User Defined Data Types

C++ Structures

In this Week

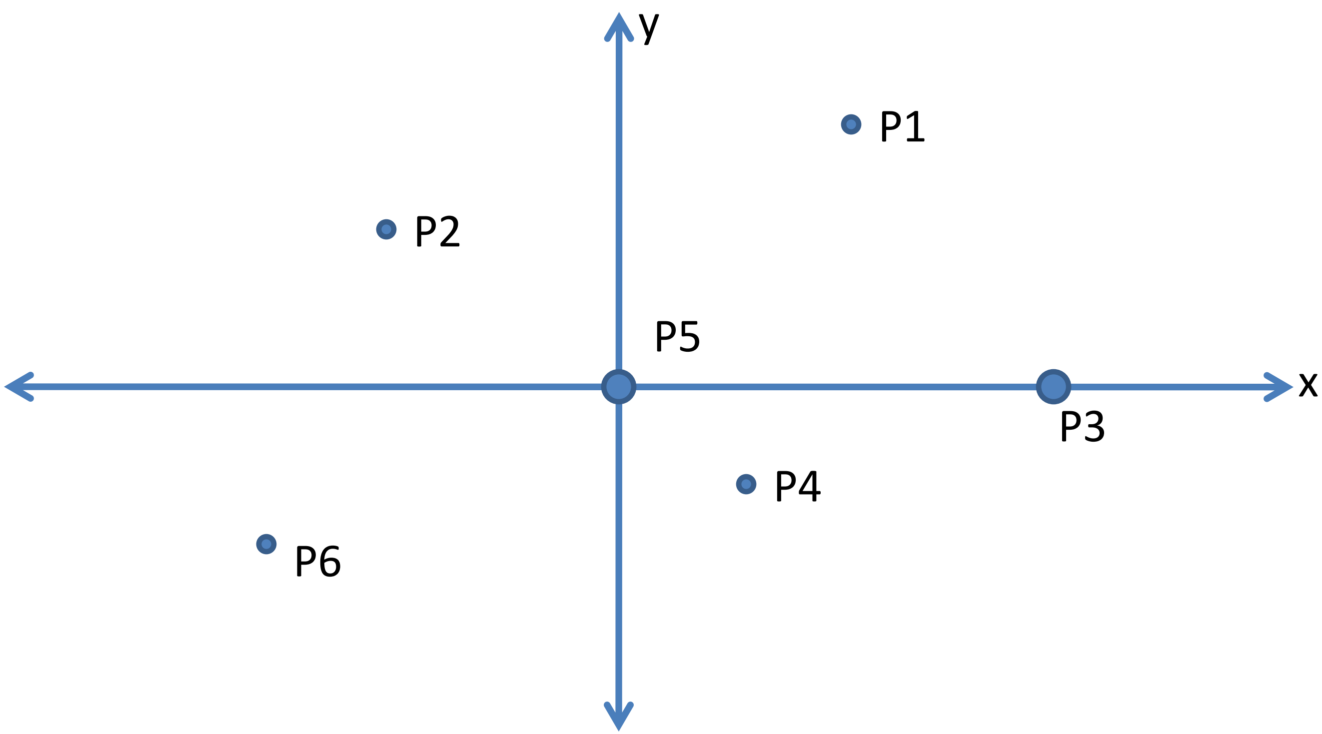
- Motivation: Why Structures?
- Structure Declaration and Definition
- Working with Structure variables
- Pointers and References to Structures
- Passing Structures to functions
- Returning Structures from functions
- Array of Structures

C++ Structures: Motivation

- So far, we have been working with variables that are simple and not related
- Sometimes, we may like to work with variables that are very much related
- For example, consider the problem of working with Points in two dimensional space
- We assume a **Point** has two coordinates (x, y)
- Each of the coordinates is a double data type

C++ Structures: Motivation

- Our aim is to create several Points in our program and work with such Point objects



C++ Structures: Motivation

- How can we implement this?
- Well consider the first Point: We could declare for this Point as follows

double x1, y1;

