can be used in place of parallel arrays

```
const int NUM_STUDENTS = 20;  
Student stuList[NUM_STUDENTS];
```

- ❖ An individual structure is accessible using subscript notation
- ❖ Fields within structures accessible using dot notation:

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

Program 11-4

```
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
```

```

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 }

```


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

11.6

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";
```

11.7



Structures as Function Arguments

Structures as Function Arguments

- ❖ `struct` variables may be passed to functions:

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

- ❖ An entire `struct` variable may be passed to function

```
showData(stu);
```

- ❖ Structures are passed by value by default
- ❖ You can use a reference parameter if the function needs to modify the contents of structure variable

Excerpts from Program 11-6

```
8  struct InventoryItem
9  {
10     int partNum;           // Part number
11     string description;    // Item description
12     int onHand;           // Units on hand
13     double price;         // Unit price
14 };

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

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

```
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

❖ A function can return a **struct**:

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

❖ The function must define a local structure

- for internal use
- for use with the **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 }
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

```

11.9



Pointers to Structures

Pointers to Structures

- ❖ A structure variable has an address
- ❖ A pointer to a structure is a variable that can hold the address of a structure:

```
Student *stuPtr;
```

- ❖ Can use & operator to assign address:

```
stuPtr = &stu1;
```

- ❖ A structure pointer can be a function parameter

Accessing Structure Members via Pointer Variables

- ❖ You must use **()** to dereference a pointer variable, not a field within the structure:

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

```
cout << *stuPtr.studentID;    //incorrect
```

- ❖ You can use a **structure pointer operator** (aka arrow operator) to eliminate **()** and use clearer notation:

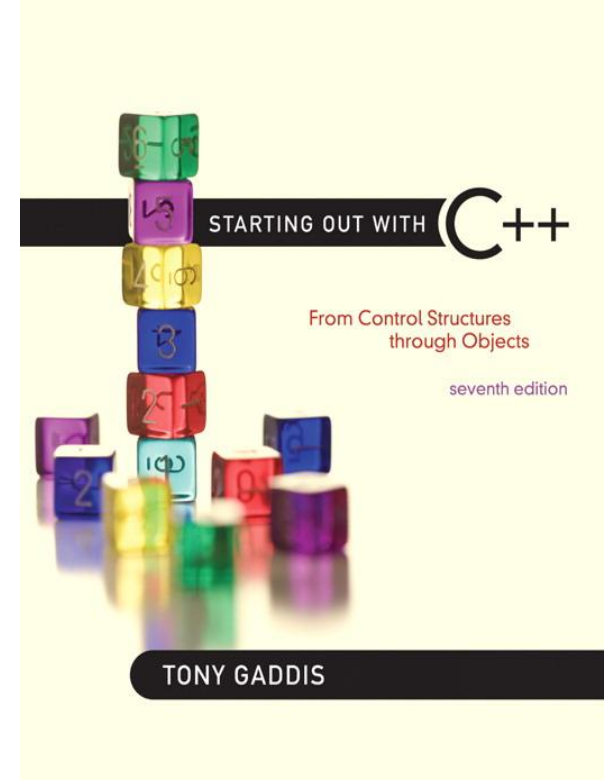
```
cout << stuPtr -> studentID;
```

From Program 11-8

```
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**
 - 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**
- ❖ Member variables are defined as in a **struct** declaration

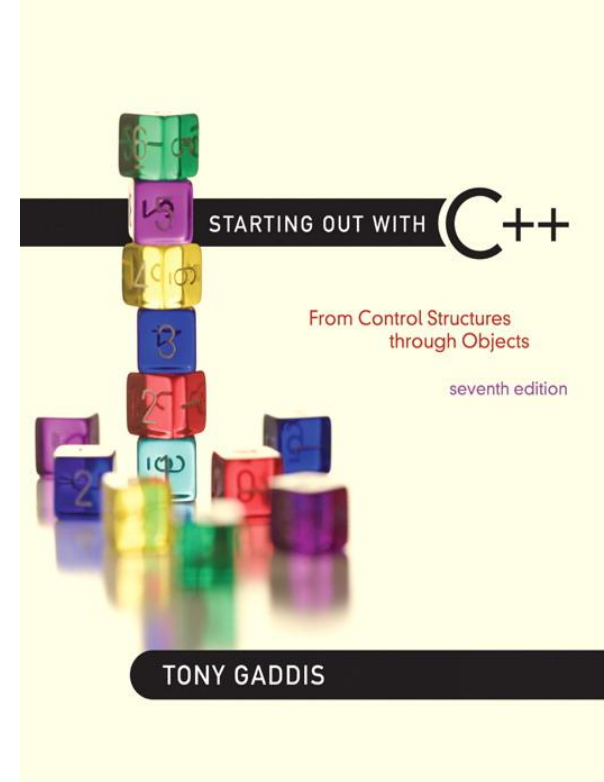
Anonymous Union

- ❖ A **union** without a union tag:

```
union { char c; int i; double d; };
```

- ❖ You must use **static** if it is declared outside of a function
- ❖ **Allocates memory at declaration time**
- ❖ You can refer to the members directly (don't need to use a dot operator). In example above you could directly refer to c, i, and d.
- ❖ **Uses only one memory location, saves space**

11.12



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**.

Enumerated Data Types

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.

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.

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 to a `DAY` type:

```
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; //integer constant 3
```

- ❖ 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

```
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
```

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 $8573.98
```

Enumerated Data Types

- ❖ Program 11-12 shows enumerators used to control a loop. `index` is an `int` variable:

```
// 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:

```
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 (as a loop control variable)

- ❖ 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 ++:

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

Program 11-13

```
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`.