System Design and Programming II

CSCI – 1943

David L. Sylvester, Sr., Professor

Chapter 11

Structured Data

Abstract Data Types

Abstract data types (ADTs) are data types created by the programmer. ADT's have their own range (or domain) of data and their own set of operations that may be performed on them.

The term abstract data type, or ADT, is very important in computer science and is especially significant in object-oriented programming. This chapter introduces you to structures, which is one of C++'s mechanisms for creating abstract data types.

Abstraction

An abstraction is a general model of something. It is a definition that includes only the general characteristics of an object.

Ex: Dog is an abstraction. It defines a general type of animal.

The term captures the essence of what all dogs are without specifying the detailed characteristics of any particular type of dog. A real-life dog is not an abstract. It is concrete.

Data Types

C++ has several primitive data types, or data types that are defined as a basic port of the language.

bool	Int	unsigned long int
char	long int	float
unsigned char	unsigned short int	double
short int	unsigned int;	long double

Each data type in the previous table has its own range of values, such as -32,768 to +32767 for shorts. Data types also define what values a variable may not hold. For example, integer variables may not be used to hold fractional numbers.

All data types in the table allow the following mathematical and relational operators to be used with them:

```
+    -    *    /    >    <    >=    <=    ==    !=
```

Except, integer data types allow operations with the modulus operator (%), which stores remainder of a mathematical expression.

The primitive data types are abstract in the sense that a data type and an object of that data type are not the same thing.

Ex: `int x = 1, y = 2, z = 3;`

x, y, and z are defined. They are three separate instances of the data type int. Each variable has its own characteristics (x is set to 1, y is set to 2, and z is set to 3). Data type int is the abstraction. x, y and z are concrete occurrences.

Abstract Data Types

Abstract data types (ADTs) are data types created by the programmer and are composed of one or more primitives. The programmer decides what values are acceptable for the data type, as well as what operations may be performed on the data type. In many cased, the programmer designs his or her own specialized operation.

For example suppose a program is created to simulate a 12-hour clock. The program could contain three ADTs: (Hours, Minutes, and Seconds.) The range of values for the hours data type would be integer 1 through 12. The range of values for the Minutes and Seconds data types would be 0 through 59. If the Hours object is set to 12 and then incremented, it will then take on the value 1. Likewise if Minutes object or Seconds object is set to 59 and then incremented, it will take on the value 0.

Hours, Minutes, Second objects may be combined to form a **single object**.

Abstract data types often combine several values. (Ex: Clock object)

Combining Data into Structures

C++ allows you to group several variables into a single item known as a structure.

With structures, you are able to create a relationship between items of different data types.

Ex: A payroll system

```
int empNumber;          // employee number
char name[SIZE];        // employee name
double hours;           // hours worked
double payRate;         // hourly pay rate
double grossPay;        // gross Pay
```

All of the variables listed are related because they hold data about the same employee. Their definition statements, though, do not make it clear that they belong together.

To create a relationship between variables, C++ gives you the ability to package them together into a structure.

Before using a structure, it must be declared.

Ex:

```
struct tag
{
        variable declaration;
        // ... more declarations
        //     may follow...
};
```

members

The tag is the name of the structure. The variable declarations that appear inside the braces declare members of the structure.

Sample structure that holds payroll data.

```
const int SIZE = 25;              // Array size
struct PayRoll
{
    int empNumber;                // employee number
    char name[SIZE];              // employee name
    double hours;                 // hours worked
    double payRate;               // hourly pay rate
    double grossPay;              // gross Pay
};
```

This example declares a structure named PayRoll. The structure has five members: empNumber, name, hours, payRate, and grossPay.

- Notice that a semicolon is required after the closing braces,
- Structure name begins with uppercase letter and,
- Members of double could be declared using one statement.

  (i.e. double hour, payRate, grossPay;)

It’s important to know that a structure declaration does not define a variable. It simply tells the compiler what the PayRoll structure is made of. In essence, it creates a new data type named PayRoll.

