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

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

struct Declaration Notes

1. Must have ‘;’ after closing ‘}’
2. struct names commonly begin with uppercase letter
3. Multiple fields of same type can be in comma-separated list:
   string name, address;

Defining Variables

1. struct declaration does not allocate memory or create variables
2. To define variables, use structure tag as type name:
   ```c++
   Student bill;
   ```

Accessing Structure Members

1. Use the dot (.) operator to refer to members of struct variables:
   ```c++
   cin >> stu1.studentID;
   getline(cin, stu1.name);
   stu1.gpa = 3.75;
   ```
2. Member variables can be used in any manner appropriate for their data type
To display the contents of a `struct` variable, must display each field separately, using the dot operator:

```cpp
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:
  ```cpp```
  ```
  if (bill == william) // won't work
  ```
- Instead, must compare on a field basis:
  ```cpp```
  ```
  if (bill.studentID == william.studentID) ...
  ```

**Initializing a Structure**

- A `struct` variable can be initialized when defined:
  ```cpp```
  ```
  Student s = {11465, "Joan", 2, 3.75};
  ```
- Can also be initialized member-by-member after definition:
  ```cpp```
  ```
  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 }
19 EmployeePay employee1 = ("Betty Boss", 141, 14.75);
20 EmployeePay employee2 = ("Bill Sandberg", 142, 17.50);
```
11.6 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:
  ```
  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 << "Unit On Hand: " << p.onHand << endl;
  cout << "Price: $" << p.price << endl;
}
```

11.8

Returning a Structure from a Function

- Function can return a struct:
  ```cpp
  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

```cpp
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;
}
```
Pointers to Structures

- A structure variable has an address
- Pointers to structures are variables that can hold the address of a structure:
  ```c++
  Student *stuPtr;
  ```
- Can use & operator to assign address:
  ```c++
  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:
  ```c++
  cout << (*stuPtr).studentID;
  ```
- Can use structure pointer operator to eliminate () and use clearer notation:
  ```c++
  cout << stuPtr->studentID;
  ```
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:

```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.

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.

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

```cpp
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.
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:

```
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 (Continued)

```cpp
// This program demonstrates an enumerated data type.
enum {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
enum Day (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY);
int main()
{
    enum Day MY_DAY = FRIDAY; // The name of days
    int daily_sales[5]; // To hold sales for each day
    double total = 0.0; // Accumulator
    int index; // Long counter
    // Set the sales for each day
    daily_sales[MONDAY] = 8000; // Monday sales
    daily_sales[TUESDAY] = 9000; // Tuesday sales
    daily_sales[WEDNESDAY] = 10000; // Wednesday sales
    daily_sales[THURSDAY] = 11000; // Thursday sales
    daily_sales[FRIDAY] = 12000; // Friday sales
    // Get the sales for each day
    for (index = 0; index <= FRIDAY; index++)
    {
        daily_sales[index] = 0.0; // Initialize sales
    }
    // Print the sales for each day
    for (index = MONDAY; index <= FRIDAY; index++)
    {
        cout << "The sales for day " << index << " are ":
        cout << daily_sales[index] << "\n";
    }
    // Calculate the total sales.
    for (index = MONDAY; index <= FRIDAY; index++)
    {
        total += daily_sales[index];
    }
    // Display the total.
    cout << "The total sales are $" << setprecision(2)
         << total << "\n";
    // Display the sales.
    for (index = MONDAY; index <= FRIDAY; index++)
    {
        cout << daily_sales[index] << "\n";
    }
    return 0;
}
```

Program Output with Example Input Shown in Bold

Enter the sales for day 1: 1325.69 [Enter]
Enter the sales for day 2: 1986.50 [Enter]
Enter the sales for day 3: 1975.65 [Enter]
Enter the sales for day 4: 1878.31 [Enter]
Enter the sales for day 5: 1408.82 [Enter]
The total sales are $5173.95
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
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.

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

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

```c++
for (workDay = MONDAY; workDay <= FRIDAY; 
     workDay = static_cast<Day>(workDay + 1))
{
    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:

```c++
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

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

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:

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

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