- A sample program that works for a single Point would look like as follows

C++ Structures: Motivation

```cpp
#include <iostream>
using namespace std;

int main()
{
    //C++ program working with Points in two dimensional space
    //In this program, we will
    //  - Create one Point object,
    //  - Read its x and y coordinates from the user (keyboard)
    //  - Print the Point object to the screen, and finally
    //  - Print the distance of the Point from the origin

    // Step 1: Create the Point object
    double x, y;

    // Step 2: Read the Point object from the user
    cout << "Enter the x coordinate of the Point: ";
    cin >> x;
    cout << "Enter the y coordinate of the Point: ";
    cin >> y;

    // Step 3: Print the Point object to the screen
    cout << "The Point object you created is P(" << x << ", " << y << ")" << endl;

    // Step 4: Print the distance of the Point object from the origin
    double distance = sqrt(x*x + y*y);
    cout << "The distance of the Point from the origin is " << distance << endl;

    system("Pause");
    return 0;
}
```

C++ Structures: Motivation

- Here it should be emphasized that the x and y variables in our program are two **independent** and **not related** double variables
- Mathematically speaking however, our x and y variables are **related** and in fact represent **ONE** Point object
- Changing the value of either x or y for example changes the Point object's location
- So this relation between the x and y variables is not kept in our program

C++ Structures: Motivation

- Now, consider the problem where we would like to work with several Points
- One way of doing this would be to declare several x coordinates and y coordinates as follows

double x1, x2, x3,....;

double y1, y2, y3,.....;

C++ Structures: Motivation

- This is simply too much of a declaration!
- Imagine working with 10 or more Points
- Only the declaration will be too much of code
- Even worse... very much **typo** error prone
- For example you may type x2 by mistake when you are working with the third Point which will result to semantic errors that are hard to find
- Another approach might be to declare arrays for each coordinate in the Point object as follows:

C++ Structures: Motivation

- Using arrays, we may do something like

double X[10];

double Y[10];

- This is much better.... BUT then we have to remember the correspondence between different arrays: For example we need to remember X[5] and Y[5] belong to the same Point and are related
- Once again, this is very much error prone. If we modify one of the arrays with wrong indexing, say for example, all the Point objects will be messed up!
- **All this is.... because related data are stored separately!**

C++ Structures

- The natural question we should ask now is therefore,
  - **Could we combine the x and y coordinates of a Point object into a single variable?**
- For example; how wonderful would it be if we had a data type called **Point** which stores both the x and y coordinates of a Point?
- Then

  Point p1;

  would declare one **Point** object and the x and y coordinates of p1 would be embedded in p1 variable

C++ Structures

- This is exactly what C++ Structures perform!
- In a nutshell, **C++ Structure** help us to store related data together and organize our coding experience
- Just like a function combines related tasks into a single unit; a **C++ Structure** combines related data into a single data type
- Thus a structure is used to define a new user defined data type
- The related data in a structure are called members of the structure. Members must be existing data types such as int, float, string, **or other predefined structures**

Structure Declaration

- In order to define a new user defined data type we first declare a structure (using the keyword **struct**)
- ***Syntax*** **(Structure Declaration)**

```
struct StructureName
{
        data_type member_1;
        data_type member_2;
        ⋮
        data_type member_n;
};
```

Note the semicolon

Structure Declaration

- Therefore the **Point** data type can now be defined as a structure as a follows:

```
struct Point
{
        double x;
        double y;
};
```

- Now, we can declare a variable of type **Point** just like we declare an int or float or string

Structure Declaration

- Creating the new user defined data type **Point** as a structure is known as structure declaration
- Normally, we put the structure declaration at the top of our program outside any function and the main program (similar to global variables)
- This helps to declare **Point** anywhere in our program
- ***TIP:-*** If you declare your structure in the main program, then you will not be able to declare structure variable in any function in the program!