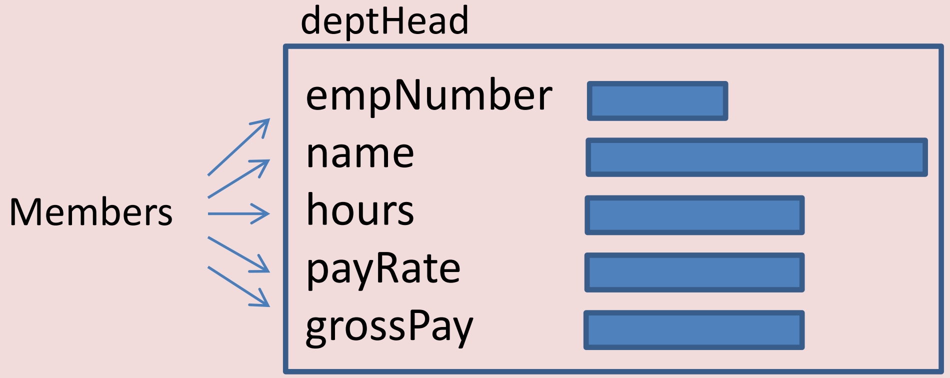
You can define variables of this type with simple definition statements, just as you would with any other data type.

Ex: Payroll deptHead;

The data type of deptHead is the PayRoll structure. The structure tag, PayRoll, is listed before the variable name just as the word int or double would be listed to define variables of those types.

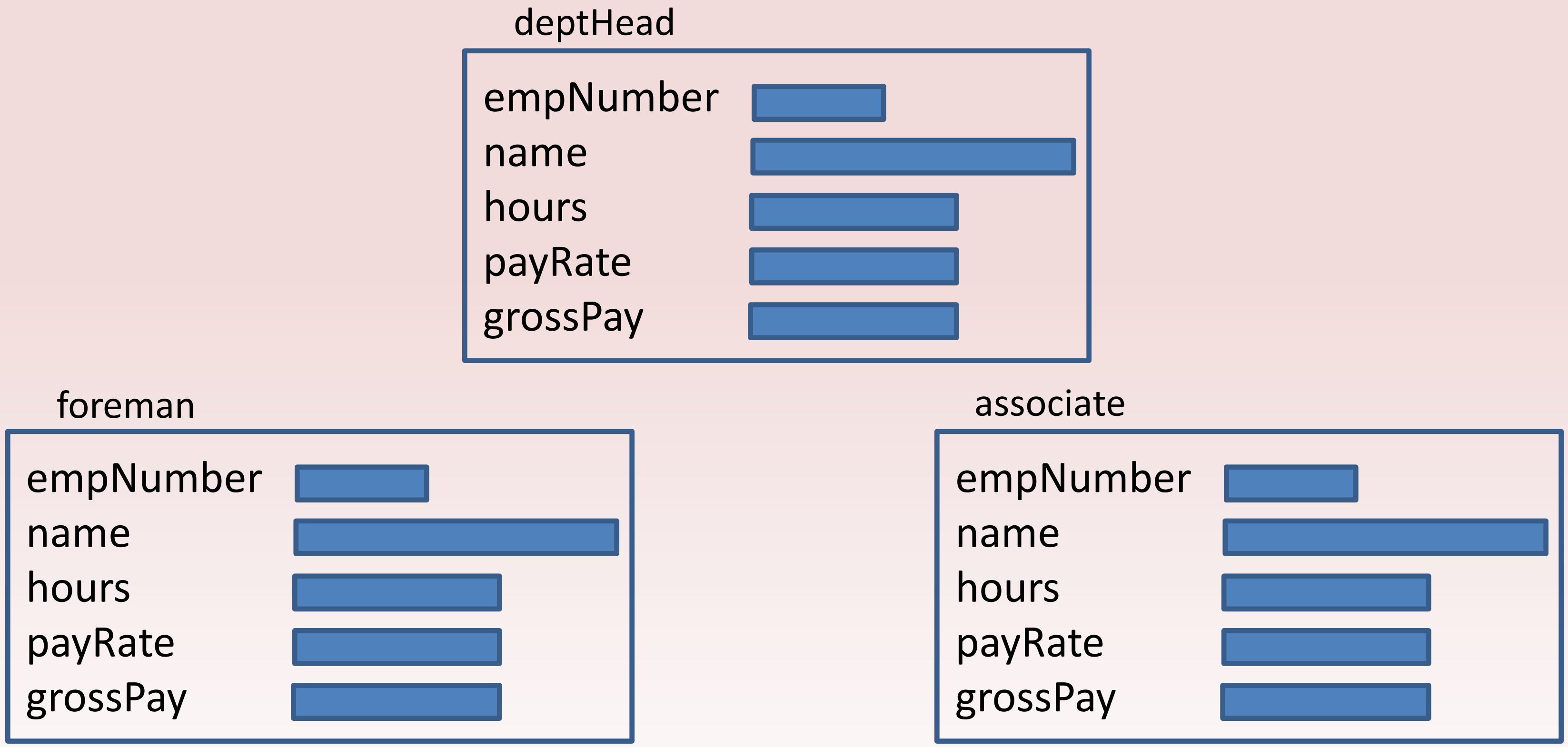
Remember that structure variables are actually made up of other variables know as members. Because deptHead is a PayRoll structure which contains the members: empNumber, name, hours, payRate, and grossPay, having their respective data types.

```
PayRoll deptHead;
```



```
PayRoll deptHead, foreman, associate;
```

