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:
  ```cpp
  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:
  ```c
  string name, address;
  ```

Defining Variables

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

Accessing Structure Members

- Use the dot (.) operator to refer to members of struct variables:
  ```c
  cin >> stu1.studentID;  
  getline(cin, stu1.name);  
  stu1.gpa = 3.75;
  ```
- Member variables can be used in any manner appropriate for their data type

11.3 Accessing Structure Members

Program 11-1

```c
// This program demonstrates the use of structures.
#include <iostream>
#include <iomanip>

int main() {
    struct Payroll {
        int empNumber;  // Employee's number
        string name;  // Employee's name
        double hours;  // Hours worked
        double paycheck;  // Hourly paycheck
        double grossPay;  // Gross pay
    };

    Payroll employee;  // Employee is a Payroll structure.
    // set the employee's number.
    cout << "Enter the employee's number: ";
    cin >> employee.empNumber;
    // set the employee's name.
    cout << "Enter the employee's name: ";
    getline(cin, employee.name);
    // set the employee's hours.
    cout << "Enter the employee's hours: ";
    cin >> employee.hours;
    // set the employee's paycheck.
    employee.paycheck = employee.hours * 8.50;
    // set the employee's gross pay.
    employee.grossPay = employee.paycheck;

    return 0;
}
```
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

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

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:
  ```c++
  Student bill = {14579};
  ```
- Cannot skip over members:
  ```c++
  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++
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 = ("Jim Sanders", 142, 17.50);
```

Arrays of Structures

- Structures can be defined in arrays
- Can be used in place of parallel arrays
- Individual structures accessible using subscript notation
- Fields within structures accessible using dot notation:
  ```c++
  cout << stuList[5].studentID;
  ```

Program 11-4

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

int main() {
  
  const int NUM_WORKERS = 3; // Number of workers
  Paycheck workers[NUM_WORKERS]; // Array of structures
  int index; // Loop counter
  
  // Get the hours worked by each employee
  for (index = 0; index < NUM_WORKERS; index++) {
    cout << "Enter the hours worked by employee ":
    cin >> workers[index].hours;
    cout << "Enter the gross pay for each employee:"
    cin >> workers[index].gross;
    cout << "Enter the pay rate for employee ":
    cin >> workers[index].payRate;
    workers[index].grossPay = workers[index].hours * workers[index].payRate;
    cout << "Employee ":
    cout << "1":
    cout << "2":
    cout << "3":
  }
  
  // Display each employee’s gross pay
  for (index = 0; index < NUM_WORKERS; index++) {
    cout << "Employee ":
    cout << "1":
    cout << "2":
    cout << "3":
  }
  
  // Display the total gross pay for each employee
  for (index = 0; index < NUM_WORKERS; index++) {
    cout "Employee ":
    cout << "1":
    cout << "2":
    cout << "3":
  }
  ```
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:

```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:

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

- May pass entire struct variables to functions:

```c++
showData(stu);
```

- Can use reference parameter if function needs to modify contents of structure variable
### Structures as Function Arguments - Notes

- Using a value parameter for a structure can slow down a program, waste space.
- Using a reference parameter will speed up a program, but the function may change data in structure.
- Using a `const` reference parameter allows read-only access to the reference parameter, does not waste space, speed.

### Revised showItem Function

```cpp
void showItem(const InventoryItem &p)
{
    cout << fixed << showpoint << setprecision(1);
    cout << "Part Number: " << p.partNum << endl;
    cout << "Description: " << p.description << endl;
    cout << "Units On Hand: " << p.onHand << endl;
    cout << "Price: $" << p.price << endl;
}
```

### 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

```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: `Student *stuPtr;`
- Can use & operator to assign address: `stuPtr = &stu;`
- 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 };```

In memory...

<table>
<thead>
<tr>
<th>Enum</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONDAY</td>
<td>0</td>
</tr>
<tr>
<td>TUESDAY</td>
<td>1</td>
</tr>
<tr>
<td>WEDNESDAY</td>
<td>2</td>
</tr>
<tr>
<td>THURSDAY</td>
<td>3</td>
</tr>
<tr>
<td>FRIDAY</td>
<td>4</td>
</tr>
</tbody>
</table>

- Using the Day declaration, the following code...

  ```
  cout << MONDAY << " "
       << WEDNESDAY << " "
       << FRIDAY << endl;
  ```

...will produce this output:

```
0 2 4
```
Comparing Enumerator Values

Enumerators 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
// Get the sales for each day.
for (index = MONDAY; index <= FRIDAY; index++)
{
    cout << "Enter the sales for day " << index << "\n";
    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 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] = 1525.8;
  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 ";
    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 ";
    cin >> sales[workDay];
  }
  ```

Program 11.11

```cpp
1 // This program demonstrates an enumerated data type.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5 enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
6 int main()
7 {
8   const int NUM_DAYS = 5; // The number of days
9   double sales[NUM_DAYS]; // To hold sales for each day
10   double total = 0.0; // Accumulates
11   Day workDay; // Loop counter
12   // Loop through a sales array
13   for (workDay = MONDAY; workDay <= FRIDAY; workDay++)
14   {
15     cout << "Enter the sales for day " << workDay << " : ";
16     cin >> sales[workDay];
17     total += sales[workDay];
18   }
19   cout << "The total sales were: \$" << fixed << setprecision(2) << total << " (\$" << total/5.0 << " per day)";
20   return 0;
21 }
```
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
- Prefix the enumerator with the name of the enum, followed by the :: operator:
  ```cpp
ten class Presidents { MCKINLEY, ROOSEVELT, TAFT };
ten class VicePresidents { ROOSEVELT, FAIRBANKS, SHERMAN };
ten Prefix the enumerator with the name of the enum,
  followed by the :: operator:
  ```cpp
  Presidents prez = Presidents::ROOSEVELT;
  VicePresidents vp = VicePresidents::ROOSEVELT;
  ```cpp
- 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.