Module 4

User-Defined Data Types
Enumeration data type
Objectives

• In this chapter, you will:
  – create and manipulate your own simple data type called the enumeration type
  – learn the use of `typedef` definition
Enumeration Type

- **Data type**: a set of values with a set of operations on them
- **Enumeration type**: a simple data type created by the programmer
- To define an enumeration type, you need:
  - A name for the data type
  - A set of values for the data type
  - A set of operations on the values
• You can specify the name and the values, but not the operations
• Syntax:

```cpp
enum typeName {value1, value2, ...};
```

- `value1, value2, ...` are identifiers called enumerators
- List specifies the ordering:

```
value1 < value2 < value3 <...
```
Enumeration Type (cont’d.)

- The enumeration type is an ordered set of values
  - Default value assigned to enumerators starts at 0
- A value used in one enumeration type cannot be used by another in same block
- Same rules apply to enumeration types declared outside of any blocks
EXAMPLE 7-1

The statement:

```cpp
enum colors {BROWN, BLUE, RED, GREEN, YELLOW};
```

defines a new data type called `colors`, and the values belonging to this data type are `BROWN`, `BLUE`, `RED`, `GREEN`, and `YELLOW`.

---

EXAMPLE 7-2

The statement:

```cpp
enum standing {FRESHMAN, SOPHOMORE, JUNIOR, SENIOR};
```

defines `standing` to be an enumeration type. The values belonging to `standing` are `FRESHMAN`, `SOPHOMORE`, `JUNIOR`, and `SENIOR`.
Enumeration Type (cont’d.)

EXAMPLE 7-3

Consider the following statements:

```cpp
enum grades {'A', 'B', 'C', 'D', 'F'}; //illegal enumeration type
enum places {1ST, 2ND, 3RD, 4TH};  //illegal enumeration type
```

EXAMPLE 7-4

Consider the following statements:

```cpp
enum mathStudent {JOHN, BILL, CINDY, LISA, RON};
enum compStudent {SUSAN, CATHY, JOHN, WILLIAM}; //illegal
```

Suppose that these statements are in the same program in the same block. The second enumeration type, `compStudent`, is not allowed because the value JOHN was used in the previous enumeration type `mathStudent`.
Declaring Variables

• Syntax:

```
dataType identifier, identifier, ...;
```

• Example:

```
enum sports {BASKETBALL, FOOTBALL, HOCKEY, BASEBALL, SOCCER, VOLLEYBALL};
```

– Can declare variables such as:

```
sports popularSport, mySport;
```
Assignment

• Values can be stored in enumeration data types:

```c++
popularSport = FOOTBALL;
```

– Stores `FOOTBALL` into `popularSport`
Operations on Enumeration Types

• No arithmetic operations are allowed on enumeration types:

```cpp
mySport = popularSport + 2;  // illegal
popularSport = FOOTBALL + SOCCER;  // illegal
```

• ++ and -- are illegal, too:

```cpp
popularSport++;  // illegal
popularSport--;  // illegal
```

• Solution: use a static cast

```cpp
popularSport = static_cast<sports>(popularSport + 1);
```
Relational Operators

• An enumeration type is an ordered set of values:

  FOOTBALL <= SOCCER is true
  HOCKEY > BASKETBALL is true
  BASEBALL < FOOTBALL is false

• An enumeration type is an integral data type and can be used in loops:

  for (mySport = BASKETBALL; mySport <= SOCCER;
       mySport = static_cast<sports>(mySport + 1))
Input /Output of Enumeration Types

• An enumeration type cannot be input/output (directly)
  – Can input and output indirectly

```cpp
enum courses {ALGEBRA, BASIC, PASCAL, CPP, PHILOSOPHY, ANALYSIS, CHEMISTRY, HISTORY};
courses registered;

switch (registered)
{
    case ALGEBRA:
        cout << "Algebra";
        break;
    case ANALYSIS:
        cout << "Analysis";
        break;
```
Functions and Enumeration Types

- Enumeration types can be passed as parameters to functions either by value or by reference.
- A function can return a value of the enumeration type.
Declaring Variables When Defining the Enumeration Type

• Can declare variables of an enumeration type when you define an enumeration type:

```cpp
enum grades {A, B, C, D, F} courseGrade;
```
Anonymous Data Types

• **Anonymous type**: values are directly specified in the declaration, with no type name

• Example:

```c
enum {BASKETBALL, FOOTBALL, BASEBALL, HOCKEY} mySport;
```
typedef Statement

• **typedef statement**: used to create synonyms or aliases to a data type

• Syntax:

```cpp
typedef existingTypeName newName;
```

• **typedef** does not create any new data types
  – Only creates an alias to an existing data type
Records (struct)
Objectives

In this chapter, you will:

• Learn about records (structs)
• Examine operations on a struct
• Manipulate data using a struct
• Learn about the relationship between a struct and functions
• Discover how arrays are used in a struct
• Create an array of struct items
Records (structs)

• **struct**: collection of a fixed number of components (members), accessed by name
  – Members may be of different types

• Syntax:

```c++
struct structName {
    dataType1 identifier1;
    dataType2 identifier2;
    ...
    dataTypeN identifierN;
};
```
• A `struct` is a definition, not a declaration
  – Must declare a variable of that type to use it

```cpp
struct houseType
{
    string style;
    int numOfBedrooms;
    int numOfBathrooms;
    int numOfCarsGarage;
    int yearBuilt;
    int finishedSquareFootage;
    double price;
    double tax;
};
```

```cpp
    //variable declaration
    houseType newHouse;
```
Records (structs) (cont’d.)

```c++
struct newHouse {
    int style;
    int numOfBedrooms;
    int numOfBathrooms;
    int numOfCarsGarage;
    int yearBuilt;
    int finishedSquareFootage;
    int yearBuilt;
    double price;
    double tax;
};
```

*FIGURE 9-1 struct newHouse*
Accessing struct Members

- Syntax to access a struct member:

  `structVariableName.memberName`

- The dot (.) is called the member access operator
Accessing struct Members (cont’d.)