Just as it is possible to define multiple int or double variables, it's possible to define multiple structure variables



Although the structure variables are separate, each contains members with the same name.

Sample structures:

```
struct Time
{
	int hour;
	int minutes;
	int seconds;
};
// Structure variable def.
Time now;
```

```
struct Date
{
	int day;
	int month;
	int year;
};
// Structure variable def.
Date today;
```

Accessing Structure Members

The dot operator (.) allows you to access structure members in a program.

Ex:
```
deptHead.empNumber = 475;
```

This statement assigns the number 475 to the empNumber member of deptHead. The dot operator connects the name of the member variable with the name of the structure variable it belongs to. The following statements assign values to the empNumber members of the deptHead, foreman, and associate structure variables:

```
deptHead.empNumber = 745;
foreman.empNumber = 897;
associate.empNumber = 729;
```

With the dot operator you can use member variables just like regular variables.

Display deptHead members:

```
        cout << deptHead.empNumber << endl;
        cout << deptHead.name << endl;
        cout << deptHead.hours << endl;
        cout << deptHead.payRate << endl;
        cout << deptHead.grossPay << endl;
```

Sample Program

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

const int SIZE = 25;            // Array size

struct PayRoll
{
    int empNumber;              // Employee number
    char name[SIZE];            // Employee's name
    double hours;               // Hours worked
    double payRate;             // Hourly pay rate
    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 number.
    cout << "Enter the employee's name: ";
    // removing the annoying new lines
    // stored at the end of the stream
    cin.ignore();// To skip the remaining '\n' characters
    cin.getline(employee.name, SIZE);

    // 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 pay rate? ";
    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 pay rate: " << employee.payRate << endl;
    cout << fixed << showpoint << setprecision(2);
    cout << "Gross pay: $:" << employee.grossPay;
    return 0;
}
```

```
cin.ignore();
```

This statement causes cin to ignore the next character in the input buffer. It is necessary for the cin.getline statement to work properly in the program.

The contents of a structure variable cannot be displayed by passing the entire variable to cout. The following example will not work:

```
cout << employee << endl;     //  Will not work!
```

Instead, each member must be separately passed to cout.

```
cout << employee.name << endl;      // Will work
```

Comparing Structure Variables

```
struct Circle
{
      double radius, diameter, area;
};
```

You cannot perform comparison operations directly on structure variables. Assume that circle1 and circle2 are circles structure variables. The following would produce an error.

```
if (circle1 == circle2)                // Error!
```

In order to compare two structures, you must compare the individual members. Ex:

```
            if (circle1.radius == circle2.radius &&
                circle1.diameter == circle2.diameter &&
                circle1.area == circle2.area)
```

Strings as Structure Members

When a character array is a structure member, you can use the same string manipulation techniques with it as you would with any other character array. Given product.description to be a character array, the following statement copies into it the string “19-inch television”:

```
strcpy(product.description, “19-inch television”);
```

Also, assume that product.partNum is a 15-element character array. The following statement reads into it a line of input:

```
cin.getline(product.partNum, 15);
```

Sample Program

```
// This program uses a structure
// to hold someone's first, middle,
// and last name.
#include <iostream>
#include <cstring>
using namespace std;

// constant for array lengths
const int LENGTH = 15;
const int FULL_LENGTH = 45;

struct Name
{
    char first[LENGTH];         // Holds first name
    char middle[LENGTH];    // Holds middle name
     char last[LENGTH];    // Holds last name
     char full[FULL_LENGTH];// Holds full name
};

int main()
{
    // personName is a Name structure variable
    Name personName;

