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:**
- 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
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
{
  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;
```

```c
bill
  studentID
  name
  yearInSchool
  gpa
```
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

```
int empNumber; // Employee number
string name; // Employee's name
double hours; // Hours worked
double payRate; // Hourly payRate
double grossPay; // Gross pay
```

Program 11.1

```cpp
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    int main()
16    {
17        PayRoll employee; // employee is a PayRoll structure.
18
19        // Get the employee's number.
20        cout << "Enter the employee's number: ";
21        cin >> employee.empNumber;
22
23        // Get the employee's name.
24        cout << "Enter the employee's name: ";
```
ACCESSING STRUCTURE MEMBERS

```cpp
26 cin.ignore(); // To skip the remaining '\n' character
27 getline(cin, employee.name);
28 // Get the hours worked by the employee.
29 cout << "How many hours did the employee work? ";
30 cin >> employee.hours;
31 // Get the employee's hourly pay rate.
32 cout << "What is the employee's hourly pay rate? ";
33 cin >> employee.payRate;
34 // Calculate the employee's gross pay.
35 employee.grossPay = employee.hours * employee.payRate;
36 // Display the employee data.
37 cout << "Here is the employee's payroll data:\n";
38 cout << "Name: " << employee.name << endl;
39 cout << "Number: " << employee.empNumber << endl;
40 cout << "Hours worked: " << employee.hours << endl;
41 cout << "Hourly payRate: " << employee.payRate << endl;
42 cout << "Gross Pay: $" << employee.grossPay << endl;
43 return 0;
44 }
```

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

- 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:
  ```c
  Student s = {11465, "Joan", 2, 3.75};
  ```

• Can also be initialized member-by-member after definition:
  ```c
  s.name = "Joan";
  s.gpa = 3.75;
  ```

• 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

---

PROGRAM

```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 Ross", 141, 18.75};
20  EmployeePay employee2 = {"Jill Sandburg", 142, 17.50};
```
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
```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    // Structure for employee data
    struct Worker
    {
        int index;
        double hours;
        double payRate;
    } workers[NUM_WORKERS];

    // Get employee pay data.
    for (int index = 0; index < NUM_WORKERS; index++)
    {
        // Get the hours worked by an employee.
        cout << "Hours worked by employee " << (index + 1) << " : ";
        cin >> workers[index].hours;

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

    // Display each employee's gross pay.
    for (int index = 0; index < NUM_WORKERS; index++)
    {
        double gross = workers[index].hours * workers[index].payRate;
        cout << "Employee " << (index + 1) << " : " << 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
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;
};
```

• Use the dot operator multiple times to refer to fields of nested structures:

```c
Student s;
s.pData.name = "Joanne";
s.pData.city = "Tulsa";
```

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

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

```c
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;
}
```
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

```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;
}
```
RETURNING A STRUCTURE FROM A FUNCTION

Program 11.7

```cpp
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 }
```

RETURNING A STRUCTURE FROM A FUNCTION

---

Lecture 11 - Structured Data
RETURNING A STRUCTURE FROM A FUNCTION

```c
40  // Definition of function getInfo. This function uses a local *
41  // variable, tempCircle, which is a circle structure. The user *
42  // enters the diameter of the circle, which is stored in *
43  // tempCircle.diameter. The function then calculates the radius *
44  // which is stored in tempCircle.radius. tempCircle is then *
45  // returned from the function. *
46  //***************************************************************************
47   
48   Circle getInfo()
49   {  
50      // Temporary structure variable  
51      Circle tempCircle;  
52      // Store circle data in the temporary variable.  
53      cout << "Enter the diameter of a circle: ";  
54      cin >> tempCircle.diameter;  
55      tempCircle.radius = tempCircle.diameter / 2.0;  
56      // Return the temporary variable.  
57      return tempCircle;
58   }
```

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

POINTER 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;
  ```

```cpp
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;
}
```
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

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:
  ```
  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.
ENUMERATED DATA TYPES

- 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.

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

In memory...

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

- 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:

```cpp
workDay = 3;  // Error!
```

• Instead, you must cast the integer:

```cpp
workDay = static_cast<Day>(3);
```

• You **CAN** assign an enumerator to an **int** variable. For example:

```cpp
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.”

```cpp
if (FRIDAY > MONDAY) {
    cout << "Friday is greater than Monday.\n";
}
```
ENUMERATED DATA TYPES

Program 11-12

```cpp
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
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:

```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 \texttt{ENUM} VARIABLES

- You can run into problems when trying to perform math operations with \texttt{enum} variables. For example:

  ```c
  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 \texttt{enum} variable.

- You can fix this by using a cast to explicitly convert the result to `Day`, as shown here:

  ```c
  // This will work.
  day2 = static_cast<Day>(day1 + 1);
  ```

### USING AN \texttt{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:

  ```c
  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 `++`:

```cpp
for (workDay = MONDAY; workDay <= FRIDAY;
    workDay = static_cast<Day>(workDay + 1))
{
    cout << "Enter the sales for day "
    << workDay << ": ";
    cin >> sales[workDay];
}
```
USING AN \texttt{ENUM} VARIABLE TO STEP THROUGH AN ARRAY’S ELEMENTS

\textbf{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
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  return 0;
33}

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 $6573.99

Lecture 11 - Structured Data
23
**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:

```c
enum Presidents { MCKINLEY, ROOSEVELT, TAFT };
enum 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`. 