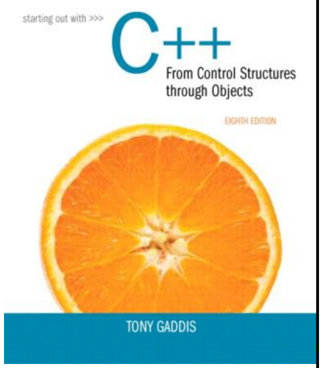
Chapter 11:

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:

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

Example `struct` Declaration

```
struct Student          ← structure tag
{
   int studentID;       ← structure members
   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;
```

bill

studentID	
name	
yearInSchool	
gpa	

11.3

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;
```

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

Program 11-1

```
// This program demonstrates the use of structures.
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;

struct PayRoll
{
   int empNumber;      // Employee number
   string name;        // Employee's name
   double hours;       // Hours worked
   double payRate;     // Hourly payRate
   double grossPay;    // Gross pay
};

int main()
{
   PayRoll employee; // employee is a PayRoll structure.

   // Get the employee's number.
   cout << "Enter the employee's number: ";
   cin >> employee.empNumber;

   // Get the employee's name.
   cout << "Enter the employee's name: ";
```

```
    cin.ignore();  // To skip the remaining '\n' character
    getline(cin, employee.name);

    // Get the hours worked by the employee.
    cout << "How many hours did the employee work? ";
    cin >> employee.hours;

    // Get the employee's hourly pay rate.
    cout << "What is the employee's hourly payRate? ";
    cin >> employee.payRate;

    // Calculate the employee's gross pay.
    employee.grossPay = employee.hours * employee.payRate;

    // Display the employee data.
    cout << "Here is the employee's payroll data:\n";
    cout << "Name: " << employee.name << endl;
    cout << "Number: " << employee.empNumber << endl;
    cout << "Hours worked: " << employee.hours << endl;
    cout << "Hourly payRate: " << employee.payRate << endl;
    cout << fixed << showpoint << setprecision(2);
    cout << "Gross Pay: $" << employee.grossPay << endl;
    return 0;
}
```

Addison-Wesley is an imprint of PEARSON

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
```

Addison-Wesley is an imprint of PEARSON

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;
```

Addison-Wesley is an imprint of PEARSON

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) ...
```

Addison-Wesley is an imprint of PEARSON

11.4

Initializing a Structure

Addison-Wesley is an imprint of PEARSON

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;
```

Addison-Wesley is an imprint of PEARSON

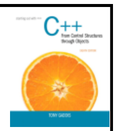
More on Initializing a Structure

- 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

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

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

11.5

Arrays 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 11-4

```
// 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);
      cout << ": ";
      cin >> workers[index].hours;

      // Get the employee's hourly pay rate.
      cout << "Hourly pay rate for employee #";
      cout << (index + 1) << ": ";
      cin >> workers[index].payRate;
      cout << endl;
   }

   // 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);
      cout << ": $" << gross << endl;
   }
   return 0;
}
```

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

- 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

Excerpts from Program 11-6

```
struct InventoryItem
{
   int partNum;                  // Part number
   string description;           // Item description
   int onHand;                   // Units on hand
   double price;                 // Unit price
};

void showItem(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;
}
```


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

- 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

```
// This program uses a function to return a structure. This
// is a modification of Program 11-2.
#include <iostream>
#include <iomanip>
#include <cmath>  // For the pow function
using namespace std;

// Constant for pi.
const double PI = 3.14159;

// Structure declaration
struct Circle
{
   double radius;      // A circle's radius
   double diameter;    // A circle's diameter
   double area;        // A circle's area
};

// Function prototype
Circle getInfo();

int main()
{
   Circle c;       // Define a structure variable
```



```

   // Get data about the circle.
   c = getInfo();

   // Calculate the circle's area.
   c.area = PI * pow(c.radius, 2.0);

   // Display the circle data.
   cout << "The radius and area of the circle are:\n";
   cout << fixed << setprecision(2);
   cout << "Radius: " << c.radius << endl;
   cout << "Area: " << c.area << endl;
   return 0;
}

```



```
//*****************************************************************
// Definition of function getInfo. This function uses a local    *
// variable, tempCircle, which is a circle structure. The user   *
// enters the diameter of the circle, which is stored in         *
// tempCircle.diameter. The function then calculates the radius  *
// which is stored in tempCircle.radius. tempCircle is then      *
// returned from the function.                                   *
//*****************************************************************

Circle getInfo()
{
   Circle tempCircle;  // Temporary structure variable

   // Store circle data in the temporary variable.
   cout << "Enter the diameter of a circle: ";
   cin >> tempCircle.diameter;
   tempCircle.radius = tempCircle.diameter / 2.0;

   // Return the temporary variable.
   return tempCircle;
}
```

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
- 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

```
void getData(Student *s)
{
   // Get the student name.
   cout << "Student name: ";
   getline(cin, s->name);

   // Get the student ID number.
   cout << "Student ID Number: ";
   cin >> s->idNum;

   // Get the credit hours enrolled.
   cout << "Credit Hours Enrolled: ";
   cin >> s->creditHours;

   // Get the GPA.
   cout << "Current GPA: ";
   cin >> s->gpa;
}
```


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

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.

Addison-Wesley is an imprint of PEARSON

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.

Addison-Wesley is an imprint of PEARSON

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.

Addison-Wesley is an imprint of PEARSON

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;
```

Addison-Wesley is an imprint of PEARSON

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.

Addison-Wesley is an imprint of PEARSON

Enumerated Data Types

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

In memory...

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

Addison-Wesley is an imprint of PEARSON

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

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

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

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
   int index;                  // Loop counter

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

```


Program 11-12 (Continued)

```
   // Calculate the total sales.
   for (index = MONDAY; index <= FRIDAY; index++)
      total += sales[index];

   // Display the total.
   cout << "The total sales are $" << setprecision(2)
        << fixed << total << endl;

   return 0;
}
```

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:

```
// 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

- 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

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

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

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 11-13 *(continued)*

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

   // Calculate the total sales.
   for (workDay = MONDAY; workDay <= FRIDAY;
                          workDay = static_cast<Day>(workDay + 1))
      total += sales[workDay];

   // Display the total.
   cout << "The total sales are $" << setprecision(2)
        << fixed << total << endl;

   return 0;
}
```

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. (Unless strongly typed)
- 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.

Using Strongly Typed **`enum`**s 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

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

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

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

- Use a cast operator to retrieve integer value:

```
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:

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

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