    // Get the first name.
    cout << "Enter your first name: ";
    cin >> personName.first;

    // Get the middle name.
    cout << "Enter your middle name: ";
    cin >> personName.middle;

    // Get the last name.
    cout << "Enter your last name: ";
    cin >> personName.last;

    // Assemble the full name.
    strcpy(personName.full, personName.first);
    strcat(personName.full, " ");
    strcat(personName.full, personName.middle);
    strcat(personName.full, " ");
    strcat(personName.full, personName.last);

    // Display the full name.
    cout << "\nYour full name is " << personName.full <<
            endl;
    return 0;
}
```

Initializing a Structure

A structure variable may be initialized when it is defined, in a fashion similar to the initialization of an array. Given the following structure declaration:

```
struct CityInfo
{
        char cityName[30];
        char state[3];
        long population;
        int distance;
};
```

A variable may be defined with the following initialization list.

```
CityInfo location = {"Asheville", "NC", 5000, 28};
```

This statement defines the variable location.

You do not have to provide initializers for all the members of a structure variable.

Ex:
```
CityInfo location = {"Tampa"};
```

Notice that the state, population and distance members are uninitialized. The following initialize the cityName and state, while leaving population and distance uninitialized.

```
CityInfo location = {"Atlanta", "GA"};
```

If you leave a structure member uninitialized, you must leave all the members that follow it uninitialized. The following is an example of an invalid initialization of a structure variable.

```
CityInfo location = {"Knoxville", "TN",  , 90};   // illegal!
```

It’s important to note that you cannot initialize a structure member in the declaration of the structure. The following declaration is illegal.

```
// illegal structure declaration
struct CityInfo
{
    char cityName[30] = “Asheville”;            // illegal
    char state[3] = “NC”;                       // illegal
    long population = 5000;                     // illegal
    int distance = 28;                          // illegal
};
```

Remember, that a structure declaration doesn’t actually create the member variable. It only declares what the structure ”looks like.” The member variables are created in memory when a structure variable is defined. Because no variables are created by the structure declaration, there’s nothing that can be initialized there.

Arrays of Structures

As we have seen in Chapter 7, data can be stored in two or more arrays, with a relationship established between the arrays through their subscripts. Because structures can hold several items of varying data types, a single array of structures can be used in place of several arrays of regular variables.

An array of structures is defined like any other array.

Ex:

```
struct BookInfo
{
    char title[50];
    char author[30];
    char publisher[25];
    double price;
};
BookInfo   bookList[20];
```

The statement **BookInfo bookList[20];** defines an array bookList that has 20 elements. Each element is a BookInfo structure.

Each element of the array may be accessed through a subscript. For example, bookList[0] is the first structure in the array, bookList[1] is the second, and so forth.

To access a member of any element, simply place the dot operator (.) and member name after the subscript.

Ex: bookList[5].title

This expression refers to the title member of bookList[5].

To display the data stored in each element of the array structure, you could use a loop.

Ex:

```
for (int index = 0; index < 20; index++)
{
    cout << bookList[index].title << endl;
    cout << bookList[index].author << endl;
    cout << bookList[index].publisher << endl;
    cout << bookList[index].price << endl << endl;
}
```

Because the members title, author, and publisher are also arrays, their individual elements may be accessed as well.

Ex:

```
cout << bookList[10].title[0];
```

This statement will display the character that is the first element of the title member of bookList[10].

Ex: `bookList[2].publisher[3] = ‘t’;`

This statement stores the character ‘t’ in the fourth element of the publisher member of bookList[2].

Sample Program

```
// 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;
     }
```

Structures as Function Arguments

Like other variables, the individual members of a structure variable may be used as function arguments.

Ex:

```
struct Rectangle
{
      double length;
      double width;
      double area;
};
```

Giving the following function definition:

```
double multiply(double x, double y)
{
     return x * y;
}
```

and assuming that box is a variable of the rectangle structure type, the following function will pass box.length into x and box.width into y. The returned value is stored into box.area.