• To initialize the members of `newStudent`:
  ```
  newStudent.GPA = 0.0;
  newStudent.firstName = "John";
  newStudent.lastName = "Brown";
  ```

![Diagram showing the structure and initialization of `newStudent`]

---

C++ Programming: Program Design Including Data Structures, Sixth Edition
• Value of one `struct` variable can be assigned to another `struct` variable of the same type using an assignment statement

• The statement:

```cpp
student = newStudent;
```

copies the contents of `newStudent` into `student`
Assignment (cont’d.)

• The assignment statement:
  student = newStudent;

  is equivalent to the following statements:
  student.firstName = newStudent.firstName;
  student.lastName = newStudent.lastName;
  student.courseGrade = newStudent.courseGrade;
  student.testScore = newStudent.testScore;
  student.programmingScore = newStudent.programmingScore;
  student.GPA = newStudent.GPA;
Comparison (Relational Operators)

• Compare `struct` variables member-wise
  – No aggregate relational operations allowed

• To compare the values of `student` and `newStudent`:

```cpp
if (student.firstName == newStudent.firstName &&
    student.lastName == newStudent.lastName)
```

...
Input/Output

• No aggregate input/output operations on a struct variable
• Data in a struct variable must be read or written one member at a time
• Example: output newStudent contents

```cpp
cout << newStudent.firstName << " " << newStudent.lastName << " " << newStudent.courseGrade << " " << newStudent.testScore << " " << newStudent.programmingScore << " " << newStudent.GPA << endl;
```
struct Variables and Functions

- A `struct` variable can be passed as a parameter by value or by reference
- A function can return a value of type `struct`  

```cpp
void printStudent(studentType student)
{
    cout << student.firstName << " " << student.lastName
    << " " << student.courseGrade
    << " " << student.testScore
    << " " << student.programmingScore
    << " " << student.GPA << endl;
}
```
Arrays in structs

- Two items are associated with a list:
  - Values (elements)
  - Length of the list
- Define a struct containing both items:

```cpp
const int ARRAY_SIZE = 1000;

struct listType
{
    int listElem[ARRAY_SIZE]; //array containing the list
    int listLength; //length of the list
};
```
structs in Arrays

• Example:

```cpp
struct employeeType
{
    string firstName;
    string lastName;
    int personID;
    string deptID;
    double yearlySalary;
    double monthlySalary;
    double yearToDatePaid;
    double monthlyBonus;
};
```
structs in Arrays (cont’d.)

```c++
employeeType employees[50];
```

![Diagram of an array of employees with fields: firstName, lastName, personID, deptID, yearlySalary, monthlySalary, yearToDatePaid, monthlyBonus.](Arrays structs cont'd. diagram)

**FIGURE 9-7** Array of `employees`
structs within a struct
Records (union)
Unions in C/C++

- C++ built in types by definition can hold one value of that type. Eg int, long double.
- In C++ structs can contain several types and even functions; for example classes
- A union is a little different. A union can hold the values of several types, however, it can only use one value at a time.
Unions in C++

• A unions declaration is similar to a struct.
  
  ```
  union S {
    int x;
    double y
  };
  ```

  Just like a struct, the union above defines a template (tag) for the union S. Inside the {} are the permissible types.

• At any given time however the union can hold one value for ONLY one of its members.
Unions in C/C++

• To declare an instance of the union’s tag we use the following syntax;

```c
S MyUnion;
```

This will create an instance of the union.

To access the members of a union we can use the same operators as you have seen for structs.
Unions

• When you create an instance of a union, all members of the union share the same memory region.

• That is, the largest of all members is what the union consumes in the computer's memory.

• Because of this, we can get some funny results.
Unions

union S {
    int x;
    double y;
};
Unions in C/C++

```c
union Numbers {
    char c;
    int a;
    float d;
};

int main() {
    Numbers MyNum;
    MyNum.c = 'A';
    cout << "A char: " << MyNum.c << endl;
    cout << "A int: " << MyNum.a << endl;
    cout << "A float: " << MyNum.d << endl;
    return 0;
}
```

Results when I compile and execute:

A char: A
A int: 2293969
A float: 3.21397e-39
Unions

- In a Union there is no mechanism that automatically records which member has been set. It is \textit{up to the programmer to remember} which member was set.

```c
union Number {
    char c;
    int i;
    long l;
    double d;
};

struct WhichMember {
    int Which;
    Number MyNumber;
};
```

As you can see we have wrapped the union within a struct. The struct describes an additional member which can be used to keep track of which member in the union is being used.
struct JetType
{
    int MaxPassengers;
    ...;
};

struct HeliType
{
    int LiftCapacity;
    ...;
};

struct PlaneType
{
    double MaxPayload;
    ...;
};

union AircraftType
{
    JetType Jetu;
    HeliType Heliu;
    PlaneType Plane;
};

struct Aircraft
{
    int Type;
    int Speed;
    AircraftType Description;
};
Unions

• In that example we have defined many different kinds of planes.
• We have formed a union called ‘AircraftType’ which combines the planes.
• We then use a struct to help us identify which plane we have been set in the union.
• The end result here is we have saved a considerable chunk of memory.
Let us now look at workshop 4 question