Declaring a Structure Variable

- When we declare a variable of type **Point**, we say we are declaring a structure variable
- ***Syntax*** **(Structure Variable)**

```
Point p1;
```

declares the variable **p1** as **Point** data type

- Now, p1 is just like any other variable... ***The only question is how do we access the members in the structure p1?***

Working with Structure Variables

- We access the members of Structures with the dot operator
- ***Syntax*** **(Accessing Members of Structures)**

  p1.x //accesses the x member variable

- Moreover

  p1.y //accesses the y member variable

- Similarly, any other member variable if there is any

Working with Structure Variables

- We now present a complete implementation of the previous example where we would like to read a point and print it using structure to combine the members of the point in one data type known as Point
- ***TIP:-*** It is a common practise to capitalize the Structure Declaration. Variables of type structure however should follow the standard practise of starting them with a lower case alphabet

Working with Structure Variables

```
#include <iostream>
#include <cmath>
using namespace std;

struct Point
{
    double x;
    double y;
};
int main()
{
    // C++ program working with Points in two dimensional space
    // In this program, we will
    //  - Create one point object,
    //  - Read its x and y coordinates from the user (keyboard),
    //  - Print the point object on to the screen, and finally
    //  - Compute and print the distance of the point object from the origin

    //Step 1. Create the point object
    Point p;
    //Step 2. Read its x and y coordinates from the user (keyboard)
    cout << "Enter the x coordinate of the Point object: ";
    cin >> p.x;
    cout << "Enter the y coordinate of the Point object: ";
    cin >> p.y;
    //Step 3. Print the point object on to the screen
    cout << "The Point object you created is: P(" << p.x << ", " << p.y << ")" << endl;
    //Step 4. Compute and print the distance of the point object from the origin
    double distance = sqrt(p.x*p.x + p.y*p.y);
    cout << "The distance of the Point object from the origin is " << distance << endl;

    system("Pause");
    return 0;
}
```

Working with Structure Variables

- As you can see working with structure variables is just like working with any other data type variable
- All you need is
  - **Create a new data type with a struct keyword,**
  - **Declare a variable of type struct, and**
  - **Work with the variable by accessing the members using a dot operator.**
- That is it!

Pointers and References to Structures

- We can create a reference (alias) to a structure variable just like we create references to int, float, string...
- In the previous example, for example, we could create **p1Alias** as a reference to **p1** and then read the point using the p1 variable and then print the point using the alias
- The following main program demonstrates this

Pointers and References to Structures

```cpp
int main()
{
    //C++ program working with Points in two dimensional space
    //In this program, we will
    //  - Create one Point object,
    //  - Read its x and y coordinates from the user (keyboard)
    //  - Print the Point object to the screen, and finally
    //  - Print the distance of the Point from the origin

    // Step 1: Create the Point object
    Point p1;
    Point &p1Alias = p1;

    // Step 2: Read the Point object from the user
    cout << "Enter the x coordinate of the Point: ";
    cin >> p1.x;
    cout << "Enter the y coordinate of the Point: ";
    cin >> p1.y;
```

Using the p1 variable

```cpp
    // Step 3: Print the Point object to the screen
    cout << "The Point object you created is P(" << p1Alias.x << ", " << p1Alias.y << ")" << endl;
```

Using the alias

```cpp
    // Step 4: Print the distance of the Point object from the origin
    double distance = sqrt(p1Alias.x*p1.x + p1.y*p1Alias.y);
    cout << "The distance of the Point from the origin is " << distance << endl;
```

Mix of both

```cpp
    system("Pause");
    return 0;
}
```

Pointers and References to Structures

- Similarly we could declare a pointer variable of type **Point** and then point it to the p1 variable

```
Point *p1Ptr;
p1Ptr = &p1;
```

- Now, **p1Ptr** is a pointer to p1
- ***CAUTION:-*** **Accessing of member variables using the dot operator does not work for pointer variables!!!**

Access Operator for Pointers to Structures

- Pointers to structures access the member variables with **->** operator
- ***Syntax*** **(Access operator for pointer to struct)**

  pointerToStruct->member_variable
- **Example:**

  p1Ptr->x

  will access the x coordinate member variable
- ***The -> operator is the dereference operator for the pointer struct variable***
- It is the same as **(*p1Ptr).x**

Access Operator for Pointers to Structures

- We now modify the previous program in order to include the following:

- Declare **p1** variable as before
- Declare **p1Alias** variable as before
- Declare **p1Ptr** as described here
- Read p1 with the pointer
- Print p1 with the reference
- Calculate the distance of p1 from origin with p1