```
box.area = multiply(box.length, box.width);
```

Sometimes it‘s more convenient to pass an entire structure variable into a function instead of individual members. (*using a structure variable as the parameter*)

Ex:

```
void showRect(Rectangle r)
{
      cout << r.length << endl;
      cout << r.width << endl;
      cout << r.area << endl;
}
```

To pass the box variable into r would be done as follows:

```
showRect(box);
```

Inside the function showRect, r’s members contain a copy of box’s members.

Once the function is called, r.length contains a copy of box.length, r.width contains a copy box.width, and r.area contains a copy of box.area. Structures, like all variables, are normally passed by value into a function. If a function is to access the members of the original argument, a reference variable may be used as the parameter.

Structures as Function Arguments

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

const int DESC_SIZE = 50;  // Array size

struct InventoryItem
{
int partNum;  // Part Number
char description[DESC_SIZE];  // Item Description
int onHand;  // Units on hand
double price;  // Unit price
};

// Function Prototypes
void getItem(InventoryItem&);  // Argument passed by reference
void showItem(InventoryItem);  //Argument passed by value

int main()
{
InventoryItem part;

getItem(part);
showItem(part);
return 0;
}

// Definition of function getItem.

void getItem(InventoryItem &p)  // Uses a reference parameter
{
// Get the part number.
cout << "Enter the part number: ";
cin >> p.partNum;

//Get the part description.
cout << "Enter the part description: ";
cin.ignore();
cin.getline(p.description, DESC_SIZE);

// Get the quantity on hand.
cout << "Enter the quantity on hand: ";
cin >> p.onHand;

// Get the unit price.
cout << "Enter the unit price: ";
cin >> p.price;
}

// Definition of function showItem

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;
}
```

Enumerated Data Types

Enumerated data types are programmer-defined data types, consisting of values known as enumerators, which represent integer constants.

Using the enum key word you can create your own data type and specify the values that belong to that type. Such a type is known as an enumerated data type.

Ex:

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

The example declaration creates an enumerated data type named Day. The identifiers MONDAY, TUESDAY, WEDNESDAY, THURSDAY, and FRIDAY are known as enumerators. They represent the values that belong to the day data type.

Note that the enumerators are not enclosed in quotation marks, therefore they are not strings. Enumerators must be legal C++ identifiers.

Once created, variables can be defined.

```
Ex:       Day workDay;
```

Because workDay is a variable of the Day data type, we may assign any of the enumerators previously created.

```
Ex:       workDay = Wednesday;
```

The enumerators can be thought of as integer name constants. The compiler will assign the integer values 0 through x; the number will be assigned in the order the enumerators are listed.

```
Ex:       Monday = 0, Tuesday = 1, Wednesday = 2 …….
```

The C++ statement

```
          cout << Monday << endl << Tuesday << endl << Wednesday
                    << endl << Thursday << endl << Friday;
```

will output   0   1   2   3   4   each number on a separate line.

Enumerators do not have to be in all uppercase letters.

Ex:

```
enum Day { monday, tuesday, wednesday, thursday, friday }
```

Although many programmers prefer to write them in all uppercase letter. (*Simply a programming style.*)

Even though enumerators are integer, you cannot directly assign an integer value to an enum variable. The follow example will produce an error.

```
workDay = 3;                    //  Will produce an error
```

When assigning a value to an enum variable, you should use a valid enumerator. However, if circumstances require that you store an integer value in an enum variable, you can do so by casting the integer.

Ex:

```
workDay = static_cast<Day>3;
```

This statement is equivalent to:

```
workDay = Thursday;
```

Assigning an Enumerator to an int Variable

Although you cannot directly assign an integer value to an enum variable, you can directly assign an enumerator to an integer variable.

Ex:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
int   x;
x = THURSDAY;
cout << x << endl;
```

This will display the number 3.

You can also assign a variable of an enumerated type to an integer variable.

Ex:

```
Day  workDay =  FRIDAY ;
int   x = workDay;
cout << x << endl;
```

This will display the number 4.

Comparing Enumerator Values

Enumerator values can be compared using the relational operators.

Ex:
```
FRIDAY  >  MONDAY
```

The expression is true because the enumerator FRIDAY is stored in memory as 4 and the enumerator MONDAY is stored as 0.

Ex:
```
if (FRIDAY  >  MONDAY)
     cout << "Friday is greater than Monday.\n";
```

You can also compare enumerator values with integer values.

Ex:
```
if (MONDAY  ==  0)
     cout << "Monday is equal to zero.\n";
```