Access Operator for Pointers to Structures

```
int main()
{
    //C++ program working with Points in two dimensional space
    //In this program, we will
    //  - Create one Point object,
    //  - Read its x and y coordinates from the user (keyboard)
    //  - Print the Point object to the screen, and finally
    //  - Print the distance of the Point from the origin

    // Step 1: Create the Point object
    Point p1;               //p1 variable
    Point &p1Alias = p1;    //An alias of p1
    Point *p1Ptr =  &p1;    //A pointer to p1

    // Step 2: Read the Point object from the user
    cout << "Enter the x coordinate of the Point: ";
    cin >> p1Ptr->x;
    cout << "Enter the y coordinate of the Point: ";
    cin >> p1Ptr->y;

    // Step 3: Print the Point object to the screen
    cout << "The Point object you created is P(" << p1Alias.x << ", " << p1Alias.y << ")" << endl;

    // Step 4: Print the distance of the Point object from the origin
    double distance = sqrt(p1.x*p1.x + p1.y*p1.y);
    cout << "The distance of the Point from the origin is " << distance << endl;

    system("Pause");
    return 0;
}
```

Creating C++ structs variables on the Heap Memory

- Similarly we can use pointers to create C++ structs on the heap using the new operator
- When the structs on the heap memory are no more needed; we should free their memory using the delete operator
- Analyze the following program and determine its output

```
int main()
{
    Point *p1;
    p1 = new Point;
    p1->x = 1.2;
    p1->y = -2.2;
    cout << "Point p1 is (" << p1->x << ", " << p1->y << ")" << endl;

    Point *p2 = new Point;
    cout << "Enter x-coordinate of p2 ";
    cin >> p2->x;
    cout << "Assigning random double value in [1.5, 3.2) to the y-coordinate of p2 ";
    p2->y = (1.0*rand() / RAND_MAX) * (3.2 - 1.5) + 1.5;
    cout << "Point p2 is (" << p2->x << ", " << p2->y << ")" << endl;

    delete p1;
    delete p2;

    system("Pause");
    return 0;
}
```

The Assignment Operator and struct variables

- Assignment Operator
  - Just like with simple data types, we can assign a struct type variable to another variable of the same struct type
  - The assignment will be performed in one to one correspondence of the member variables
- Thus given two Point variables p1 and p2 and assuming p1 has already been initialized, the statement

  p2 = p1;

  Will assign the value of the x member variable of p1 to the x member variable of p2; and the value of the y member variable of p1 to the y member variable of p2

The Assignment Operator and struct variables

- Analyze the following program and determine its output

```cpp
int main()
{
    Point p1, p2;
    p1.x = 1;
    p1.y = 2;

    p2 = p1;

    cout << "Point p1 is (" << p1.x << ", " << p1.y << ")" << endl;
    cout << "Point p2 is (" << p2.x << ", " << p2.y << ")" << endl;

    p2.x = 3;
    p2.y = 4;

    cout << "After modification, Point p2 is now (" << p2.x << ", " << p2.y << ")" << endl;
    cout << "Point p1 is still (" << p1.x << ", " << p1.y << ")" << endl;

    system("Pause");
    return 0;
}
```

Passing Structures to Functions

- We can pass structure variables to functions just like we did for any other data types
- Passing structures to functions can be done by value, by reference or by pointer
- Passing by value takes a copy of the argument and therefore any change to the structure variable in the function will not be reflected back
- Passing by reference and pointer results for any changes made in a function to be reflected back

Passing Structures to Functions

- In order to demonstrate the parameter passing for structure data types, we will modify the previous program as follows:
  - Read p1 in a function (pass by pointer)
  - Print p1 in a function (pass by value)
  - Calculate distance of p1 from the origin in a function (pass by reference)

Passing Structures to Functions

```cpp
#include <iostream>
using namespace std;

struct Point
{
    double x;
    double y;
};

void readPoint(Point *pPtr)
{
    cout << "Enter the x coordinate of the Point: ";
    cin >> pPtr->x;
    cout << "Enter the y coordinate of the Point: ";
    cin >> pPtr->y;
}

void printPoint(Point p)
{
    cout << "The Point object you created is P(" << p.x << ", " << p.y << ")" << endl;
}

double distanceFromOrigin(Point &pAlias)
{
    double d = sqrt(pAlias.x*pAlias.x + pAlias.y*pAlias.y);
    return d;
}
```

Passing Structures to Functions

```cpp
int main()
{
    //C++ program working with Points in two dimensional space
    //In this program, we will
    //  - Create one Point object,
    //  - Read its x and y coordinates from the user (keyboard)
    //  - Print the Point object to the screen, and finally
    //  - Print the distance of the Point from the origin

    // Step 1: Create the Point object
    Point p1;   //p1 variable

    // Step 2: Read the Point object from the user
    readPoint(&p1);

    // Step 3: Print the Point object to the screen
    printPoint(p1);

    // Step 4: Print the distance of the Point object from the origin
    double distance = distanceFromOrigin(p1);
    cout << "The distance of the Point from the origin is " << distance << endl;

    system("Pause");
    return 0;
}
```

Returning Structures from Functions

- We can also return structures from functions just like we do with any other data types
- We may return a struct variable, a reference to struct variable or a pointer to struct variable
- The following code demonstrates this by modifying the readPoint function so that it returns a struct

Returning Structures from Functions

```
#include <iostream>
#include <cmath>
#include <ctime>
using namespace std;

struct Point
{
    double x;
    double y;
};
Point getPoint()
{
    Point p;
    cout << "Enter the x coordinate of the Point: ";
    cin >> p.x;
    cout << "Enter the y coordinate of the Point: ";
    cin >> p.y;
    return p;
}
void printPoint(Point p)
{
    cout << "(" << p.x << ", " << p.y << ")" << endl;
}

double distanceFromOrigin(Point &pAlias)
{
    double d = sqrt(pAlias.x*pAlias.x + pAlias.y*pAlias.y);
    return d;
}
```

Returning a struct from a function

Returning Structures from Functions

```
int main()
{
    // C++ program working with Points in two dimensional space
    // In this program, we will
    //   - Create one point object,
    //   - Read its x and y coordinates from the user (keyboard),
    //   - Print the point object on to the screen, and finally
    //   - Compute and print the distance of the point object from the origin

    //Step 1. Create the point object
    Point p1;
    //Step 2. Read its x and y coordinates from the user (keyboard)
    p1 = getPoint();
    //Step 3. Print the point object on to the screen
    cout << "The Point object you created is: ";
    printPoint(p1);
    //Step 4. Compute and print the distance of the point object from the origin
    double distance = distanceFromOrigin(p1);
    cout << "The distance of the Point object from the origin is " << distance << endl;

    system("Pause");
    return 0;
}
```

Assigning a point object a value returned from a function

Array of Structures

- Just like any other data type, we can create single or multi dimensional arrays of structures
- The declaration of array of structures follows exactly the same way just like for ints, floats,...
- ***Syntax* (array of structs)**

StructDataType arrayName[constantValue] //static array

StructDataType *arrayName = new StructDataType[arraySize] //dynamic array

- **Example**

```
Point  p[10];  //static array
                OR
int size;
cout << "Enter array size: ";
cin >> size;
Point *p= new Point[size];    //dynamic array
```

Array of Structures

- Let us demonstrate array of structures by extending the previous program to create **n** points where **n** is entered from the user
- The aim is to declare the array of structures and then use loops to read the point objects, print the points and compute and print their distances from the origin
- In addition, we will compute the two farthest apart points among all the points and print these furthest apart points

Array of Structures

```
#include <iostream>
#include <cmath>
#include <ctime>
using namespace std;

struct Point
{
    double x;
    double y;
};
void printPoint(Point p)
{
    cout << "(" << p.x << ", " << p.y << ")" << endl;
}

double distanceFromOrigin(Point &pAlias)
{
    double d = sqrt(pAlias.x*pAlias.x + pAlias.y*pAlias.y);
    return d;
}

double distanceBetweenPoints(Point &p1, Point &p2)
{
    double x_diff = p1.x - p2.x;
    double y_diff = p1.y - p2.y;
    double d = sqrt(x_diff*x_diff + y_diff*y_diff);
    return d;
}
```

Array of Structures

```
int main()
{
    // C++ program working with Points in two dimensional space
    // In this program, we will
    //   - Create an array of n Point objects where n is a user input value,
    //   - Assign the x and y coordinates of each element of the array a random number in the range (-2.5, 3.2),
    //   - Print each element of the array on to the screen,
    //   - Print the distance of each element of the array from the origin, and finally
    //   - Compute and print the two farthest apart Points among all the elements of the array
    srand(time(0)); //Seed the random number generator
    // Step 1: Read the size of the array
    int size;
    cout << "Enter the size of the array: ";
    cin >> size;

    // Step 2: Create a dynamic array of the desired size
    Point *arr = new Point[size];   //dynamic array of struct variable

    // Step 3: Assign the x and y coordinates of each element of the array a random number in the range (-2.5, 3.2)
    for (int i = 0; i < size; ++i)
    {
        cout << "Assigning the x and y coordinates of the element at index " << i << endl;
        arr[i].x = ((1.0*rand())/RAND_MAX) * (3.2 - -2.5) + -2.5;
        arr[i].y = ((1.0*rand())/RAND_MAX) * (3.2 - -2.5) + -2.5;
    }
    cout << endl;

    // Step 4: Print the Point objects on to the screen
    cout << "The Point objects you created are:" << endl;
    for (int i = 0; i < size; ++i)
    {
        cout << "\tThe element at index " << i << " is P"; printPoint(arr[i]);
    }
    cout << endl;
```

Array of Structures

```
    // Step 5: Print the distances of the Point objects from the origin
    cout << endl;
    for (int i = 0; i < size; ++i)
    {
        double distance = distanceFromOrigin(arr[i]);
        cout << "The distance of the element at index " << i << " from the origin is " << distance << endl;
    }
    cout << endl;

    // Step 6: Compute and print the two farthest apart Point objects among all the elements of the array
    int index1 = 0, index2 = 0; //Initialize the furthest apart point indexes
    double max_distance = 0;
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            double distance = distanceBetweenPoints(arr[i], arr[j]);
            cout << "The distance between the elements at indexes " << i << " and " << j << " is " << distance << endl;
            if (distance > max_distance)
            {
                index1 = i;
                index2 = j;
                max_distance = distance;
            }
        }
    }
    cout << endl;
    cout << "The furthest apart points are element at index " << index1 << " and element at index " << index2 << endl;
    cout << "The distance between the furthest apart points is " << max_distance << endl << endl;
    system("Pause");
    return 0;
}
```

Exercise: Modify the code in Step 6 to remove unnecessary computations.

Structures as Member Variables

- The members of a **struct** can be any defined data type including other structures
- For example once a **struct** is declared, it can perfectly be a member of another **struct**
- To demonstrate this consider a **Line** object in two dimensional space
- A **Line** is described by two of its end Points
- Therefore can be a struct with Point members

Structures as Member Variables

- This can be declared as

```
struct Point
{
        double x, y;
};
struct Line
{
        Point start, end;
};
```

The Point struct must first be declared before using it as a data type in the Line struct

Accessing Member Variables of Structure Member Variables

- Suppose we declare a Line variable as follows:

  Line line1;

- Then in order to access the x and y coordinates of the start and end points of the line1 variable, we proceed as follows:
  - **line1.start.x** will access the x coordinate of the start point
  - **line1.start.y** will access the y coordinate of the start point
  - Similarly for the end point

Let’s Play with Line Objects

- In order to demonstrate your understanding of the struct as member variable of another struct, do the following practice
  - ➢ Declare a Line struct
  - ➢ Create an array of user desired size of Line objects
  - ➢ Read each Line object from the user
  - ➢ Print each Line object (you decide format)
  - ➢ Print each of the parallel lines pair in the array

Concluding Remarks

- Working with struct variables is the same as working with any other simple data type variables
- All you need is to remember to access member variables using the dot or -> operators
- If a struct has another struct as member variable then use the dot or -> operators to access the member variables recursively
- The process of HIDDING member variables of a structure inside the structure is known as **DATA ENCAPSULATION!!!**